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

3 Jak korzystać z biblioteki socket-lib-client

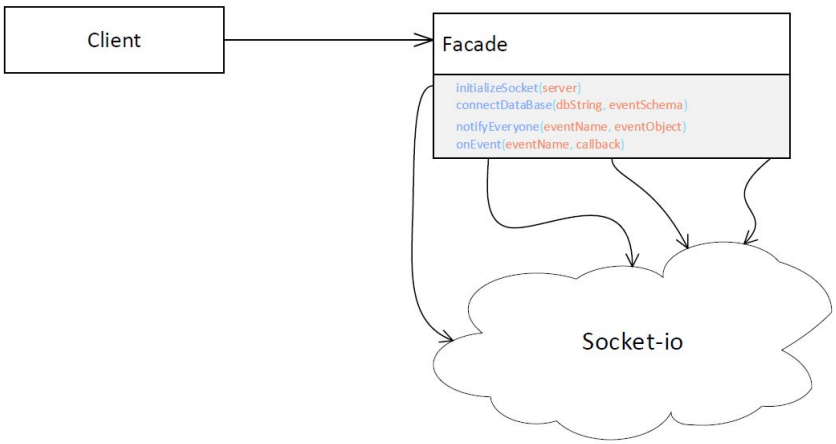
4 Wykorzystane wzorce projektowe

4.1 Fasada

Korzystamy z niej aby dostarczyć klientowi uproszczony interfejs 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 1: Implementacja fasady



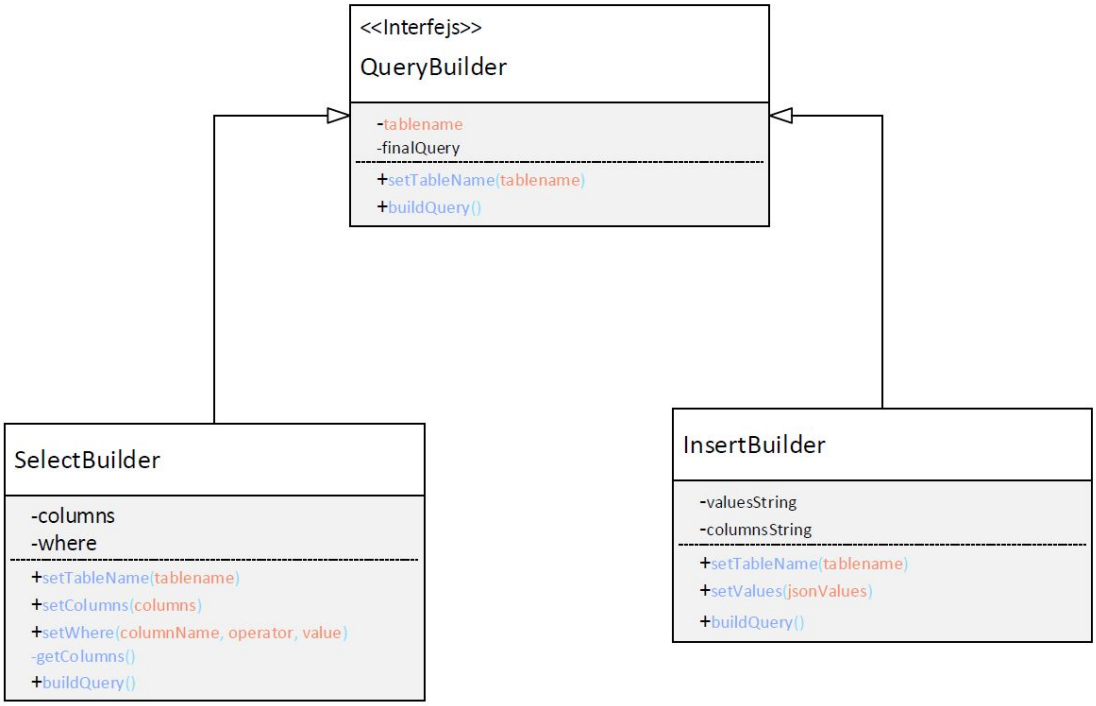
Rysunek 2: Diagram klas fasady

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. Dodatkowo pozwala on na korzystanie z różnych motorów baz danych.

```
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 + ");";
    }
```

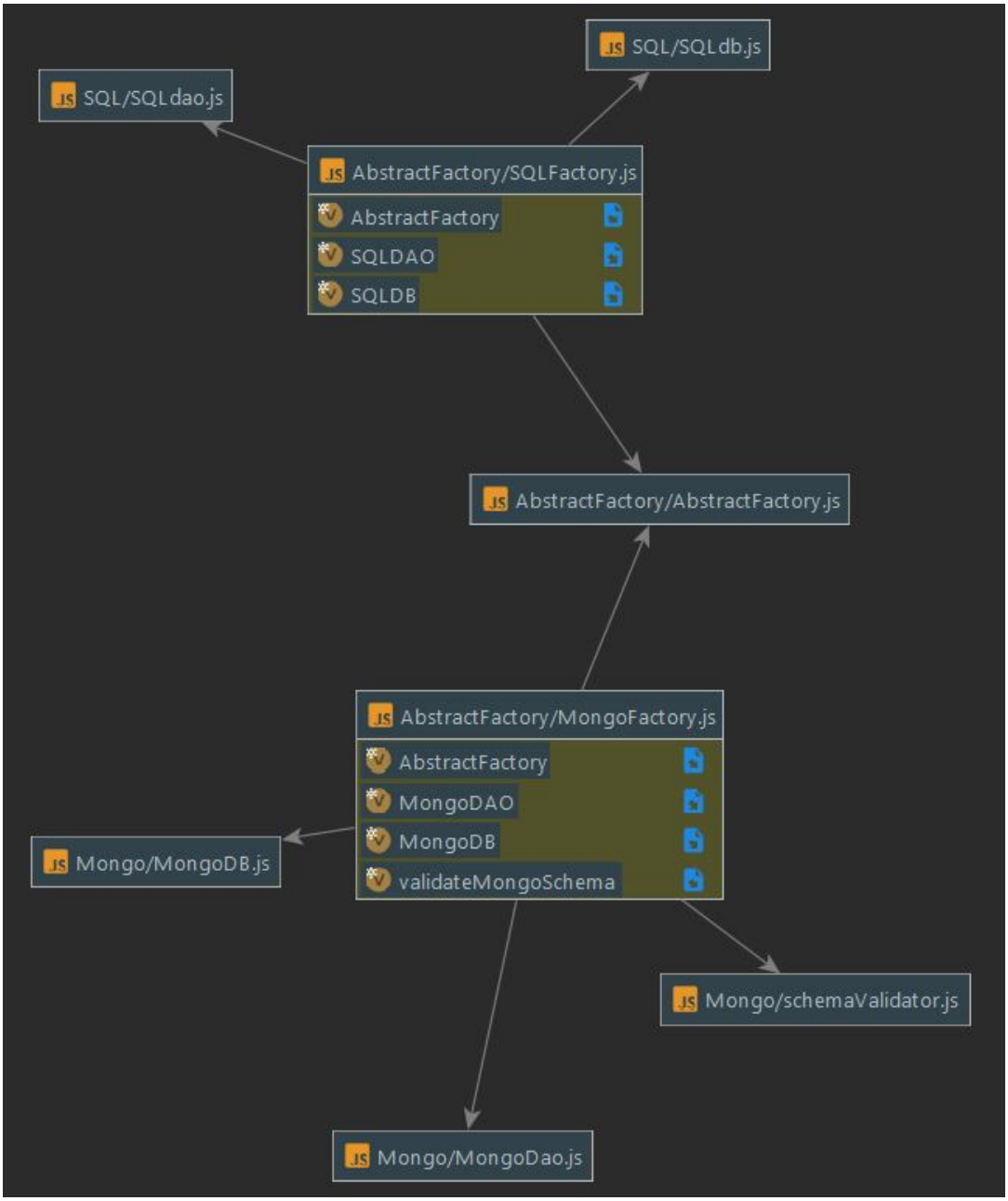
Rysunek 3: Implementacja Query Builder'a



Rysunek 4: 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 otrzymamyy obiekty ze sobą kompatybilne.



Rysunek 5: Diagram klas dla fabryki abstrakcyjnej

4.4 Singleton

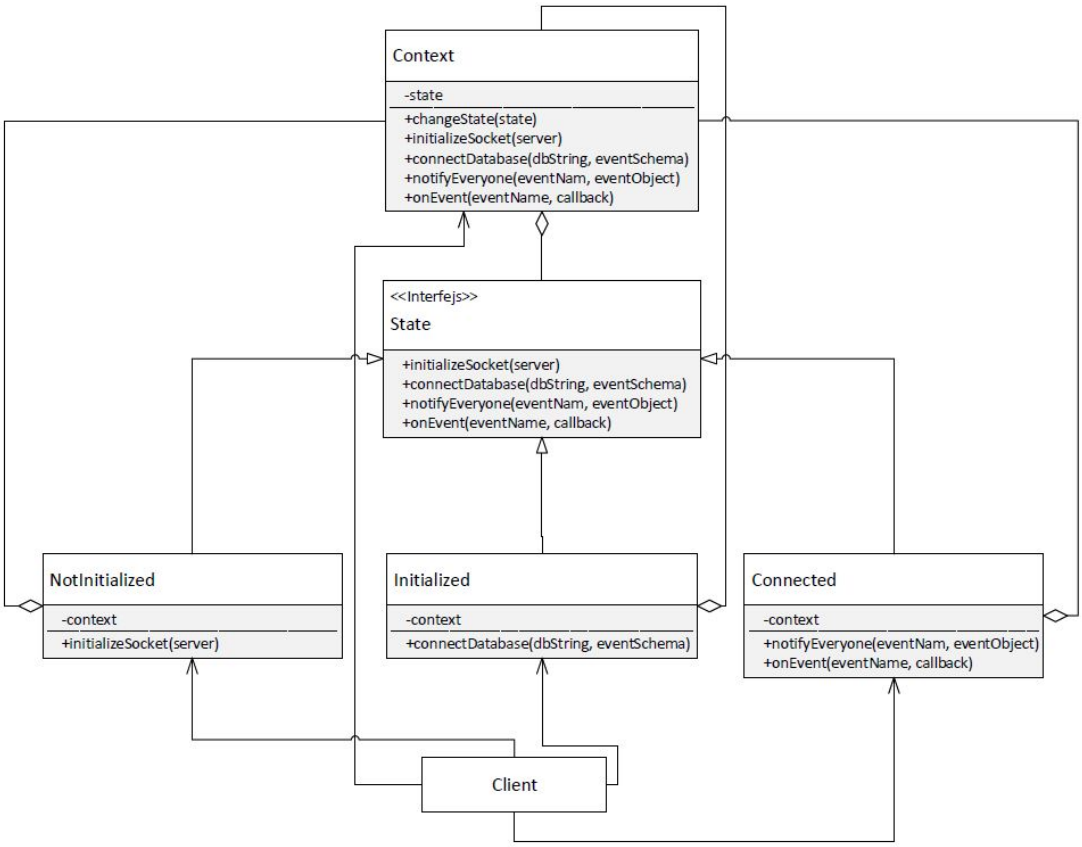
```
let _instance;

class AbstractFactory {
    constructor() {
        if (_instance) {
            return this;
        } else {
            this.isDbCreated = false;
            _instance = this;
        }
    }

    createDB(dbString, eventSchema) {
        throw new Error( message: "Not implemented");
    }

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

module.exports = AbstractFactory;
```

Rysunek 6: Implementacja fabryki abstrakcyjnej jako singletona

4.5 Maszyna Stanowa

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

    initializeSocket(endpoint) {
        this.context.io = socketIo(endpoint);
        this.context.changeState(new Initialized(this.context));
    }
}

class Initialized extends State {
    constructor(context) {
        super(context);
        this.context.error = "Not implemented, you can onEvent(eventName, callback) and sendEvent(eventName, eventObject) it this state";
    }

    onEvent(eventName, callback) {
        this.context.io.on(eventName, function(obj){
            callback(obj);
        });
    }

    sendEvent(eventName, eventObject) {
        this.context.io.emit(eventName, eventObject);
    }
```

Rysunek 7: Implementacja maszyny stanowej



Rysunek 8: Diagram klas maszyny stanowej

4.6 Dao Factory

```
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 9: Implementacja DAO Factory