A Step-by-step Guide + a Multi Site Web Builder

Site Builder Kit

Configure Your First Server & Create Your Own Online Web Building Service.

- Accept user signups
- Allow users to create their own websites
- Allow users to use custom domains
- Automatically secure your users' website with https
- Enable automatic publishing (from Github to live site)

Site Builder Kit is a project that you can use to create your own web building service. It contains a guide (this document) and a ready to deploy application.

This guide uses step-by-step instructions. Just follow the steps until you have your server live & ready. A quick explanation will be provided during the process and it will be clearer at the end once you have performed all the steps and see how they work immediately.

Prerequisites:
HTML and basic Javascript knowledge.
Basic database knowledge.

Contents

Part 1 – The Server

Part 2 – The Application

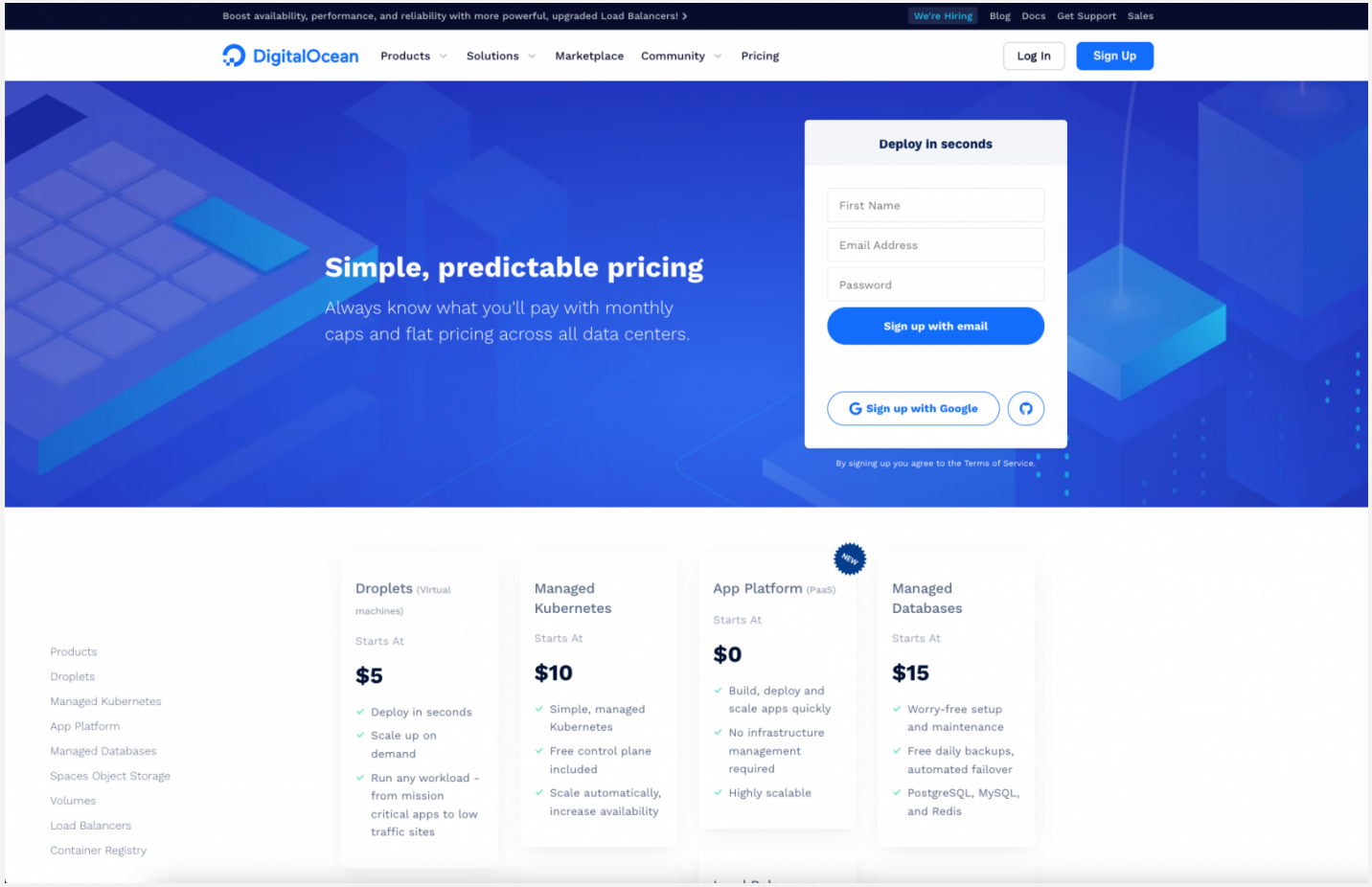
PART 1

The Server

1. Getting a Server

Find a cloud provider that provides a VPS (Virtual Private Server) and signup for an account. A VPS is essentially a virtual machine that is connected to the internet. You can try, for example:

- **https://digitalocean.com** (a VPS is called a droplet here)
- **https://aws.amazon.com/ec2**
- **https://upcloud.com**



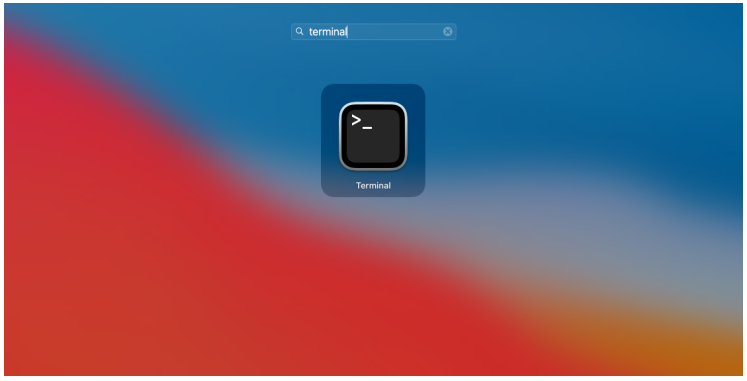
We highly recommed **digitalocean.com** for its ease of use and simple pricing. You can get a server from as low as $5 per month.

After signup you will have the option to choose an operating system. We will use **Ubuntu** (a Linux-based operating system) for our project, so choose the option for **Ubuntu**.

Once your VPS is ready, you will have the information such as server IP and a **root** user password. **Root** user is a special user account that has the highest access rights to your server. That's all that's needed to get started.

2. Connecting to Your Server

To connect to your server, you will need **Terminal** (on Mac) or **Command Prompt** (on Windows). We will use **Terminal** throughout this guide.



Search for 'terminal' on Mac.

Open the Terminal and type SSH command:

```
$ ssh root@<server-ip>
```

Example: ssh root@123.123.123.123

Then enter your password when prompted. You will be connected to your server as a root user and are now ready to manage your server.

SSH stands for **secure shell**. It provides secure connection between your computer and your server.

3. Adding a User

To protect your server, it is good practice to not use the root user on an everyday basis.

1. Now add a user and give it a name, for example: **hobnob**. Specify the password when prompted.

```
$ adduser hobnob
```

The home user directory will be created at: **/home/hobnob**

2. To allow the user to execute commands requiring root privileges, you need to add the user to a group named **sudo**.

```
$ usermod -aG sudo hobnob
```

The **-a** flag appends the user to the group without removing from other group.

3. Then close your terminal and relogin using:

```
$ ssh hobnob@<server-ip>
```

To close your terminal, type **exit** and press Enter/Return.

4. More Convenient Authentication

So far we've used password-based authentication to connect to our server. Now we'll try another authentication method called public key authentication which provides more convenient access to your server.

1. Disconnect from your server and run this command:

```
$ ssh-keygen -t rsa -b 4096 -C <your-email-address>
```

You will be prompted to enter file name and passphrase. Press Enter to accept the default file name and empty passphrase.

This will create a public/private SSH key pair saved in ~**/.ssh/id_rsa** directory.

2. Run this command to copy the generated key to your remote server you want to manage:

```
$ ssh-copy-id root@<server-ip>
```

This is for root user login.
You will be prompted to enter your root user password.

```
$ ssh-copy-id hobnob@<server-ip>
```

This is for hobnob user login.
You will be prompted to enter hobnob user password.

If you're using Windows and ssh-copy-id is not available, you can use this command:

```
$ type C:\Users\<USER>\.ssh\id_rsa.pub | ssh root@ "cat >> .ssh/authorized_keys"
```

Please replace the **C:\Users\<USER>** with your default user folder and **<server-ip>** with your VPS IP address.

3. Now try connecting to your server using SSH command:

```
$ ssh hobnob@<server-ip>
```

This time you will be logged in immediately without being prompted to enter password.

4. Optionally, you can now disable SSH password authentication. This will give an extra protection to your server. Open the SSH configuration file:

```
$ sudo vi /etc/ssh/sshd_config
```

Type **i** (insert) to start editing and search for an entry called **PasswordAuthentication**. Set it to **no**.

```
PasswordAuthentication no
```

Press **ESC** and then **:wq** to write (save) and quit.

5. Restart SSH service by typing:

```
$ sudo systemctl reload ssh.service
```

Now password-based authentication is disabled.

Note:
You can use the same SSH key (generated in step 1) to manage multiple remote servers.
To check if you already have an SSH key, use this command:
ls -l ~/.ssh/id_*.pub

5. Setting-up a Firewall

With firewall, you can secure your server from unauthorized access by controlling the allowed network traffic to your server. We will use a firewall application named **ufw** (short for uncomplicated firewall)

1. Install ufw.

```
$ sudo apt install ufw
```

2. ufw is inactive by default. To check:

```
$ sudo ufw status
Status: inactive
```

3. View a list of registered applications:

```
$ sudo ufw app list
Available applications:
  OpenSSH
```

You will see that an application named **OpenSSH** is listed. Before activating the firewall, we'll configure the firewall to allow SSH access by allowing **OpenSSH**.

4. Allow SSH access by specifying the application name **OpenSSH**.

```
$ sudo ufw allow OpenSSH
```

5. Now you can enable the firewall:

```
$ sudo ufw enable
```

6. Check the status again:

```
$ sudo ufw status
Status: active

To                         Action      From
--                         ------      ----
OpenSSH                    ALLOW       Anywhere
OpenSSH (v6)               ALLOW       Anywhere (v6)
```

The firewall is now active with rules that allow SSH.

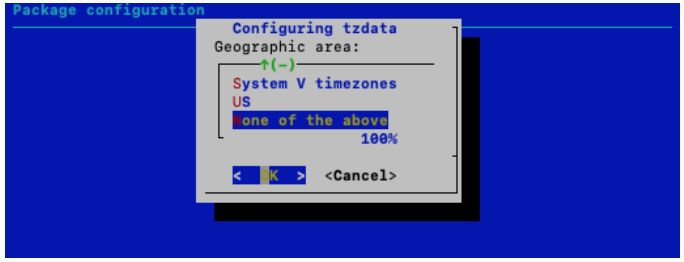
6. Configuring Time Zone

We will configure our server time to use Coordinated Universal Time or UTC time zone.

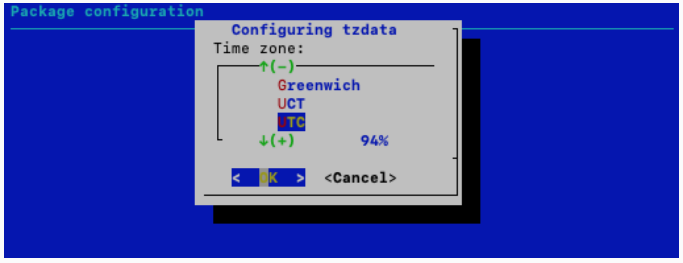
1. Run:

```
$ sudo dpkg-reconfigure tzdata
```

You will be presented with the following screen:



2. When prompted to choose the Geographic area, select **None of the above**.



3. When prompted to choose the time zone, select **UTC**.

Select OK to finish.

Congratulation! Now you have successfully setup a secure VPS. Next we will start creating a simple website.

7. Creating a Simple Website using Node.js

1. To handle Node.js versions and installation, we'll use **nvm** (Node Version Manager). First, install the nvm.

```
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.39.0/install.sh | bash
```

2. Then close your terminal and relogin.

3. Install Node.js using nvm.

```
$ nvm install node
```

To check the Node.js version:

```
$ node -v
v17.3.1
```

3. Create a new directory named **apps**:

```
$ mkdir apps
```

Go to the new directory:

```
$ cd apps
```

4. Create another directory named **mywebsite**:

```
$ mkdir mywebsite
```

Go to the new directory:

```
$ cd mywebsite
```

As you login using the user **hobnob** and work from your home directory, your newly created directory will be located at:

/home/hobnob/apps/mywebsite/

In this directory, we will create our first website.

5. We'll start creating our project by running the init command.

```
$ npm init -y
```

This will generate a **package.json** file which contains information about your project and its dependencies.

6. Install the Express framework (https://expressjs.com):

```
$ npm i express
```

Express is a web framework for Node.js.

7. Create a new file named **server.js**.

```
$ touch server.js
```

8. Edit the file using the vi command:

```
$ vi server.js
```

Type **i** (insert) to start editing and paste the following code:

```
const express = require('express')
const app = express()
const port = 8080

app.get('/', (req, res) => res.send('Hello World!'))
app.listen(port, () => console.log(`Listening on port ${port}!`))
```

Here we use the Express framework and listen for / or http://localhost:8080/.
Our first website will run on port 8080, displaying a 'Hello World!' message.

Press **ESC** and then **:wq** to write (save) and quit.

To run the code:

```
$ node server.js
```

To stop, just press **CTRL-C**.

For now, your website is inaccessible from the outside. We will fix this shortly and later we will assign a domain name and make it secure with SSL.

8. Making Your Website Publicly Accessible

1. Install NGINX

```
$ sudo apt update
```

```
$ sudo apt install nginx
```

2. Reconfigure Firewall

First, check the application list:

```
$ sudo ufw app list
Available applications:
Apache
Apache Full
Apache Secure
Bind9
Nginx Full
Nginx HTTP
Nginx HTTPS
OpenSSH
```

3. Allow **Nginx Full** (this is to allow both http and https):

```
$ sudo ufw allow 'Nginx Full'
```

4. Edit configuration

```
$ cd /etc/nginx/sites-available/
```

```
$ sudo vi default
```

Update with the following code:

```
server {
        listen 80 default_server;
        listen [::]:80 default_server;
        location / {
                proxy_pass http://localhost:8080;
        }
}
```

Here we configure NGINX as a reverse proxy where we redirect **http://<server-ip>/** to **http://localhost:8080** (our Node.js website)

5. To check the status:

```
$ sudo systemctl status nginx
```

6. Start NGINX

First you need to stop Apache if it is running on your UBUNTU server:

```
$ sudo systemctl stop apache2
```

```
$ sudo systemctl disable apache2
```

```
$ sudo apt remove apache2
```

```
$ sudo apt autoremove
```

Then start NGINX:

```
$ sudo service nginx restart
```

7. Run your website

Go to your website directory:

```
$ cd /home/hobnob/apps/mywebsite/
```

Run your Node.js code:

```
$ node server.js
```

Congratulations! Now your website is publicly accessible. Open your browser and enter:

http://<server-ip>/

9. Keeping Your Website Alive

Now we will make our website run automatically on server start (to keep it alive). For this, we will use PM2 to run our Node.js code. PM2 is a Production Process Manager for Node.js applications.

1. Go to your website directory

```
$ cd /home/hobnob/apps/mywebsite/
```

2. Install PM2:

```
$ npm install -g pm2
```

-g will perform global installation.

3. Now run your Node.js code using PM2:

```
$ pm2 start server.js
```

Your website is now started.

To keep PM2 alive (runs on server startup):

```
$ pm2 startup
[PM2] Init System found: systemd
[PM2] To setup the Startup Script, copy/paste the following command:
sudo env PATH=....
```

Copy the command (see highlighted) and run it.

Then run:

```
$ pm2 save
```

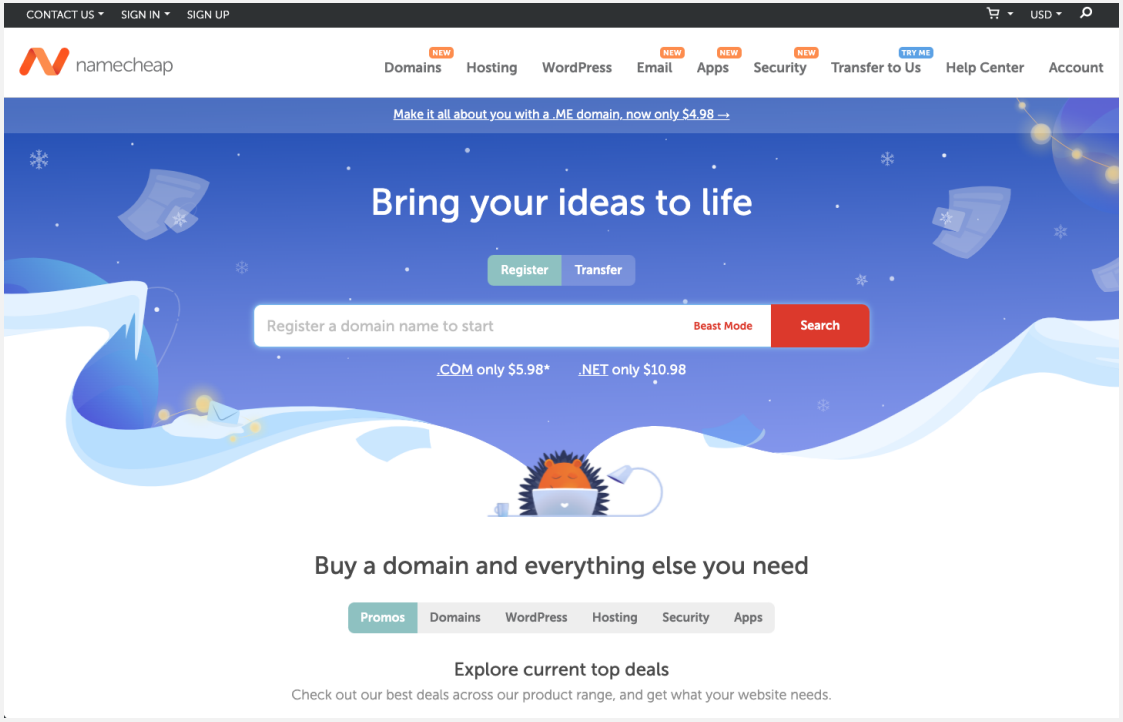
To check the list of running applications:

```
$ pm2 list
```

10. Adding a Domain Name

1. Register a domain

You can register a domain name at a domain registrar, for example at: **namecheap.com**.

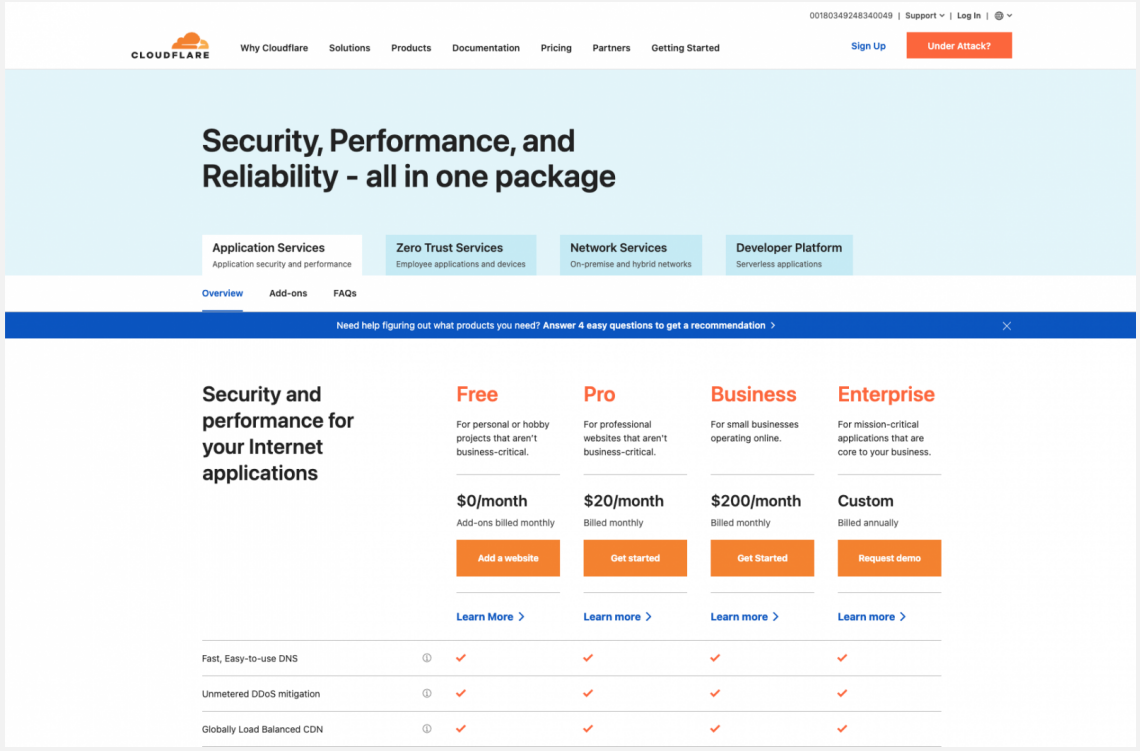


As an example in this guide, we registered a domain named: **sitemagz.com**

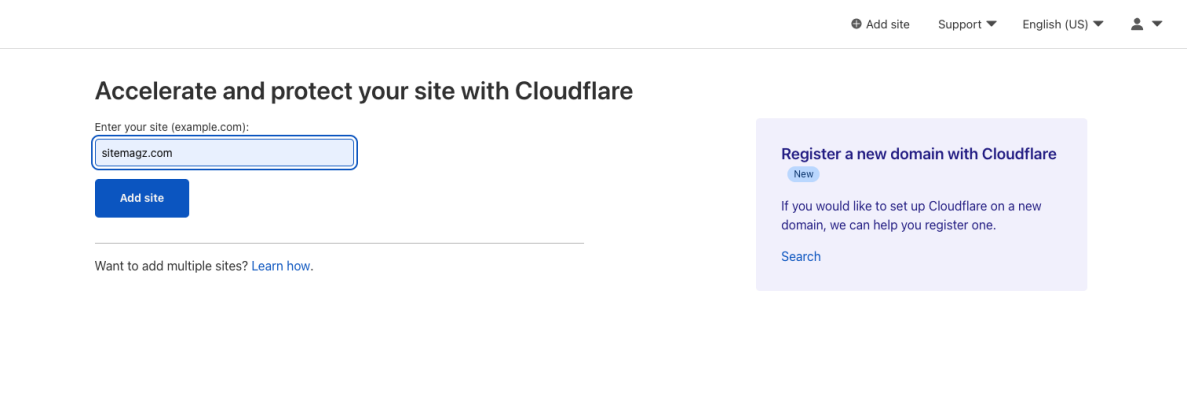
2. Get a DNS service

We then need a DNS (Domain Name System) service to associate our registered domain name with our server (VPS). We recommend a DNS service provided by Cloudflare (https://www.cloudflare.com).

1. Signup at Cloudflare.



2. Login to your Cloudflare panel and add your domain name.

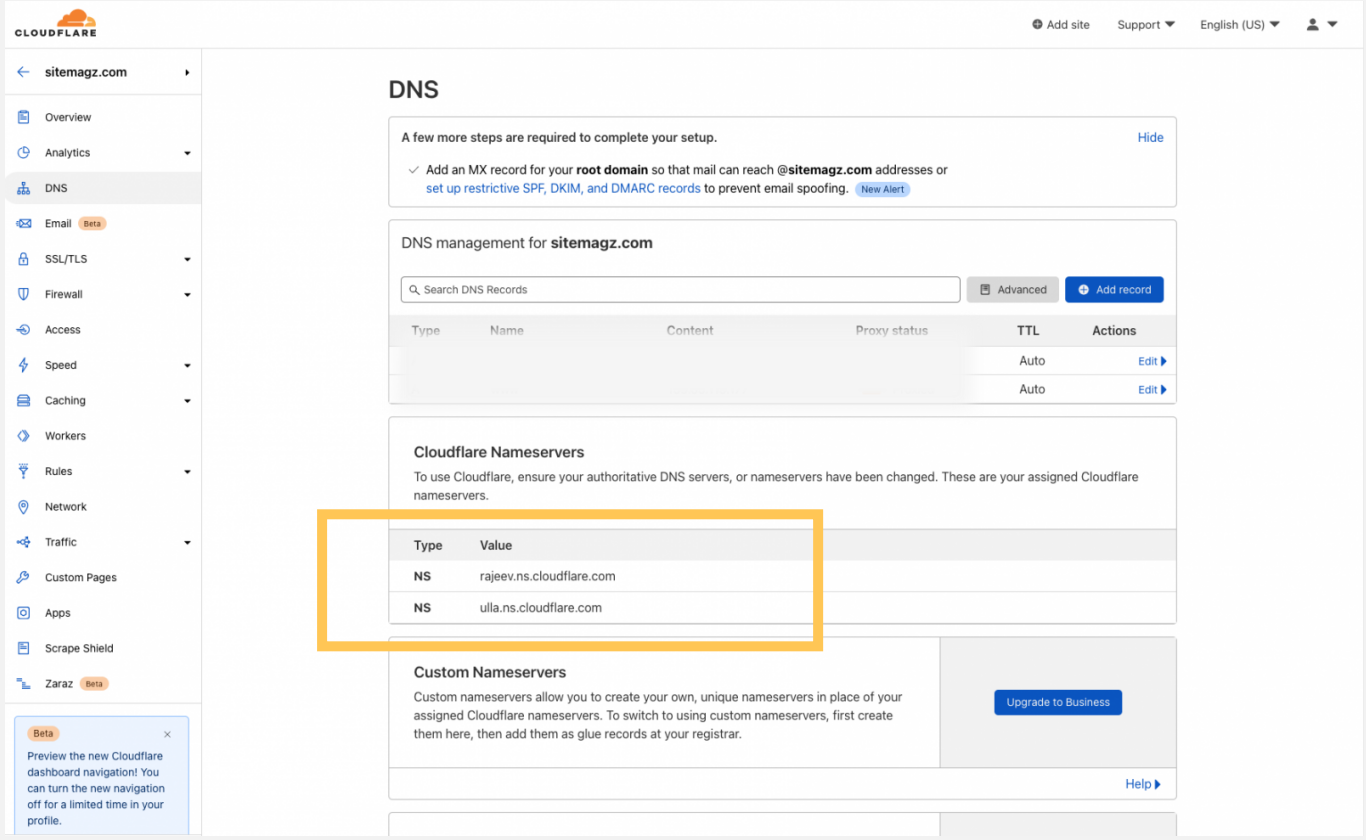


3. After your domain is added, open your domain settings and find the nameservers information.

As seen on the screenshot, we have these nameservers info:

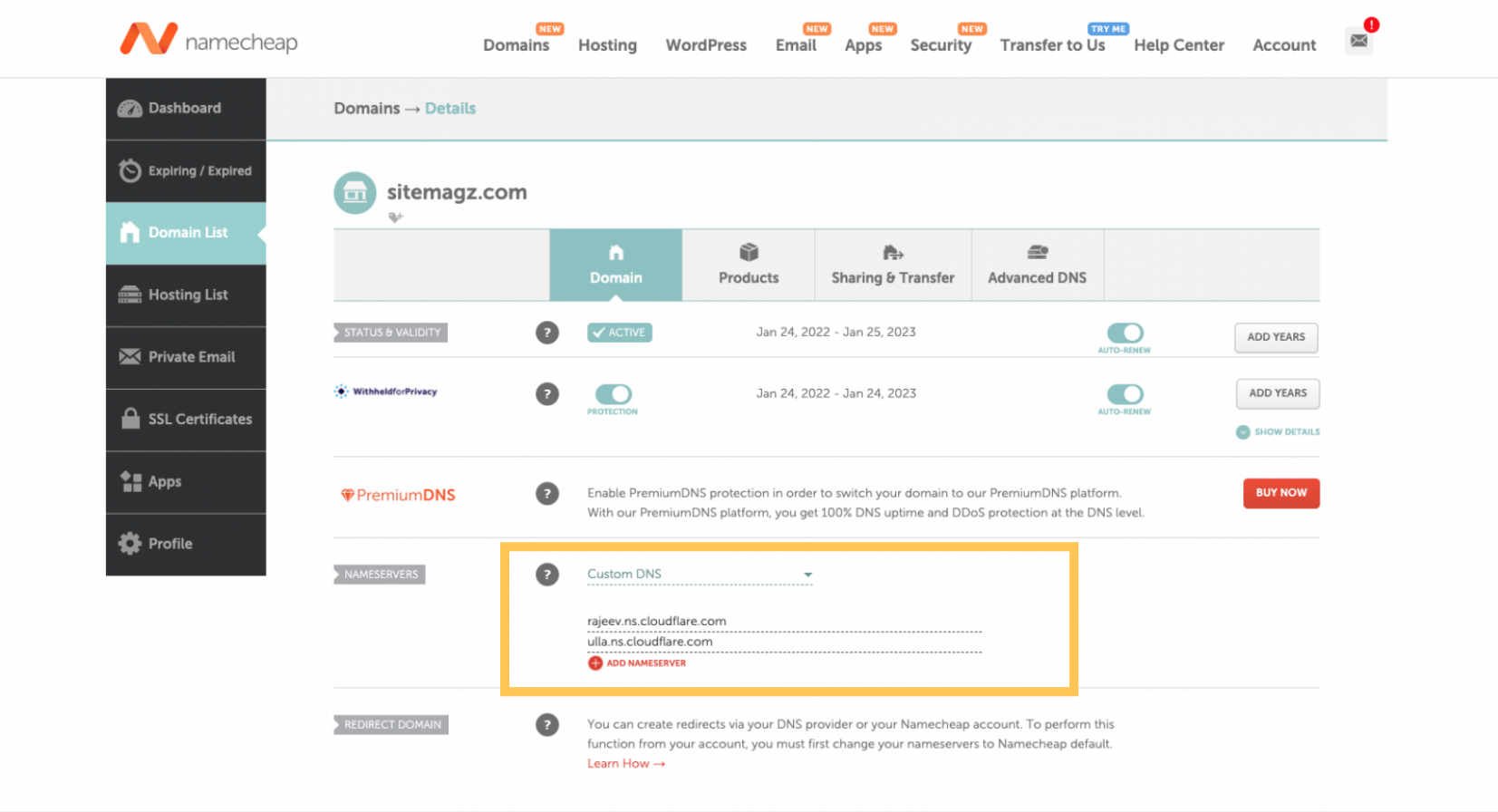
rajeev.ns.cloudflare.com

ulla.ns.cloudflare.com



3. Use the nameservers

Now login to your domain registrar (eg. namecheap.com) and enter your nameservers information.

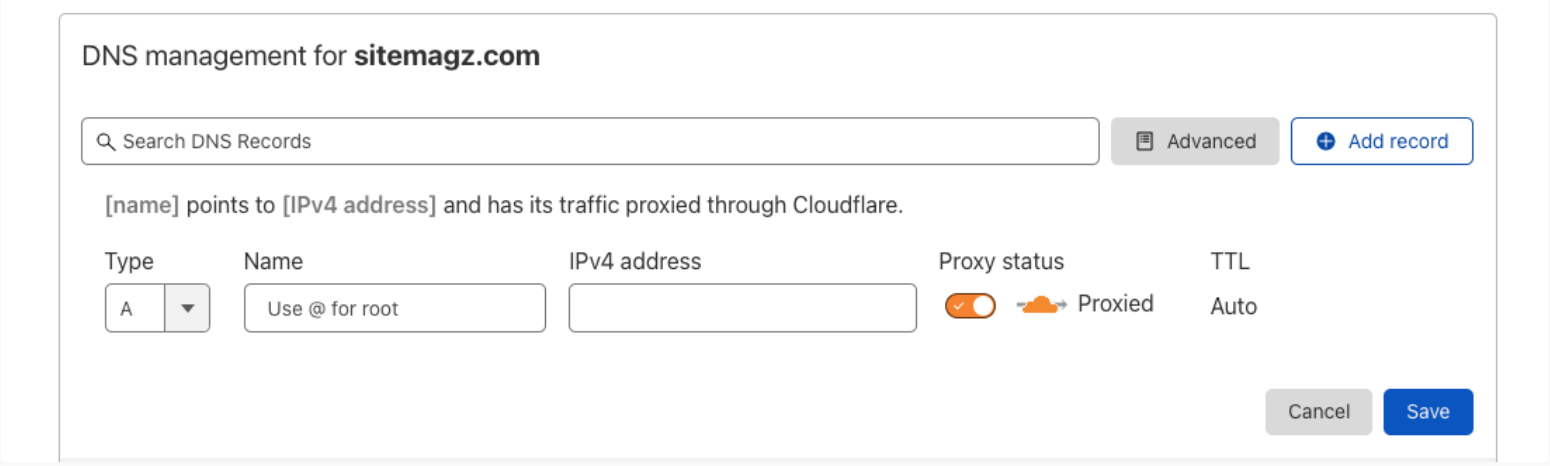


4. Configure your domain

Login back to your DNS service (eg. cloudflare.com) and add the following records and click Save:

Type: A
Name: <domain-name> (sitemagz.com or just type @)
IPv4 address: <your-server-ip>

Type: A
Name: www
IPv4 address: <your-server-ip>



5. Update NGINX configuration

Now connect to your VPS and reconfigure NGINX:

```
$ cd /etc/nginx/sites-available/
```

```
$ sudo vi default
```

Update the file with the following:

```
server {
    listen 80 default_server;
    listen [::]:80 default_server;
    server_name .sitemagz.com;
    location / {
            proxy_pass http://localhost:8080;
    }
}
```

Here you can now specify your domain name.

Then restart the service:

```
$ sudo service nginx restart
```

Congratulations! Now you can open your website using the new domain name. From your browser, open:

http://sitemagz.com

But it’s not secure yet. We’ll use https shortly.

11. Accepting Multiple Custom Domains and Automatically Make Your Websites Secure with HTTPS

With the ability to accept multiple custom domains, you can create an application that allows users to signup and create their own website. They can specify their domain name and your server will do the rest to make their websites accessible and secure.

To make a website secure with https, the website domain needs an SSL certificate (also known as TLS certificate). An SSL certificate is a bit of code on your web server that provides the security for transferring data or information. You can request a free SSL certificate from Let's Encrypt (https://letsencrypt.org). Since we will host multiple domains on our VPS, we will make the request certificate process automatic. The request will be made on the very first time the domain URL is accessed. To implement this, we will use OpenResty. OpenResty is a web server that extends NGINX.

1. First, disable NGINX since we will use OpenResty

```
$ sudo systemctl disable nginx
```

```
$ sudo systemctl stop nginx
```

2. Install OpenResty.
Follow these steps to get the latest version and install it.

```
$ sudo apt-get -y install wget gnupg ca-certificates apt-utils curl
```

```
$ sudo wget -O - https://openresty.org/package/pubkey.gpg | sudo apt-key add -
```

```
$ echo "deb http://openresty.org/package/ubuntu $(lsb_release -sc) main" \   | sudo tee /etc/apt/sources.list.d/openresty.list
```

```
$ sudo apt-get update
```

```
$ sudo apt-get -y install openresty
```

3. Install Lua package manager. **Lua** is a programming language and **luarocks** is the package manager for Lua.

```
$ sudo apt-get -y install luarocks
```

4. Now with **luarocks** we can install a **lua-resty-auto-ssl** plugin.

```
$ sudo luarocks install lua-resty-auto-ssl
```

If there is an error with message:
sh: 1: make: not found
Please run:
sudo apt-get install build-essential

5. Create 2 directories:

```
$ sudo mkdir /var/log/openresty
```

```
$ sudo mkdir /etc/resty-auto-ssl
```

OpenResty will create new SSL certificates in this directory.

Change the directory ownership to the user **www-data**. We will use this user to run OpenResty and write new SSL certificate in this directory. Note that the **www-data** user and group are provided by the server.

```
$ sudo chown www-data:www-data /etc/resty-auto-ssl
```

6. Now create a self-signed certificate that we will use as a fallback certificate in case certificate request for a domain fails. A self-signed certificate is a security certificate that is not signed by a certificate authority (CA).

```
$ sudo openssl req -new -newkey rsa:2048 -days 3650 -nodes -x509 -subj '/CN=sni-support-required-for-valid-ssl'
-keyout /etc/ssl/resty-auto-ssl-fallback.key -out /etc/ssl/resty-auto-ssl-fallback.crt
```

7. Setup OpenResty config file

```
$ cd /etc/openresty
```

Remove the current config file:

```
$ sudo rm nginx.conf
```

Create a new config:

```
$ sudo vi nginx.conf
```

Pres **i** and insert the following configuration:

```
user www-data;

events {
  worker_connections 1024;
}

http {
  lua_shared_dict auto_ssl 1m;
  lua_shared_dict auto_ssl_settings 64k;
  resolver 8.8.8.8 ipv6=off;

  init_by_lua_block {
    auto_ssl = (require "resty.auto-ssl").new()
    auto_ssl:set("allow_domain", function(domain)
      return true
    end)
    auto_ssl:set("ca", "https://acme-staging-v02.api.letsencrypt.org/directory")
    auto_ssl:init()
  }

  init_worker_by_lua_block {
    auto_ssl:init_worker()
  }

  server {
    listen 443 ssl;
    ssl_certificate_by_lua_block {
      auto_ssl:ssl_certificate()
    }

    ssl_certificate /etc/ssl/resty-auto-ssl-fallback.crt;
    ssl_certificate_key /etc/ssl/resty-auto-ssl-fallback.key;
  }

  server {
    listen 80;
    location /.well-known/acme-challenge/ {
      content_by_lua_block {
        auto_ssl:challenge_server()
      }
    }
  }

  server {
    listen 127.0.0.1:8999;
    client_body_buffer_size 128k;
    client_max_body_size 128k;

    location / {
      content_by_lua_block {
        auto_ssl:hook_server()
      }
    }
  }
}
```

Press **ESC** and then **:wq** to write (save) and quit.

8. Start OpenResty

```
$ sudo systemctl restart openresty
```

9. Now you can test from your browser and open your website domain URL with https:

https://sitemagz.com

At first, the green lock icon in the browser address bar may not be visible and the site is still not secure. But as long as the OpenResty welcome page is opened, it works as expected and you can continue to the next steps.

10. Update your OpenResty config file:

```
$ sudo vi /etc/openresty/nginx.conf
```

Remove this line:

```
auto_ssl:set("ca", "https://acme-staging-v02.api.letsencrypt.org/directory")
```

Add this block inside **server { listen 443 ssl; ... }** (before the curly bracket ends):

```
location / {
    proxy_pass http://localhost:8080;
    proxy_set_header Host $host;
}
```

This will redirect the request to your Node.js website.

And add this block inside **server { listen 80; ... }** (before the curly bracket ends):

```
location / {
  return 301 https://$host$request_uri;
}
```

This will force http to https.

Here is the update file:

```
user www-data;

events {
  worker_connections 1024;
}

http {
  lua_shared_dict auto_ssl 1m;
  lua_shared_dict auto_ssl_settings 64k;
  resolver 8.8.8.8 ipv6=off;

  init_by_lua_block {
    auto_ssl = (require "resty.auto-ssl").new()
    auto_ssl:set("allow_domain", function(domain)
      return true
    end)
    auto_ssl:init()
  }

  init_worker_by_lua_block {
    auto_ssl:init_worker()
  }

  server {
    listen 443 ssl;
    ssl_certificate_by_lua_block {
      auto_ssl:ssl_certificate()
    }

    ssl_certificate /etc/ssl/resty-auto-ssl-fallback.crt;
    ssl_certificate_key /etc/ssl/resty-auto-ssl-fallback.key;

    location / {
        proxy_pass http://localhost:8080;
        proxy_set_header Host $host;
    }
  }

  server {
    listen 80;
    location /.well-known/acme-challenge/ {
      content_by_lua_block {
        auto_ssl:challenge_server()
      }
    }
    location / {
      return 301 https://$host$request_uri;
    }
  }

  server {
    listen 127.0.0.1:8999;
    client_body_buffer_size 128k;
    client_max_body_size 128k;

    location / {
      content_by_lua_block {
        auto_ssl:hook_server()
      }
    }
  }
}
```

11. Recreate our **resty-auto-ssl** folder.

```
$ sudo rm -r /etc/resty-auto-ssl
```

```
$ sudo mkdir /etc/resty-auto-ssl
```

```
$ sudo chown www-data:www-data /etc/resty-auto-ssl
```

12. Restart OpenResty

```
$ sudo systemctl restart openresty
```

13. Open your browser and test again.

https://sitemagz.com

Now your server is able to accept multiple custom domains and all websites will be secured automatically with https!
You (or your users) can add a custom domain by adding an A-record that points to your VPS IP. A new certificate will be created on the very first request when opening the domain URL.

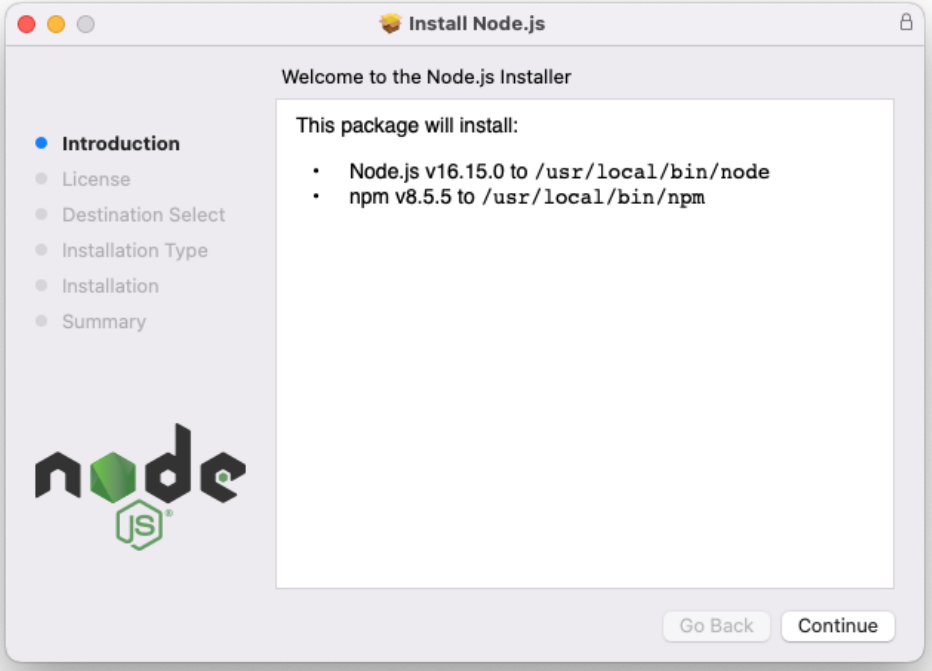
12. Preparing Your Local Computer for Development

In chapter 7, we install and use Node.js to create a website on our server. We also need to have Node.js installed on our local computer for local development.

One of the most common ways to install Node.js is through the official installer. You can download the installer from:

https://nodejs.org/en/download/

Then you can start the installation.



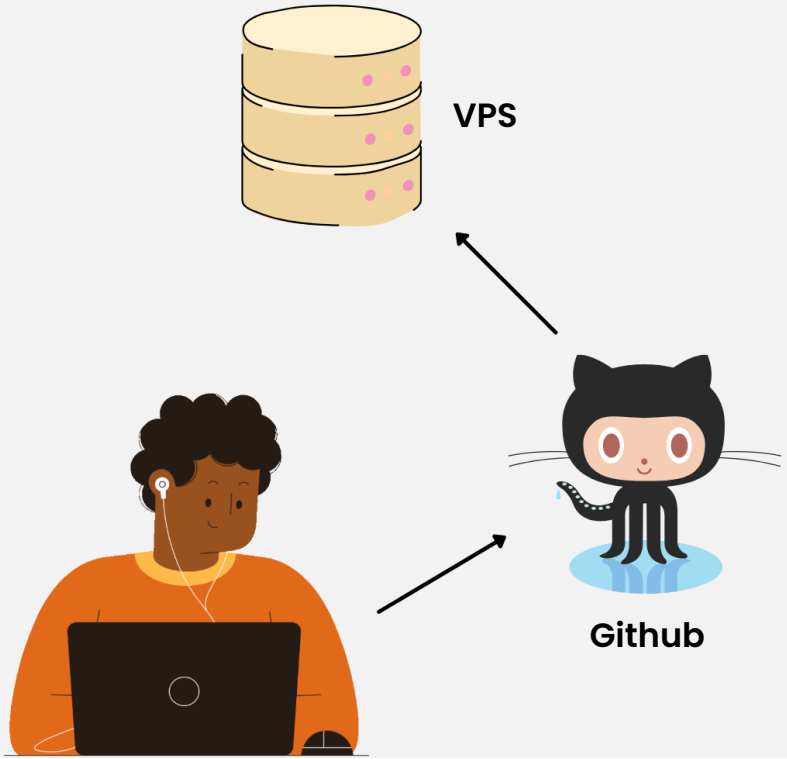
That's all that's needed to start the development. In the next chapter we'll start creating a website on our local computer and then publish the project code on Github.

13. Automating Your Website Publishing

We will setup a project on our local computer and publish the project code on Github. Then we will configure Github to connect to our VPS for automatic deployment and publishing. With this, everytime you push an update from your local computer, your live production website will also get updated.

Steps:

1. Create a New Web Project on Your Local Computer
2. Host Your Web Project on Github
3. Clone Your Github Project to Your VPS
4. Setting Up Automatic Deployment



1. Create a New Web Project on Your Local Computer

1. Open terminal and run this command. We'll create a project named **mywebapp**.

```
$ npx create-next-app mywebapp
```

This will create your website using Next.js framework (https://nextjs.org). Don't worry if you haven't used it before. Just follow the steps to quickly get started and see what it's like.

2. Go to your project folder and run the project:

```
$ cd mywebapp
```

```
$ npm run dev
```

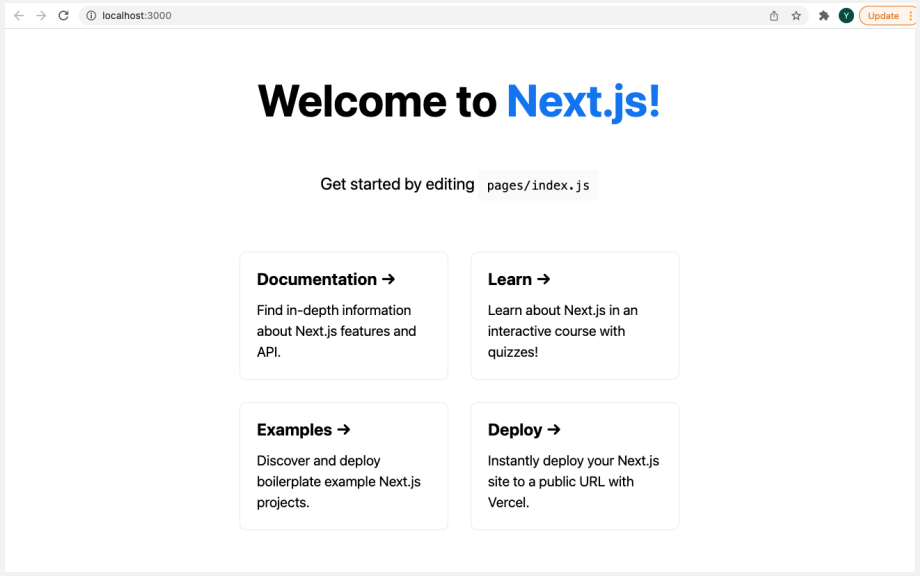
3. Now open your browser and go to:

http://localhost:3000

Congratulations! Your website has been created.

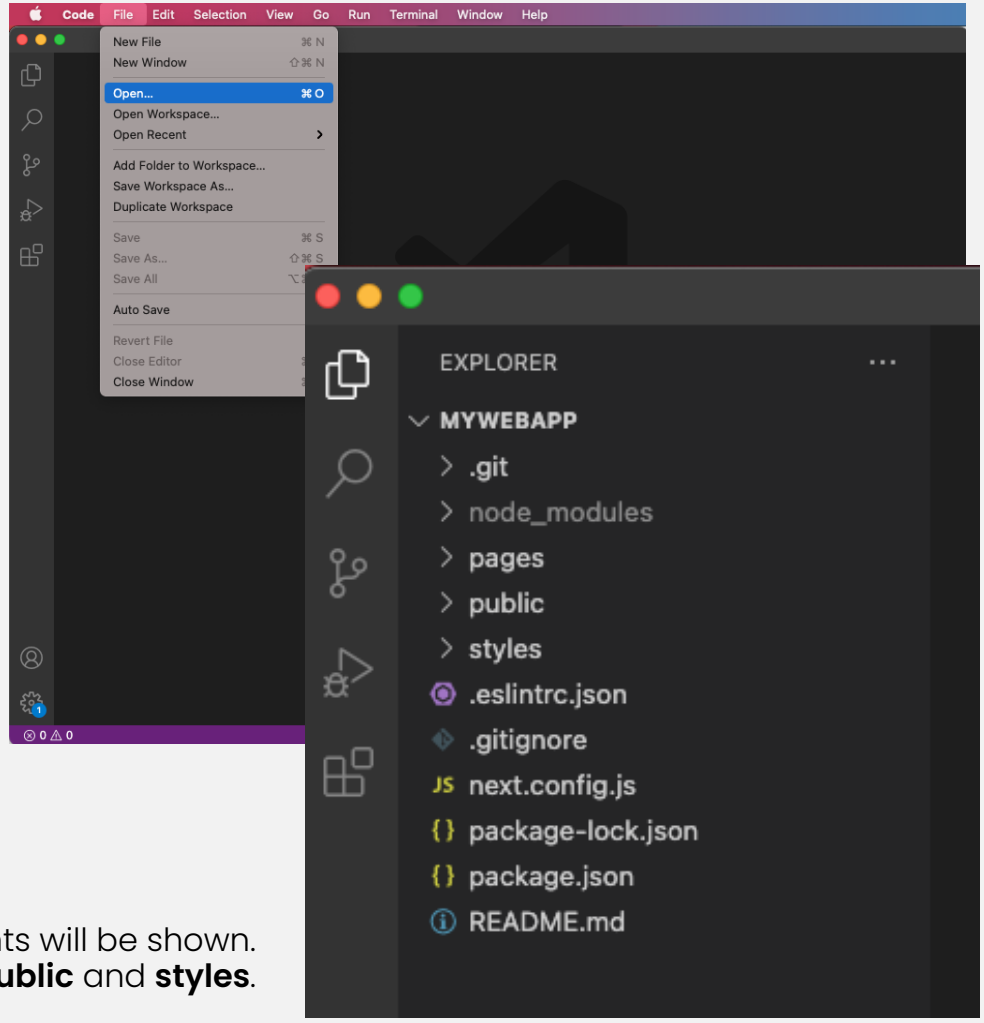
What's so great about Next.js? Next.js is one-of-a-kind framework that allows you to work with both frontend and backend parts of your application. So it is a full-stack framework. It is built on top of the popular React and Node.js, and with its server side rendering feature, your website will be great for SEO. Next.js framework is used by leading companies, including Netflix, Github, Hulu, and AT&T.

This is the default home page of your website:



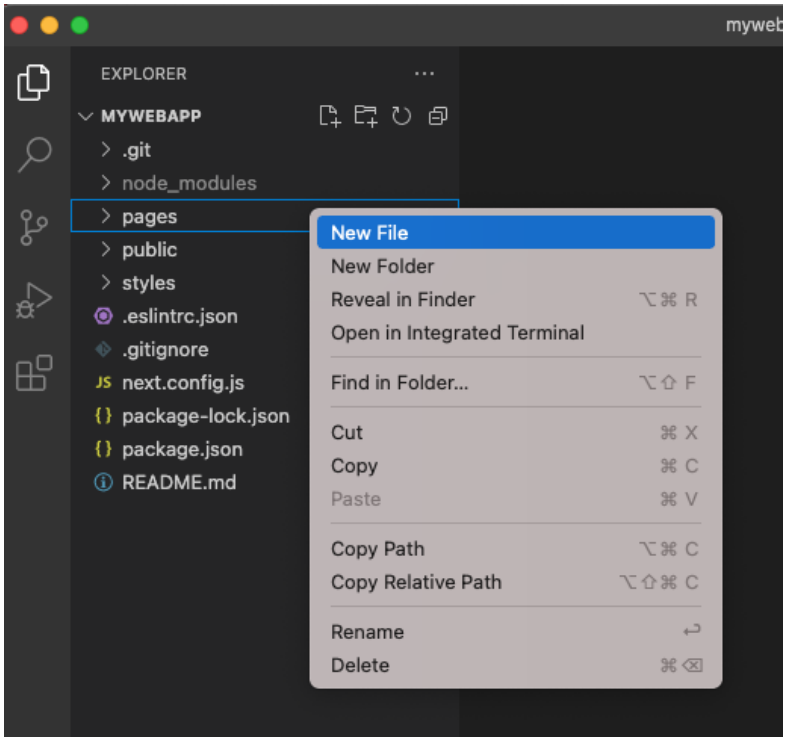
Now, let's try creating a new page. We will use Visual Studio Code as our code editor. You can download Visual Studio Code from https://code.visualstudio.com/download.

4. Open Visual Studio Code. From the **File** menu, select **Open**. Then browse your local computer and select your project folder named **mywebapp**.



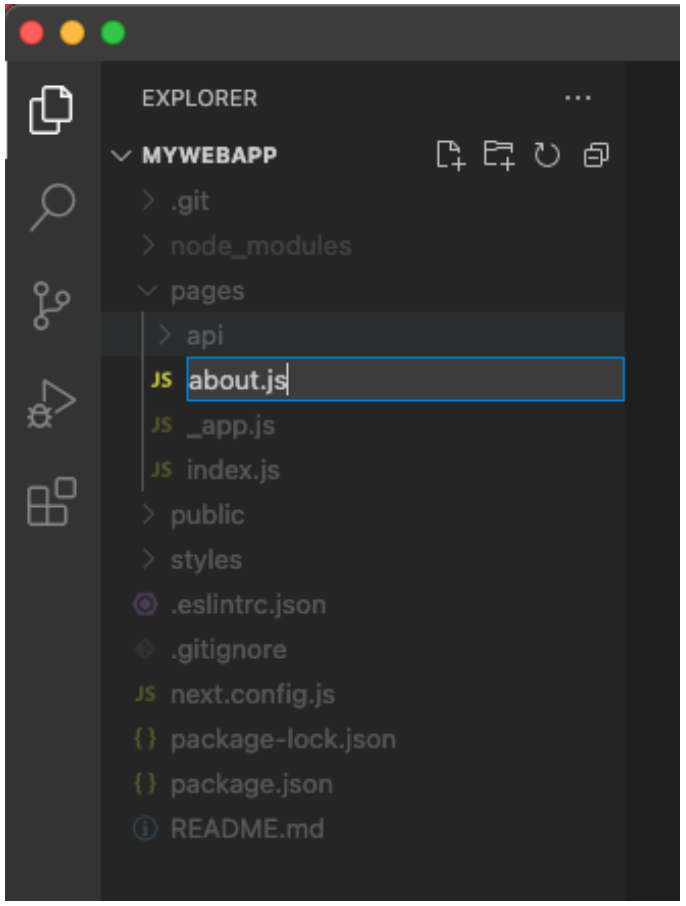
The project contents will be shown.
It starts with 3 folders: **pages**, **public** and **styles**.

5. Create a new file named **about.js** inside the **pages** folder.



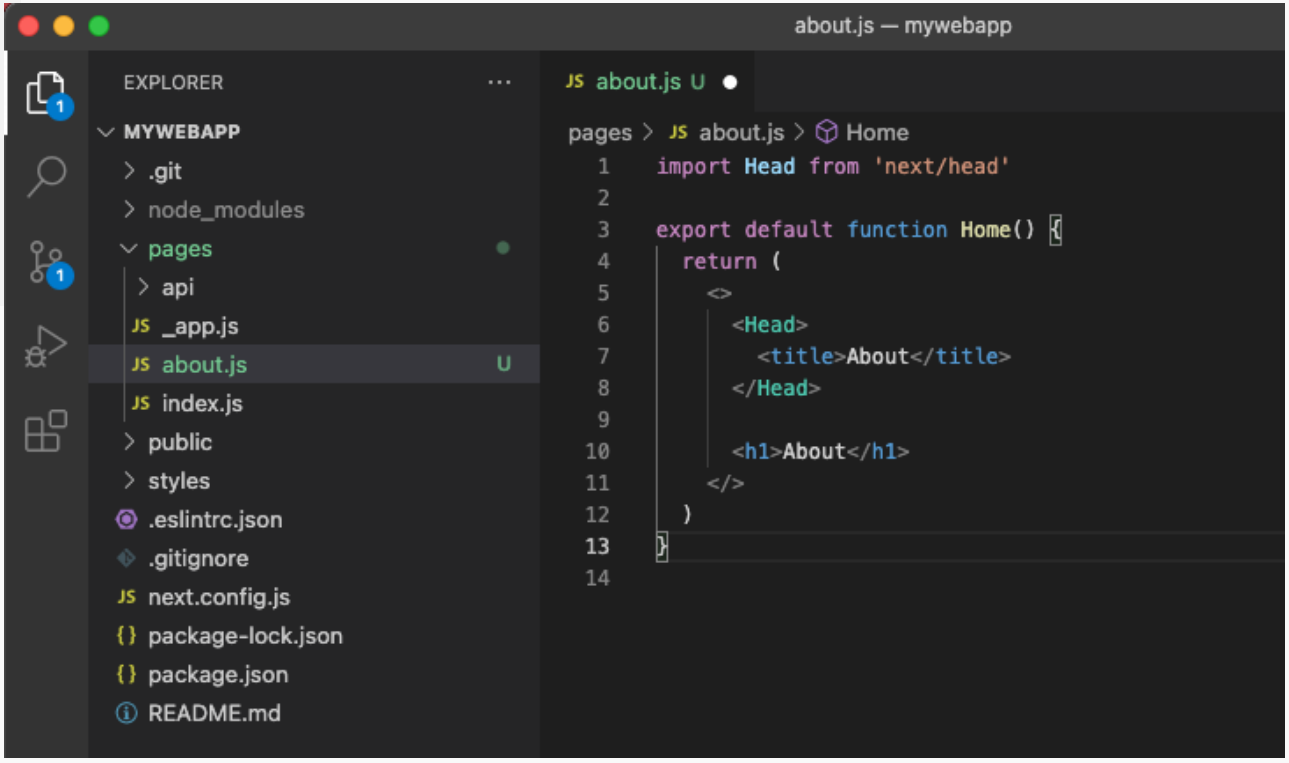
Right click on the **pages** folder and select **New File**.

Enter **about.js** as file name.



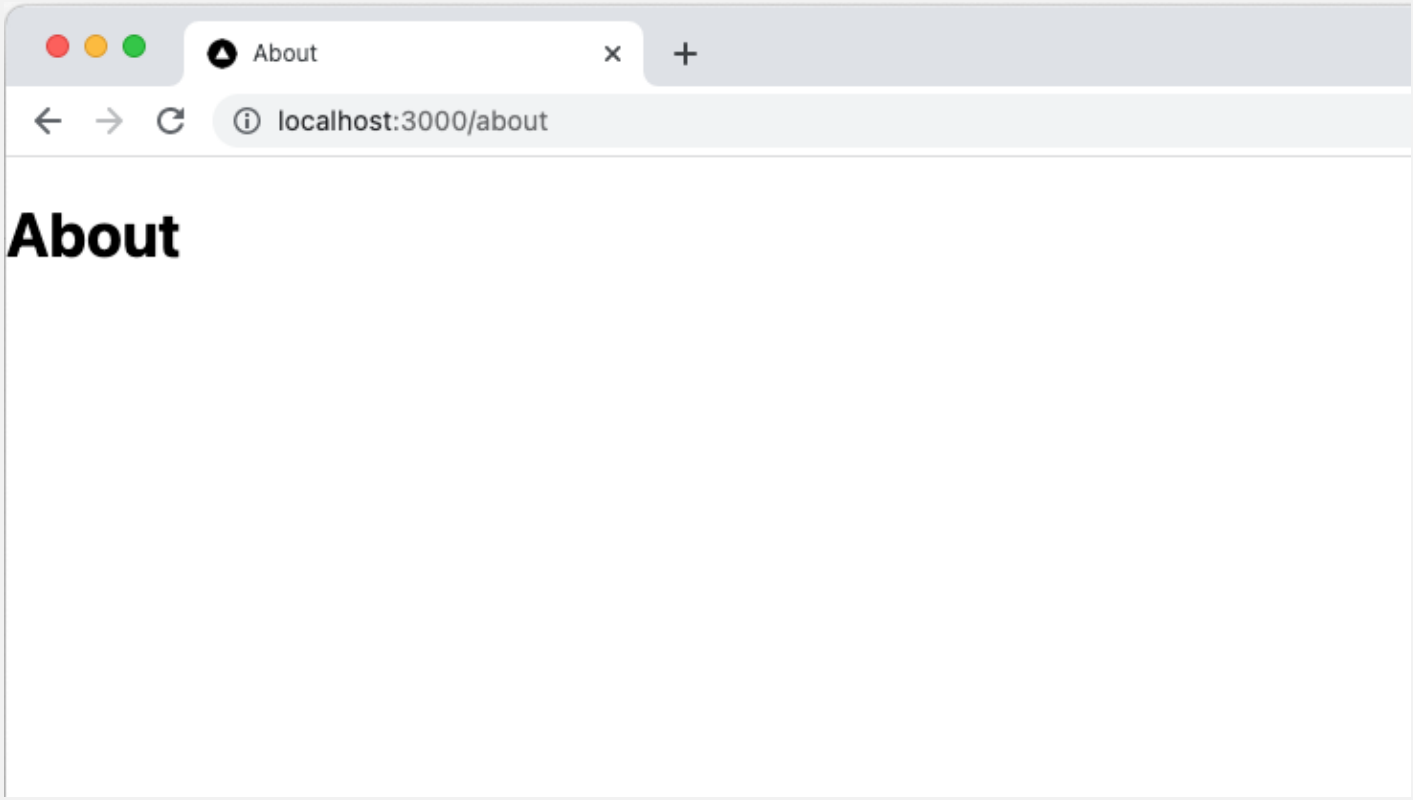
6. Copy the following code and paste it into **about.js**. Then save the file.

```
import Head from 'next/head'

export default function Home() {
  return (
    <>
      <Head>
        <title>About</title>
      </Head>

      <h1>About</h1>
    </>
  )
}
```



Now from your web browser, open:

http://localhost:3000/about

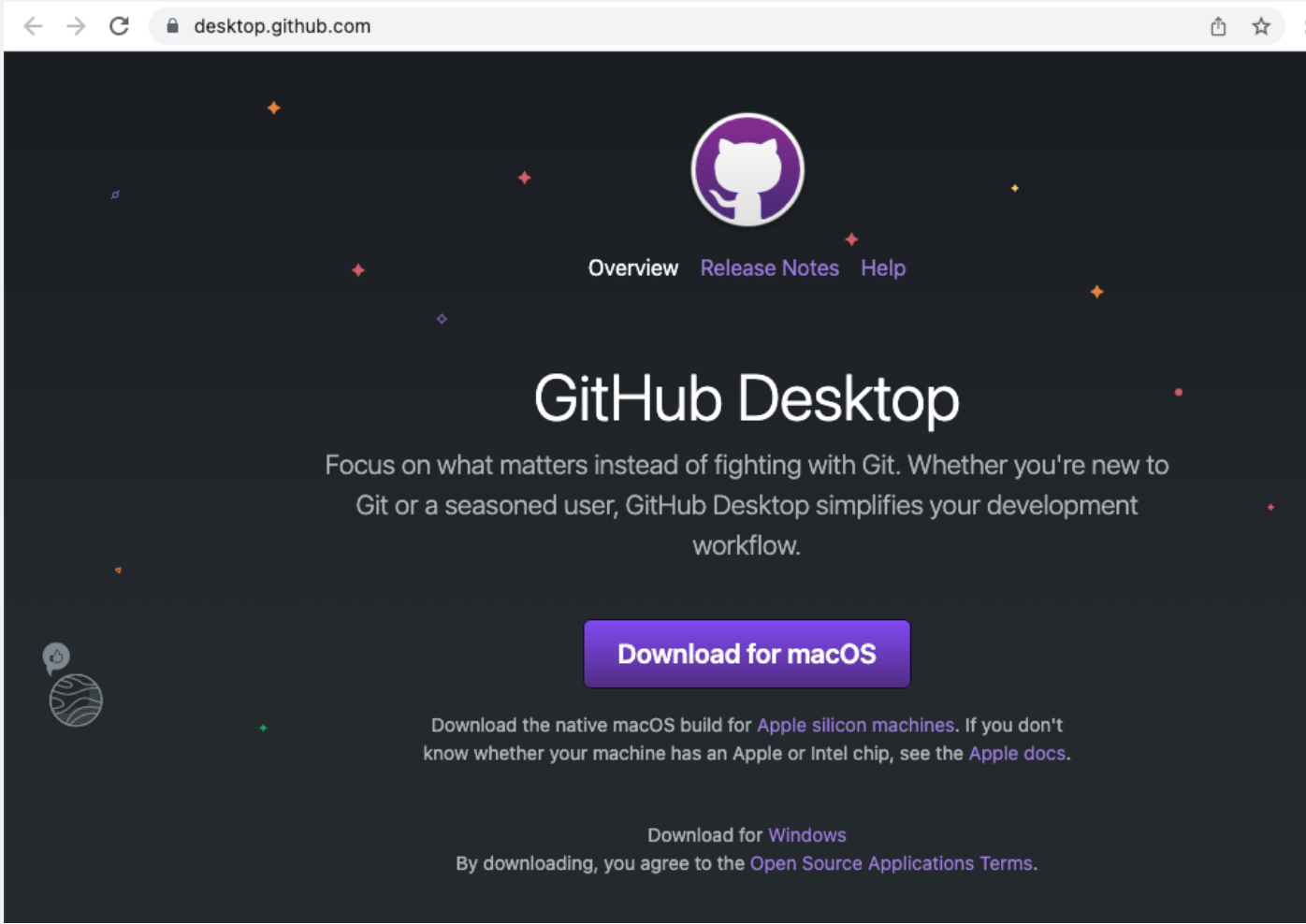


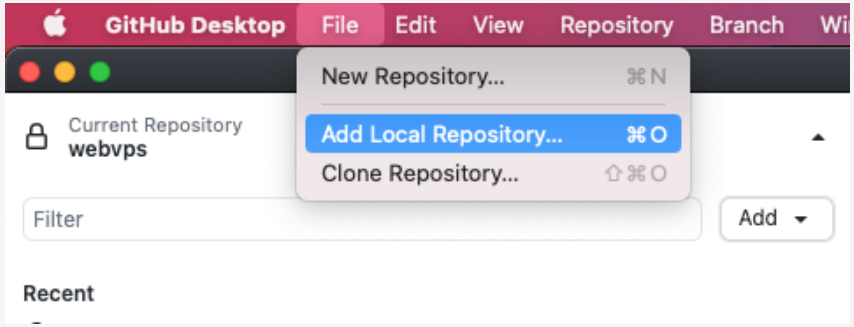
That’s great! Now you have the second page.

2. Host Your Project on Github

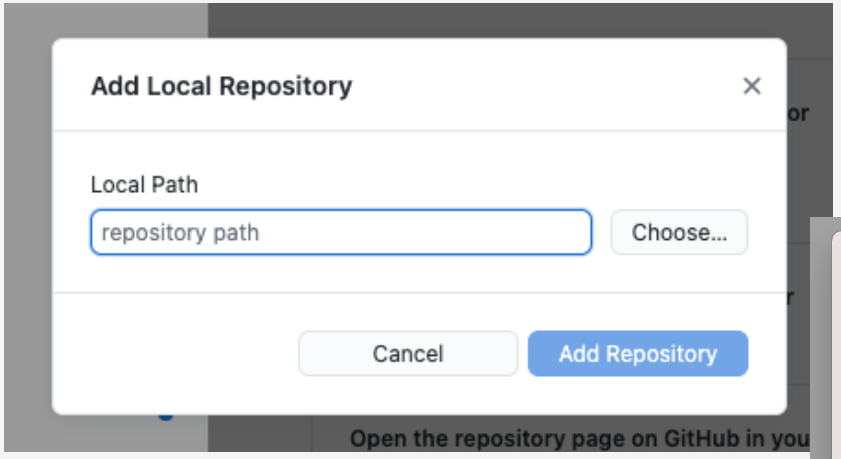
Github (https://github.com) is a code hosting platform. It helps you track code changes and work collaboratively from anywhere. If you don't have a Github account, you can signup for one.

1. To work with Github, we will use Github Desktop. You can download and install Github Desktop from https://desktop.github.com.

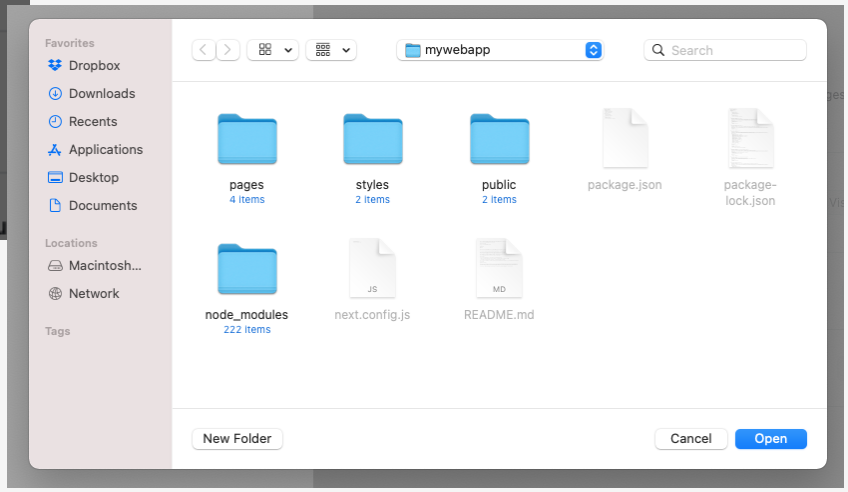


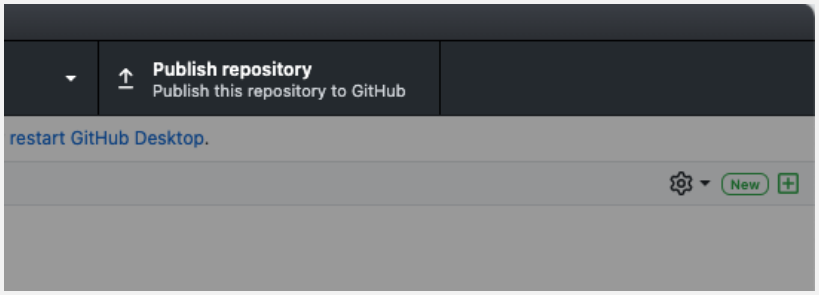


2. Open Github Desktop. From the **File** menu, select **Add Local Repository**.

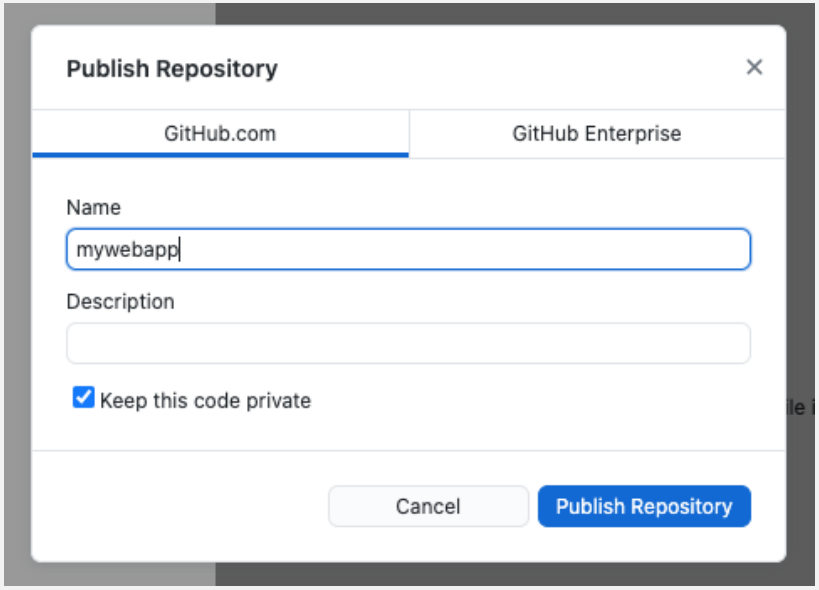


3. Click **Choose**. Browse your local computer to select your project folder **mywebapp** and click **Add Repository**.



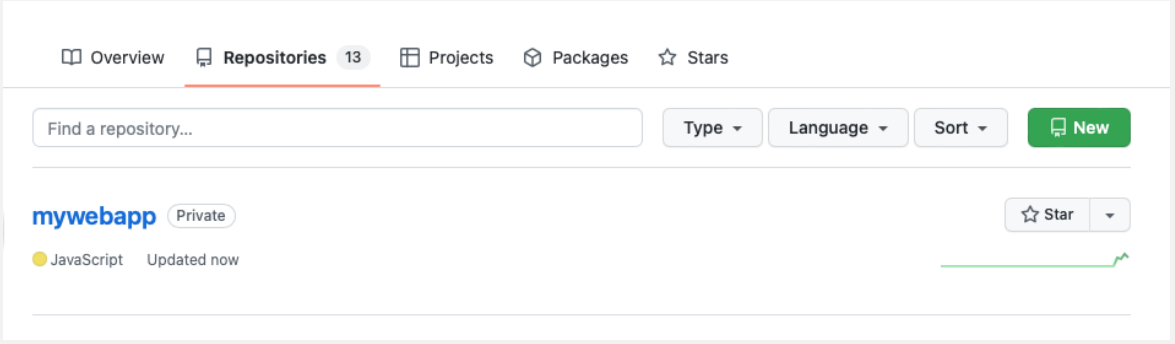


4. Then click **Publish Repository**.



5. A new dialog opens. Here you can enter your project name. Let's make the project private by checking **Keep this code private**. Then click **Publish Repository**.

6. Your project is now hosted on Github.
Open https://github.com and login. Select **Repositories** tab.
You will see your project **mywebapp** is listed.



3. Clone Your Github Project to Your VPS

1. Login to your server as the user **hobnob** and go to the **app/** directory that we created previously:

```
$ cd apps
```

2. Create a new directory for your web project.

```
$ mkdir mywebapp
```

3. Goto the directory:

```
$ cd mywebapp
```

As you login using the user **hobnob** and work from your home directory, the new created directory will be located at:

/home/hobnob/apps/mywebapp/

Now we want to clone our project code from Github into this directory.

4. We will need to install **git** first. Git is a software that helps you manage versions of your code. It is also used to connect with our remote project on Github.

```
$ sudo apt-get update
```

```
$ sudo apt-get install git-core
```

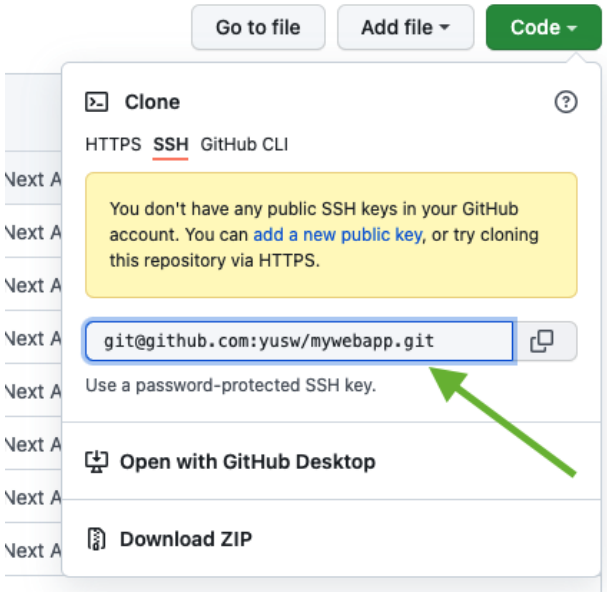
5. Go to your project directory:

```
$ cd ~/apps/mywebapp
```

6. Run git init to create a new project repository.

```
$ git init
```

7. Go to github.com and open your project. Click the **Code** button and choose **SSH** tab. Copy the displayed SSH URL.



8. Back to our VPS and run this command with the copied SSH URL as remote origin:

```
$ git remote add origin git@github.com:<username>/mywebapp.git
```

To be able to clone from a Github project, we need to prepare the security requirement. The security method we will use is called SSH key authentication.

9. Create a **.ssh** directory and a sub directory for our Github usage:

```
$ mkdir -p ~/.ssh/github
```

10. Generate an SSH key:

```
$ ssh-keygen -f ~/.ssh/github/id_rsa -t rsa -N ''
```

11. Display the key:

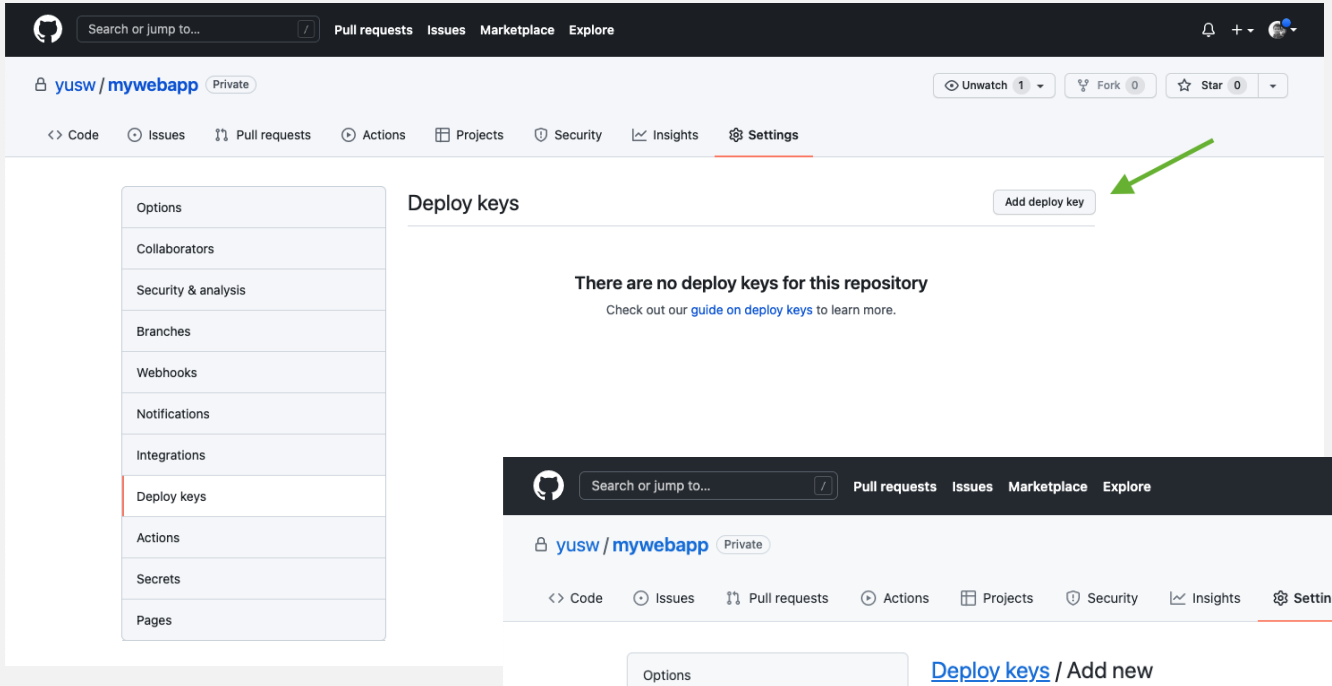
```
$ cat ~/.ssh/github/id_rsa.pub
```

12. A long key code is displayed on the screen:

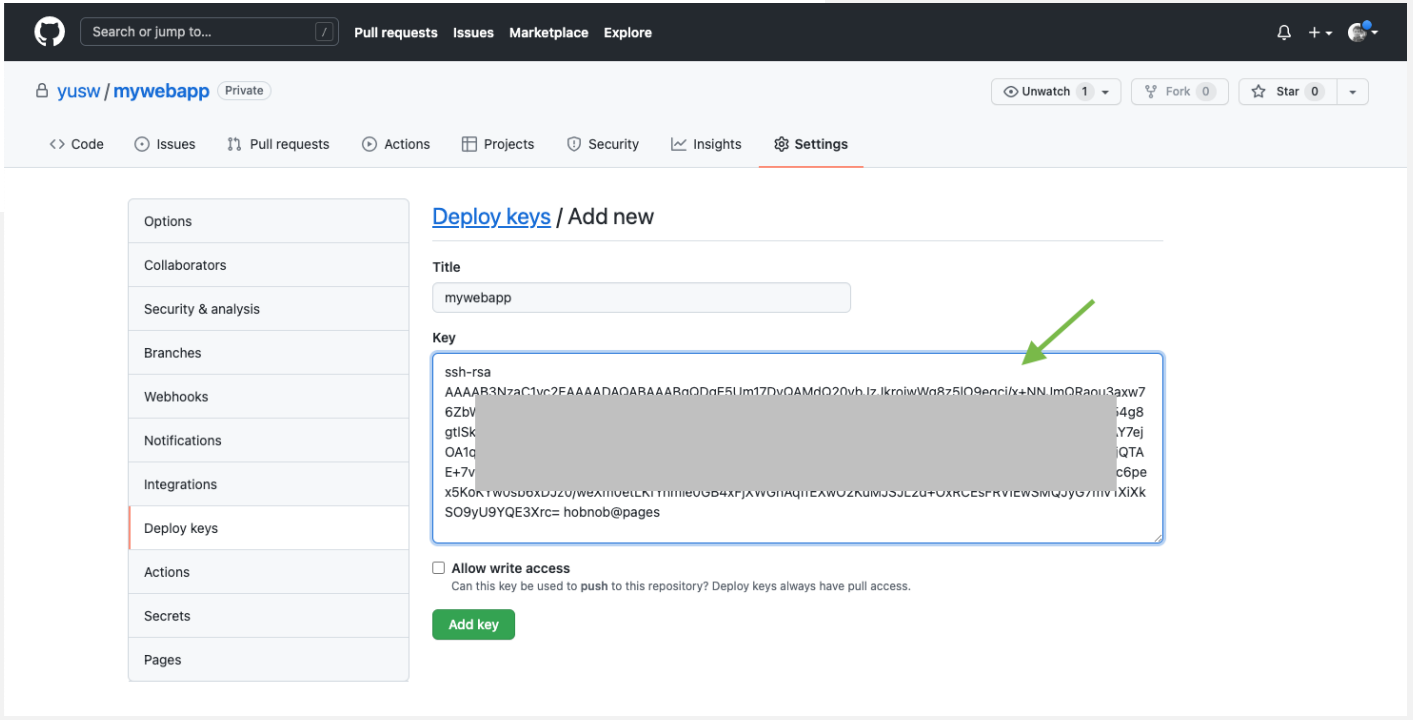
```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQDgE5Um17DyQAMdQ20vbJ...
```

Select and copy the key.

13. Now go to **github.com** and open your project. Go to **Settings** > **Deploy keys**. Click the **Add deploy key** button.



14. Add a title and paste the key. Then click **Add key**.



15. Back to your VPS, create a config file:

```
$ touch ~/.ssh/config
```

16. Edit the file:

```
$ vi ~/.ssh/config
```

Press **i** and insert the following configuration:

```
Host github.com
    IdentityFile ~/.ssh/github/id_rsa
```

Press **ESC** and then **:wq** to write (save) and quit.

17. Test the authentication by running:

```
$ ssh -T git@github.com
```

If prompted for confirmation, answer with **yes**. The test should return a successful message:

```
Hi .../mywebapp! You've successfully authenticated, ...
```

18. Now start cloning your project from Github to your VPS. Go to the the project directory:

```
$ cd ~/apps/mywebapp/
```

19. Start cloning. Your project code from Github will be copied into **~/apps/mywebapp**.

```
$ git pull -f origin main
```

20. Perform an installation. This will install your Next.js website dependencies.

```
$ npm ci
```

21. Build the application.

```
$ npm run build
```

22. Start your website.

```
$ npm start
ready – started server on 0.0.0.0:3000, url: http://localhost:3000
```

Your Next.js website has been started using port 3000.
Press **CTRL-C** to stop

22. Now run your website using PM2.
This will ensure that your website starts automatically when your server restart:

```
$ pm2 start npm --name "mywebapp" -- start
```

–name assigns a name to the PM2 process

```
$ pm2 save
```

INFO:

PM2 commands you can use:

```
$ pm2 status
$ pm2 restart <app_name>
$ pm2 reload <app_name>
$ pm2 stop <app_name>
$ pm2 delete <app_name>
```

The difference between restart and reload is that restart kills and restarts the process, whereas reload achieves a 0-second-downtime.

You will need to reconfigure OpenResty so that your website can be publicly accessible.

23. Edit the OpenResty config file.

```
$ sudo vi /etc/openresty/nginx.conf
```

Change the proxy_pass address inside **server { listen 443 ssl; ... }** with your new website address **http://localhost:3000**:

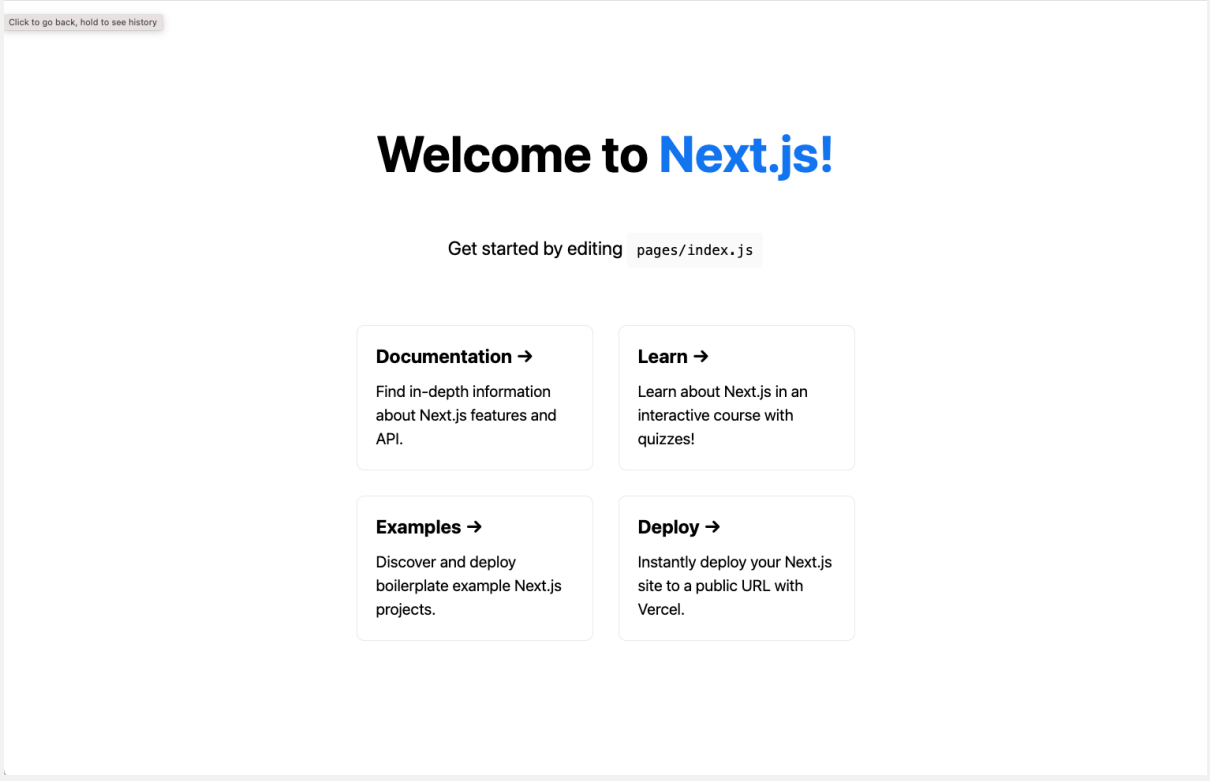
```
location / {
    proxy_pass http://localhost:3000;
    proxy_set_header Host $host;
}
```

24. Restart OpenResty.

```
$ sudo systemctl restart openresty
```

25. Your website is now ready! You can open your website from your browser using your domain URL:

https://sitemagz.com

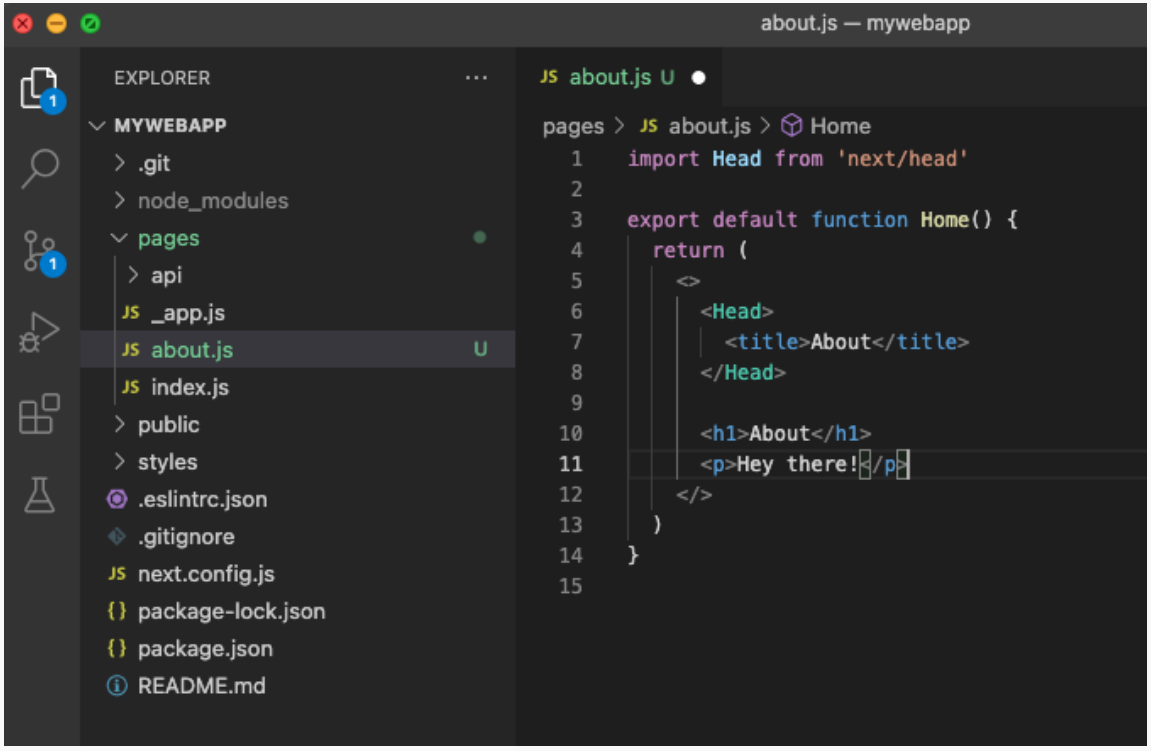


4. Setting Up Automatic Deployement

Now we will implement automatic deployment for our project. With this, everytime you make changes to your code and push to your Github, your live website will also get updated.

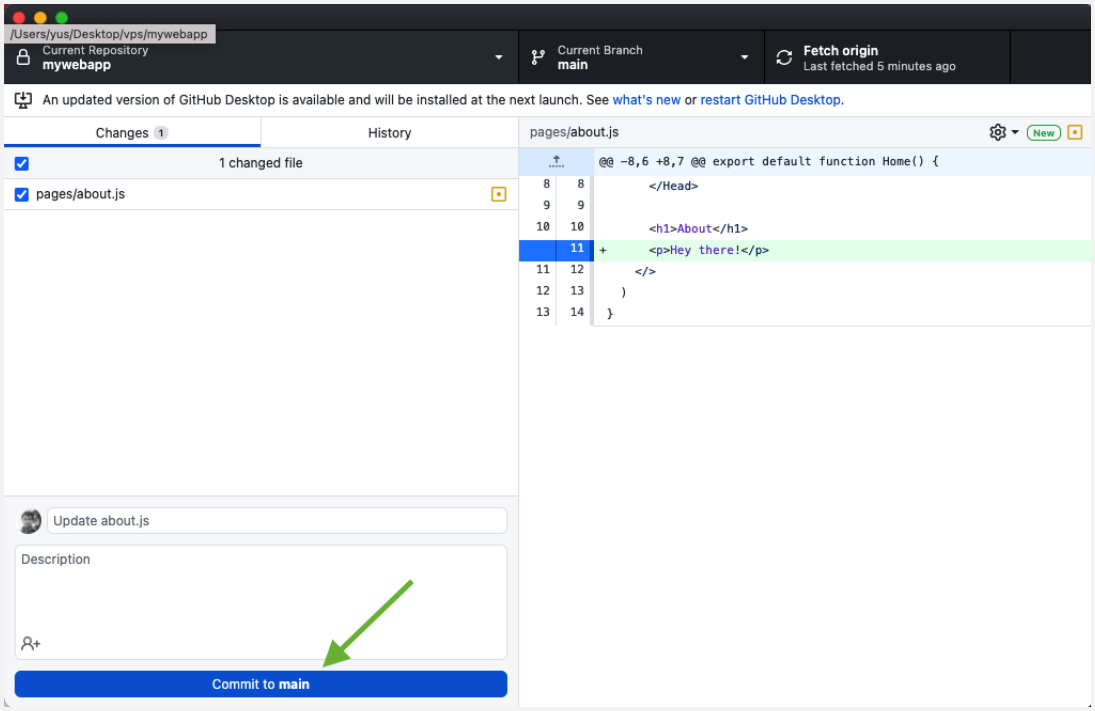
First, let's see how you can push your code changes to Github.

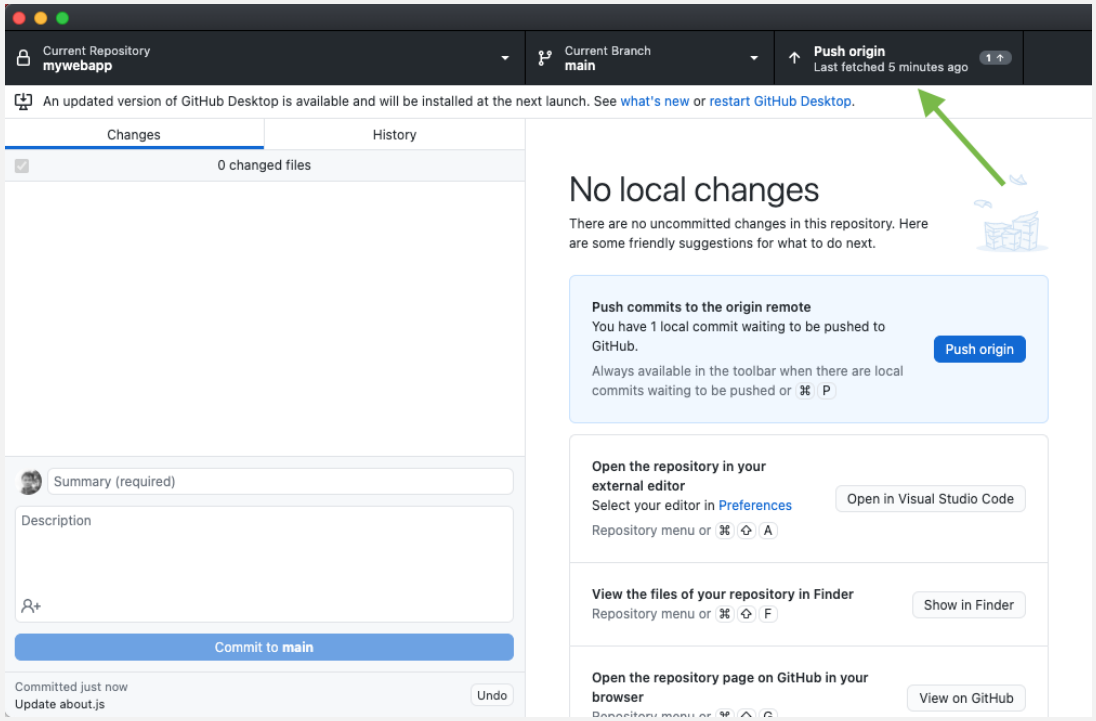
Try editing the **about.js** file you have created previously. For example, add a new paragraph ** Hey there! ** and then save.



Open Github Desktop and you will see the list of changed files. Here only the **about.js** file is shown.

Click the **Commit to main** button. This will save your changes as a ‘safe’ stage version locally.





Then click the **Push origin** button to send the commited version from local to a remote location which is your Github repository. And that’s it. You have pushed your local changes to your Github project.

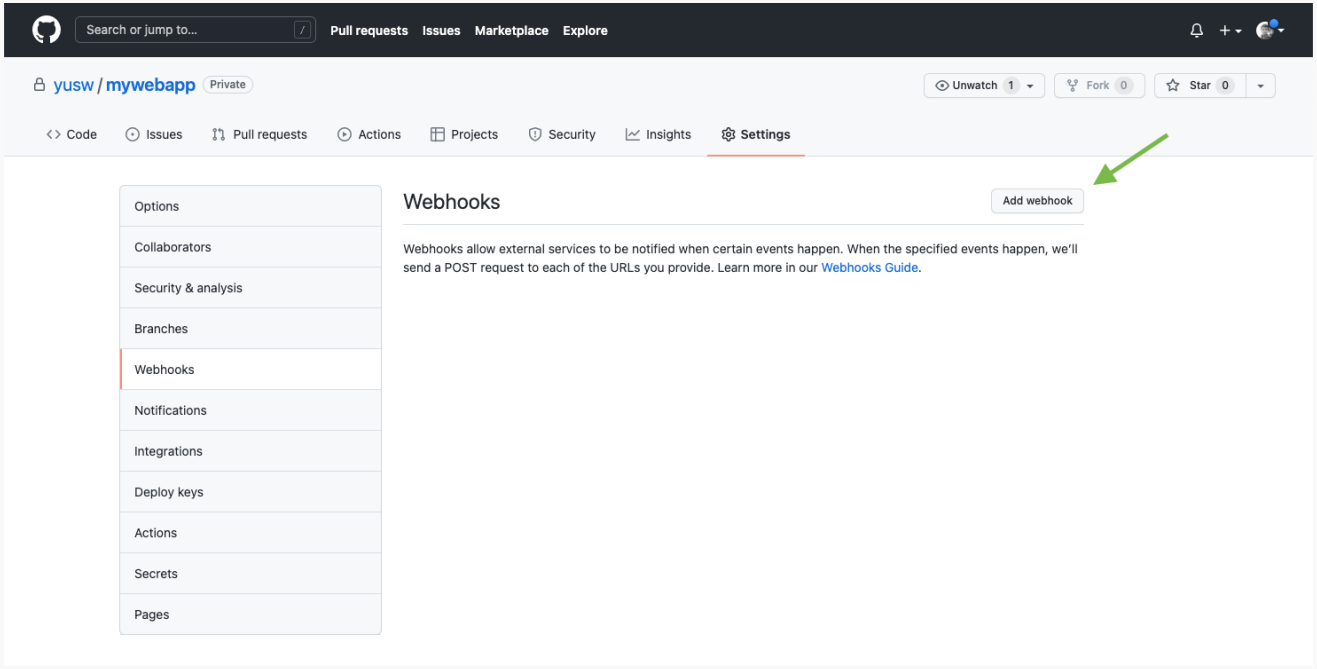
Github allows you to configure an event (called webhook) to send a HTTP request to your VPS every time you push code changes. Your VPS will be prepared to listen for the request to start the deployment process.

So here's how automatic deployment works:

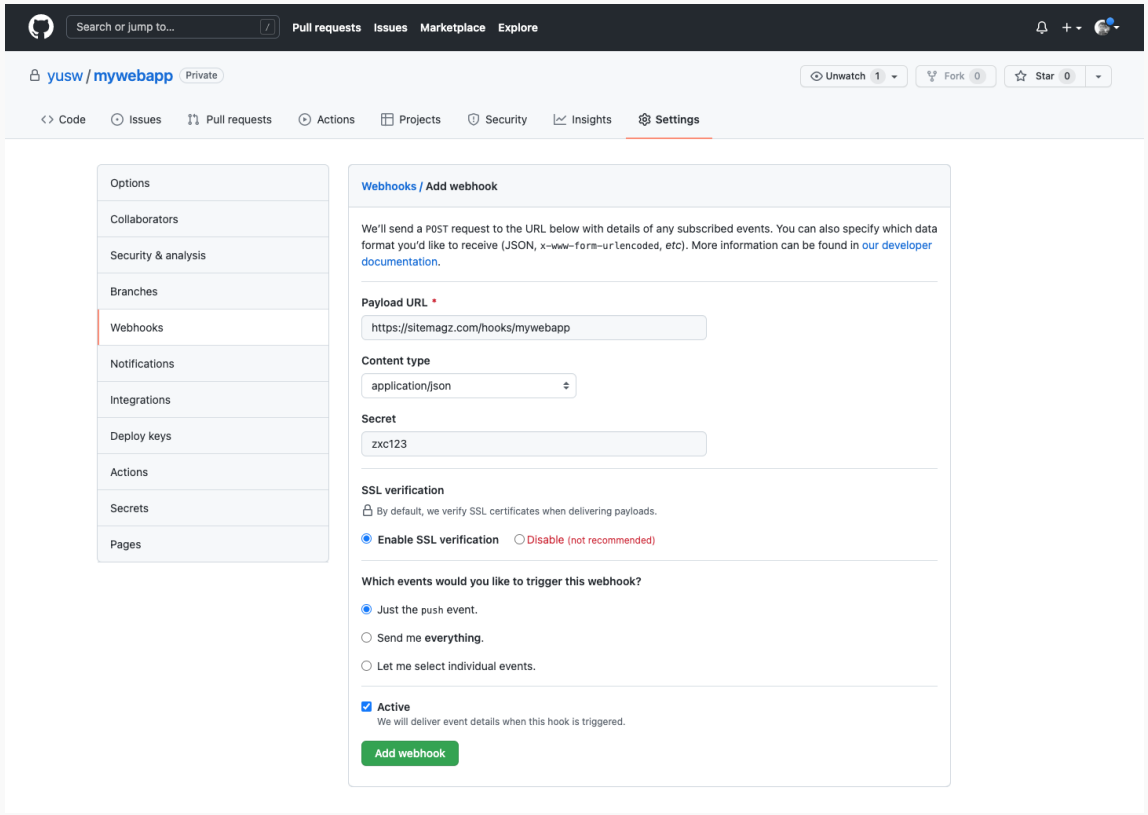
1. You push code changes to your Github
2. Github triggers a webhook that sends an HTTP request to your VPS
3. Your VPS receives the request and starts the deployment process to update your live website.

Let's follow the steps:

1. Go to github.com and open your project repository.
Open **Settings** > **Webhooks** and click the **Add webhook** button.



2. Enter Payload URL with your VPS domain URL following with **/hooks/mywebapp** as your listener path (we will configure this path on our VPS later). The Payload URL will be: **https://sitemagz.com/hooks/mywebapp**.



Choose Content Type: **application/json**, specify a secret code, for example: **zxc123**, then click **Add webhook**.

We will now create a listener that accepts requests from the Github webhook. We will create a simple Node.js application to accept the request and then run the deployment process.

3. Login to your VPS and go to the **/apps** directory

```
$ cd apps
```

4. Create a new directory named **/webhooks** for your Node.js application.

```
$ mkdir webhooks
```

5. Go to the new directory.

```
$ cd webhooks
```

6. Create a new file named **mywebapp-deploy.js**.

```
$ vi mywebapp-deploy.js
```

7. Edit **mywebapp-deploy.js** and insert the following code: (press **i** to insert).

```
const secret = "zxc123";
const repo = "/home/hobnob/apps/mywebapp/";

const http = require('http');
const crypto = require('crypto');
const exec = require('child_process').exec;

http.createServer(function (req, res) {
    req.on('data', function(chunk) {
        let sig = "sha1=" + crypto.createHmac('sha1', secret).update(chunk.toString()).digest('hex');

        if (req.headers['x-hub-signature'] == sig) {
            exec('cd ' + repo + ' && git pull -f origin main && npm ci && npm run build && pm2 restart mywebapp');

        }
    });

    res.end();
}).listen(9000);
```

Press **ESC** and then **:wq** to write (save) and quit.

Some notes on **mywebapp-deploy.js** code:

You will need to make sure that the **secret** value is the same as the one entered on Github:

```
const secret = "zxc123";
```

And your website project location is specified correctly:

```
const repo = "/home/hobnob/apps/mywebapp/";
```

Your listener app will run on port 9000:

```
http.createServer(function (req, res) {
    ...
}).listen(9000);
```

Every time Github sends a request, the listener will receive and check the secret code included, which is passed as SHA1-hashed string on the header named **x-hub-signature**. If matched, then the deployment process will be executed.

```
exec('cd ' + repo + ' && git pull -f origin main && npm ci && npm run build && pm2 restart mywebapp');
```

As seen on the code, we execute a series of commands separated with **&&**.

The commands are:

`cd <repo>`	Go to the project directory.
`git pull -f origin main`	Get (pull) the latest code from Github.
`npm ci`	Install project dependencies. Normally, **npm install** is used, but for auto deployment scenario, using **npm ci** is recommended.
`npm run build`	Build the project.
`pm2 restart mywebapp`	Perform PM2 restart.

These will perform the complete deployment, from getting the latest code from Github, building and restarting your website.

8. Run the listener code using PM2 to ensure that it will be automatically started when your server restarts.

```
$ pm2 start /home/hobnob/apps/webhooks/mywebapp-deploy.js --name "mywebapp-deploy"
```

–name assigns a name to the PM2 process. Here we give a name: **mywebapp-deploy**.

```
$ pm2 save
```

Lastly, you will need to reconfigure OpenResty so that your listener can be publicly accessible or able to receive request from Github.

9. Edit the OpenResty config file.

```
$ sudo vi /etc/openresty/nginx.conf
```

Previously you specify your listener URL (the payload URL) on the Github webhook settings: **https://<domain-name>/hooks/mywebapp** .
Now you need to add this path on the OpenResty config file and point it to your listener address: **http://localhost:9000**.
So you need to add this block inside **server { listen 443 ssl; ... }**.

```
location /hooks/ {
  proxy_pass http://localhost:9000/;
}
```

The complete code of the OpenResty config file (**nginx.conf**) can be seen blow:

```
user www-data;

events {
  worker_connections 1024;
}

http {
  lua_shared_dict auto_ssl 1m;
  lua_shared_dict auto_ssl_settings 64k;
  resolver 8.8.8.8 ipv6=off;

  init_by_lua_block {
    auto_ssl = (require "resty.auto-ssl").new()
    auto_ssl:set("allow_domain", function(domain)
      return true
    end)
    auto_ssl:init()
  }

  init_worker_by_lua_block {
    auto_ssl:init_worker()
  }

  server {
    listen 443 ssl;
    ssl_certificate_by_lua_block {
      auto_ssl:ssl_certificate()
    }

    ssl_certificate /etc/ssl/resty-auto-ssl-fallback.crt;
    ssl_certificate_key /etc/ssl/resty-auto-ssl-fallback.key;

    location / {
        proxy_pass http://localhost:3000;
        proxy_set_header Host $host;
    }

    location /hooks/ {
        proxy_pass http://localhost:9000/;
    }
  }

  server {
    listen 80;
    location /.well-known/acme-challenge/ {
      content_by_lua_block {
        auto_ssl:challenge_server()
      }
    }
    location / {
      return 301 https://$host$request_uri;
    }
  }
```

```
  server {
    listen 127.0.0.1:8999;
    client_body_buffer_size 128k;
    client_max_body_size 128k;

    location / {
      content_by_lua_block {
        auto_ssl:hook_server()
      }
    }
  }
}
```

10. Restart OpenResty.

```
$ sudo systemctl restart openresty
```

Congratulations! Now you can try making an update to your local code and push the changes to Github. Then wait a few seconds and check your website from your browser. The changes should be applied and can be seen immediately.

PART 2

The Application

This will be a continuation of our simple Next.js project from Part 1 (chapter 13). In this part, we will enhance our web application to accept user signups and allow users to create their own website.

1. Setting-up Database

We now need a backend database for our application. We will use MongoDB.

1. Installing MongoDB

1. Connect to your server as the user, **hobnob**.

```
$ ssh hobnob@<server-ip>
```

2. Install the MongoDB package.

```
$ curl -fsSL https://www.mongodb.org/static/pgp/server-5.0.asc | sudo apt-key add -
```

```
$ echo "deb [ arch=amd64,arm64 ] https://repo.mongodb.org/apt/ubuntu bionic/mongodb-org/5.0 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-5.0.list
```

```
$ sudo apt update
```

```
$ sudo apt install mongodb-org
```

3. Start the MongoDB service.

```
$ sudo systemctl start mongod.service
```

4. Enable MongoDB to run on startup.

```
$ sudo systemctl enable mongod
```

INFO

To check service status:

```
$ sudo systemctl status mongod
```

To check connection to MongoDB:

```
$ mongo --eval 'db.runCommand({ connectionStatus: 1 })'
```

2. Adding a database user

1. Enter the MongoDB shell. The MongoDB shell is an interface to run database queries.

```
$ mongosh
```

2. Show the list of databases:

```
show dbs
  admin 135 kB
  config 36.9 kB
  local 73.7 kB
```

3. Specify that you want to use the **admin** database.

```
use admin
```

4. Create a database user. In this example we create a user **hobnob**.

```
db.createUser(
  {
    user: "hobnob",
    pwd: "c7dah5sw12",
    roles: [ { role: "userAdminAnyDatabase", db: "admin" }, "readWriteAnyDatabase" ]
  }
)
```

5. To display users:

```
db.getUsers()
```

6. Exit the MongoDB shell.

```
exit
```

3. Securing & Configuring Remote Access

1. Add a rule to allow your computer (IP) to access the MongoDB server.

```
$ sudo ufw allow from <your-computer-ip> to any port 27017
```

Update <your-computer-ip> with your computer public IP address. To check your computer's public IP address, you can open Google and search with keyword: **my ip**.

2. Check the added rule

```
$ sudo ufw status
```

2. Edit the MongoDB configuration file.

```
$ sudo vi /etc/mongod.conf
```

Update the following lines:

```
. . .
# network interfaces
net:
 port: 27017
 bindIp: 127.0.0.1,<mongodb-server-ip>

security:
 authorization: enabled
```

Update <mongodb-server-ip> with your MongoDB server IP.

3. Restart the service.

```
$ sudo systemctl restart mongod
```

Now you will need to be authenticated to enter the MongoDB shell.
Try running the shell using the newly added user, **hobnob**:

```
$ mongosh -u hobnob -p --authenticationDatabase admin
```

To test remote connection, open another terminal and run:

```
$ nc -zv <mongodb-server-ip> 27017
```

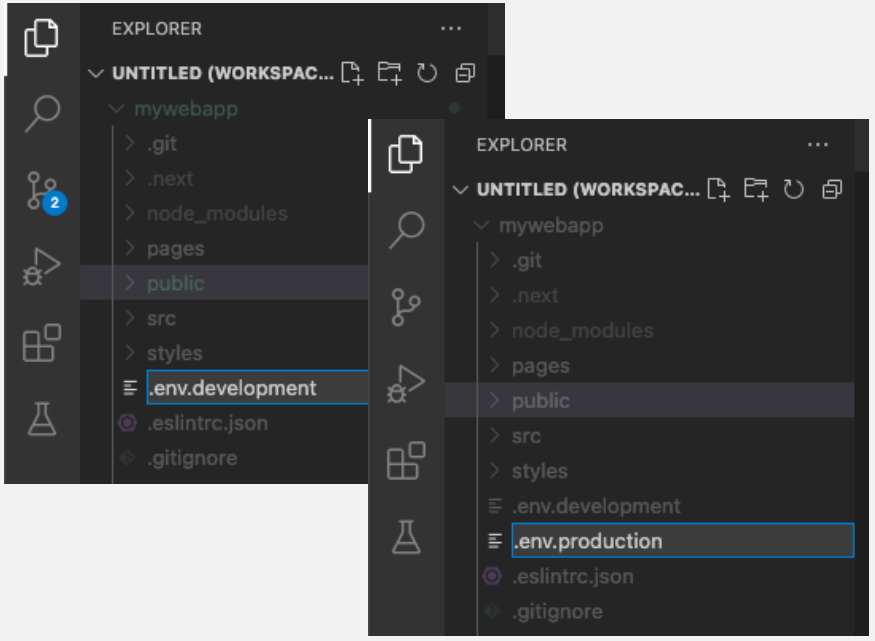
4. Connecting to the database

Now we will try connecting our web application to the MongoDB database. Let's go back again to our previous simple Next.js website/application.

1. Create a new file named **.env.development** and add the following code.

```
MONGODB_USERNAME=hobnob
MONGODB_PASSWORD=c7dah5sw12
MONGODB_HOSTNAME=<mongodb-server-ip>
MONGODB_PORT=27017
MONGODB_DATABASE=nextsite
```

Then create another file named **.env.production** and add the same code.



Here we specify configuration variables to connect to our database. The **.env.development** file will be used during development while **.env.production** will be used in production/deployment.
For our application, we name our database **nextsite**, so we set **MONGODB_DATABASE=nextsite**.

2. Open terminal, go to your project folder and run the app:

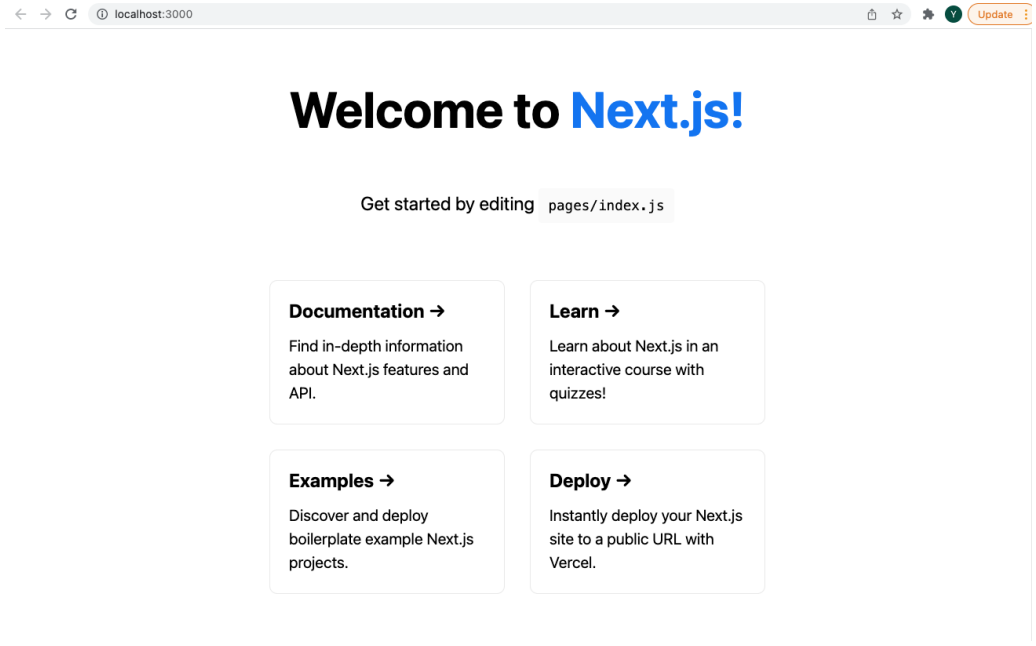
```
$ cd mywebapp
```

```
$ npm run dev
```

If you already have the application running, stop the app and re-run. This will apply the updated configuration we've just made.

3. Open your browser and go to:

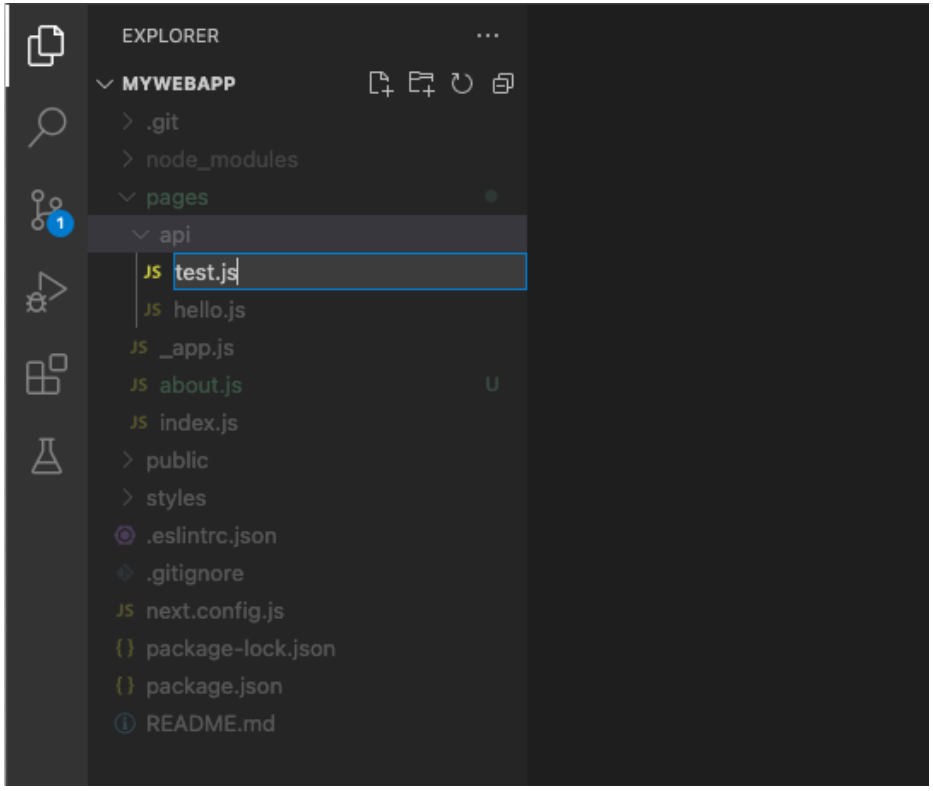
http://localhost:3000



4. Go back to the terminal, and install the MongoDB library for our application.

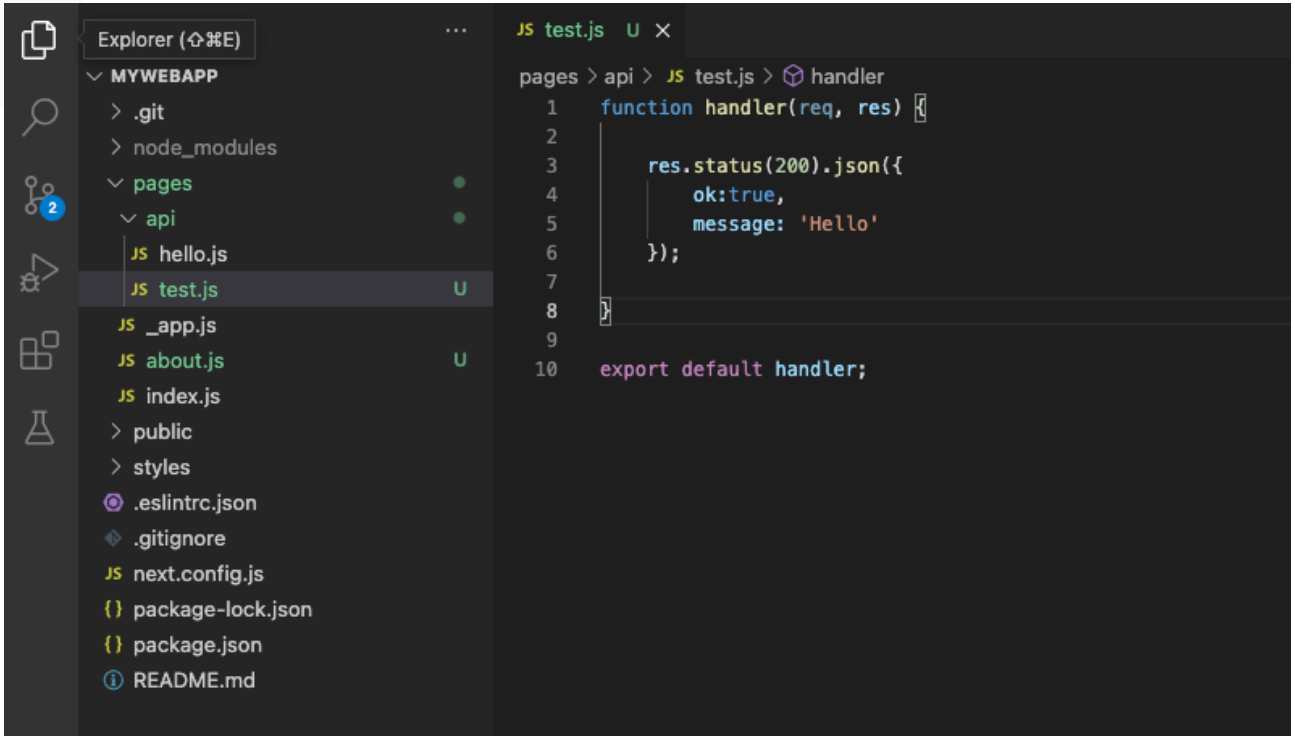
```
$ npm i mongodb
```

5. Head to Visual Studio Code and create a new file **pages/api/test.js**.



6. Enter the following code:

```
function handler(req, res) {

    res.status(200).json({
        ok:true,
        message: 'Hello'
    });

}

export default handler;
```



Here you have created a simple API (stands for Application Programming Interface) route that can be accessed from:
http://localhost:3000/api/test

An API route is a specific path (in this example **/api/test**) that is available for use by our application to get specific information. In this example, the API route returns a data in JSON format, that is:

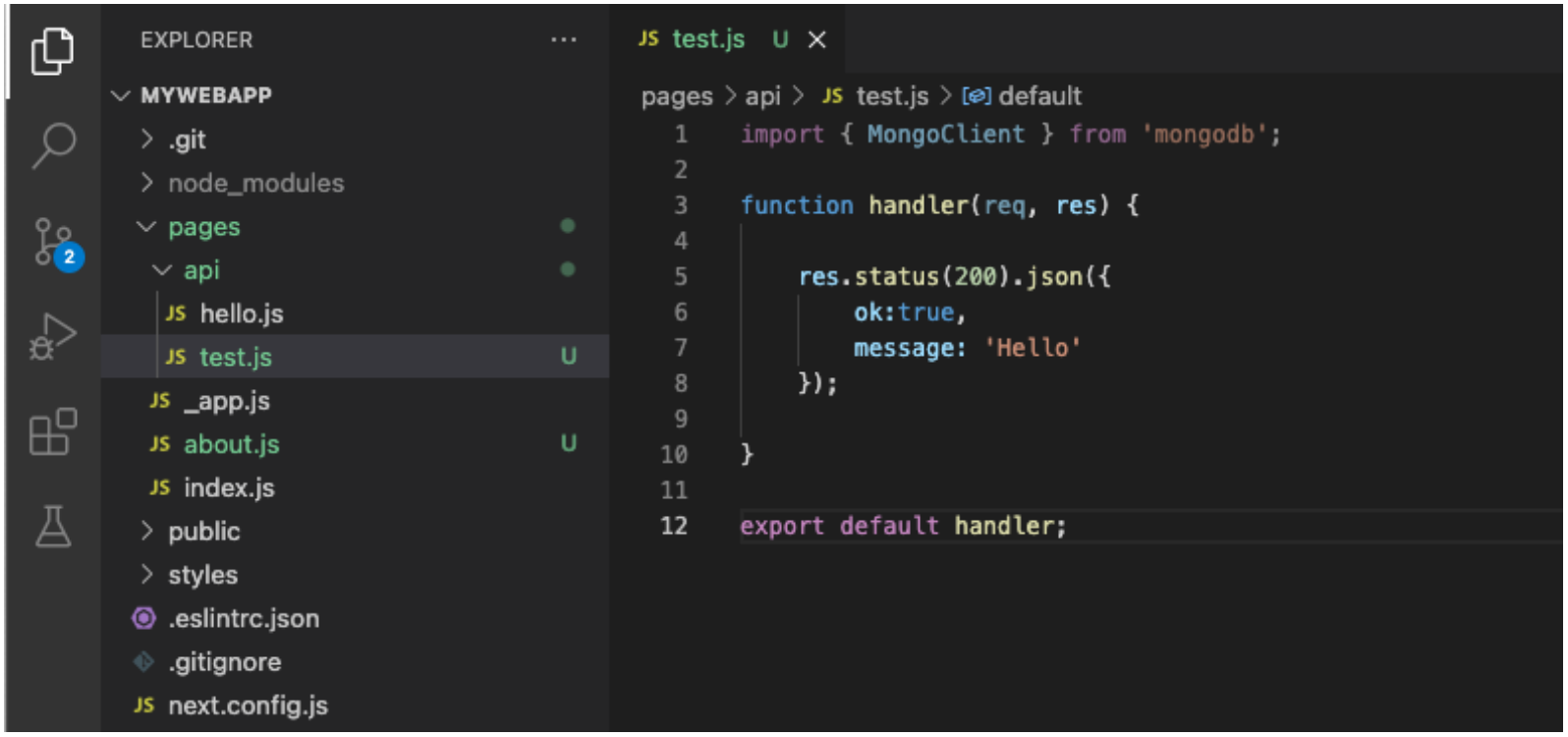
```
{
  ok:true,
  message: 'Hello'
}
```

7. To test, open your browser and go to:

http://localhost:3000/api/test

8. Let's add more code to the file **pages/api/test.js**.
We want to use the MongoDB library we have installed previously. For this, add an import statement:

```
import { MongoClient } from 'mongodb';
```



9. Under the import line, create a new function named **connectDatabase()**.

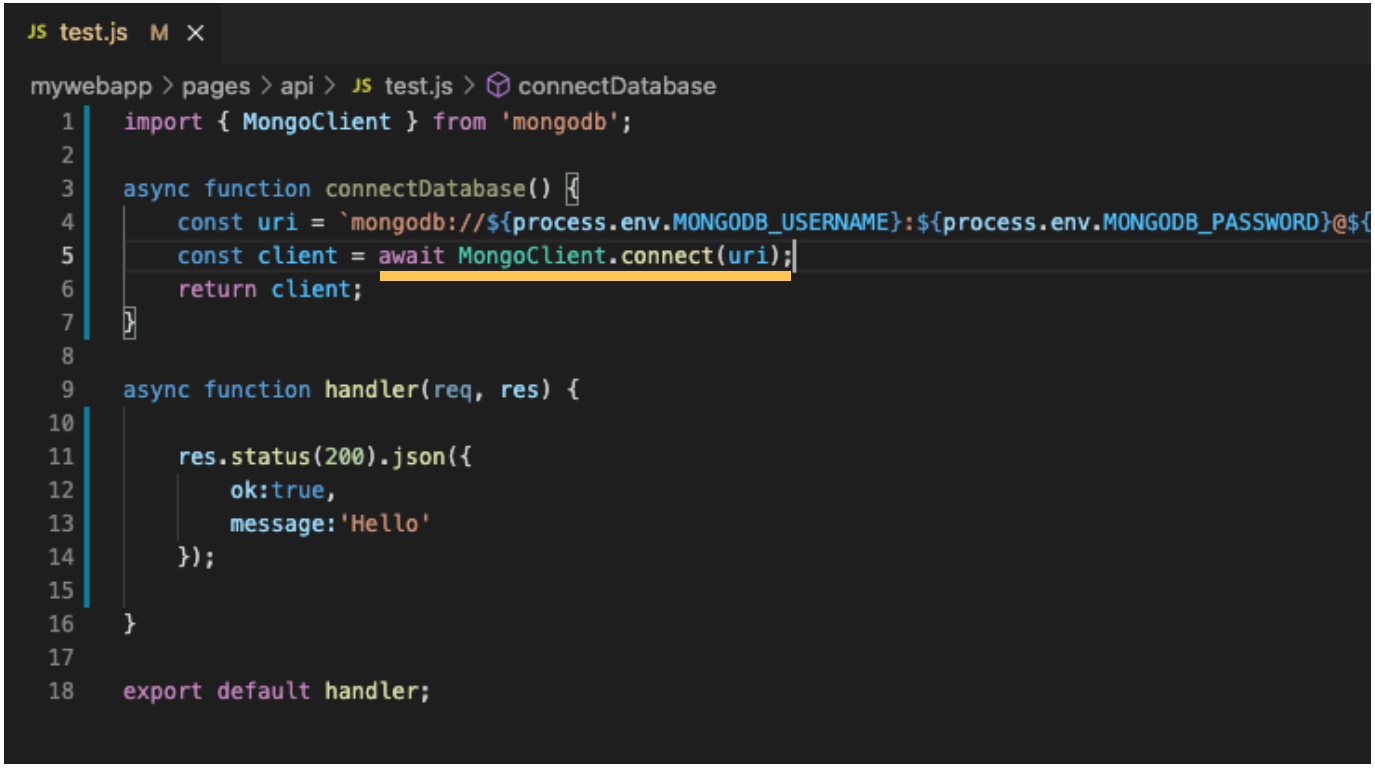
```
async function connectDatabase() {
    const uri =
`mongodb://${process.env.MONGODB_USERNAME}:${process.env.MONGODB_PASSWORD}@${process.env.MONGODB_HOSTNAME}:
${process.env.MONGODB_PORT}/${process.env.MONGODB_DATABASE}?authSource=admin`;
    const client = await MongoClient.connect(uri);
    return client;
}
```

Within this function, we construct a URI string. URI (stands for Uniform Resource Identifier) is similar to URL. The URI contains our database info with format:
mongodb://<username>:<password>@<hostname>:<port>/<database>?<options>

We get the database info from variables we added previously in the **.env.development** and **.env.production** files. To get the variable values, we simply use:

- process.env.MONGODB_USERNAME
- process.env.MONGODB_PASSWORD
- process.env.MONGODB_HOSTNAME
- process.env.MONGODB_PORT
- process.env.MONGODB_DATABASE

Then the constructed URI is passed as a parameter in **MongoClient.connect(uri)** which performs the database connection.



The keyword **await** prior to **MongoClient.connect(uri)** makes the process wait until the result returns. By using the await keyword, we also need to add the **async** keyword prior to the function. The async keyword allows the function to deliver the result asynchronously. Asynchronous means it will wait for the result (promise) in a non-blocking way until the promise fulfills.

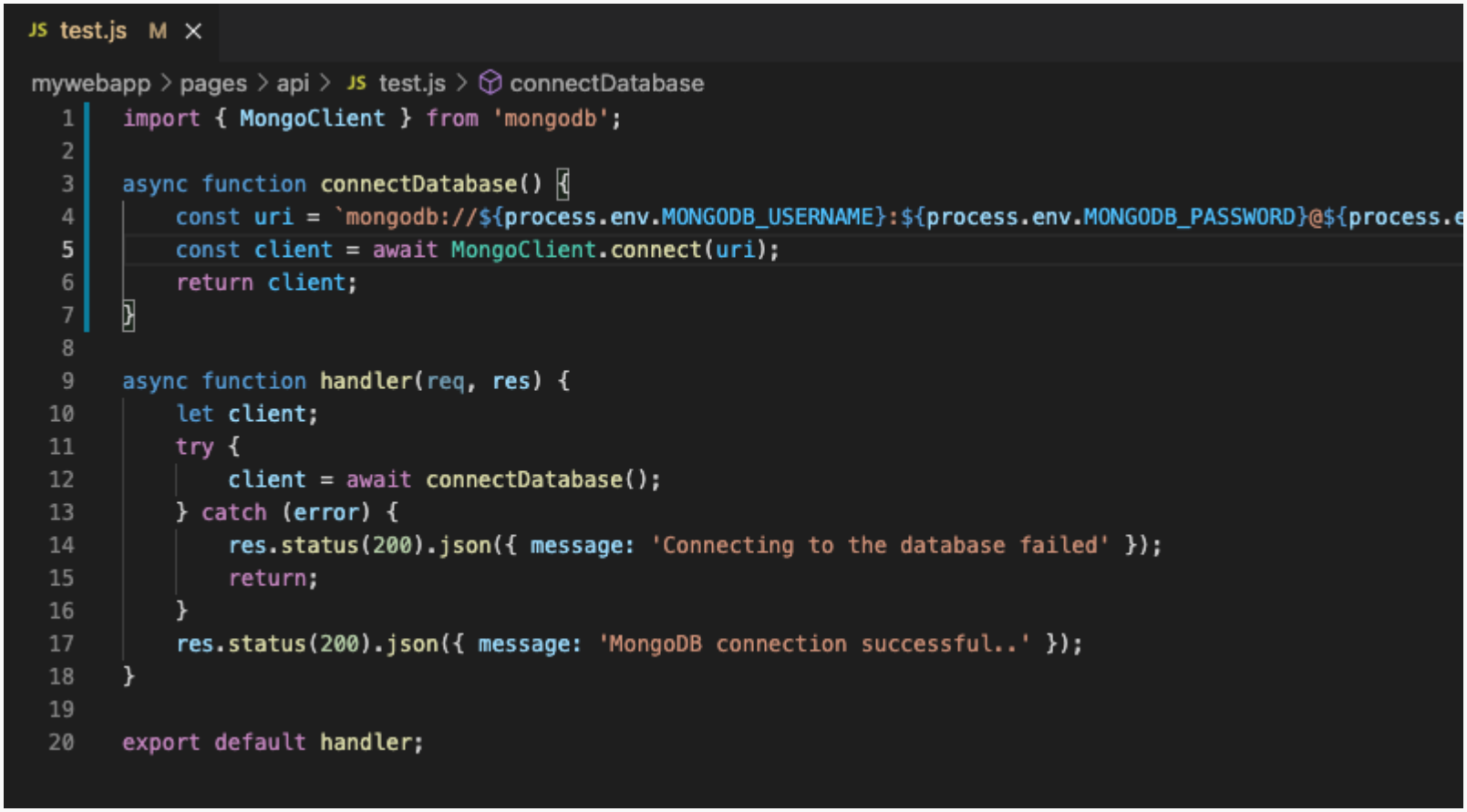
The function then returns an object (we name it **client**) that we will later use for database operations.

10. Now update the handler function with the new code:

```
async function handler(req, res) {
    let client;
    try {
        client = await connectDatabase();
    } catch (error) {
        res.status(200).json({ message: 'Connecting to the database failed' });
        return;
    }
    res.status(200).json({ message: 'MongoDB connection successful..' });
}
```

Here we try to connect to the database. If connection is successful, we return a JSON data with the message "**MongoDB connection successful..**".

Here is the complete code.



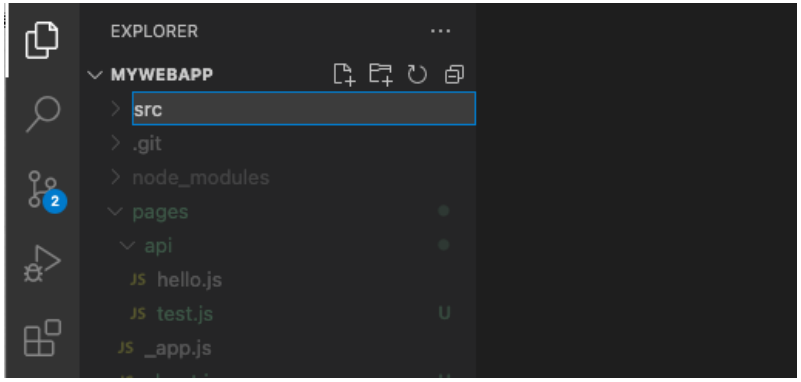
11. To test, open your browser again and go to:

http://localhost:3000/api/test

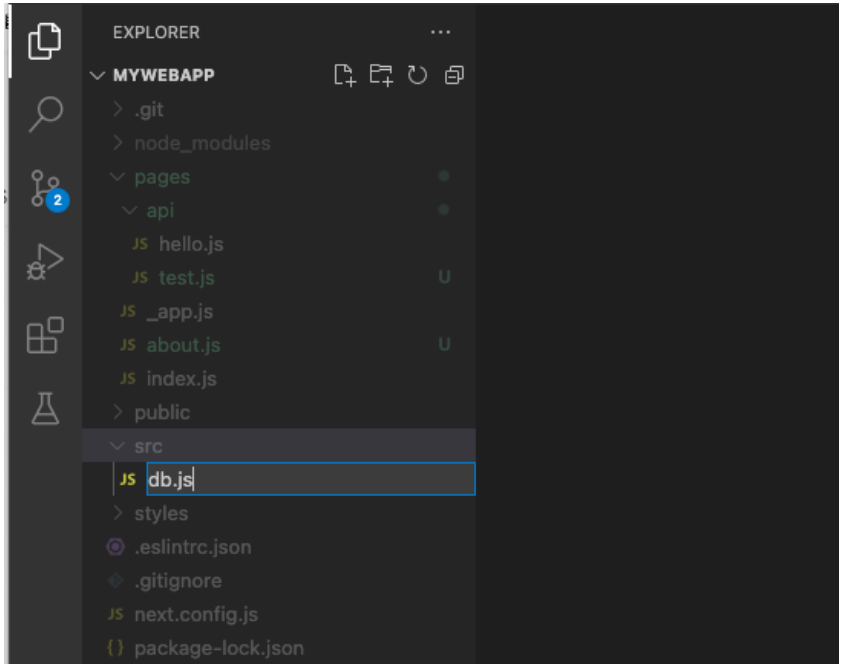
Congratulations! Now your web application is ready to use the database.

2. Creating a Database Utility

1. Now let's create a folder in our project named **src**.



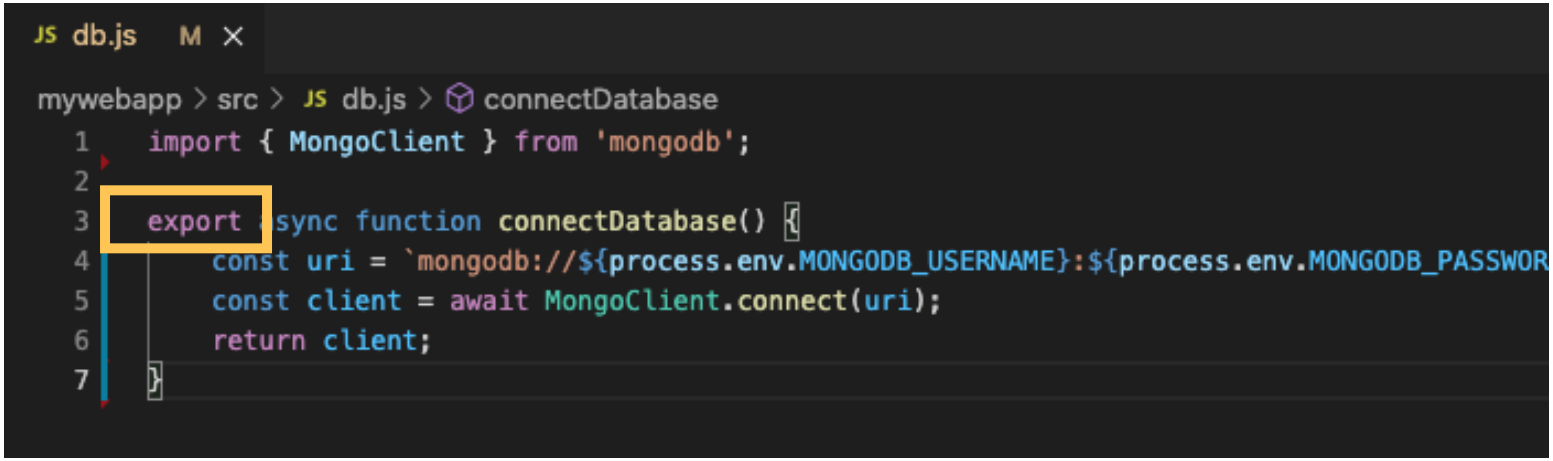
2. Inside the **src** folder, create a new file named **db.js**.



3. Copy the previous code (the **connectDatabase** function) into this file with a little change:

```
import { MongoClient } from 'mongodb';

export async function connectDatabase() {
    const uri =
`mongodb://${process.env.MONGODB_USERNAME}:${process.env.MONGODB_PASSWORD}@${process.env.MONGOD
B_HOSTNAME}:${process.env.MONGODB_PORT}/${process.env.MONGODB_DATABASE}?authSource=admin`;
    const client = await MongoClient.connect(uri);
    return client;
}
```

Here we just added the keyword **export** prior to the function so that it can be imported to be used in other places. This file will be our utility to simplify the database connection needed in our application.

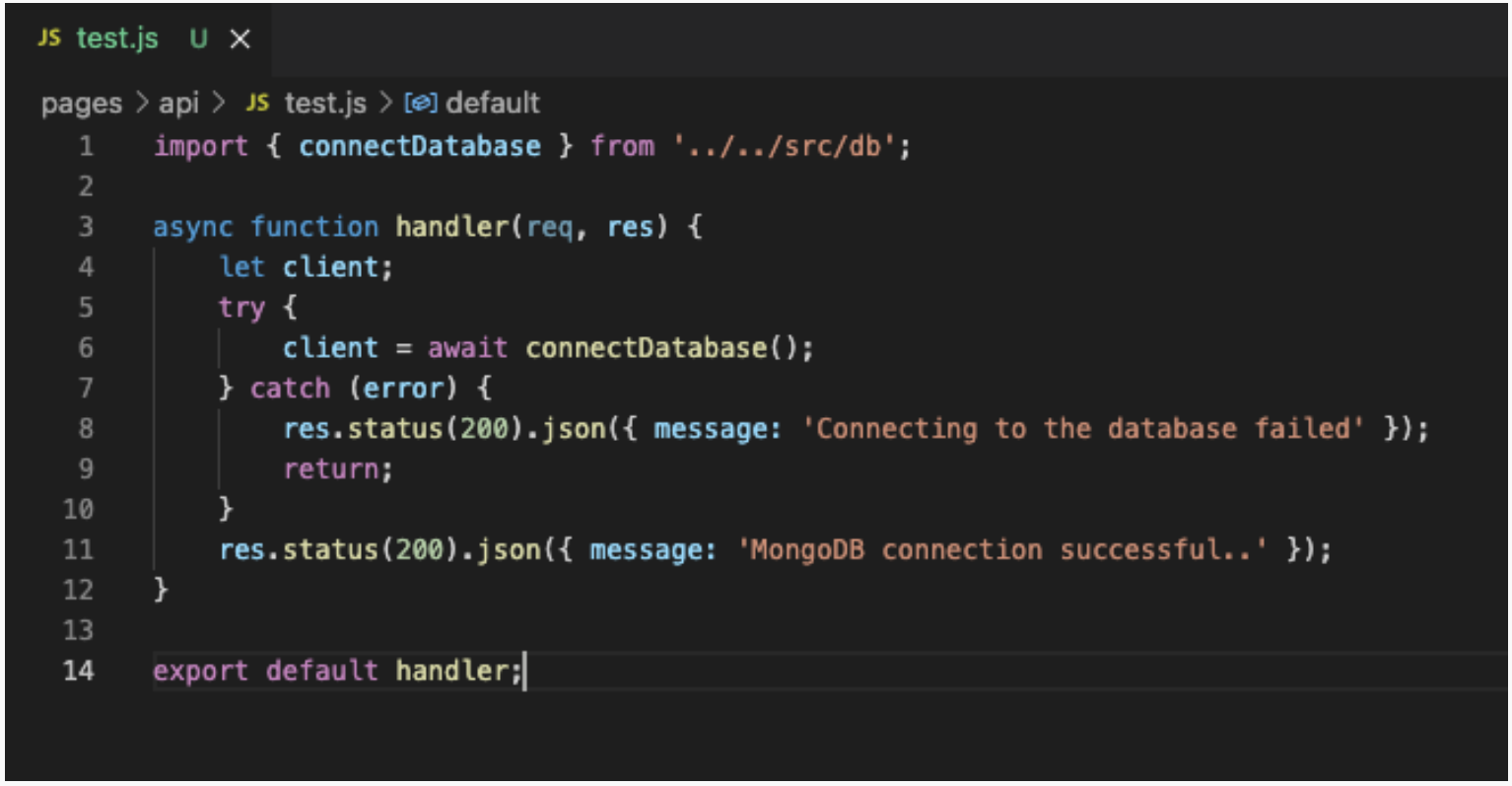


4. Head back to **pages/api/test.js** and import the **connectDatabase** function from our **db.js** utility.

```
import { connectDatabase } from '../../src/db';

async function handler(req, res) {
    let client;
    try {
        client = await connectDatabase();
    } catch (error) {
        res.status(200).json({ message: 'Connecting to the database failed' });
        return;
    }
    res.status(200).json({ message: 'MongoDB connection successful..' });
}

export default handler;
```

When importing, we specify the path relative to the current file: '**../../src/db**'. We also don't need to specify the **js** extension here.

Here is the simplified code.



5. Let’s test again. Open your browser and go to:

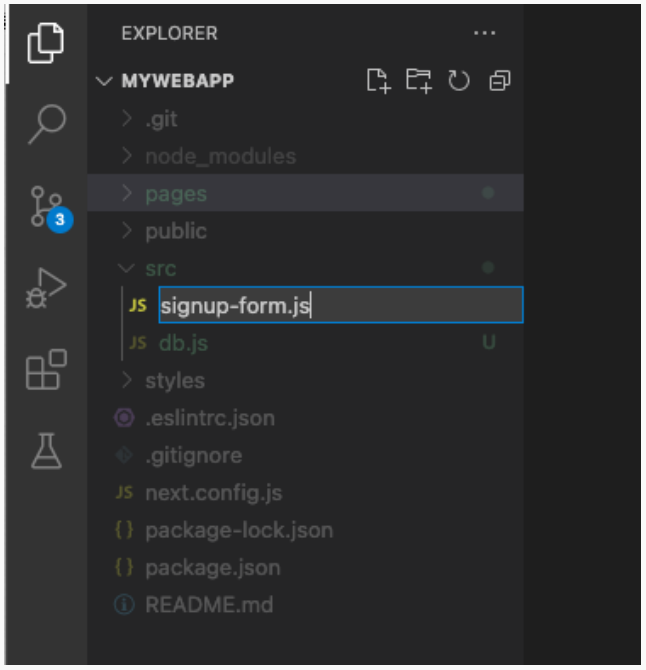
http://localhost:3000/api/test

3. Creating a Component

Remember that how our website is created using Next.js framework? The Next.js framework is built on top of React (client side) and Node.js (server side). Previously, we worked on the server side parts (the database connection, the API route). Now we will work with the client side part, that is the React part.

In this chapter we start with creating a simple **React component**. Don't worry if you haven't used React before. Just follow the steps and you'll see how it works in our application.

1. Create a new js file in the **src** directory. Let's name it **signup-form.js**.



2. Enter the following code.

```
function SignupForm() {
    return <h1>Signup</h1>;
}

export default SignupForm;
```

The keyword **export** in the last line makes the function available to be used in other places in our application

This function will display a title using <h1> element and that's it! You have created a simple component that is available for use in our application.

The **SignupForm()** function just returns an HTML, but actually it a **JSX** (JavaScript XML). JSX is a syntax extension to Javascript that converts the HTML tags into React elements. So what's the difference between writing in JSX and in HTML? Let's see this quick comparison:

In HTML, you write the class and style attributes like this:

```
<div class="dashboard" style="width:400px;">
</div>
```

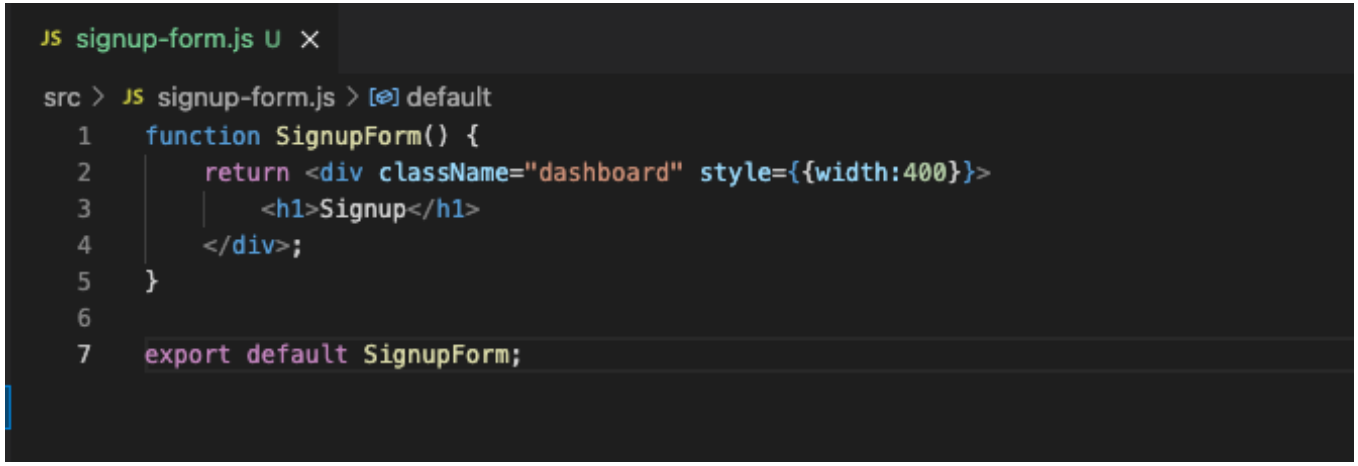
In JSX, you'd write:

```
<div className="dashboard" style={{width:400}}>
</div>
```

To see more on JSX, you can visit: https://reactjs.org/docs/introducing-jsx.html

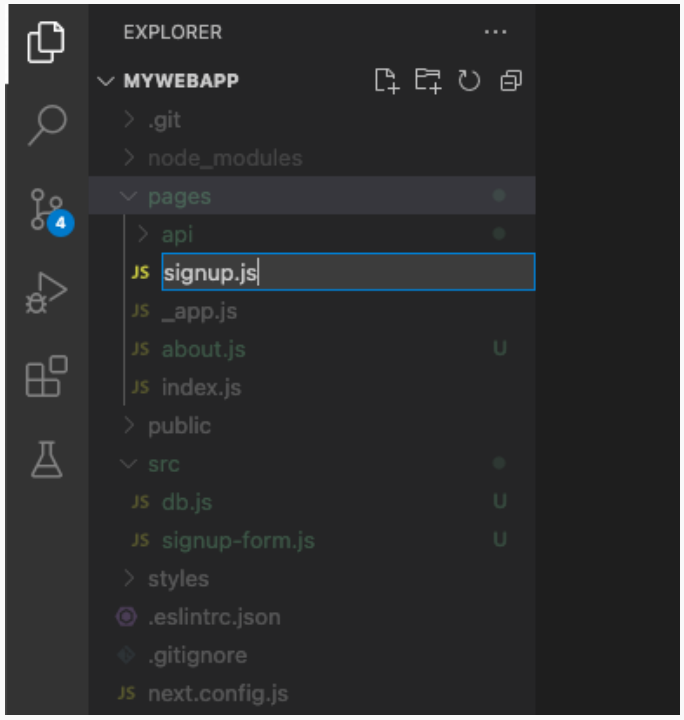
3. Let's modify our **SignupForm()** function using the previous JSX example, just to see how it is implemented:

```
function SignupForm() {
    return <div className="dashboard" style={{width:400}}>
        <h1>Signup</h1>
    </div>;
}

export default SignupForm;
```



With the updated JSX, we wrapped <h1> inside with class name **dashboard** and added an inline style containing **width** set to 400px. Let's leave this and start using our **SignupForm** component.

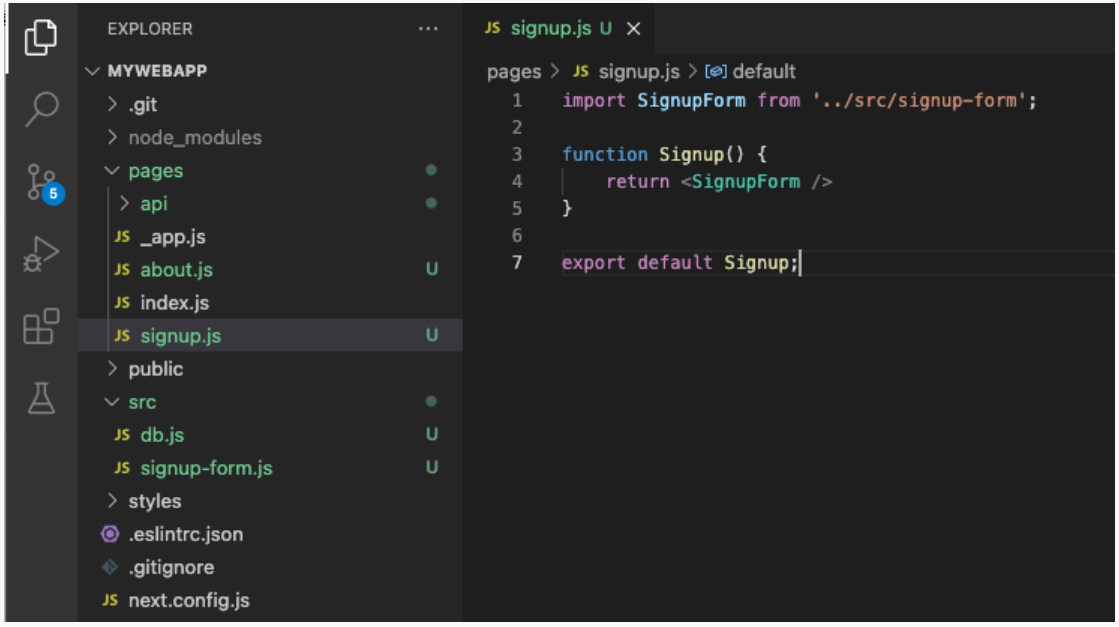
4. Create a new page by adding a new js file in the **pages** directory. Let's name it **signup.js**.



5. Add the following code.

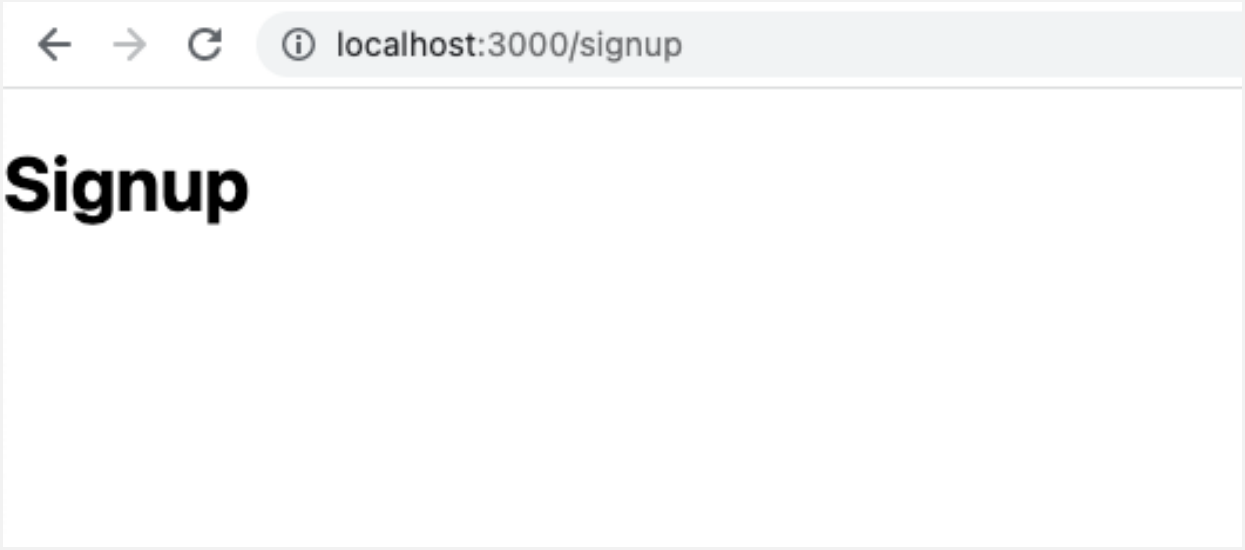
```
import SignupForm from '../src/signup-form';

function Signup() {
    return <SignupForm />
}

export default Signup;
```

First we import the **SignupForm** function from our component **src/signup-form.js**. Then we can render it as a custom element **<SignupForm />**.



6. To see the result, open your browser and goto:

http://localhost:3000/signup



Great! You have created a new page that uses a component in it.

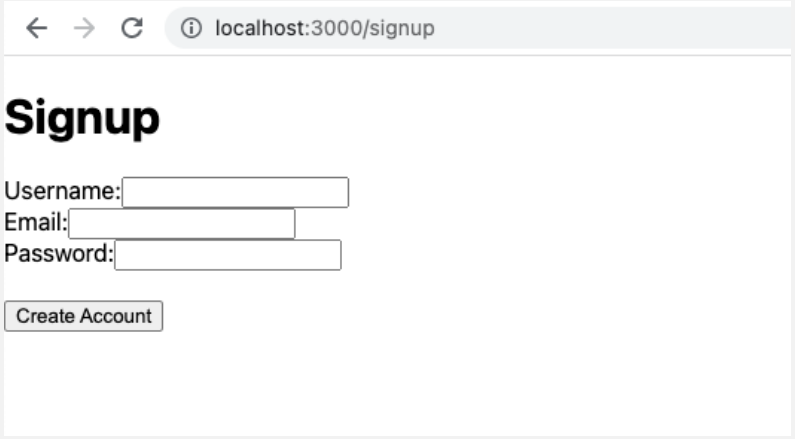
4. Applying Stylesheet

1. Let's add more code to our SignupForm component (**src/signup-form.js**).

```
function SignupForm() {
    return <div className="dashboard" style={{width:400}}>
        <h1>Signup</h1>

        <form>
            <div>
                <label htmlFor="username">Username:</label>
                <input type="text" id="username" />
            </div>
            <div>
                <label htmlFor="email">Email:</label>
                <input type="text" id="email" />
            </div>
            <div>
                <label htmlFor="password">Password:</label>
                <input type="password" id="password" />
            </div>
            <div style={{marginTop:20, marginBottom:20}}>
                <button>Create Account</button>
            </div>
        </form>
    </div>;
}

export default SignupForm;
```

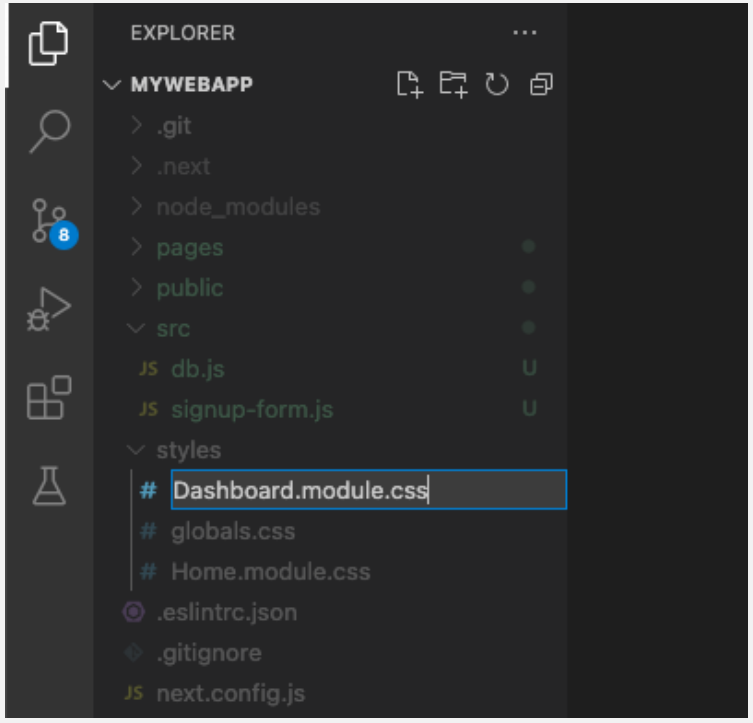
We added a normal form here, written in JSX. It contains text inputs to enter username, email and password, and a **Create Account** button.

6. Check the result: **http://localhost:3000/signup**



The form is displayed, but it doesn't look good.
We need to apply stylesheet here.

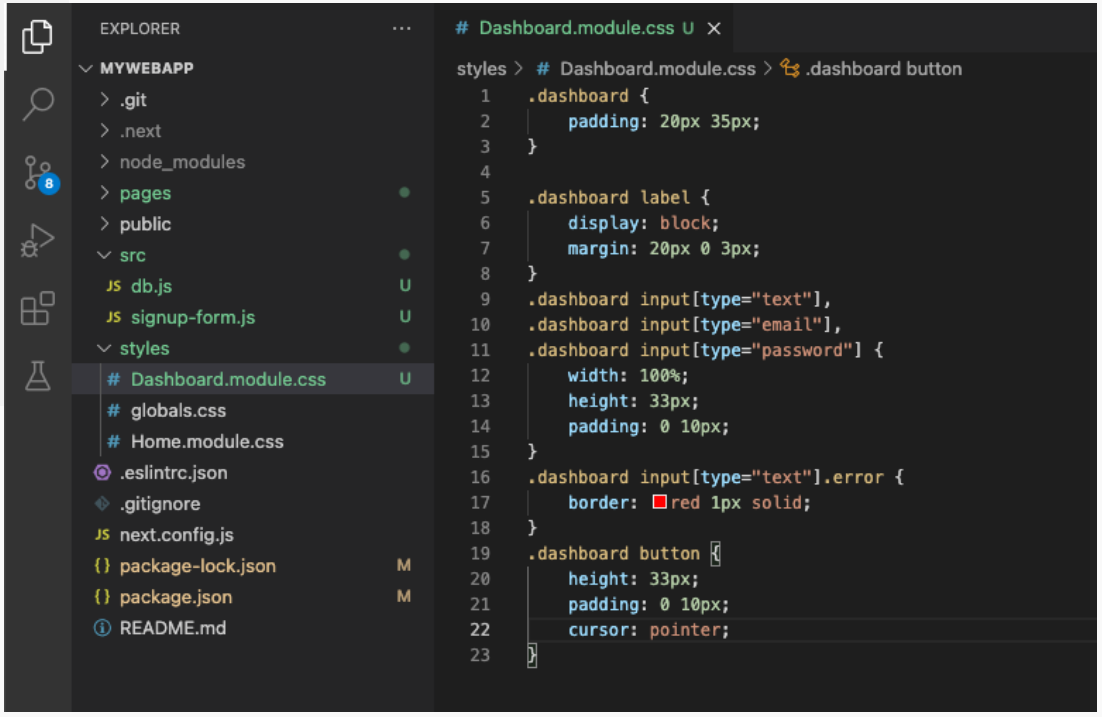
7. Find **styles** folder in your project. Inside this folder, create a new file **Dashboard.module.css**.



8. Add the following styles.

```
.dashboard {
    padding: 20px 35px;
}

.dashboard label {
    display: block;
    margin: 20px 0 3px;
}
.dashboard input[type="text"],
.dashboard input[type="email"],
.dashboard input[type="password"] {
    width: 100%;
    height: 33px;
    padding: 0 10px;
}
.dashboard input[type="text"].error {
    border: red 1px solid;
}
.dashboard button {
    height: 33px;
    padding: 0 10px;
    cursor: pointer;
}
```

Previously we wrapped our Signup form in a with class name **dashboard**. So here we added the class rule **.dashboard** in our css and also other css rules to style the input text and button.

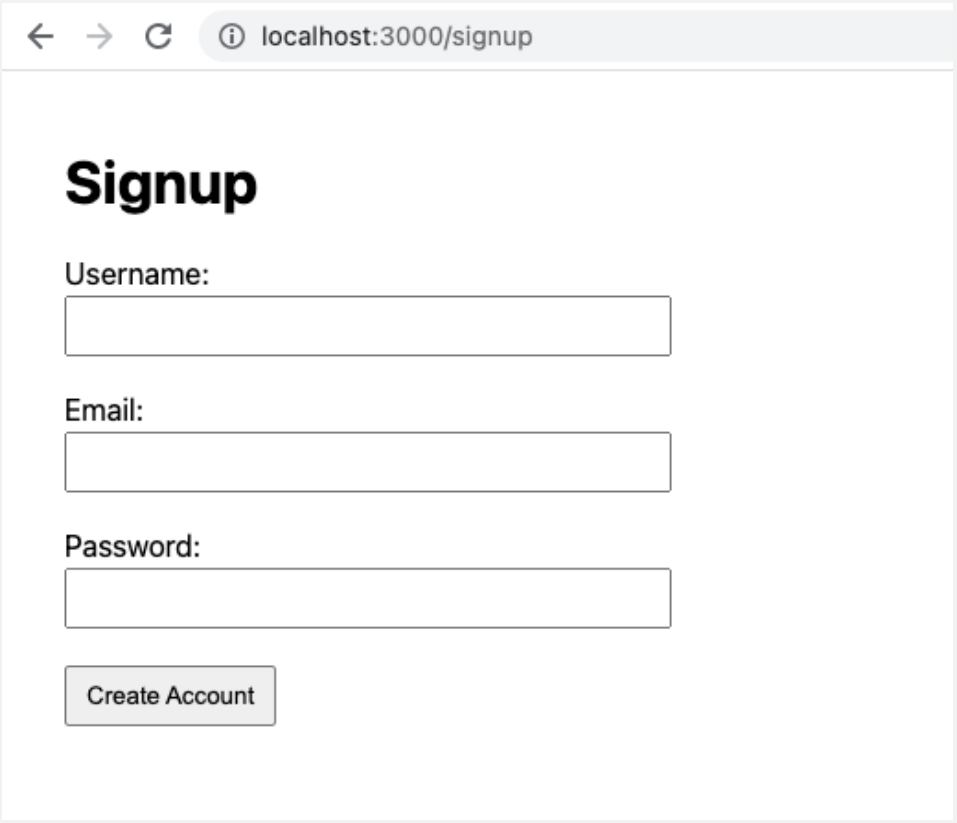


9. We need to import the css file in our component. Open **src/signup-form.js** and update the code.

```
import styles from '../styles/Dashboard.module.css';

function SignupForm() {
    return <div className={styles.dashboard} style={{width:400}}>
        <h1>Signup</h1>

        <form>
            <div>
                <label htmlFor="username">Username:</label>
                <input type="text" id="username" />
            </div>
            <div>
                <label htmlFor="email">Email:</label>
                <input type="text" id="email" />
            </div>
            <div>
                <label htmlFor="password">Password:</label>
                <input type="password" id="password" />
            </div>
            <div style={{marginTop:20, marginBottom:20}}>
                <button>Create Account</button>
            </div>
        </form>
    </div>;
}

export default SignupForm;
```

Once the style is imported, you can get the dashboard classname using: **styles.dashboard**

10. Let's check the result: **http://localhost:3000/signup**



Now that looks better.

5. Working with Form

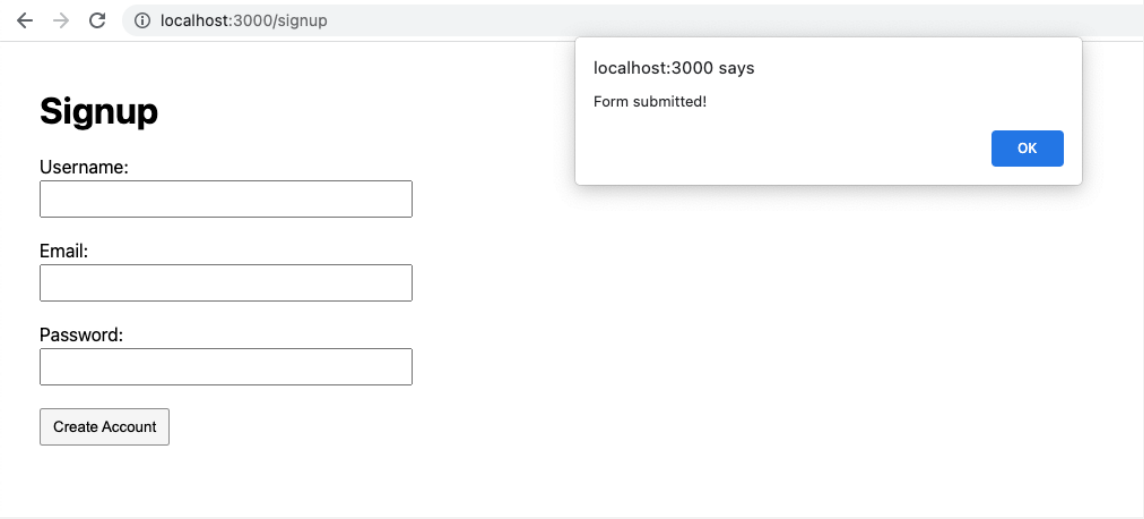
1. Open **src/signup-form.js** and update the code as follows:

```
import styles from '../styles/Dashboard.module.css';

function SignupForm() {

    function submitHandler(event) {
        event.preventDefault();

        alert('Form submitted!');
    }

    return <div className={styles.dashboard} style={{width:400}}>
        <h1>Signup</h1>

        <form onSubmit={submitHandler}>
            ...
            ...
        </form>
    </div>;
}

export default SignupForm;
```

Here we created a function **submitHandler()** that will be called when the form is submitted, that is when the button inside the form is clicked.

As seen, the form **onSubmit** attribute is set with a reference to the function. When onSubmit is triggered it passes an **event** to the function. We run **event.preventDefault()** to prevent the page from reloading. We also run an alert showing the '**Form submitted!**' message.

2. To test, open **http://localhost:3000/signup** and click the **Create Account** button.



An alert will be displayed on the screen with the '**Form Submitted!**' message.

3. Let's add a few more lines to our component **src/signup-form.js**.

```
import { useState } from 'react';
import styles from '../styles/Dashboard.module.css';

function SignupForm() {

    const [message, setMessage] = useState('');

    function submitHandler(event) {
        event.preventDefault();

        setMessage('Form submitted!');
    }

    return <div className={styles.dashboard} style={{width:400}}>
        <h1>Signup</h1>

        <form onSubmit={submitHandler}>
            ...
            ...
            <div>
                {message}
            </div>
        </form>
    </div>;
}

export default SignupForm;
```

We added an import statement:
import { useState } from 'react'

With **useState**, you can have a state variable in your component. Shortly, we'll see what we can do with the state variable.

To create the state variable you'll need to use:
const [state, setState] = useState(initialstate)

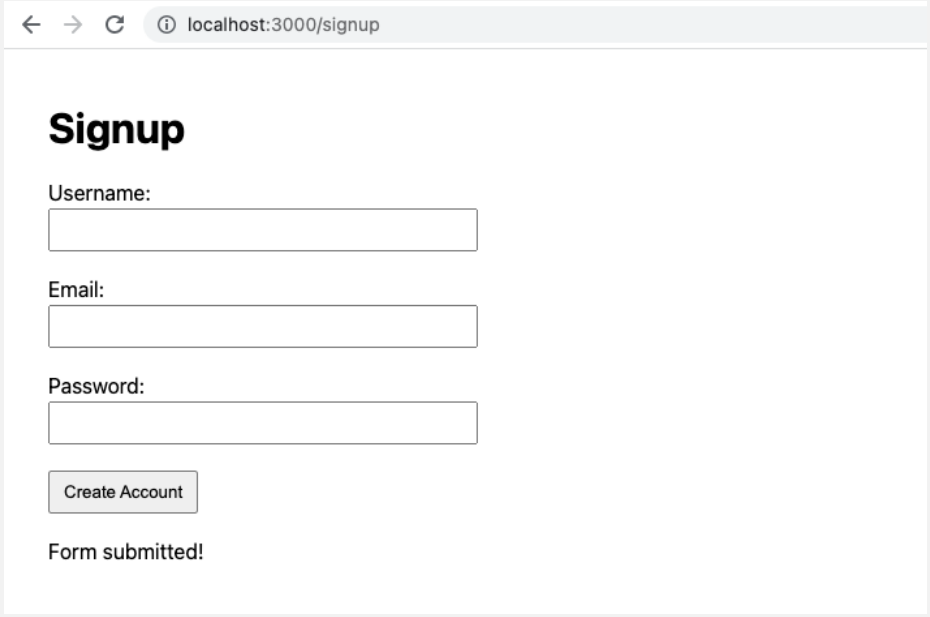
So in our component, we have:
const [message, setMessage] = useState('')
The initial state is set to an empty string here.

Then we replace the alert with:
setMessage('Form Submitted')
Here we change the state from its initial value (empty string) to '**Form Submitted!**' text.

Then you can use the state (**message**) within your JSX. Here we display the message using:

```
<div>
  {message}
</div>
```

4. To test, open **http://localhost:3000/signup** and click the **Create Account** button.



Now the '**Form Submitted!**' message is displayed right on the page (after the button), according to where we place the **message** variable.

5. Now we have learn how a state variable is created and used to store a message. We can also use a state variable to store a text input value. Let's add a few lines for this.

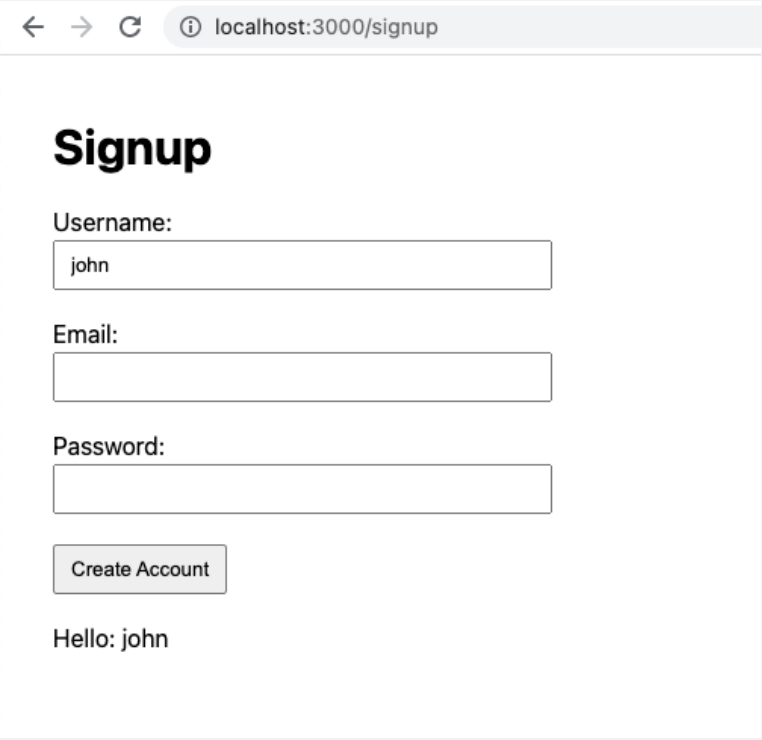
```
import { useState } from 'react';
import styles from '../styles/Dashboard.module.css';

function SignupForm() {

    const [message, setMessage] = useState('');
    const [username, setUsername] = useState('');

    function updateUsername(event) {
        setUsername(event.target.value);
    }

    function submitHandler(event) {
        event.preventDefault();

        setMessage('Hello: ' + username);
    }

    return <div className={styles.dashboard} style={{width:400}}>
        <h1>Signup</h1>

        <form onSubmit={submitHandler}>
            <div>
                <label htmlFor="username">Username:</label>
                <input type="text" id="username" onChange={updateUsername} />
            </div>
            ...
            <div>
                {message}
            </div>
        </form>
    </div>;
}

export default SignupForm;
```

We added another state variable named **username** that will always update everytime we change the <input> value.

To read the input value, we use:
event.target.value

Then we display the **username** as part of the message.

6. To test, open **http://localhost:3000/signup** , enter a username and click the **Create Account** button.



Now the username value is displayed.

We will add another state variable for the other inputs, i.e. for email and password. With state variables applied, it is easy to get the text input values from our form for further processing.

7. Let's update our code with the following.

```
import { useState } from 'react';
import styles from '../styles/Dashboard.module.css';

function SignupForm() {

    const [message, setMessage] = useState('');
    const [username, setUsername] = useState('');
    const [email, setEmail] = useState('');
    const [password, setPassword] = useState('');

    function submitHandler(event) {
        event.preventDefault();

        setMessage('Hello: ' + username);
    }

    return <div className={styles.dashboard} style={{width:400}}>
        <h1>Signup</h1>

        <form onSubmit={submitHandler}>
            <div>
                <label htmlFor="username">Username:</label>
                <input type="text" id="username" required onChange={(e)=>setUsername(e.target.value)} />
            </div>
            <div>
                <label htmlFor="email">Email:</label>
                <input type="text" id="email" required onChange={(e)=>setEmail(e.target.value)} />
            </div>
            <div>
                <label htmlFor="password">Password:</label>
                <input type="password" id="password" required onChange={(e)=>setPassword(e.target.value)} />
            </div>
            <div style={{marginTop:20, marginBottom:20}}>
                <button>Create Account</button>
            </div>
            <div>
                {message}
            </div>
        </form>
    </div>;
}

export default SignupForm;
```

Here we modified the **onChange** event attribute to shorten our code. We also added the **required** attribute to the input field to protect from an empty value.

6. Posting Form Data

We still need to save user input data to our database. To perform this task, the data (username, email and password) has to be posted to the server. An API route will be prepared to receive the posted data.

To post the data we will use the Javascript **fetch** method.

1. Open **src/signup-form.js** and add the following **createUser()** function.

```
async function createUser(username, email, password) {
    const reqBody = { username, email, password };
    let result = await fetch('/api/auth/signup', {
        method:'POST',
        body: JSON.stringify(reqBody),
        header: {
            'Content-Type': 'application/json'
        }
    });
    result = await result.json();
    return result;
}
```

The **createUser()** function accepts our data (username, email and password).

The **fetch()** method accepts 2 parameters:

- URL
- options: an array of properties (including the **body** property for passing our data).

For the URL we specify: **/api/auth/signup**. This is an API route that we will create later to receive the posted data.
The posted data has to be a string, so we use **JSON.stringify()** to convert our data into string.

2. Update the **submitHandler()** function.

```
function submitHandler(event) {
    event.preventDefault();

    createUser(username, email, password);

    setMessage('User created successfully.');
}
```

Here we add the line to call the **createUser()** function.

Next we'll prepare an API route to receive the posted data and save it to our database.

7. Saving Form Data

Before creating an API route to receive the form data, we'll need to first create a utility to manage the password data. We cannot save password into our database as plain text. We need to convert the password into another string, called hashed password. For this task we will use the **bcryptjs** module.

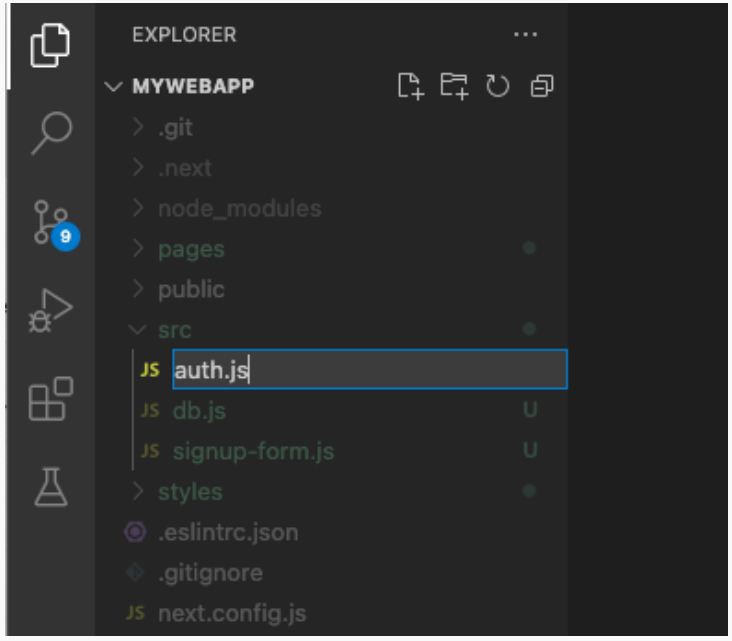
1. Open terminal, go to your project folder and install the **bcryptjs** module.

```
$ cd mywebapp
```

```
$ npm i bcryptjs
```

With the **bcryptjs** module, we can convert the password into another string, called hashed password, so that we don't store the password in our database as plain text.

2. Create a new file **src/auth.js**.



3. Add the following code into **src/auth.js**.

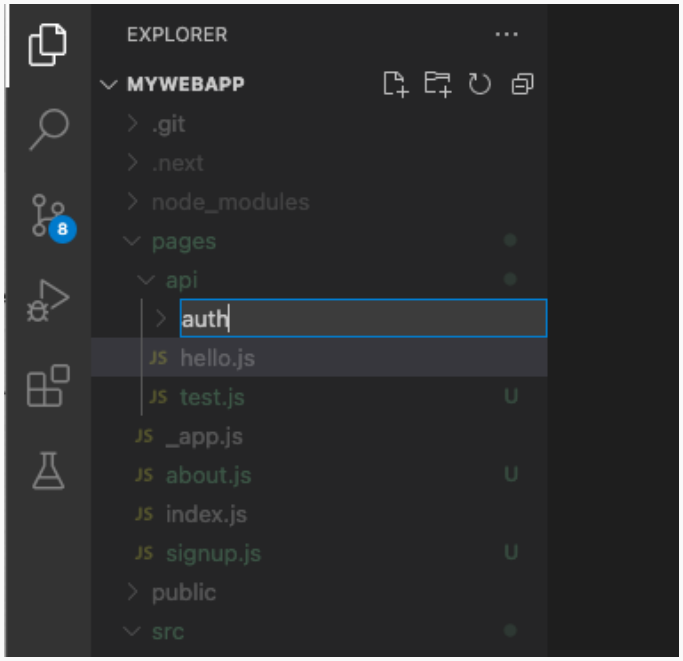
```
import { hash, compare } from 'bcryptjs';

export async function hashPassword(password) {
    const hashedPassword = await hash(password, 12);
    return hashedPassword;
}

export async function verifyPassword(password, hashedPassword) {
    const isValid = await compare(password, hashedPassword);
    return isValid;
}
```

The **hashPassword()** function converts a plain text password into a hashed password.

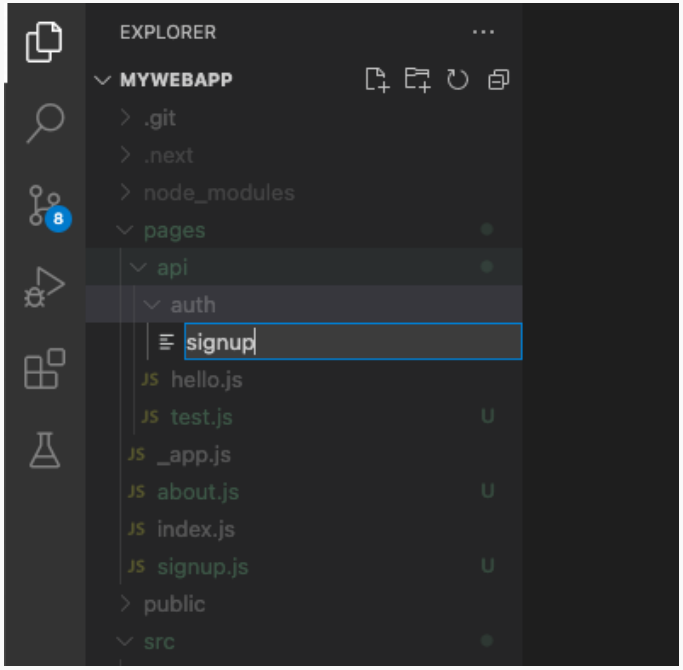
The **verifyPassword()** function compares a plain text password with its corresponding hashed password. It returns true (for matched) or false (for unmatched).

Our utility for hashing passwords is ready. Next we'll start creating the API route to receive data and save it to our database.

4. Create a new folder **pages/api/auth**.



5. Create a new file **signup.js** inside the folder **pages/api/auth**.



Now we start working with our new API route **pages/api/auth/signup.js**.

6. Add the required imports, including one from our hashed password utility we have created.

```
import { connectDatabase } from '../../../src/db';
import { hashPassword } from '../../../src/auth';
```

7. Prepare the handler function. We will add more code inside this handler.

```
async function handler(req, res) {

}

export default handler;
```

8. Add a checking to accept POST requests only. Then start parsing the received data using **JSON.parse()**.

```
    if(req.method !== 'POST') return;

    const data = JSON.parse(req.body)
    const username = data.username;
    const email = data.email;
    const password = data.password;
```

9. Check if username, email and password are not empty and also check for valid email and password.

```
    if(!username ||
        !email || !email.includes('@') ||
        !password || password.trim().length < 7) {
        res.status(200).json({ ok:true, status: 422, error: 'Invalid input - password should be at least 7 characters
long.' });
        return;
    }
```

If there is an invalid input, we return a JSON data to the client with some error details. The status 200 means success (other status, for example, 404 means not found). But here, regardless of error, we send status 200 and return the actual error code (422) in the JSON response. 422 means unprocessable data (because of invalid input). By sending status 200 to the client, the browser won't raise an application error.

10. Start the database connection.

```
    let client;
    try {
        client = await connectDatabase();
    } catch (error) {
        res.status(200).json({ ok:true, status: 500, error: 'Connecting to the database failed.' });
        return;
    }
```

11. Now we start accessing our database. Our database will have a **users** collection to store user accounts. Initially, the **users** collection does not exist. It will be created automatically when we insert our first user data.

Before inserting a user data, we need to check first if username and email already exist.

```
    try {
        const db = client.db();

        const existingUser = await db.collection('users').findOne({
            username: username
        });
        const existingEmail = await db.collection('users').findOne({
            email: email
        });
        if(existingUser) {
            res.status(200).json({ ok:true, status: 422, error: 'Username already used.' });
            client.close();
            return;
        }
        if(existingEmail) {
            res.status(200).json({ ok:true, status: 422, error: 'Email already registered.' });
            client.close();
            return;
        }

        // to be continued here...

    } catch (error) {
        res.status(200).json({ ok:true, status: 500, error: 'Unable to create user.' });
        client.close();
    }
```

Here we use the **findOne()** method on the **users** collection to check if a username or email exists.

12. We hash the password and then insert our user data into the **users** collection

```
        const hashedPassword = await hashPassword(password);

        await db.collection('users').insertOne({
            username: username,
            email: email,
            password: hashedPassword
        });
```

Later in our application we also need a **pages** collection, where users can save their pages. After inserting a user, it would be better to automatically insert an initial page for the user. It will become the user's home page.

So, here we insert the user's initial home page into the **pages** collection.

```
        await db.collection('pages').insertOne({
            title: 'Home - ' + username,
            desc: '',
            slug: '',
            html: '',
            username: username,
        });
```

Each page will have:

- title
- desc (description)
- slug (part of a URL that identifies a particular page, similar to filename). For home page, we set it to empty.
- html (the page content)
- username (the owner)

13. Lastly, we send a successful response to the client.

```
        res.status(200).json({ ok:true, status: 201 });
        client.close();
```

Here is the complete code for **pages/api/auth/signup.js**.

```
import { connectDatabase } from '../../../src/db';
import { hashPassword } from '../../../src/auth';

async function handler(req, res) {

    if(req.method !== 'POST') return;

    const data = JSON.parse(req.body)
    const username = data.username;
    const email = data.email;
    const password = data.password;

    if(!username ||
        !email || !email.includes('@') ||
        !password || password.trim().length < 7) {
        res.status(200).json({ ok:true, status: 422, error: 'Invalid input - password should be at least 7 characters long.' });
        return;
    }

    let client;
    try {
        client = await connectDatabase();
    } catch (error) {
        res.status(200).json({ ok:true, status: 500, error: 'Connecting to the database failed.' });
        return;
    }

    try {
        const db = client.db();

        const existingUser = await db.collection('users').findOne({
            username: username
        });
        const existingEmail = await db.collection('users').findOne({
            email: email
        });
        if(existingUser) {
            res.status(200).json({ ok:true, status: 422, error: 'Username already used.' });
            client.close();
            return;
        }
        if(existingEmail) {
            res.status(200).json({ ok:true, status: 422, error: 'Email already registered.' });
            client.close();
            return;
        }
```

```
        const hashedPassword = await hashPassword(password);

        await db.collection('users').insertOne({
            username: username,
            email: email,
            password: hashedPassword
        });

        await db.collection('pages').insertOne({
            title: 'Home - ' + username,
            desc: '',
            slug: '',
            html: '',
            username: username,
        });

        res.status(200).json({ ok:true, status: 201 });
        client.close();

    } catch (error) {
        res.status(200).json({ ok:true, status: 500, error: 'Unable to create user.' });
        client.close();
    }

}

export default handler;
```

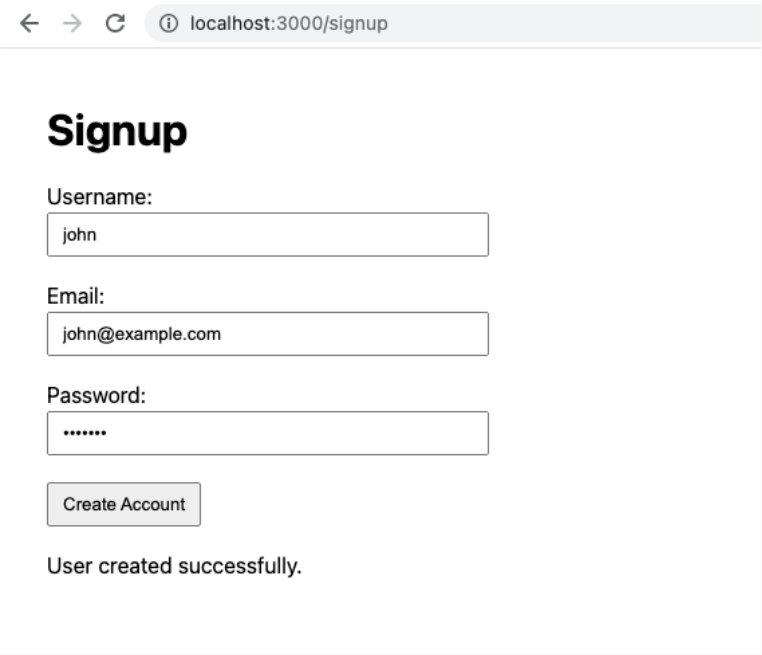
The API route URL will be: **/api/auth/signup**, as specified in the **fetch()** method in our signup form.

Now on the client side, we can display a message after user signup according to the response returned by the server.

14. Open **src/signup-form.js** and update the **submitHandler()** function with the following code.

```
async function submitHandler(event) {
    event.preventDefault();

    try {
        const result = await createUser(username, email, password);
        if(!result.error) {
            setMessage('User created successfully.');
        } else {
            setMessage(result.error);
        }
    } catch(error) {
        setMessage('Something went wrong!');
    }
}
```

15. To test, open **http://localhost:3000/signup** and try creating an account.



16. Open terminal and connect to your server as the user **hobnob**.

```
$ ssh hobnob@<server-ip>
```

17. Enter the MongoDB shell.

```
$ mongosh -u hobnob -p --authenticationDatabase admin
```

18. Change database to **nextsite**. Remember that we named our database **nextsite** in the **.env.development** and **.env.production** configuration files.

```
use nextsite
```

19. Display the **users** collection.

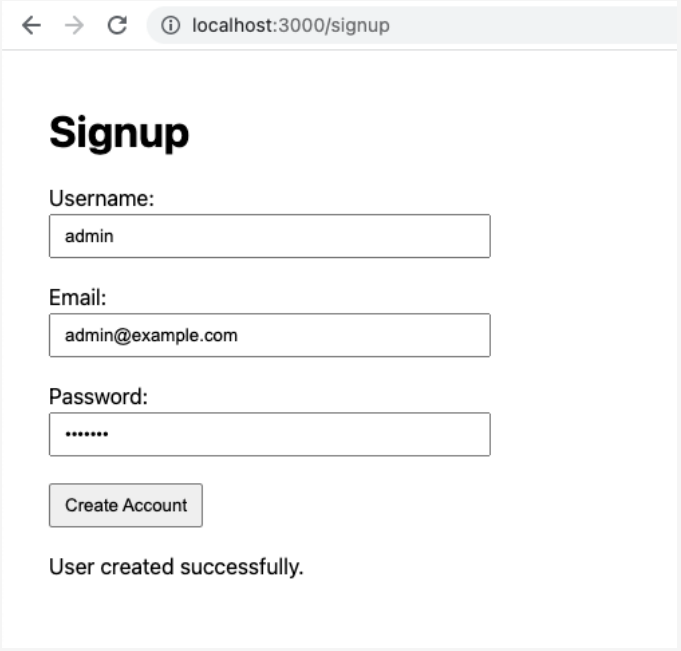
```
db.users.find()
[
 {
 _id: ObjectId("6204cc12da201f1a329e6585"),
 username: 'john',
 email: 'john@example.com',
 password: '$2a$12$cJM/zD.....'
 }
]
```

The user we've just created will be displayed.

20. To delete the user, use:

```
db.users.deleteOne(
   { "username": "john" }
)
```

21. Now, signup for a new account with username **admin**. We will use it in the next chapter.



8. Adding User Login

To add user login or athentication, we will use NextAuth.js library. NextAuth.js library has a wide range of providers you can use, for example, you can sign-in with Google, Facebook or Apple.

In this chapter we will build our own authentication from stratch using email and password combination with the help of the NextAuth.js library.

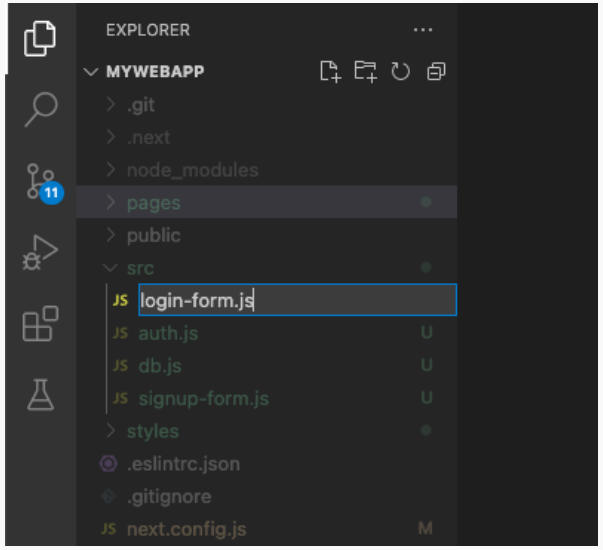
1. Open terminal and go to our project folder.

```
$ cd mywebapp
```

2. Install the **NextAuth.js** package.

```
$ npm i next-auth
```

3. Create a new file **src/login-form.js**.



Now we start working with our new login component **src/login-form.js**.

4. Import **signIn** from the NextAuth.js, **useState** from React and **styles** from our css.

```
import { signIn } from 'next-auth/react';
import { useState } from 'react';
import styles from '../styles/Dashboard.module.css';
```

5. Add the following codes.

```
function LoginForm() {

    const [message, setMessage] = useState('');
    const [email, setEmail] = useState('');
    const [password, setPassword] = useState('');

    async function submitHandler(event) {
        event.preventDefault();

    }

    return (
        <div className={styles.dashboard} style={{maxWidth:400}}>
            <h1>Login</h1>
            <form onSubmit={submitHandler}>
                <div>
                    <label htmlFor="email">Your Email</label>
                    <input type="email" id="email" required onChange={(e)=>setEmail(e.target.value)} />
                </div>
                <div>
                    <label htmlFor="password">Your Password</label>
                    <input type="password" id="password" required onChange={(e)=>setPassword(e.target.value)} />
                </div>
                <div style={{marginTop:20, marginBottom:20}}>
                    <button>Login</button>
                </div>
                <div>
                    {message}
                </div>
            </form>
        </div>
    )
}

export default LoginForm;
```

This is similar to the signin form we have created. We only need two inputs here, email and password.

6. Now the important part is the login process. Replace the **submitHandler()** function with the following code.

```
async function submitHandler(event) {
    event.preventDefault();

    const result = await signIn('credentials', {
        redirect: false,
        email: email,
        password: password
    });
    if(!result.error) {
        setMessage('You are successfully logged in.');
    } else {
        setMessage(result.error);
    }
}
```

The login process is started by calling the **signIn()** method provided by the NextAuth.js library. Next, we will prepare the server to process this request.

7. We need to specify configuration variables required by NextAuth.js.

Open **.env.development** and add the following lines.

```
NEXTAUTH_URL=http://localhost:3000
NEXTAUTH_SECRET=INp8IvdIyeMcoGAgFGoA61DdBglwwSqnXJZkgz8PSnw
```

Then open **.env.production** and add the following lines.

```
NEXTAUTH_URL=https://sitemagz.com
NEXTAUTH_SECRET=INp8IvdIyeMcoGAgFGoA61DdBglwwSqnXJZkgz8PSnw
```

We specify **NEXTAUTH_URL** and **NEXTAUTH_SECRET** in **.env.development** for development and **.env.production** for production/deployment.

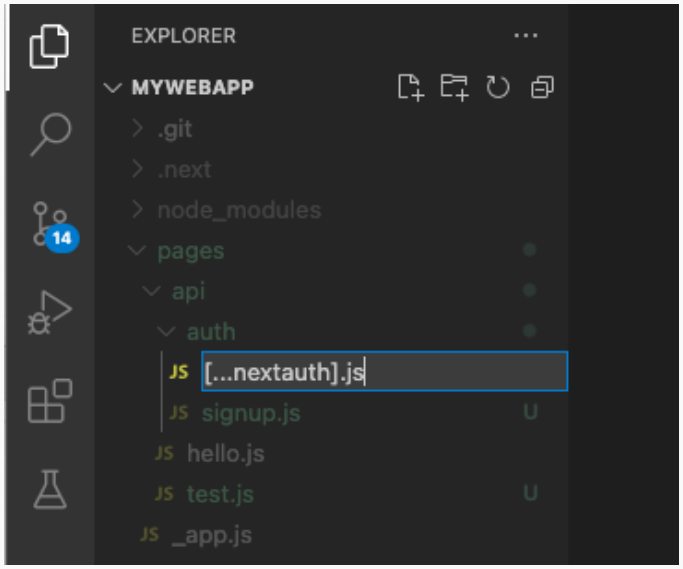
You can generate the secret code for **NEXTAUTH_SECRET** using:

https://generate-secret.now.sh/32

8. Open terminal and rerun the project.

```
$ npm run dev
```

9. Create a new file **pages/api/auth/[...nextauth].js**.



The filename **[...nextauth].js** has a special behavior in Next.js application. It will catch all requests to **api/auth/***. This is needed by the NextAuth.js library to handle the authentication requests.

10. Add the following code on the **pages/api/auth/[...nextauth].js**.

```
import { connectDatabase } from '../../../src/db';
import { verifyPassword } from '../../../src/auth';
import NextAuth from 'next-auth';
import CredentialsProvider from 'next-auth/providers/credentials'

export default NextAuth({
    secret: process.env.NEXTAUTH_SECRET,
    session: {
        jwt: true,
        maxAge: 30 * 24 * 60 * 60, // 30 days
    },
    providers: [
        CredentialsProvider({
            async authorize(credentials) {

                // to be continued here

            }
        })
    ],
});
```

The NextAuth handler contains some configurations needed for authentication purposes. Here we use the JWT (stands for JSON Web Token) method with a user session that should last for 30 days (**maxAge**). For the **secret**, we get our previously defined **NEXTAUTH_SECRET** using **process.env.NEXTAUTH_SECRET**.

We will continue with the **authorize()** part which contains the process to authenticate user.

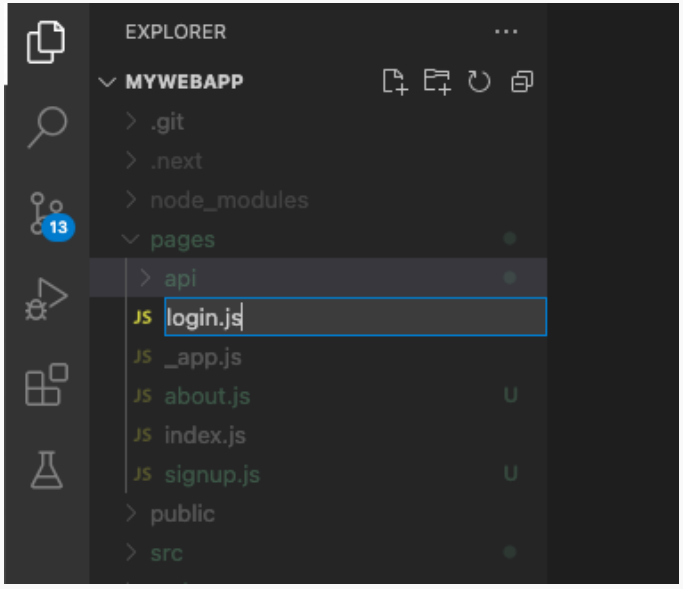
11. Add the following codes inside the **authorize()**.

```
let client;
try {
    client = await connectDatabase();
} catch (error) {
    throw new Error('Connecting to the database failed.');
}

const db = client.db();

const user = await db.collection('users').findOne({email: credentials.email});
if(!user) {
    // User not found
    client.close();
    throw new Error(`Sorry, we couldn't log you in. Please check that you have entered the correct account details.`);
}

const isValid = await verifyPassword(credentials.password, user.password);
if(!isValid) {
    // Incorrect password
    client.close();
    throw new Error('Oops! The password you entered is incorrect. Please try again.');
}

client.close();

const userData = { name: user.username, email: user.email };
return userData;
```

First we check if the user email exists in our **users** collection. Then we use **verifyPassword** (from our hash password utility) to check user entered password with the hashed password from the database.
Valid login will then returns user JSON data.

So far we have a login component (**src/login-form.js**) and and an API route (**pages/api/aut/[...nextauth].js**) to process the authentication. Now we will create a page to use the component.

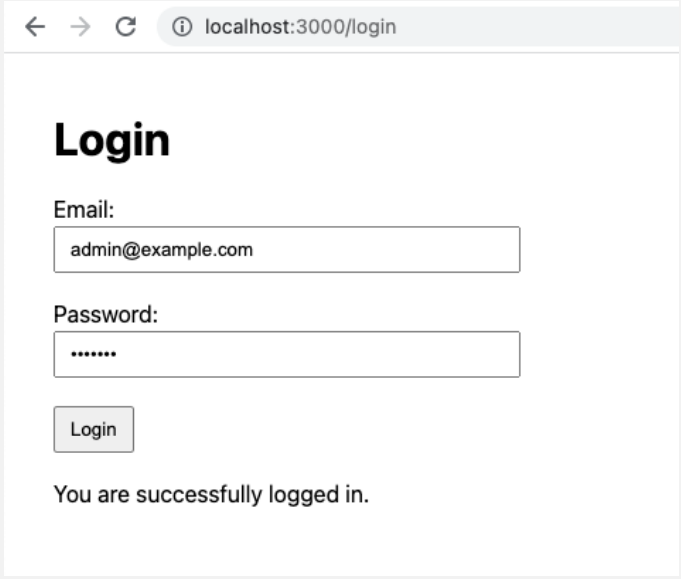
12. Create a new file **pages/login.js**.



13. Add the following codes.

```
import LoginForm from '../src/login-form';

function Login() {
    return <LoginForm />
}

export default Login;
```

14. Open your browser and go to: **http://localhost:3000/login**.
Try login with admin account you have created previously.



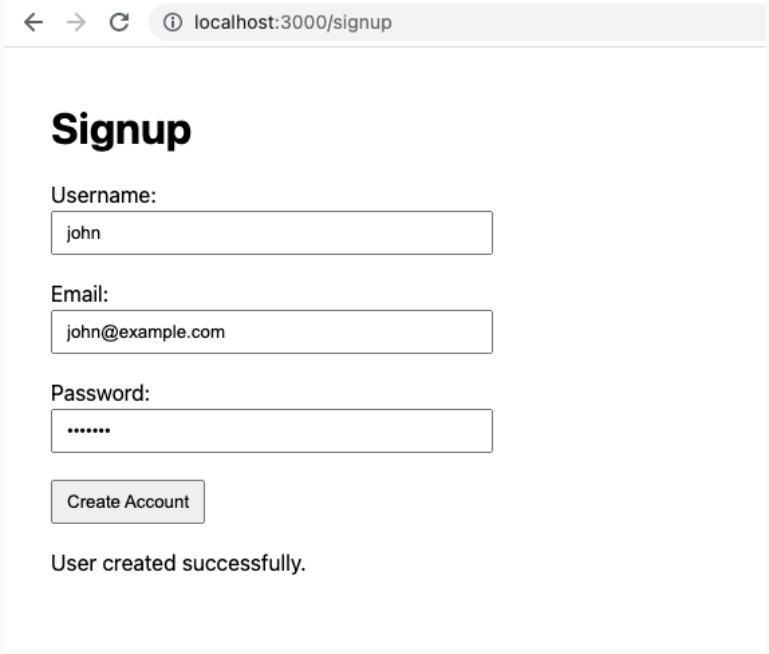
Congratulations! Now your website is ready to accept user signup. You also have a login page for you or your website users.

So far you have learned the following:

- Working with MongoDB Database
- Creating React Component (Client)
- Creating API Route (Server)
- Using NextAuth.js for Authentication

9. Displaying User Page

1. Let's create a new account. In this example, we use **john** as the username.



So far, we have 2 users: **admin** and **john**. **admin** will be the one who own the main site, and **john** is the user. The main site accepts multiple user registrations and each user will be able to create their own site. Later, users can assign a custom domain name for the their website.

Before using custom domain, user sites can be accessed from the main site. In this guide, our main site is **sitemagz.com**. So we will use:

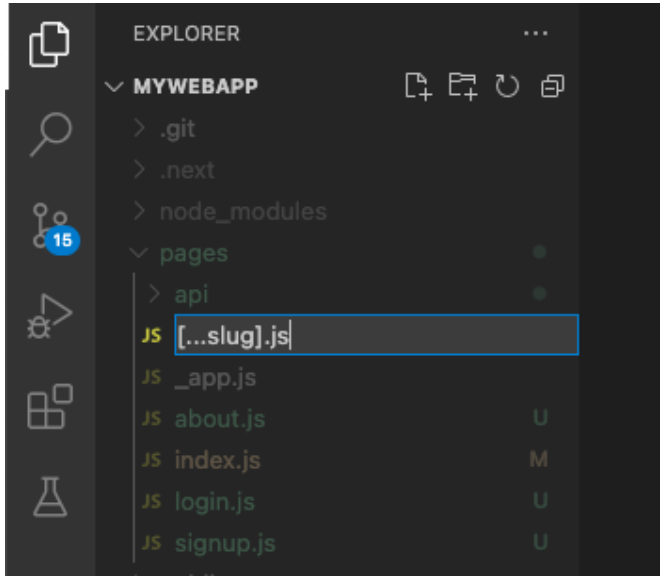
http://sitemags.com/site/john -> For user site

http://sitemagz.com -> For the main site

When the user admin and john are created, a home page is created for each of them, stored in **pages** collection. Now we want to display the user page whenever the above URL is requested.

For this purpose, we need to create a page that catch all request and read the URL to see whose site is being requested.

2. Create a new file **pages/[...slug].js**. This will be the page that catch all the requests.



3. Add the following code.

```
import { connectDatabase } from '../src/db';
import Head from 'next/head';

function Home(props) {

    // to be continued here

}

export async function getServerSideProps(context) {

    // to be continued here

}

export default Home;
```

The **getServerSideProps()** function will contain the process to read the requested URL and find the page data from our database. The **Home()** function will do the page rendering. We also import **Head** since we will render the head section later in our page.

4. Add this code inside the **getServerSideProps()**.

```
//Reading requested URL: yourdomain.com/part1/part2/part3/part4
let part1, part2, part3, part4;
if(context.query.slug) [part1, part2, part3, part4]  = context.query.slug;

let siteOwner;
let slug = '';
let invalidRequest = false;
if(!part1) {
  // Example request: yourdomain.com => requesting home page (owner: admin)
  siteOwner = 'admin';
} else if (part1==='site' && part2 && !part3) {
  // Example request: yourdomain.com/site/john => requesting home page (owner: john)
  siteOwner = part2;
} else if(part1 && !part2) {
  // Example request: yourdomain.com/contact => requesting contact page (owner: admin)
  siteOwner = 'admin';
  slug = part1;
} else if(part1==='site' && part2 && part3) {
  // Example request: yourdomain.com/site/john/contact => requesting contact page (owner: john)
  siteOwner = part2;
  slug = part3;
  if(part4) {
    // Invalid Request (only part1, part2 & part3 is valid for user pages)
    invalidRequest = true;
  }
}
```

Here we read the requested URL using **context.query.slug** and assign each part of the path into parts (**part1**, **part2**, **part3** and **part4**).

For example, if someone requests:

https://sitemagz.com/site/john, then **part1** = **site**, and **part2** = **john** (site owner).

https://sitemagz.com/site/john/contact, then **part1** = **site**, **part2** = **john** (site owner), and **part3** = **contact** (page slug)

https://sitemagz.com/contact, then **part1** = **contact** (page slug). Site owner is **admin** (since it is a main site).

https://sitemagz.com/, then **part1** = empty (the home page is requested). Site owner is **admin**.

So we know whose site and which page is being requested. For the moment, we only have a single page (the home page) for the registered users. So we will only test opening the home page.

Valid request for user sites is only up to **part3**. If there is **part4** then it is an invalid request.

5. Once we know the site owner, then we can query the **pages** collection to get the page (based on page slug).

```
let client = await connectDatabase();
const db = client.db();

let site = await db.collection('users').findOne({
  username: siteOwner
});
if(!site) {
  return {
    redirect: { permanent: false, destination: '/notfound' }, props:{},
  };
}

let page;
if(siteOwner) {
  // Get the requested home page
  if(!invalidRequest) {
    page = await db.collection('pages').findOne({
      username: siteOwner,
      slug: slug
    });
  }

  if(!page && slug!=='notfound') {
    // Redirect (needs to create a page with slug=notfound, otherwise, use default 404 message)
    return {
      redirect: { permanent: false, destination: 'notfound' }, props:{},
    };
  }
} else {
  return {
    redirect: { permanent: false, destination: '/notfound' }, props:{},
  };
}

client.close();

return {
  props: {
    siteOwner: siteOwner?siteOwner:'',
    slug: slug,
    title: page?page.title:'',
    desc: page?page.desc:'',
    html: page?page.html:'',
    notFound: page?false:true,
   }
};
```

The result will be returned as a JSON data.
We added the **notFound** property in case the requested page is not found.

Now we move to the **Home()** function. The **Home()** function receives the returned JSON data from the **getServerSideProps()** in **props** parameter.

6. Add this code inside the **Home()** function.

```
if(props.notFound) {
    return <>
        <Head>
            <title>404</title>
            <link rel="icon" href="/favicon.ico" />
        </Head>
        <h1>404</h1>
        <p>Oops!</p>
    </>
} else {
    return <>
        <Head>
            <title>{props.title}</title>
            <link rel="icon" href="/favicon.ico" />
        </Head>

        <p>Welcome!</p>
    </>
}
```

We check if **props.notFound** is true, the it returns a 404 page. If page exists, it returns a simple welcome page with the page title displayed.

Here is the complete code for **pages/[...slug].js**.

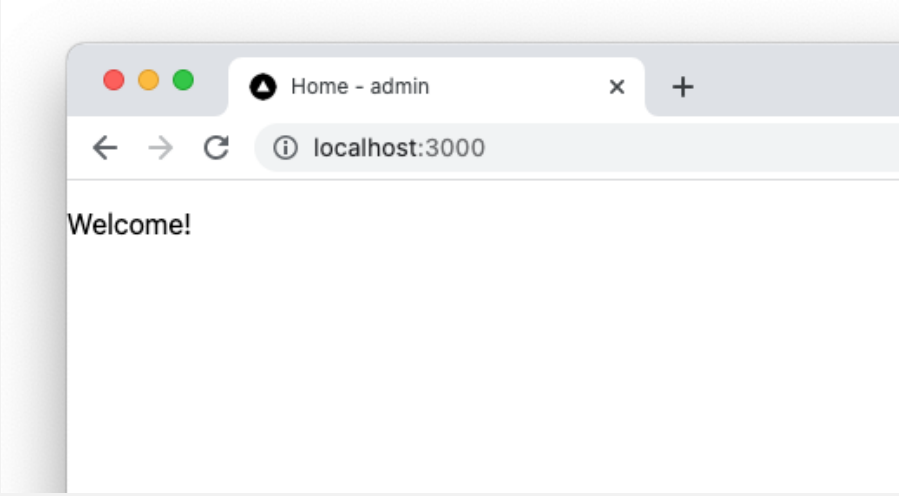
```
import { connectDatabase } from '../src/db';
import Head from 'next/head';

function Home(props) {
    if(props.notFound) {
        return <>
            <Head>
                <title>404</title>
                <link rel="icon" href="/favicon.ico" />
            </Head>
            <h1>404</h1>
            <p>Oops!</p>
        </>
    } else {
        return <>
            <Head>
                <title>{props.title}</title>
                <link rel="icon" href="/favicon.ico" />
            </Head>
            <p>Welcome!</p>
        </>
    }
}

export async function getServerSideProps(context) {

  //Reading requested URL: yourdomain.com/part1/part2/part3/part4
  let part1, part2, part3, part4;
  if(context.query.slug) [part1, part2, part3, part4]  = context.query.slug;

  let siteOwner;
  let slug = '';
  let invalidRequest = false;
  if(!part1) {
    // Example request: yourdomain.com => requesting home page (owner: admin)
    siteOwner = 'admin';
```

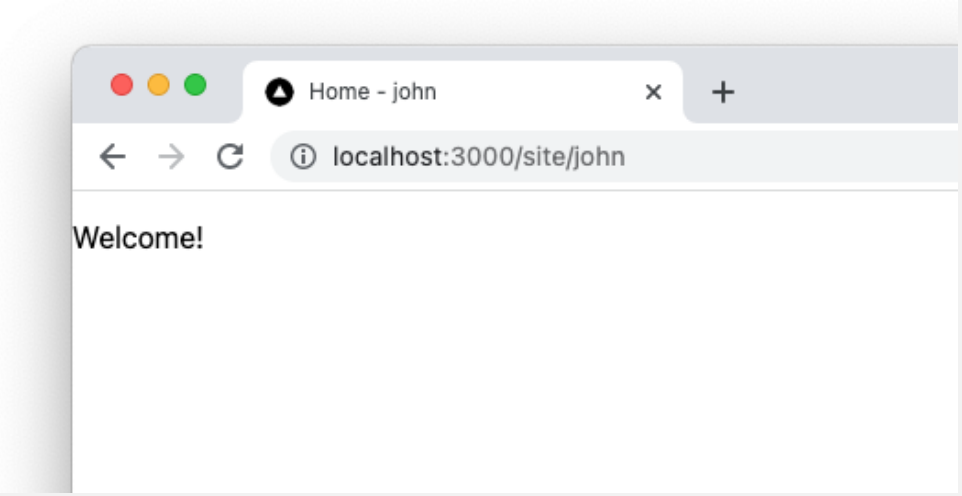
```
} else if (part1==='site' && part2 && !part3) {
  // Example request: yourdomain.com/site/john => requesting home page (owner: john)
  siteOwner = part2;
} else if(part1 && !part2) {
  // Example request: yourdomain.com/contact => requesting contact page (owner: admin)
  siteOwner = 'admin';
  slug = part1;
} else if(part1==='site' && part2 && part3) {
  // Example request: yourdomain.com/site/john/contact => requesting contact page (owner: john)
  siteOwner = part2;
  slug = part3;
  if(part4) {
    // Invalid Request (only part1, part2 & part3 is valid for user pages)
    invalidRequest = true;
  }
}

let client = await connectDatabase();
const db = client.db();

let site = await db.collection('users').findOne({
  username: siteOwner
});
if(!site) {
  return {
    redirect: { permanent: false, destination: '/notfound' }, props:{},
  };
}

let page;
if(siteOwner) {
  // Get the requested home page
  if(!invalidRequest) {
    page = await db.collection('pages').findOne({
      username: siteOwner,
      slug: slug
    });
  }
```

```
    if(!page && slug!=='notfound') {
      // Redirect (needs to create a page with slug=notfound, otherwise, use default 404 message)
      return {
        redirect: { permanent: false, destination: 'notfound' }, props:{},
      };
    }
  } else {
    return {
      redirect: { permanent: false, destination: '/notfound' }, props:{},
    };
  }

  client.close();

  return {
    props: {
      siteOwner: siteOwner?siteOwner:'',
      slug: slug,
      title: page?page.title:'',
      desc: page?page.desc:'',
      html: page?page.html:'',
      notFound: page?false:true,
     }
  };
}

export default Home;
```

7. Important: the **pages/[...slug].js**. is actually not able to catch the root path **/** request. The root path **/** can only be catched by **pages/index.js**. So we need to copy all the code from **pages/[...slug].js** and paste it to **pages/index.js**. That's all needed. Both page will have the same code.

8. To test, open from your browser: **http://localhost:3000** and **http://localhost:3000/site/john**.

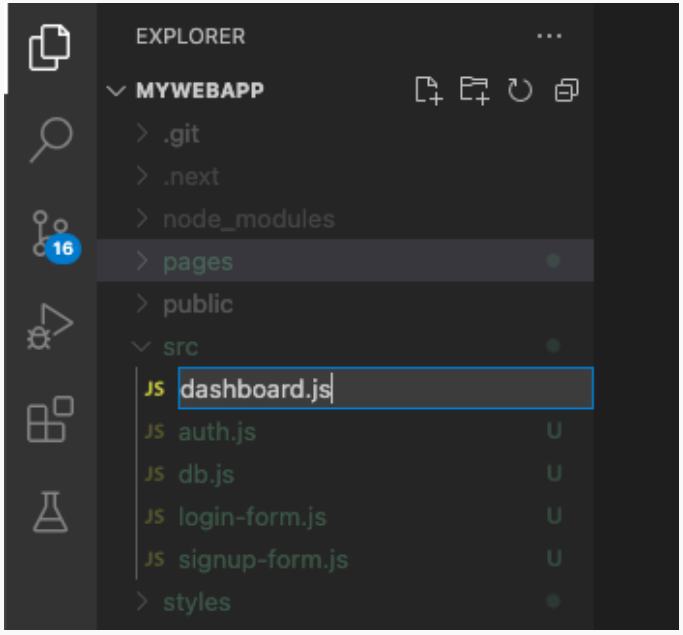




If you push your local project to Github, you can try it online using: **https://sitemagz.com** and **https://sitemagz.com/site/john**.

10. Adding User Dashboard

1. Create a new file **src/dashboard.js**. This will be our dashboard component.



2. Add the following code.

```
import { useEffect, useState } from 'react';
import { useSession, signOut } from 'next-auth/react';
import Link from 'next/link';
import styles from '../styles/Dashboard.module.css';

function Dashboard(props) {

    const { data: session, status } = useSession();

    // to be continued here
}

export default Dashboard;
```

We added new imports here: **useEffect** from React, **useSession** and **signOut** from NextAuth.js and **Link**.
useEffect() allows you to run something after component is rendered, for example, fetching a data.
useSession() is used to check the user login **status**.
signOut() is used to logout the user.
Link component is from the Next.js framework, used for rendering a hyperlink.

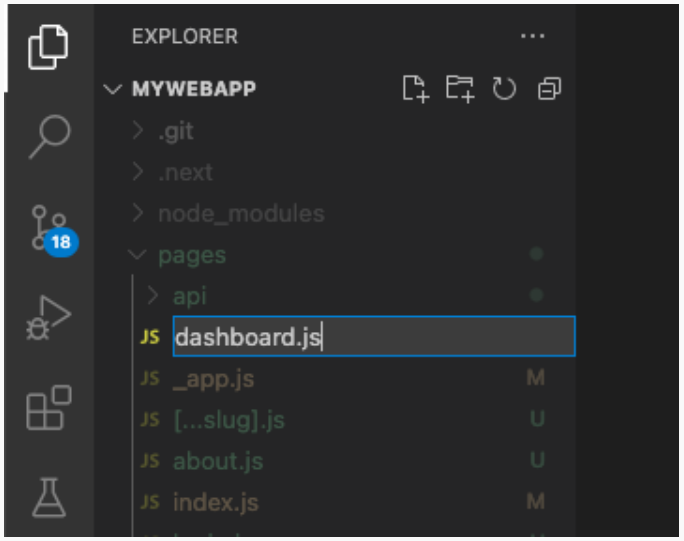
We will continue adding a little more code inside the **Dashboard()** function.

3. Add the following code inside the **Dashboard()** function.

```
function logoutHandler() {
    signOut();
}

if(status==='loading') {

    return <div className={styles.dashboard}>
        <p>Loading...</p>
    </div>

} else if (status === 'authenticated') {

    return <div className={styles.dashboard}>

        <h1>Dashboard</h1>

        <button onClick={logoutHandler}>Logout</button>
    </div>

} else if(status==='unauthenticated') {

    return <div className={styles.dashboard}>
        <Link href="/login">
            <a>Login</a>
        </Link>
    </div>

}
```

Here we check the user login status and display the relevant content. If a user is authenticated, the Dashboard content will be displayed, showing the Dashboard title and a logout button.
The Logout button will call the **logoutHandler()**. The handler then calls the **signOut()** function provided by the NextAuth.js library for logging out the user.

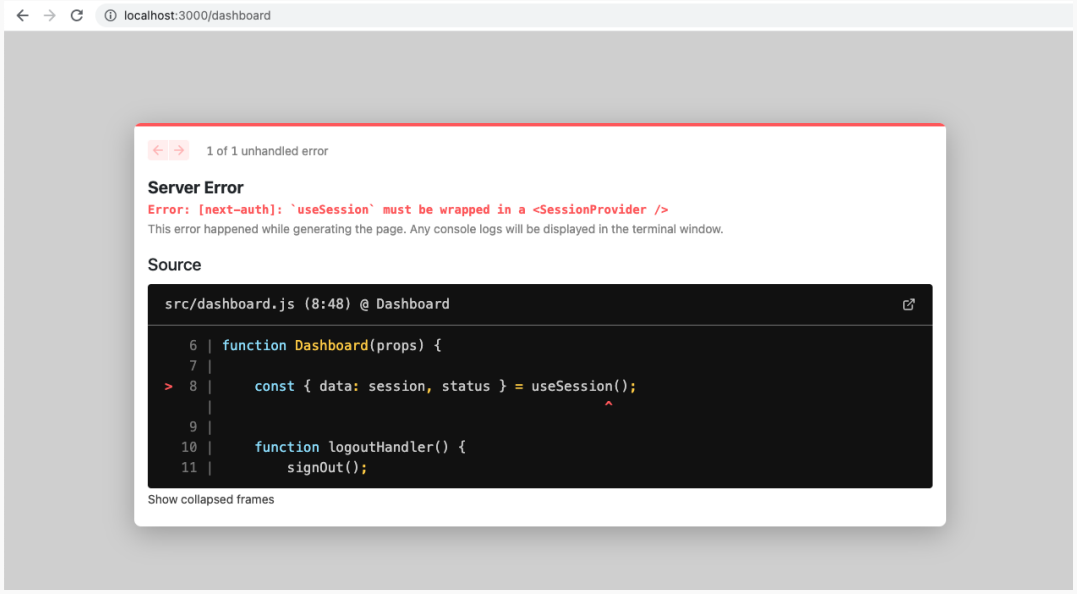
4. Create a new file **pages/dashboard.js**. This new page will use our dashboard component.



5. Add the following code.

```
import Dashboard from '../src/dashboard';

function UserDashboard() {
    return <Dashboard />
}

export default UserDashboard;
```

6. Open the page from your browser: **http://localhost:3000/dashboard**

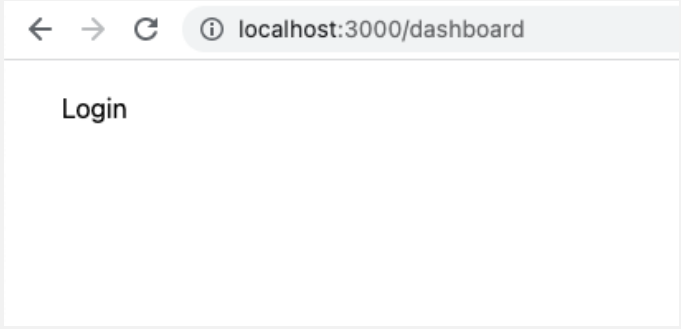


We see an error message is displayed: **'useSession' must be wrapped in a <SessionProvider />**

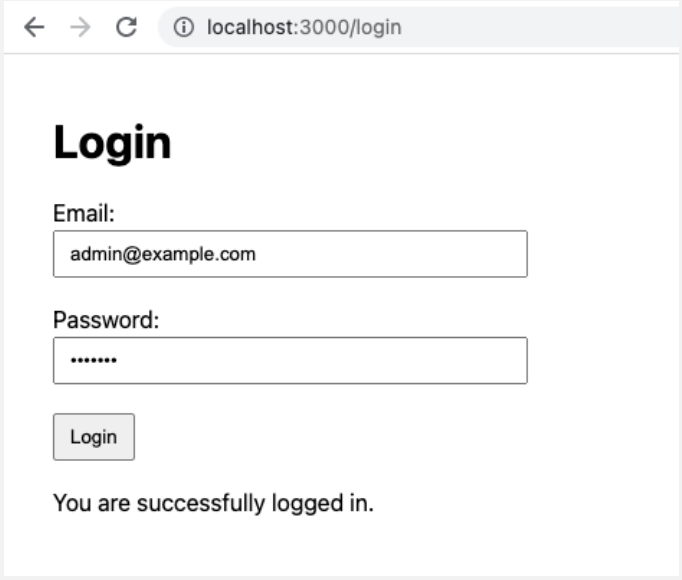
To fix this, open file pages/_app.js and replace the existing code with the following:

```
import '../styles/globals.css';
import { SessionProvider } from "next-auth/react";

export default function App({
  Component,
  pageProps: { session, ...pageProps },
}) {
  return (
    <SessionProvider session={session}>
      <Component {...pageProps} />
    </SessionProvider>
  )
}
```

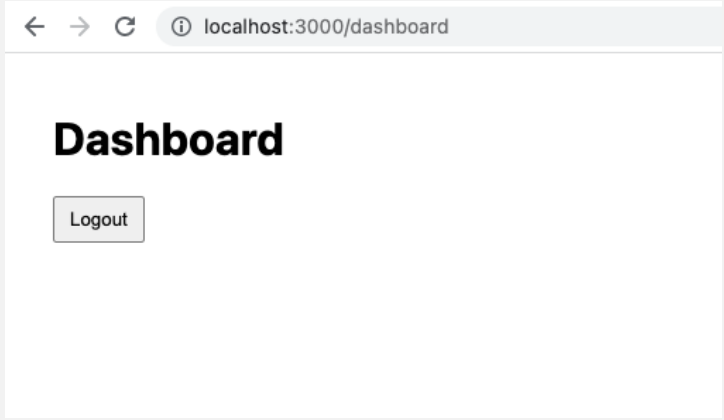
7. Test again: **http://localhost:3000/dashboard**



If you are not logged in yet, click the login link and try login as **admin**.



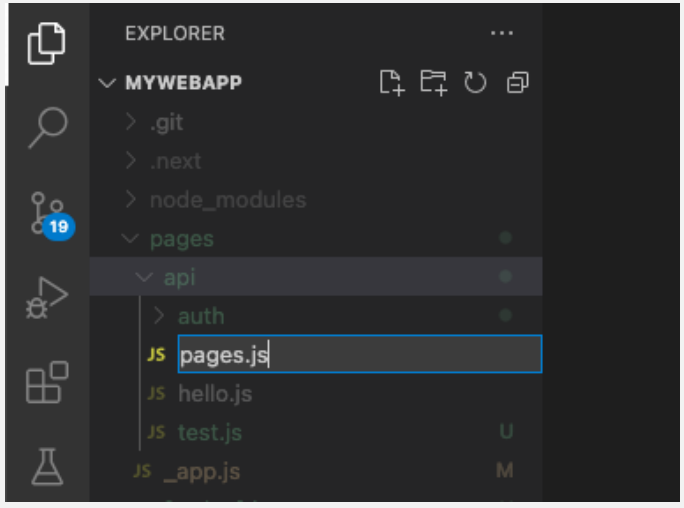
After logged in, go to: **http://localhost:3000/dashboard**



Great! Now you have a working dashboard page. Next, we will display user pages on this page.

We will prepare first an API route to get pages data from our database.

8. Create a new file **pages/api/pages.js**.



9. Add the following code.

```
import { connectDatabase } from '../../src/db';
import { getSession } from 'next-auth/react';

async function handler(req, res) {

    const session = await getSession( { req: req } );
    if(!session) {
        res.status(200).json({ ok:true, status: 400, error: 'Not authenticated!' });
        return;
    }

    let client;
    try {
        client = await connectDatabase();
    } catch (error) {
        res.status(200).json({ ok:true, status: 500, error: 'Connecting to the database failed.' });
        return;
    }

    try {
        const db = client.db();
        const pages = await db.collection('pages').find({
            username: session.user.name
        }).sort({ _id: -1 }).toArray();

        client.close();
        res.status(200).json({ ok:true, status: 200, pages: pages });

    } catch(e) {
        res.status(500).json({ ok:true, status: 500, error: 'Something went wrong.' });
    }
}

export default handler;
```

We have a new import here, **getSession** from NextAuth.js library. With **getSession()** function, we can get the logged-in user session detail. We need the username to get user pages from our **pages** collection.

Then we return the result as JSON data containing **pages** property set with an array of user pages.

10. Prepare the stylesheet by adding the following styles into **styles/Dashboard.module.css**.

```
.pagelist {
    display: flex;
    flex-flow: row wrap;
    justify-content: flex-start;
}
.card {
    padding: 25px;
    max-width: 255px;
    border: #626262 1px solid;
    margin: 0 50px 50px 0;
    width: 255px;
    min-height: 250px;
    display: flex;
    flex-direction: column;
    justify-content: space-between;
    letter-spacing: 1px;
    box-shadow: 6px 5px 0px 0px #ebebeb;
    position: relative;
}
```

11. Open our dashboard component **src/dashboard.js** and add the following code inside **Dashboard()** function.

```
const [loadedData, setLoadedData] = useState();

useEffect(()=>{
    getPages();
}, []);

function getPages(callback) {
    fetch('/api/pages')
    .then(response=>response.json())
    .then(data=>{
      if(callback) callback();
      setLoadedData(data);
    });
}
```

Here we created **getPages()** function that fetch our API route **/api/pages** we have created previously. The result will be set to a state variable **loadedData** using **setLoadedData()** function.

The **getPages()** function should be called after the component is rendered, so we call it inside **useEffect()** function.

In the **getPages()** function we pass a **callback** parameter. It is not used for the moment, but it will help in case you need something to run after the pages data received.

12. Add the following code under the title **<h1>Dashboard</h1>**.

```
<div className={styles.pagelist}>
{loadedData?loadedData.pages.map(page=>{
        return (
            <div key={page.slug} className={styles.card}>
                <h2>{page.slug?'/'+page.slug:'/...home'}</h2>
                <p>{page.title}</p>
            </div>
        )
    }):<p>Loading...</p>
}
</div>
```

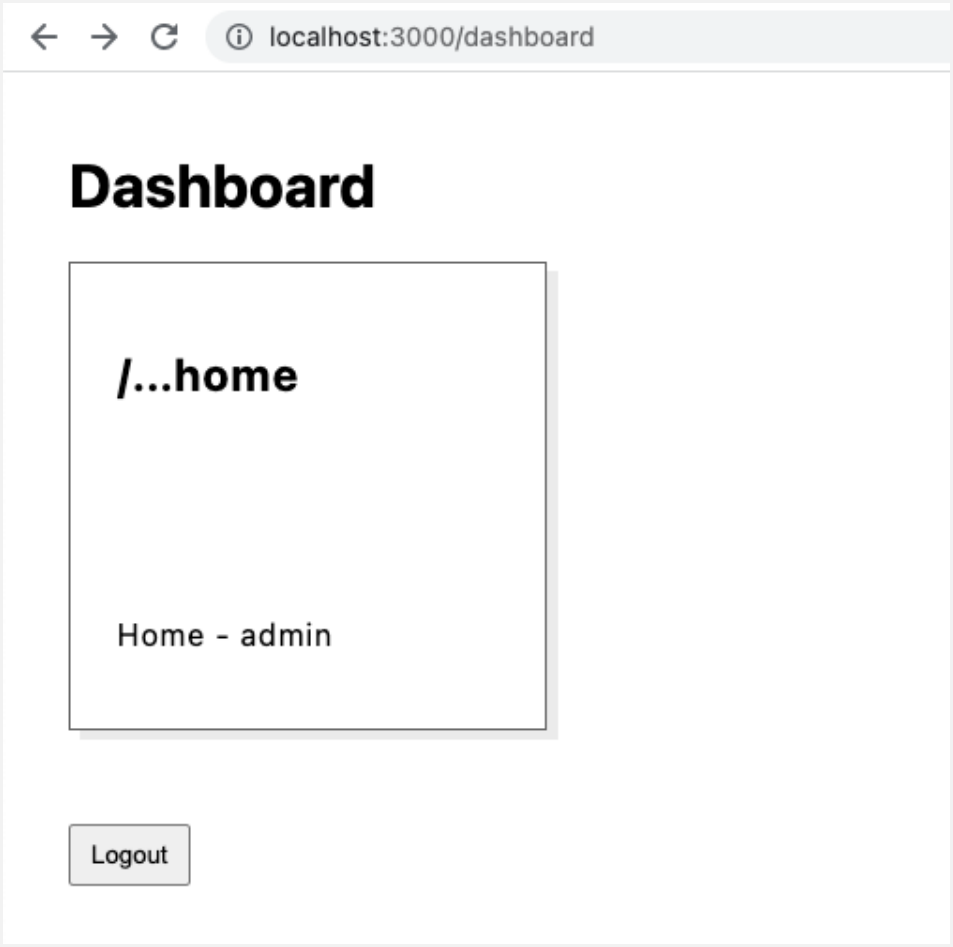
Here we check if **loadedData** is not empty. Then we read the data and render each item/page. Now we still have one (home) page for each user so it will show only one page on the dashboard.

Here is the complete code for the dashboard component **src/dashboard.js**.

```
import { useEffect, useState } from 'react';
import { useSession, signOut } from 'next-auth/react';
import Link from 'next/link';
import styles from '../styles/Dashboard.module.css';

function Dashboard(props) {

    const { data: session, status } = useSession();
    const [loadedData, setLoadedData] = useState();

    useEffect(()=>{
        getPages();
    }, []);

    function getPages(callback) {
        fetch('/api/pages')
        .then(response=>response.json())
        .then(data=>{
          if(callback) callback();
          setLoadedData(data);
        });
    }

    function logoutHandler() {
        signOut();
    }

    if(status==='loading') {

        return <div className={styles.dashboard}>
            <p>Loading...</p>
        </div>

    } else if (status === 'authenticated') {

        return <div className={styles.dashboard}>
```

```
            <h1>Dashboard</h1>

            <div className={styles.pagelist}>
                {loadedData?loadedData.pages.map(page=>{
                        return (
                            <div key={page.slug} className={styles.card}>
                                <h2>{page.slug?'/'+page.slug:'/...home'}</h2>
                                <p>{page.title}</p>
                            </div>
                        )
                    }):<p>Loading...</p>
                }
            </div>

            <button onClick={logoutHandler}>Logout</button>
        </div>

    } else if(status==='unauthenticated') {

        return <div className={styles.dashboard}>
            <Link href="/login">
                <a>Login</a>
            </Link>
        </div>

    }
}

export default Dashboard;
```

13. Open **http://localhost:3000/dashboard** to see the result.



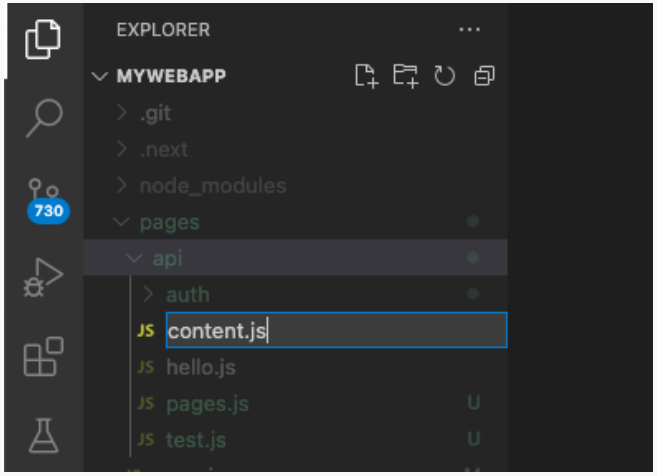
Congratulations! Now you have a working dashboard for your site users.

11. Adding Page Builder

In this chapter, we will allow users to edit their home page.
We will use ContentBox.js, a page builder Javascript library.

1. Adding API Route for Getting & Saving Page

1. First, we will create an API route for getting and saving page data. Create a new file **pages/api/content.js**.



2. Add the following code.

```
import { connectDatabase } from '../../src/db';
import { getSession } from 'next-auth/react';

async function handler(req, res) {

    const session = await getSession( { req: req } );
    if(!session) {
        res.status(200).json({ ok:true, status: 400, error: 'Not authenticated!' });
        return;
    }

    if(req.method !== 'POST') return;

    const data = JSON.parse(req.body)
    const action = data.action;
    const slug = data.slug;
    const html = data.html;
    const mainCss = data.mainCss;
    const sectionCss = data.sectionCss;

    if(action==='save') {

        let client;
        try {
            client = await connectDatabase();
        } catch (error) {
            res.status(200).json({ ok:true, status: 500, error: 'Connecting to the database failed.' });
            return;
        }

        try {
            const db = client.db();
            const myquery = { slug: slug, username: session.user.name };
            const newvalues = {
                $set: {
                    html: html,
                    mainCss: mainCss,
                    sectionCss: sectionCss,
                }
            };
            await db.collection('pages').updateOne(myquery, newvalues);
```

```
            client.close();
            res.status(200).json({ ok:true, status: 201 });

        } catch(e) {
            res.status(200).json({ ok:true, status: 500, error: 'Something went wrong.' });
        }

    } else {

        let client;
        try {
            client = await connectDatabase();
        } catch (error) {
            res.status(200).json({ ok:true, status: 500, error: 'Connecting to the database failed.' });
            return;
        }

        try {
            const db = client.db();
            const content = await db.collection('pages').findOne({
                slug: slug,
                username: session.user.name
            });

            client.close();
            res.status(200).json({ ok:true, status: 200, content });

        } catch(e) {
            res.status(500).json({ ok:true, status: 500, error: 'Something went wrong.' });
        }
    }
}

export default handler;
```

Our API route receives the POST data and check if it contains action='save', then it will perform a saving into the database. The posted data contains:

- **slug**
- **html**
- **mainCss**
- **sectionCss**.

The **mainCss** and **sectionCss** contains the stylesheet needed by the content (the **html**) edited using the ContentBox.js library.

If the posted data is not for saving, then it reads the saved page data from the database and returns it to the client.

2. Adding File Upload

We will also need to create an API route for file upload. The file upload functionality is needed by ContentBox.js for uploading media files, such as images or videos during page editing.

File upload requires further consideration on where we want to store our files. You can choose, for example, cloud storage provided by Amazon (Amazon S3), Digital Ocean Spaces, or other cloud storage services.

In this guide, we will create our own file handling using a separate Node.js application, on the same server we're using.

On the server, we have our Next.js project located in:

/home/hobnob/apps/mywebapp

Now we will create a project located in:

/home/hobnob/apps/mywebassets

We also need to create the same on our local computer, so that we can use it during development.

1. Login to your server using the user **hobnob**.

```
$ ssh hobnob@<server-ip>
```

2. Create a folder for our new Node.js project.

```
$ cd apps
```

```
$ mkdir mywebassets
```

```
$ cd mywebassets
```

3. We'll start creating our project by running the init command.

```
$ npm init -y
```

4. Install the Express framework

```
$ npm i express
```

5. Create a new file **mywebassets.js**.

```
$ vi mywebassets.js
```

6. Add the following code.

```
const express = require('express');
const fs = require('fs');
const app = express();

app.use('/uploads',express.static(__dirname + '/uploads'));
app.listen(8081, function() {
    console.log('App running on port 8081');
});
```

This will make the Express framework serves static files **stored** in the **uploads** folder.
The application will run on port 8081.

7. We haven't created the **uploads** folder. Let's create it.

```
$ mkdir uploads
```

8. Run the application using PM2.

```
$ pm2 start mywebassets.js
```

```
$ pm2 save
```

9. Update the OpenResty configuration file.

```
$ sudo vi /etc/openresty/nginx.conf
```

10. Make our new application accessible from the web using the **/uploads/** path. Add the following inside **server { listen 443 ssl; ..**

```
    location /uploads/ {
        proxy_pass http://localhost:8081;
        proxy_set_header Host $host;
    }
```

11. Restart OpenResty.

```
$ sudo systemctl restart openresty
```

Our new Node.js application for serving static files is now ready. All files stored in the **uploads** folder is accessible from the web. For example, you can open:

https://sitemagz.com/uploads/example.jpg (if example.jpg exists in the folder)

Now we will recreate the same project on our local computer so that we can try it locally. For this, we need to repeat steps 2 - 6.

12. Repeat steps 2 - 6 on your local computer.

If, for example, your Next.js project is located on:

/Users/john/mywebapp (on Mac), or

c:\Users\john\mywebapp (on Windows)

You can create the new Node.js project on the same level folder.

/Users/john/mywebassets (on Mac), or

c:\Users\john\mywebassets (on Windows)

13. After repeating steps 2 - 6, from the **mywebassets** folder, run the project.

```
$ node mywebassets.js
```

Our new Node.js application for serving static files is now ready in our local computer. All files stored in **uploads** folder is now accessible. For example, you can open:

http://localhost:8081/uploads/example.jpg (if example.jpg exists in the **uploads** folder)

Now we can start creating an API route for file upload. Files will be uploaded into the **mywebassets/** project, in the **uploads** folder so that they can be accessible.

14. Go to your Next.js project **mywebapp/**.

Open file **.env.development** and add the following lines.

```
WEB_ASSETS_PATH=/Users/john/mywebassets
WEB_ASSETS_URL=http://localhost:8081
```

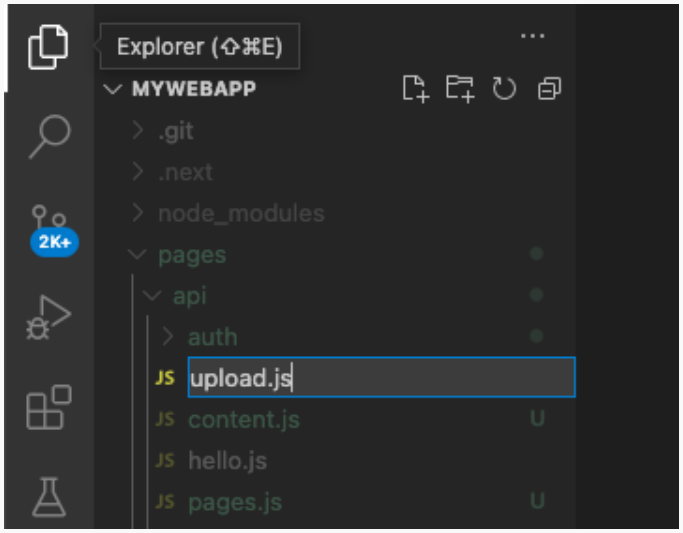
Here we add 2 configuration variables (**WEB_ASSETS_PATH** & **WEB_ASSETS_URL**). We will use them later in our API route. Please make sure that the **WEB_ASSETS_PATH** variable points to the location of your newly created **mywebassets** project on your local computer.

We'll also need to add the variables in the **.env.production** file that will be used in production/deployment.

Open file **.env.production** and add the following lines.

```
WEB_ASSETS_PATH=/home/hobnob/apps/mywebassets
WEB_ASSETS_URL=
```

15. Create a new file **pages/api/upload.js**.



16. Add the following code.

```
import { getSession } from 'next-auth/react';
import fs from 'fs';
import path from 'path';

async function handler(req, res) {

    const session = await getSession( { req: req } );
    if(!session) {
        res.status(200).json({ ok:true, status: 400, error: 'Not authenticated!' });
        return;
    }

    if(req.method === 'POST') {
        const username = session.user.name;
        const base64Data = req.body.image;
        const filename = req.body.filename;
        const directoryPath = path.join(process.env.WEB_ASSETS_PATH, 'uploads/' + username);

        if (!fs.existsSync(directoryPath)){
            fs.mkdirSync(directoryPath, { recursive: true });
        }

        fs.writeFile(directoryPath + '/' + filename, base64Data, 'base64', (err)=>{
            if (err) {
                res.status(500).json({ ok:true, status: 500, error: 'Something went wrong.' });
                return
            }
            res.status(200).json({ ok:true, status: 200, url:  process.env.WEB_ASSETS_URL + '/uploads/' + username + '/'
+ filename});
        });

    }
}

export default handler;
```

Here we receive the posted data containing filename and content in base64 format. We get our previously defined variable **WEB_ASSETS_PATH** using **process.env.WEB_ASSETS_PATH** and combine it with **uploads** folder as the location for saving files.

Since we will have multiple users, we specify the upload folder for each user based on the username:
<WEB_ASSETS_PATH>/uploads/<username>/

For saving the file, we use the **writeFile()** function we imported from **fs** (file system) module.

Now we have 2 API routes that we'll use for page editing:
- **/api/content** => For getting & saving page content
- **/api/upload** => for uploading media files

Next, we'll install ContentBox.js library for page editing.

3. Adding Page Builder

1. Open terminal, go to your project folder and install **ContentBox.js** library.

```
$ cd mywebapp
```

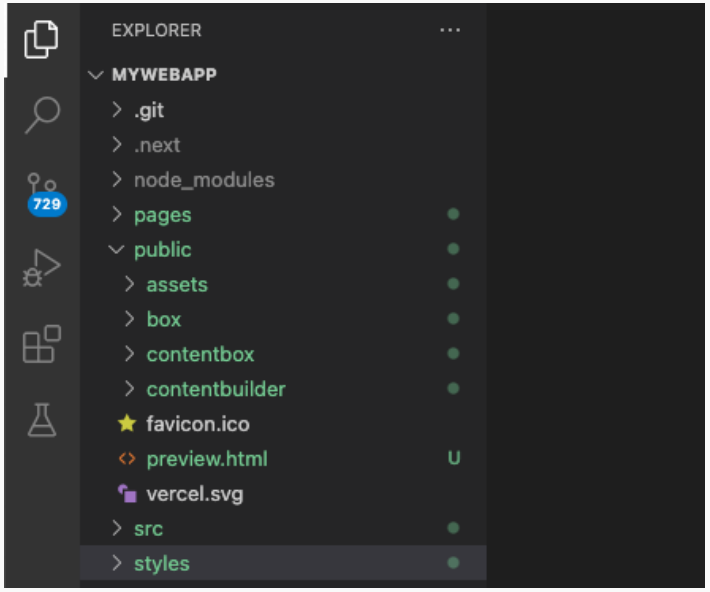
```
$ npm i @innovastudio/contentbox
```

2. Rerun the project.

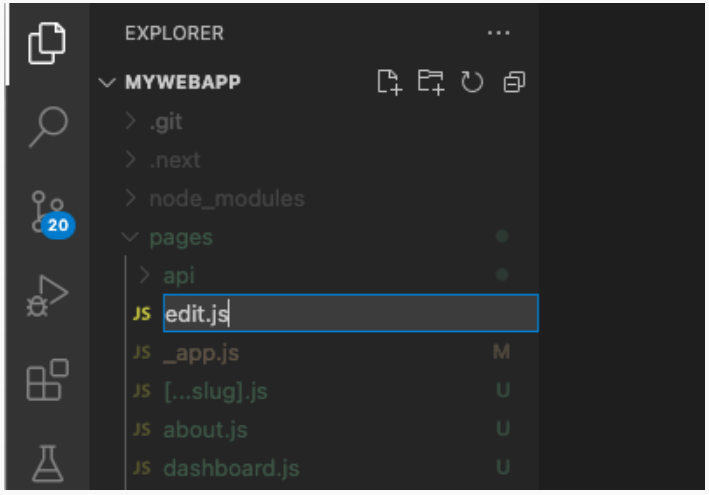
```
$ npm run dev
```

3. From ContentBox.js package copy the following into **public/** folder in your project.

- assets
- box
- contentbox
- contentbuilder
- preview.html



4. Create a new page **pages/edit.js**.



5. Before adding code into the file, let's review what we'll add in simplified structure.

```
import { getSession } from 'next-auth/react';
import { useEffect } from 'react';
import Head from 'next/head';
import Script from 'next/script';
import ContentBox from '@innovastudio/contentbox';

export default function Home() {

    let obj;
    useEffect(async ()=>{
        builder();

    }, []);

    async function builder() {

        obj = new ContentBox({
            wrapper: '.is-wrapper',
            // more options..
        });

        // Get content from /api/content and render it to div.is-wrapper element
    }

    return (
        <>
            <div className="is-wrapper" style={{opacity: 0}}>
            </div>
        </>
    )
}
```

Inside the **useEffect()** , we call the **builder()** function which starts the **ContentBox** page builder. Then an API call to **/api/content** is performed to load page content into the builder for editing.

Page content will be rendered in the **<div className="is-wrapper"..>** element. Note that we added opacity=0 here to make the element invisible during the loading process. It will show after the content ready for editing.

6. Here is the complete code. Add this into **pages/edit.js**.

```
import { getSession } from 'next-auth/react';
import { useEffect } from 'react';
import Head from 'next/head';
import Script from 'next/script';
import ContentBox from '@innovastudio/contentbox';

export default function Home() {

    let obj;
    let timeoutId;

    useEffect(()=>{
        builder();
        return () => {
            obj.destroy();
        };
    }, []); // eslint-disable-line react-hooks/exhaustive-deps

    async function builder() {
        const session = await getSession();
        if(!session) return;

        addExternalStyles([
            '/assets/minimalist-blocks/content.css',
            '/box/box-flex.css',
            '/contentbox/contentbox.css',
            '/contentbuilder/contentbuilder.css',
            '/assets/scripts/glide/css/glide.core.css',
            '/assets/scripts/glide/css/glide.theme.css',
            '/assets/scripts/navbar/navbar.css'
        ]);

        obj = new ContentBox({
            wrapper: '.is-wrapper',

            designUrl1: '/assets/designs/basic.js',
            designUrl2: '/assets/designs/examples.js',
            designPath: '/assets/designs/',
            contentStylePath: '/assets/styles/',

            snippetUrl: '/assets/minimalist-blocks/content.js',
            snippetPath: '/assets/minimalist-blocks/',
            snippetPathReplace: ['assets/','/assets/'], /* for snippets */
            modulePath: '/assets/modules/',
            assetPath: '/assets/', // Used for the location of ionicons/
            fontAssetPath: '/assets/fonts/',
            slider: 'glide',
```

```
    navbar: true,
    onChange:  ()=>{
        //Auto Save
        clearTimeout(timeoutId);
        timeoutId = setTimeout(()=>{
            save();
        }, 1000);
    },
    onUploadCoverImage: (e) => {
        uploadFile(e, (response)=>{
            if(!response.error) {
                const uploadedImageUrl = response.url;
                if(uploadedImageUrl) obj.boxImage(uploadedImageUrl);
            }
        });
    },
    onMediaUpload: (e)=>{
        uploadFile(e, (response)=>{
            if(!response.error) {
                const uploadedImageUrl = response.url;
                if(uploadedImageUrl) obj.returnUrl(uploadedImageUrl);
            }
        });
    },
    onVideoUpload: (e)=>{
        uploadFile(e, (response)=>{
            if(!response.error) {
                const uploadedFileUrl = response.url;
                if(uploadedImageUrl) obj.returnUrl(uploadedFileUrl);
            }
        });
    },
});

// Adding custom buttons
obj.addButton({
    'pos': 2, // button position
    'title': 'Undo', // title
    'html': '<svg class="is-icon-flex" style="width:14px;height:14px;"><use xlink:href="#ion-ios-undo"></use></svg>',
    'onClick': ()=>{
        obj.undo();
    }
});
```

```
obj.addButton({
    'pos': 3,
    'title': 'Redo',
    'html': '<svg class="is-icon-flex" style="width:14px;height:14px;"><use xlink:href="#ion-ios-redo"></use></svg>',
    'onClick': ()=>{
        obj.redo();
    }
});
obj.addButton({
    'pos': 5,
    'title': 'Zoom',
    'html': '<svg class="is-icon-flex" style="width:15px;height:15px;"><use xlink:href="#icon-zoom-in"></use></svg>',
    'onClick': ()=>{
        obj.viewZoom();
    }
});
obj.addButton({
    'pos': 6,
    'title': 'Preview',
    'html': '<svg class="is-icon-flex" style="width:16px;height:16px;"><use xlink:href="#ion-eye"></use></svg>',
    'onClick': ()=>{

        let html = obj.html();
        localStorage.setItem('preview-html', html);
        let mainCss = obj.mainCss();
        localStorage.setItem('preview-maincss', mainCss);
        let sectionCss = obj.sectionCss();
        localStorage.setItem('preview-sectioncss', sectionCss);

        window.open('/preview.html', '_blank').focus();
    }
});

//Get slug from query string ?page=slug
let search = window.location.search;
let params = new URLSearchParams(search);
let slug = params.get('page');
if(!slug) slug = '';

const reqBody = { slug };
```

```
    let result = await fetch('/api/content', {
        method:'POST',
        body: JSON.stringify(reqBody),
        header: {
            'Content-Type': 'application/json'
        }
    });
    result = await result.json();
    if(!result.error) {
        //Load content
        const html = result.content.html || '';
        const mainCss = result.content.mainCss || '';
        const sectionCss = result.content.sectionCss || '';

        obj.loadHtml(html); // Load html
        obj.loadStyles(mainCss, sectionCss); // Load styles

        // Add required scripts for viewing the content
        addExternalScripts(['/box/box-flex.js']);
    }
}

async function uploadFile(e, callback) {
    const selectedFile = e.target.files[0];
    const filename = selectedFile.name;
    const reader = new FileReader();
    reader.onload = async (e) => {
        let base64 = e.target.result;
        base64 = base64.replace(/^data:(.*?);base64,/, "");
        base64 = base64.replace(/ /g, '+');

        const reqBody = { image: base64, filename: filename };
        let result = await fetch('/api/upload', {
            method:'POST',
            body: JSON.stringify(reqBody),
            headers: {
                'Content-Type': 'application/json'
            }
        });
```

```
        result = await result.json();
        if(!result.error) {
            callback(result);
        }
    }
    reader.readAsDataURL(selectedFile);
}

async function save() {
    obj.saveImages('', async ()=>{

        //Save content
        let html = obj.html();
        let mainCss = obj.mainCss(); // Default typography style for the page
        let sectionCss = obj.sectionCss(); // Typography styles for the sections

        //Get slug from query string ?page=slug
        let search = window.location.search;
        let params = new URLSearchParams(search);
        let slug = params.get('page');
        if(!slug) slug = '';

        const reqBody = { action: 'save', slug: slug, html: html, mainCss: mainCss, sectionCss: sectionCss };
        let result = await fetch('/api/content', {
            method:'POST',
            body: JSON.stringify(reqBody),
            header: {
                'Content-Type': 'application/json'
            }
        });
        result = await result.json();
        if(result.error) {
            alert(result.error);
        }

    }, async (img, base64, filename)=>{

        // Upload base64 images
        const reqBody = { image: base64, filename: filename };
        let result = await fetch('/api/upload', {
            method:'POST',
            body: JSON.stringify(reqBody),
```

```
            headers: {
                'Content-Type': 'application/json'
            }
        });
        result = await result.json();
        if(!result.error) {
            const uploadedImageUrl = result.data.url;
            img.setAttribute('src', uploadedImageUrl); // Update image src
        }
    });
}

function addExternalScripts(arrScript, callback) {
    if(arrScript.length===0) {
        if(callback) callback();
        return;
    } else {
        const url = arrScript[0];
        const includes = document.head.querySelectorAll('script');
        includes.forEach((link) => {
            if(link.src.indexOf(url)!==-1) {
                link.parentNode.removeChild(link); // Remove existing
            }
        });
        const script = document.createElement('script');
        script.onload = () =>{
            addExternalScripts(arrScript.slice(1), callback);
        };
        script.src = url;
        document.head.appendChild(script);
    }
}

function addExternalStyles(arrStyle) {
    const includes = document.head.querySelectorAll('[data-my-css-link]');
    includes.forEach((link) => {
        link.parentNode.removeChild(link); // Remove existing
    });
    for(let i=0;i<=arrStyle.length-1;i++){
        const url = arrStyle[i];
        const link = document.createElement('link');
        link.rel = 'stylesheet';
        link.setAttribute('data-my-css-link','');
```

```
            link.href = url;
            document.head.appendChild(link);
        }
    }

    return (
        <>
            <Head>
                <title>Edit Content</title>
                <meta name="description" content="Edit Content" />
                <link rel="icon" href="/favicon.ico" />
            </Head>

            <div className="is-wrapper" style={{opacity: 0}}>
            </div>

            <Script src="assets/scripts/glide/glide.js" />
            <Script src="/assets/scripts/navbar/navbar.min.js" />
        </>
    )
}
```

We get the requested page from the query string. For example, if you open:

http://localhost:3000/edit?page=about => it will load a page which has slug=about for editing.

However, currently we only have single page (the home page, which has empty slug). To edit the home page, just open:

http://localhost:3000/edit

For ContentBox.js page builder configuration, you can refer to the documentation at:
https://demo.innovastudio.com/docs/ContentBox.pdf

7. To test, open: **http://localhost:3000/edit**.

The ContentBox page builder is now visible. The home page is still empty. You can try adding content by clicking the (+) icon from the left sidebar.

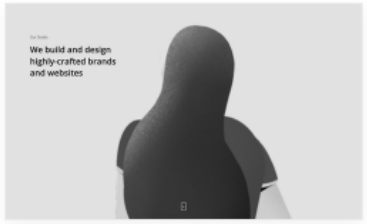
localhost:3000/edit

SIMPLE START:

RANDOM | BASIC | SLIDER | MORE

QUICK START:

HEADER | ARTICLE | MORE



Desgining and building highly-crafted brands and websites.

Nav Bar

One. Two. Three.

LOVE IT

Branding

UI/UX

Front-end

01 02 03 04

Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book.

Introduction

Hi there, I design & build highly-crafted brands & websites

8. Try adding a section from the template. During editing, the page will perform auto saving.

9. Then, to display the saved page on our website, we need add/modify some code in file:

- **pages/[...slug].js**
- **pages/index.js**

Remember that [...slug].js and index.js catch all requests to the website to display user's page? They also have the same code. Now let's open **pages/[...slug].js** first.

To display user's page, we need to read **pages** collection from our database and return:
- **html**
- **mainCss**
- **sectionCss**

mainCss and **sectionCss** contains the styles needed by the **html** content as edited by the ContentBox page builder.

10. Add the **mainCss** and **sectionCss** to the **getServerSideProps()** returned data in **pages/[...slug].js**.

```
export async function getServerSideProps(context) {

    ...

    return {
      props: {
        ...
        html: page?page.html:'',
        mainCss: page?page.mainCss?page.mainCss:'':'',
        sectionCss: page?page.sectionCss?page.sectionCss:'':'',
        ...
      }
    };
}
```

11. Add new **Script** import.

```
import { connectDatabase } from '../src/db';
import Head from 'next/head';
import Script from 'next/script';
...
```

12. Replace this part:

```
<Head>
    <title>{props.title}</title>
    <link rel="icon" href="/favicon.ico" />
</Head>
<p>Welcome!</p>
```

with the following:

```
<Head>
  <title>{props.title}</title>
  <meta name="description" content={props.desc} />
  <link rel="icon" href="/favicon.ico" />
```

```
  {
  /* Render mainCss.
      Since adding raw HTML / string into head section is not supported in Next.js,
      we dynamically create <link rel="stylesheet"> based on the mainCss string.
  */
    props.mainCss?content.props.split('<link').map(string=>{ // extract each link found
      if(string!=='') {
        string = '<link' + string;

        let s = string.split('href="')[1];
        const href = s.substring(0, s.indexOf('"'));

        return (
          <link key={attributes['href']} href={attributes['href']} rel="stylesheet" />
        );
      }
  }):null
  }

  { /* Render sectionCss */
  props.sectionCss?props.sectionCss.split('<link').map(string=>{
      if(string!=='') {
        string = '<link' + string;

        let s = string.split('href="')[1];
        const href = s.substring(0, s.indexOf('"'));

        s = string.split('data-class="')[1];
        const className = s.substring(0, s.indexOf('"'));

        return (
            <link key={className} data-name="contentstyle" data-class={className} href={href} rel="stylesheet" />
        );
      }
  }):null
  }
</Head>

{/* Render HTML */}
<div className="is-wrapper" dangerouslySetInnerHTML={{__html: props.html}}/>

<Script src="/box/box-flex.js" />
<Script src="/assets/scripts/glide/glide.js" />
<Script src="/assets/scripts/navbar/navbar.min.js" />
```

This part renders **mainCss**, **sectionCss** and **html** for displaying page content.

13. Now copy the complete code from **pages/[...slug].js** to **pages/index.js**.

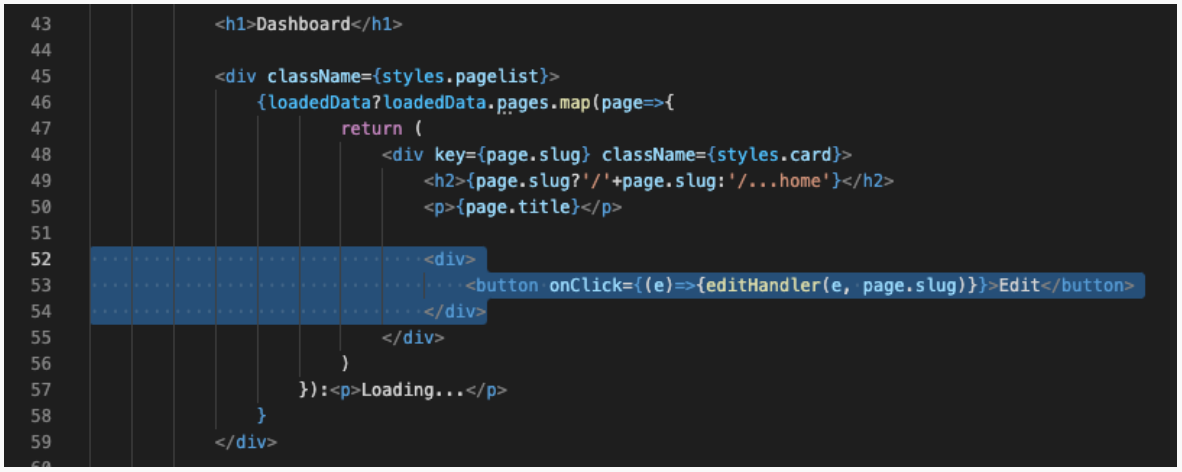
14. Open **pages/_app.js** and add some css imports needed by the page content.

```
import '../public/assets/minimalist-blocks/content.css'
import '../public/box/box-flex.css'
import '../public/assets/scripts/glide/css/glide.core.css'
import '../public/assets/scripts/glide/css/glide.theme.css'
import '../public/assets/scripts/navbar/navbar.css'
```

15. Let's test it by opening: **http://localhost:3000**

16. Great! Now you have created a page builder and display the saved page on the site. One small thing is still needed. Open **src/dashboard.js**. Add the following code inside the dashboard content:

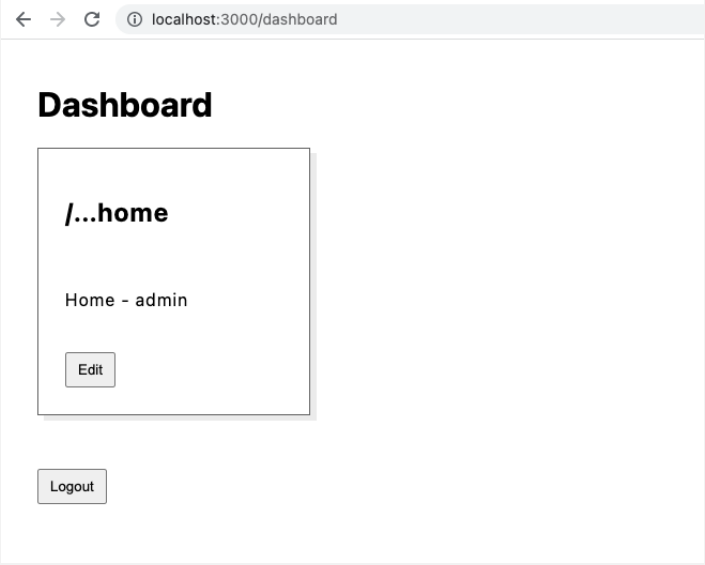
```
<div>
    <button onClick={(e)=>{editHandler(e, page.slug)}}>Edit</button>
</div>
```



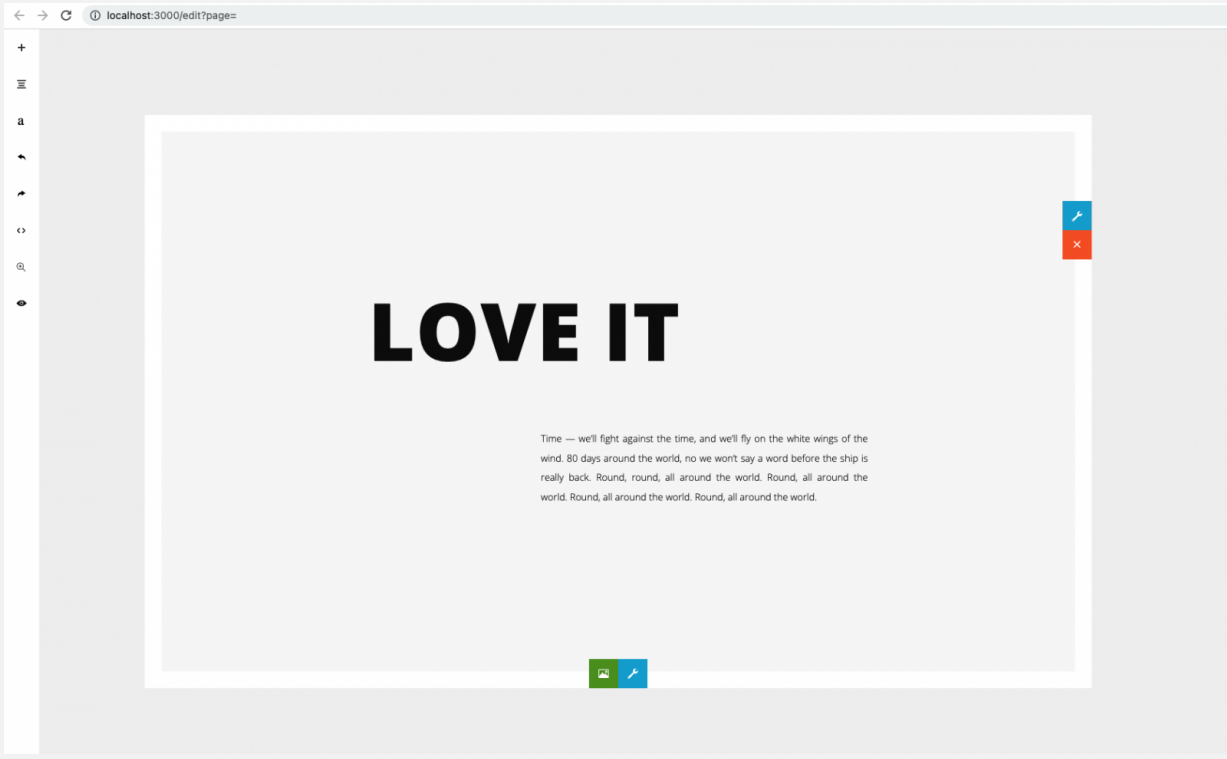
17. Then add the **editHandler()** function.

```
function editHandler(e, slug) {
    e.preventDefault();
    window.open('/edit?page='+slug, '_blank');
}
```

18. Now open **http://localhost:3000/dashboard**.



Edit button is now added. To edit the page, click the button and it will open: **http://localhost:3000/edit?page=<slug>**



Congratulations! Now users can edit their home page and display the result on their site!

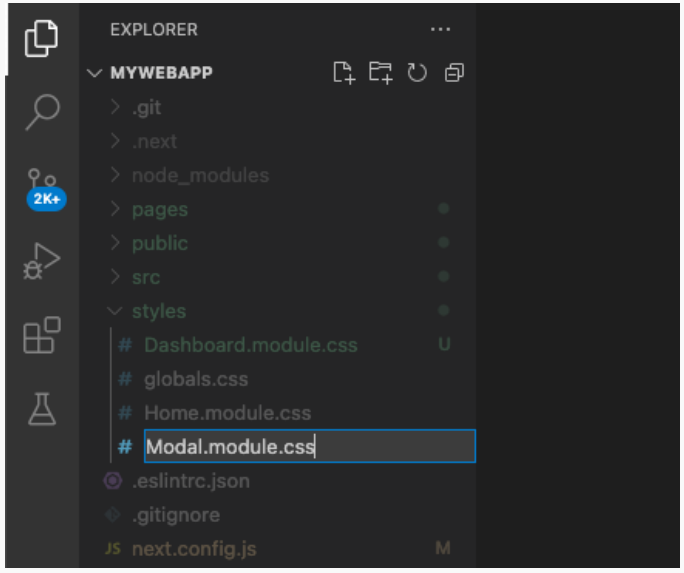
12. Adding Site Settings

In this chapter, we will add a **Site Settings** modal dialog that allows users to enter a custom domain name for their site. The user custom domain name will be stored in **users** collection in our database.

1. Adding Modal Component

Before creating Site Settings modal dialog, we need to add first a modal component. We will not explain this component so that we can focus on our main project. Just follow the steps to add this component and see how it can be used.

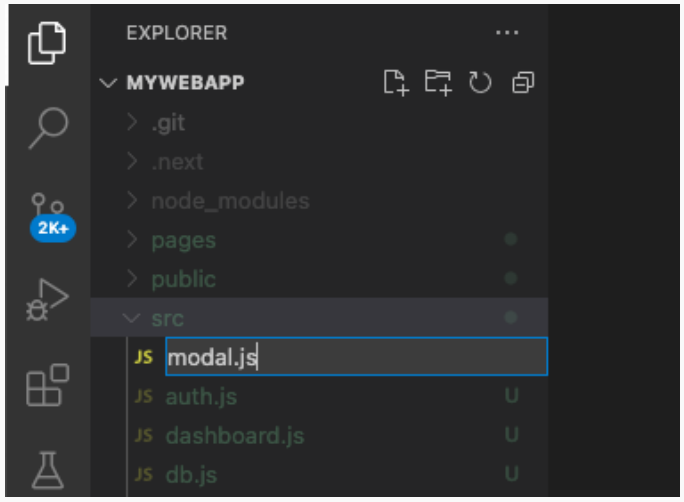
1. Create a new css file **styles/Modal.module.css**.



2. Add the following styles.

```
.modal {
    border: #686868 1px solid;
    box-shadow: 6px 5px 0px rgb(185 185 185 / 20%);
    background-color: white;
    padding: 60px;
    width: 100%;
    max-width: 490px;
    z-index: 10;
    position: relative;
  }
.backdrop {
    position: fixed;
    z-index: 1;
    background-color: rgb(255 255 255 / 50%);
    width: 100%;
    height: 100vh;
    top: 0;
    left: 0;
    display:flex;
    justify-content:center;
    align-items:center;
}
```

3. Create a new file **src/modal.js**.



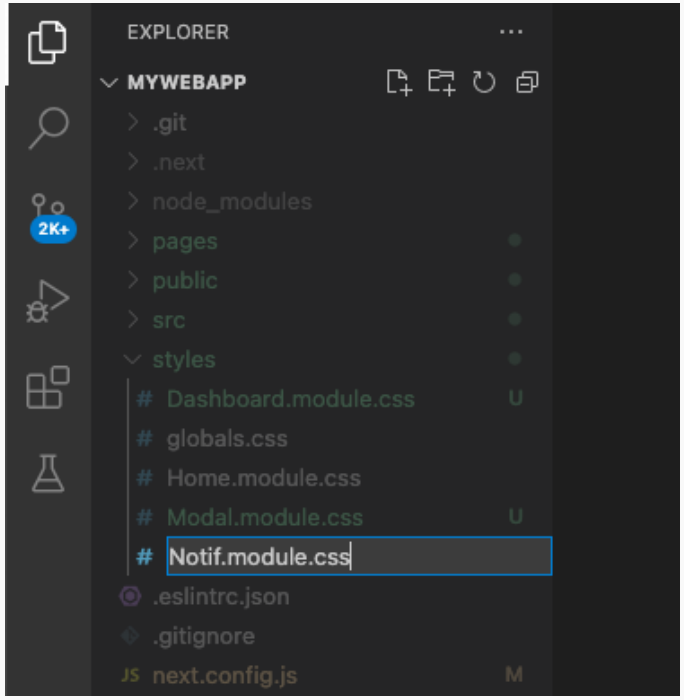
4. Add the following code.

```
import styles from '../styles/Modal.module.css';

function Modal(props) {

    let prevElmFocus;

    function modalClick(e) {
        try{
            if(e.target.className.indexOf('backdrop') !== -1 && e.target == prevElmFocus) {
                props.onClose(e);
                e.preventDefault();
                e.stopPropagation();
            }
            prevElmFocus = null;
        } catch(e) {

        }
    }
    function mouseDownHandler(e) {
        prevElmFocus = e.target;
    }
    return <>
    {props.show?
        <div className={styles.backdrop} onClick={modalClick} onMouseDown={mouseDownHandler}>
            <div className={styles.modal} style={{height:props.height?
props.height:'auto',maxWidth:props.width?props.width:490, padding: props.padding?props.padding:60}}>
            {props.children}
            </div>
        </div>: null
    }
    </>;
}

export default Modal;
```

2. Adding Notification Component

We also need a notification component to display messages, for example, a 'Please wait...' message during a saving process.

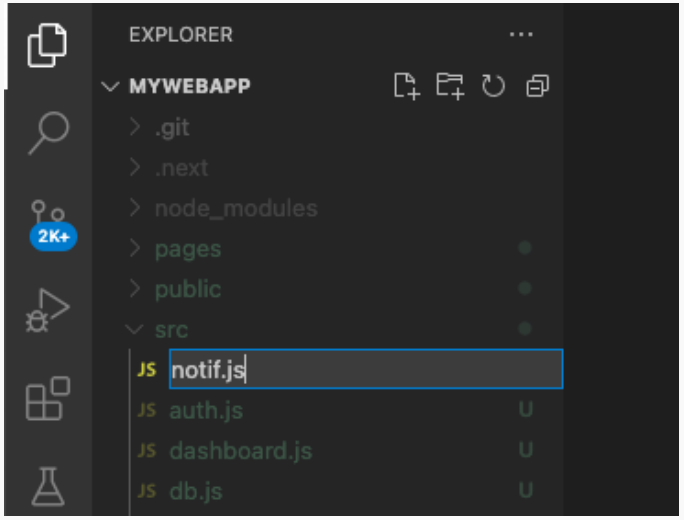
1. Create a new css file **styles/Notif.module.css**.



2. Add the following styles.

```
.notif {
    opacity: 0;
    transition: all 0.3s ease;
    position: fixed;
    top: 0;
    width: 500px;
    left: calc(50% - 250px);
    background: #4e4ef5;
    color: #fff;
    padding: 12px;
    display: flex;
    justify-content: center;
    align-items: center;
    font-weight: 300;
    letter-spacing: 1px;
    line-height: 1.4;
    z-index:1;
  }
.wait {
    opacity: 1;
    background: #4e4ef5;
    color: #fff;
  }
.error {
    opacity: 1;
    background: orangered;
    color: #fff;
}
```

3. Create a new file **src/notif.js**.

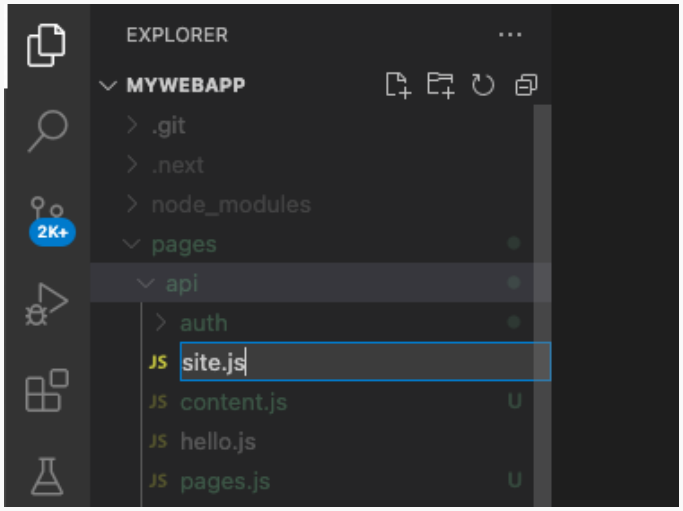


4. Add the following code.

```
import styles from '../styles/Notif.module.css';

function Notif(props) {
    if(props.mode) {
        if(props.mode==='wait') {
            return <div className={`${styles.notif} ${styles.wait}`}>
                {props.text}
            </div>;
        } else if(props.mode==='error') {
            return <div className={`${styles.notif} ${styles.error}`}>
                {props.text}
            </div>;
        } else {
            return <div className={props.notif}>
                {props.text}
            </div>;
        }
    } else {
        return null;
    }
}

export default Notif;
```

3. Creating an API Route to Get or Save Site Settings

1. Create a new file **pages/api/site.js**.



2. Add the following code.

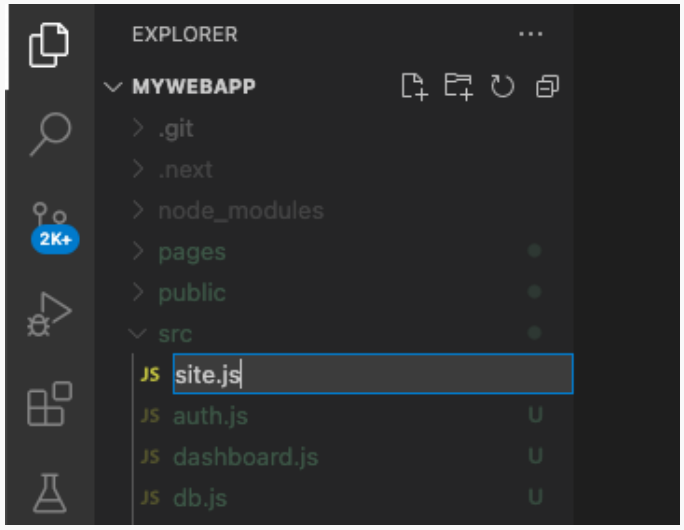
```
import { connectDatabase } from '../../src/db';
import { getSession } from 'next-auth/react';

async function handler(req, res) {

    const session = await getSession( { req: req } );
    if(!session) {
        res.status(200).json({ ok:true, status: 400, error: 'Not authenticated!' });
        return;
    }

    if(req.method === 'POST') { // Save

        const data = JSON.parse(req.body)
        const domainName = data.domainName;

        let client;
        try {
            client = await connectDatabase();
        } catch (error) {
            res.status(200).json({ ok:true, status: 500, error: 'Connecting to the database failed.' });
            return;
        }

        try {
            const db = client.db();
            const user = await db.collection('users').findOne({
                username: session.user.name
            });
            if(user) {
                const myquery = { username: session.user.name };
                const newvalues = {
                    $set: {
                        domainName: domainName,
                    }
                };
                await db.collection('users').updateOne(myquery, newvalues);
            }
```

```
            client.close();
            res.status(200).json({ ok:true, status: 200 });

        } catch(e) {
            res.status(200).json({ ok:true, status: 500, error: 'Something went wrong.' });
        }
    } else {
        let client;
        try {
            client = await connectDatabase();
        } catch (error) {
            res.status(200).json({ ok:true, status: 500, error: 'Connecting to the database failed.' });
            return;
        }

        try {
            const db = client.db();
            const user = await db.collection('users').findOne({
                username: session.user.name
            });
            if(!user) {
                client.close();
                res.status(200).json({ ok:true, status: 401, error: 'User not found.' });
                return;
            }
            let domainName = '';
            if(user.domainName) domainName = user.domainName;

            client.close();
            res.status(200).json({ ok:true, status: 200, domainName: domainName });

        } catch(e) {
            res.status(500).json({ ok:true, status: 500, error: 'Something went wrong.' });
        }
    }
}

export default handler;
```

Our API route will update **users** collection if receive a POST request. Other than POST, it will read **users** collection and returns user's **domainName**. For the moment, we don't have a domainName entered yet. If domainName is not found, an empty string will be returned.

4. Creating Site Settings Component

1. Create a new file **src/site.js**.



2. Add the following code.

```
import { useState } from "react";
import Notif from './notif';
import Modal from './modal';

function Site() {

    const [modalOpen, setModalOpen] = useState(false);
    const [notif, setNotif] = useState({});

    async function submitHandler(e) {
        e.preventDefault();

        setNotif({
            mode: 'wait',
            text: 'Updating..'
        });
        setTimeout(()=>{
            setNotif({});
        }, 1000);
    }

    return (<>
        <div style={{marginBottom: '30px'}}>
            <button onClick={()=>setModalOpen(true)}>Site Settings</button>
        </div>

        <Modal show={modalOpen} onClose={()=>setModalOpen(false)}>
            <h3>Site Settings</h3>

            <form onSubmit={submitHandler}>
                <div style={{marginTop:20}}>
                    <button>Update</button>
                </div>
            </form>
        </Modal>

        <Notif mode={notif.mode} text={notif.text} />
        </>
    )
}

export default Site;
```

On our site component, we import **Modal** and **Notif** component. Then we prepare two state variables, **modalOpen** and **notif**.

The **modalOpen** value can be true or false. If true, it will be used to show the modal.
As seen, the **modalOpen** variable is set to the **show** property of the **<Modal />** component.
Then on the **Site Settings** button, the **onClick** event will set the **modalOpen** variable to **true**. This will open the modal.

The **notif** variable has an object value, for example:

```
{
    mode: 'wait',
    text: 'Updating..'
}
```

This will show a notification with a text message '**Updating..**'. The mode can be **wait** or **error**, which show different colors.

If the object is empty:

```
{}
```

the notification will not be displayed.

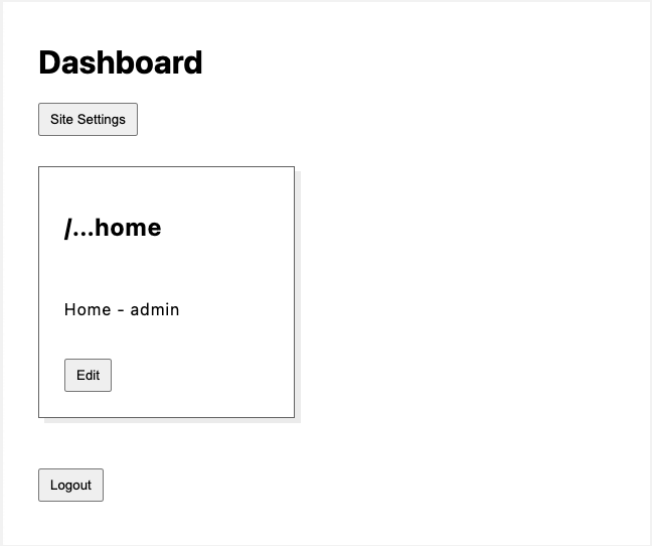
3. Let's use the site component in our dashboard. Open **src/dashboard.js** and add the import.

```
import Site from './site';
```

4. Add the site component under the **<h1>Dashboard</h1>** title.

```
<Site />
```

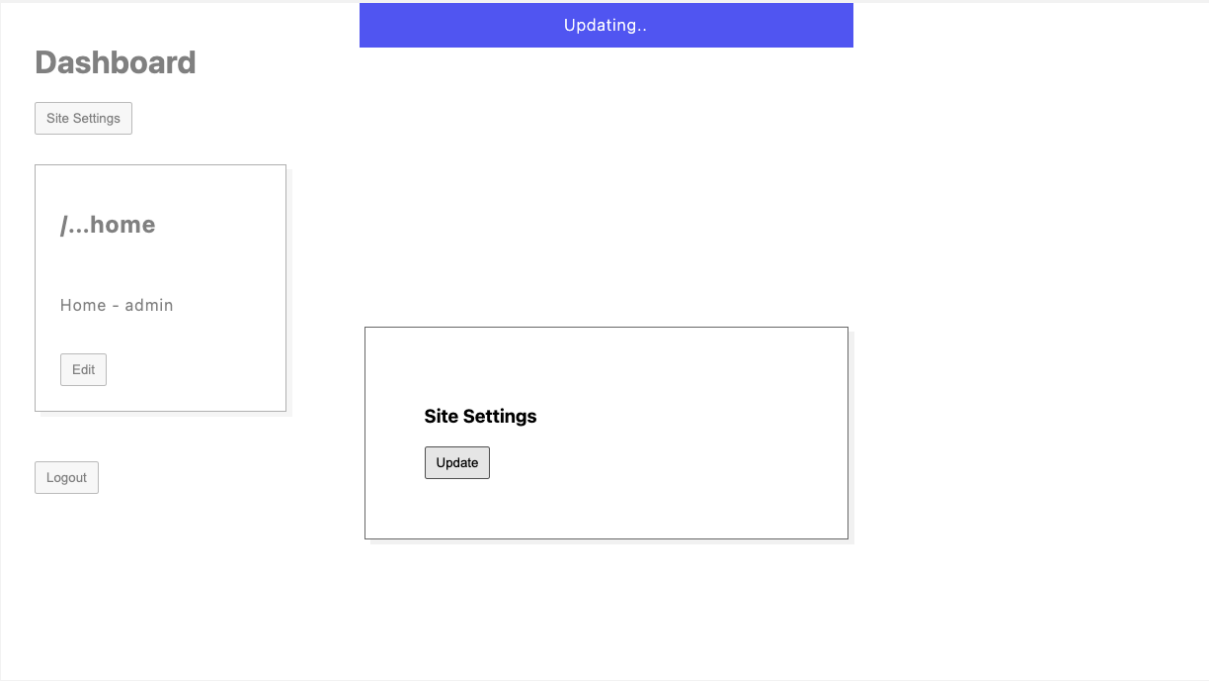
5. Open: **http://localhost:3000/dashboard**.



The **Site Settings** button is displayed under the title.
Click the button to open the modal.

The **Site Settings** modal is displayed.
You can click anywhere outside the modal to close it.

Now click the **Update** button inside the modal.



The notification is displayed.

In our site component, we hide the notification after 1 second, using the **setTimeout** function.

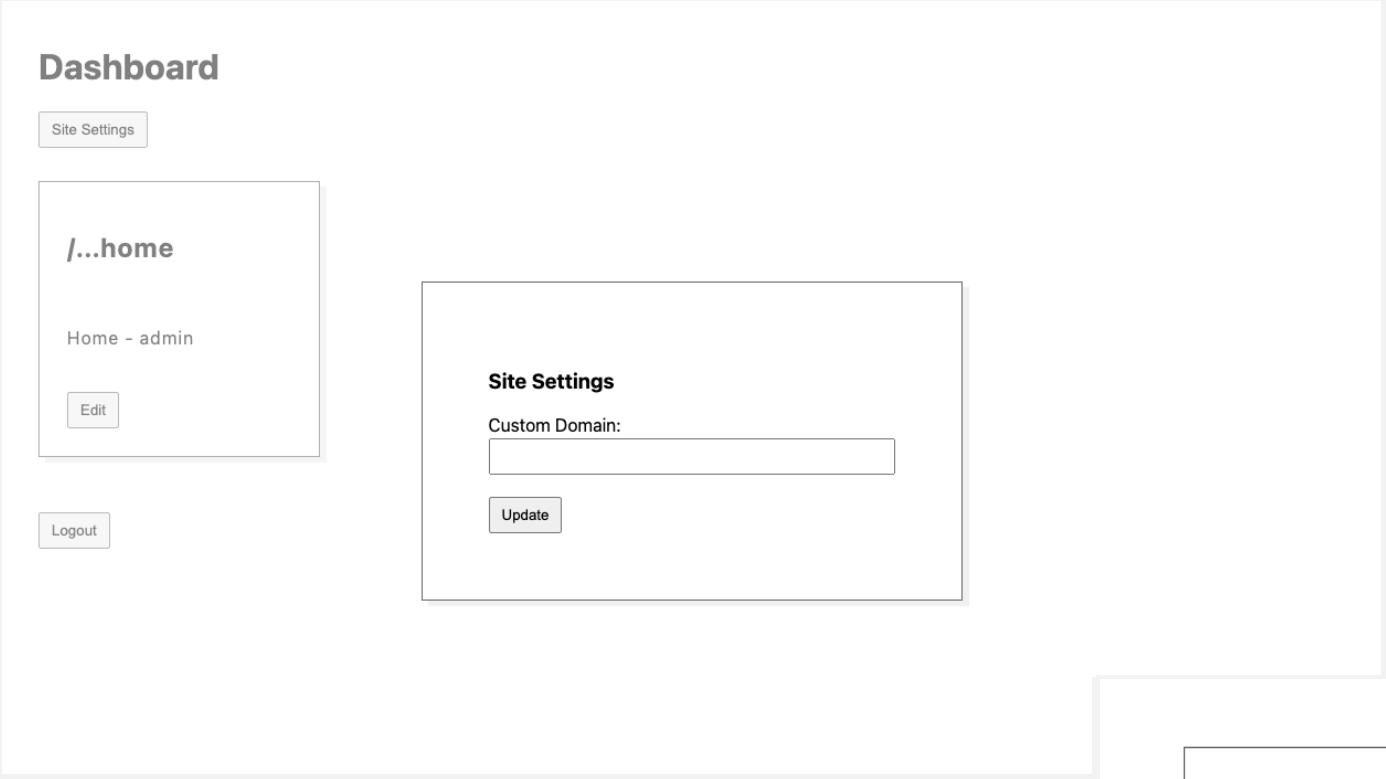
Now you know how to use the modal and notification components. We can then improve our site component to allow users to enter a custom domain name for their site. For saving, we'll use the API route **/api/site** we've created previously.

6. Replace our site component **src/site.js** with the following code.

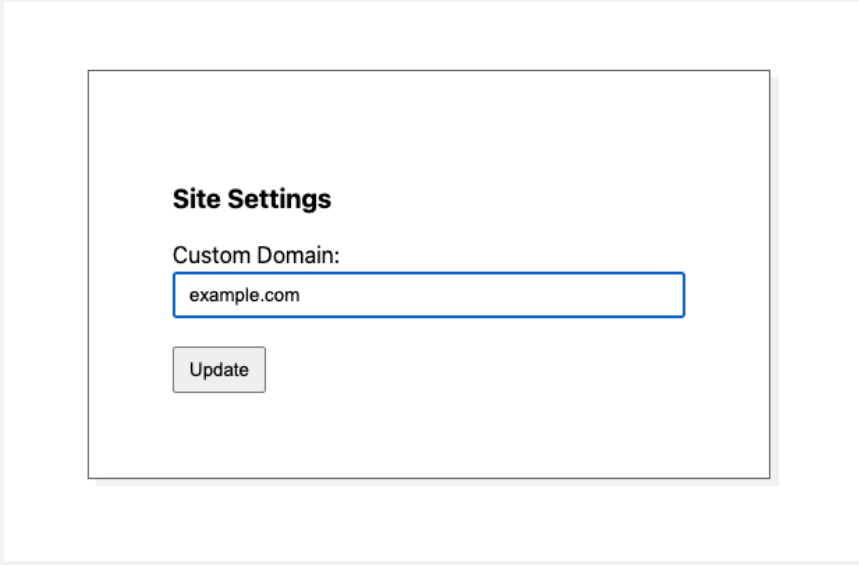
```
import { useState, useEffect } from "react";
import Notif from './notif';
import Modal from './modal';

function Site() {

    const [modalOpen, setModalOpen] = useState(false);
    const [notif, setNotif] = useState({});

    const [domain, setDomain] = useState('');
    const [siteData, setSiteData] = useState(false);

    useEffect(()=>{
        getSiteInfo();
    }, []);

    async function getSiteInfo(){
        let result = await fetch('/api/site');
        result = await result.json(); // {ok: true, status: 200, domainName: 'example.com'}

        setSiteData(true);
        if(result.domainName) setDomain(result.domainName);
    }

    async function submitHandler(e) {
        e.preventDefault();

        setNotif({
            mode: 'wait',
            text: 'Updating..'
        });

        const reqBody = { domainName: domain };
        let result = await fetch('/api/site', {
            method:'POST',
            body: JSON.stringify(reqBody),
            header: {
                'Content-Type': 'application/json'
            }
        });
```

```
        result = await result.json();
        if(!result.error) {
            setNotif({});
            setModalOpen(false);
        } else {
            setNotif({
                mode: 'error',
                text: 'Unable to save.'
            });
            setTimeout(()=>{
                setNotif({});
            }, 2000);
        }
    }

    return (<>
        <div style={{marginBottom: '30px'}}>
            <button onClick={()=>setModalOpen(true)}>Site Settings</button>
        </div>

        <Modal show={modalOpen} onClose={()=>setModalOpen(false)}>
            <h3>Site Settings</h3>

            <form onSubmit={submitHandler}>
                {siteData?  <>
                    <label htmlFor='domainname'>Custom Domain:</label>
                    <input type='text' id="domainname" value={domain} onChange={(e)=>setDomain(e.target.value)} />

                    <div style={{marginTop:20}}>
                        <button>Update</button>
                    </div>
                </> :
                <p>Please wait..</p>}
            </form>
        </Modal>

        <Notif mode={notif.mode} text={notif.text} />
        </>
    )
}

export default Site;
```

7. Open **http://localhost:3000/dashboard** and click the **Site Settings** button.



Now users can enter their own domain name to be used for their site.



13.
Displaying User Site from a Custom Domain

In this chapter, we will make users website accessible from a custom domain.
So far, we have users custom domain stored in **users** collection in our database.

1. Open **.env.development** and add the following line.

```
MAIN_HOST=localhost:3000
```

2. Open **.env.production** and add the following line.

```
MAIN_HOST=sitemagz.com
```

Since we will support multiple custom domains, we will need to specify the main domain here in a variable **MAIN_HOST** to differentiate with user custom domain. The main domain is **owned** by the admin user.
We specify **MAIN_HOST** in **.env.development** for development and **.env.production** for production/deployment.

3. Open terminal and rerun the project.

```
$ npm run dev
```

4. Open **pages/[...slug].js** and replace the following code inside the **getServerSideProps()** function:

```
  let site = await db.collection('users').findOne({
    username: siteOwner
  });
  if(!site) {
    return {
      redirect: { permanent: false, destination: '/notfound' }, props:{},
    };
  }
```

with the following:

```
// Check requested domain name
let host = '';
const { req } = context;
if (req) host = req.headers.host; //yourdomain.com

let isMainHost = false;
if(host === process.env.MAIN_HOST) isMainHost = true;

let site;
if(isMainHost) {
  site = await db.collection('users').findOne({
    username: siteOwner
  });
  if(!site) {
    return {
      redirect: { permanent: false, destination: '/notfound' }, props:{},
    };
  }
} else {
  // Check requested user's custom domain
  site = await db.collection('users').findOne({
    domainName: host.toLowerCase()
  });
  if(site) {
    siteOwner = site.username;
    if(part1) slug = part1;
    if(part2) {
      // Invalid Request (only part1 is valid for user page from a custom domain)
      invalidRequest = true;
    }
  } else {
    return {
      props:{
        invalidDomain: true
      },
    };
  }
}
```

```
let client = await connectDatabase();
const db = client.db();

// Check requested domain name
let host = '';
const { req } = context;
if (req) host = req.headers.host; //yourdomain.com

let isMainHost = false;
if(host === process.env.MAIN_HOST) isMainHost = true;

let site;
if(isMainHost) {
  site = await db.collection('users').findOne({
    username: siteOwner
  });
  if(!site) {
    return {
      redirect: { permanent: false, destination: '/notfound' }, props:{},
    };
  }
} else {
  // Check requested user's custom domain
  site = await db.collection('users').findOne({
    domainName: host.toLowerCase()
  });
  if(site) {
    siteOwner = site.username;
    if(part1) slug = part1;
    if(part2) {
      // Invalid Request (only part1 is valid for user page from a custom domain)
      invalidRequest = true;
    }
  } else {
    return {
      props:{
        invalidDomain: true
      },
    };
  }
}

let page;
if(siteOwner) {
  // Get the requested home page
  if(!invalidRequest) {
```

Here we check from which domain the page is requested and then it search into the users collection to find whose site it must be returned.

To get the **MAIN_HOST** variable, we use:
process.env.MAIN_HOST

5. Then add the following block inside the **Home()** function to catch an invalid domain request.

```
import { connectDatabase } from '../src/db';
import Head from 'next/head';

function Home(props) {
    if(props.invalidDomain) {
        return <>
            <p>Invalid Domain</p>
            <p>{props.host}</p>
        </>;
    } else if(props.notFound) {
        return <>
            <Head>
                <title>404</title>
                <link rel="icon" href="/favicon.ico" />
            </Head>
            <h1>404</h1>
            <p>Oops!</p>
        </>
    } else {

        ...
```

5. Copy the complete code from **pages/[...slug].js** into **pages/index.js**. That's it! Now users site can be accessed from their custom domain.

6. To test, you must push your project into Github. The automatic deployment process will run to update your live site. Then login as a user and add a custom domain from the Site Settings.
Note that the domain name A-record needs to be pointed to your VPS IP as explained in part 1 of this guide. That's all needed.

14. The Complete Project

All the important concepts of the project have been described through the steps. Now it's time to run the complete project (provided in the package) to have more complete functions, such as asset manager, adding favicon, updating user account, etc.

You can just replace these folders:
- pages
- public
- src
- styles

Run the project locally first before pushing to your Github. On the first start, an admin user account will be automatically created. You can login using:
Email: you@example.com
Password: demo
You can change your email and password after logging in.