

```
{() => fs}
```



## b Deploying app to internet

Next, let's connect the frontend we made in part 2 to our own backend.

In the previous part, the frontend could ask for the list of notes from the json-server we had as a backend, from the address <http://localhost:3001/notes>. Our backend has a slightly different URL structure now, as the notes can be found at <http://localhost:3001/api/notes>. Let's change the attribute **baseUrl** in the frontend notes app at *src/services/notes.js* like so:

```
import axios from 'axios'
const baseUrl = 'http://localhost:3001/api/notes'

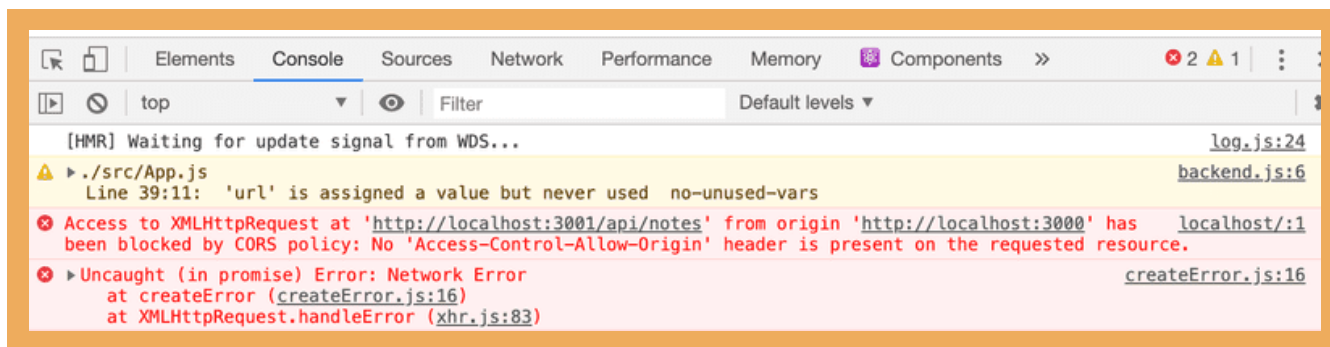
const getAll = () => {
  const request = axios.get(baseUrl)
  return request.then(response => response.data)
}

// ...

export default { getAll, create, update }
```

[copy](#)

Now frontend's GET request to <http://localhost:3001/api/notes> does not work for some reason:



What's going on here? We can access the backend from a browser and from postman without any problems.

## Same origin policy and CORS

The issue lies with a thing called `same origin policy`. A URL's origin is defined by the combination of protocol (AKA scheme), hostname, and port.

`http://example.com:80/index.html`

copy

protocol: http  
host: example.com  
port: 80

When you visit a website (e.g. <http://catwebsites.com>), the browser issues a request to the server on which the website ([catwebsites.com](http://catwebsites.com)) is hosted. The response sent by the server is an HTML file that may contain one or more references to external assets/resources hosted either on the same server that [catwebsites.com](http://catwebsites.com) is hosted on or a different website. When the browser sees reference(s) to a URL in the source HTML, it issues a request. If the request is issued using the URL that the source HTML was fetched from, then the browser processes the response without any issues. However, if the resource is fetched using a URL that doesn't share the same origin(scheme, host, port) as the source HTML, the browser will have to check the `Access-Control-Allow-origin` response header. If it contains `*` or the URL of the source HTML, the browser will process the response, otherwise the browser will refuse to process it and throws an error.

The **same-origin policy** is a security mechanism implemented by browsers in order to prevent session hijacking among other security vulnerabilities.

In order to enable legitimate cross-origin requests (requests to URLs that don't share the same origin) W3C came up with a mechanism called **CORS**(Cross-Origin Resource Sharing). According to [Wikipedia](#):

*Cross-origin resource sharing (CORS) is a mechanism that allows restricted resources (e.g. fonts) on a web page to be requested from another domain outside the domain from which the first resource was served. A web page may freely embed cross-origin images, stylesheets, scripts, iframes, and videos. Certain "cross-domain" requests, notably Ajax requests, are forbidden by default by the same-origin security policy.*

The problem is that, by default, the JavaScript code of an application that runs in a browser can only communicate with a server in the same origin. Because our server is in localhost port 3001, while our frontend is in localhost port 5173, they do not have the same origin.

Keep in mind, that same-origin policy and CORS are not specific to React or Node. They are universal principles regarding the safe operation of web applications.

We can allow requests from other *origins* by using Node's cors middleware.

In your backend repository, install *cors* with the command

```
npm install cors
```

[copy](#)

take the middleware to use and allow for requests from all origins:

```
const cors = require('cors')
```

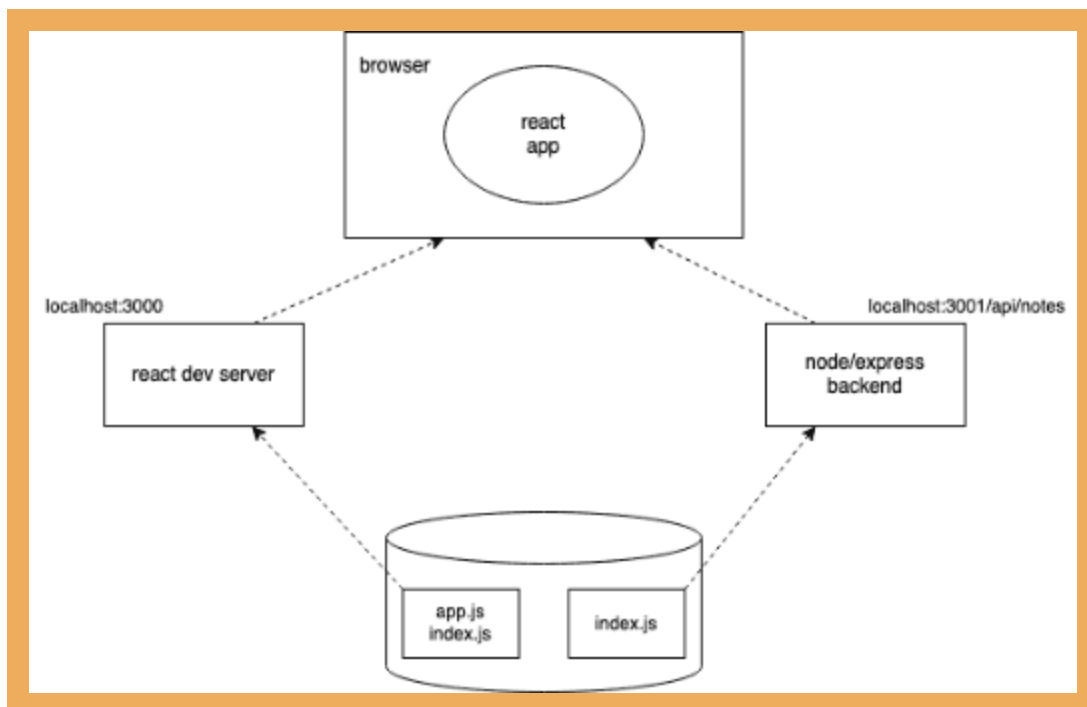
[copy](#)

```
app.use(cors())
```

And the frontend works! However, the functionality for changing the importance of notes has not yet been implemented on the backend.

You can read more about CORS from [Mozilla's page](#).

The setup of our app looks now as follows:



The react app running in the browser now fetches the data from node/express-server that runs in localhost:3001.

## Application to the Internet

Now that the whole stack is ready, let's move our application to the internet.

There is an ever-growing number of services that can be used to host an app on the internet. The developer-friendly services like PaaS (i.e. Platform as a Service) take care of installing the execution environment (eg. Node.js) and could also provide various services such as databases.

For a decade, Heroku was dominating the PaaS scene. Unfortunately the free tier Heroku ended at 27th November 2022. This is very unfortunate for many developers, especially students. Heroku is still very much a viable option if you are willing to spend some money. They also have a student program that provides some free credits.

We are now introducing two services Fly.io and Render that both have a (limited) free plan. Fly.io is our "official" hosting service since it can be for sure used also on the parts 11 and 13 of the course. Render will be fine at least for the other parts of this course.

Note that despite using the free tier only, Fly.io *might* require one to enter their credit card details. At the moment Render can be used without a credit card.

Render might be a bit easier to use since it does not require any software to be installed on your machine.

There are also some other free hosting options that work well for this course, at least for all parts other than part 11 (CI/CD) that might have one tricky exercise for other platforms.

Some course participants have also used the following

- Cyclic
- Replit
- CodeSandBox

If you know some other good and easy-to-use services for hosting NodeJS, please let us know!

For both Fly.io and Render, we need to change the definition of the port our application uses at the bottom of the *index.js* file in the backend like so:

```
const PORT = process.env.PORT || 3001
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`)
})
```

copy

Now we are using the port defined in the environment variable `PORT` or port 3001 if the environment variable `PORT` is undefined. Fly.io and Render configure the application port based on that environment variable.

## Fly.io

*Note that you may need to give your credit card number to Fly.io even if you are using only the free tier!* There has been actually conflicting reports about this, it is known for a fact that some of the students in this course are using Fly.io without entering their credit card info. At the moment Render can be used without a credit card.

By default, everyone gets two free virtual machines that can be used for running two apps at the same time.

If you decide to use Fly.io begin by installing their `flyctl` executable following this guide. After that, you should create a Fly.io account.

Start by authenticating via the command line with the command

```
fly auth login
```

[copy](#)

Note if the command `fly` does not work on your machine, you can try the longer version `flyctl`. Eg. on MacOS, both forms of the command work.

*If you do not get the `flyctl` to work in your machine, you could try Render (see next section), it does not require anything to be installed in your machine.*

Initializing an app happens by running the following command in the root directory of the app

```
fly launch
```

[copy](#)

Give the app a name or let Fly.io auto-generate one. Pick a region where the app will be run. Do not create a Postgres database for the app and do not create an Upstash Redis database, since these are not needed.

The last question is "Would you like to deploy now?". We should answer "no" since we are not quite ready yet.

Fly.io creates a file `fly.toml` in the root of your app where the app is configured. To get the app up and running we *might* need to do a small addition to the configuration:

```
[build]
```

[copy](#)

```
[env]
```

```
PORT = "3000" # add this
```

```
[http_service]
  internal_port = 3000 # ensure that this is same as PORT
  force_https = true
  auto_stop_machines = true
  auto_start_machines = true
  min_machines_running = 0
  processes = ["app"]
```

We have now defined in the part [env] that environment variable PORT will get the correct port (defined in part [http\_service]) where the app should create the server. Note that the definition might be already there, but some times it has been missing.

We are now ready to deploy the app to the Fly.io servers. That is done with the following command:

```
fly deploy
```

[copy](#)

If all goes well, the app should now be up and running. You can open it in the browser with the command

```
fly open
```

[copy](#)

A particularly important command is `fly logs`. This command can be used to view server logs. It is best to keep logs always visible!

**Note:** If you are using Fly.io, Fly may create 2 machines for your app, if it does then the state of the data in your app will be inconsistent between requests, i.e. you would have two machines each with its own notes variable, you could POST to one machine then your next GET could go to another machine. You can check the number of machines by using the command "`$ fly scale show`", if the COUNT is greater than 1 then you can enforce it to be 1 with the command "`$ fly scale count 1`". The machine count can also be checked on the dashboard.

**Note:** In some cases (the cause is so far unknown) running Fly.io commands especially on Windows WSL (Windows Subsystem for Linux) has caused problems. If the following command just hangs

```
flyctl ping -o personal
```

[copy](#)

your computer can not for some reason connect to Fly.io. If this happens to you, [this](#) describes one possible way to proceed.

If the output of the below command looks like this:

```
$ flyctl ping -o personal
35 bytes from fd00::8a3d::3 (gateway), seq=0 time=65.1ms
35 bytes from fd00::8a3d::3 (gateway), seq=1 time=28.5ms
35 bytes from fd00::8a3d::3 (gateway), seq=2 time=29.3ms
...
```

[copy](#)

then there are no connection problems!

Whenever you make changes to the application, you can take the new version to production with a command

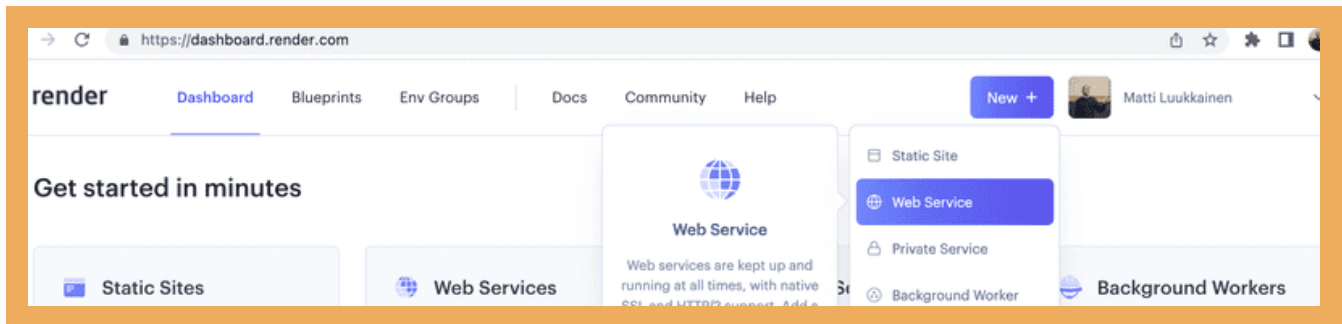
```
fly deploy
```

[copy](#)

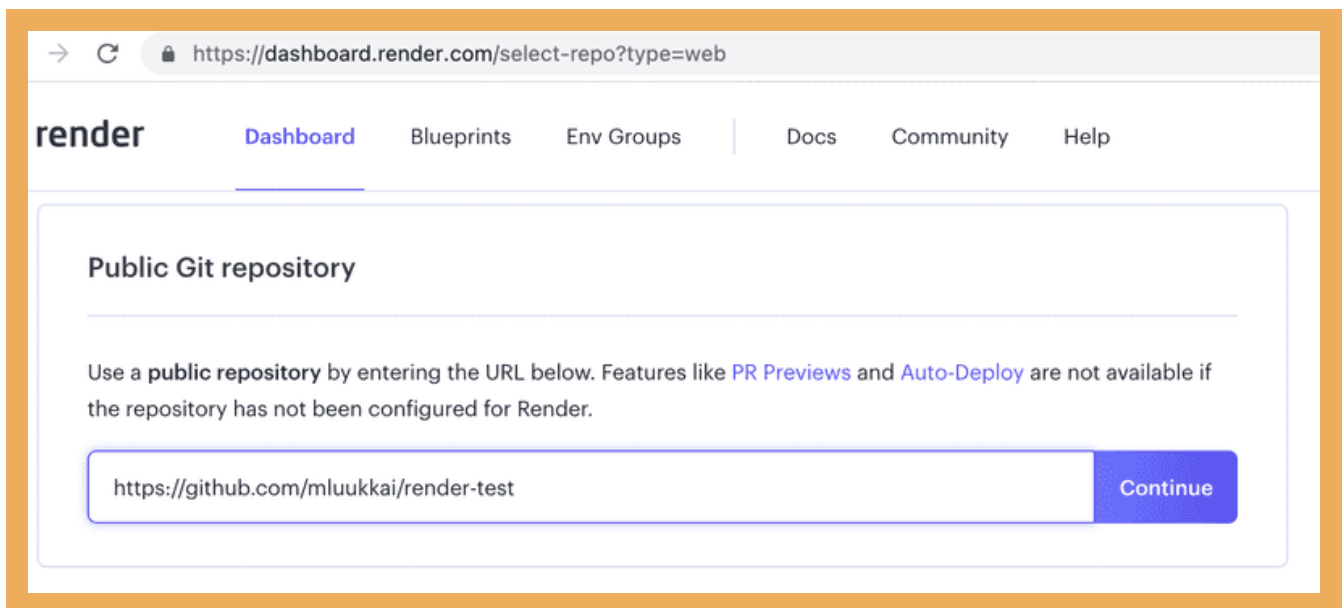
## Render

The following assumes that the sign in has been made with a GitHub account.

After signing in, let us create a new "web service":

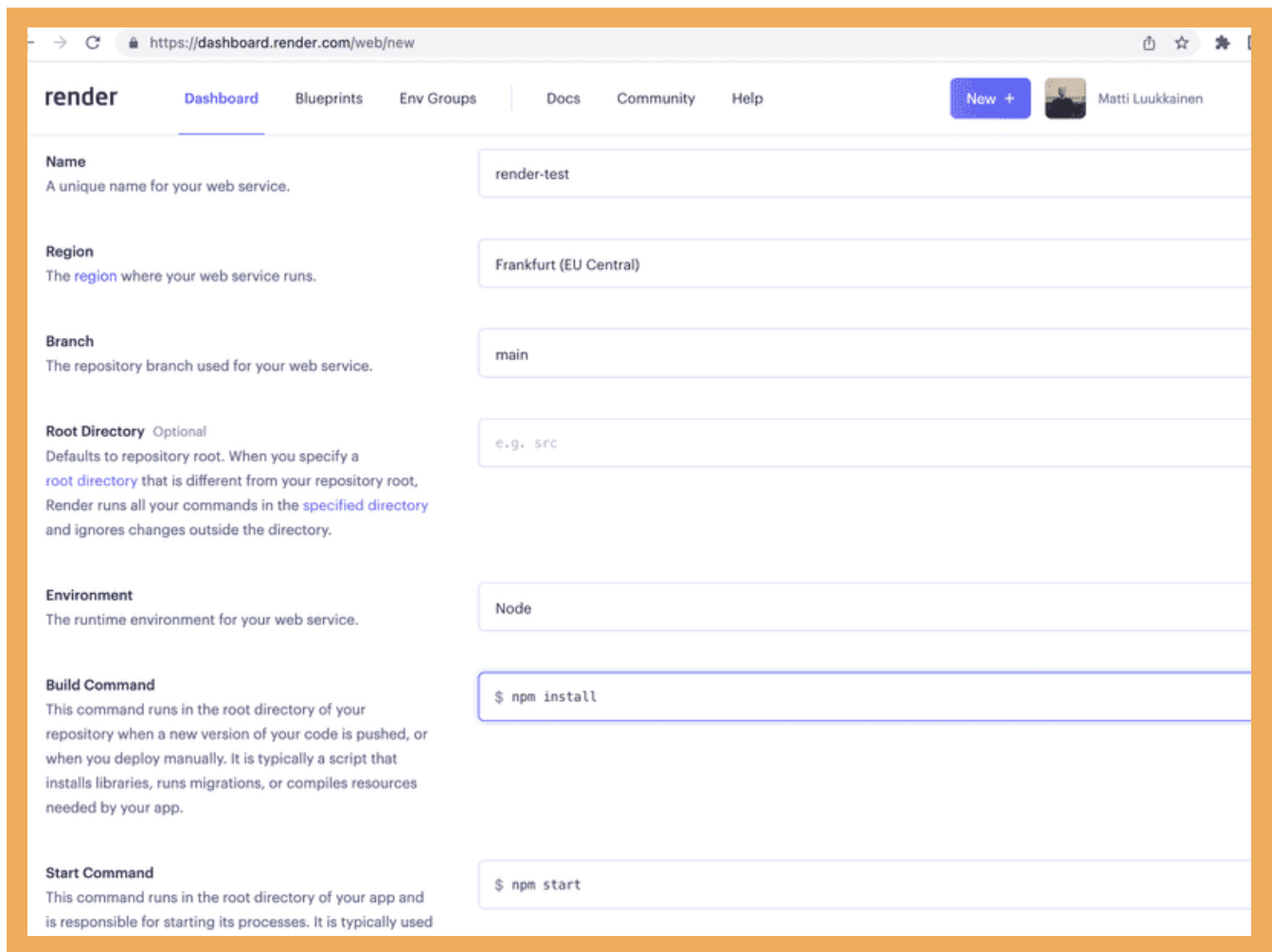


The app repository is then connected to Render:



The connection seems to require that the app repository is public.

Next we will define the basic configurations. If the app is *not* at the root of the repository the *Root directory* needs to be given a proper value:



The screenshot shows the 'New Web Service' form on the Render dashboard. The form is titled 'render' and has a navigation bar with links to 'Dashboard', 'Blueprints', 'Env Groups', 'Docs', 'Community', and 'Help'. A 'New +' button and a user profile for 'Matti Luukkainen' are also visible. The form fields are as follows:

- Name:** A unique name for your web service. Value: `render-test`
- Region:** The region where your web service runs. Value: `Frankfurt (EU Central)`
- Branch:** The repository branch used for your web service. Value: `main`
- Root Directory:** Optional. Defaults to repository root. When you specify a root directory that is different from your repository root, Render runs all your commands in the specified directory and ignores changes outside the directory. Value: `e.g. src`
- Environment:** The runtime environment for your web service. Value: `Node`
- Build Command:** This command runs in the root directory of your repository when a new version of your code is pushed, or when you deploy manually. It is typically a script that installs libraries, runs migrations, or compiles resources needed by your app. Value: `$ npm install`
- Start Command:** This command runs in the root directory of your app and is responsible for starting its processes. It is typically used. Value: `$ npm start`

After this, the app starts up in the Render. The dashboard tells us the app state and the url where the app is running:



The screenshot shows the Render dashboard for a web service named 'render-test'. The top navigation bar includes 'Dashboard', 'Blueprints', 'Env Groups', 'Docs', 'Community', and 'Help'. The user 'Matti Luukkainen' is logged in. The service is on the 'Free Plan' and is connected to a GitHub repository 'mluukkai/render-test' at the 'main' branch. The URL is 'https://render-test-yu7p.onrender.com'. The left sidebar lists various sections: Events, Logs, Disks, Environment, Shell, PRs, Jobs, Metrics, Scaling, and Settings. The main content area shows a build log for a deployment on January 18, 2023, at 11:48 AM. The log indicates that 7 packages are looking for funding, no vulnerabilities were found, and the container image is being generated and uploaded. A notification at the top suggests upgrading to a paid plan for faster builds.

According to the [documentation](#) every commit to GitHub should redeploy the app. For some reason this is not always working.

Fortunately, it is also possible to manually redeploy the app:

This screenshot shows the same Render dashboard, but with the 'Manual Deploy' dropdown menu open. The menu options are 'Deploy latest commit' (highlighted with a red box), 'Deploy a specific commit', and 'Clear build cache & deploy'. Below the menu, the 'Events' section shows a successful deployment event: 'Deploy live for 81a315c: initial' on January 18, 2023, at 11:50 AM, marked with a green checkmark.

Also, the app logs can be seen in the dashboard:

WEB SERVICE

render-test Node Free Plan mluukkai/render-test main Connect

<https://render-test-yu7p.onrender.com>

Events

Logs

Disks

Environment

Shell

PRs

Jobs

Metrics

Scaling

Settings

Search logs Search

Jan 18 12:04:41 PM > node index.js

Jan 18 12:04:41 PM

Jan 18 12:04:43 PM Server running on port 10000

Jan 18 12:04:48 PM Method: GET

Jan 18 12:04:48 PM Path: /

Jan 18 12:04:48 PM Body: {}

Jan 18 12:04:48 PM

Jan 18 12:04:57 PM ==> Starting service with 'npm start'

Jan 18 12:05:00 PM

Jan 18 12:05:00 PM > render-test@1.0.0 start /opt/render/project/src

Jan 18 12:05:00 PM > node index.js

Jan 18 12:05:00 PM

Jan 18 12:05:01 PM Server running on port 10000

Jan 18 12:05:28 PM Method: GET

Jan 18 12:05:28 PM Path: /

Jan 18 12:05:28 PM Body: {}

Jan 18 12:05:28 PM

We notice now from the logs that the app has been started in the port 10000. The app code gets the right port through the environment variable `PORT` so it is essential that the file `index.js` has been updated in the backend as follows:

```
const PORT = process.env.PORT || 3001
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`)
})
```

copy

## Frontend production build

So far we have been running React code in *development mode*. In development mode the application is configured to give clear error messages, immediately render code changes to the browser, and so on.

When the application is deployed, we must create a production build or a version of the application which is optimized for production.

A production build of applications created with Vite can be created with the command `npm run build`.

Let's run this command from the *root of the notes frontend project* that we developed in Part 2.

This creates a directory called `dist` which contains the only HTML file of our application (`index.html`) and the directory `assets`. Minified version of our application's JavaScript code will be generated in the `dist` directory. Even though the application code is in multiple files, all of the JavaScript will be minified into

one file. All of the code from all of the application's dependencies will also be minified into this single file.

The minified code is not very readable. The beginning of the code looks like this:

```
!function(e){function r(r){for(var n,f,i=r[0],l=r[1],a=r[2],c=0,s=
[];c<i.length;c++)f=i[c],o[f]&&s.push(o[f][0]),o[f]=0;for(n in
l)Object.prototype.hasOwnProperty.call(l,n)&&(e[n]=l[n]);for(p&&p(r);s.length;)s.shift()
();return u.push.apply(u,a||[]),t()}function t(){for(var e,r=0;r<u.length;r++){for(var
t=u[r],n=!0,i=1;i<t.length;i++){var l=t[i];0!==o[l]&&(n=!1)}n&&(u.splice(r-
,1),e=f(f.s=t[0]))}return e}var n={},o={2:0},u=[];function f(r){if(n[r])return
n[r].exports;var t=n[r]={i:r,l:!1,exports:{}};return
e[r].call(t.exports,t,t.exports,f),t.l=!0,t.exports}f.m=e,f.c=n,f.d=function(e,r,t)
{f.o(e,r)||Object.defineProperty(e,r,{enumerable:!0,get:t})},f.r=function(e)
{"undefined"!==typeof
Symbol&&Symbol.toStringTag&&Object.defineProperty(e,Symbol.toStringTag,{value:"Module"})}
```

copy

## Serving static files from the backend

One option for deploying the frontend is to copy the production build (the *dist* directory) to the root of the backend repository and configure the backend to show the frontend's *main page* (the file *dist/index.html*) as its main page.

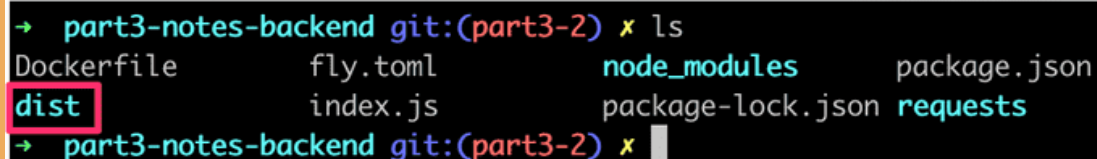
We begin by copying the production build of the frontend to the root of the backend. With a Mac or Linux computer, the copying can be done from the frontend directory with the command

```
cp -r dist ../backend
```

copy

If you are using a Windows computer, you may use either copy or xcopy command instead. Otherwise, simply copy and paste.

The backend directory should now look as follows:



```
→ part3-notes-backend git:(part3-2) x ls
Dockerfile      fly.toml        node_modules    package.json
dist            index.js        package-lock.json requests
→ part3-notes-backend git:(part3-2) x
```

To make express show *static content*, the page *index.html* and the JavaScript, etc., it fetches, we need a built-in middleware from Express called static.

When we add the following amidst the declarations of middlewares

```
app.use(express.static('dist'))
```

copy

whenever express gets an HTTP GET request it will first check if the *dist* directory contains a file corresponding to the request's address. If a correct file is found, express will return it.

Now HTTP GET requests to the address *www.serversaddress.com/index.html* or *www.serversaddress.com* will show the React frontend. GET requests to the address *www.serversaddress.com/api/notes* will be handled by the backend code.

Because of our situation, both the frontend and the backend are at the same address, we can declare `baseUrl` as a relative URL. This means we can leave out the part declaring the server.

```
import axios from 'axios'
const baseUrl = '/api/notes'

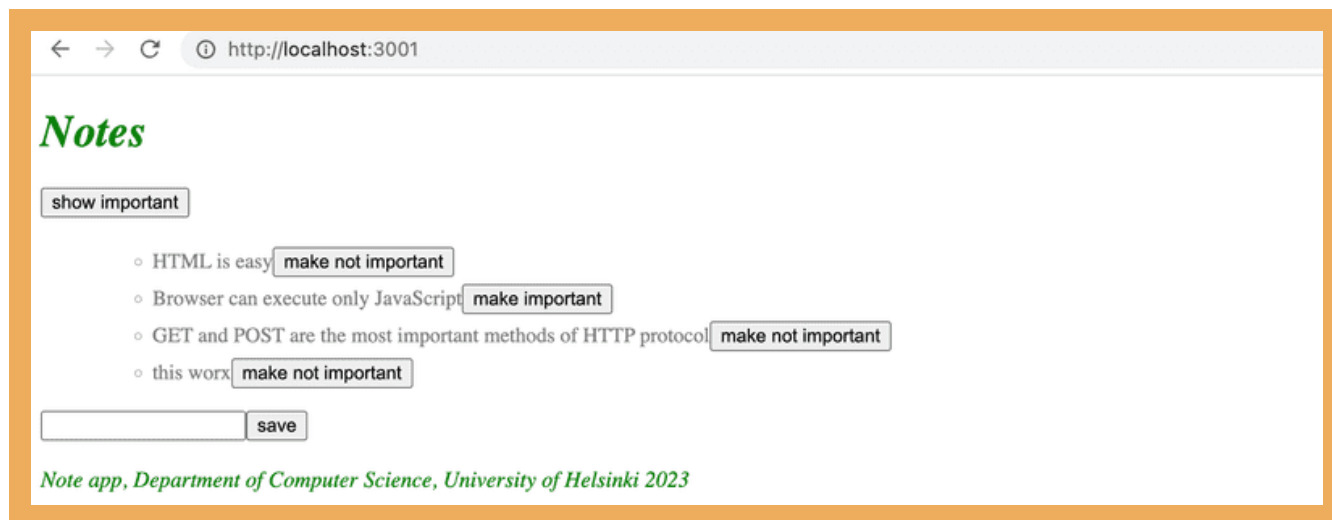
const getAll = () => {
  const request = axios.get(baseUrl)
  return request.then(response => response.data)
}

// ...
```

copy

After the change, we have to create a new production build of the frontend and copy it to the root of the backend repository.

The application can now be used from the *backend* address <http://localhost:3001>:



Our application now works exactly like the single-page app example application we studied in part 0.

When we use a browser to go to the address <http://localhost:3001>, the server returns the *index.html* file from the *dist* directory. The contents of the file are as follows:

```

<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite + React</title>
    <script type="module" crossorigin src="/assets/index-5f6faa37.js"></script>
    <link rel="stylesheet" href="/assets/index-198af077.css">
  </head>
  <body>
    <div id="root"></div>

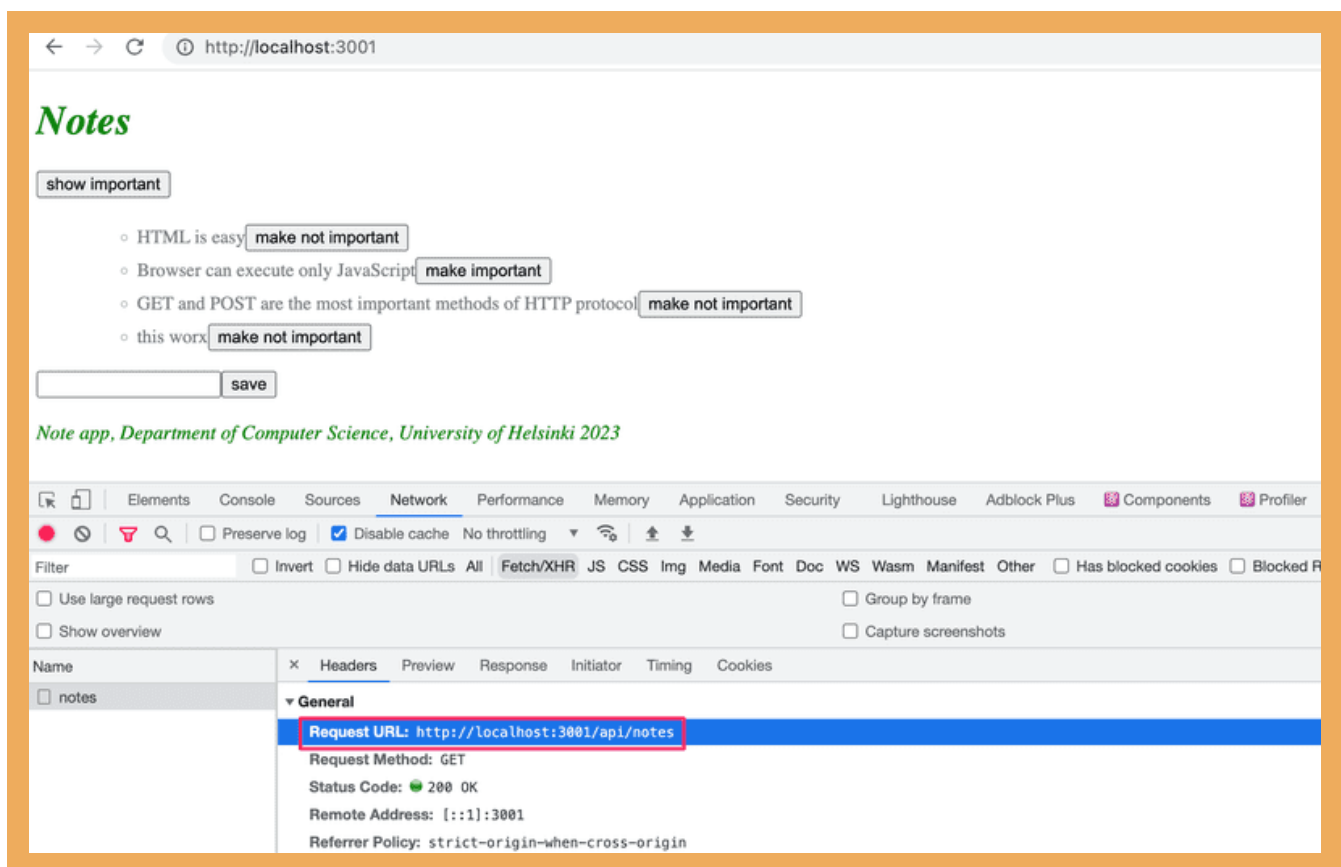
  </body>
</html>

```

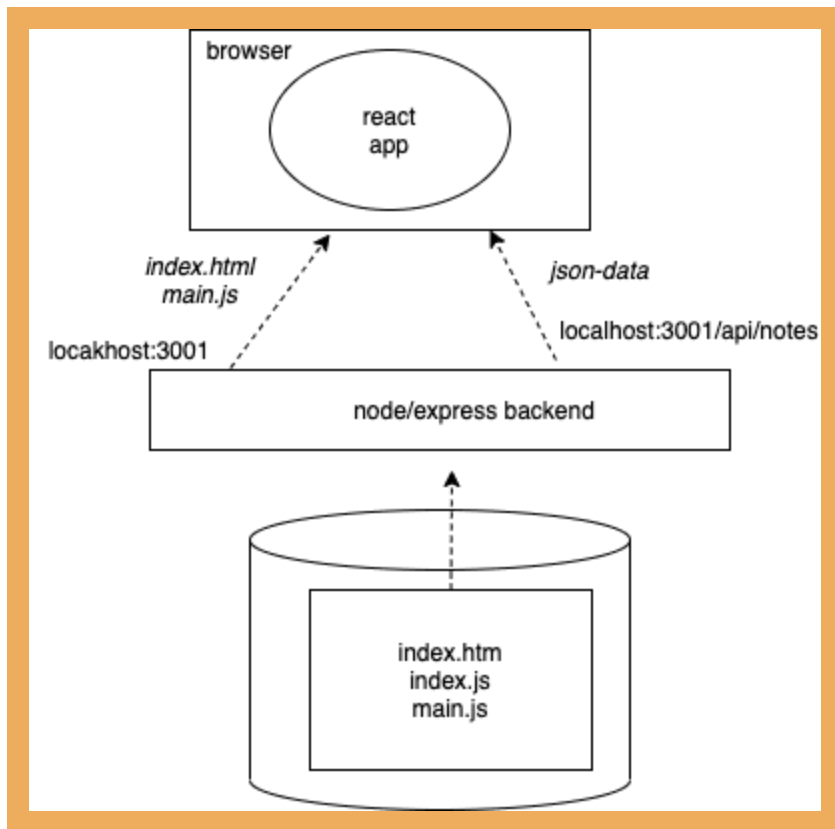
copy

The file contains instructions to fetch a CSS stylesheet defining the styles of the application, and one *script* tag that instructs the browser to fetch the JavaScript code of the application - the actual React application.

The React code fetches notes from the server address <http://localhost:3001/api/notes> and renders them to the screen. The communication between the server and the browser can be seen in the *Network* tab of the developer console:



The setup that is ready for a product deployment looks as follows:



Unlike when running the app in a development environment, everything is now in the same node/express-backend that runs in localhost:3001. When the browser goes to the page, the file *index.html* is rendered. That causes the browser to fetch the production version of the React app. Once it starts to run, it fetches the json-data from the address localhost:3001/api/notes.

## The whole app to the internet

After ensuring that the production version of the application works locally, commit the production build of the frontend to the backend repository, and push the code to GitHub again.

If you are using Render a push to GitHub *might* be enough. If the automatic deployment does not work, select the "manual deploy" from the Render dashboard.

In the case of Fly.io the new deployment is done with the command

```
fly deploy
```

[copy](#)

The application works perfectly, except we haven't added the functionality for changing the importance of a note to the backend yet.

**NOTE:** If you are using Fly.io, there could be a *.dockerignore* file that specifies the exclusion of the *./build* directory during deployment. To ensure it gets deployed, consider renaming the *./build* directory to *./static\_build* or an equivalent name.

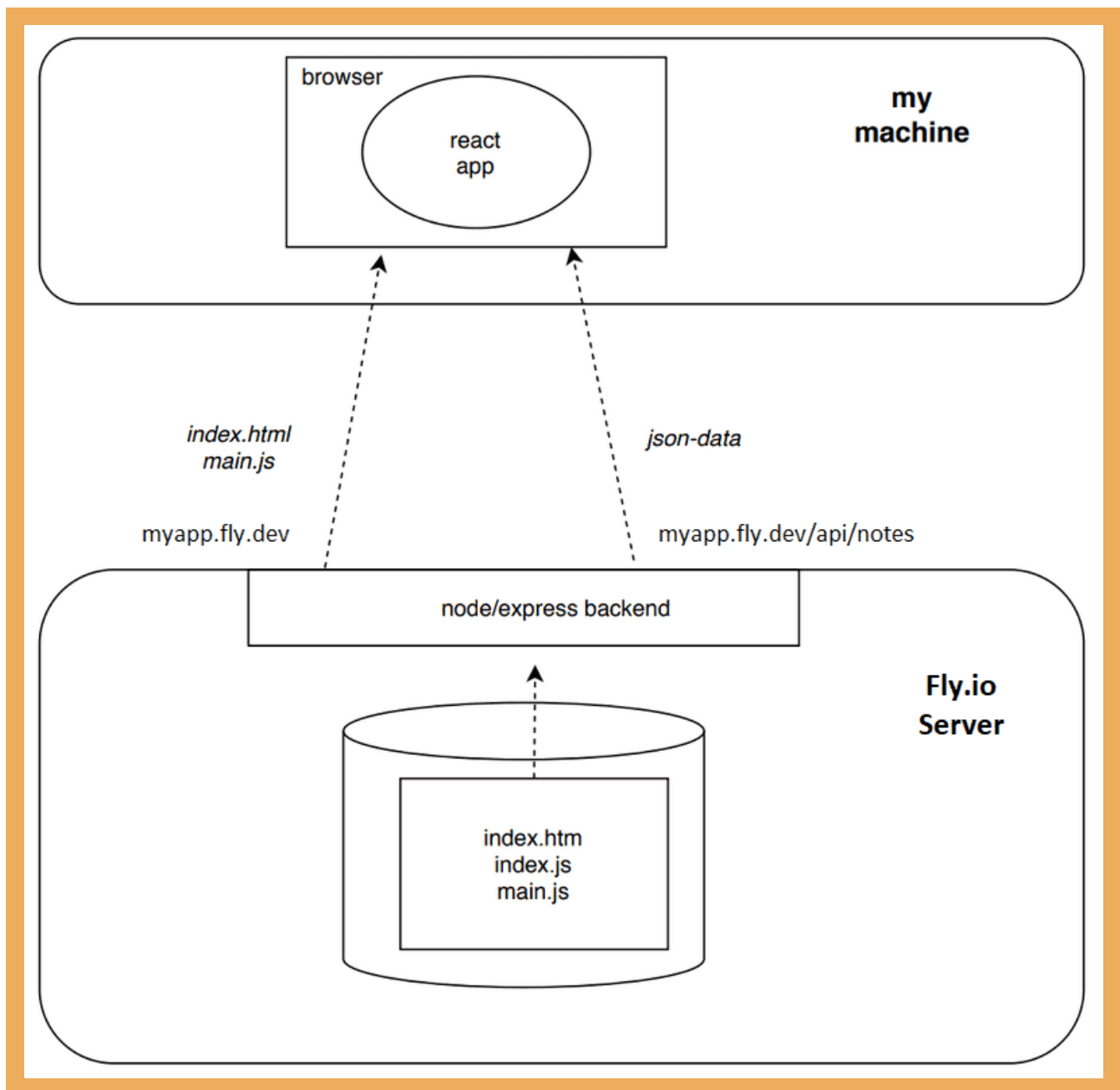


**NOTE:** changing of the importance DOES NOT work yet since the backend has no implementation for it yet.

Our application saves the notes to a variable. If the application crashes or is restarted, all of the data will disappear.

The application needs a database. Before we introduce one, let's go through a few things.

The setup now looks like as follows:



The node/express-backend now resides in the Fly.io/Render server. When the root address is accessed, the browser loads and executes the React app that fetches the json-data from the Fly.io/Render server.

## Streamlining deploying of the frontend

To create a new production build of the frontend without extra manual work, let's add some npm-scripts to the *package.json* of the backend repository.

### Fly.io script

The script looks like this:



```
{
  "scripts": {
    // ...
    "build:ui": "rm -rf dist && cd ../notes-frontend/ && npm run build && cp -r dist
../notes-backend",
    "deploy": "fly deploy",
    "deploy:full": "npm run build:ui && npm run deploy",
    "logs:prod": "fly logs"
  }
}
```

copy

### Note for Windows users

Note that the standard shell commands in `build:ui` do not natively work in Windows. Powershell in Windows works differently, in which case the script could be written as

```
"build:ui": "@powershell Remove-Item -Recurse -Force dist && cd ../frontend && npm run build && @powershell Copy-Item dist -Recurse ../backend",
```

copy

If the script does not work on Windows, confirm that you are using Powershell and not Command Prompt. If you have installed Git Bash or another Linux-like terminal, you may be able to run Linux-like commands on Windows as well.

The script `npm run build:ui` builds the frontend and copies the production version under the backend repository. The script `npm run deploy` releases the current backend to Fly.io.

`npm run deploy:full` combines these two scripts, i.e., `npm run build:ui` and `npm run deploy`.

There is also a script `npm run logs:prod` to show the Fly.io logs.

Note that the directory paths in the script `build:ui` depend on the location of repositories in the file system.

### Render

Note: When you attempt to deploy your backend to Render, make sure you have a separate repository for the backend and deploy that github repo through Render, attempting to deploy through your Fullstackopen repository will often throw "ERR path ....package.json".

In case of Render, the scripts look like the following

```
{
  "scripts": {
    //...
    "build:ui": "rm -rf dist && cd ../frontend && npm run build && cp -r dist
```

copy

```
../backend",  
  "deploy:full": "npm run build:ui && git add . && git commit -m uibuild && git push"  
}  
}
```

The script `npm run build:ui` builds the frontend and copies the production version under the backend repository. `npm run deploy:full` contains also the necessary *git* commands to update the backend repository.

Note that the directory paths in the script *build:ui* depend on the location of repositories in the file system.

**NB** On Windows, npm scripts are executed in `cmd.exe` as the default shell which does not support bash commands. For the above bash commands to work, you can change the default shell to Bash (in the default Git for Windows installation) as follows:

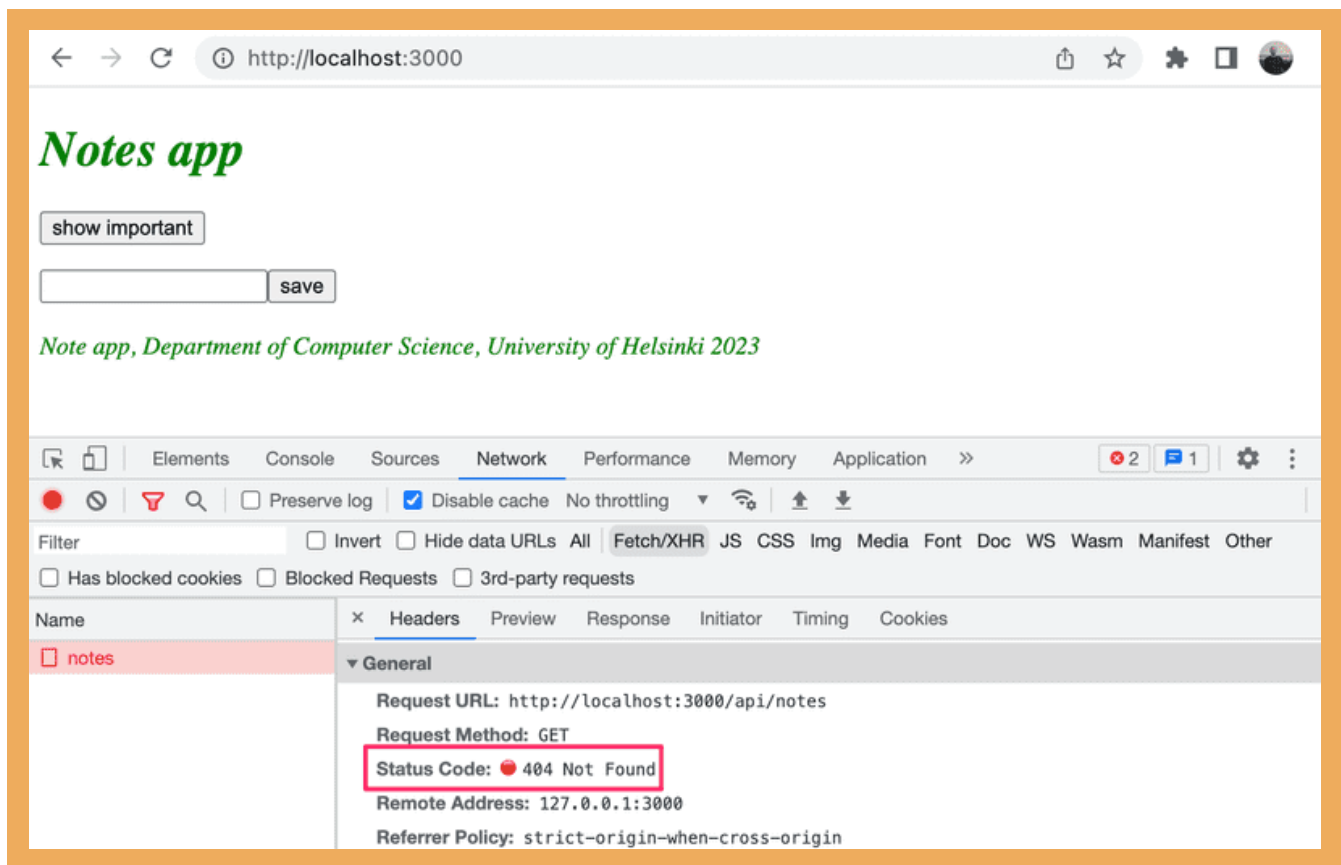
```
npm config set script-shell "C:\\Program Files\\git\\bin\\bash.exe"
```

[copy](#)

Another option is the use of shx.

## Proxy

Changes on the frontend have caused it to no longer work in development mode (when started with command `npm run dev`), as the connection to the backend does not work.



This is due to changing the backend address to a relative URL:

```
const baseUrl = '/api/notes'
```

[copy](#)

Because in development mode the frontend is at the address `localhost:5173`, the requests to the backend go to the wrong address `localhost:5173/api/notes`. The backend is at `localhost:3001`.

If the project was created with Vite, this problem is easy to solve. It is enough to add the following declaration to the `vite.config.js` file of the frontend repository.

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [react()],
```

[copy](#)

```

server: {
  proxy: {
    '/api': {
      target: 'http://localhost:3001',
      changeOrigin: true,
    },
  },
},
},
})

```

After a restart, the React development environment will work as a proxy. If the React code does an HTTP request to a server address at <http://localhost:5173> not managed by the React application itself (i.e. when requests are not about fetching the CSS or JavaScript of the application), the request will be redirected to the server at <http://localhost:3001>.

Note that with the vite-configuration shown above, only requests that are made to paths starting with */api*-are redirected to the server.

Now the frontend is also fine, working with the server both in development and production mode.

A negative aspect of our approach is how complicated it is to deploy the frontend. Deploying a new version requires generating a new production build of the frontend and copying it to the backend repository. This makes creating an automated deployment pipeline more difficult. Deployment pipeline means an automated and controlled way to move the code from the computer of the developer through different tests and quality checks to the production environment. Building a deployment pipeline is the topic of part 11 of this course. There are multiple ways to achieve this, for example, placing both backend and frontend code in the same repository but we will not go into those now.

In some situations, it may be sensible to deploy the frontend code as its own application.

The current backend code can be found on Github, in the branch *part3-3*. The changes in frontend code are in *part3-1* branch of the frontend repository.

## Exercises 3.9.-3.11

The following exercises don't require many lines of code. They can however be challenging, because you must understand exactly what is happening and where, and the configurations must be just right.

### 3.9 phonebook backend step9

Make the backend work with the phonebook frontend from the exercises of the previous part. Do not implement the functionality for making changes to the phone numbers yet, that will be implemented in exercise 3.17.

You will probably have to do some small changes to the frontend, at least to the URLs for the backend. Remember to keep the developer console open in your browser. If some HTTP requests fail, you should check from the *Network*-tab what is going on. Keep an eye on the backend's console as well. If you did

not do the previous exercise, it is worth it to print the request data or *request.body* to the console in the event handler responsible for POST requests.

### 3.10 phonebook backend step10

Deploy the backend to the internet, for example to Fly.io or Render.

Test the deployed backend with a browser and Postman or VS Code REST client to ensure it works.

**PRO TIP:** When you deploy your application to Internet, it is worth it to at least in the beginning keep an eye on the logs of the application **AT ALL TIMES**.

Create a README.md at the root of your repository, and add a link to your online application to it.

**NOTE:** as it was said, you should deploy the BACKEND to the cloud service. If you are using Fly.io the commands should be run in the root directory of the backend (that is, in the same directory where the backend package.json is). In case of using Render, the backend must be in the root of your repository.

You shall NOT be deploying the frontend directly at any stage of this part. It is just backend repository that is deployed throughout the whole part, nothing else.

### 3.11 phonebook full stack

Generate a production build of your frontend, and add it to the internet application using the method introduced in this part.

**NB** If you use Render, make sure the directory *dist* is not gitignored

Also, make sure that the frontend still works locally (in development mode when started with command `npm run dev` ).

If you have problems getting the app working make sure that your directory structure matches the example app.

## Propose changes to material

Part 3a  
**Previous part**

Part 3c  
**Next part**

About course

Course contents

FAQ

Partners

Challenge



UNIVERSITY OF HELSINKI

