# Pokedex React Tutorial

Jeroen Nouws

# Pokedex React Tutorial - T.O.C.

Click here if you rather have a PDF

# Preface

## Asumptions

The reader of this tutorial should have a basic understanding of web based technologies such as html, css and javascript.

## Versions used

- node: 14 (LTS)
- yarn: 1.22.5
- react: 17

## Code Examples

Reading a tutorial can give you insights into a tech stack, but writing code and testing it gives you a good feeling of learning to work with React. All code is available on Github.

## Commands and CLI

All commands in this tutorial are run in a bash shell on linux and not tested in windows based terminals like command prompt or Powershell.

# I. Introduction

# Chapter 1. Typescript Primer

## 1.1. What is Typescript? A quick overview

- superset of javascript
- adding typescript to your project
  - installing typescript
  - adding tsconfig
  - adding package.json script
- typing
  - types
  - interfaces
- classes
- enums

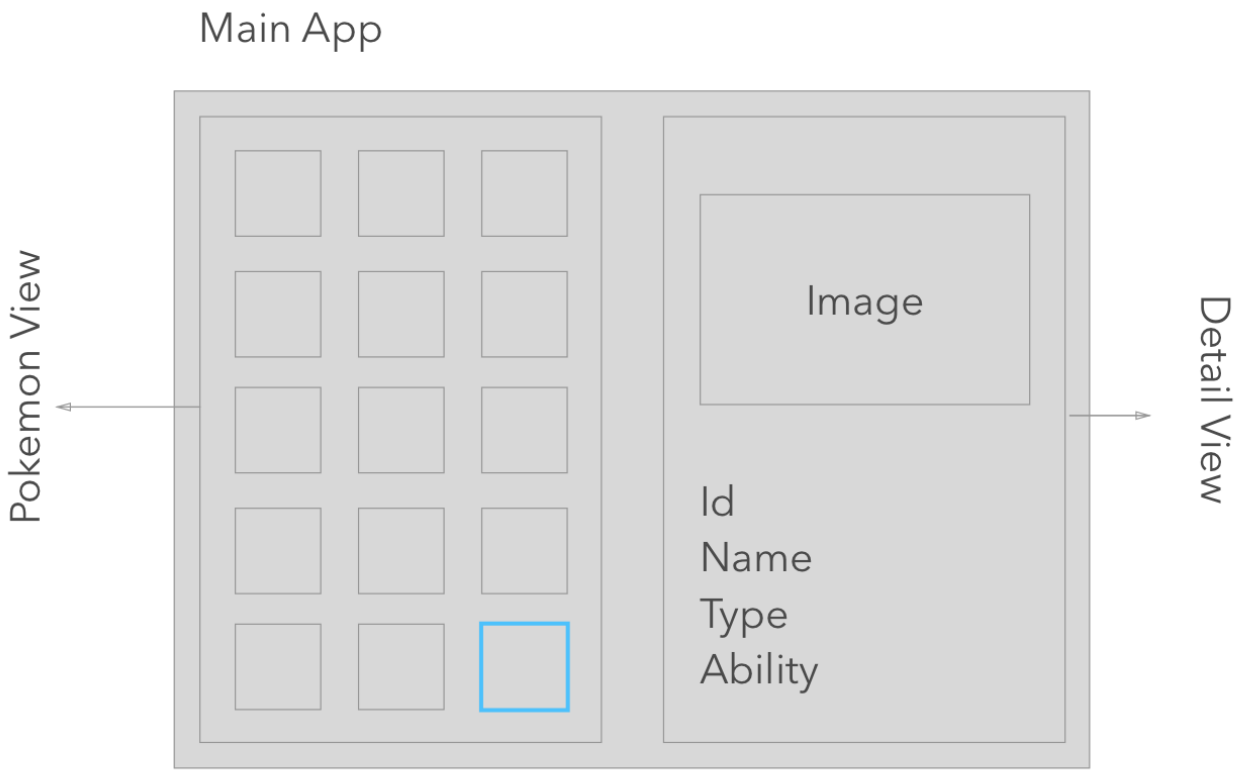
# Chapter 2. Project Setup

## 2.1. What are we building?

In this tutorial we'll be building a pokedex react app. We'll do so by implementing the commonly used component/container pattern (representational/logical components).
We'll also touch visual testing and documentation by writing stories for Storybook and use Jest for component testing and snapshot testing.
Finally we'll introduce redux into our project for managing state and thunk middleware to handle asyncronous api calls.
This setup might seem overkill for a project of this size and there are better solutions to handle state than through redux for such a small project, but the purpose of this tutorial is to teach you some commonly used technologies and patterns within the React ecosystem.

### 2.1.1. Wireframe

Our Application has a very basic layout consisting of 2 columns, in the left hand column we'll display a list of selectable pokemons, while in the right side column we'll display either the details of the selected pokémon or a placeholder text.




### 2.1.2. Datasource

All our Pokémon information will be pulled from the Poke API. This is a standard REST-Api, altough at the time of writing they are going into beta with their GraphQL api.
This API holds an extensive source of information about all different Pokémons, going from names,

to types, to abilities and more. If you like this tutorial go give those guys your support.

## 2.2. Creating a new React Project

Before we start creating a new React project we should have a little talk about what React is and isn't.
React is an open source javascript **library** that's used for building user interfaces and single page applications. Altough there are other popular frameworks around, React stands out compared to other frameworks like Angular and Vue.js by being:

- **Declerative** instead of imperative
- **Not opinionated**, except on how to render views, React doesn't care how you structure or build your project.
- **Lightweight**
- **Library not a framework**, React focusses on creating components and building pages, solutions like state management and routing aren't part of React.js, but are managed by the community.

| NOTE | Since React isn't opinionated you'll often find different solutions for the same problem i.e.: Flux vs Redux for state management, css vs styled components for styling,... |
|---|---|

### 2.2.1. Using create-react-app to scaffold a new React project

Creating a new react app is as easy as running one command, so let's get started. Open up your terminal and run `yarn create react-app pokedex --template typescript`, this will create a new directory pokedex and populates it with a default react project structure and files. If all went well you should see output similar to this

```
Success! Created pokedex at
Inside that directory, you can run several commands:

  yarn start
    Starts the development server.

  yarn build
    Bundles the app into static files for production.

  yarn test
    Starts the test runner.

  yarn eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd pokedex
  yarn start

Happy hacking!
Done in 45.15s.
```

Inside your pokedex folder you should have a file structure that looks like this:

```
pokedex
|_ public
  |_ ...
|_ src
  |_ ...
|_ package.json
|_ tsconfig.json
|_ README.md
```

Let's quickly discuss the structure and some of the files in our pokedex folder:

- **public**/ this folder contains our static files like index.html, favicon, etc.
- **src**/ this folder contains all your source that makes up your application.
- **package.json** this file contains the details of your project including but not limited to dependencies, tslint config, build scripts,...
- **tsconfig.json** in order to compile typescript we need a tsconfig file telling how to compile the project.

So now that we have our project setup under our belts, let's get started by creating our first component.

# II. Components & Containers

In the next chapters we'll focus on building the UI with React, we'll do so by implementing the component/container pattern. By doing so we'll make a clear seperation of presentational components and logical components.

| | |
|---|---|
| **NOTE** | *Seperation of Concerns* is a design principle for seperating your code into distinctive sections or concerns. In our case presentational vs logical. |

# Chapter 3. First Component: PokeListItem

Before we start building our first component we need to make a clear distinction between representational components and logical components.

- **Presentational Components**
  - Main concern: how things look
  - Usually contain both DOM markup and styles of their own
  - Have no dependencies on other part of the application like state
- **Logical Components**
  - Main concern: how things work
  - Is composed of presentational components, can but mostly don't have DOM markup
  - Loads data from the store and provide it to the presentational components

## 3.1. Component: PokeListItem

The first component we are going to create is a PokeListItem. This is just a styled button that handles an onClick event.
Since this is a presentational component it needs to receive all it's props from the Container above and bubble it's events upwards.
We'll start by creating this component with basic html styling, afterwards we'll refac this to an implementation with styled-components.

### 3.1.1. Styling with css

- Create a new folder `src/component/PokoListItem`
- Create a new file inside this folder `pokelistitem.css` with following content in it.

src/components/PokeListItem/pokelistitem.css

```
.pokelistitem {
  width: 120px;
  height: 120px;
  border: none;
  border-radius: 5px;
  box-shadow: 0 2px 2px 0 rgba(0, 0, 0, 0.16);
  margin: 10px;
  background-position: center;
  background-repeat: no-repeat;
}
```

- Create a new file inside this folder `index.tsx` with following content

src/components/PokeListItem/index.tsx

```
import React, { Component } from 'react'; ①
import 'pokelistitem.css'; ②

export interface PokeListItemProps { ③
    sprite: string;
    onClick: (event?: MouseEvent) => void ④
}

class PokeListItemComponent extends Component<PokeListItemProps, {}> { ⑤

    render() { ⑥
        return (
            <button
                role="button"
                className="pokelistitem"
                style={{
                    backgroundImage: this.props.sprite ⑦
                }}
            />);
    }
}

export default PokeListItemComponent; ⑧
```

① Every tsx/jsx file starts with importing the default export of react, this is neccesary for the transpilation process. During this process JSX is wrapped inside `React.createElement` and without this import we would get errors. We also import the Component class for later use.

② import your css file.

③ Define the properties your component will receive

④ onClick is a Function that receives an optional parameter event of type MouseEvent and has return type void.

⑤ In React there are multiple ways of creating a component, here we create a class that extends

the Component class of react. In order to make our component aware of the shape of it's properties and state (more on that in chapter 5) we have to add the correct types. In this case our properties will have the shape of PokeListItemProps, but since we don't use state we pass an empty object.

⑥ Every react component has a render function, this function returns, in a descriptive way, how our component will render in the dom. This syntax is called JSX and is a combination of javascript and XML. Being an extension of XML it follows basic XML rules of closing elements and having only one top element.

⑦ Every react class has access to it's properties through `this.props`, the shape of this properties object is the same as the type we used to type our Component.

⑧ Exporting our PokeListItem as a module with a default export.

And there you have it, our first React component. In the next part we'll refactor this piece of code to use Styled Components instead of css for styling.

## 3.1.2. Styling with Styled Components

Now that we have our first component, styled with css, we'll go over it once more and refactor it using styled components.
One of the drawbacks of styling with css is that your component and it's styling are split in 2 locations, which is harder to process. Furthermore if you rely on conditional styling, when using css, your components will end up with a multitude of conditional checks for the right class names.
Using Styled Components will help keep our concerns of styling and component behaviour seperated, but since this is all javascript (or in our case typescript) and Styled Components return valid React Components. We can also pass properties and do our checks directly in our component, removing the need of complex conditionals to set the correct classnames on our component. Another added benefit of Styled Components being React Elements is that we can add specific css tests for it through the use of jest-styled-components.

### Adding Styled Components to our project

Before we can begin using Styled Components we need to install 2 dependencies, the first one is the Styled Components library itself. But since we're using typescript we'll also need to add the types of Styled Components.

- `yarn add styled-components` will add the Styled Components library.
- `yarn add -D @types/styled-components` will add the types to your project for typescript to use.

### Refactoring our PokeListItem

src/components/PokeListItem/index.tsx

```
import React, { FC } from "react";
import styled from "styled-components"; ①

interface StyledPokeListItemProps { ②
  backgroundImage: string;
};

export interface PokeListItemProps {
  sprite: string;
  onClick: (event?: any) => void;
}

const StyledPokeListItem = styled.button<StyledPokeListItemProps>` ③
  width: 120px;
  height: 120px;
  border: none;
  border-radius: 5px;
  box-shadow: 0 2px 2px 0 rgba(0, 0, 0, 0.16);
  margin: 10px;
  ${({ backgroundImage }) => `background-image: url(${backgroundImage});`} ④
  background-position: center;
  background-repeat: no-repeat;
`;

const PokeListItem: FC<PokeListItemProps> = ({ sprite, onClick })  => ( ⑤
  <StyledPokeListItem backgroundImage={sprite} onClick={onClick} role="button"/>
);

export default PokeListItem;
```

① When we're using styled components to create components we'll mostly do so by using a function from styled components. In order to do so we must import the default export from styled components.

② On our css component we've used the style attribute to set the background image, since we're using styled components that's no longer necesarry as we can pass it as a prop. But in order to do so we have to define a property interface for our styled component.

③ This is the way to create a button DOM element that's been styled with Styled Components, when we pass a generic to the function, we're defining the shape of the props, which we can use later to set dynamic styling.

④ Here we'll use *String Interpolation* to set a dynamic style, we'll extract the backgroundImage from our properties with object deconstruction and use it to create our css string.

⑤ Another big difference is the way we've created our component, we've changed from a class component to a Functional Component, hence the return type FC. The type FC takes one generic that defines the shape of the properties. We use object deconstruction here to extract the sprite and onClick function out our properties. A functional component doesn't have a render function, since what is returned is what's being rendered.

| NOTE | if your editor is giving you errors on the styled-components import, make sure you'll also have imported the types. |
|---|---|

## 3.2. Visual testing & documentation with Storybook

By now we've spent some time developing our first component, but we could write all the code we want, without testing it there's no way of telling it works.
So how do we go about testing our component, first we'll start by creating stories for Storybook, this will add documentation and a visual way to check your components. Afterwards we'll test the behaviour of our component through unit testing and we'll also create a snapshot test, which tests the rendering of the component.

### 3.2.1. Setting up storybook

So let's dive right in and add Storybook to our project. Luckily for us, Storybook has a handy CLI tool for adding Storybook to existing projects. This removes the need of manually adding the correct dependencies and doing the setup of Storybook.
Run `npx sb init` in the root directory of your project. At the time of writing both Storybook and create-react-app come with a dependency on babel-loader, however the versions don't match and this will throw compilation errors. To resolve this issue add a resolutions block in your package.json with the correct version to babel-loader.

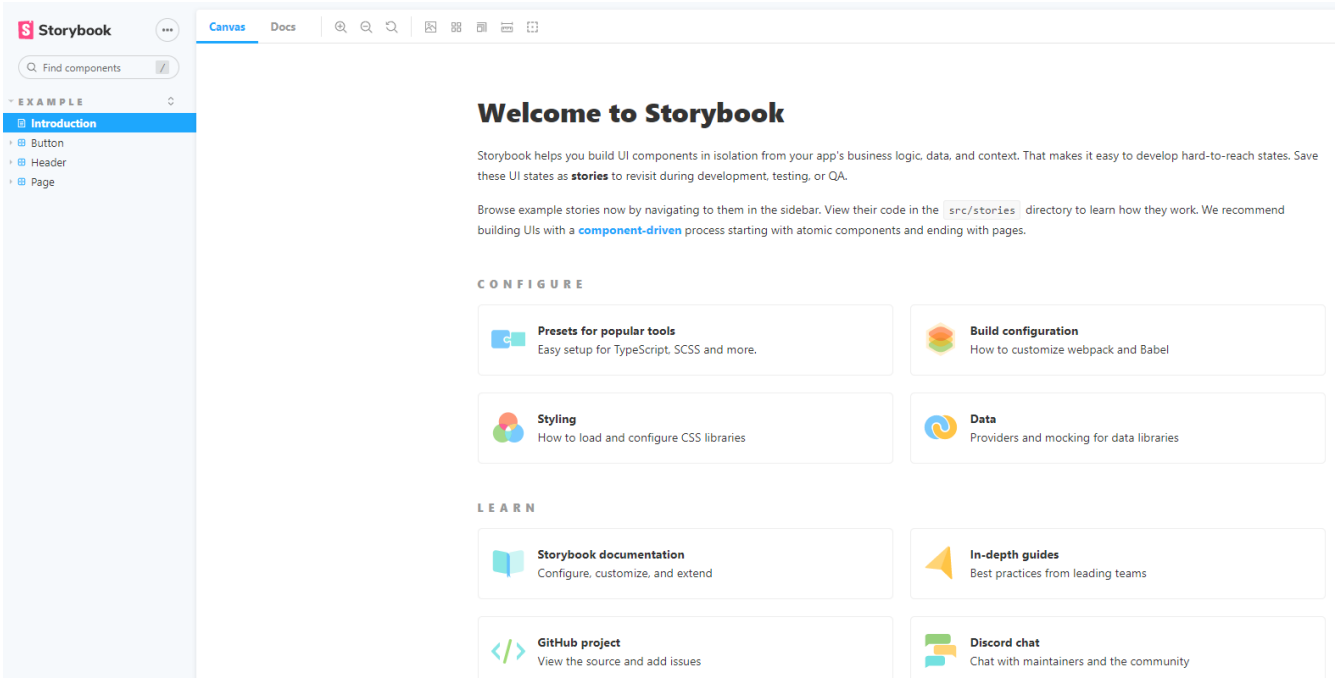
package.json

```
{
  ...rest omitted for brevity
  "resolutions": {
    "babel-loader": "8.1.0"
  }
}
```

Storybook itself comes with a couple of examples so we can test our installation straight out of the box. Run `yarn storybook` to start storybook in development mode, this will add a live reload server so you don't have to restart after every change.
If everything went well you should see output like this:

```
|   Storybook 6.3.1 started                          |
|   11 s for preview                                 |
|                                                    |
|    Local:            http://localhost:6006/        |
|    On your network:  http://192.168.0.176:6006/    |
```

If you go to http://localhost:6060 and are greeted by following screen everything went well and Storybook is ready to be used.




### 3.2.2. Writing our first story

foobar

```
import React from 'react';
import { Story, Meta } from '@storybook/react';

import PokeListItem, {PokeListItemProps} from '.';

export default {
    title: "Components/PokeListItem",
    component: PokeListItem,
    argTypes: {onClick: {action: 'clicked'}}
} as Meta; ①

const Template: Story<PokeListItemProps> = (args) => <PokeListItem {...args} />; ②

export const Default = Template.bind({}); ③
Default.args = {
    sprite:
"https://raw.githubusercontent.com/PokeAPI/sprites/master/sprites/pokemon/1.png",
} ④
```

① foobar

② foobar

③ foobar

④ foobar

## 3.3. Quality through testing

### 3.3.1. Testing components with Jest

### 3.3.2. Snapshot testing with Jest

# Chapter 4. Composed Components: building components out of components

In the previous chapter we saw how we can build a simple standalone component, although pretty basic this example already showed how versatile a tool like React can be.
Thanks to tooling like Storybook we have a visual tool to show and document our components with and with Jest we made sure our components are tested.

## 4.1. Children

As you saw in the previous chapter, React components can be composed from different DOM elements. However if we always had to create react components out of DOM elements the reusability would soon be non-existent. So how exactly can we create composed components without having to repeat ourselves?
*Children!* Children are a special kind of prop in React that represent content that can be included in your component. This property accepts everything that is a valid React Element, every React component, DOM-element and String is a valid child of a react component.

| | |
|---|---|
| **NOTE** | Writing reusable code is one of the most important principles in software development. It is aimed at reducing the repetition of software patterns through means of abstraction. This principle is commonly known as the DRY principle (Don't Repeat Yourself) |

Take a look at the examples below and see for yourself:

```
const ChildComponent: FC<any> = () => (<h1>I am a child component.</h1>);

const ComponentWithChildren: FC<PropsWithChildren<any>> = ({children}) => ( ①
    <div>
        {children} ②
    </div>
);

const DomElementChildren: FC<any> = () => (
    <ComponentWithChildren>
        <p>FooBar paragraph</p> ③
    </ComponentWithChildren>
);

const StringChildren: FC<any> = () => (
    <ComponentWithChildren>
        Hello World! ④
    </ComponentWithChildren>
);

const ReactComponentChildren = () => (
    <ComponentWithChildren>
        <ChildComponent />
        <ChildComponent /> ⑤
    </ComponentWithChildren>
);
```

① Before we can start using our children we first need to make typescript aware we are using PropsWithChildren as a type, this is a convenience type that's located in the React package. This type also takes a generic, in this case any, to represent the props other than children. We use object deconstruction to extract the children out our properties.

② In order for React to know where to render the children we need to declare the correct position of the children.

③ This is an example of how to pass DOM elements as children to a React component.

④ This is an example of how to pass a String to a react component as a child.

⑤ This is an example of how to pass another React component as a child to the parent React component, as shown here, you are not restricted to just passing 1 child, you can add multiple children, as well as a combination of all 3 different React Elements.

**NOTE** Using children to compose more complex structures is called *composition*.

## 4.2. Component: PokeList

As an example of a composed component we'll be building the PokeList, this is a simple component that provides a surrounding div with a background image. This div will hold our PokeListItems we created in the previous chapter.
We'll build this component in 2 different ways, just like in the previous chapter. Why? Because just as with styling React isn't opinionated on how you build your components. So in our first example we'll build a PokeList component that accepts children and we'll pass our PokeListItems as children. In the second example we'll pass an array of pokemons which will be used to generate the children.

Before we begin, create a new folder under `src/components` called *PokeList* with a file index.tsx inside.

### 4.2.1. PokeList: creating a component that accepts children

We'll start by creating the first example, where we'll pass the PokeListItems from the previous chapter as children.
First we'll need the PokeList Component, as stated in the previous chapter you're free to choose how you do your styling (css/SASS/styled-components/...), but for the sake of consistency we'll use styled-components here and every other example throughout this tutorial.

src/components/PokeList/index.tsx

```
import React, { FC, PropsWithChildren } from "react";
import styled from "styled-components";

const StyledPokeList = styled.div`
  background-color: #bff9ff;
  box-shadow: inset 0 0 1.25rem rgba(0, 0, 0, 0.5);
  padding: 1rem;
  display: flex;
  justify-content: space-between;
`;

const PokeList: FC<PropsWithChildren<any>> = ({children}) => (
    <StyledPokeList>
        {children}
    </StyledPokeList>
);

export default PokeList;
```

We then can use this component in the following fashion:

```
const pokemons: any[] = [
    {
        id: 1,
        sprite:

"https://raw.githubusercontent.com/PokeAPI/sprites/master/sprites/pokemon/1.png",
      },{
        id: 2,
        sprite:

"https://raw.githubusercontent.com/PokeAPI/sprites/master/sprites/pokemon/2.png",
      },
  ];

const PokeOverview = () => (
    <PokeList>
        { pokemons.map(pokemon => (
                <PokeListItem
                    sprite={pokemon.sprite}
                    onClick={() => {console.log(`you clicked pokemon: ${pokemon}`)}}
                    key={pokemon.id} ①
                />
            ));
        }
    </PokeList>
);
```

① When creating react components through a map function you have to provide a unique key to

every react component you created. In our case it's the pokemon ID.

### 4.2.2. PokeList: passing properties

Now that we've seen how we can build a component with children through composition, we'll now take a look at how to render our PokeListItems when passing properties. This approach has some advantages over the method with passing children, most important we can typecheck the properties we're expecting to receive.
In our case, since Pokemon is a complex type, we'll have to create an interface first to define our type, for now we'll keep it together with our component but in a later phase we'll move this into it's own file.

src/components/PokeList/index.tsx

```
import React, { FC } from "react";
import styled from "styled-components";
import PokeListItem from "./../PokeListItem"; ①

export interface Pokemon { ②
  id: number;
  sprite: string;
}

export interface PokeListProps {
  pokemons: Pokemon[]; ③
}

const PokeListFrame = styled.div`
  background-color: #bff9ff;
  box-shadow: inset 0 0 20px rgba(0, 0, 0, 0.5);
  padding: 1rem;
  display: flex;
  justify-content: space-between;
`;

const handlePokeListItemClick = () => { ④
  console.log("you have clicked a poke list item!");
};

const PokeList: FC<PokeListProps> = ({ pokemons }) => ( ⑤
  <PokeListFrame>
    {pokemons.map((pokemon: Pokemon) => ( ⑥
      <PokeListItem
        sprite={pokemon.sprite}
        onClick={handlePokeListItemClick}
        key={pokemon.id}
      />
    ))}
  </PokeListFrame>
);

export default PokeList;
```

① Import the PokeListItem component into our PokeList component.

② We first define the structure of our complex Pokemon type through an interface.

③ We define pokemons as an array of Pokemon type objects in our component's properties interface.

④ Since we moved everything inside the pokelist, we also have to create an onClick handler to pass down to the PokeListItem.

⑤ We pass PokeListProps as type to FC and extract the pokemons property out of it using object deconstruction.

⑥ The map function that renders the children is now included in the PokeList component, by doing this we move all responsibilities concerning the rendering of this component's child components to the PokeList component. This makes our code less error prone and more future proof.

After this refactoring we can utilize this component in a more concise way, like this:

```
const pokemons: Pokemon[] = [
    {
        id: 1,
        sprite:

"https://raw.githubusercontent.com/PokeAPI/sprites/master/sprites/pokemon/1.png",
      },{
        id: 2,
        sprite:

"https://raw.githubusercontent.com/PokeAPI/sprites/master/sprites/pokemon/2.png",
      },
  ];

const PokeOverview = () => (
    <PokeList pokemons={pokemons} />
);
```

| NOTE | When splitting a piece of software in distinct sections of code that each addresses it's own responsibility we talk about *Separation of Concerns*. This is a design principle in software development that makes your code modular and reusable. i.e.: presentational components vs logical components |
|---|---|

# 4.3. Quality, Quality, Quality

As stated in the previous chapter, quality is often an overlooked aspect of software development. However, if you, by accident, break a piece of code you've written 3 months ago that didn't had automated testing on it. It's a real PITA to fix.
So do yourself a favor and create a habit of writing tests/documentation right after you've written your component (or even better before, if you're following TDD principles).

| NOTE | TDD uitleggen XD |
|---|---|

## 4.3.1. Documentation with Storybook

## 4.3.2. Snapshot testing with Jest

# Chapter 5. Stateless vs Stateful Components

# III. State management & Middleware

# Chapter 6. redux stuff

# Chapter 7. Middleware

## 7.1. Redux Thunk

## 7.2. React router

# Chapter 8. What’s next?

## 8.1. Securing your application

## 8.2. Serverless