Akademia Górniczo-Hutnicza
Wydział Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej
Kierunek Informatyka, III rok, 2018/2019

Wprowadzenie do wzorców projektowych

„Tablica ogłoszeń”

Implementacja mechanizmu server-push z wykorzystaniem techniki streaming w technologii ReactJS oraz NodeJS.

Agnieszka Zadworny, Maciej Bielech, Tomasz Pęcak, Piotr Morawiecki

Kraków, 13 lutego 2019

1 Informacje ogólne

1.1 Cel projektu

Celem projektu jest zaimplementowanie metody Server Push w technice streaming przy użyciu technologii NodeJS i ReactJS.

1.2 Opis aplikacji

Utworzona została aplikacja webowa o nazwie „Tablica ogłoszeń", która umożliwia publikowanie ogłoszeń o wybranej tematyce. Klient otwierając stronę aplikacji otrzymuje listę ogłoszeń sortowaną od najnowszego, jeśli w czasie przebywania na witrynie pojawi się nowe ogłoszenie, serwer wyśle jego treść dzięki wykorzystaniu techniki Server Push, która została zaimplementowana przy użyciu websocketów.

1.3 Server Push

Push technology, czyli server push, to styl komunikacji internetowej, w którym żądanie danej transakcji jest inicjowane przez serwer. W przeciwieństwie do pull/get, gdzie żądanie przekazania informacji jest inicjowane przez odbiorcę lub klienta.

Usługi Push są często oparte na preferencjach informacyjnych wyrażanych z wyprzedzeniem. Nazywa się to modelem publikacji/subskrybcji. Klient śubskrybujeżóżne "kanały"informacyjne dostarczane przez serwer; gdy na jednym z tych kanałów dostępne są nowe treści, serwer "pushuje"te informacje do klienta.

1.4 Websocket

Protokół WebSocket umożliwia interakcję między klientem internetowym (np. przeglądarką), a serwerem WWW przy dokonywaniu mniejszej ilości zapytań, ułatwiając przesyłanie danych w czasie rzeczywistym z i na serwer. Jest to możliwe dzięki zapewnieniu znormalizowanego sposobu, w jaki serwer może wysyłać treści do klienta bez uprzedniego żądania z jego strony i pozwalającego na przekazywanie wiadomości w obu kierunkach przy zachowaniu otwartego połączenia. W ten sposób może mieć miejsce dwukierunkowe połączenie pomiędzy klientem i serwerem.

2 Jak korzystać z biblioteki socket-lib

1. Zainstaluj paczkę socket-lib:

```
npm i @maciekb05/socket-lib
```

2. Zainicjalizuj socket wcześniej utworzonym serwerem
3. Połącz się z bazą danych. Musisz podać URI twojej bazy danych oraz schemat modelu
4. Zdefiniuj callback do funkcji onEvent()

```
const server = http.createServer(app);

lib.initializeSocket(server);

lib.connectDataBase(
     dbString: "postgres://bvdxjgrk:hs9ZiISsjYQZobeeJ-zNHyPfRYIWc5Wy@manny.db.elephantsql.com:5432/bvdxjgrk",
     eventSchema: {
        description: "varchar(255)",
        additiondate: "varchar(255)",
        expirationdate: "varchar(255)",
        email: "varchar(255)",
        username: "varchar(255)",
        location: "varchar(255)",
        phone: "varchar(255)",
        subject: "varchar(255)"
    }
);

lib.onEvent( eventName: "advertisement",  callback: data => {
    lib.notifyEveryone( eventName: "advertisement", data);
});
```

Rysunek 1: Przykład korzystania z biblioteki socket-lib

3 Jak korzystać z biblioteki socket-lib-client

1. Zainstaluj paczkę socket-lib:

   npm i @maciekb05/socket-lib-client

2. Zainicjalizuj socket endpointem wcześniej utworzonego serwera
3. Zdefiniuj callback do funkcji onEvent() dla zdarzeń

```
state = {
    advertisements: [],
    endpoint: "http://127.0.0.1:4001"
};

componentDidMount() {
    socket.initializeSocket(this.state.endpoint);
    socket.onEvent( eventName: "advertisement",  callback: data => this.addAdvertisement(data));
    socket.onEvent( eventName: "history",  callback: (history) => this.addHistoryToState(history));
}
```

Rysunek 2: Przykład korzystania z biblioteki socket-lib-client

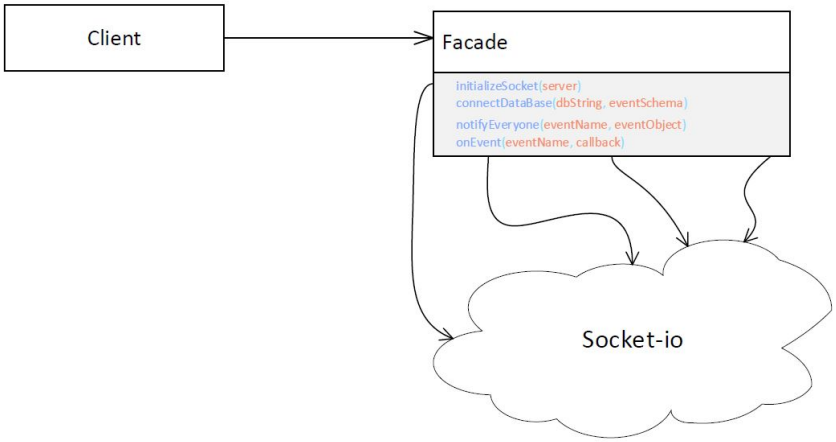
4 Wykorzystane wzorce projektowe

4.1 Fasada

Zastosowaliśmy ją w obu bibliotekach, aby dostarczyć klientom uproszczony interfejs. Ukrywa ona złożoność zarówno naszej implementacji jak i biblioteki socket-io.

```
//Facade
class ServerPush {
    constructor() {
        this.state = new NotInitialized(this);
    }

    changeState(state) {
        this.state = state;
    }

    initializeSocket(server) {
        this.state.initializeSocket(server);
    }

    connectDataBase(dbString, eventSchema) {
        this.state.connectDataBase(dbString, eventSchema);
    }

    notifyEveryone(eventName, eventObject) {
        this.state.notifyEveryone(eventName, eventObject);
    }

    async onEvent(eventName, callback) {
        await this.state.onEvent(eventName, callback);
    }
}
```

Rysunek 3: Implementacja fasady



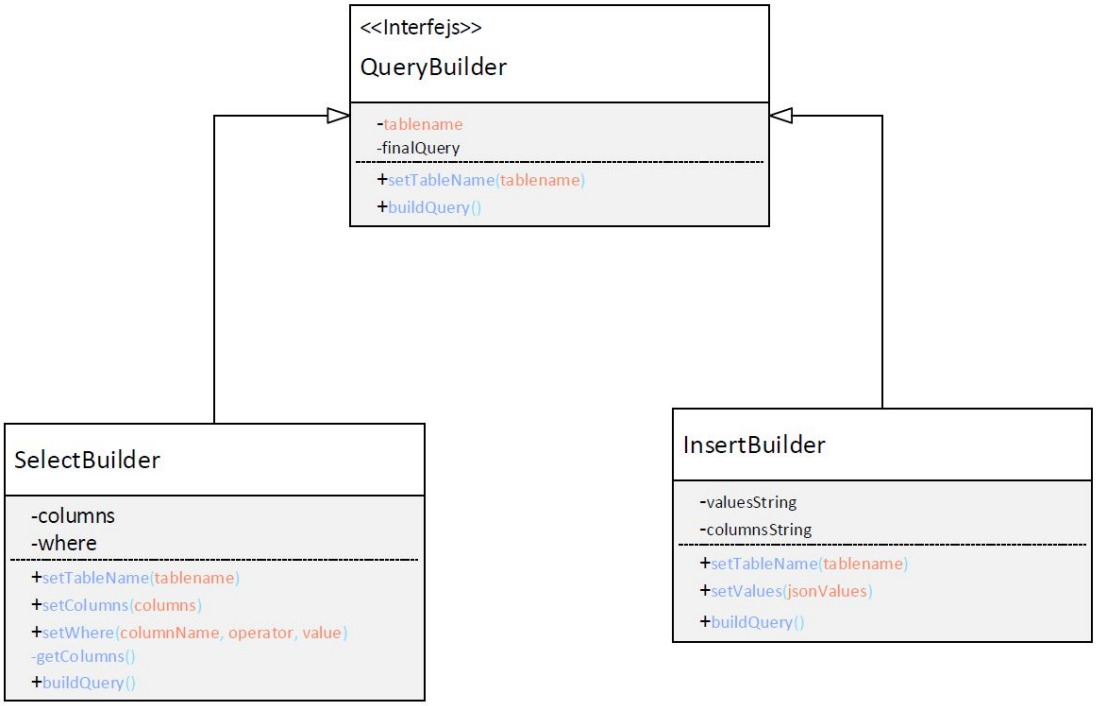
Rysunek 4: Diagram klas fasady

4.2 Query Builder

Obiekt reprezentuje zapytanie kierowane do bazy danych. Generuje zapytanie SQL, bazując na klasach i polach klas. Uzyskaliśmy dzięki niemu niezależność aplikacji od schematu bazy danych. Pozwala na konstrukcje kwerendy krok po kroku, łatwiejsze budowanie dynamicznych zapytań o obiekty których specyfikacji nie znamy.

```
class InsertBuilder extends QueryBuilder {

    constructor(tablename) {
        super(tablename);
        this.valuesString = '';
        this.columnsString = '';
    }

    setTableName(tablename) {
        this.tableName = tablename;
        return this;
    }

    setValues(jsonValues) {
        this.columnsString = ''
        this.valuesString = '';
        for (var key in jsonValues) {
            this.columnsString += key + ', ';
            if (!isNaN(jsonValues[key])) {
                this.valuesString += jsonValues[key] + ', ';
            } else {
                this.valuesString += '\'' + jsonValues[key] + '\', ';
            }
        }
        this.columnsString = this.columnsString.substr( from: 0,  length: this.columnsString.length - 2);
        this.valuesString = this.valuesString.substr( from: 0,  length: this.valuesString.length - 2);
        return this;
    }

    buildQuery() {
        return "INSERT INTO " + this.tableName + " (" + this.columnsString + ") VALUES (" + this.valuesString + ");";
    }
```

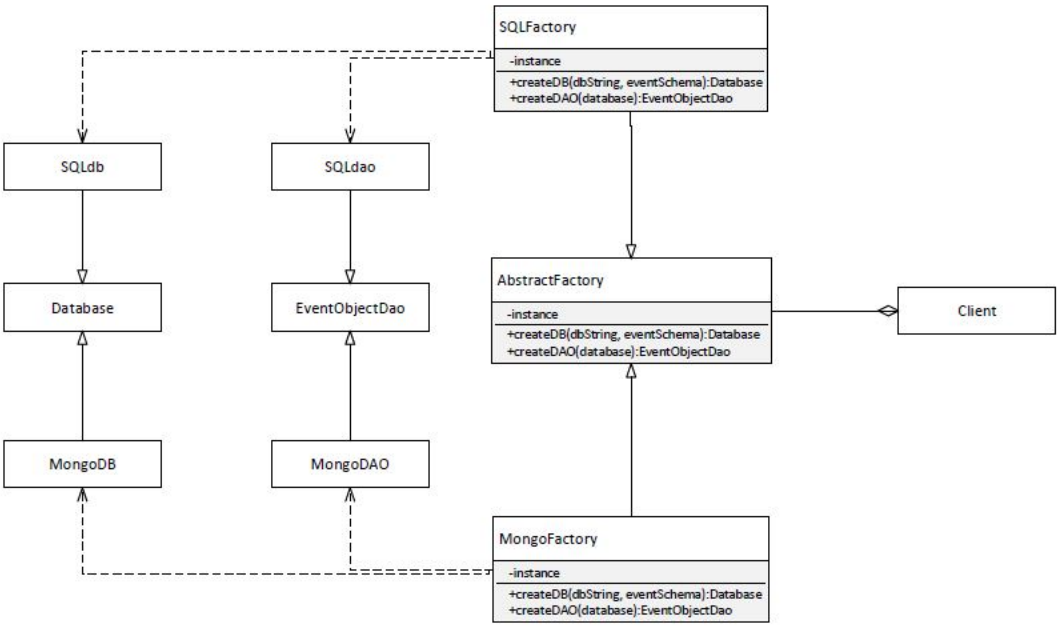
Rysunek 5: Implementacja Query Builder'a



Rysunek 6: Diagram klas Query Builder'a

4.3 Fabryka Abstrakcyjna

Wzorzec wymusza zależności pomiędzy klasami konkretnymi – zależności obiektów w rodzinie co w naszym przypadku jest pożądane, bo otrzymamy obiekty ze sobą kompatybilne. Dodatkowo ułatwi nam to w przyszłości wprowadzanie nowych wariantów baz danych.



Rysunek 7: Diagram klas dla fabryki abstrakcyjnej

4.4 Singleton

Singleton użyliśmy w dwóch klasach. Dla fabryki abstrakcyjnej, żeby nie stworzyć innych fabryk i nie mieć różnych systemów bazodanowowych w aplikacji. Dla klasy bazy danych po to, aby nawiązać tylko jedno połączenie z bazą.

```
let _instance;

class Database {
    constructor(dbString, eventSchema) {
        if (_instance) {
            return _instance;
        } else {
            this.isDbCreated = false;
            _instance = this;
            this.dbString = dbString;
            this.eventSchema = eventSchema;
        }
    }

    Connect() {
        throw new Error( message: "Not implemented");
    }

    RegisterNewSchema() {
        throw new Error( message: "Not implemented");
    }
```

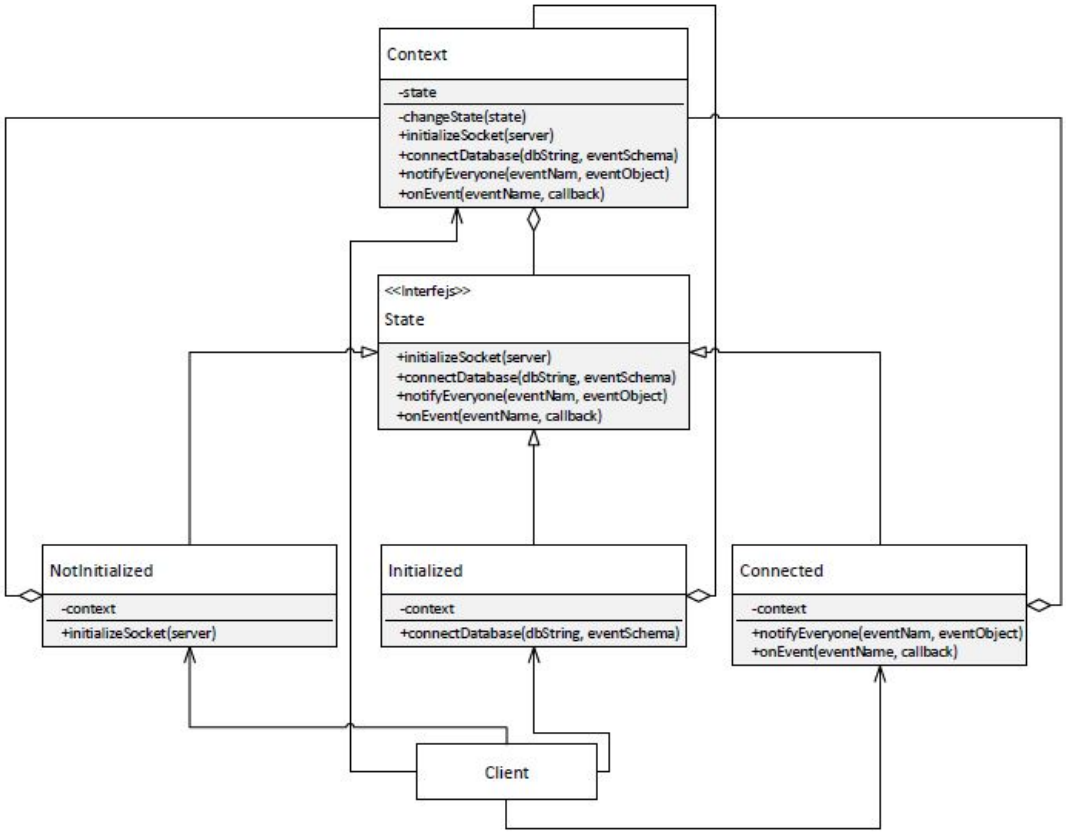
Rysunek 8: Implementacja fabryki abstrakcyjnej jako singletona

4.5 Maszyna Stanowa

Wzorzec state gwarantuje, że jeżeli klient użyje naszych funkcji w nieprawidłowy sposób zostanie o tym poinformowany. Dzięki użyciu tego wzorca nie trzeba korzystać z wielu instrukcji warunkowych, kod jest bardziej czytelny, a te same metody w zależności od tego w jakim stanie jest moduł działają w inny sposób. Zabezpiecza on w naszym przypadku kolejność wykonywania działań, czyli gdy nasza aplikacja jest Niezainicjowana, nie pozwala, aby ktoś uruchomił funkcję korzystającą z socket'a, bo jeszcze nie jest on zainicjalizowany, ani podłączony do bazy danych.

```
class NotInitialized extends State {
    constructor(context) {
        super(context);
        this.context.error = "Not implemented, you can only initializeSocket(server) it this state";
    }

    initializeSocket(server) {
        this.context.io = socketIo(server);
        this.context._changeState(new Initialized(this.context));
    }
}

class Initialized extends State {
    constructor(context) {
        super(context);
        this.context.error = "Not implemented, you can only connectDataBase(dbString, eventSchema) it this state";
    }

    connectDataBase(dbString, eventSchema) {
        if (isMongoString(dbString)) {
            this.context.factory = new MongoFactory();
        } else if (isPostgresString(dbString)) {
            this.context.factory = new SQLFactory();
        } else {
            throw new Error( message: "This database is not supported");
        }

        this.context.dataBase = this.context.factory.createDB(dbString, eventSchema);
        this.context.dataBase.Connect();

        this.context.dao = this.context.factory.createDAO(this.context.dataBase);

        this.context._changeState(new Connected(this.context));
    }
```

Rysunek 9: Implementacja maszyny stanowej



Rysunek 10: Diagram klas maszyny stanowej

4.6 Dao Factory

Jest odpowiedzialne za dostep do danych w bazie danych. Obsługuje operacje CRUD dla danego obiektu. Separuje logikę dostępu do bazy danych.

```
const mongoose = require('mongoose');
const Events = mongoose.model( name: 'events');
const EventObjectDao = require('../EventObjectDao');

class MongoDao extends EventObjectDao {
    constructor(database) {
        super(database);
    }

    async FindEvents() {
        try {
            let events = await Events.find({});
            return events;
        } catch (e) {
            console.log(e);
        }
    }

    async AddEvent(_event) {
        try {
            let event = new Events(_event);
            await event.save();
        } catch (e) {
            console.log(e);
        }
    }
```

Rysunek 11: Implementacja DAO Factory