10
Caching with Redis and Deployment on Ubuntu (DigitalOcean) and Netlify

In this chapter, we are going to explore yet another deployment setup – a robust Uvicorn/Gunicorn/Nginx solution that has been tried and tested with Django and other WSGIs but also ASGI web applications. This should give you more than enough choices when starting your next FARM stack project. We will also add a simple caching solution with Redis, relieving MongoDB from some requests that could (and should!) be cached and served directly. Finally, we will deploy our React-based frontend on Netlify, another very popular deployment option, whose simplicity matches its flexibility.

In this chapter, we will cover the following topics:

- Creating an account on DigitalOcean (optional)
- Preparing our Ubuntu server with Nginx
- Deployment of a FastAPI instance through Uvicorn, Gunicorn, and Nginx
- Caching with Redis
- Creating a free account on Netlify
- Deployment of the React Frontend on Netlify

By the end of this chapter, you should feel confident when it comes to deploying FARM stack-based applications on a variety of serving platforms, including a bare-bones Ubuntu (or any Linux) server. You will be able to recognize where and how to add caching and implement it effortlessly with Redis. Finally, with the knowledge of possible deployment solutions, you will be able to make solid decisions when the time comes to deploy your application.

Deploying FastAPI on DigitalOcean (or really any Linux server!)

In this section, we are going to take our simple analytics application and deploy it on a Ubuntu server on *DigitalOcean* (`www.digitalocean.com`) as an **Asynchronous Server Gateway Interface** (**ASGI**) application. We are going to end up with a pretty robust and customizable setup that includes our development web server – Uvicorn – but also Gunicorn (`https://gunicorn.org`), an excellent and robust web server that plays very nicely with *Nginx*, and a virtual machine running Ubuntu – a *DigitalOcean* droplet. Though in this example we are going to use DigitalOcean, the procedure should apply to any Debian or Ubuntu-based setup; you can try it out on your own machine running Ubuntu. The following instructions rely heavily on the excellent tutorials on setting up an Ubuntu server on *DigitalOcean* by Brian Boucheron (`https://www.digitalocean.com/community/tutorials/initial-server-setup-with-ubuntu-20-04`) and on deploying an Async Django application by Mason Egger and Erin Glass (`https://www.digitalocean.com/community/tutorials/how-to-set-up-an-asgi-django-app-with-postgres-nginx-and-uvicorn-on-ubuntu-20-04`). You should read them as they are very useful and well written!

Important Note

In this section, we will make heavy use of SSH – the Secure Shell Protocol. SSH is a cryptographic protocol developed for accessing secure network services over insecure networks. If that doesn't make much sense, do not worry – there are plenty of great resources on basic SSH operations on the internet. If you are willing to dive a bit deeper into DevOps, you can read the following book: `https://www.amazon.com/Mastering-Ubuntu-Server-configuring-troubleshooting/dp/1800564643`. *Mastering Ubuntu Server* is an excellent guide on the subject. In the following pages, we will just log into a DigitalOcean droplet, which is nothing more than a remote Ubuntu computer that we will be able to control. While I will show the procedure for deploying a fully functional FastAPI instance on a DigitalOcean droplet, the best way to try out this procedure would be to practice on your Ubuntu-based server. If you have a spare box (even an older one), install Ubuntu and try connecting to it from your main computer.

The deployment procedure will be broken into easy steps.

DigitalOcean is one of the leaders in providing cloud computing and **Infrastructure as a Service** (**IaaS**). Users can benefit from different types of virtual machines that can be modeled according to our needs. In our case, we just want a solution for hosting our FastAPI server, similar to how we did with Heroku in the previous chapters.

While *DigitalOcean* doesn't provide a completely free tier, it is reasonably cheap to get started (around 4 USD per month). It has a flexible and scalable system where you can easily scale up or down according to your needs and it offers complete control of the virtual machines – droplets, a fact that brings us a whole new level of flexibility, a word that we often used in this book. Another advantage of *DigitalOcean* is its excellent community and an endless list of well-written articles on any service or setup you may want to achieve, so it represents a good place to start if you are entering the world of deployment, database setup, and so on. Just to be clear, *DigitalOcean*, as well as its competitors (Linode, for instance) is perfectly able to host our complete full-stack setup – we could install MongoDB on the server as well, add Node.js and Next or a React frontend, and orchestrate everything through Nginx, a powerful and fast server. In this example, however, we only want to serve our FastAPI instance and showcase a different type of deployment. Follow these steps:

1. Create an account on DigitalOcean! Head over to the DigitalOcean signup page at `https://cloud.digitalocean.com/registrations/new` and fill in your data. You can sign in with GitHub or Google if you wish, and you can use a referral code if you have one so that you can try out the service for a determined time. Once you submit your data (and once you have some credit to spend – be it from a referral program or after you connect your credit card), you will be able to create your first droplet.
2. Create a droplet. I have used a Ubuntu 22.04 x64 Ubuntu distribution, the plan is Basic (the cheapest), and the CPU options are $4/month with 512 MB/1 CPU (you will have to hit the left arrow to find this plan!). Since I am in Europe, I selected the Frankfurt data center region. Finally, to simplify things, I opted for password authentication, so I entered a root password (that I am not going to disclose here!). I gave the hostname a name – `farmstack`. Although we will be using the IP address to access this brand-new machine through SSH, it is useful to have a user-friendly machine name.

   Give DigitalOcean some time to prepare your droplet. After about 30 seconds, you will be able to click on the lefthand menu under **Droplets** and you will be taken to a page that displays information about your droplet. You now have a Ubuntu-based server under your control!
3. To verify that you are indeed able to log in as root on your brand-new machine, click on the IP address of the droplet to copy it, open **Cmder** (or whatever shell you have been using this whole time) on Windows or a bash/shell if you are on Linux or macOS, and try to access the droplet:

   ```
   ssh root@<your_IP_address_that_you_just_copied>
   ```
4. *Cmder* will kindly inform you that the authenticity of the host cannot be established, which is normal at this stage, and ask you if you want to continue connecting. Type `yes`; you will be greeted with a shell that should read as follows:

   ```
   root@farmstack:~#
   ```
5. It is good practice to create a new user account that will have all the necessary privileges so that we don't use the root account for our web hosting. Let's create an account called `farmuser`:

   ```
   adduser farmuser
   ```

6. You will be asked to provide a password (twice), a name, and some other information, such as a room number (!). It is important to remember the password! This newly created user will need to be able to perform various administrative tasks, so we should grant them adequate privileges. In the same SSH session, type the following:

   ```
   usermod -aG sudo farmuser
   ```

   After this, when we log in as `farmuser`, we will be able to just type `sudo` before performing actions that require superuser powers.

   We will make use of the *UFW* firewall to make sure that only certain types of connections to our server are permitted. There are different options when it comes to DigitalOcean's firewalls, but this should be more than enough and easy to set up on different machines. Things may get tricky, though – we need to make sure that when we leave our SSH root shell, we *will* be able to get back in with our `farmuser` account!

7. To be sure that OpenSSH is allowed to access the machine, type the following:

   ```
   ufw allow OpenSSH
   ```

8. You should see a message saying **rules updated**. Now, let's enable `ufw` and check its status:

   ```
   ufw enable
   ufw status
   ```

   The preceding commands should warn you that they may disrupt existing *SSH* connections; confirm the first one anyway. The second should just inform you that the service is active.

9. Great. Now, **Keep the SSH session alive** and open a new terminal so that we can test our connection with our regular yet highly privileged `farmuser`:

   ```
   ssh farmuser@<your_IP_address_that_you_just_copied>
   ```

   You should be greeted with a prompt; that is, `farmuser@farmstack:~$`. That's great – now, we can proceed with this (regular) user and use `sudo` when we need to do tricky stuff!

   It is time to update our Ubuntu packages and add some more. Logged in as `farmuser` (or whatever your regular, non-root username was), issue the following command:

   ```
   sudo apt update
   sudo apt install python3-venv nginx curl
   ```

   `sudo` will prompt you for a password – your regular `farmuser` password – so kindly provide it. Apart from Python 3, we are installing `Nginx`, our powerful web server and reverse proxy solution, and `curl` (to test our API service locally).

Now, we are entering the second, project-related phase of our deployment. It is time to create a virtual environment, just like we did numerous times during the development phase. This is a bare-bones server, so we have to do everything manually. There is no helpful guiding hand like there was with *Heroku* or *Vercel*. Follow these steps:

1. Let's create a directory called `apiserver` in our home folder and `cd` into it (you can always see where you are currently located with PWD!):

   ```
   mkdir ~/apiserver
   cd ~/apiserver
   ```

2. Now, let's create a Python 3 environment:

   ```
   python3 -m venv venv
   ```

3. After the setup has finished, go ahead and activate this environment with the following command:

   ```
   source venv/bin/activate
   ```

   You should see `venv` prepending the command prompt.

4. It is time to grab the address of the GitHub repository that you created for the backend and change the directory to our `/apiserver`. Now, clone the GitHub repo inside to get all the code:

   ```
   git clone <your repo address>
   ```

   This will create a folder with the same name as the repository – in my case, it is a bit cumbersome: `FARM-chapter9-backend`. Cloning the code from the repo will not copy the `.env` file with the necessary keys for MongoDB and Sendgrid (and Cloudinary in the previous app).

5. Although we could set the environment variables manually through the shell, we are just going to blatantly copy them using the secure copy `scp` command. Make sure you're in your local computer's `/backend` folder and take note of the remote folder. Then, issue the following command:

   ```
   scp .env farmuser@207.154.254.114:~/apiserver/FARM-
   chapter9-backend
   ```

6. Now try out the `ls` command to make sure that the folder with the code is indeed there, but keep in mind that the `.env` file will not be shown! You will have to use something such as nano `.env` to verify that the file is indeed there and that it contains the necessary information. If you don't want to mess with `scp`, you can just create and type in the `.env` file using nano – the powerful command-line text editor provided by Linux systems.

7. Once the code is in the Ubuntu droplet, `cd` into the directory and install all the dependencies with the following command:

   ```
   pip install -r requirements.txt
   ```

Important Note

After committing the code for the backend, you should update the `requirements.txt` file by typing `pip freeze > requirements.txt`, while being within the activated virtual environment on your local machine. This file should then be committed to GitHub – it will be our magic ingredient for recreating the same virtual environment on other machines, including our droplet!

8. Once the dependencies have been installed, we can test our application with the standard Uvicorn command:

   ```
   uvicorn main:app -reload
   ```

   The prompt should inform you that Uvicorn is running on `http://127.0.0.1:8000` but that we cannot access it yet from the outside.

9. Stop the server with *Ctrl* + *C*. To be able to test that the API is working, we have to disable our UFW firewall. To do that, you have to `sudo` your way through it:

   ```
   sudo ufw disable
   ```

Notice

This is a dangerous practice – a bit like leaving your front door open.

Now, if you try to rerun the *Uvicorn* server, you should be able to access your API with a REST client or a browser at the IP address of your droplet, on port `8000`! So far, we are only trying out what we have been doing throughout this book on DigitalOcean. Now, it is time to introduce Gunicorn.

Important Note

Gunicorn is a mature and battle-tested WSGI Python server for UNIX. It is often used in conjunction with Uvicorn since it is highly configurable and able to handle Uvicorn workers efficiently. The Uvicorn documentation itself recommends a setup that includes Gunicorn and Nginx and that is exactly what we are going to do! Gunicorn is an interesting and powerful project in its own right and its documentation is is a useful read (`https://gunicorn.org/`).

Let's build our deployment now. Follow these steps:

1. Install `gunicorn` with a simple call to `pip`:

   ```
   pip install gunicorn
   ```

2. After installing `gunicorn`, we can start our API server with the following command (while staying in the source code directory!):

   ```
   gunicorn --bind 0.0.0.0:8000 main:app -w 4 -k uvicorn.
   workers.UvicornWorker
   ```

   The preceding command starts a *gunicorn* server with four *uvicorn* workers. Gunicorn provides also load balancing functionality for our Uvicorn servers – an async request that might be taking a bit too long won't hog up the system. Now, we can test our app on port `8000`.

   Now, we are going to use Linux's powerful `systemd` service and socket files to make the server start and stop programmatically.

Important Note

systemd is a process and system manager for Linux systems. If you wish to get to know its capabilities and functionalities, I can recommend (another) very useful article from the DigitalOcean knowledge database: `https://www.digitalocean.com/community/tutorials/systemd-essentials-working-with-services-units-and-the-journal`. Again, in these pages, we will only explain the commands that we will be using – starting, stopping, and enabling and disabling services, servers, and so on.

3. We are going to have to use a bit of `nano`, the command-line text editor of choice for the majority of Linux distributions. Stop the `gunicorn` server with *Crtl* + *C* and deactivate the virtual environment with a simple `deactivate`. The prepended `venv` should be gone.
4. Now, let's create a `gunicorn` socket. Sockets are simply communication points on the same or different computers that enable systems to exchange data. When we create a Gunicorn socket, it is just a way of telling the system that the created socket can be used to access data that the server will provide:

   ```
   sudo nano /etc/systemd/system/gunicorn.socket
   ```

   The file's content should be as follows (fully adapted from the aforementioned ASGI Django guide):

   ```
   [Unit]
   Description=gunicorn socket

   [Socket]
   ListenStream=/run/gunicorn.sock

   [Install]
   WantedBy=sockets.target
   ```

5. To leave nano, just type *Ctrl* + *X* and type `yes` when asked to confirm. The filename should remain the same as what we gave it initially.
6. Now, we are going to create the `gunicorn.service` file. Again, fire up *nano* with the following command:

```
sudo nano /etc/systemd/system/gunicorn.service
```

7. Begin typing the following:

```
[Unit]
Description=gunicorn daemon
Requires=gunicorn.socket
After=network.target

[Service]
User=farmuser
Group=www-data
WorkingDirectory=/home/farmuser/apiserver/FARM-chapter9-
backend
ExecStart=/home/farmuser/apiserver/venv/bin/gunicorn \
          --access-logfile - \
          -k uvicorn.workers.UvicornWorker \
          --workers 3 \
          --bind unix:/run/gunicorn.sock \
          main:app

[Install]
WantedBy=multi-user.target
```

I have highlighted the essential parts and paths that you should triple-check before saving. It is important to emphasize that the working directory is the directory *hosting our code*, while `execstart` is referring to the `virtualenv` directory. In our case, they are side by side inside the `apiserver` folder! This should be enough for `systemd`.

8. Save the file and let's try it out. Start and enable the newly created `gunicorn` socket with the following commands:

```
sudo systemctl start gunicorn.socket
sudo systemctl enable gunicorn.socket
```

9. If everything went right, there shouldn't be any errors. You should, however, check the status of the socket:

```
sudo systemctl status gunicorn.socket
```

10. You should also check for the existence of the `gunicorn.sock` file:

```
file /run/gunicorn.sock
```

11. Now, activate the socket:

```
sudo systemctl status gunicorn
```

12. With that, we should be able to (finally!) test our API with `curl`:

```
curl --unix-socket /run/gunicorn.sock localhost/cars/all
```

You should get a bunch of cars flooding the terminal since we've hit our `cars` endpoint!

We're nearly there, hang on! Now, we will use Nginx to route the incoming traffic. Follow these steps:

Important Note

Nginx is an extremely powerful, reliable, and fast web server, load balancer, and proxy server. At its most basic, Nginx reads its configuration and, based on this information, decides what to do with each request that it encounters – it can simultaneously handle multiple websites, multiple processes, and the most diverse configurations that you throw at it. You may have a bunch of static files, images, and documents in one location on the server, a Node.js API managed by PM2, a Django or Flask website, and maybe a FastAPI instance all at once. With the proper configuration, Nginx will be able to effortlessly take care of this mess and always serve the right resource to the right client. At least some basic knowledge of how Nginx operates can be a very useful tool to have under your belt, and the `nginx.org` website is a great place to start.

13. Nginx operates in server blocks, so let's create one for our `apiserver`:

```
server {
    listen 80;
    server_name <your droplet's IP address>
    location = /favicon.ico { access_log off; log_not_
found off; }

    location / {
        include proxy_params;
        proxy_pass http://unix:/run/gunicorn.sock;
        }
    }
```

Once you get used to Nginx's server block syntax, you will be serving websites (or processes, to be precise) in no time. In the preceding code, we instructed Nginx to listen on the default port (`80`) for our machine (IP address) and to redirect all traffic to our Unix Gunicorn socket!

14. Now, enable the file by copying it to the `sites-enabled` folder of Nginx, as follows:

```
sudo ln -s /etc/nginx/sites-available/myproject /etc/
nginx/sites-enabled
```

There is a very handy command that allows us to check if the Nginx configuration is valid:

```
sudo nginx -t
```

15. If Nginx is not complaining, we can restart it by typing the following command; then, we should be good to go:

```
sudo systemctl restart nginx
```

16. The last thing we must do is set up the `ufw` firewall again, allow Nginx to pass through, and close port `8000` by removing the rule that allowed it:

```
sudo ufw delete allow 8000
sudo ufw allow 'Nginx Full'
```

Congratulations! You are now serving your API through a robust setup that consists of Uvicorn, Gunicorn, and Nginx. With this setup, we have a plethora of options. You could serve static files (images, stylesheets, or documents) blazingly fast through Nginx. You could also set up a Next.js project and manage it through PM2 (`https://pm2.keymetrics.io/`), a powerful Node.js process manager. We will stop here, although there are many – not so complicated – steps to go through before we have a production-ready system.

Adding caching with Redis

Redis is among the top technologies when it comes to NoSQL data storage options, and it is very different from MongoDB. Redis is an in-memory data structure store, and it can be used as a database, cache, message broker, and also for streaming. Redis provides simple data structures – hashes, lists, strings, sets, and more –and enables scripting with the Lua language. While it can be used as a primary data store, it is often used for caching or running analytics and similar tasks. Since it is built to be incredibly fast (much faster than MongoDB, to be clear), it is ideal for caching database or data store queries, results of complex computations, API calls, and managing the session state. MongoDB, on the other hand, while being fast and flexible, if it scales sufficiently, could slow down a bit. Bearing in mind that we often (as is the case in this chapter) host MongoDB on one server (Atlas Cloud) and our FastAPI code on another one (DigitalOcean or Heroku), latency also might affect the response times. Imagine if we wanted to perform a complex aggregation instead of the simple ones that we have created in this chapter. By throwing in some data science, such as algorithms with interpolations or machine learning algorithms, we could be in trouble should our website become popular (and it will!).

Caching to the rescue! What is caching? It is a really simple concept that has been around for decades – the basic idea is to store some frequently requested data (from a Mongo database, in our case) in some type of temporary storage for some time until it expires. The first user requesting said resource (a list of cars) will have to wait for the whole query to complete and will get the results. These results will then automatically be added to this temporary storage (in our case, Redis, the Usain Bolt of databases) and served to all subsequent requests for the same data. By the same data, we usually imply the same endpoint. This process persists until the data stored in Redis (or any other caching solution that you may use) expires – if valid data is not found in the cache, the real database call is made again and the process repeats.

The expiry time is of crucial importance here – in our case, if we are working with a car-selling company, we can be generous with caching and extend the expiry period to 10 minutes or even more. In more dynamic applications, such as forums or similar conversational environments, a much lower expiry time would be mandatory to preserve functionality.

Installing Redis on Linux is quite simple, while on Windows it is not officially supported. You could follow the official guide for installing Redis for development purposes on Windows (`https://redis.io/docs/getting-started/installation/install-redis-on-windows/`) but that is beyond the scope of our application. We will, however, install Redis on our DigitalOcean Linux box and add caching to our FastAPI application!

Connect to your DigitalOcean box (or to your Linux system of choice – if you are developing on Linux or Mac, you should install it there as well) by following the steps from this chapter, while using SSH from a terminal:

1. Now, install Redis by typing the following command:

   ```
   sudo apt install redis-server
   ```

Important Note

In a production environment, you should secure your Redis server with a disgustingly long password. Since Redis is fast, an attacker could potentially run hundreds of thousands of passwords in mere seconds against it during a brute-force attack. You should also disable or rename some potentially dangerous Redis commands. In these pages, we are only showing how to add a bare-bones, not-secured Redis instance to our setup.

2. Now, we should restart the Redis service. Although it should happen automatically, let's make sure by typing the following command:

   ```
   sudo systemctl restart redis.service
   ```

3. Test it by typing the following command to see if it is working:

   ```
   sudo systemctl status redis
   ```

   The Terminal will send an ample response, but what you are looking for is the green word **Active** (running). It should also be started automatically with every reboot – so we get that going for us, which is nice.

4. The traditional way to test that Redis is responding is to start the client:

   ```
   redis-cli
   ```

   Then, in the Redis shell, type `ping`.

Redis should respond with `pong` and the prompt should say **127.0.0.1:6379**. This means that Redis is running on localhost (the Linux server) on port `6379`. Remember this address, or better, write it down somewhere (I know, I know). We are going to need it for our FastAPI server.

There are many ways to make Redis talk to Python, but here, we will opt for a simple module aptly named *Fastapi-cache* (`https://github.com/long2ice/fastapi-cache`). Now, we will have to edit our backend code in the `/backend` folder. When we're done, we will push the changes to GitHub and repeat the deployment procedure. Or, if you just want to quickly try out the caching, you could edit the files on *DigitalOcean* directly by navigating to the directory and using *nano*.

Anyway, activate the virtual environment of your choice and install the package and *aioredis* (the async Python Redis driver):

```
pip install fastapi-cache2 aioredis
```

Now, our FastAPI project structure dictates which files need to be updated. We need to update our `main.py` file and add the following imports:

```
import aioredis
from fastapi_cache import FastAPICache
from fastapi_cache.backends.redis import RedisBackend
```

Then, we need to update our startup event handler:

```
@app.on_event("startup")
async def startup_db_client():
    app.mongodb_client = AsyncIOMotorClient(DB_URL)
```

```
    app.mongodb = app.mongodb_client[DB_NAME]
    redis = aioredis.from_url(
        "redis://localhost:6379", encoding="utf8", decode_
            responses=True
    )
    FastAPICache.init(RedisBackend(redis), prefix="fastapi-
        cache")
```

The code makes sense – we're getting a Redis client, just like we did with Mongo, and we are passing the URL and a couple of (suggested) settings. Finally, we initialized the *FastAPICache*. Now, we need to add the caching decorator to our endpoints, which are located in the `/routers/cars.py` file. We will add one import:

```
from fastapi_cache.decorator import cache
```

Now, we can decorate the routes that we wish to cache (only GET requests, but that's all we have in this project really). Edit the `/sample` route:

```
@router.get("/sample/{n}", response_description="Sample of N
cars")
@cache(expire=60)
async def get_sample(n: int, request: Request):
    query = [
        {"$match": {"year": {"$gt": 2010}}},
        {
            "$project": {"_id": 0,}
        },
        {"$sample": {"size": n}},
        {"$sort": {"brand": 1, "make": 1, "year": 1}},
    ]
    full_query = request.app.mongodb["cars"].aggregate(query)
    results = [el async for el in full_query]
    return results
```

This route is now cached, which means that when it's hit, it will provide a sample of size *N* and then, for all subsequent requests in the next 60 seconds, it will send the same cached response. Go ahead and try it out, either on your DigitalOcean API or local environment, depending on where you implemented caching. Try *hitting* the API for 1 minute – you should always get the same result until the cache expires. Congratulations – you have just added a top-of-the-class caching solution to your API!

Deploying the Frontend on Netlify

Similar to Vercel, Netlify is one of the top companies providing services for static web hosting and serverless computing, but also a rather simple CMS and goodies such as form handling. It is widely regarded as one of the best solutions for hosting *JAMStack* websites and its **content delivery network** (**CDN**) can speed up the hosted websites significantly. It is also one of the easiest ways to host a React application. This is what we are going to use it for in this section.

After logging in with your Google or GitHub account, you will be presented with a screen that offers you the possibility to deploy a new project:

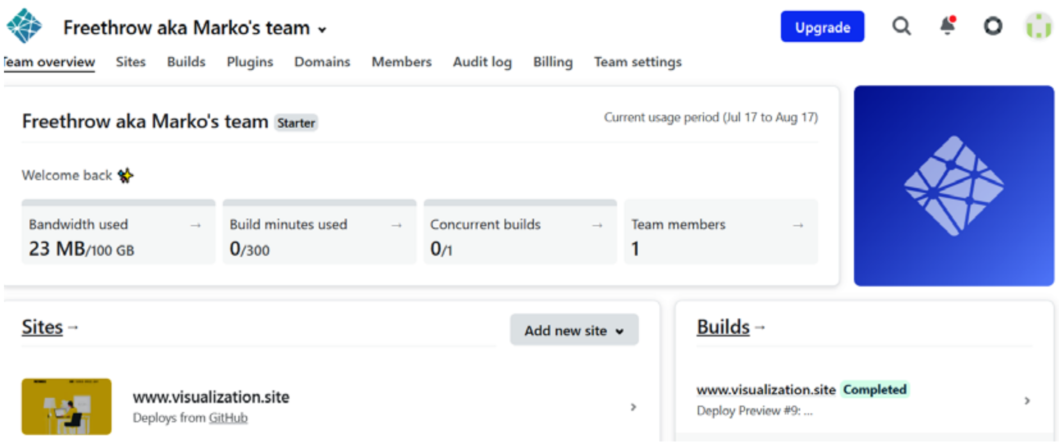


Figure 10.1 – The Netlify Add New Site button

Next, you will be asked whether you are importing an existing project (yes!); you should choose your React frontend project from GitHub. If you logged in with GitHub, you won't have to authorize Netlify again – if not, please authorize it:

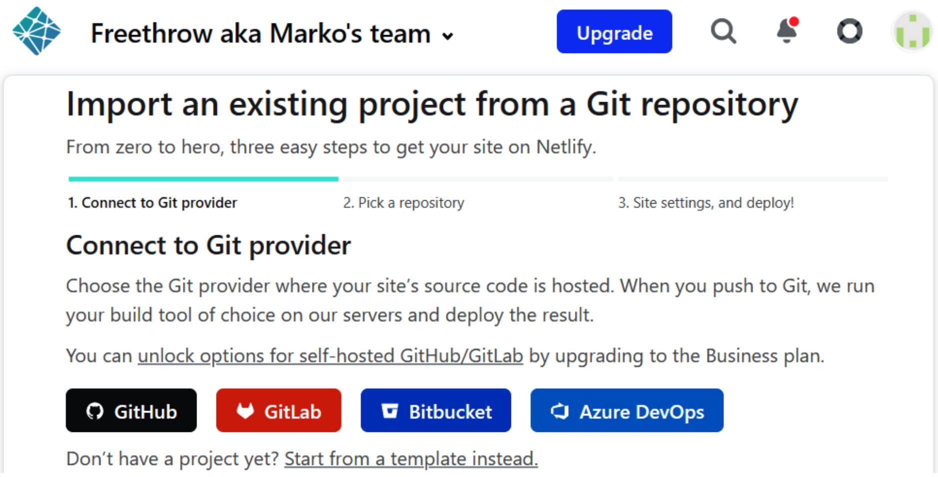
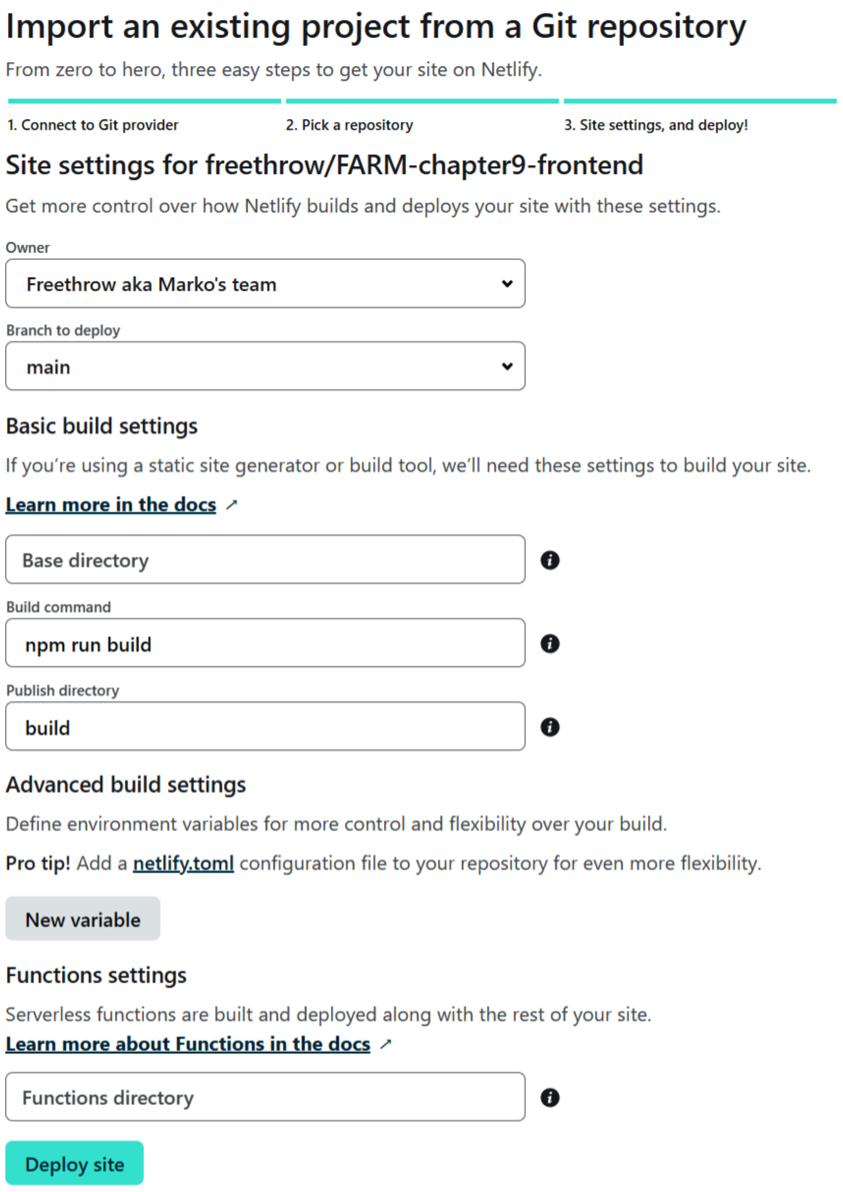

Freethrow aka Marko's team
Upgrade
Import an existing project from a Git repository
From zero to hero, three easy steps to get your site on Netlify.
1. Connect to Git provider
2. Pick a repository
3. Site settings, and deploy!
Connect to Git provider
Choose the Git provider where your site's source code is hosted. When you push to Git, we run your build tool of choice on our servers and deploy the result.
You can unlock options for self-hosted GitHub/GitLab by upgrading to the Business plan.
GitHub
GitLab
Bitbucket
Azure DevOps
Don't have a project yet? Start from a template instead.

Figure 10.2 – The Import and existing project page on Netlify

After browsing through your GitHub projects, point Netlify to the React frontend and leave all the defaults that Netlify was able to cleverly infer from the project. You will be presented with a page on which you could potentially modify any deployment setting, but we will limit ourselves to just adding a single environment variable. You've guessed it – it's the handy `REACT_APP_API_URL`!



Import an existing project from a Git repository

From zero to hero, three easy steps to get your site on Netlify.

1. Connect to Git provider
2. Pick a repository
3. Site settings, and deploy!

Site settings for freethrow/FARM-chapter9-frontend

Get more control over how Netlify builds and deploys your site with these settings.

Owner: Freethrow aka Marko's team

Branch to deploy: main

Basic build settings

If you're using a static site generator or build tool, we'll need these settings to build your site.

Learn more in the docs ↗

Base directory

Build command: npm run build

Publish directory: build

Advanced build settings

Define environment variables for more control and flexibility over your build.

Pro tip! Add a netlify.toml configuration file to your repository for even more flexibility.

New variable

Functions settings

Serverless functions are built and deployed along with the rest of your site.

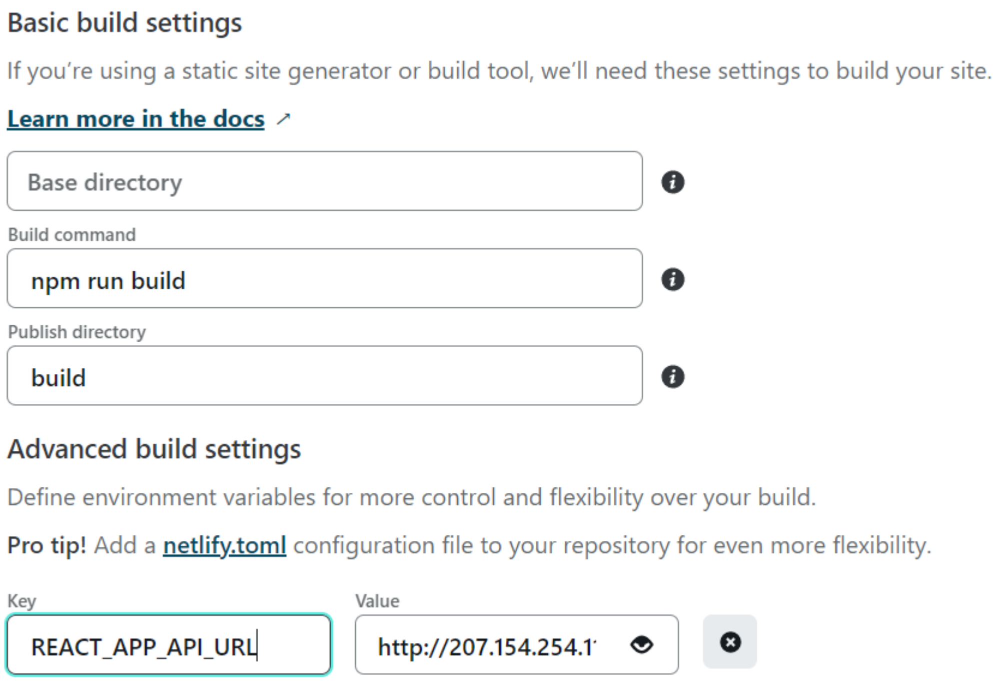
Learn more about Functions in the docs ↗

Functions directory

Deploy site

Figure 10.3 – Netlify's pre-deployment setting page

You will have to add just one variable in the advanced settings: you've guessed it – `REACT_APP_API_URL`. Create a **New variable** by hitting the respective button and name it `REACT_APP_API_URL`. The value should be `https://yourdomain.com`:


Basic build settings

If you're using a static site generator or build tool, we'll need these settings to build your site.

Learn more in the docs

Base directory

Build command: npm run build

Publish directory: build

Advanced build settings

Define environment variables for more control and flexibility over your build.

Pro tip! Add a netlify.toml configuration file to your repository for even more flexibility.

Key	Value
REACT_APP_API_URL	http://207.154.254.1

Figure 10.4 – Adding the new environment variable in Netlify

After some time, maybe a minute or so, you will have your deployment ready for the world to see! In case of any problems (and there will be problems), you should inspect Netlify's deployment console and watch for hiccups.

Your React frontend with all its fancy charts and fast pagination will now be served from Netlify's fast **content delivery network** (**CDN**) while operating on a FastAPI (cached) backend served by Nginx on DigitalOcean. Throw in our previously explored Heroku and Vercel deployments and you have a lot of options to start tinkering!

This doesn't mean that these are your only deployment options! A popular and rock-solid choice is to use a Docker container and containerize your application (together or separately) and provide this Docker image to some of the giants – **Amazon Web Services** (**AWS**), Microsoft Azure, or Google App Engine. This type of deployment isn't much different from the Heroku deployment, although it requires creating the proper type of account and setting the environment the right way. These solutions also tend to have higher upfront costs.

Summary

In this chapter, we added a very simple yet powerful caching solution based on Redis – an incredibly powerful product in its own right. We went through the tedious but often necessary procedure of hosting the API on a Ubuntu server behind Gunicorn and the mighty Nginx – a server that offers so much flexibility and configurability that it simply has to be put in the conversation of the FARM stack. As a bonus, we explored yet another cheap (well, free) frontend hosting option – Netlify – which offers premiere continuous deployment and plays very nicely with all our frontend solutions, be it plain React or Next.js or maybe, in the future, React-Remix. Now, you should feel confident enough to dive head-first into your next project and peruse the numerous options that FastAPI, React, and MongoDB have to offer by playing nicely with each other.

In the next chapter, we will try to address some of the best practices that pertain to the components of the stack in every project, as well as some topics that we haven't touched on but are equally important, such as testing, using static templates with Jinja2, site monitoring, and more.