Crișan Alexandra Bianca
Group 248/1

Proof of Concept - Next.js

- ***What is Next.js?***

  Next.js is an open-source *React front-end development web framework* created by Vercel. It enables features such as server-side rendering and generating static websites.

- ***Setup***

  - **Requirements**:
    - *Node.js* 10.13 or later
    - MacOS/Windows/Linux

  - **Creating a new Next.js app** using the terminal – the easiest way is using **"create-next-app"** because it sets up everything you need automatically

```
npx create-next-app
# or
yarn create next-app
```

  Afterwards for running the application:

```
npm run dev
```

- ***TVTime application***

  The application created as a proof of concept for this framework is one that displays movies and tv shows along with details about each of them, such as description, score, number of episodes and the cover photo. It allows users to browse through the lists of the most popular movies and tv shows.

- ***Pages***

  In Next.js, a **page** is a React Component exported from a file in the "pages" directory, each one associated with a **route** based on its filename.

Index routes – the router will automatically route files that are named "index" to the root of the directory. So "pages/index.js" will always be associated with the '/' (home) route.

And in another example, "pages/movies/first-movie.js" is associated with the "movies/first-movie" route. **Nested routes** are also supported.

So each desired page should be created inside the pages directory with the name of the route you want it to have.

- ***Dynamic routes***

Next.js also supports dynamic routing for pages with routes that need external data.
This is done by adding brackets to a page name: for example the page **"pages/movies/[id].js"** will match any route like "/movies/1", "/movies/12" – in this case it's intended for displaying the details of a movie with a given **id**.

- ***API routes***

You can also build your own API with Next.js, by adding files inside a folder **"pages/api".** Just like in case of pages, the files will be mapped to routes like "/api/*", but they will be treated as API endpoints, instead of pages.

For example, the following API route "pages/api/index.js" returns a json response with the status code 200 that gets all the movies from the database.

```
pages > api > movie > JS index.js > ...
1   import { find } from './db';
2
3   export default function handler(req, res) {
4     res.status(200).json(find({}))
5   }
```

The function needs to be exported as default and receives as parameters:
- **req**: an instance of http.IncomingMessage
- **res**: an instance of http.ServerResponse

Moreover, these API routes can also be **dynamic**, for example fetching a movie by its id inside a file called **"pages/api/movie/[id].js":**

pages > api > movie > JS [id].js > movieHandler

```
import { find } from './db';

export default function movieHandler(req, res) {
  const {
    query: { id },
    method,
  } = req
  switch (method) {
    case 'GET':
      const item = find({ id });
      if (item) {
        res.status(200).json(item)
      } else {
        res.status(404).end('Not found')
      }
      break
    default:
      res.setHeader('Allow', ['GET'])
      res.status(405).end(`Method ${method} Not Allowed`)
  }
}

```

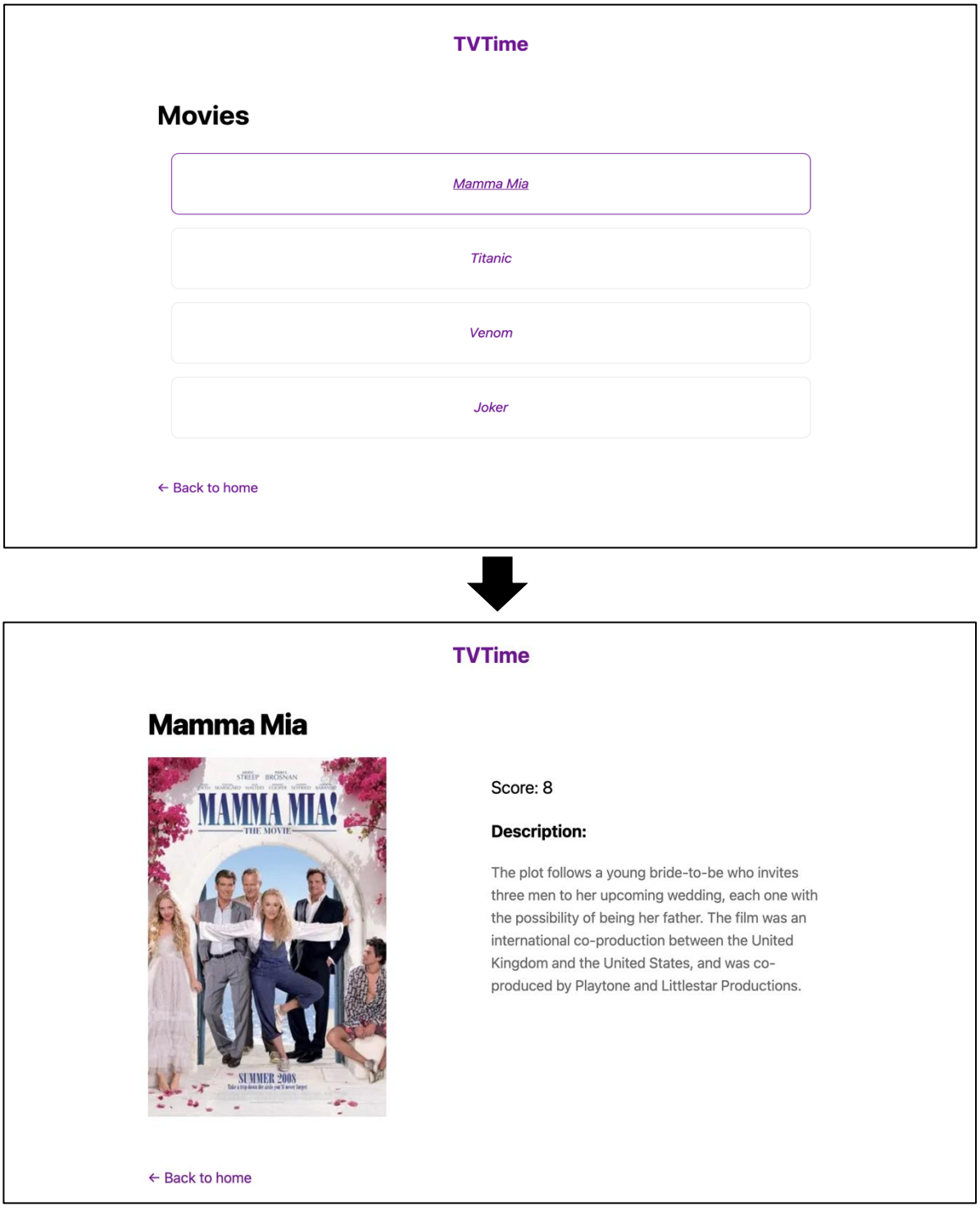
It's very similar except that it also uses a query parameter for the id which is passed to the 'find' function that searches the db. It also returns a 404 error status if the movie is not found.

- ***Links***

Client-side route transitions are allowed, similar to a single page application, using a React component called **"Link".**

```
{movies.map(movie => (
  <div key={movie.id}  className={utilStyles.card}>
    <Link href={`/movies/${movie.id}`}>
      <a><em>{movie.title}</em></a>
    </Link>
  </div>
))}
```

In this case each element of the movies array has a link to its personal page containing its details. **Dynamic paths** are present here as well, in the value of the href tag with the movie id as a parameter. When clicking on a movie, you should be taken to its details page.



- ***Pre-rendering***

  By default, Next.js pre-renders every page, generating HTML in advance for each page, instead of doing it by client-side Javascript. This is one of the features that contributes to this framework's level of *performance*.

There are 2 forms of pre-rendering in Next.js, the difference between them being **WHEN** it generates the HTML for pages:

- ***Static generation*** (recommended) – HTML generated at build time & reused on each request

- ***Server-side rendering*** – HTML generated ***on each request***

A "hybrid" application can be developed by using both types of pre-rendering for different pages, but the Static Generation is recommended because it is much faster. However, if your pages need frequently updated data, server-side rendering might be more appropriate.

Static Generation without data

This is done by default by the framework if the page does not need any external data to be fetched.

Static Generation with data

However if fetching external data is required (from a file, an API, a database etc), there are 2 functions that can be used:

- **getStaticProps** – the page content depends on external data

  The function needs to be implemented in the same file, and exported as an async function. It gets called at build time and passes the fetched data to the page's props.

```
export async function getStaticProps() {
  let staticMovies = null;
  try {
    staticMovies = await fetcher('api/movie')
  } catch (err) {
  }
  return {
    props: {
        staticMovies,
    }
  }
}
```

- **getStaticPaths** – the page paths depend on external data

  This is similar to the previous function, it's still an async exported function but in this case you return the paths that you want to pre-render.

Server-side rendering

To make sure your page is updated more frequently, another function needs to be implemented instead of getStaticProps – **getServerSideProps** – still an async exported function, but it will be called by the server on each request, not at build time.

```
export async function getServerSideProps(context) {
  return {
    props: {
      // props for your component
    }
  }
}
```

Client-side rendering

This strategy is useful if a page contains frequently updating data that *doesn't need* to be pre-rendered.

- First, statically pre-render parts of the page that do not require external data (ex: loading states).
- Then fetch external data from the client using JavaScript and display it.

SWR – stale-while-revalidate

SWR is a React hook for data fetching created by the team behind Next.js with multiple nice features such as caching, revalidation, refetching on interval etc. It is highly recommended for client-side fetching as the updates are done constantly and automatically, keeping the UI fast and reactive.

It does not require any additional setup, just importing the **useSwr** hook and using it inside your component. The function accepts a key string (the API url) and a fetcher function that returns the data. Based on the status, 2 values can be returned: **data** and **error**.

```
pages > movies > JS index.js > ...
1   import useSwr from "swr";
2   import Layout from '../../components/layout'
3
4   const fetcher = (url) => fetch(url).then((res) => res.json())
5
6   export default function Index({ staticMovies }) {
7     const { data: movies, error } = useSwr('/api/movie', fetcher, { initialData: staticMovies })
8     return (
9       <Layout>
10        ...
11      </Layout>
12    )
13  }
14
15  export async function getStaticProps() {
16    let staticMovies = null;
17    try {
18      staticMovies = await fetcher('api/movie')
19    } catch (err) {
20    }
21    return {
22      props: {
23          staticMovies,
24      }
25    }
26  }
27
```

When used together with Static Generation, the pre-fetched data from getStaticProps can be passed as the initial value to the **initialData** option inside the fetcher function – in this case the staticMovies which are actually the props.

So the page can be cached and accessed very fast but the fetched data can be dynamic and update itself over time.

- ***Fast Refresh***

The fast refresh is a very useful feature of Next.js because it gives instant feedback to the programmer regarding the edits performed on the code, without losing component state.

It is also worth mentioning that when making syntax errors, fixing it and saving the file will make the error dissapear automatically without the need of reloading the app.