

Backend

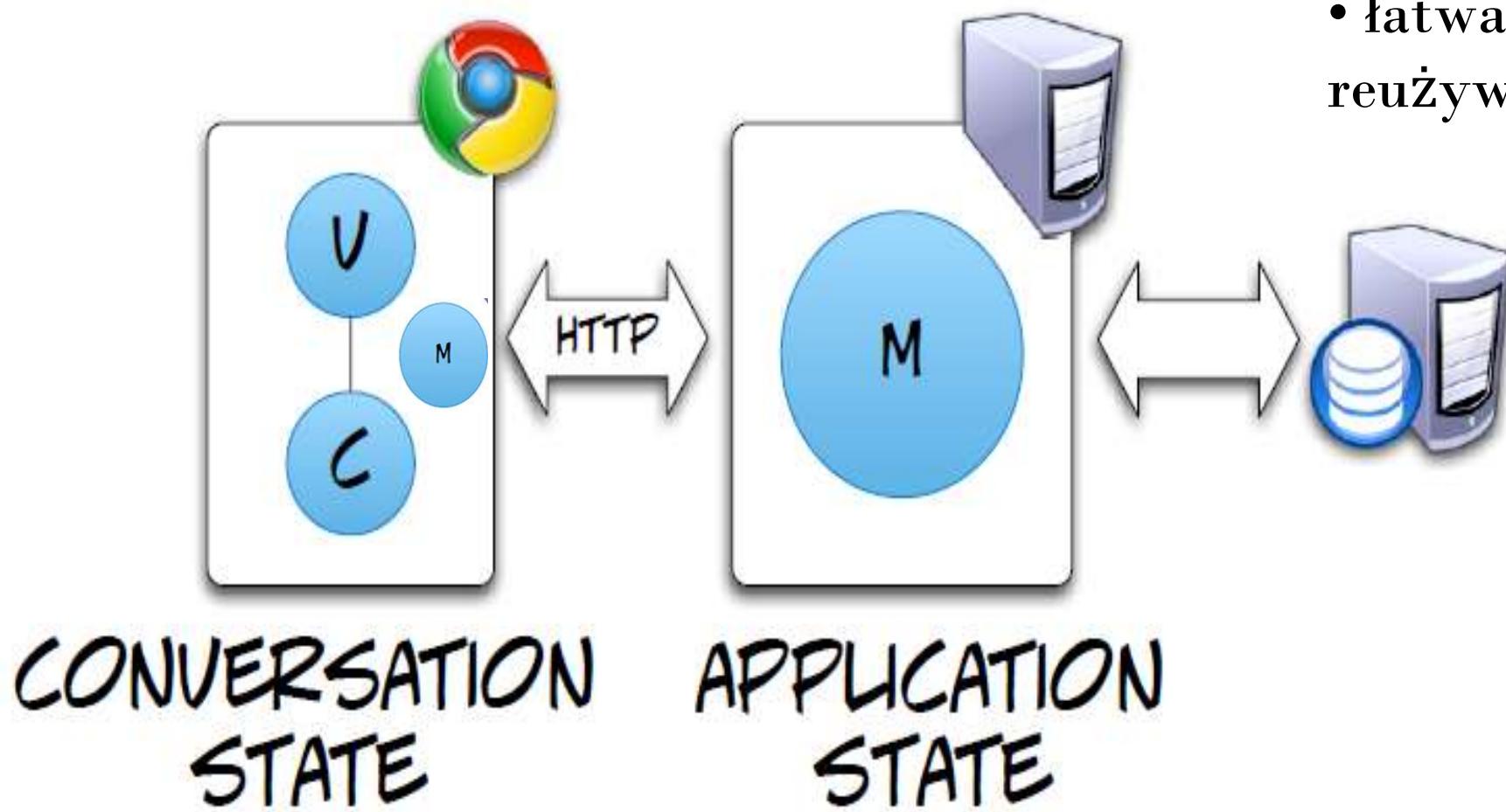
Stos MEAN vs Firebase

Node.js - Serverside Javascript

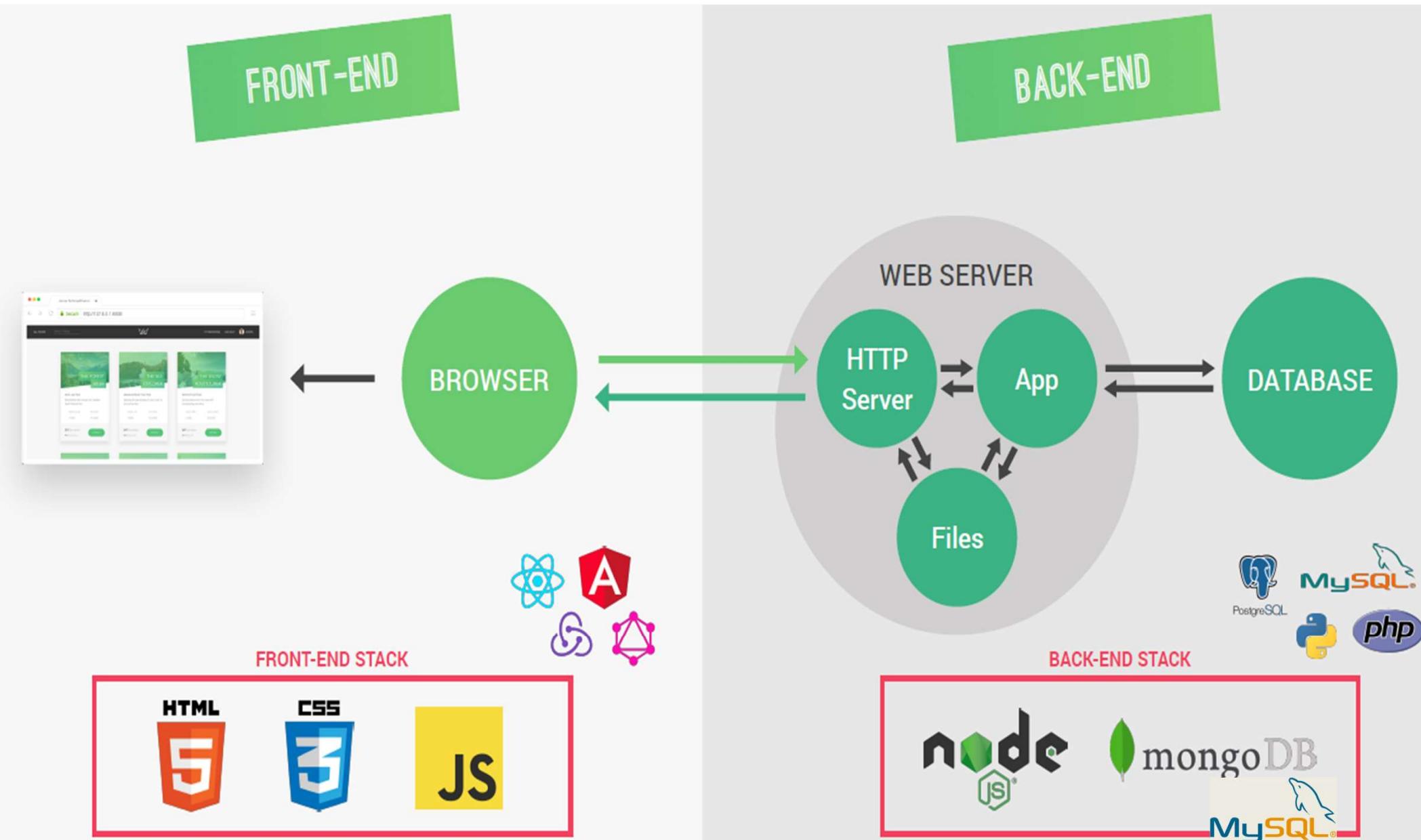
dr inż. Grzegorz Rogus

Aplikacje Webowe dzisiaj

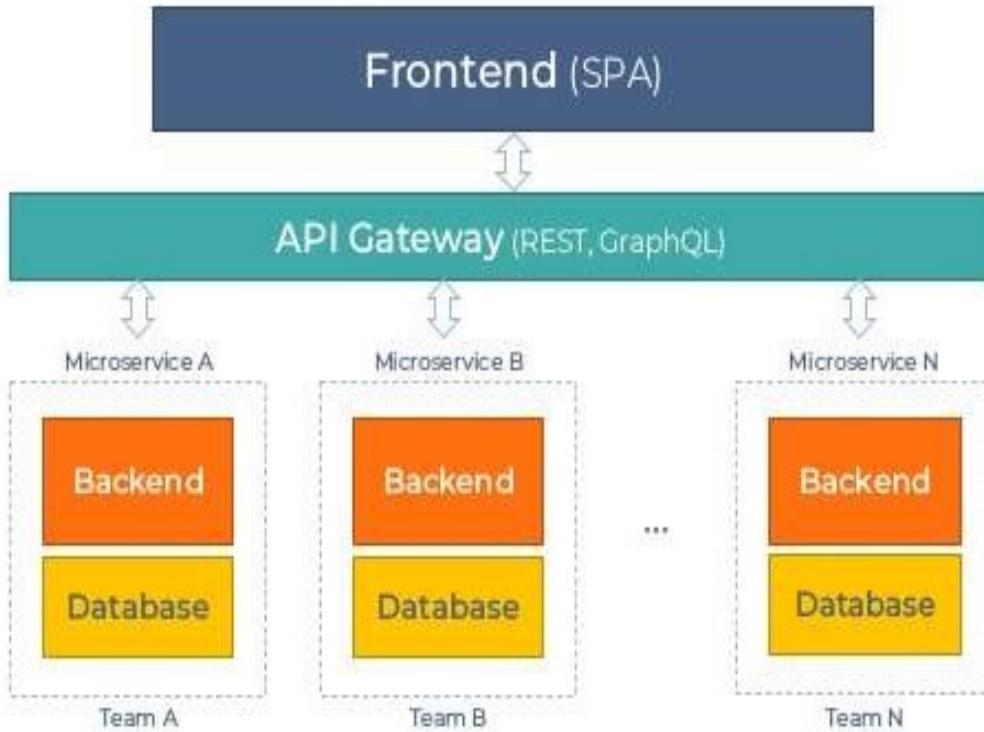
- łatwe do skalowania
- łatwa reużywalność



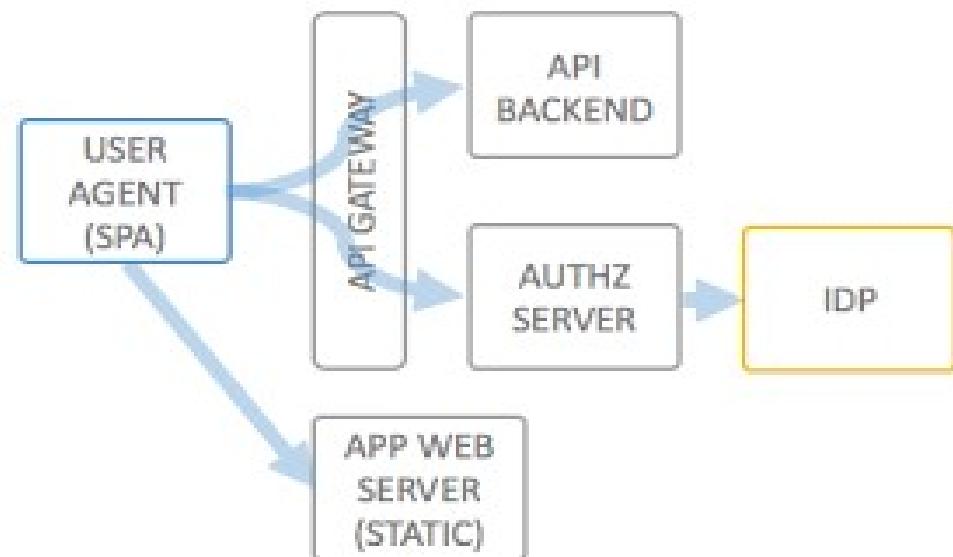
Nasz stos technologiczny



Rola Backend w SPA



- autentykacja użytkownika
- dostarczanie danych
- persystencja danych w bazie danych
- współdzielona logika biznesowa



Backend – sposoby realizacja

Technologie



Ruby



PHP



Python



C#



BAZY



MongoDB



MySQL

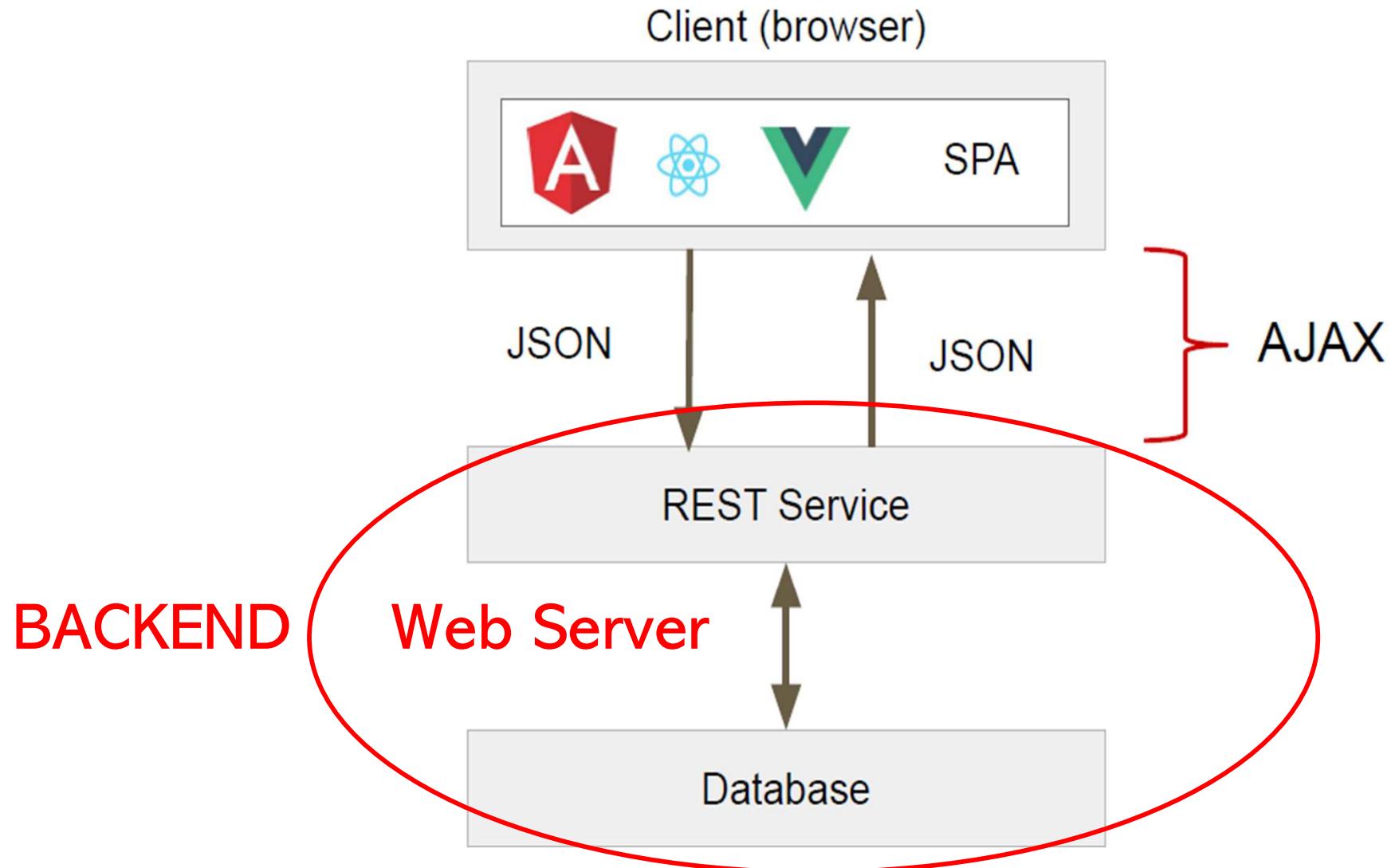


Oracle



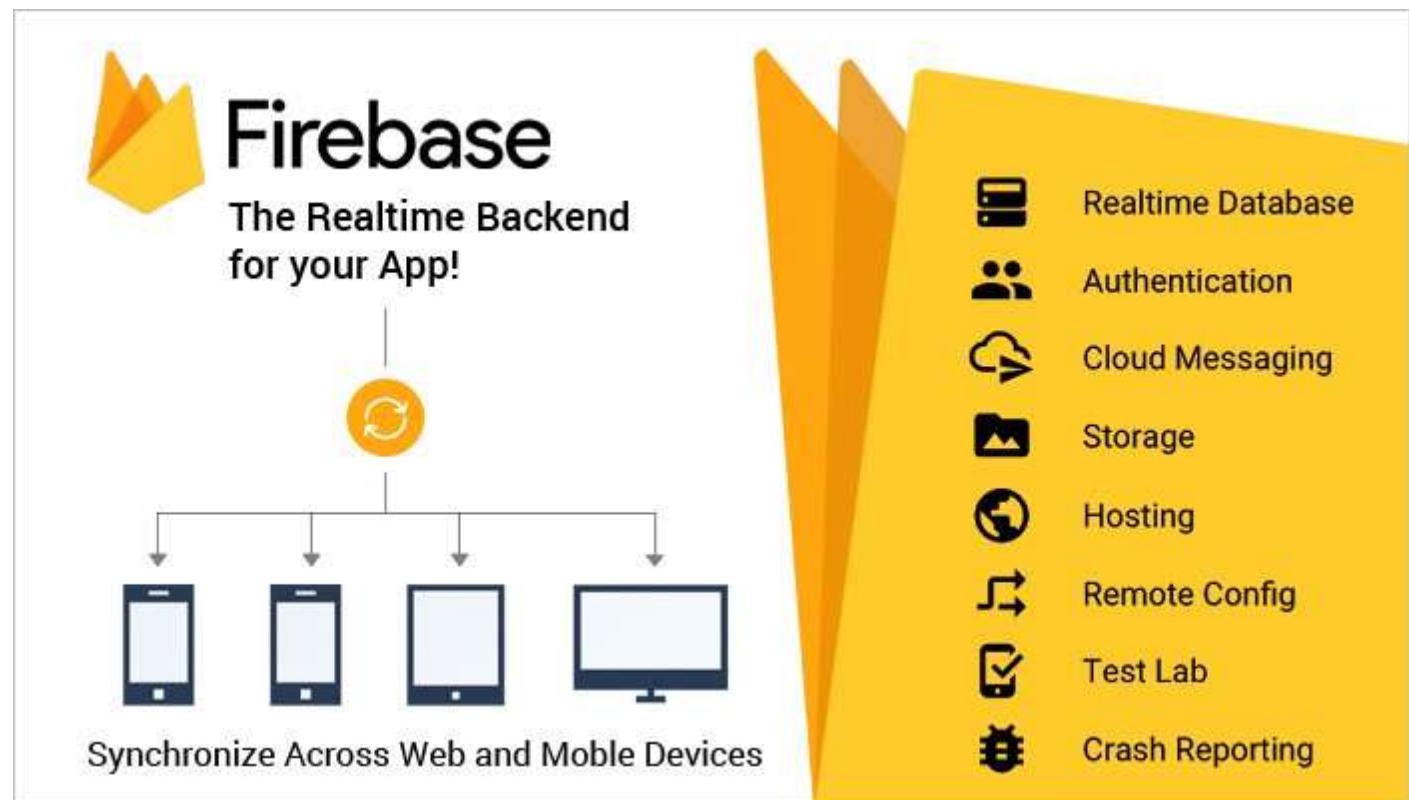
Postgresql

Single Page Application (SPA)

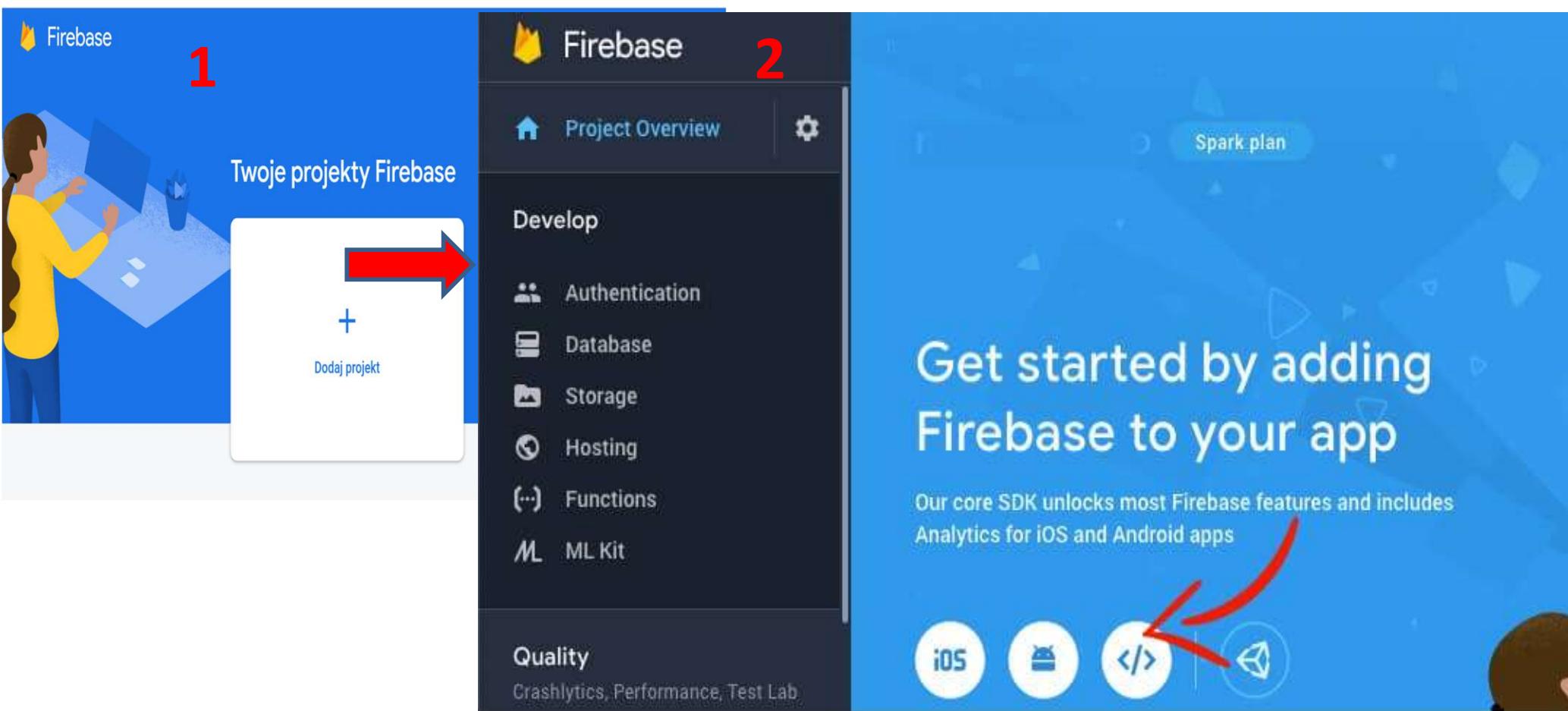


Nasz wybór

- Samodzielnie stworzony serwer w oparciu o stos MEAN
- Firebase (Backend-as-a-Service — BaaS) — Google CloudPlatform.



Firebase – konfiguracja konta



Dodaj Firebase do swojej aplikacji internetowej

1 Zarejestruj aplikację

Pseudonim aplikacji 

ListaKomentarzyGR

Aplikacje internetowe

Sk

 ListaKomentarzyGR

H

Pseudonim aplikacji

ListaKomentarzyGR 

Identyfikator aplikacji 

1:745908412191:web:090f06f8653fb028f0da2a

[Połącz z witryną w Hostingu Firebase](#)

2 Doda

Firebase SDK snippet

CDN  Konfiguracja 

Zanim zaczniesz używać usług Firebase, skopiuj i wklej poniższe skrypty na końcu tagu <body>

```
const firebaseConfig = {
  apiKey: "AIzaSyBj0mFv3il1fNt7x1CBuvbeHSGxrjNKqZI",
  authDomain: "listakomentarzy-e981e.firebaseio.com",
  databaseURL: "https://listakomentarzy-e981e.firebaseio.com",
  projectId: "listakomentarzy-e981e",
  storageBucket: "listakomentarzy-e981e.appspot.com",
  messagingSenderId: "745908412191",
  appId: "1:745908412191:web:090f06f8653fb028f0da2a"
};
```

The screenshot shows the Firebase Project Overview page. On the left, there's a sidebar with 'Programowanie' and several service links: Authentication, Database (circled in red), Storage, Hosting, Functions, and ML Kit. The main area shows 'Project Overview' with tabs for 'Develop' and 'Database'. Under 'Database', there are 'Data' and 'Rules' tabs. A large callout box labeled 'Cloud Firestore' contains information about it being the next generation of the Realtime Database. Another callout box labeled 'Realtime Database' provides details about its real-time synchronization. Red arrows and numbers indicate the steps: 1 points to the 'Cloud Firestore' callout, 2 points to the 'Realtime Database' callout, and another arrow points from the 'Database' link in the sidebar to the 'Database' tab in the main area.

Cloud Firestore

Database

Realtime Database
Store and sync data in realtime across all connected clients

Cloud Firestore
The next generation of the Realtime Database with more powerful queries and automatic scaling

lub wybierz Realtime Database



Realtime Database

Oryginalna baza danych Firebase.
Obsługuje synchronizację danych w czasie rzeczywistym, tak samo jak Cloud Firestore.

[Zobacz dokumentację](#)

[Więcej informacji](#)

[Utwórz bazę danych](#)

listakomentarzy-e981e

Format danych w Real Database

items

1 + X

```
hide: false
imie: "Grzegorz"
komentarz: "ABC Grzegorz"
```

2

```
hide: false
imie: "Iza"
komentarz: "CDE Iza"
```

5

```
hide: false
imie: "Kuba"
komentarz: "test Jakuba"
```

-LuPoSm_7D1h67NzUbBe

```
hide: true
imie: "Alicja"
komentarz: "test Alicji"
```

+ -LuPogisyawvTGBkQpj3

+ -LuPpuvmk142JmhXKk4G



Konfigurowanie Cloud Storage

1 Ustaw reguły zabezpieczeń w
Cloud Storage

2 Ustaw lokalizację Cloud Storage

Domyślnie reguły zezwalają na wszystkie operacje odczytu i zapisu przez uwierzytelnionych użytkowników.

Po zdefiniowaniu struktury danych **musisz stworzyć reguły, które je zabezpieczą.**[Więcej informacji](#)

Database



Cloud Firestore ▾

Dane

Reguły

Indeksy

Wykorzystanie

Home > komentarze > ZWtNtqopMZw2...

listakomentarzy-e981e

komentarze

⋮

ZWtNtqopMZw27YMI63r2

+ Utwórz kolekcję

+ Dodaj dokument

+ Utwórz kolekcję

komentarze

ZWtNtqopMZw27YMI63r2

>

tGAI70Jx02jm3w8scmUp

+ Dodaj pole

hide: false

imie: "Grzegorz"

komentarz: "Test GR"



Project Overview



Programowanie

Authentication

Database

Storage

Hosting

Functions

ML Kit

ListaKomentarzy ▾

Authentication

Użytkownicy

Metoda logowania

Szablony

Wykorzystanie

Dostawcy logowania

Dostawca	Stan
E-mail/hasło	Wyłączono
Telefon	Wyłączono
Google	Wyłączono
Gry Play	Wyłączono
Game Center <small>Beta</small>	Wyłączono
Facebook	Wyłączono

Wyszukaj według adresu e-mail, numeru telefonu lub identyfikatora UID użytkownika

Dodaj użytkownika

Identyfikator

Dostawcy

Utworzono

Zalogowano

Identyfikator UID użytkownika ↑

Dodaj użytkownika, używając adresu e-mail / hasła

E-mail

rogus@agh.edu.pl

Hasło

Test123

Angular – konfiguracja dla Firebase

1. `npm install firebase @angular/fire --save`

2.

```
export const environment = {  
  production: true,  
  firebase: {  
    apiKey: "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",  
    authDomain: "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",  
    databaseURL: "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",  
    projectId: "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",  
    storageBucket: "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",  
    messagingSenderId: "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"  
  }  
};
```

environment.ts



```
<script src="https://www.gstatic.com/firebasejs/4.9.0.firebaseio.js"></script>  
<script>  
  // Initialize Firebase  
  // TODO: Replace with your project's customized code snippet  
  var config = {  
    apiKey: "<API_KEY>",  
    authDomain: "<PROJECT_ID>.firebaseapp.com",  
    databaseURL: "https://<DATABASE_NAME>.firebaseio.com",  
    storageBucket: "<BUCKET>.appspot.com",  
    messagingSenderId: "<SENDER_ID>",  
  };  
  firebase.initializeApp(config);  
</script>
```

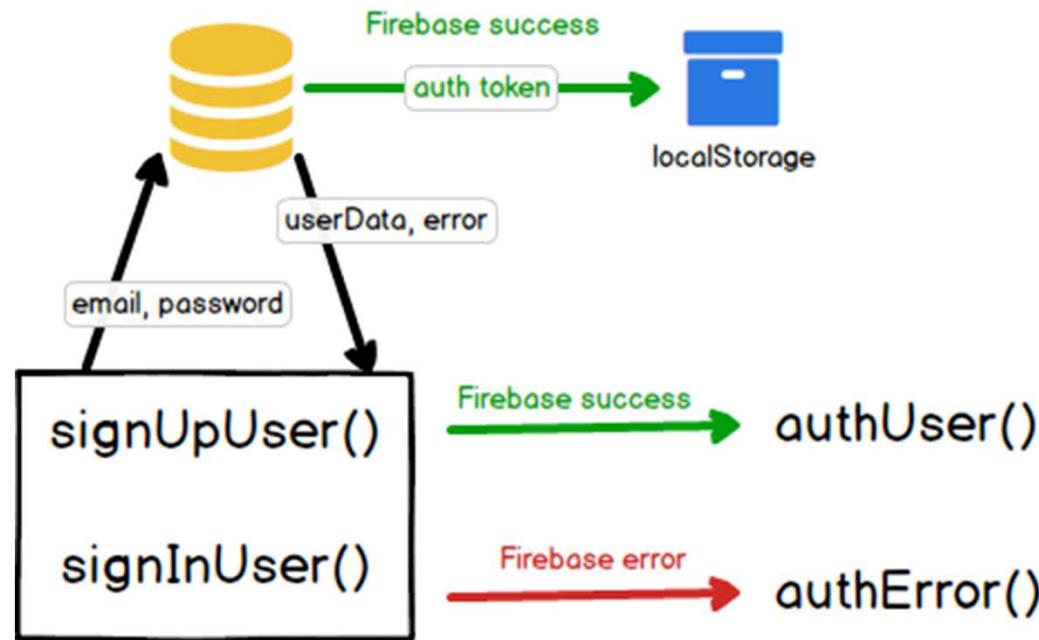
3.

```
import { AngularFireModule } from "@angular/fire";  
import { AngularFirestoreModule } from "@angular/fire/auth";  
import { AngularFirestoreModule } from '@angular/fire/firestore';  
import { AngularFireDatabaseModule } from '@angular/fire/database';  
import { environment } from './environments/environment';
```

app.module.ts

```
imports: [  
  AngularFireModule.initializeApp(environment.firebaseio),  
  AngularFireAuthModule, // do obsługi autentykacji  
  AngularFirestoreModule, // do obsługi baz danych  
  AngularFireDatabaseModule // do obsługi baz danych  
],
```

Angular – autentykacja z Firebase



```
import { AngularFireAuth } from "@angular/fire/auth";
.....
constructor(public afAuth: AngularFireAuth)
```

```
SignIn() {
  return this.afAuth.auth.signInWithEmailAndPassword(email, password)
    .then((result) => {
      localStorage.setItem('user', JSON.stringify(result));
    })
    .catch((error) => {
      console.log(error.message);
    });
}

SignOut() {
  return this.afAuth.auth.signOut().then(() => {
    localStorage.removeItem('user');
    this.router.navigate(['sign-in']);
  });
}
```

```
SignInUser (email, password) {
  return this.afAuth.auth.signInWithEmailAndPassword(email, password)
    .then((result) => {
      localStorage.setItem('user', JSON.stringify(result));
    })
    .catch((error) => {
      console.log(error.message);
    });
}
```

```
SignUpUser (email, password) {
  return this.afAuth.auth.createUserWithEmailAndPassword(email, password)
    .then((result) => {
      /* Np. wyslij mail w celu weryfikacji */
    })
    .catch((error) => {
      window.alert(error.message);
    });
}
```

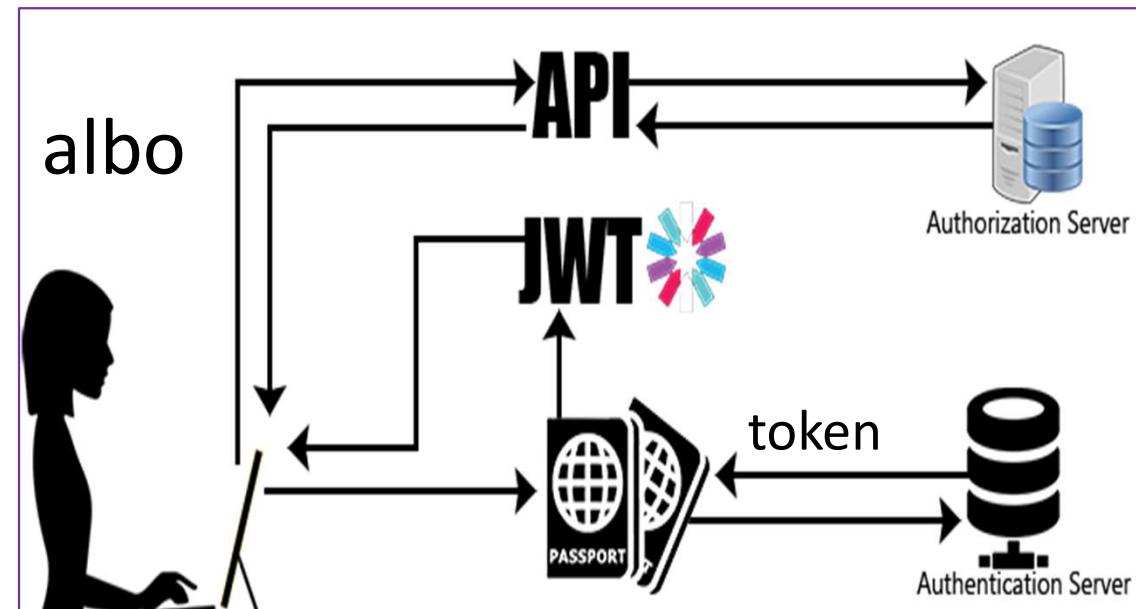
Angular - Autoryzacja

```
CanRead(user: User): boolean {  
  const allowed = [,admin', ,edytor', reader'];  
  return this.checkAuthorization(user,allowed);  
}
```

```
CanEdit(user: User): boolean {  
  const allowed = [,admin', ,edytor'];  
  return this.checkAuthorization(user,allowed);  
}
```

```
CanDelete(user: User): boolean {  
  const allowed = [,admin'];  
  return this.checkAuthorization(user,allowed);  
}
```

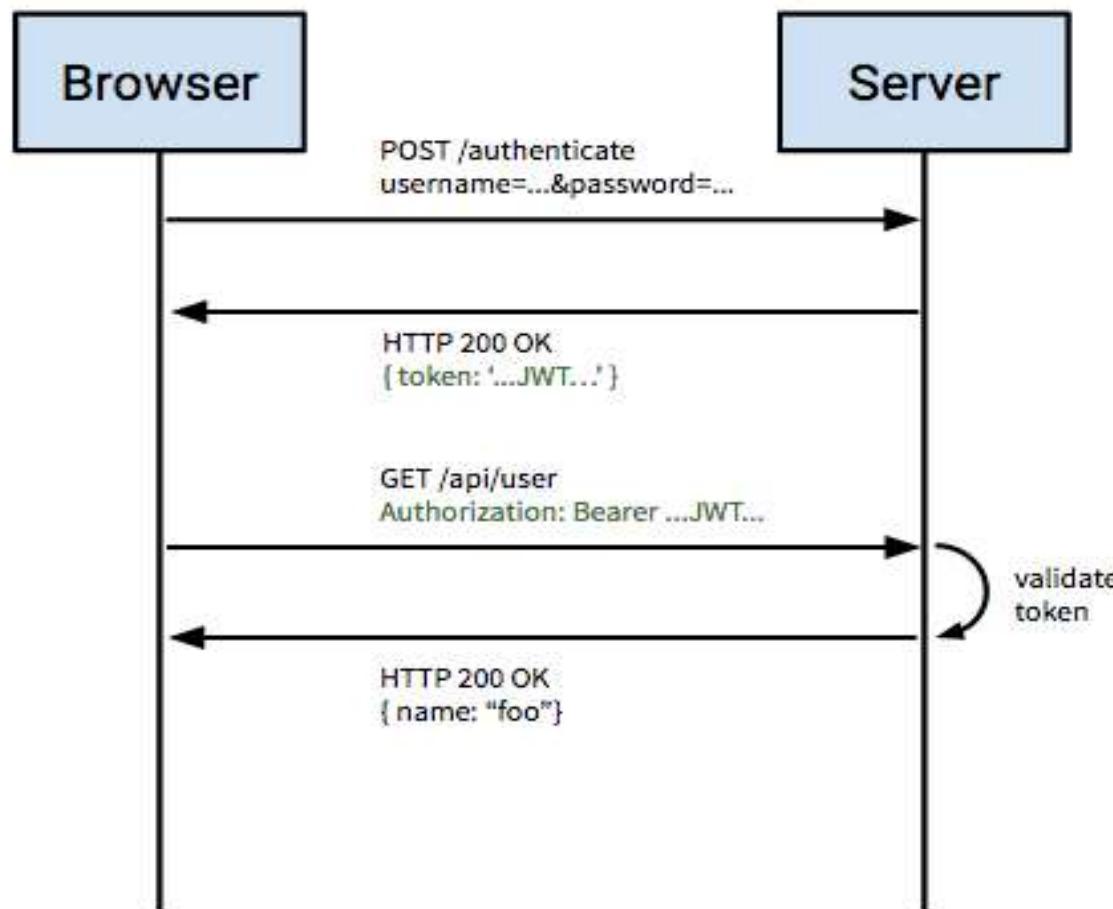
```
private checkAuthorization (user: User, allowedRoles: string[]): boolean {  
  if (!user) return false;  
  for (const role of allowedRoles) {  
    if (user.roles[role]) { return true; }  
  }  
}
```



```
Data: User = { uid: user.id  
               email: user.email;  
               roles: { admin: true;  
                         reader: true }  
 }
```

Rest API - autoryzacja

Modern Token-Based Auth



Firebase – realtime database

```
constructor(private db: AngularFireDatabase) {}

getComments(): Observable<any> {
    return this.db.list(PATH_KOMENTARZE).valueChanges();  }

getComment (id: number): Observable<any> {
    return this.db.object(PATH_KOMENTARZE + id).valueChanges();  }

addComment(item: Comment) {
    this.db.list(PATH_KOMENTARZE).set(String(item.id), item);  }

updateComment(item: Comment) {
    this.db.object(PATH_KOMENTARZE + item.id).update(item);  }

deleteAllComments(item: Comment) {
    this.db.object(PATH_KOMENTARZE + item.id).remove();  }
```

Firebase – FireStore database

```
deleteComment(item_id){  
    return  
this.db.collection('comments').doc(item_id).delete(); }
```

```
updateComment() {  
    this.db.collection('comments').doc('my-custom-id').set({'imie': this.imie,  
        'komentarz': this.komentarz, 'hide': this.hide}); }
```

```
addComment(){  
    this.db.collection('comments').add({'imie': this.imie, 'komentarz':  
this.komentarz});
```

```
getComment (commentId) {  
    this.wynik = this.db.doc('comments/'+commentId);  
    this.comment = this.wynik.valueChanges(); }
```

```
getComments(){  
    this.afs.collection('comments').snapshotChanges().subscribe( snapshots => {  
        resolve(snapshots)    })  
}
```

HttpClient – współpraca z serwerem danych

Konfiguracja

1. Import modułu httpClient

```
import { HttpClient Module} from  
'@angular/common/http';
```

```
@NgModule({  
  declarations: [  
    AppComponent  
  ],  
  imports: [  
    BrowserModule,  
    HttpClientModule  
  ],  
  providers: [],  
  bootstrap: [AppComponent]  
})  
export class AppModule {  
}
```

AppModule

2. Import HttpClient w serwisie

```
import { HttpClient, HttpHeaders } from '@angular/common/http';  
  
export class ProductService{  
  ...  
  constructor(private httpClient: HttpClient) {  
    ...  
  }  
  ...  
}
```

ProductService.service.ts

HttpClient – współpraca z serwerem danych odbiór danych – get()

3. Odczyt danych

```
export class ProductService{  
  productsUrl = 'api/products';  
  
  constructor(private httpClient: HttpClient) {  
  }  
  
  getProducts(): Observable<Product[]> {  
    return this.httpClient.get<Product[]>(this.productsUrl);  
  }  
  ...  
  ...  
}
```

ProductService.service.ts

HttpClient.get zwraca response jako untyped JSON object.
Zastosowanie specyfikatora typu np. <Product[]>, daje obiekt zrzutowany.

HttpClient – współpraca z serwerem danych

```
constructor (private http: HttpClient) {}

getProducts(): Observable<Product[]> {
    return this.http.get<Product[]>(PATH);  }

getProduct(id: number): Observable<any> {
    return this.http.get(PATH+id);  }

addProduct(produkt: Produkt) {
    this.http.post<Produkt>(PATH, produkt).subscribe(
        res => { console.log(res) },
        (err: HttpErrorResponse) => { console.log(err) }      );
}

updateProduct(produkt: Produkt) {
    this.http.put(PATH + produkt.id, produkt).subscribe();  }

deleteAllProduct(produkt: Produkt) {
    this.http.delete(PATH + produkt.id).subscribe();  }
```

HttpClient – współpraca z serwerem danych

```
getCustomers(): Observable<Customer[]> {  
    return this.http.get('/api/customers');  
}
```

1 wersja

```
getCustomer(id): Observable<Customer> {  
    return this.http.get(`/api/customers/${id}`);  
}
```

```
getCustomers(): Observable<Customer[]> {  
    return this.http.get <Customer[]> ('/api/customers');  
}
```

2 wersja

```
getCustomer(id): Observable<Customer> {  
    return this.http.get <Customer> (`/api/customers/${id}`);  
}
```

```
postCustomer(customer): Promise {  
    return this.http.post('/api/customers', customer)  
        .toPromise()  
        .then((data) => data);  
}
```

3 wersja

httpClient w akcji

Odwołania do serwerowego API w klasie usługi

```
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import { of } from 'rxjs/observable/of';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Student } from './student';

const httpOptions = { headers: new HttpHeaders({ 'Content-Type': 'application/json' }) };

@Injectable()
export class StudentService {

  private studentsApiUrl = 'http://localhost:5000/api/student';

  constructor(private http: HttpClient) { }

  getStudents(): Observable<Student[]> {
    return this.http.get<Student[]>(this.studentsApiUrl);
  }

  updateStudent(student: Student): Observable<any> {
    const url = `${this.studentsApiUrl}/${student.id}`;
    return this.http.put(url, student, httpOptions);
  }
}
```

httpClient w akcji

Wykorzystanie usługi dostępowej do API w klasach komponentów

```
...
students: Student[];
...
constructor(private studentService: StudentService) { }
...
getStudents(): void {
  this.studentService.getStudents()
    .subscribe(students => this.students = students);
}
```

```
...
student: Student;
...
constructor(private studentService: StudentService) { }
...
save(): void {
  this.studentService.updateStudent(this.student)
    .subscribe(() => this.goBack());
}
...
```

Czym jest NodeJS?



- Data utworzenia: **2009 r.**
- Niskopoziomowa implementacja silnika **V8 JavaScript'u** po stronie serwera.
- Napisany w języku **C/C++** (8000 linii) oraz **JavaScript** (2000 linii), obsługuje programy napisane w **JavaScript**.

*« A platform built on Chrome's
JavaScript runtime for easily building
fast, scalable network applications. »*
<http://nodejs.org/>

Platformę utworzoną na podstawie środowiska uruchomieniowego JavaScript przeglądarki internetowej Chrome, przeznaczoną do łatwego tworzenia szybkich, skalowalnych aplikacji.



<http://nodejs.org/>

Node.js, w skrócie Node, to środowisko uruchomieniowe języka JavaScript open source po stronie serwera.

Za pomocą środowiska Node.js można uruchamiać aplikacje i kod JavaScript w wielu miejscach poza przeglądarką, na przykład na serwerze.

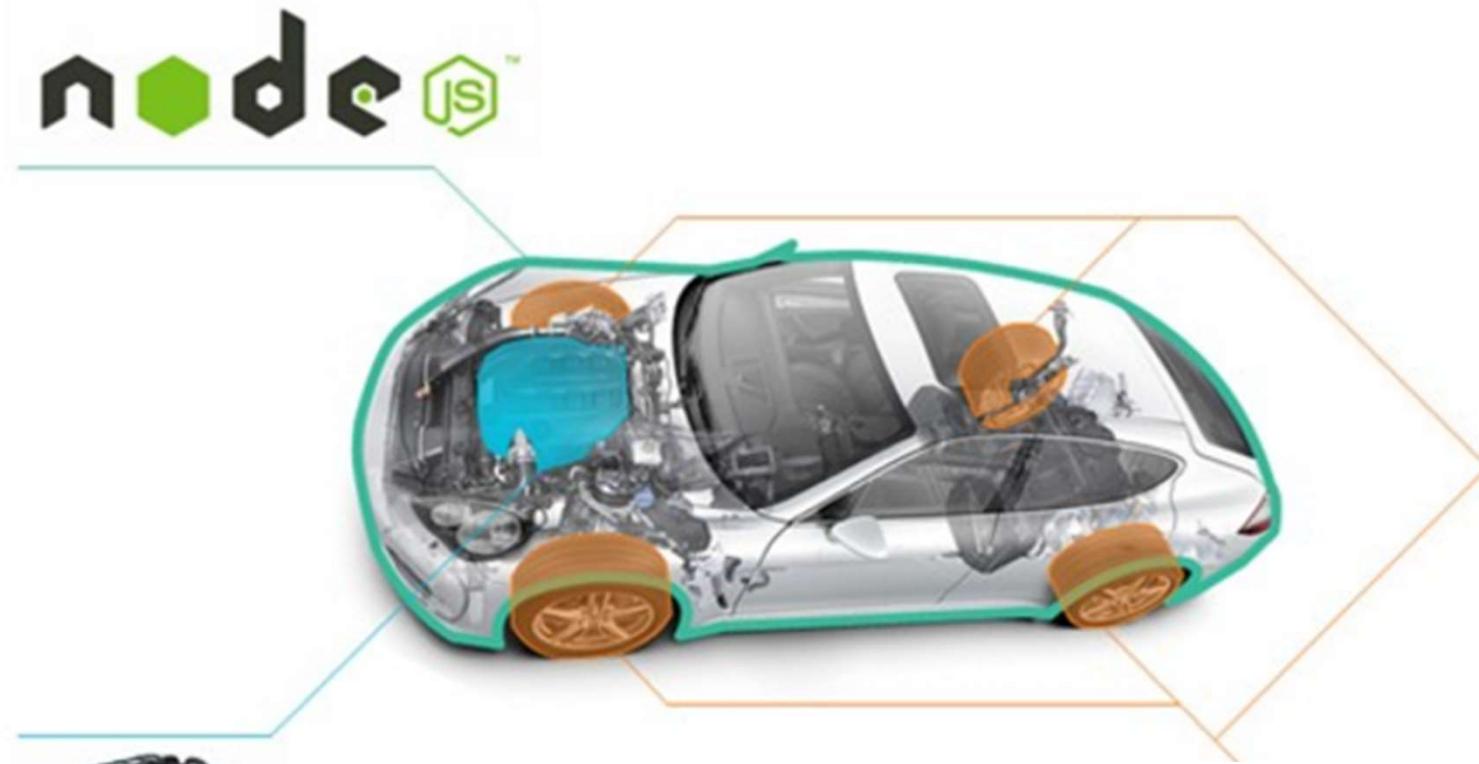
Node.js - nowoczesne środowisko uruchomieniowe języka JavaScript, działające na serwerze (poza przeglądarką).

Jest dystrybuowane na zasadzie otwartego oprogramowania (OpenSource).

Jego podstawową funkcją (nie jedyną) jest możliwość tworzenia serwerów aplikacji internetowych opartych protokoły HTTP.

Sterowany zdarzeniami wykorzystując system wejścia/wyjścia (I/O) który jest asynchroniczny.

Czym jest Node.js

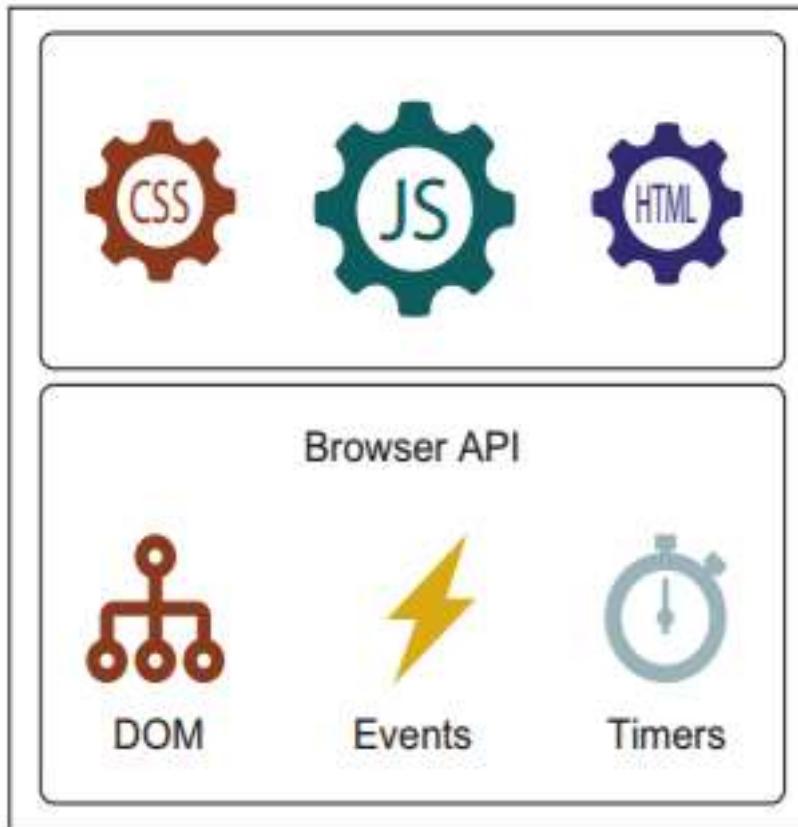


Node.js używa opartego na zdarzeniach, nieblokującego modelu wejścia-wyjścia, co zapewnia lekkość i efektywność.

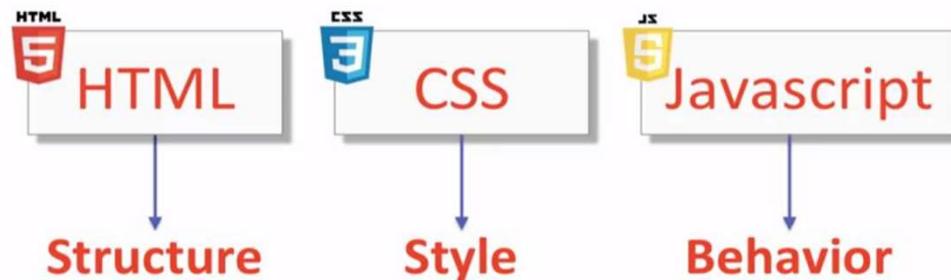
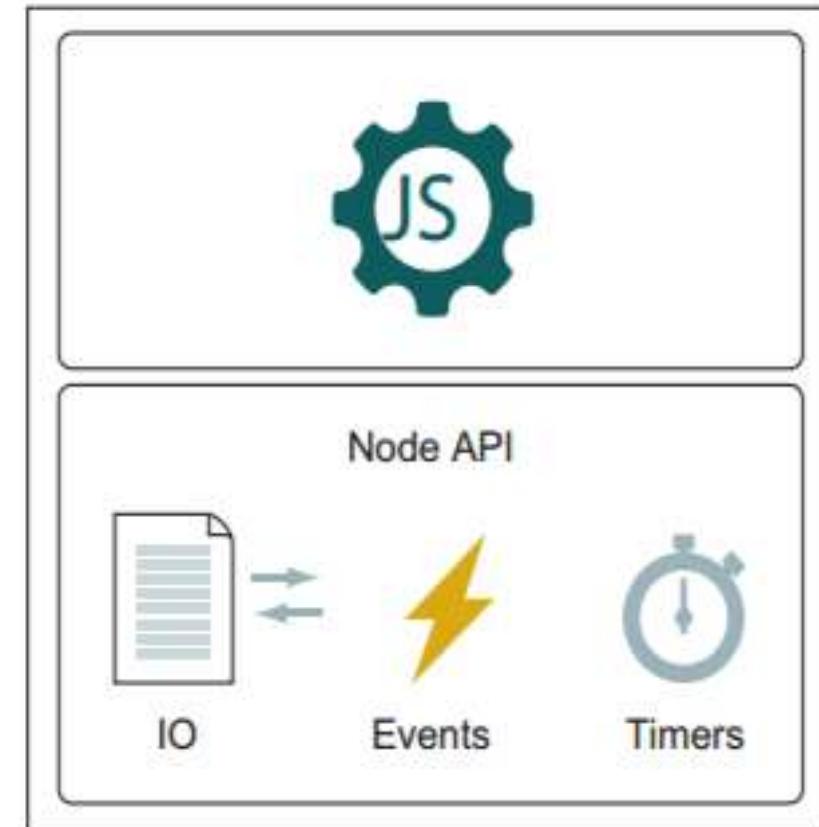
Stanowi doskonałe rozwiązanie dla działających w czasie rzeczywistym aplikacji intensywnie korzystających z danych oraz aplikacji rozproszonych w różnych urządzeniach

Aplikacje JS w różnych środowiskach

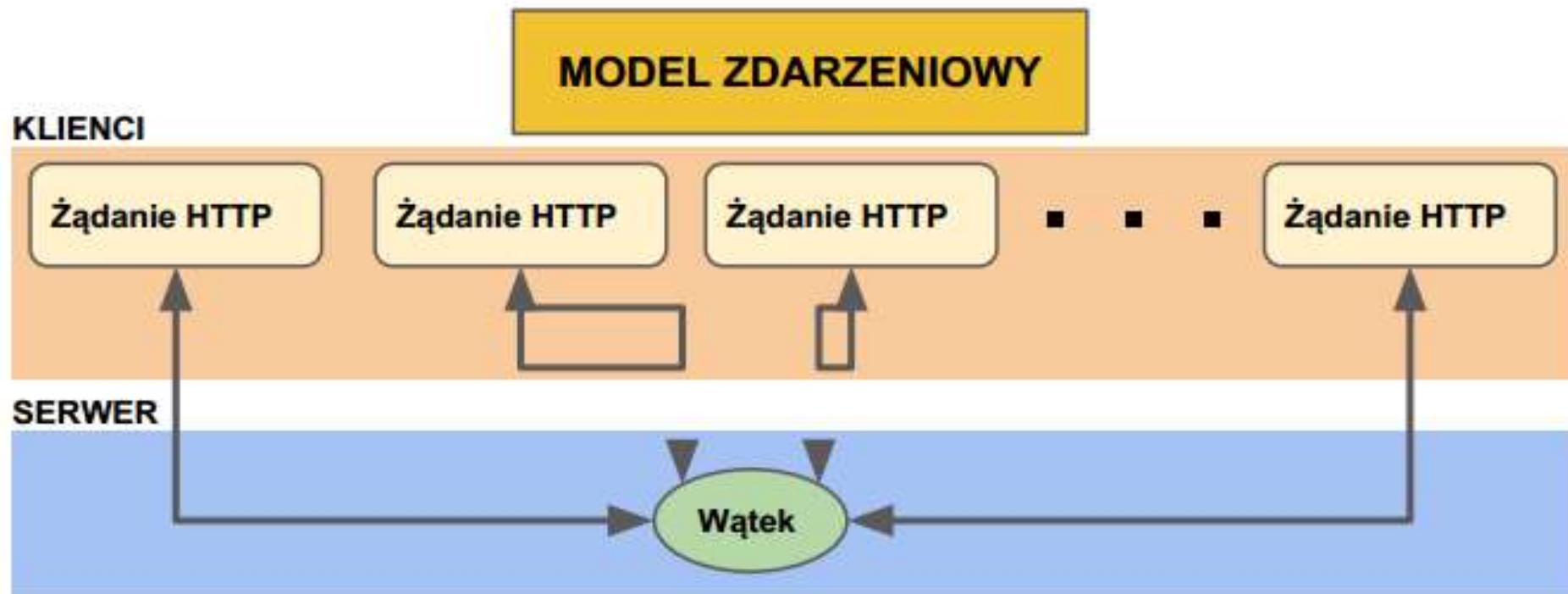
Browser infrastructure



Node.js infrastructure



NodeJS

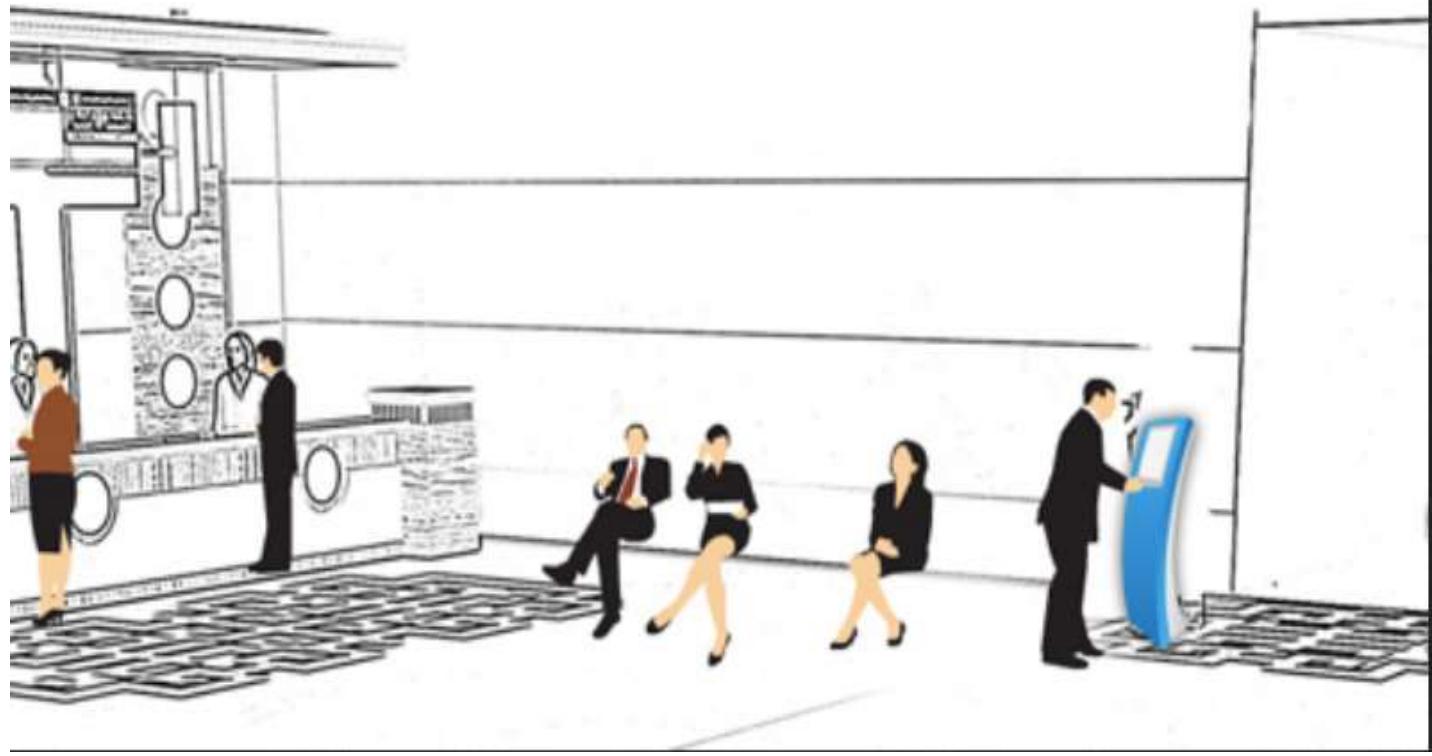


W modelu zdarzeniowym Node.js wykorzystuje tylko jeden wątek do obsługi wielu zadań, oraz “pętlę zdarzeń” co powoduje że aplikacja taka jest bardzo wydajna i skalowalna. W praktyce przy żądaniach które nie wymagają złożonych operacji obliczeniowych można obsłużyć nawet do 1 miliona żądań jednocześnie.



I/O blokujące

I/O nieblokujące

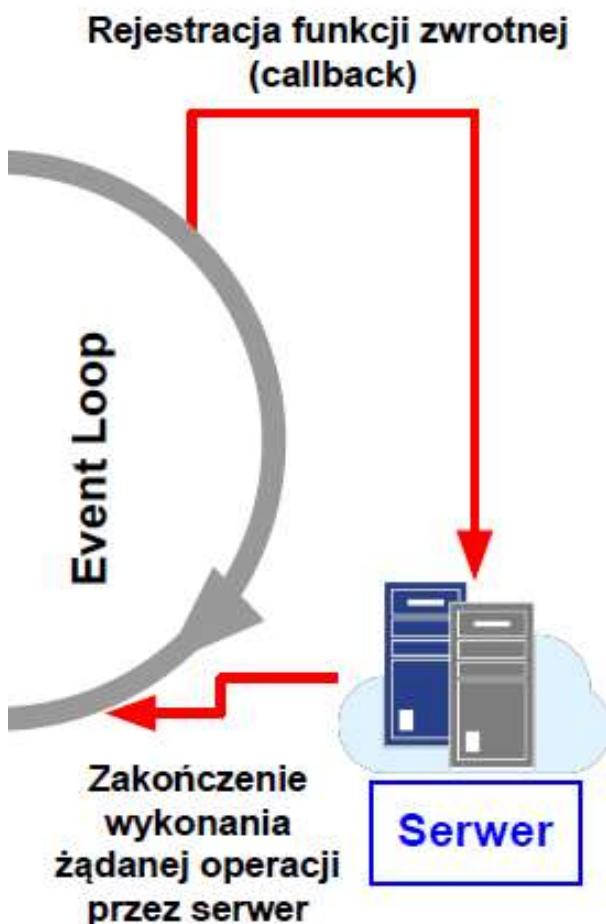


NodeJS



NodeJS

Koncepcja działania aplikacji Node.js:



Podejście klasyczne:

```
var user = User.findById(1);
console.log(user);
```

W podejściu klasycznym wykonanie zapytania metodą `findById()`, wstrzyma wątek aż do zakończenia operacji odpytania bazy danych, następnie dopiero po uzyskaniu danych wątek zostanie zwolniony.

Podejście przez wywołanie zwrotne (callback):

```
User.findById(1, (user) => {
  console.log(user);
});
```



Wywołanie zwrotne będzie wykonane po zakończeniu operacji bazodanowej, ale w czasie tego wątek będzie wolny i może obsługiwać inne operacje. Funkcja callback to tak naprawdę funkcja "czekająca".

Cechy NodeJS

Asynchroniczność i sterowanie zdarzeniami

- Serwer wywołuje kolejne API, jedno po drugim.
- Nie czeka na zakończenie wywołanej funkcji aby zwrócić dane.
- Za wszystko odpowiada mechanizm obsługi zdarzeń.

Duża szybkość - silnik Chrome V8 zapewnia dużą szybkość w wykonywaniu kodu.

Jednowątkowość z zachowaniem dużej skalowalności

- Node.js używa modelu jednowątkowego z pętlą zdarzeń.
- Potrafi obsłużyć znacznie więcej żądań niż tradycyjne serwery (np. Apache).
- Mechanizm zdarzeń pomaga serwerowi na udzielanie odpowiedzi w sposób nieblokujący,
- pozwala to na wysoką skalowalność.
- W klasycznych serwerach jest tworzona ograniczona liczba wątków obsługujących żądania.

Licencja - Node.js jest tworzony na licencji MIT.

Projekty, aplikacje, firmy - długa lista wykorzystywania Node.js: eBay, General Electric, GoDaddy, Microsoft, PayPal, Uber, Wikipins, Yahoo, Yammer...

Dlaczego NodeJS?

- Programiści mogą tworzyć aplikacje sieciowe w jednym języku, co pomaga w redukcji operacji przełączania kontekstu między programowaniem po stronie klienta i serwera, a także pozwala na współdzielenie kodu między klientem i serwerem. Dzięki temu ten sam kod można stosować na przykład do weryfikacji formularzy lub w logice gry.
- JSON to obecnie bardzo popularny format wymiany danych, a ponadto jest rodzimym formatem języka JavaScript.
- JavaScript to język używany w różnych bazach danych typu NoSQL (takich jak CouchDB i MongoDB), a więc praca z nimi nie nastręcza trudności (na przykład język powłoki i zapytań MongoDB to JavaScript, natomiast funkcjonalność Map/Reduce w CouchDB również opiera się na JavaScript).
- Node używa jednej maszyny wirtualnej (V8) zgodnej ze standardem ECMAScript5. Innymi słowy, z użyciem nowych funkcji języka JavaScript w Node nie musisz czekać do chwili, aż we wszystkich przeglądarkach internetowych zostanie wprowadzona ich obsługa

Kiedy stosować NodeJS?

- zalecany do tworzenia aplikacji:
 - z dużą liczbą operacji wejścia/wyjścia,
 - strumieniowania danych, np. video,
 - Single Page Applications (SPA),
 - udostępniających API w formacie JSON,
 - z intensywną wymianą danych w czasie rzeczywistym na wielu urządzeniach, np. portale społecznościowe,
- nie zalecany przy aplikacjach intensywnie wykorzystujących procesor (CPU),
- npm - system pakietów Node.js - największy zbiór bibliotek open source na świecie (260000) → łatwa produkcja aplikacji internetowych,
- Node.js = środowisko uruchomieniowe + biblioteki JavaScript

Aplikacje typu DIRT (ang
Data Intensive Real Time)

Do czego stosować NodeJS

- Serwery internetowe HTTP
- Mikrousługi lub bezserwerowe zaplecza interfejsu API
- Sterowniki umożliwiające dostęp do bazy danych i wykonywanie zapytań
- Interakcyjne interfejsy wiersza poleceń
- Aplikacje klasyczne
- Biblioteki serwera i klienta IoT w czasie rzeczywistym
- Wtyczki dla aplikacji klasycznych
- Skrypty powłoki do manipulowania plikami lub uzyskiwania dostępu do sieci
- Biblioteki i modele uczenia maszynowego

Pierwszy projekt w NodeJS

Większość aplikacji Node.js składa się z 3 części:

- import wymaganych modułów - używa się dyrektywy **require**,
- utworzenie serwera - serwer będzie oczekiwany na żadania klientów i zwracał odpowiedzi,
- odczytywanie żądań i zwracanie odpowiedzi - podstawowe działanie serwera.

```
var http = require("http");

var server = http.createServer(function (request, response) {
    // Wysyłanie nagłówków protokołu HTTP
    // Status HTTP: 200 : OK, Content Type: text/plain
    response.writeHead(200, {'Content-Type': 'text/plain'});

    // Wysyłanie ciała odpowiedzi, niezależnie od rodzaju żądania
    response.end('Pierwszy projekt w Node.js\n');
});

server.listen(5000);
console.log('Server działa na http://127.0.0.1:5000/');
```

Tworzenie serwera w NodeJS

```
var http = require('http');
var fs = require('fs');
var url = require('url');

http.createServer( function (request, response) {
    // Parsuje żądanie zawierające nazwę pliku
    var pathname = url.parse(request.url).pathname;
    // Wyświetlanie nazwy pliku, którego dotyczyło żądanie
    console.log("Request for " + pathname + " received.");

    fs.readFile(pathname.substr(1), function (err, data) {
        if (err) {
            console.log(err);
            response.writeHead(404, {'Content-Type': 'text/html'});
        } else {
            response.writeHead(200, {'Content-Type': 'text/html'});
            response.write(data.toString()); // zwracanie treści wybranego pliku
        }
        response.end(); // wysyłanie odpowiedzi
    });
}).listen(5000);

console.log('Serwer uruchomiony na http://127.0.0.1:5000/');
```

Tworzenie serwera z użyciem NodeJS

index.html

```
<html>
<head>
<title>Pierwszy projekt</title>
</head>
<body>
Pierwszy projekt w Node.js
</body>
</html>
```

```
$ node server.js
```

Serwer uruchomiony na <http://127.0.0.1:5000/>

Otwieramy w przeglądarce: <http://localhost:5000/index.html>

REST Client



Browser /
HTTP Tools/plug-ins



Any windows /
.NET app



Any Java app

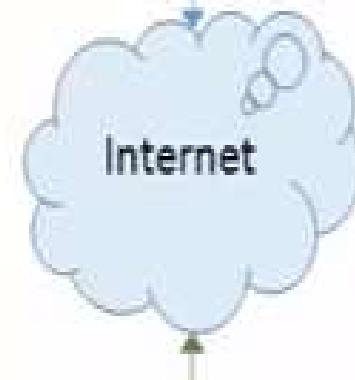


PHP, JSP, Servlet, CF,
ASP.NET, ASP.NET MVC,
Any Javascript SPA,
Angular JS,
etc.



iOS,
Android,
Windows Phone,
PhoneGap/Cordova,
Sencha,
etc.

GET, POST, PUT, DELETE etc.



REST Service



- * core
 - db.js
 - httpMsgs.js
 - server.js
- * controllers
 - employee.js
 - app.js



NodeJS

Wróćmy do struktury podstawowego serwera WWW:

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

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Czy da się przedstawić tą strukturę w bardziej uporządkowany i usystematyzowany sposób ?

TAK...

Express.js



Express.js jest “pakietem” które pozwala na tworzenie aplikacji internetowych o architekturze MVC (serwisowej).

Ułatwia obsługę żądań HTTP i wprowadza szerokie możliwości tworzenia aplikacji internetowych / sieciowych.

```
> npm install express
```

Express - instalacja

```
$ npm install express --save
```

Inne ważne moduły, które warto od razu zainstalować:

- **body-parser** - warstwa pośrednia obsługująca JSON, Raw, Text i dane formularza przekazane w URL,
- **cookie-parser** - przetwarza nagłówki ciasteczek (cookie header) i dodaje obiekt do `req.cookies`, w którym klucze to nazwy przesłanych ciasteczek,
- **multer** - warstwa pośrednia w Node.js do obsługi multipart/form-data (kodowanie danych z formularza).

Express.js

```
//- plik app.js

const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.send('Hello World!')
});

app.listen(3000, () => {
  console.log('Serwer dziala na porcie 3000!')
});
```



Prosta aplikacja napisana z użyciem Express.js nastuchująca żądań na porcie 3000 i odsyłająca odpowiedź w postaci tekstu.

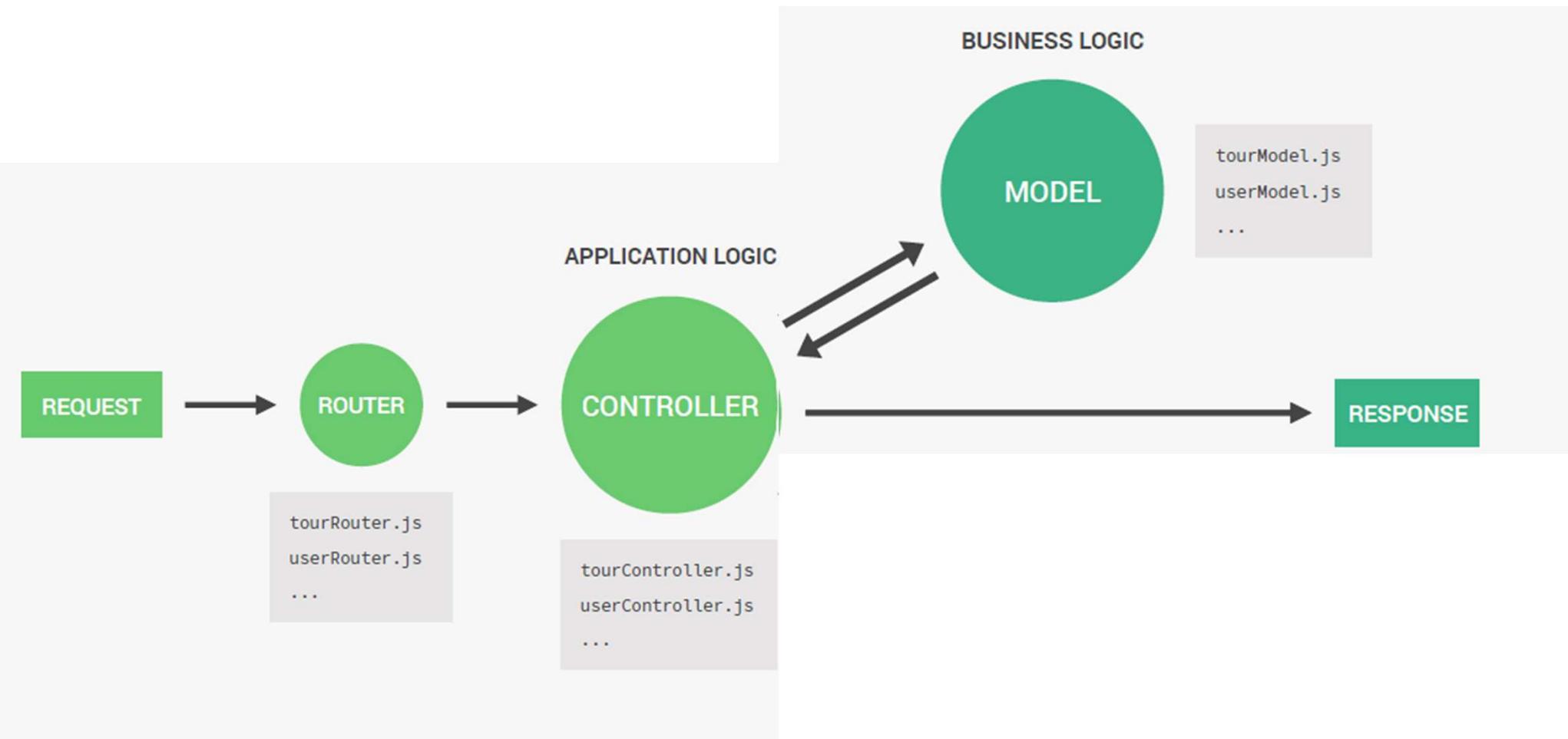
Przedstawiony kod w zasadzie niczym nie różni się od “prostego” serwera HTTP....
ma jednak kilka zalet

Express.js - inna wersja

```
var express = require('express');

var app = express();
app.get('/', function(req, res) {
    res.send('Pozdrowienia od GR!');
});
app.listen(3000, function() {
    console.log('Aplikacja nasluchuje na porcie 3000!');
});
```

Architettura MVC w EXPRESS APP



Sample Web Server – Project Structure

```
✓ NODEJS-EXPRESS-MYSQL
  ✓ app
    ✓ config
      JS db.config.js
    ✓ controllers
      JS tutorial.controller.js
    ✓ models
      JS db.js
      JS tutorial.model.js
    ✓ routes
      JS tutorial.routes.js
  > node_modules
  {} package-lock.json
  {} package.json
  JS server.js
```

Sample Web Server – server.js

```
const express = require("express");
const cors = require("cors");

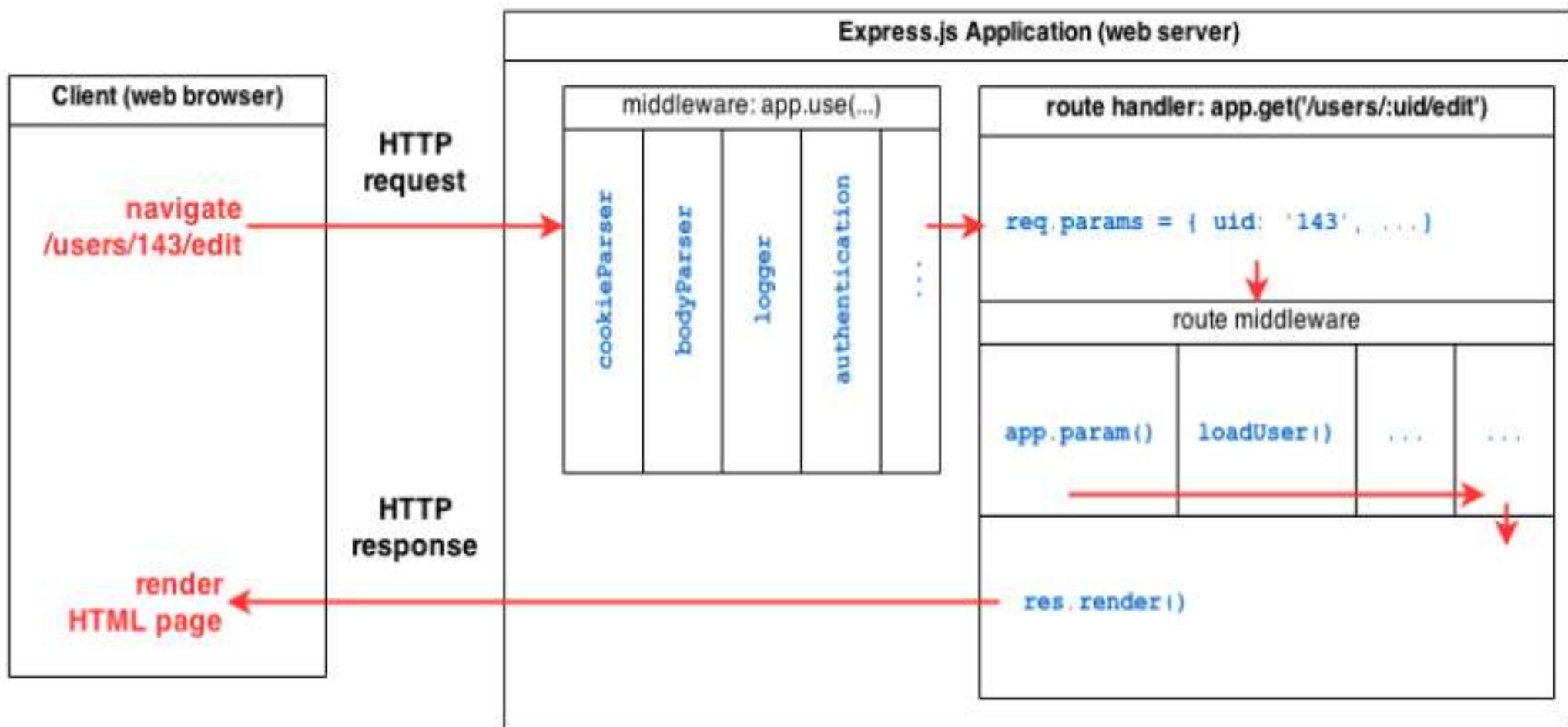
const app = express();
var corsOptions = { origin: "http://localhost:8081" };
app.use(cors(corsOptions));
// parse requests of content-type - application/json
app.use(express.json());
// parse requests of content-type - application/x-www-form-urlencoded
app.use(express.urlencoded({ extended: true }));
// simple route
app.get("/", (req, res) => { res.json({ message: "Web Server app" }); });

require("./app/routes/studentsRouter.js")(app);

// set port, listen for requests
const PORT = process.env.PORT || 8080; app.listen(PORT, () => {
console.log(`Server is running on port ${PORT}.`);});
```

ExpressJS i warstwy pośrednie

Warstwy pośredniczące dodajemy do ExpressJS używając `app.use` dla dowolnej metody albo `app.VERB` (np. `app.get`, `app.delete`, `app.post`, `app.update`, ...)



Express.js

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');

var routes = require('./routes/index');
var users = require('./routes/users');

var app = express();

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');

// uncomment after placing your favicon in /public
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', routes);
app.use('/users', users);
module.exports = app;
```

Plik: app.js

Pliki tras kontrolera,
wszystkie które znajdują
się w kartotece "routes"

Określenie języka
szablonów oraz katalogu
w którym się znajdują

Zmienna przechowująca
informację o katalogu
projektu.

Definicja ścieżki do
kartoteki "public"
widocznej dla klienta

Związanie plików
kontrolerów z odpowiednimi
trasami. Odwzorowanie
adresów URL na strukturę
katalogów

Express.js

```
var express = require('express');
var router = express.Router();

/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});

module.exports = router;
```

Plik: routes/index.js

req - obiekt żądania przychodzącego od klienta

res - obiekt odpowiedzi (do klienta)

Pojedyncza trasa obsługująca określone żądanie przychodzące od klienta.

Odpowiedź na żądanie klienta jest formułowana za pomocą mechanizmu funkcji wywołania zwrotnego "callback".

Najważniejszą rolę w kodzie powyżej spełnia obiekt "router", który jest odpowiedzialny za "trasowanie" czyli obsługę wszystkich wywołań adresów URL.

Każda trasa może posiadać jedną lub więcej funkcji obsługi trasy. W momencie rejestracji żądania router określa która funkcja obsługi trasy zostanie wykonana.

Ogólna postać funkcji obsługi trasy wygląda następująco:

router.METODA(TRASA, OBSŁUGA TRASY)

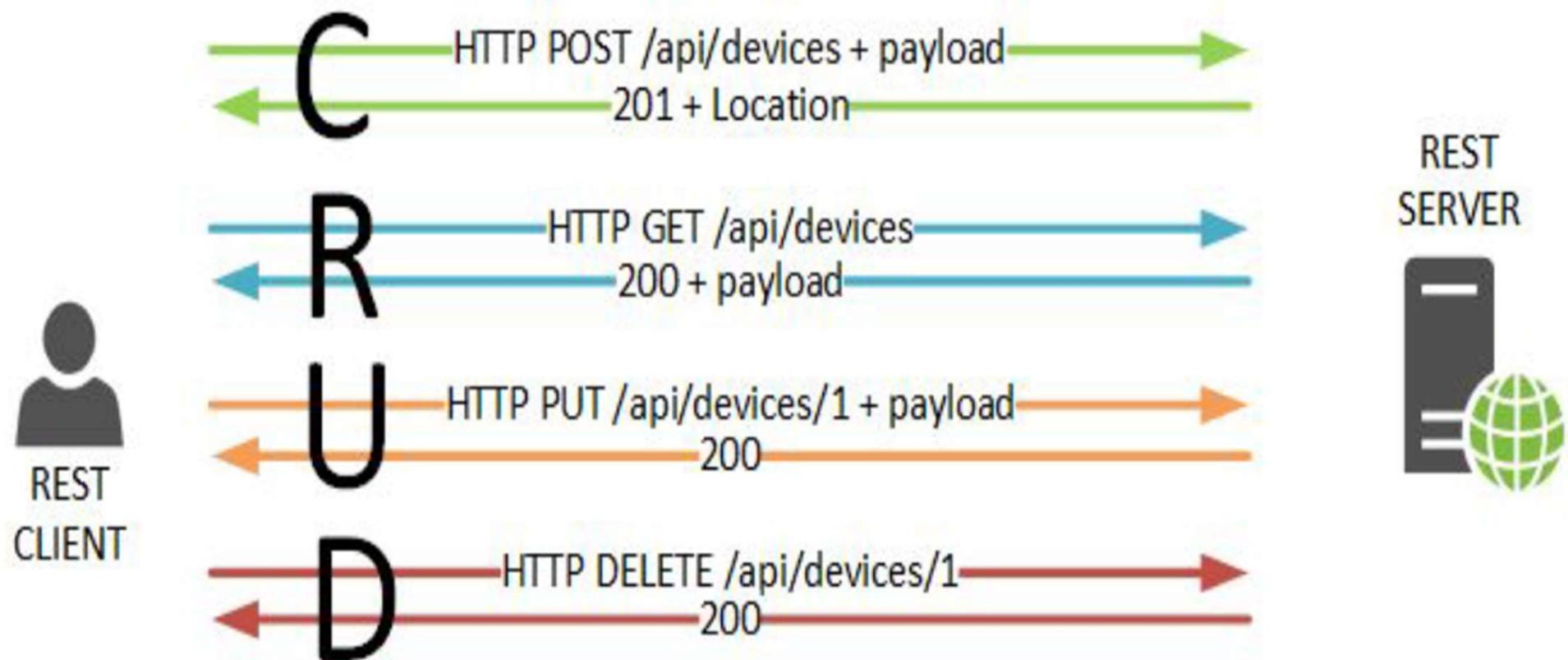
Metoda HTTP np.
GET, POST, PUT,
DELETE

Trasa: adres URL
żądania klienta

Najczęściej wywołanie zwrotne
realizujące określoną funkcjonalność.

```
app.get('/', (req, res) => res.send('GET'));
app.post('/', (req, res) => res.send('POST'));
app.put('/', (req, res) => res.send('PUT'));
app.delete('/', (req, res) => res.send('DELETE'));
```

CRUD - Create, Read, Update, Delete



Express.js – prosty routing

Przykłady funkcji obsługi żądań:

```
router.get('/', function (req, res) {
  res.send('Hello World!');
});

router.post('/', function (req, res) {
  res.send('Żądanie POST');
});

router.put('/user', function (req, res) {
  res.send('Żądanie PUT dla URL /user');
});

router.delete('/user', function (req, res) {
  res.send('Żądanie DELETE dla URL /user');
});
```

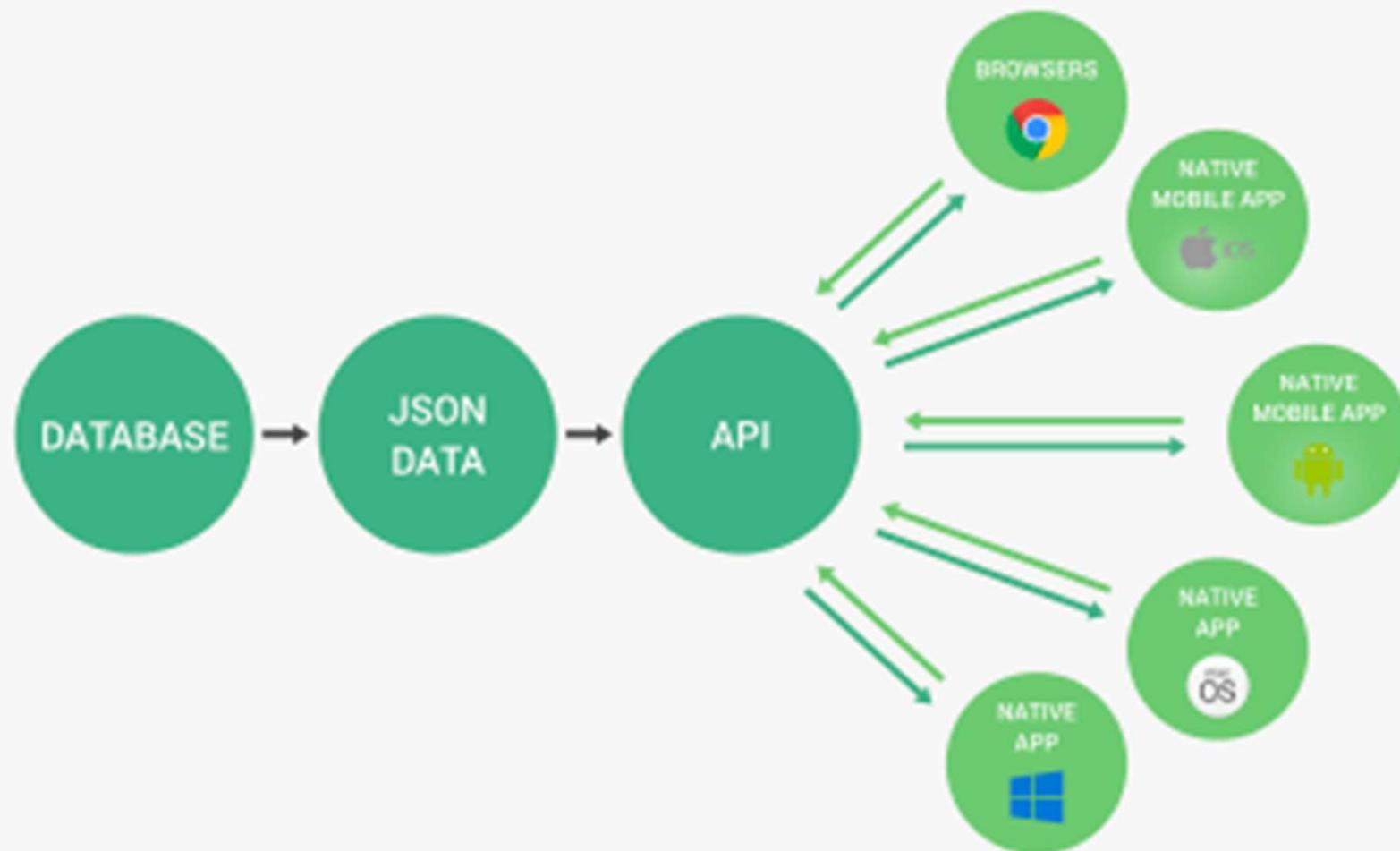
Funkcja obsługująca tą samą trasę “ / ”, ale dla dwóch różnych możliwych metod HTTP.

Bardzo często “trasa” w postaci adresu URL oraz metoda (np. GET) w literaturze określana jest jako “END-POINT”.

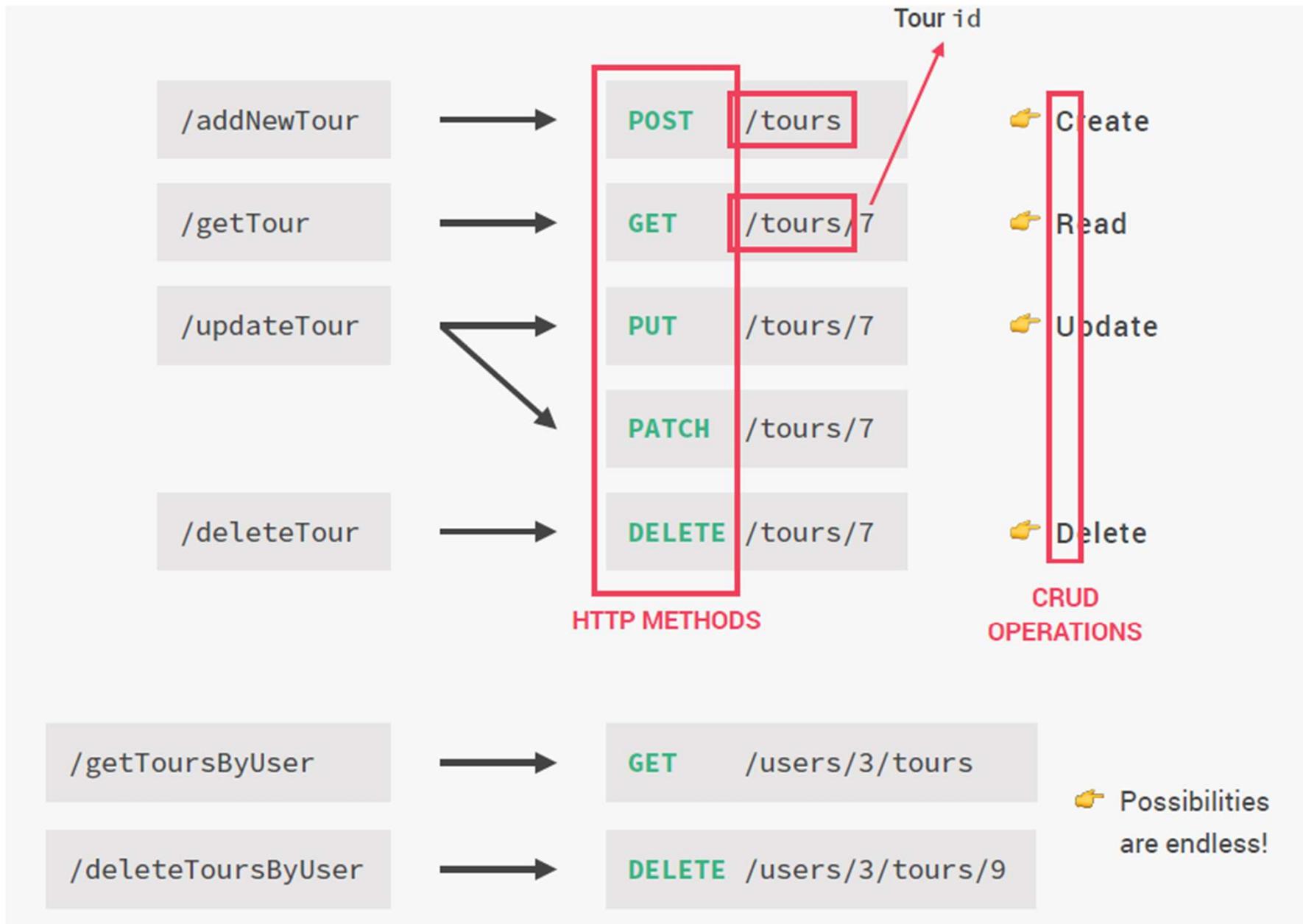
Podobnie jak poprzednio dwie usługi “END-POINT”.

Jedno API wielu konsumentów

👉 Web APIs



Używaj HTTP metod w REST API server nie używaj ich w nazwie adresu



REST API

Klasyczne API RESTful, zawiera cztery podstawowe operacje CRUD:

Resource (URI)	POST (create)	GET (read)	PUT (update)	DELETE (destroy)
/zadania	nowe zad.	lista zad.	błąd	błąd
/zadania/:id	błąd	zad. o :id	aktualizacja :id	usuń 1 zad. o ID

Express.js – REST API

Parametry trasy (adresu URL).

W Express.js możliwe jest wykorzystanie adresu URL do przekazania od klienta do serwera danych (podobnie jak to jest w samym HTTP gdzie można przez adres przekazać dane). Umiejętność przekazywania i odbierania parametrów za pomocą adresu URL jest również konieczna przy tworzeniu aplikacji o charakterze API.

Przykład adresu URL:

```
http://localhost:3000/users/10/car/9483
```

Trasa mapująca ten adres:

```
Route: /users/:userId/car/:carId
```

Przekazywanymi parametrami są:

```
{ userId: 10, carId: 9483 }
```

Dane przekazywane w adresie URL są przenoszone przez protokół HTTP a następnie w aplikacji Express.js są enkapsulowane do obiektu żądania:

`req.params`

W naszym przykładzie dostępu uzyskujemy przez:

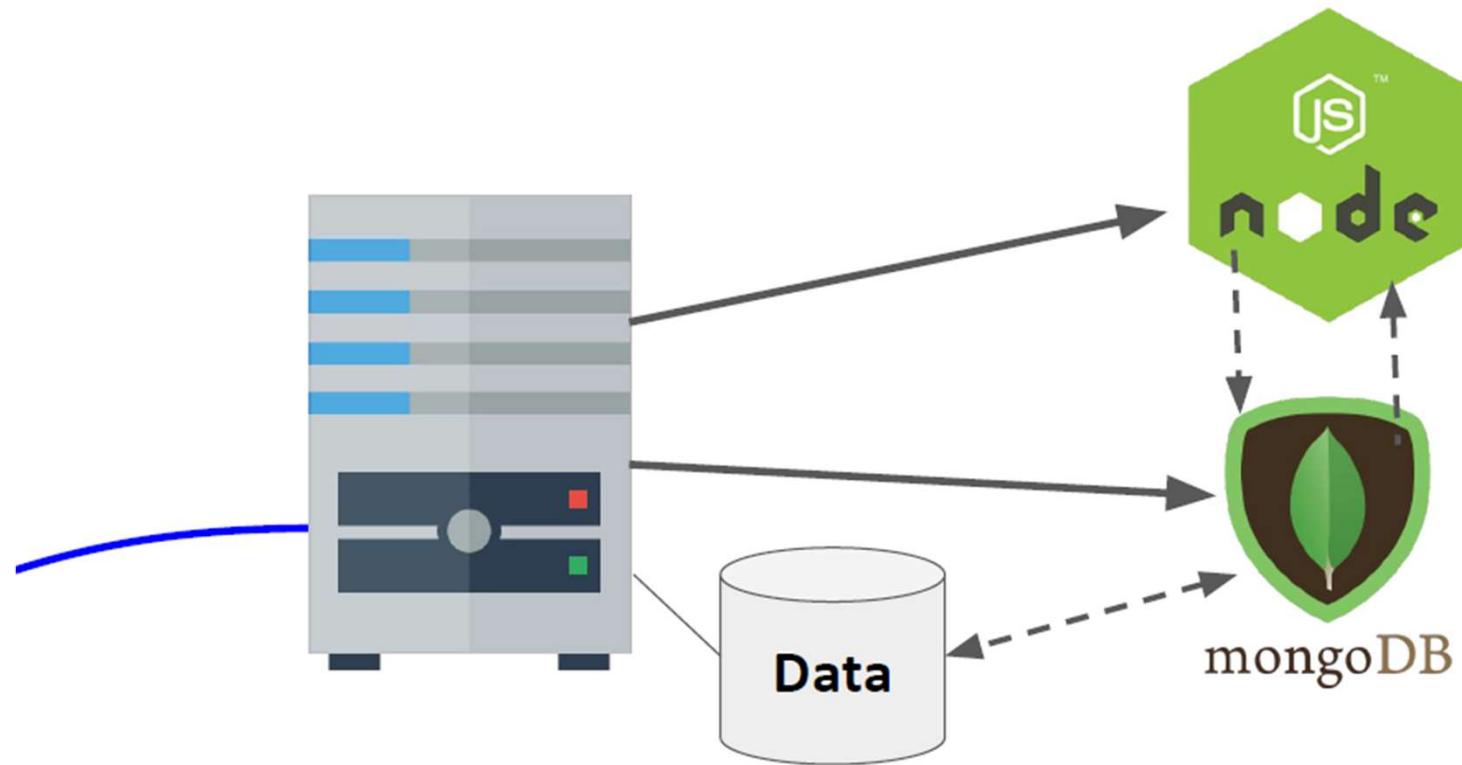
`req.params.userId;`

`req.params.carId;`

Przykładowe API dla kolekcji produkty

Zadanie	Metoda	Ścieżka	Przyjmuje	Zwraca
Pobieranie danych	GET	/produkty	nic	tablica obiektów
Tworzenie obiektu	POST	/produkty	pojedynczy obiekt	zapisany obiekt (albo błąd)
Pobieranie obiektu	GET	/produkty/<id>	nic	pojedynczy obiekt
Aktualizacja obiektu	PUT	/produkty/<id>	pojedynczy obiekt	zapisany obiekt (albo błąd)
Usuwanie obiektu	DELETE	/produkty/<id>	pojedynczy obiekt	nic

Czas na przechowywanie danych



Integracja z bazami danych

Jednym z istotnych aspektów tworzenia nowoczesnych aplikacji internetowych jest przechowywania i wymiana danych. Najczęściej dane które są używane w aplikacjach są przechowywane w bazach danych.

Node.js oraz Express.js posiadają interfejsy do najpopularniejszych typów baz danych:

- MySQL
- PostgreSQL
- Oracle
- MongoDB (noSQL)



Integracja z bazami danych



Wszystko zależy od typu problemu jaki chcemy rozwiązać w oparciu o bazy danych.

- **brak związków encji**
- **nie możliwe do wykorzystania w niektórych złożonych problemach.**
- **skalowalne,**
- **rozproszone,**
- **niski stopień złożoności**
- **elastyczność**

Integracja z bazami danych

Jedną z najszerzej wykorzystywanych silnikiem baz danych NoSQL w aplikacjach Node.js jest Mongo DB. Jest to aplikacja bazodanowa która bardzo dobrze sprawdza się w zastosowaniach “cloud” (w chmurze).



www.mongodb.org

MongoDB jest rozwiązaniem które można wykorzystywać na zasadzie otwartej licencji do wszystkich zastosowań.

W MongoDB dane przechowywane są w postaci dokumentów o bardzo elastycznej strukturze przypominającej format JSON:

```
...  
{  
    "name": "Jan",  
    "lname": "Kowalski",  
    "email": "jan@kowalski.pl",  
    "age": 21,  
    "groups": ["users", "mail", "root"]  
}  
...
```

Pojedynczy dokument
przechowujący dane

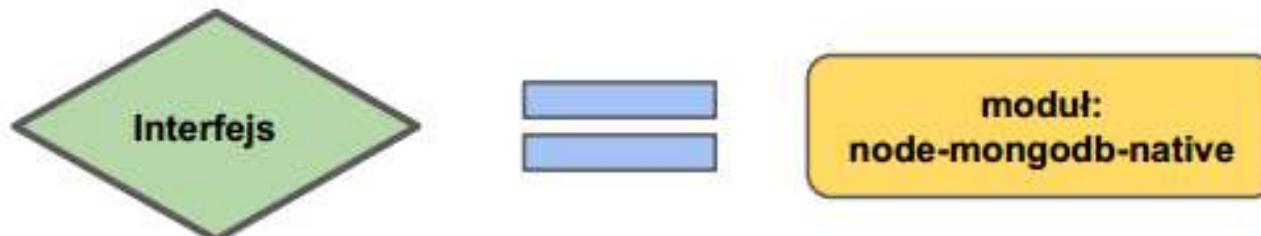
Integracja z bazami danych



&



Node.js posiada moduł stanowiący interfejs programistyczny do komunikacji z bazami opartymi o MongoDB potrafiący wykonać serializację i deserializację danych.



Instalacja pakietu za pomocą repozytorium NPM:

```
~/node> npm install mongodb
```

Pakiet dodajemy za pomocą metody require:

```
var mongodb = require('mongodb');
```

Integracja z bazami danych

Wykorzystując interfejs mongodb, musimy połączyć się z serwerem który obsługuje bazę danych:

```
var mongo = require('mongodb');
var serwer = new mongo.Server('127.0.0.1', 27017);
```

a następnie z samą bazą danych:

```
new monogodb.Db('nazwabazy',serwer).open(
  function(err,client){
    if (err) throw err;
    console.log('Polaczyles sie baza Mongo');
    var users = new mongo.Collection(client,'users');
  });

```

W przypadku kiedy chcemy połączyć się z bazą która nie istnieje zostanie ona automatycznie utworzona.

Integracja z bazami danych

W MongoDB dane przechowywane są w postaci kolekcji dokumentów:

```
{  
  "_id": 1,  
  "name": "Jan",  
  "lname": "Kowalski",  
  "email": "jan@kowalski.pl",  
  "age": 21,  
  "groups": ["users"]}  
  
{  
  "_id": 2,  
  "name": "Anna",  
  "lname": "Nowak",  
  "email": "anna@nowak.pl",  
  "age": 55,  
  "groups": ["users"]}  
  
{  
  "_id": 3,  
  "name": "Magdalena",  
  "lname": "Zauska",  
  "email": "magda@zauska.pl",  
  "age": 18,  
  "groups": ["users"]}
```

Mechanizm zapytania zwraca tylko dokumenty spełniające zadane w zapytaniu kryteria:

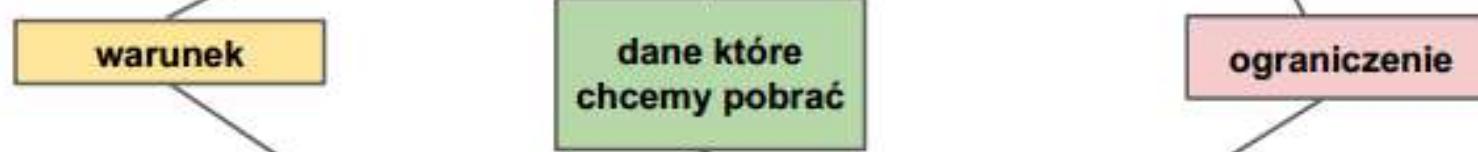
```
... .find( {age: {$gt: 20}}); ...
```

w przypadku tak zdefiniowanego zapytania interfejs powinien zwrócić tylko dokumenty oznaczone numerem 1 i 2.

Integracja z bazami danych

Zobaczmy jak działa funkcja do wykonania zapytania:

```
users.find( { "age": { $gt: 20 }, { "name": 1, "email": 1, "_id": 0 } } ).limit(5);
```



W klasycznym SQL zapytanie to miałyby postać:

```
SELECT name, email FROM user WHERE age>20 LIMIT 5;
```

Każdy dokument w MongoDB będący częścią kolekcji posiada unikalny identyfikator “`_id`”, zapisany za pomocą ciągu MD5. Gdy nie określmy w zapytaniu jednoznacznie że nie chcemy zwracać tych danych, będzie zwracany w sposób domyślny.

Integracja z bazami danych

Aby dodać rekord do istniejącej bazy musimy skorzystać z metody:

```
users.insert( {  
    "name": "Janina",  
    "lname": "Nowakowska",  
    "email": "jan@nowakowska.pl",  
    "age": 41  
}  
);
```

W klasycznym SQL zapytanie to miałyby postać:

```
INSERT INTO users (name, lname, email, age)  
VALUES (Janina, Nowakowska, jan@nowakowska.pl, 41);
```

W przypadku operacji insert tworzony jest nowy dokument w danej kolekcji i generowane jest unikalne “_id”.

Integracja z bazami danych

Uaktualnianie istniejących rekordów w bazie:

```
: users.update(  
:   { "age": $gt 18},  
:   { $set: { "email": "UNKNOWN" } },  
:   { multi: true}  
: );
```

W klasycznym SQL zapytanie to miałoby postać:

```
: UPDATE users SET email = "UNKNOWN" WHERE age > 18 ;
```

Operacja usuwania rekordów:

```
: users.remove(  
:   { "age": 18}  
: );  
  
: DELETE FROM users WHERE age = 18;
```

Integracja z bazami danych

Jak teraz odczytać dane z bazy danych ?



```
.....  
: router.get('/',function(req,res,next){  
    Projekty.findAll()  
        .success(function(projects){  
            res.render('projekty',{projekty:projekt});  
        })  
        .error(next);  
});  
.....
```

`.findAll()` - zwraca wszystkie rekordy z relacji Projekty

`.sucess()` - obsługuje zdarzenie w przypadku prawidłowej odpowiedzi z bazy danych

`.error()` - obsługuje zdarzenie w przypadku błędu

Integracja z bazami danych

Inne przydatne metody:



`.findById([id])` - metoda poszukuje rekordu o konkretnym identyfikatorze

`.findOne({where: { [warunki] } })` - metoda poszukuje rekordu spełniającego konkretne warunki podane jako obiekt

```
.....  
: Projekty.findOne({  
:   where: {  
:     tytul: 'projekt1'  
:   }  
: })  
:   .then()  
.....
```

```
SELECT *  
FROM `Projekty`  
WHERE (  
  `Projekty`.`tytul` = 'Projekt1')  
LIMIT 1;
```

Integracja z bazami danych

Inne przydatne metody:



`.findById([id])` - metoda poszukuje rekordu o konkretnym identyfikatorze

`.findOne({where: { [warunki] } })` - metoda poszukuje rekordu spełniającego konkretne warunki podane jako obiekt

```
Projekty.findOne({  
  where: {  
    tytul: 'Projekt1',  
    id: {  
      $or: [  
        [1,2,3],  
        { $gt: 10 }  
      ]  
    }  
  }  
})
```

```
SELECT *  
FROM `Projekty`  
WHERE (  
  `Projekty`.`tytul` = 'Projekt1'  
  AND (`Projekty`.`id` IN (1,2,3) OR  
  `Projekty`.`id` > 10)  
)  
LIMIT 1;
```

Integracja z bazami danych

Usuwanie rekordów z bazy danych DELETE



`.destroy()` - usuwa rekord z bazy danych spełniający określone warunki

```
Projekt.findById(4)
    .success(function(proj) {
        proj.destroy()
            .success( )
            .error( )
    })
    .error( );
```

```
DELETE FROM `Projekt`
WHERE `id` = 4
```

mongoose

elegant `mongodb` object modeling for `node.js`

<http://mongoosejs.com/>

Motywacja

- tworzenie walidacji dla MongoDB, rzutowania i szablonów logiki biznesowej jest dużym obciążeniem ...
- dlatego używamy gotowych bibliotek, np. mongoose.

Mongoose

- biblioteka dla Node.js,
- udostępnia mapowanie obiektowe (Object Data Mapping, ODM (ORM))
 - znany z Node.js interfejs,
 - zamiana obiektów z bazy danych do JavaScript i odwrotnie.

mongoose - sposób działania

- Instalujemy mongoose w projekcie: `npm install mongoose --save`
- Dołączamy mongoose do projektu i łączymy się z bazą danych test (zostanie utworzona jeśli takiej nie było):

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/test');
```

- Sprawdzamy czy połączenie się udało:

```
var db = mongoose.connection;
db.on('error', console.error.bind(console, 'błąd połączenia...'));
db.once('open', function() {
  // połączenie udane!
});
```

- W mongoose wszystko zaczyna się od **schematu**:

```
var friendSchema = mongoose.Schema({
  nazwa: String
});
```

mongoose - sposób działania

- Mając schemat, na jego bazie tworzymy **model**

```
var Friend = mongoose.model('Friend', friendSchema);
```

- Na bazie modelu tworzymy dokumenty (zawierają pola i typy jak w schemacie):

```
var franek = new Friend({ nazwa: 'Franek' });
console.log(franek.nazwa); // 'Franek'
```

- Przyjaciele mogą się witać - zobaczymy, jak dodać funkcjonalność do naszych dokumentów:

```
// metody należy dodać do schematu ZANIM utworzy się z niego model
friendSchema.methods.sayHello = function () {
  var powitanie = this.nazwa
    ? "Cześć, mam na imię " + this.nazwa
    : "Witaj, nie wiem jak się nazywam ...";
  console.log(powitanie);
}
```

Schematy w mongoose

Schemat odpowiada kolekcji w MongoDB, określa przechowywane dokumenty.

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var blogSchema = new Schema({
  title: String,
  author: String,
  body: String,
  comments: [{ body: String, date: Date }],
  date: { type: Date, default: Date.now },
  hidden: Boolean,
  meta: {
    votes: Number,
    favs: Number
  }
});|
```

- Klucz określa pole dokumentu, wartość określa typ pola dokumentu.
- Pole tytuł będzie typu String.
- Kluczom można przypisywać obiekty zagnieżdżone, które zawierają kolejne definicje klucz:typ, np. pole meta.

Utworzenie modelu

Mając utworzony schemat:

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var blogSchema = new Schema({
  title: String,
  author: String,
  body: String,
  comments: [{ body: String, date: Date }],
  date: { type: Date, default: Date.now },
  hidden: Boolean,
  meta: {
    votes: Number,
    favs: Number
  }
});
```

tworzymy na jego bazie model:

```
var Blog = mongoose.model('Blog', blogSchema);
// wszystko gotowe, można pracować na bazie danych!
```

mongoose - utworzenie modelu i schematu

- Mając model Osoba
- tworzymy nowy obiekt na bazie modelu (**new**) i zapisujemy go (**save**):

```
var Osoba = mongoose.model('nazwaBezZnaczenia', osobaSchema);

var rob = new Osoba({imie:'Robert', nazwisko:'Lewandowski', wiek:27});

// zapis do bazy
rob.save(function (err) {
  if (err) {
    console.log(err);
  } else {
    console.log('zapisano');
  }
});
```

mongoose - zapis do bazy, create

- **create** - skrót do utworzenia nowego dokumentu, który jest automatycznie zapisywany w bazie jeśli jest prawidłowy;
- można przekazywać jeden dokument, listę dokumentów oddzielonych przecinkiem albo całą tablicę:

```
Osoba.create({ imie: 'Jan', nazwisko: "Nowak" }, function (err, os) {  
  if (err) return handleError(err);  
  console.log(JSON.stringify(os));  
})
```

```
Osoba.create( { imie: 'Franek', nazwisko: "Marchewka" },  
             { imie: 'Zofia', nazwisko: "Głęb" }, function (err, os) {  
  if (err) return handleError(err);  
  console.log(JSON.stringify(os)); // wyświetli tylko ostatni zapisany dokument  
})
```

```
Osoba.create( [ { imie: 'Jola' }, { imie: 'Józefa' },  
               { imie:"Adam"} ], function (err, os) {  
  if (err) return handleError(err);  
  console.log(JSON.stringify(os)); // wyświetli wszystkie zapisane dokumenty  
})
```

mongoose - zapytania

- dostępna jest bogata składnia zapytań z MongoDB:
- find(), findById(), findOne() i where()

```
Osoba.find(function (err, osoby) { // wszystkie osoby z kolekcji
  if (err) return next(err);
  console.log(JSON.stringify(osoby));
});
```

```
Osoba.find( {imie:new RegExp('an', 'i')} ,function (err, osoby) {
  if (err) return next(err);
  console.log(JSON.stringify(osoby)); // osoby z imieniem zaw. 'an'
});
```

```
Osoba.findOne( {nazwisko:new RegExp('k','i')},
               wiek: {$gte: 25} } ,function (err, osoby) {
  if (err) return next(err);
  console.log(JSON.stringify(osoby)); // tylko pierwsza os. spełniająca warunki
});
```

```
Osoba.findById( "570929216e8a2a5f23863676" ,function (err, osoba) {
  console.log(JSON.stringify(osoba)); // osoba z podanym id
});
```

mongoose - usuwanie dokumentów

- **remove(warunki, [funZw])** - usuwa wszystkie dokumenty spełniające podane warunki
- **findByIdAndRemove(id, [opcje], [funZw])** - usuwa dokument o podanym id, zwraca usuwany dokument przez funZw
- **findOneAndRemove(warunki, [opcje], [funZw])** - usuwa pierwszy z dokumentów spełniających podane warunki, zwraca usuwany dokument przez funZw

```
Osoba.remove( { imie : 'Józefa' } ,function (err, osoba) {  
  // {"ok":1,"n":1} - liczba usuniętych dokumentów  
  console.log(JSON.stringify(osoba));  
});
```

```
Osoba.findByIdAndRemove( "57092733d5073d6d22307d4c", function (err, osoba) {  
  console.log(JSON.stringify(osoba)); // wyświetla usuwany dokument  
});
```

```
Osoba.findOneAndRemove( {nazwisko:{$exists:false} } , function (err, osoba) {  
  console.log(JSON.stringify(osoba)); // wyświetla usunięty dokument  
});
```

mongoose - aktualizacja dokumentów

- Są różne sposoby na aktualizację danych.
- Najprościej zmienić pobrany dokument i ponownie go zapisać:

```
Osoba.findById( "57092733d5073d6d22307d4a" ,function (err, osoba) {  
  osoba.nazwisko = 'Dąbek';  
  osoba.save(function (err) {  
    if (err) return handleError(err);  
    console.log(JSON.stringify(osoba));  
  });  
});
```

- **update(warunki, aktualizacja, [opcje], [funZw])** - aktualizuje wiele dokumentów, wszystkie, które spełniają warunki, nie zwraca zmienionych dokumentów

```
Osoba.update( { _id:"57092733d5073d6d22307d4a" },  
  { $set: {wiek: 18} }, function (err, osoba) {  
  // {"ok":1,"n":1} - liczba zaktualizowanych dokumentów  
  console.log(JSON.stringify(osoba));  
});
```

mongoose - aktualizacja dokumentów

- **findByIdAndUpdate(id, [aktualizacja], [opcje], [funZw])** - aktualizuje dokument o podanym id, zwraca usuwany dokument przez funZw
- **findOneAndUpdate([warunki], [aktualizacja], [opcje], [funZw])** aktualizuje pierwszy z dokumentów spełniających podane warunki, zwraca usuwany dokument przez funZw

```
Osoba.findByIdAndUpdate( "57092733d5073d6d22307d4a",
    { $set: { wiek: 28 } }, function (err, osoba) {
  console.log(JSON.stringify(osoba)); // wyświetla aktualizowany dokument
});
```

```
Osoba.findOneAndUpdate( { nazwisko: { $exists:false } },
    { $set: { nazwisko:"Rał" }}, function (err, osoba) {
  if (err) return handleError(err);
  console.log(JSON.stringify(osoba)); // wyświetla aktualizowany dokument
});
```

Proste API RESTful - całość aplikacji

api.js

```
var express = require('express');
var mongoose = require('mongoose');
var bodyParser = require('body-parser');
var app = express();

// parse application/x-www-form-urlencoded
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json()); // parse application/json

// ... schemat i model
// ... obsługa API

app.get('/', function(req, res) {
  res.end("Witam Was serdecznie na mojej stronie :)");
})

app.listen(5000);
```

Proste API RESTful - schemat i model

- Schemat zawiera 5 pól, 2 zawierają wartości domyślne.
- Model tworzy kolekcję o nazwie tasks.
- Obiekt Zadanie pozwala na dostęp do bazy danych.

api.js

```
var zadSchema = new mongoose.Schema({  
    nazwa: String,  
    zakonczone: Boolean,  
    notatka: String,  
    rodzaj: { type: String, default: "praca" },  
    utworzone: { type: Date, default: Date.now }  
});  
  
var Zadanie = mongoose.model('task', zadSchema);
```

Proste API RESTful - lista dokumentów

- wykorzystanie metody GET dla wyświetlenia danych
- wykorzystanie metody find() dla odczytu wszystkich danych
- dane zwrócone w postaci JSONa

api.js

```
/* GET /zadania - lista zadań */
app.get('/zadania', function(req, res, next) {
  Zadanie.find(function (err, zadania) {
    if (err) return next(err);
    res.json(zadania);
  });
});
```

Proste API RESTful - dodawanie dokumentu

- wykorzystanie metody POST dla utworzenia danych
- wykorzystanie metody create() - tworzy i od razu zapisuje dane
- dane z zapytania pobrane dzięki warstwie body-parser
- dane zwrócone w postaci JSONa

api.js

```
/* POST /zadania - dodawanie zadania */
app.post('/zadania', function(req, res, next) {
    Zadanie.create(req.body, function (err, zad) {
        if (err) return next(err);
        res.json(zad);
    });
});
```

Proste API RESTful - tylko wybrany dokument

- wykorzystanie metody GET dla wyświetlenia danych
- wykorzystanie metody findById() dla odczytu jednego dokumentu
- parametr id określa zwracany dokument
- dane zwrócone w postaci JSONa

api.js

```
/* GET /zadania/id - wyświetlenie wybranego zadania */
app.get('/zadania/:id', function(req, res, next) {
  Zadanie.findById(req.params.id, function (err, zad) {
    if (err) return next(err);
    res.json(zad);
  });
});
```

Proste API RESTful - aktualizacja dokumentu

- wykorzystanie metody PUT dla aktualizacji danych
- wykorzystanie metody `findByIdAndUpdate()` do aktualizacji wybranego dokumentu
- parametr `id` określa wybrany (i zwracany) dokument
- parametr `req.body` odpowiada za opcje, np. jakie pola dokumentu mają być zwrócone: `{ "select": { "nazwa":true } }`
- dane zwrócone w postaci JSONa

api.js

```
/* PUT /zadania/:id - aktualizacja zadania */
app.put('/zadania/:id', function(req, res, next) {
  Zadanie.findByIdAndUpdate(req.params.id,
                            req.body, function (err, zad) {
    if (err) return next(err);
    res.json(zad);
  });
});
```

Proste API RESTful - usuwanie dokumentu

- wykorzystanie metody DELETE dla usuwania danych
- wykorzystanie metody findByIdAndRemove() do usunięcia wybranego dokumentu
- parametr id określa usuwany (i zwracany) dokument
- parametr req.body odpowiada za opcje, np. jakie pola dokumentu mają być zwrócone: { "select": { "nazwa":true } }
- dane zwrócone w postaci JSONa

api.js

```
/* DELETE /zadania/:id - usuwanie zadania */
app.delete('/zadania/:id', function(req, res, next) {
  Zadanie.findByIdAndRemove(req.params.id,
    req.body, function (err, zad) {
      if (err) return next(err);
      res.json(zad);
    });
});
```

Connecting a MySQL database to a Node.js app

Several **NodeJS libraries** allow you to establish a connection with a MySQL database. Of these, the two most popular are :

- ✓ **mysql**, a basic MySQL driver for Node.js written in javascript and not requiring compilation. It is the simplest and fastest solution to set up to interact with a MySQL database in Node.
- ✓ **Sequelize**, the most popular library for using SQL-based database management systems with Node.js. It supports MySQL but also Postgres, Microsoft SQL, MariaDB... This powerful ORM (*Object-Relational Mapping*) allows, among other things, the use of promises and the customisation of error messages for each field.

Mysql installation and connection

To install the Mysql module, type the command:

```
npm install mysql
```

To initialise the module, we import it into the entry point file:

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

Your NodeJS app can now **connect to your MySQL database.**

Configure & Connect to MySQL database

```
module.exports = {  
  HOST: "localhost",  
  USER: "GR",  
  PASSWORD: "123456",  
  DB: "testdb"  
};  
  
const mysql = require("mysql");  
const dbConfig = require("../config/db.config.js");  
// Create a connection to the database  
const connection = mysql.createConnection({ host:  
  dbConfig.HOST, user: dbConfig.USER, password:  
  dbConfig.PASSWORD, database: dbConfig.DB });  
  
// open the MySQL connection  
connection.connect(error => {  
  if (error) throw error;  
  console.log("Successfully connected to the database.");  
});  
  
module.exports = connection;
```

Let's create *config* folder in the *app* folder, under application root folder, then create *db.config.js* file inside that *config* folder with content like this:

Now create a database connection that uses configuration above. The file for connection is *db.js*, we put it in *app/models* folder

Now the user is connected to MySQL db and can execute requests.

Running SQL queries on a MySQL database with NodeJS

```
con. connect(function(err) {  
  if (err) throw err;  
  console. log("Connected to MySQL database!");  
  con. query("SELECT students.id as 'student_id', students.name as 'student_name',  
  students.course_id, course.name as 'course_name', course.date as 'course_date' FROM  
  students JOIN course on students.course_id = course.id", function (err, result) {  
    if (err) throw err;  
    console. log(result);  
  });  
});
```

Using a MySQL database in NodeJS with Sequelize

To use **Sequelize** to interact with your MySQL database in Node, you must first install the **mysql2 driver**. This is a separate driver from the mysql module, and is less popular than the mysql module, but offers some additional functionality.

- ✓ Install mysql2 with npm install :

```
npm install mysql2
```

- ✓ To install Sequelize, type:

Npm install sequelize

- ✓ In your app code, initialize **Sequelize** with **require()** :

```
const { Sequelize } = require('sequelize');
```

Configure MySQL database & Sequelize

```
module.exports = {  
  HOST: "localhost",  
  USER: "root",  
  PASSWORD: "123456",  
  DB: "testdb",  
  dialect: "mysql",  
  pool: { max: 5, min: 0, acquire: 30000, idle: 10000 } };
```

First five parameters are for MySQL connection.

pool is optional, it will be used for Sequelize connection pool configuration:

max: maximum number of connection in pool

min: minimum number of connection in pool

idle: maximum time, in milliseconds, that a connection can be idle before being released

acquire: maximum time, in milliseconds, that pool will try to get connection before throwing error

Configure MySQL database & Sequelize

Now create **app/models/index.js**

```
const dbConfig = require("../config/db.config.js");
const Sequelize = require("sequelize");

const sequelize = new Sequelize(dbConfig.DB, dbConfig.USER, dbConfig.PASSWORD, {
  host: dbConfig.HOST,
  dialect: dbConfig.dialect,
  operatorsAliases: false,
  pool: { max: dbConfig.pool.max, min: dbConfig.pool.min, acquire:
dbConfig.pool.acquire, idle: dbConfig.pool.idle }
});

const db = {};

db.Sequelize = Sequelize;
db.sequelize = sequelize;
db.students = require("./student.model.js")(sequelize, Sequelize);

module.exports = db;
```

Define the Sequelize Model

```
module.exports = (sequelize, Sequelize) => {  
  
  const Student = sequelize.define("student", {  
    name: {  
      type: Sequelize.STRING  
    },  
    age: {  
      type: Sequelize.INTEGER  
    },  
    passed: {  
      type: Sequelize.BOOLEAN  
    }  
  });  
  
  return Student;  
};
```

This Sequelize Model represents **Students** table in MySQL database.
These columns will be generated automatically:
id, name, age, passed, createdAt, updatedAt.

Define the Sequelize Model

After initializing Sequelize, we don't need to write CRUD functions, Sequelize supports all of them:

- create a new Student: `create(object)`
- find a Student by id: `findByPk(id)`
- get all Students: `findAll()`
- update a Student by id: `update(data, where: { id: id })`
- remove a Student `destroy(where: { id: id })`
- remove all Students: `destroy(where: {})`
- find all Students by name: `findAll({ where: { name: ... } })`

These functions will be used in our Controller.

Define the controller

Inside **app/controllers** folder, let's create *student.controller.js* with CRUD functions

```
const db = require("../models");
const Student = db.student;
const Op = db.Sequelize.Op;

// Create and Save a new Student
exports.create = (req, res) => { };

// Retrieve all Students from the database.
exports.findAll = (req, res) => { };

// Find a single Student with an id
exports.findOne = (req, res) => { };

// Update a Student by the id in the request
exports.update = (req, res) => { };

// Delete a Student with the specified id in the request
exports.delete = (req, res) => { };
// Delete all Students from the database.
exports.deleteAll = (req, res) => { };
// Find all passed Students
exports.findAllPassed = (req, res) => { };
```

Implementation function in controller

Create a new object

```
exports.create = (req, res) => {
  // Validate request
  if (!req.body.name) {
    res.status(400).send({ message: "Content can not be empty!" });
    return;
  }
  // Create a Student
  const student = {
    name: req.body.name,
    age: req.body.age,
    passed: req.body.passsed ? req.body.passed : false
  };

  // Save Student in the database
  Student.create(student)
    .then(data => {
      res.send(data);
    })
    .catch(err => { res.status(500).send({ message: err.message || "Some error occurred while creating the Student." }); });
};
```

Implementation function in controller

Find a single Student with an id:

```
exports.findOne = (req, res) => {
  const id = req.params.id;

  Student.findPk(id)
    .then(data => {
      if (data) {
        res.send(data);
      }
      else {
        res.status(404).send({ message: `Cannot find Student with id=${id}.` });
      }
    })
    .catch(err => {
      res.status(500).send({ message: "Error retrieving Student with id=" + id});
    });
};
```

Implementation function in controller

Delete a Student with the specified id

```
exports.delete = (req, res) => {
  const id = req.params.id;

  Student.destroy({ where: { id: id } })
    .then(num => {
      if (num == 1) {
        res.send({ message: "Student was deleted successfully!" });
      }
      else {
        res.send({ message: `Cannot delete Student with id=${id}.` });
      }
    })
    .catch(err => {
      res.status(500).send({ message: "Could not delete Student with id=" + id });
    });
};
```



Websockety

Komunikacja w czasie rzeczywistym

Komunikacja jednostronna vs dwustronna



Socket.io dla realizacji WebSocket

```
$ npm install --save socket.io
```

```
var express = require('express');
var app = express();
var server = require('http').createServer(app);
var io = require('socket.io')(server);

io.on('connection', function(client) {
  console.log('Client connected...');
});

app.get('/', function (req, res) {
  res.sendFile(__dirname + '/index.html');
});

server.listen(8080);
```

app.js

Server do komunikacji

```
<script src="/socket.io/socket.io.js"></script>
```

```
<script>
```

```
  var socket = io.connect('http://localhost:8080');
```

index.html

klient w komunikacji

Wysłanie komunikatu do klienta

```
io.on('connection', function(client) {  
    console.log('Client connected...');  
  
    emit the 'messages' event on the client  
    client.emit('messages', { hello: 'world' });  
});
```

app.js

```
<script src="/socket.io/socket.io.js"></script>  
<script>  
    var socket = io.connect('http://localhost:8080');  
    socket.on('messages', function (data) {  
        alert(data.hello);  
    });  
    listen for 'messages' events  
</script>
```

index.html

Wysłanie komunikatu do serwera

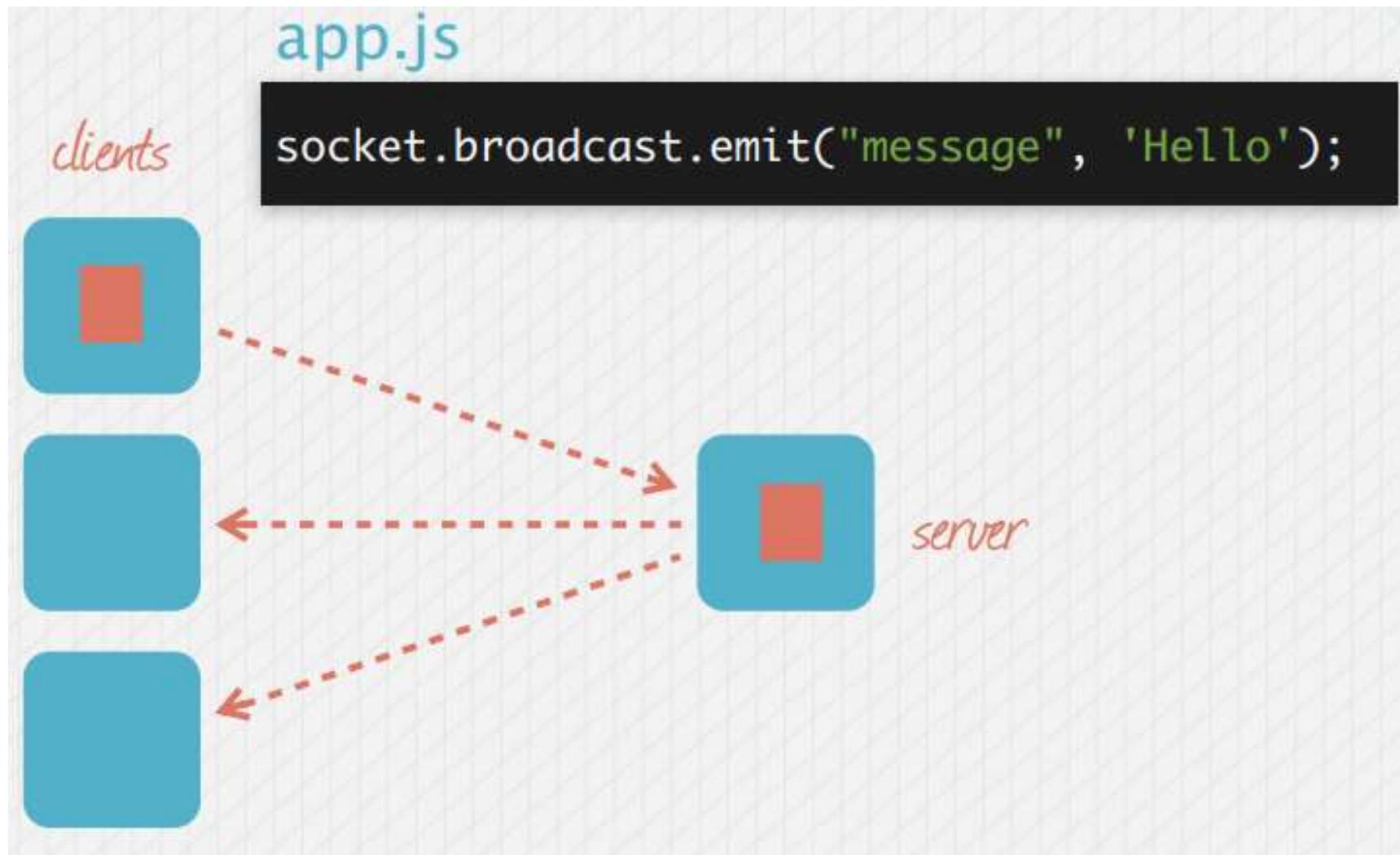
```
io.on('connection', function(client) {  
    client.on('messages', function (data) {  
        console.log(data);  
    });  
});
```

app.js

```
<script>  
    var socket = io.connect('http://localhost:8080');  
    $('#chat_form').submit(function(e){  
        var message = $('#chat_input').val();  
        emit the 'messages' event on the server  
        socket.emit('messages', message);  
    });  
</script>
```

index.html

Rozgłaszenie komunikatów



Rozgłaszenie komunikatów

```
io.on('connection', function(client) {  
    client.on('messages', function (data) {  
        client.broadcast.emit("messages", data);  
    });      broadcast message to all other clients connected  
});
```

app.js

```
<script>  
...  
socket.on('messages', function(data) { insertMessage(data) });  
</script>
```

index.html
insert message into the chat

Zapisywanie danych w sokecie

```
io.on('connection', function(client) {  
    client.on('join', function(name) {  
        client.nickname = name;    set the nickname associated  
        with this client  
    });  
});
```

app.js

```
<script>  
    var server = io.connect('http://localhost:8080');  
    server.on('connect', function(data) {  
        $('#status').html('Connected to chattr');  
        nickname = prompt("What is your nickname?");  
  
        server.emit('join', nickname);   notify the server of the  
        users nickname  
    });  
</script>
```

index.html