Programowanie Nowoczesnych Aplikacji Internetowych

Frontend w React'ie
dla aplikacji wykonanej w framework'u Spring

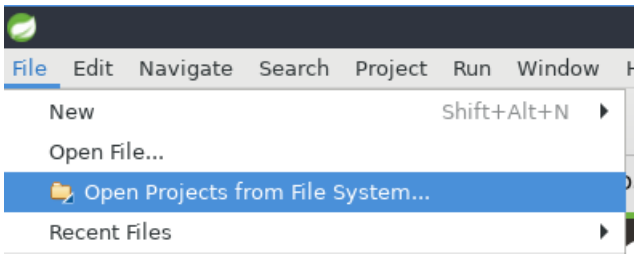
Krzysztof Kamycki, KIiA Prz, 2019

Spis treści

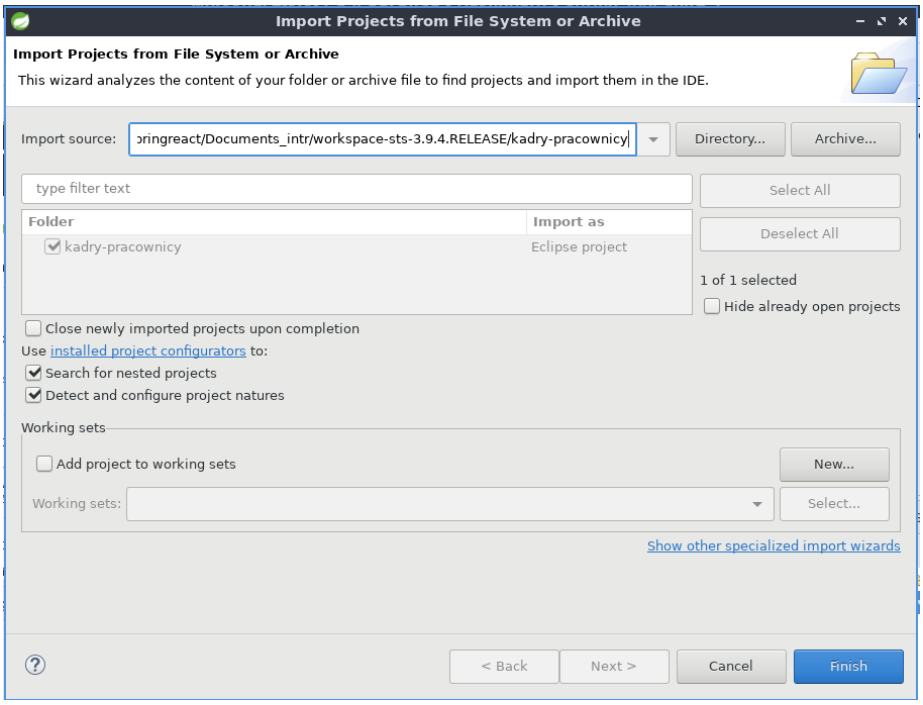
1. Konfiguracja aplikacji backend'owej

Przed uruchomieniem aplikacji, konieczne jest odtworzenie bazy danych, która była używana podczas implementacji backend'u. Proces ten został dokładnie opisany w instrukcji z pierwszych zajęć na temat framework'u Spring. Należy utworzyć nowego użytkownika i bazę danych zgodnie z podpunktem nr 6, skonfigurować połączenie z bazą danych wewnątrz Spring Tool Suite (utworzona została przykładowa konfiguracja, wystarczy ją zduplikować, rozłączyć przykład i w kopii ustawić odpowiednie dane użytkownika i nazwy bazy danych). Gdyby była konieczność utworzenia nowego połączenia, plik mysql-connector-java znajduje się pod ścieżką */home/springreact/sts-bundle/libs/mysql*.

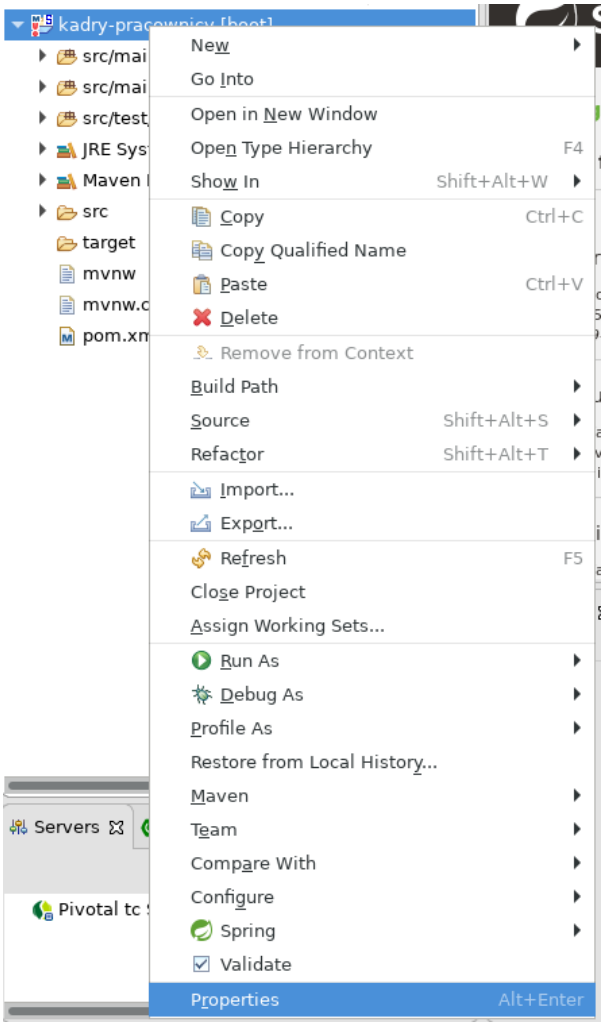
Następnie można wczytać projekt aplikacji. Można to zrobić poprzez wybranie w menu File -> Open Projects from File System…



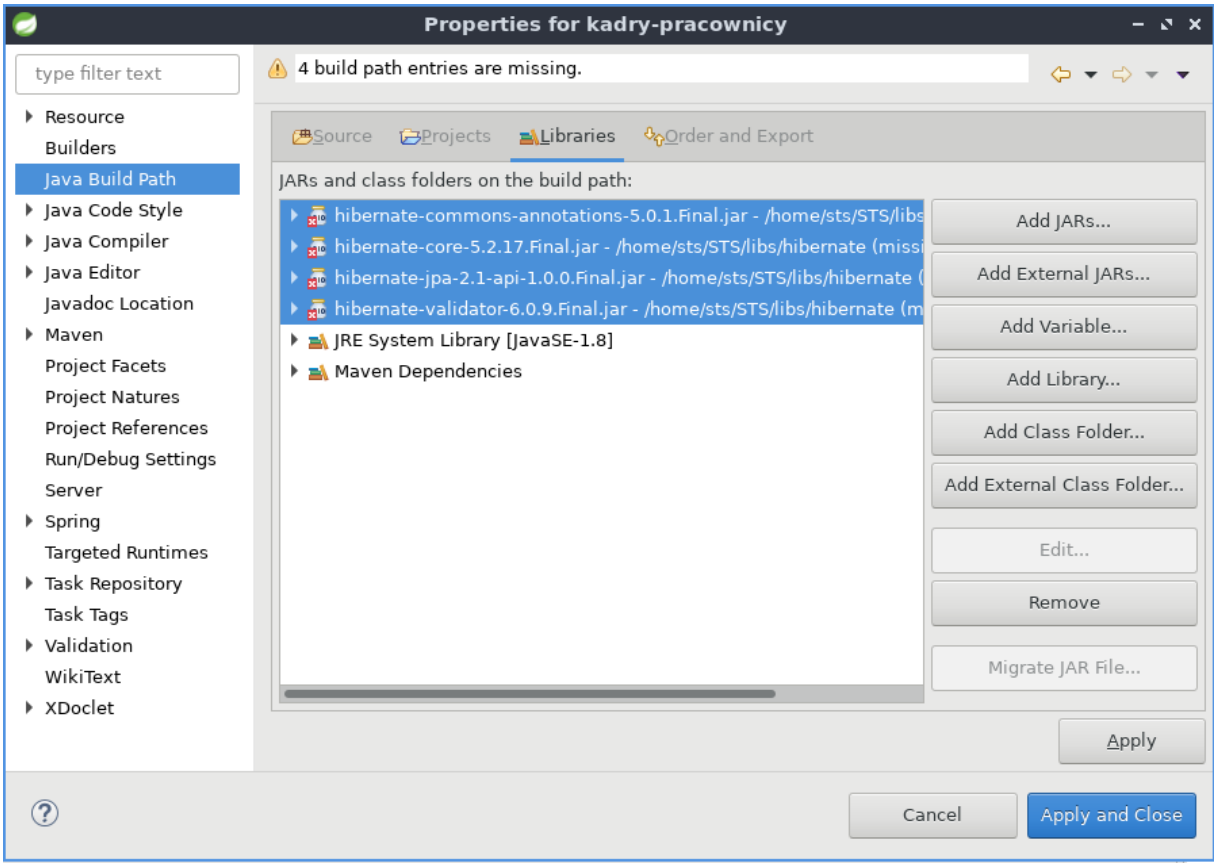
Powinno pojawić się okno widoczne poniżej. Po naciśnięciu przycisku *Directory…*, należy wybrać folder z projektem (domyślnie nazywa się kadry-pracownicy), nacisnąć OK, a następnie Finish.



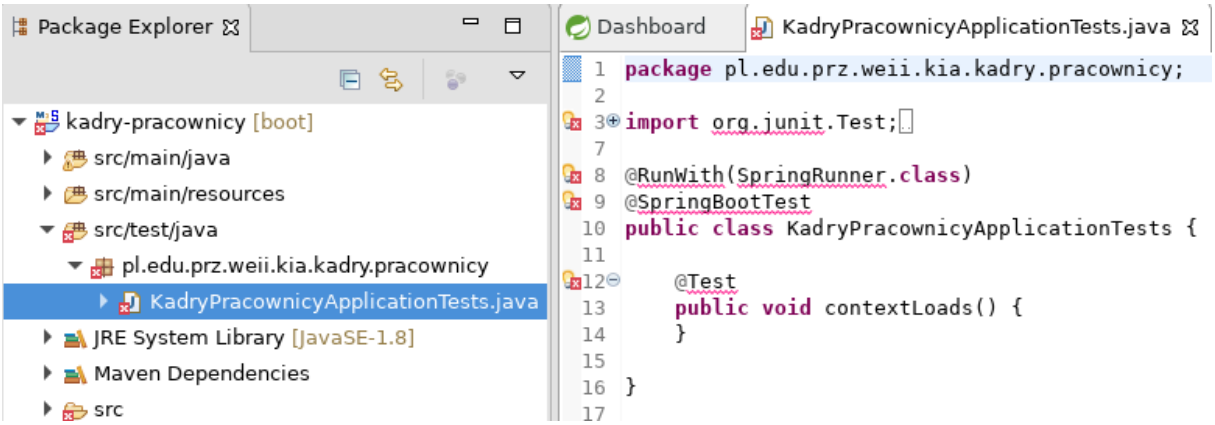
Aplikacja nie powinna jeszcze działać, ponieważ ścieżki do plików biblioteki Hibernate nie są poprawnie ustawione. Wszystkie ścieżki powinny być skonfigurowane w STS, podobnie jak było to opisane na wcześniejszych zajęciach, dlatego wystarczy wejść we właściwości projektu.



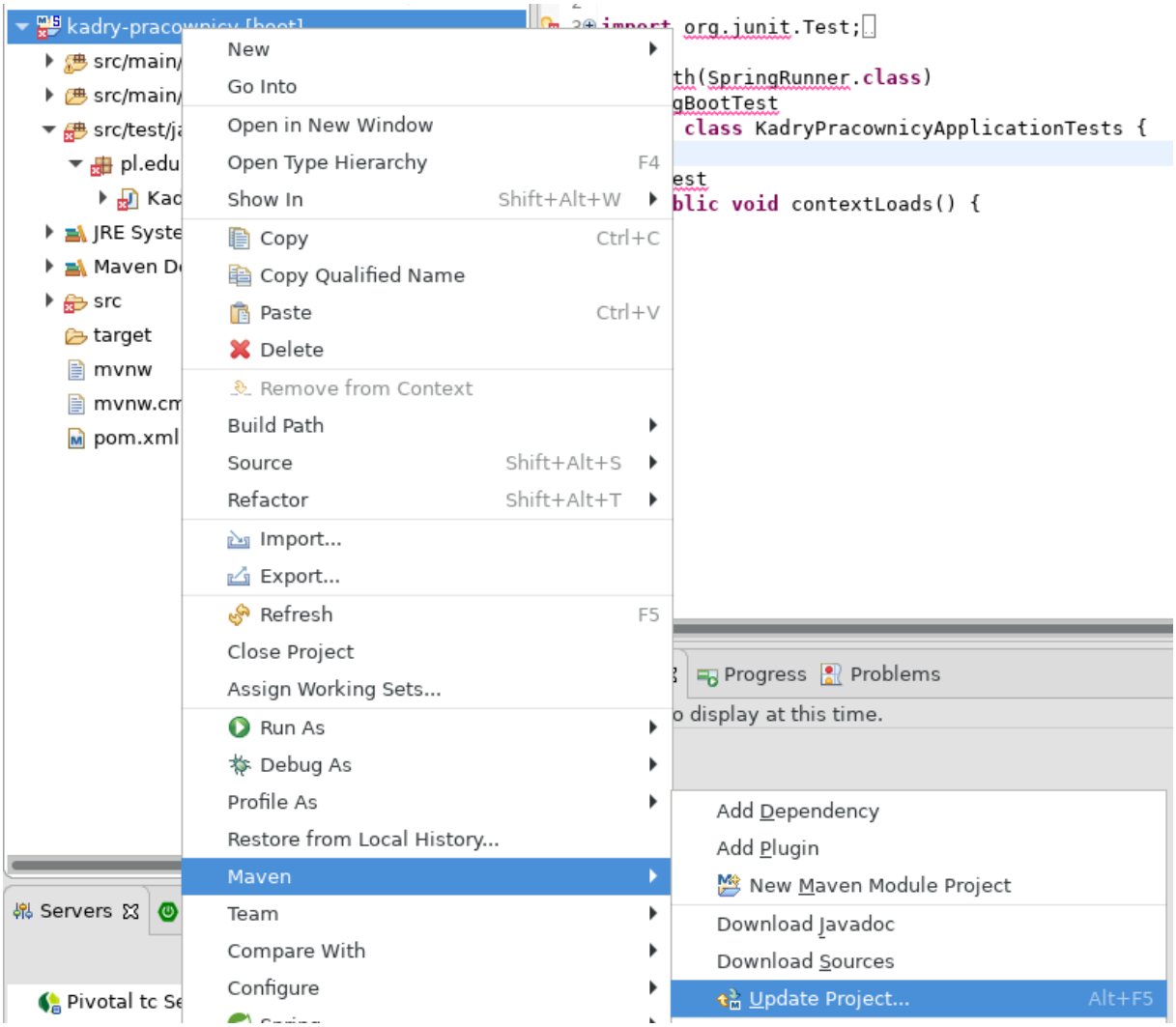
Następnie wybierz pozycję *Java Build Path*, kliknij zakładkę Libraries, zaznacz wszystkie cztery pliki i naciśnij *Remove*.



Po imporcie projektu ze starszej wersji STS, mogą wystąpić błędy jak np. te widoczne poniżej.



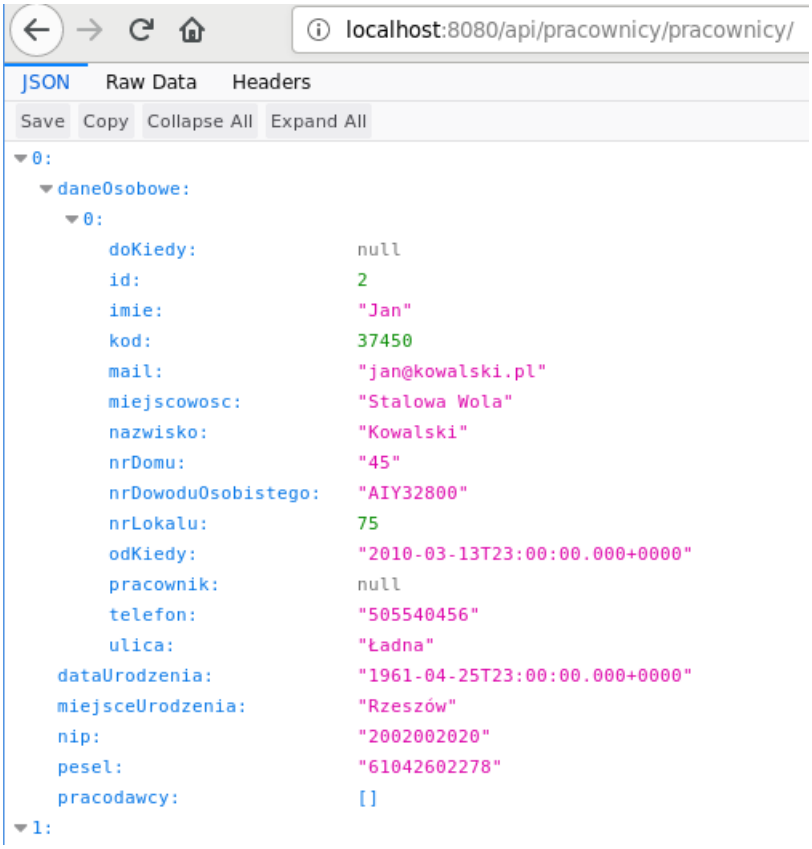
W celu zaktualizowania projektu, naciśnij prawym na projekcie i wybierz Maven->Update Project…, a następnie upewnić się, czy projekt jest zaznaczony i nacisnąć OK.



Ostatnim krokiem konfiguracji jest sprawdzenie, czy dane bazy danych wewnątrz pliku *application.properties* (w folderze *src/main/resources*) są poprawne.

Gdyby okazało się, że projekt nie może znaleźć plików biblioteki Hibernate, skonfiguruj je zgodnie z podpunktem nr 8 instrukcji z wcześniejszych zajęć. Wszystkie konieczne pliki znajdziesz pod ścieżką */home/springreact/sts-bundle/libs/hibernate*.

Jeśli aplikacja działa poprawnie, wpisując w przeglądarce lub w Postmanie adres localhost:8080/api/pracownicy/pracownicy, powinniśmy zobaczyć listę pracowników.

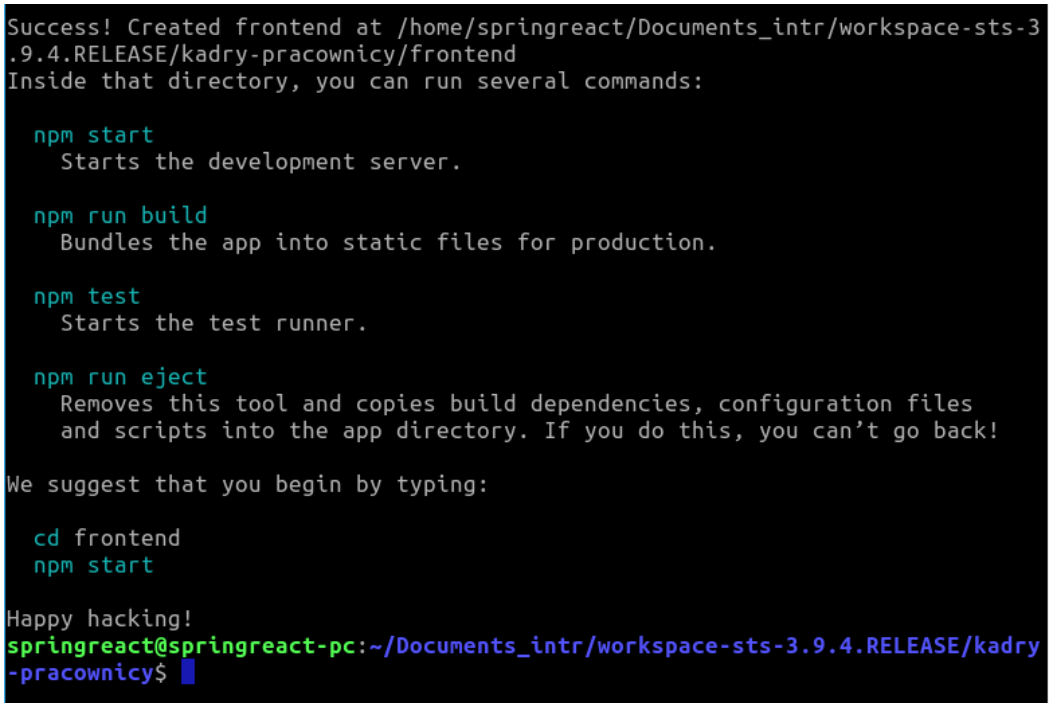


2. Utworzenie nowej aplikacji React

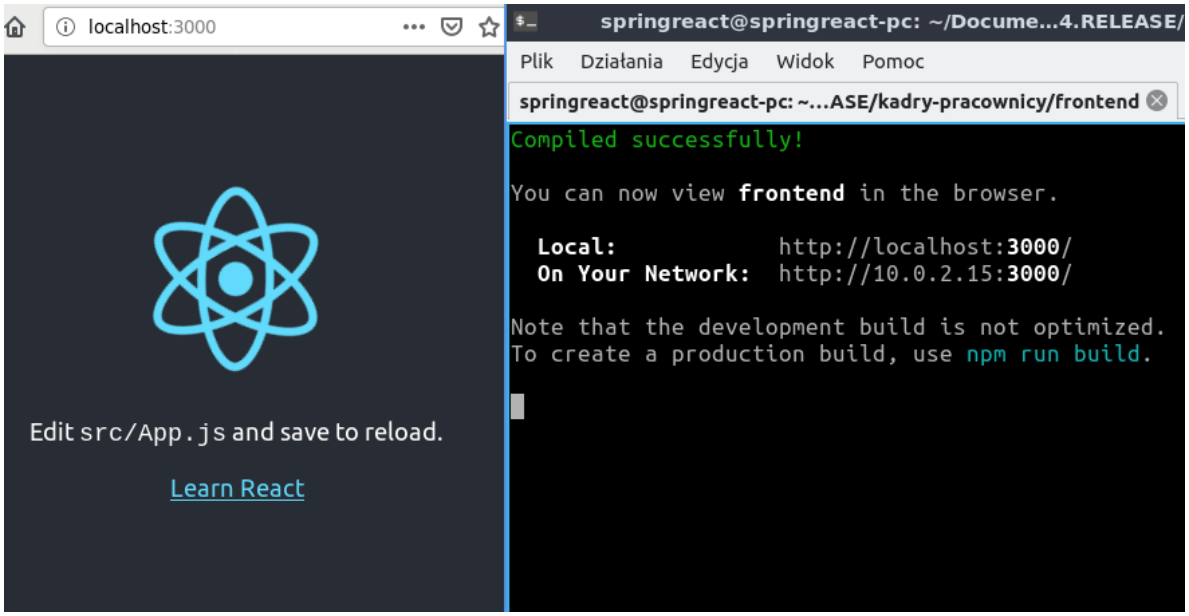
Do utworzenia nowej aplikacji React konieczne jest wykorzystanie terminalu. Można go wyszukać lub wybrać w menu aplikacji -> Narzędzia systemowe -> QTerminal. Następnie wybieramy folder projektu aplikacji backend'owej (nazwa „kadry-pracownicy", w nim powinien się znajdować m.in. plik pom.xml), a następnie tworzymy projekt aplikacji React poleceniem:

```
npx create-react-app frontend
```

,gdzie *frontend* będzie nazwą aplikacji. Jeśli generowanie szkieletu aplikacji ukończyło się pomyślnie, powinniśmy zobaczyć poniższy komunikat.



Przechodząc do folderu *frontend* i wpisując `npm start` możemy sprawdzić, czy aplikacja działa poprawnie. **Nie trzeba samemu uruchamiać przeglądarki**, wystarczy wpisać polecenie i poczekać. Po chwili powinniśmy zobaczyć aplikację oraz odpowiedź i powodzeniu kompilacji.



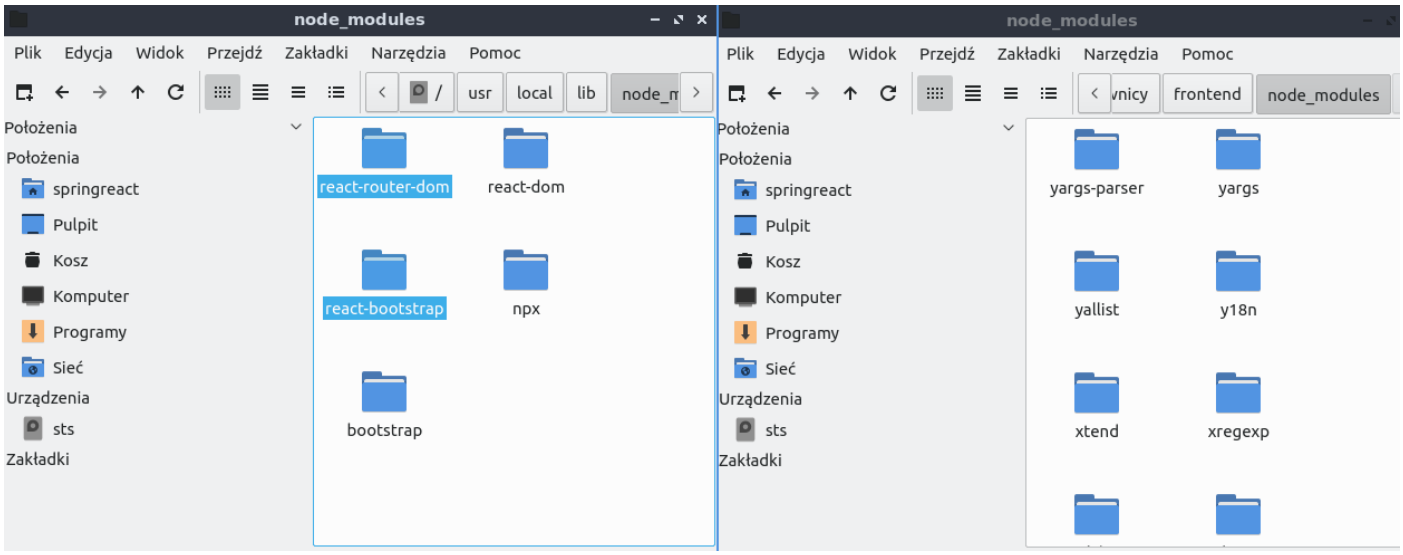
3. Konfiguracja aplikacji React

Wyłącz aplikację naciskając kombinację klawiszy Ctrl+C w terminalu. Następnie otwórz IDE o nazwie Atom, najprościej poprzez skrót na pulpicie. Następnie otwórz folder *frontend* i wyświetl zawartość pliku *package.json*. Aplikacja będzie korzystała z bibliotek *bootstrap*, *react-bootstrap* i *react-router-dom*, dlatego trzeba je dopisać do *dependencies*.

```
"dependencies": {
    "bootstrap": "^4.3.1",
    "react": "^16.8.6",
    "react-bootstrap": "^1.0.0-beta.8",
    "react-dom": "^16.8.6",
    "react-router-dom": "^5.0.0",
    "react-scripts": "3.0.1"
  },
```

Wszystkie biblioteki zostały zainstalowane globalnie, dlatego ręcznie musimy je dopisać. Jeśli biblioteki nie byłyby ściągnięte wcześniej, można by je ściągnąć używając polecenia np. `npm install bootstrap --save`. Gdyby przez pomyłkę pominęło się parametr `--save`, to biblioteka zostałaby ściągnięta, ale nie byłaby dopisana do *dependencies* i konieczne byłoby jej ręczne dopisanie.

Niestety, biblioteki *react-bootstrap* i *react-router-dom* działają jedynie, gdy są zainstalowane lokalnie w projekcie. Wejdź do folderu */usr/local/lib/node_modules*, skopiuj foldery o nazwach *react-bootstrap* i *react-router-dom* i wklej je do */frontend/node_modules*.



Poza bibliotekami, na końcu pliku dopiszemy parametr *proxy* i ustawimy go na http://localhost:8080 (**nie zapomnij o przecinku po ostatnim obiekcie)**. Skróci to długość zapytać do aplikacji backend'owej w dalszej części instrukcji.

```
      "last 1 safari version"
    ]
  },
  "proxy": "http://localhost:8080"
}
```

Następnie otwórz plik *public/index.html* i wewnątrz znacznika *head* dodaj pliki css biblioteki Bootstrap przy pomocy:

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
```

Wewnątrz tego pliku możemy znaleźć div'a o id „root". W tym miejscu znajdzie się nasza aplikacja. Możemy to sprawdzić (lub w przyszłości zmienić) wewnątrz pliku *index.js*. Przy pomocy linii

```
ReactDOM.render(<App />, document.getElementById('root'));
```

wybieramy miejsce renderowania aplikacji.
Na koniec do pliku *App.css* dopisz:

```
#root{
  padding: 5px;
}

input[type="date"].form-control{
  line-height: 1.42857143;
}
```

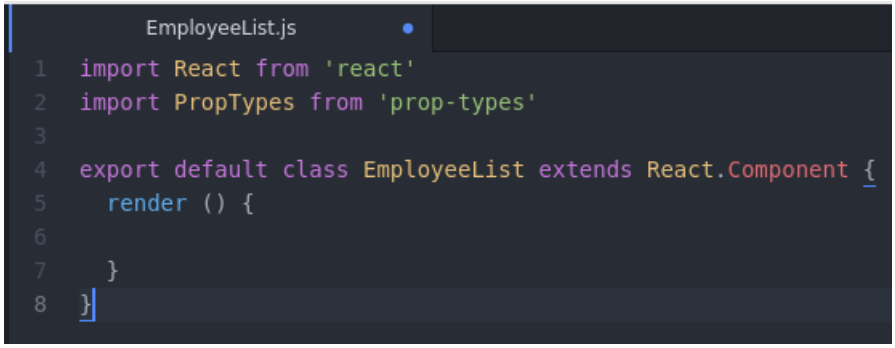
Gdybyśmy pominęli ten krok, tekst wewnątrz kontrolek na daty nie byłby wyśrodkowany.

4. Przygotowanie routingu aplikacji i podstron

Po tym jak projekt został skonfigurowany, można ponownie włączyć aplikację poleceniem `npm start`. Wszystkie zmiany, które wprowadzimy w plikach projektu, powinny być od razu kompilowane (w konsoli możemy zobaczyć błędy) oraz widoczne w przeglądarce po odświeżeniu strony.

Wewnątrz folderu *src* utwórz nowy plik o nazwie *EmployeeList.js*. Wewnątrz utwórz klasę o tej samej nazwie. W tym celu możesz wykorzystać snippet i edytorze wpisać skrót *rcd*.

Poprzez *export default* określana jest klasa, która będzie importowana, gdy dany plik zostanie zaimportowany przez inną klasę. Ten zapis jest równoważny do dopisania *export default* przed deklaracją klasy.



Klasa ta będzie odpowiedzialna za pobieranie i wyświetlanie pracowników, ale na ten moment jedynie wyświetlimy nazwę klasy. Ponadto można usunąć import PropTypes, ponieważ nie będzie ona używana.

```
import React from 'react';

export default class EmployeeList extends React.Component {
  render(){
        return <h2>Lista pracowników</h2>
  }
}
```

Następnie utwórz klasę *CreateEmployee.js*. Klasa będzie wyglądała identycznie jak *EmployeeList*. Z tego powodu możesz zduplikować plik, zmienić jego nazwę i podmienić tekst do wyświetlenia na „Utwórz nowego pracownika”.

```
import React from 'react';

export default class CreateEmployee extends React.Component {
  render(){
        return <h2> Utwórz nowego pracownika </h2>

  }
```

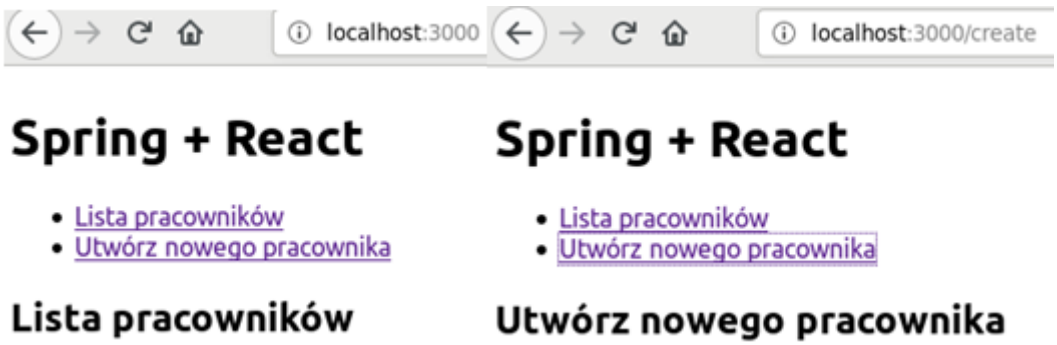
Po zapisaniu obu plików, otwórz *App.js*. Usuń import logo i dopisz import obiektów z biblioteki *react-router-dom* oraz klas, które utworzyliśmy.

```
import React from 'react';
import {
  BrowserRouter as Router,
  Route,
  Link
} from 'react-router-dom';
import './App.css';

import EmployeeList from './EmployeeList'
import CreateEmployee from './CreateEmployee'
```

Podmień funkcję na klasę App, usuń ciało funkcji *render* i przepisz poniższy kod JSX.

```
class App extends React.Component {
    render() {
      return (
        <div>
          <h1>Spring + React</h1>
          <Router>
            <div>
              <ul>
                <li>
                  <Link to="/">Lista pracowników</Link>
                </li>
                <li>
                  <Link to="/create">Utwórz nowego pracownika</Link>
                </li>
              </ul>
              <Route exact path="/" component={EmployeeList} />
              <Route path="/create" component={CreateEmployee} />
            </div>
          </Router>
        </div>
      );
    }
}

export default App;
```

Po zapisaniu wszystkich zmian, w przeglądarce powinniśmy widzieć działającą aplikację. Po naciśnięciu odpowiednich linków, zawartość strony powinna się zmieniać.



Obiekty *Link* pozwalają nam na tworzenie linków, które będą współpracowały z resztą elementów biblioteki *react-router-dom*. Wpisujemy adres podstrony jako wartość atrybutu *to*, a pomiędzy znacznikami wstawiamy tekst do wyświetlenia.
Obiekt *Route* łączy ścieżkę podaną w atrybucie *path* z odpowiednimi klasami, które podajemy wewnątrz *component*. Wartość podajemy wewnątrz klamerek, ponieważ jako wartość przekazujemy klasę, a nie tylko jej nazwę.
Obiekty *Link* i *Route*, działają jedynie wtedy, gdy znajdują się pomiędzy znacznikami *Router*.

5. Wyświetlanie listy pracowników

Otwórz plik *EmployeeList.js* i dodaj konstruktor do klasy.

```
export default class EmployeeList extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      employees: []
    };
  }
```

Pole *state* pozwala nam na zapisywanie własnych zmiennych do klasy. Za każdym razem, gdy zmieniana jest jakakolwiek wartość wewnątrz *state*, wymuszone zostaje ponowne wyrenderowanie strony. W naszym wypadku, przygotowujemy pustą listę na pracowników.

Następnie utwórz metodę *getEmployeeList()*. Jej zadaniem będzie odczytanie z backend'u listy pracowników, sprawdzenie czy operacja się powiodła i wypełnienie wcześniej przygotowanej pustej listy otrzymanym wynikiem. Przed wykonaniem zapytania http, ustawiona zostanie flaga informująca, że lista jest wczytywana. Implementacja tej metody może wyglądać następująco.

```
getEmployeeList(){
    this.setState({isLoading: true});

    fetch('/api/pracownicy/pracownicy')
    .then(response => {
      if(response.ok){
        return response.json();
      } else {
        throw new Error("Odczytanie listy pracowników nie powiodło się, spróbuj później.")
      }
    }).then(data => {
      this.setState({employees: data, isLoading: false});
      //console.log(this.state.employees);
    }).catch(error => {
        window.alert(error.message);
    });
  }
```

Na początku ustawiamy flagę, następnie wysyłamy zapytanie. Przez to, że w konfiguracji ustawiliśmy parametr *proxy*, nie musimy podawać całego adresu wraz z portem. Funkcja *fetch* działa asynchronicznie i zwraca tzw. Promise. Do obsłużenia obiektu Promise można wykorzystać metodę *then*, która wykona się po zakończeniu poprzedniej funkcji. Następnie sprawdzany jest status odpowiedzi. Jeśli operacja się powiodła, wynik jest konwertowany do obiektu JSON, w przeciwnym wypadku tworzony jest obiekt błędu z komunikatem. Po konwersji, wynik jest zapisywany do przygotowanej wcześniej tabeli oraz flaga informująca o trwaniu operacji zostaje ponownie ustawiona na *false*. Przypominam, że każde wywołanie metody *this.setState* wymusza ponowne renderowanie strony.

Zanim przejdziemy do renderowania strony, konieczne jest wymuszenie wysłania zapytania o listę pracowników po wyświetleniu strony. W tym celu możemy nadpisać metodę:

```
componentDidMount() {
    this.getEmployeeList();
}
```

Ostatnia metoda, którą trzeba napisać to *render*. Na początku sprawdzimy czy lista jest wczytana, jeśli nie zapiszemy odpowiedni komunikat.

```
if(this.state.isLoading){
    return <p>Ładowanie...</p>;
}
```

Jeśli aplikacja zna już listę pracowników, to wyświetlimy odpowiednią tabelę.

```
return(
  <div>
    <h2>Lista pracowników</h2>
    <table className="table table-striped">
      <thead>
        <tr>
          <th>Imię</th><th>Nazwisko</th><th>Pesel</th>
        </tr>
      </thead>
      <tbody>{employees}</tbody>
    </table>
  </div>
);
```

Metoda *render* musi zwracać jeden element, dlatego nagłówek z tabelą zostały otoczone *div'em*. Wynikiem jest kod JSX, a nie HTML , dlatego zamiast *class* podajemy atrybut *className*. Na ten moment zmienna *employees* nie jest zdefiniowana, natomiast *this.state.employees* to obiekt JSON, który nie zostałby tutaj poprawnie wyświetlony. Z tego powodu, pomiędzy if'em, a powyższym kodem musimy zadeklarować brakującą zmienną. Wewnątrz niej będą zapisane obiekty klasy *Employee*. Klasa ta określi, w jaki sposób będzie wyświetlany każdy pracownik.

```
const employees = this.state.employees.map(employee => <Employee key={employee.pesel} employee={employee}
/>);
```

Lista nie będzie ulegała modyfikacji, dlatego jest typu *const*. Następnie przypisujemy do niej wynik funkcji mapującej, która wykona funkcję, podaną dalej, dla każdego elementu listy *employees*. Metoda *map* wczyta każdego pracownika pojedynczo i zapamiętuje go jako zmienną *employee*, a następnie utworzy obiekt klasy Employee. Metoda *map* we framework'u React wymaga, by każdy wygenerowany obiekt miał unikalny klucz, dlatego jako wartość *key* podajemy pesel pracownika, który jest też identyfikatorem pracownika w bazie danych. Następnie konieczne jest przekazanie informacji o pracowniku, dlatego przesyłamy go jako parametr o nazwie *employee*. **Nie zapomnij o odpowiednim zamknięciu znacznika Employee.**

Wiemy, że w klasie *EmployeeList* będziemy korzystali z klasy *Employee*, dlatego możemy dodać odpowiedni import na początku pliku.

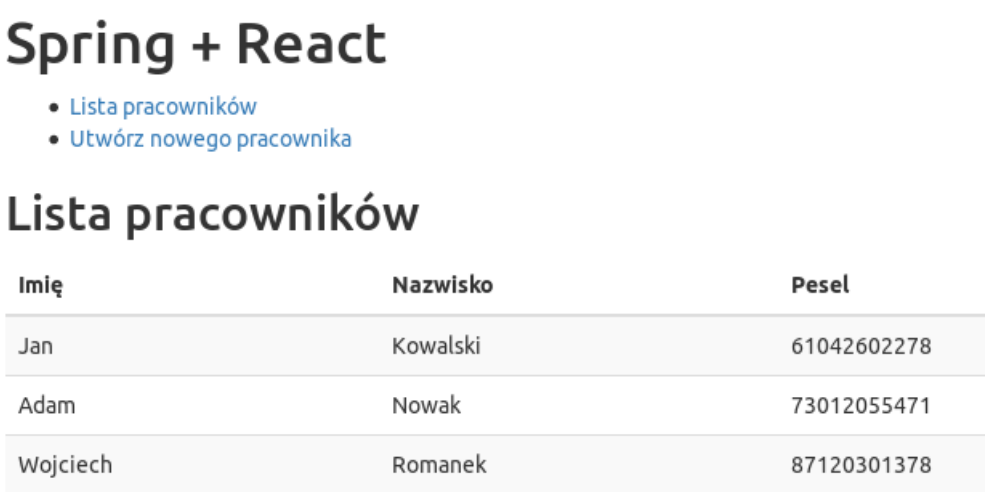
```
import Employee from './Employee';
```

Następnie możemy utworzyć nowy plik *Employee.js* z klasą *Employee*. Do wyświetlenia pojedynczego pracownika, wystarczy napisać ciało metody *render*. Dane pojedynczego pracownika zostały przekazane jako parametr podczas tworzenia obiektu klasy. Wartość parametru można odczytać wewnątrz obiektu klasy przy pomocy pola *this.props*. Można wielokrotnie odwoływać się do *this.prop.employee* albo, jak to zostało wykonane niżej, wykorzystać zmienną pomocniczą, przez co wykorzystanie tego parametru w dalszej części programu jest prostsze.
Dane osobowe pracownika są zapisane jako tablica, dlatego wyświetlany jest ostatni indeks tej tablicy.

Mamy dostęp do wszystkich danych pracownika, ale dla uproszczenia zadania, wyświetlimy tylko trzy dane. Taka implementacja mogłaby służyć do testowania aplikacji, jednak w wersji produkcyjnej konieczne byłoby udostępnienie funkcji po stronie aplikacji backend'owej, który udostępniałaby tylko wyświetlane informacje.

```
import React from 'react';

export default class Employee extends React.Component {
  render(){

    const { employee } = this.props
    const latestPersonalDataIndex = employee.daneOsobowe.length - 1;

    return (
      <tr>
        <td>{employee.daneOsobowe[latestPersonalDataIndex].imie}</td>
        <td>{employee.daneOsobowe[latestPersonalDataIndex].nazwisko}</td>
        <td>{employee.pesel}</td>
      </tr>
    );
  }
}
```

Po odświeżeniu, strona powinna wyglądać następująco:



6. Usuwanie pracownika

Celem tego podpunktu jest dodanie przycisku „Usuń” koło każdego z pracowników. Po naciśnięciu, powinien pojawić się komunikat pytający o potwierdzenie, czy na pewno dany pracownik ma zostać usunięty. Następnie należy wysłać zapytanie i, w zależności od odpowiedzi serwera, użytkownik powinien zostać przekierowany do strony z listą pracowników albo, po niepowodzeniu danej operacji, użytkownik powinien otrzymać odpowiedni komunikat z informacją.

Konieczne jest wprowadzenie zmian w klasach *EmployeeList* i *Employee*. Otwórz plik klasy *EmployeeList*. W metodzie *render* musimy jedynie wyświetlić dodatkową kolumnę, gdzie będą wyświetlane przyciski.

```
return(
      <div>
        <h2>Lista pracowników</h2>
        <table className="table table-striped">
          <thead>
            <tr>
              <th>Imię</th><th>Nazwisko</th><th>Pesel</th><th>Usuń</th>
            </tr>
          </thead>
          <tbody>{employees}</tbody>
        </table>
      </div>
    );
```

W pierwszym akapicie było wspomniane, że po usunięciu pracownika, lista powinna być ponownie wyrenderowana. Do tego służy funkcja *getEmployeeList()*, jednak gdy przekażemy tą metodę klasie *Employee*, funkcja nie będzie już powiązana z obiektem klasy *EmployeeList*. W tym celu do konstruktora musimy dopisać bindowanie metody z obiektem klasy.

```
export default class EmployeeList extends React.Component {
  constructor(props) {
    super(props);
    this.getEmployeeList = this.getEmployeeList.bind(this);
    this.state = {
      employees: []
    };
  }
```

Po tej zmianie, możemy przekazać funkcję jako parametr podczas utworzenia obiektów klasy *Employee*.

```
const employees = this.state.employees.map(employee => <Employee key={employee.pesel} employee={employee}
getEmployeeList={this.getEmployeeList}/>);
```

Następnie możemy przejść do *Employee*. Na samym początku dodamy import dla klasy Button.

```
import Button from 'react-bootstrap/Button';
```

W konstruktorze utworzymy flagę, która, podczas usuwania, zablokuje przycisk oraz od razu zbindujemy funkcję usuwającą użytkownika, ponieważ będzie ona przekazana do obiektu *Button*.

```
export default class Employee extends React.Component {
  constructor(props) {
    super(props);
    this.deleteEmployee = this.deleteEmployee.bind(this);
    this.state = {
      isDeleting: false,
    };
  }
```

Metoda *render* będzie musiała dodać nową kolumnę z przyciskiem Usuń.

```
render(){
    const { isDeleting } = this.state;
    const { employee } = this.props
    const latestPersonalDataIndex = employee.daneOsobowe.length - 1;

    return (
      <tr>
        <td>{employee.daneOsobowe[latestPersonalDataIndex].imie}</td>
        <td>{employee.daneOsobowe[latestPersonalDataIndex].nazwisko}</td>
        <td>{employee.pesel}</td>
        <td><Button pesel={employee.pesel} variant="danger" disabled={isDeleting}
onClick={!isDeleting?this.deleteEmployee:null}>{isDeleting?'Usuwanie...':'Usuń'}</Button></td>
      </tr>
    );
  }
```

Stan zmiennej *isDeleting* został przepisany do stałej. Następnie w czwartej kolumnie utworzony zostaje obiekt klasy *Button*, który przechowuje informację o peselu pracownika. Aplikacja napisana we framework'u Spring do usunięcia pracownika potrzebuje jedynie peselu. Następnie, przy pomocy argumentu *variant* ustawiany jest kolor przycisku, a jego aktywność zależy od wartości *isDeleting*. Przy pomocy *onClick*, ustawiana jest metoda, która zostanie wywołana po naciśnięciu danego przycisku. Dla pewności, podczas usuwania, metoda ta zostaje odłączona. Na koniec, w zależności od stanu usuwania, ustawiany jest odpowiedni tekst wewnątrz przycisku.

Ostatnią metodą, którą trzeba zaimplementować, jest metoda *deleteEmployee*. Nasłuchuje ona kliknięcia przycisku, dlatego jako parametr otrzymuje ona event.

```
deleteEmployee(event){

}
```

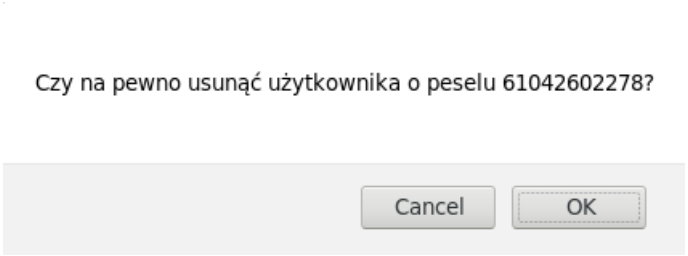
Na początku odczytamy, który przycisk został naciśnięty oraz pesel pracownika, do którego ten przycisk był przypisany.

```
const target = event.target;
const pesel = target.getAttribute('pesel');
```

Następnie powinien zostać wyświetlony komunikat, proszący o potwierdzenie usunięcia.

```
if(!window.confirm(`Czy na pewno usunąć użytkownika o peselu ${pesel}?`)){
    return;
}
```

Używają tzw. backtick (ten sam klawisz co tylda), możemy dodać do tekstu wcześniej odczytany pesel, poprzez użycie składni ${zmienna}. Funkcja *confirm* generuje i wyświetla okno z podanym komunikatem oraz dwoma przyciskami.



Jeśli użytkownik naciśnie OK, to funkcja zwraca wartość *true*, a przycisk Cancel – *false*. Właśnie z tego powodu przed całym wyrażeniem jest dodana negacja.

Dalsza część metody wykona się jedynie, gdy użytkownik potwierdził swój wybór. Oznacza to, że możemy utworzyć zapytanie oraz uniemożliwić dalsze naciskanie przycisku.

```
const url = `/api/pracownicy/pracownicy/${pesel}/delete`
this.setState({isDeleting:true});
```

Na koniec można wysłać zapytanie do serwera. Tym razem konieczne jest ustawienie metody jako DELETE. Drugi parametr funkcji *fetch* pozwala na określenie opcji i powinien być podawany w klamerkach, jak widać niżej.

```
fetch(url,{
      method: 'delete'
    }).then(response => {
      if(response.ok){
        this.props.getEmployeeList();
        //odkomentuj jeśli chcesz przetestować blokowanie przycisku podczas usuwania
        /*this.sleep(1000).then(()=>{
          this.props.getEmployeeList();
        })*/
      } else {
        throw new Error("Usuwanie pracownika nie powiodło się, spróbuj później.")
      }
    }).catch(error => {
        this.setState({isDeleting: false});
        window.alert(error.message);
      });
```

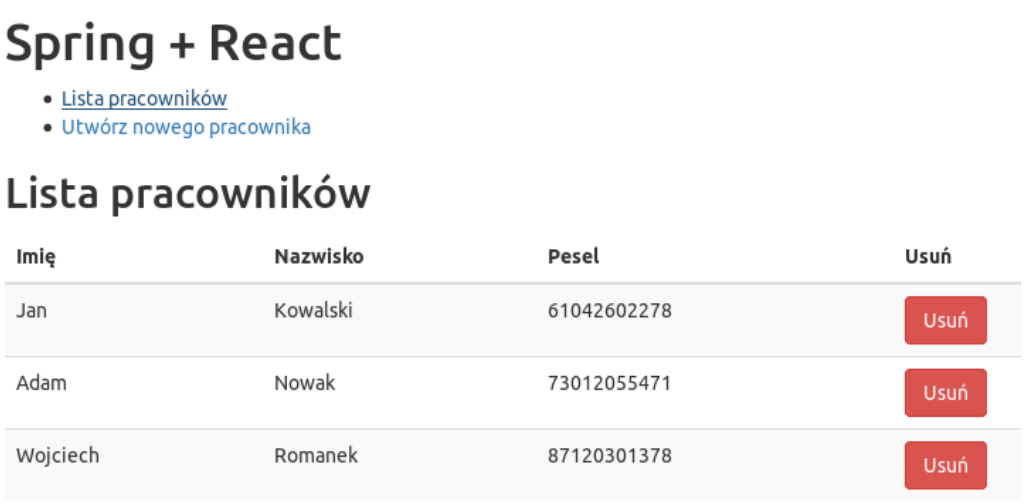
Jeśli operacja wykonała się poprawnie, wywołujemy metodę *getEmployeeList()*, podaną jako parametr podczas tworzenia obiektu, przez co lista z pracownikami zostanie odczytana na nowo. Jeśli podczas operacji wystąpił błąd, wyświetlony zostaje odpowiedni komunikat oraz flaga, informująca o procesie usuwania, zostaje ustawiona na *false*.

Jeśli wyłączanie przycisku wykonuje się zbyt szybko, możesz zakomentować wywołanie *getEmployeeList()* i odkomentować metodę, która odświeży listę pracowników dopiero po 1000 milisekundach. Nie jest to wbudowana funkcja dlatego musiałbyś też dopisać w tej klasie:

```
sleep(time) {
  return new Promise((resolve) => setTimeout(resolve, time));
}
```

Obecna wersja aplikacji nie pozwala na dodawanie nowych pracowników do bazy danych. W celu przetestowania usuwania pracowników, konieczne jest wcześniejsze dodanie pracownika przy pomocy Postman'a w ten sam sposób, który został omówiony we wcześniejszych instrukcjach (str. 13).

Po tych zmianach aplikacja powinna wyglądać następująco:



7. Utworzenie nowego pracownika

Ostatnim elementem tej aplikacji będzie formularz umożliwiający dodawanie nowych pracowników. Otwórz plik *CreateEmployee.js*, zmodyfikuj metodę *render* tak, by korzystała z klas *col-md-10* i *col-md-offset-1*, które są dostępne w bibliotece Bootstrap. Następnie dodaj obiekt klasy *CreateForm*.

```
import React from 'react';

import CreateForm from './CreateForm';

export default class CreateEmployee extends React.Component  {

  render() {
    return (
      <div id="create" className="col-md-10 col-md-offset-1">
        <h2>Utwórz nowego pracownika</h2>
        <CreateForm/>
      </div>
    );
  }
}
```

Wewnątrz klasy *CreateForm* generowany i obsługiwany będzie formularz. Utwórz plik *CreateForm*.js i wewnątrz – klasę *CreateForm*.

```
import React from 'react';
import Form from 'react-bootstrap/Form';
import Button from 'react-bootstrap/Button';
import {
  Redirect
} from 'react-router-dom';

export default class CreateForm extends React.Component {
```

Form i *Button* zostaną wykorzystane podczas renderowania formularza, *Redirect* przeniesie użytkownika do listy pracowników, po tym, jak utworzy nowego pracownika.

W konstruktorze zbindujemy dwie metody, które zostaną w dalszej części opisane: *handleSubmit* i *onChange*. Ponadto ustawimy flagę *createdEmployee* na *false*, która po ustawieniu na *true*, przeniesie użytkownika do listy pracowników. Na koniec przygotujemy pole *formData*, które będzie przechowywać dane z formularza.

```
constructor(props){
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
    this.onChange = this.onChange.bind(this);
    this.state = {
      createdEmployee: false,
      formData: {
        "daneOsobowe":[{}]
      }
    };
```

Aby obiekt JSON *formData* wyglądał tak samo jak ciało zapytania, które w aplikacji backend'owej służy do dodania nowego pracownika, od początku przypisana jest do niego lista *daneOsobowe*, która na początku zawiera pusty obiekt. **Ostatnia klamerka widocznego kodu zamyka this.state, a nie konstruktor.**

Następnie zadeklarowane zostaną kontrolki. Formularz będzie statyczny, dlatego wszystkie pola można zadeklarować w konstruktorze.

```
const inputList = [
      { daneOsobowe: true, name: "imie", label: "Imię", type: "text"},
      { daneOsobowe: true, name: "nazwisko", label: "Nazwisko", type: "text"},
      { daneOsobowe: true, name: "mail", label: "E-mail", type: "email" },
      { daneOsobowe: true, name: "telefon", label: "Telefon", type: "text" },
      { daneOsobowe: false, name: "dataUrodzenia", label: "Data urodzenia", type: "date" },
      { daneOsobowe: false, name: "miejsceUrodzenia", label: "Miejsce urodzenia", type: "text" },
      { daneOsobowe: true, name: "miejscowosc", label: "Miejscowość", type: "text" },
      { daneOsobowe: true, name: "ulica", label: "Ulica", type: "text" },
      { daneOsobowe: true, name: "nrDomu", label: "Nr domu", type: "text" },
      { daneOsobowe: true, name: "nrLokalu", label: "Nr lokalu", type: "number" },
      { daneOsobowe: false, name: "nip", label: "NIP", type: "text" },
      { daneOsobowe: false, name: "pesel", label: "Pesel", type: "text" },
      { daneOsobowe: true, name: "nrDowoduOsobistego", label: "Nr dowodu osobistego", type: "text" },
      { daneOsobowe: true, name: "odKiedy", label: "Od kiedy", type: "date" },
      { daneOsobowe: true, name: "kod", label: "Kod", type: "number" }

    ]
```

Każdy obiekt listy odpowiada jednej kontrolce. Z powodu nietypowej konstrukcji *body* dla zapytania, prócz nazwy, wyświetlanej nazwy oraz typu, dodane jest też pole *daneOsobowe*. Wykorzystanie tej flagi będzie zrozumiałe po analizie kodu metody *onChange()*.

Na końcu konstruktora, podobnie jak w przypadku listy pracowników, na stałej *inputList* wywołana zostanie metoda *map*, która dla każdego elementu listy, wygeneruje odpowiedni kod JSX.

```
this.inputControls = inputList.map(
      ({daneOsobowe, name, label, type}, index) =>(
        <Form.Group key={index}>
          <Form.Label>{label}</Form.Label>
          <Form.Control className={daneOsobowe?"daneOsobowe":""} name={name} type={type}
placeholder={label} onChange={this.onChange} required/>
        </Form.Group>
      )
    );
```

Każdy obiekt ma cztery pola, dlatego na początku są one odczytywane. Ponadto przy każdej iteracji zwracany jest indeks, który zostanie użyty jako unikalny klucz każdego wygenerowanego elementu. W zależności od wartości pola *daneOsobowe*, do elementu dopisywana (lub nie) zostaje klasa o tej samej nazwie. Ponadto każde pole jest wymagane i przy zmianie zostaje wykorzystana metoda *onChange()*.

```
onChange(event){
    const target = event.target;
    const name = target.name;
    const value = target.value;

    if(target.classList.contains("daneOsobowe")){
      let {daneOsobowe} = this.state.formData;
      daneOsobowe[0][name] = value;
      this.setState({daneOsobowe});
    } else {
      let {formData} = this.state
      formData[name] = value;
      this.setState({formData});
    }
    //console.log(this.state.formData);
  }
```

Podobnie jak w *onClick*, *onChange* otrzymuje event przy zmianie wartości każdej z kontrolek. Gdy dana kontrolka opisuje daną, która powinna znaleźć się wewnątrz listy *daneOsobowe*, wczytywana jest obecna wartość ten list, następnie dopisywana jest obecna wartość kontrolki i na koniec aktualizowana jest wartość listy w *this.state*. Gdy cel obsługiwanego event'u nie miał tej klasy, wartość jest dopisywana do *formData*. Aby lepiej zrozumieć działanie tej metody, odkomentuj logowanie, włącz konsolę deweloperską (F12 w przeglądarce) i sprawdź jak się zmieniają wartości *formData* po wpisaniu wartości do różnych inputów.

Przed wysłaniem komunikatu, zaktualizujemy metodę *render* w następujący sposób.

```
render() {
  const { createdEmployee } = this.state;

  if(createdEmployee){
    return <Redirect to='/'/>
  }

  return (
    <Form onSubmit={this.handleSubmit}>
      {this.inputControls}
      <Button as="input" type="submit" value="Zapisz"/>
    </Form>
  );
}
```

Na początku jest sprawdzane, czy użytkownik został już utworzony. Jeśli tak, użytkownik zostaje przekierowany na główną stronę aplikacji, czyli do listy pracowników. W przeciwnym wypadku renderowany zostaje formularz z obsługą event'u *onSubmit()*. Wewnątrz formularza wyświetlone zostają wcześniej zadeklarowane kontrolki oraz przycisk *submit*.

Na koniec konieczna jest implementacja metody *onSubmit()*, która wyśle dane nowego pracownika do serwera i obsłuży jego odpowiedź.

Aby zapobiec odświeżeniu strony, należy użyć metody *event.preventDefault()*. Następnie wysyłane jest zapytanie metodą POST, a jako *body* konwertujemy obiekt JSON do stringa.

```
handleSubmit(event){
  event.preventDefault();

  fetch("/api/pracownicy/pracownicy/create",{
    method: 'POST',
    headers: {
      'Accept': 'application/json',
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(this.state.formData)
  }).then(response => {
    if(response.ok){
      return response.json();
    } else {
      throw new Error("Utworzenie nowego pracownika nie powiodło się, spróbuj później.")
    }
  }).then(data => {
      //console.log(data);
      this.setState({createdEmployee: true});
  }).catch(error => {
      window.alert(error.message);
  });
}
```

Jeśli udało się utworzyć pracownika, flaga *createdEmployee* jest ustawiana na *true*. W odpowiedzi otrzymujemy też wszystkie dane o utworzonym użytkowniku wraz z jego identyfikatorem. Można to sprawdzić odkomentowując logowanie. Jeśli operacja się nie powiodła, wyświetlany jest odpowiedni komunikat.
Działająca aplikacja powinna wyglądać następująco: