Frontend development

Wykład 7

Frameworki i Biblioteki Frontendowe – React, Angular, Vue.js

Mateusz Pawełkiewicz

1. Przegląd popularnych frameworków i bibliotek

Współczesny rozwój aplikacji webowych wymaga użycia narzędzi, które ułatwiają tworzenie, zarządzanie i skalowanie kodu. Frameworki i biblioteki frontendowe dostarczają struktury oraz funkcjonalności, które przyspieszają proces tworzenia interaktywnych interfejsów użytkownika.

1.1 React

React to biblioteka JavaScript stworzona przez Facebooka w 2013 roku. Skupia się na budowie interfejsów użytkownika i wprowadza koncepcję komponentów, które są samowystarczalnymi blokami UI. React wykorzystuje wirtualny DOM, co zwiększa wydajność aplikacji przez minimalizację bezpośrednich manipulacji na rzeczywistym DOM.

- **Zalety**:
  - Wysoka wydajność dzięki wirtualnemu DOM.
  - Duża społeczność i wsparcie.
  - Elastyczność i możliwość integracji z innymi bibliotekami.
- **Wady**:
  - Szybki rozwój może utrudniać utrzymanie aktualnej wiedzy.
  - Skupia się tylko na warstwie widoku, wymaga dodatkowych bibliotek do pełnej funkcjonalności.

1.2 Angular

Angular to kompleksowy framework JavaScript opracowany przez Google. Pierwsza wersja, AngularJS, została wydana w 2010 roku. W 2016 roku zaprezentowano Angular 2, który jest całkowicie przebudowaną wersją frameworka i opiera się na TypeScript. Angular oferuje pełen zestaw narzędzi do tworzenia aplikacji, w tym moduły, komponenty, usługi, routowanie i wiele innych.

- **Zalety**:
  - Kompleksowość i bogactwo funkcji.
  - Wsparcie dla TypeScriptu, co zwiększa czytelność i bezpieczeństwo kodu.
  - Silna struktura aplikacji, ułatwiająca skalowanie.
- **Wady**:
  - Wyższa krzywa uczenia się ze względu na złożoność.
  - Większy narzut kodu i mniejsza elastyczność w porównaniu z innymi rozwiązaniami.

1.3 Vue.js

Vue.js to progresywny framework JavaScript stworzony przez Evana You w 2014 roku. Łączy zalety Reacta i Angulara, oferując prostotę i wydajność. Vue jest elastyczny i może być stopniowo wprowadzany do projektów, co ułatwia integrację z istniejącym kodem.

- **Zalety**:
  - Łatwa krzywa uczenia się.
  - Lekkość i wydajność.

    - Prosta integracja z istniejącymi projektami.
- **Wady**:
    - Mniejsza społeczność w porównaniu z Reactem i Angularem.
    - Mniejsze wsparcie korporacyjne.

---

2. Wprowadzenie do React

2.1 Komponenty

Komponenty to podstawowe bloki budujące aplikacje React. Każdy komponent reprezentuje fragment interfejsu użytkownika i może być ponownie wykorzystywany w różnych miejscach aplikacji.

- **Komponenty funkcyjne**: Zdefiniowane jako funkcje JavaScript, które przyjmują właściwości (props) i zwracają elementy React.

```
function Przywitanie(props) {
  return <h1>Witaj, {props.imie}!</h1>;
}
```

- **Komponenty klasowe**: Zdefiniowane jako klasy ES6 dziedziczące po React.Component. Umożliwiają korzystanie ze stanu (state) i metod cyklu życia.

```
class Przywitanie extends React.Component {
  render() {
    return <h1>Witaj, {this.props.imie}!</h1>;
  }
}
```

2.2 Stan i właściwości

- **Props (Właściwości)**: Dane przekazywane do komponentu z jego rodzica. Są one niezmienne w trakcie życia komponentu.

```
<Przywitanie imie="Jan" />
```

- **State (Stan)**: Dane wewnętrzne komponentu, które mogą ulegać zmianie i wpływać na renderowanie.
    - W komponentach klasowych stan jest inicjowany w konstruktorze:

```
class Zegar extends React.Component {
  constructor(props) {
    super(props);
    this.state = { czas: new Date() };
  }
}
```

- W komponentach funkcyjnych używamy hooka useState:

```
function Zegar() {
  const [czas, setCzas] = React.useState(new Date());
}
```

2.3 JSX

JSX to rozszerzenie składni JavaScript, które umożliwia pisanie elementów przypominających HTML w kodzie JavaScript. JSX ułatwia tworzenie interfejsów użytkownika poprzez intuicyjną składnię.

- **Przykład JSX**:

```
const element = <h1>Witaj, świecie!</h1>;
```

- **Kompilacja**: JSX nie jest zrozumiały dla przeglądarek i musi być przetworzony (np. przez Babel) do standardowego JavaScriptu.
- **Wstawianie wyrażeń JavaScript**: Wewnątrz JSX możemy używać wyrażeń JavaScript w nawiasach klamrowych { }.

```
const imie = 'Anna';
const element = <h1>Witaj, {imie}!</h1>;
```

---

3. Zarządzanie stanem aplikacji

W miarę rozwoju aplikacji React, zarządzanie stanem staje się coraz bardziej skomplikowane. Aby ułatwić ten proces, stosuje się różne narzędzia i wzorce.

3.1 Redux

Redux to biblioteka JavaScript do zarządzania stanem aplikacji. Wprowadza jednokierunkowy przepływ danych i przechowuje cały stan aplikacji w jednym globalnym obiekcie (store).

- **Główne koncepcje**:
  - **Store**: Przechowuje stan aplikacji.
  - **Actions**: Obiekty opisujące, co się stało.
  - **Reducers**: Funkcje, które określają, jak stan aplikacji zmienia się w odpowiedzi na akcje.
- **Przykład akcji**:

```
const dodajTodo = (tekst) => {
  return {
    type: 'DODAJ_TODO',
    tekst
  };
};
```

- **Przykład reduktora**:

```
const todos = (state = [], action) => {
 switch (action.type) {
  case 'DODAJ_TODO':
   return [...state, { tekst: action.tekst, ukończone: false }];
  default:
   return state;
 }
};
```

- **Zalety**:
  - Przewidywalność stanu.
  - Ułatwia debugowanie i testowanie.
- **Wady**:
  - Więcej kodu do napisania.
  - Może być zbyt skomplikowany dla prostych aplikacji.

3.2 Context API

Context API to wbudowany mechanizm w React, który pozwala na przekazywanie danych przez drzewo komponentów bez konieczności przekazywania propsów na każdym poziomie.

- **Tworzenie kontekstu**:

```
const TematKontekst = React.createContext('jasny');
```

- **Provider**: Dostarcza wartość kontekstu dla komponentów potomnych.

```
<TematKontekst.Provider value="ciemny">
 <Aplikacja />
</TematKontekst.Provider>
```

- **Consumer**: Umożliwia komponentom dostęp do wartości kontekstu.

```
function Przycisk() {
 const temat = React.useContext(TematKontekst);
 return <button className={temat}>Kliknij mnie</button>;
}
```

- **Zalety**:
  - Prostota i łatwość użycia.
  - Unikanie problemu "prop drilling" (przekazywanie propsów przez wiele poziomów komponentów).
- **Wady**:
  - Mniej wydajny przy częstych zmianach stanu w porównaniu z Reduxem.

4. Routing w aplikacjach frontendowych

Routing pozwala na nawigację między różnymi widokami w aplikacji jednopodstronowej (SPA) bez przeładowywania strony.

4.1 React Router

React Router to biblioteka służąca do obsługi routingu w aplikacjach React.

- **Instalacja**:

  ```
  npm install react-router-dom
  ```

- **Podstawowe komponenty**:
  - **BrowserRouter**: Komponent najwyższego poziomu, który używa API HTML5 do nawigacji.
  - **Route**: Definiuje ścieżkę i odpowiadający jej komponent.
  - **Link**: Umożliwia nawigację między różnymi ścieżkami bez przeładowania strony.
- **Przykład użycia**:

  ```
  import { BrowserRouter, Route, Link } from 'react-router-dom';

  function App() {
    return (
      <BrowserRouter>
        <nav>
          <Link to="/home">Strona główna</Link>
          <Link to="/o-nas">O nas</Link>
        </nav>
        <Route path="/home" component={Home} />
        <Route path="/o-nas" component={About} />
      </BrowserRouter>
    );
  }
  ```

4.2 Routing w Angular

Angular ma wbudowany moduł do obsługi routingu.

- **Konfiguracja routingu**:

  ```
  import { RouterModule, Routes } from '@angular/router';

  const routes: Routes = [
    { path: 'home', component: HomeComponent },
    { path: 'o-nas', component: AboutComponent },
    { path: '', redirectTo: '/home', pathMatch: 'full' },
  ];

  @NgModule({
    imports: [RouterModule.forRoot(routes)],
    exports: [RouterModule],
  ```

```
})
export class AppRoutingModule {}
```

- **Router Outlet**: Miejsce, w którym wyświetlane są komponenty w zależności od ścieżki.

```
<router-outlet></router-outlet>
```

- **Linki nawigacyjne**:

```
<a routerLink="/home">Strona główna</a>
```

4.3 Routing w Vue.js

Vue Router to oficjalny router dla Vue.js.

- **Instalacja**:

```
npm install vue-router
```

- **Konfiguracja routingu**:

```
import Vue from 'vue';
import VueRouter from 'vue-router';
import Home from './components/Home.vue';
import About from './components/About.vue';

Vue.use(VueRouter);

const routes = [
  { path: '/home', component: Home },
  { path: '/o-nas', component: About },
];

const router = new VueRouter({
  routes,
});

new Vue({
  router,
  render: (h) => h(App),
}).$mount('#app');
```

- **Linki nawigacyjne**:

```
<router-link to="/home">Strona główna</router-link>
```

- **Router View**: Komponent wyświetlający aktywną trasę.

```
<router-view></router-view>
```

5. Obsługa Guardów w Aplikacjach Frontendowych

Guards (strażnicy) to mechanizmy służące do kontrolowania dostępu do określonych ścieżek w aplikacji, np. w celu zabezpieczenia stron przed nieautoryzowanym dostępem.

5.1 Guards w Angular

Angular dostarcza wbudowane mechanizmy guardów.

- **Typy guardów**:
  - **CanActivate**: Sprawdza, czy można wejść na daną trasę.
  - **CanDeactivate**: Sprawdza, czy można opuścić daną trasę.
  - **CanActivateChild**, **Resolve**, **CanLoad**.
- **Tworzenie guarda**:

```
import { Injectable } from '@angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';

@Injectable({
  providedIn: 'root',
})
export class AuthGuard implements CanActivate {
  canActivate(next: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
    // Logika autoryzacji
    return true; // lub false
  }
}
```

- **Konfiguracja w routingu**:

```
const routes: Routes = [
  {
    path: 'panel',
    component: PanelComponent,
    canActivate: [AuthGuard],
  },
];
```

5.2 Implementacja guardów w React

React Router nie ma wbudowanych guardów, ale można je zaimplementować samodzielnie.

- **Przykład komponentu chronionego**:

```
function ProtectedRoute({ component: Component, ...rest }) {
  return (
    <Route
      {...rest}
      render={(props) =>
        isAuthenticated() ? <Component {...props} /> : <Redirect to="/login" />
```

```
      }
    />
  );
}
```

- **Użycie**:

```
<ProtectedRoute path="/panel" component={Panel} />
```

5.3 Guards w Vue.js (Vue Router)

Vue Router oferuje **navigation guards**.

- **Globalne guardy**:

```
router.beforeEach((to, from, next) => {
  if (to.meta.requiresAuth && !isAuthenticated()) {
    next('/login');
  } else {
    next();
  }
});
```

- **Guardy na trasie**:

```
const routes = [
  {
    path: '/panel',
    component: Panel,
    meta: { requiresAuth: true },
  },
];
```

- **Guardy w komponencie**:

```
export default {
  beforeRouteEnter(to, from, next) {
    // Logika przed wejściem na trasę
    next();
  },
};
```

---

6. Komunikacja z API

Aplikacje webowe często muszą komunikować się z serwerem za pomocą API. Istnieją różne biblioteki i metody ułatwiające tę komunikację.

6.1 Axios

Axios to popularna biblioteka do wykonywania zapytań HTTP.

6.1.1 Wprowadzenie do Axiosa

- Obsługuje promisy.
- Umożliwia wykonywanie zapytań GET, POST, PUT, DELETE i innych.
- Wspiera interceptery do obsługi żądań i odpowiedzi.

6.1.2 Konfiguracja i podstawowe użycie

- **Instalacja**:

```
npm install axios
```

- **Wykonanie zapytania GET**:

```
axios.get('/api/uzytkownicy')
  .then(response => {
    console.log(response.data);
  })
  .catch(error => {
    console.error(error);
  });
```

- **Wykonanie zapytania POST**:

```
axios.post('/api/uzytkownicy', {
  imie: 'Jan',
  nazwisko: 'Kowalski'
})
  .then(response => {
    console.log(response.data);
  })
  .catch(error => {
    console.error(error);
  });
```

6.1.3 Interceptery w Axiosie

Interceptery pozwalają na przechwytywanie i modyfikowanie żądań lub odpowiedzi.

- **Dodawanie tokena autoryzacji do każdego żądania**:

```
axios.interceptors.request.use(config => {
  const token = localStorage.getItem('token');
  if (token) {
    config.headers['Authorization'] = `Bearer ${token}`;
  }
  return config;
}, error => {
```

```
  return Promise.reject(error);
});
```

- **Obsługa błędów globalnie**:

```
axios.interceptors.response.use(response => {
  return response;
}, error => {
  if (error.response.status === 401) {
    // Wyloguj użytkownika
  }
  return Promise.reject(error);
});
```

6.2 Fetch API

Fetch API to wbudowane w przeglądarki API do wykonywania zapytań HTTP.

- **Podstawowe użycie**:

```
fetch('/api/uzytkownicy')
  .then(response => response.json())
  .then(data => {
    console.log(data);
  })
  .catch(error => {
    console.error(error);
  });
```

- **Zapytanie POST**:

```
fetch('/api/uzytkownicy', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({ imie: 'Jan', nazwisko: 'Kowalski' })
})
  .then(response => response.json())
  .then(data => {
    console.log(data);
  });
```

- **Obsługa błędów**:

```
fetch('/api/uzytkownicy')
  .then(response => {
    if (!response.ok) {
      throw new Error('Błąd sieciowy');
    }
    return response.json();
  })
  .then(data => {
    console.log(data);
```

```
})
.catch(error => {
  console.error(error);
});
```

---

7. Narzędzia wspomagające rozwój

Narzędzia takie jak Webpack i Babel są kluczowe w nowoczesnym ekosystemie JavaScript, ułatwiając zarządzanie kodem i kompatybilność z różnymi przeglądarkami.

7.1 Webpack

Webpack to modułowy bundler dla JavaScriptu i zasobów statycznych.

7.1.1 Funkcje Webpacka

- **Łączenie modułów**: Łączy wiele plików JavaScript w jeden lub kilka pakietów.
- **Obsługa zasobów**: Pozwala na importowanie plików CSS, obrazów i innych zasobów.
- **Hot Module Replacement**: Umożliwia aktualizację modułów w czasie rzeczywistym bez przeładowania strony.

7.1.2 Konfiguracja Webpacka

- **Plik konfiguracyjny webpack.config.js**:

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
  module: {
    rules: [
      { test: /\.js$/, exclude: /node_modules/, use: 'babel-loader' },
      { test: /\.css$/, use: ['style-loader', 'css-loader'] },
    ],
  },
};
```

- **Uruchomienie Webpacka**:

```
npx webpack --config webpack.config.js
```

7.2 Babel

Babel to transpilator JavaScript, który pozwala na korzystanie z nowoczesnych funkcji języka w przeglądarkach, które ich jeszcze nie obsługują.

7.2.1 Funkcje Babela

- **Transpilacja**: Przekształca kod ES6+ do kodu kompatybilnego z ES5.
- **Pluginy**: Pozwalają na dodawanie specyficznych funkcjonalności (np. transformacja JSX).

7.2.2 Konfiguracja Babela

- **Instalacja**:

  ```
  npm install @babel/core @babel/cli @babel/preset-env --save-dev
  ```

- **Plik konfiguracyjny .babelrc**:

  ```
  {
    "presets": ["@babel/preset-env"]
  }
  ```

- **Użycie z Webpackiem**:

  W pliku webpack.config.js dodajemy loader babel-loader.

  ```
  module: {
    rules: [
      { test: /\.js$/, exclude: /node_modules/, use: 'babel-loader' },
      // ...
    ],
  },
  ```