

# REST API

API - Application Programming Interface - czyli interfejs do wzajemnej komunikacji między aplikacjami

URL - Uniform Resource Locator

Protokół - HTTP / HTTPS - HTTPS - port TCP (80) , HTTPS (443)

REST - REpresentational State Transfer

REST API vs RESTful API - oznacza to samo

## Metody HTTP:

1. GET - Odczyt
2. POST - Zapis
3. PUT - Aktualizacja całości
4. PATCH - Aktualizacja części
5. DELETE - usuwanie
6. 1xx - informacyjne
7. 2xx - potwierdzenia
8. 3xx - przekierowania
9. 4xx - błąd klienta
10. 5xx - błąd serwera

Zwracane obiekty:

1. XML
2. JSON
3. Więcej → [org.springframework.org](http://org.springframework.org) → MediaType

## Idempotentność / Safe metody HTTP

Idempotentność - bezstanowość - niezależnie ile razy wywołamy dany endpoint, zawsze uzyskamy ten sam rezultat.

Metody idempotentne:

1. GET
2. PUT - niezależnie ile razy zmienimy całość, będzie ten sam rezultat
3. DELETE

Metody nie idempotentne:

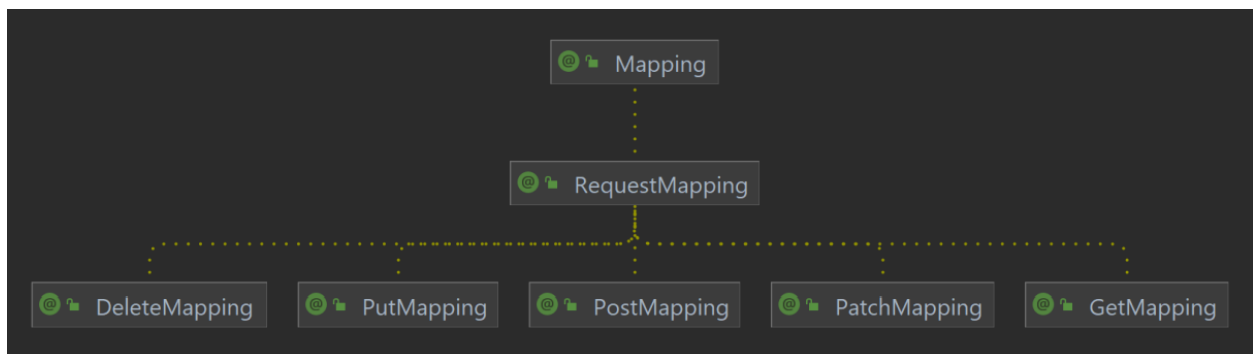
1. POST - za każdym razem tworzymy nowy obiekt

Zależy:

1. PATCH
2. W pliku konfiguracyjnym możemy ustalić port oraz domyślną ścieżkę naszego API:





```
server:  
  port: 8190  
  servlet:  
    context-path: /rootApiPath
```

## 2. Annotacje



RequestMapping - definiuje root path dla całej klasy

## 3. Annotacje mapujące parametry

 RequestHeader RequestParam PathVariable ResponseBody

1. RequestHeader - anotacja głównie służąca do przekazywania metadanych
2. RequestParam - definiowanie rzędania wraz z jego parametrami w formie key=value (najczęściej stosowane, jeżeli chcemy mieć jakieś sortowanie, paginacje itp.)
3. PathVariable - anotacja do wymagania parametru ścieżki - służy do definiowania uniarnych rekordów np. /books/{id} → /books/1

Kiedy RequestParam, a kiedy PathVariable →

<https://stackoverflow.com/questions/30967822/when-do-i-use-path-parameters-vs-query-parameters-in-a-restful-api>

4. RequestBody - Anotacja do przekazywania w ciele żądania, które później są mapowane do obiektów klasowych Java z Json.
5. **ModelAttribute** - pobiera parametry zapytania URL i danych formularzy na obiekt Java.

## Serializacja i deserializacja

1. Jackson - domyślna biblioteka używana przez Spring Boot służąca do serializacji / deserializacji obiektów (konwersja obiekt → json, json → object)

### ObjectMapper - klasa Jackson do serializacji / deserializacji

Jest to klasa która służy do zamiany obiektów Json na obiekty Java POJO

Często należy go nadpisać własną konfiguracją.

Przykładowa konfiguracja:

```
@Configuration
public class BeanConfiguration {
    @Bean
```

```

public ObjectMapper objectMapper() {
    return new ObjectMapper()
        .registerModule(new JavaTimeModule())
        .registerModule(new Jdk8Module())
        .configure(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS, false)
        .configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false)
        .setSerializationInclusion(JsonInclude.Include.NON_NULL); // (
    }
}

```

## Budowanie REST API

1. Annotacja `@RestController` - różnica pomiędzy `@Controller` jest taka, że `@RestController` nadaje każdemu endpointowi annoację `@ResponseBody`
2. Można zamiast tego stosować zwykły `@Controller`, ale trzeba potem nadawać annoację `@ResponseBody` nad endpointami

## ResponseBody i RequestBody

`@ResponseBody` - wiąże ciało zapytania HTTP z klasą Java (HTTP → Java Class)

`@RequestBody` - wiąże ciało odpowiedzi na żądanie HTTP z klasą

## Http Message Converters

Marshall Java Object - zamiana obiektu Java na wiadomość

Unmarshall Java Object - zamiana wiadomości na obiekt Java

Basic Http Message Converters -

- `StringHttpMessageConverter`
- `Jaxb2RootElementHttpMessageConverter`
- `MappingJackson2HttpMessageConverter`

Każdy `HttpMessageConverter` jest skojarzony z określonym MIME (Multipurpose Internet Mail Extensions)

Typ ten jest związany z negocjacją zawartości

Generalnie najważniejsze jest zrozumienie, że serwer może przyjmować różne MIME, oraz wysyłać różne MIME. Najczęściej będzie to jedak media type = application/json

## Nagłówki HTTP związane z media type

1. Accept - klient oznacza, jaki media type oczekuje i jest w stanie przetworzyć
2. Content-type - wskazuje typ nośnika zarówno w żądaniu jak i odpowiedzi.

## Zmiana MeduaType wysyłanego przez API

dodanie w argumencie anotacji mapującej (produces = MediaType.OTHER\_MEDIA\_TYPE)

możemy wybrać jaki typ wysyłamy

## Obsługa XML wymaga dodania zależności !

```
com.fasterxml.jackson.dataformat:jackson-dataformat-xml'
```

## Walidacja

Annotacje:

@Valid przed @ResuestBody + anotacje walidacyjne w klasie DTO:

1. @NotNull
2. @NotBlank
3. @NotEmpty
4. @Size
5. @Min
6. @Max
7. @Pattern
8. @Email

9. @Positive, @Negative
10. @Future , @Past
11. @AssertTrue, @AssertFalse

zastosowanie annocacji Valid powoduje, że podczas mapowania http request z body, na obiekt, sprawdzane są warunki podane w klasie DTO obiektu, na który występuje mapowanie. Jeżeli warunki nie zostaną spełnione, zostanie wyrzucony: `MethodArgumentNotValidException`, czyli domyślnie status 400 - Bad Request

## Przyadtnie parametry:

1. @RequestParam(required = true) , oraz (defaultValue="")

## Przykłady

### GetMapping

```
@GetMapping(value = EMPLOYEE_ID)
public EmployeeDTO showEmployeeDetails(@PathVariable Integer employeeId) {
    return employeeRepository.findById(employeeId)
        .map(employeeMapper::map)
        .orElseThrow(() -> new EntityNotFoundException(
            String.format("EmployeeEntity with id: {%s} not found ", employeeId)
        ))
}
```

### PostMapping

```
@PostMapping
public ResponseEntity<EmployeeDTO> addEmployee(@Valid @RequestBody EmployeeEntity newEmployee) {
    EmployeeEntity employeeEntity = EmployeeEntity.builder()
        .name(newEmployee.getName())
        .surname(newEmployee.getSurname())
        .salary(newEmployee.getSalary())
        .phone(newEmployee.getPhone())
        .email(newEmployee.getEmail())
        .build();
    employeeRepository.save(employeeEntity);
    return ResponseEntity.ok(employeeMapper.map(employeeEntity));
}
```

```

        .build();
        EmployeeEntity created = employeeRepository.save(newEmployee);
        return ResponseEntity
            .created(URI.create("/employees/%s".formatted(created.getEmplo
            .build());
    }

```

## PutMapping

```

@PutMapping(EMPLOYEE_ID)
public ResponseEntity<?> updateEmployee(
    @PathVariable Integer employeeId,
    @Valid @RequestBody EmployeeDTO employeeDTO
) {
    EmployeeEntity existingEmployee = employeeRepository.findById(
        employeeId
    ).orElseThrow(() -> new EntityNotFoundException(
        String.format("EmployeeEntity not found, employeeId: [%s]", employeeId)
    ));
    existingEmployee.setName(employeeDTO.getName());
    existingEmployee.setSurname(employeeDTO.getSurname());
    existingEmployee.setSalary(employeeDTO.getSalary());
    existingEmployee.setPhone(employeeDTO.getPhone());
    existingEmployee.setEmail(employeeDTO.getEmail());
    employeeRepository.save(existingEmployee);
    return ResponseEntity.ok().build();
}

```

## DeleteMapping

```

@RestController
@RequestMapping(EmployeesController.BASE_PATH)
@ControllerAdvice
30 |
public class EmployeesController {
    public static final String BASE_PATH = "/employees";
}

```

```

public static final String EMPLOYEE_ID = "{employeeId}";
public static final String EMPLOYEE_ID_RESULT = "%s";
private EmployeeRepository employeeRepository;
private EmployeeMapper employeeMapper;
// ...
@DeleteMapping(EMPLOYEE_ID)
public ResponseEntity<?> deleteEmployee(@PathVariable Integer
try {
employeeRepository.deleteById(employeeId);
return ResponseEntity.ok().build();
} catch (Exception e) {
return ResponseEntity.notFound().build();
}
}
}
}

```

## PatchMapping

```

@RestController
@RequestMapping(EmployeesController.BASE_PATH)
@AllArgsConstructor
public class EmployeesController {
    public static final String BASE_PATH = "/employees";
    // ...
    public static final String EMPLOYEE_UPDATE_SALARY = "{employeeId}/updateSalary";
    private EmployeeRepository employeeRepository;
    private EmployeeMapper employeeMapper;
    // ...
    @PatchMapping(EMPLOYEE_UPDATE_SALARY)
    public ResponseEntity<?> updateEmployeeSalary(
        @PathVariable Integer employeeId,
        @RequestParam(required = true) BigDecimal newSalary
    ) {
        EmployeeEntity existingEmployee = employeeRepository.findById(employeeId)
            .orElseThrow(() -> new EntityNotFoundException("Employee not found"));
    }
}

```



```

        String.format("EmployeeEntity not found, employeeId: [%s]", employeeId), existingEmployee.setSalary(newSalary);
        employeeRepository.save(existingEmployee);
        return ResponseEntity.ok().build();
    }
}

```

## Klasa ResponseEntity

Jest to reprezentacja odpowiedzi HTTP. Używamy jej do bardziej rozszerzonej odpowiedzi.

Przyjmuje on typ generyczny, czyli możemy

ex1 - podać do środka np. Stringa.

ex2 - Podać nagłówek (header) oraz status np.

1. ok()
2. created(URL location);
3. accepted();
4. badRequest();
5. notFound();
6. noContent();
7. chainowanie .body() np. ResponseEntity.badRequest().body("Invalid request data");

```

HttpHeaders headers = new HttpHeaders();
headers.add("Location", BASE_PATH + EMPLOYEE_ID_RESULT.format(employeeId));
return new ResponseEntity<>(
    headers,
    HttpStatus.CREATED
);

```

Jeżeli chcemy zwrócić konkretny status, to możemy użyć:

status(HttpStatus status) lub status(int status)

## Obsługa HTTP Error - @RestControllerAdvice i @ExceptionHandler

```
@Slf4j
@RestControllerAdvice
@Order(Ordered.HIGHEST_PRECEDENCE)
public class GlobalExceptionHandler extends ResponseEntityExceptionHandler {
    private static final Map<Class<?>, HttpStatus> EXCEPTION_STATUSES = Map.of(
        ConstraintViolationException.class, HttpStatus.BAD_REQUEST,
        EntityNotFoundException.class, HttpStatus.NOT_FOUND
    );

    @Override
    protected ResponseEntity<Object> handleExceptionInternal(
        @NonNull Exception exception,
        @Nullable Object body,
        @NonNull HttpHeaders headers,
        @NonNull HttpStatusCode statusCode,
        @NonNull WebRequest request
    ) {
        final String errorId = UUID.randomUUID().toString();
        log.error("Exception: ID={}, HttpStatus={}", errorId, statusCode);
        return super.handleExceptionInternal(
            exception,
            ErrorMessage.of(errorId), headers, statusCode, request
        );
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<Object> handle(final Exception exception) {
        return doHandle(exception, getHttpStatusFromException(exception));
    }
}
```

```

private ResponseEntity<Object> doHandle(final Exception exception,
final String errorId = UUID.randomUUID().toString());
log.error("Exception: ID={}, HttpStatus={}", errorId, status,
return ResponseEntity
.status(status)
.contentType(MediaType.APPLICATION_JSON)
.body(ExceptionMessage.of(errorId));
}
public HttpStatus getHttpStatusFromException(final Class<?> exceptionClass) {
return EXCEPTION_STATUS.getOrDefault(exceptionClass, HttpStatus.INTERNAL_SERVER_ERROR);
}

```

## Obsługa nagłówków (Headers)

Do tego służy @RequestHeader

```

@RequestHeader(value = HttpHeaders.ACCEPT) MediaType accept,
@RequestHeader(value = "httpStatus", required = true) int httpStatus

```

## Popularne narzędzia do pracy z API

1. Postaman
2. IntelliJ Idea HTTP Client

## Popularne biblioteki do konsumpcji API

1. **WebClient** (ulepszona wersja RestTemplate) - Please, consider using the org.springframework.web.reactive.client.WebClient which has a more modern API and supports sync, async, and streaming scenarios. WebClient obsługuje operacje synchroniczne i asynchroniczne, dlatego jest zalecane niż RestTemplate.
2. Feign
3. Rest Assured

#### 4. OkHttp

### Przykładowe API :

<https://petstore3.swagger.io/api/v3/>

### Różnica pomiędzy operacją synchroniczną, a asynchroniczną

Operacja synchroniczna czeka na zwrócenie wartości przez API, i dopiero po pobraniu może kontynuować działanie. Natomiast operacje asynchroniczne nie czekają na pobranie danych z API i kontynuują wykonywanie programu

### Konfiguracja WebClient

Do jego użycia należy dodać zależność: spring-boot-starter-webflux (Spring reactive Web)

Następnie należy dodać klasę konfiguracyjną np. WebClientConfiguration

```
@Configuration
public class WebClientConfiguration {
    private static final String BASE_URL = "https://petstore3.swagger.io/api/v3/";
    public static final int TIMEOUT = 5000;
    @Bean
    public WebClient webClient(final ObjectMapper objectMapper) {
        final var httpClient = HttpClient.create()
            .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, TIMEOUT)
            .responseTimeout(Duration.ofMillis(TIMEOUT))
            .doOnConnected(conn ->
                conn.addHandlerLast(new ReadTimeoutHandler(TIMEOUT, TimeUnit.MILLISECONDS))
                .addHandlerLast(new WriteTimeoutHandler(TIMEOUT, TimeUnit.MILLISECONDS))
            )
            .final var exchangeStrategies = ExchangeStrategies.builder()
                .codecs(configurer -> {
```

```

    configurer
    .defaultCodecs()
    .jackson2JsonEncoder(
    new Jackson2JsonEncoder(
    objectMapper,
    MediaType.APPLICATION_JSON));
    configurer
    .defaultCodecs()
    .jackson2JsonDecoder(
    new Jackson2JsonDecoder(
    objectMapper,
    MediaType.APPLICATION_JSON));
    }).build();
    return WebClient.builder()
    .baseUrl(BASE_URL)
    .exchangeStrategies(exchangeStrategies)
    .clientConnector(new ReactorClientHttpConnector(httpClient))
    .build();
    }
}

```

W powyższym Beanie konfigurujemy:

1. **Timeout** to pojęcie, które oznacza maksymalny czas, w którym system lub aplikacja oczekuje na zakończenie jakiejś operacji, np. odpowiedzi z serwera, wykonania zadania, nawiązania połączenia
2. Sekcja ExchangeStrategies - podajemy naszą konfigurację ObjectMappera to mapowania odpowiedzi serwera na nasz obiekt
3. Sekcja 3 - Tworzymy instancję WebClient. HttpClient jest klasą wykorzystywana przez WebClient do odbierania danych z API

Następnie należy stworzyć klasę, do której wstrzykujemy WebClient'a

```
@Component
```

```

@AllArgsConstructor
public class PetClientImpl implements PetDAO {
    private final WebClient webClient;
    @Override
    public Optional<Pet> getPet(final Long petId) {
        try {
            Pet result = webClient
                .get()
                .uri("/pet/" + petId)
                .retrieve()
                .bodyToMono(Pet.class)
                .block();
            return Optional.ofNullable(result);
        } catch (Exception e) {
            return Optional.empty();
        }
    }
}

```

## Swagger (implementacja OpenAPI)

Swagger implementuje **OpenAPI**, czyli specyfikacja, która definiuje format RESTful API, który ujednolici dokumentację. Dokumenty OpenAPI zazwyczaj są w formacie Json lub Yaml.

Swagger to zestaw narzędzi, który opiera się na specyfikacji OpenAPI. Jest on używany do projektowania, budowania, dokumentowania i testowania API.

Swagger posiada zestaw narzędzi:

- **Swagger Editor** – umożliwia pisanie specyfikacji OpenAPI w formacie JSON lub YAML i podgląd wygenerowanej dokumentacji w czasie rzeczywistym.
- **Swagger UI** – generuje interaktywną, webową dokumentację API, która umożliwia testowanie operacji bezpośrednio z poziomu przeglądarki.

- **Swagger Codegen** – generuje kod kliencki lub serwerowy w oparciu o specyfikację OpenAPI.
- **SwaggerHub** – platforma do współpracy, projektowania i dokumentowania API oparta na OpenAPI.

Więcej tutaj: <https://swagger.io/blog/api-strategy/difference-between-swagger-and-openapi/>

Dokumentacja dla Spring 3.x - <https://springdoc.org/>

(zależność gradle: implementation 'org.springdoc:springdoc-openapi-starter-webmvc-ui:2.0.0')

## Podstawowa konfiguracja Swagger'a

```
@Configuration
public class SpringDocConfiguration {
    @Bean
    public GroupedOpenApi groupedOpenApi() {
        return GroupedOpenApi.builder()
            .group("default")
            .pathsToMatch("/**")
            .packagesToScan(SpringRestExampleApplication.class.getPackage()
            .build());
    }
    @Bean
    public OpenAPI springDocOpenApi() {
        return new OpenAPI()
            .components(new Components())
            .info(new Info()
            .title("Employee application")
            .contact(contact())
            .version("1.0"));
    }
}
```

```

private Contact contact() {
    return new Contact()
        .name("Michał Bialek")
        .url("www.michal-bialek.com")
        .email("kontakt.michalbialek@gmail.com");
}
}

```

GroupedOpenApi - służy do tworzenie różnych zestawów endpointów.

W naszym przypadku

- **group("default")** – definiuje nazwę grupy API jako "default".
- **pathsToMatch("/")\*\*** – określa, że wszystkie ścieżki API w aplikacji będą pasować do tej grupy (symbol `/**` oznacza wszystkie ścieżki).
- **packagesToScan(SpringRestExampleApplication.class.getPackageName())** – wskazuje, że wszystkie klasy znajdujące się w pakiecie głównym aplikacji (tam, gdzie znajduje się `SpringRestExampleApplication`) mają być brane pod uwagę przy generowaniu dokumentacji API.

OpenAPI

W nim podajemy informacje odnośnie API - tytuł, kontakt, wersje itp.

Contact

W nim definiujemy nasze dane kontaktowe jako twórców API

Po dodaniu zależności i konfiguracji, możemy wejść na `localhost:8190/RootEndpoint/swagger-ui/index.html`, utworzy to nam dokumentację API

Jeżeli dostaniemy od kogoś API, możemy wejść na stronę edytora i odczytać endpointy jako GUI z JSON'a i XML'a - <https://editor.swagger.io/>

Endpointy zostały wychwycone przez Spring, lecz jeżeli chcemy uzyskać dodatkowy opis, możemy skorzystać z specjalnych anotacji Swagger'a dostępnych pod linkiem: <https://github.com/swagger-api/swagger-core/wiki/Annotations>



## Quick Annotation Overview

Name	Description
<a href="#">@Api</a>	Marks a class as a Swagger resource.
<a href="#">@ApiImplicitParam</a>	Represents a single parameter in an API Operation.
<a href="#">@ApiImplicitParams</a>	A wrapper to allow a list of multiple ApiImplicitParam objects.
<a href="#">@ApiModel</a>	Provides additional information about Swagger models.
<a href="#">@ApiModelProperty</a>	Adds and manipulates data of a model property.
<a href="#">@ApiOperation</a>	Describes an operation or typically a HTTP method against a specific path.
<a href="#">@ApiParam</a>	Adds additional meta-data for operation parameters.
<a href="#">@ApiResponse</a>	Describes a possible response of an operation.
<a href="#">@ApiResponses</a>	A wrapper to allow a list of multiple ApiResponse objects.
<a href="#">@Authorization</a>	Declares an authorization scheme to be used on a resource or an operation.
<a href="#">@AuthorizationScope</a>	Describes an OAuth2 authorization scope.

## Swagger 2 to Swagger 3 annotations

Swagger 3 is an updated version of Swagger 2 and has some changes in annotations:

- `@Api` → `@Tag`
- `@ApiIgnore` → `@Parameter(hidden = true)` or `@Operation(hidden = true)` or `@Hidden`
- `@ApiImplicitParam` → `@Parameter`
- `@ApiImplicitParams` → `@Parameters`
- `@ApiModel` → `@Schema`
- `@ApiModelProperty(hidden = true)` → `@Schema(accessMode = READ_ONLY)`
- `@ApiModelProperty` → `@Schema`
- `@ApiOperation(value = "foo", notes = "bar")` → `@Operation(summary = "foo", description = "bar")`
- `@ApiParam` → `@Parameter`
- `@ApiResponse(code = 404, message = "foo")` → `@ApiResponse(responseCode = "404", description = "foo")`

# Implementacja API poprzez specyfikację OpenAPI

1. W folderze: src/main/resources dodaj plik petstorev3.0.2.json z całą zawartością API w postaci JSON/XML

np. <https://petstore3.swagger.io/api/v3/openapi.json>

2. Dodanie zależności + konfiguracja gradle

```
plugins {  
    id 'org.springframework.boot' version "${springBootVersion}"  
    id 'io.spring.dependency-management' version "${springBootDep  
    id "org.openapi.generator" version "${openapiGeneratorVersion}"  
    id 'java'  
}  
// ...  
dependencies {  
    implementation 'org.springframework.boot:spring-boot-starter-  
    implementation 'org.springframework.boot:spring-boot-starter-  
    implementation 'org.springframework.boot:spring-boot-starter-  
    implementation 'org.springframework.boot:spring-boot-starter-  
    implementation 'org.flywaydb:flyway-core'  
    runtimeOnly 'org.postgresql:postgresql'  
    implementation "org.mapstruct:mapstruct:${mapstructVersion}"  
    annotationProcessor "org.mapstruct:mapstruct-processor:${mapstructVersion}"  
    compileOnly 'org.projectlombok:lombok'  
    annotationProcessor 'org.projectlombok:lombok'  
    annotationProcessor "org.projectlombok:lombok-mapstruct-binding:0.2.0"  
    implementation "org.openapitools:jackson-databind-nullable:${openapiVersion}"  
    implementation "javax.annotation:javax.annotation-api:${javax.annotation-api-version}"  
    implementation "io.swagger:swagger-annotations:${swaggerAnnotationsVersion}"  
    implementation "com.google.code.findbugs:jsr305:${jsr305Version}"  
    implementation "org.springdoc:springdoc-openapi-starter-webmvc-ui:${springdocVersion}"  
    testImplementation 'org.springframework.boot:spring-boot-starter-test'  
}
```

```

openApiGenerate {
    generatorName = "java"
    library = "webclient"
    configOptions = [
        serializableModel : "true",
        dateLibrary : "java8",
        serializationLibrary: "jackson"
    ]
    outputDir = "$buildDir/generated-sources/openapi".toString()
    inputSpec = "$rootDir/src/main/resources/petstore-v3.0.2.json"
    apiPackage = "com.example.springrest.infrastructure.petstore.api"
    modelPackage = "com.example.springrest.infrastructure.petstore.model"
}
sourceSets {
    main {
        java.srcDirs += "$buildDir/generate-sources/openapi/src/main/java"
    }
}
compileJava.dependsOn tasks.openApiGenerate ⑪

```

Przydatne linki:

<https://github.com/OpenAPITools/openapi-generator/tree/master/modules>

<https://petstore3.swagger.io/>

<https://openapi-generator.tech/docs/generators/java/>

<https://openapi-generator.tech/docs/plugins>

Konfiguracja Maven + dokumentacja:

<https://github.com/OpenAPITools/openapi-generator/tree/master/modules>

<https://openapi-generator.tech/docs/generators/java/>

<https://github.com/OpenAPITools/openapi-generator/tree/master/modules/openapi-generator-maven-plugin#usage>

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://maven.apache.org/POM/4.0.0" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.0.0</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.example</groupId>
  <artifactId>rest-api</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>rest-api</name>
  <properties>
    <java.version>17</java.version>
    <openapi-generator-maven-plugin.version>6.0.1</openapi-generator-maven-plugin.version>
  </properties>
  <dependencies>
    <!-- ... -->
  </dependencies>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.testcontainers</groupId>
        <artifactId>testcontainers-bom</artifactId>
        <version>${testcontainers.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
  <build>
    <finalName>car-dealership-boot-rest</finalName>
    <plugins>

```

```

<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <configuration>
    <excludes>
      <exclude>
        <groupId>org.projectlombok</groupId>
<artifactId>lombok</artifactId>
      </exclude>
    </excludes>
  </configuration>
</plugin>
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>${maven-compiler-plugin.version}</version>
  <configuration>
    <release>${java.version}</release>
    <annotationProcessorPaths>
      <path>
        <groupId>org.projectlombok</groupId>
<artifactId>lombok</artifactId>
<version>${lombok.version}</version>
      </path>
    </annotationProcessorPaths>
  </configuration>
</plugin>
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>${maven-surefire-plugin.version}</version>
  <configuration>
    <trimStackTrace>false</trimStackTrace>
    <argLine>${surefireArgLine}</argLine>
    <excludes>
      <exclude>**/IT*.java</exclude>

```

```

</excludes>
</configuration>
</plugin>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-failsafe-plugin</artifactId>
<version>${maven-failsafe-plugin.version}</version>
<configuration>
<classesDirectory>${project.build.outputDirectory}</classesDi
</configuration>
</plugin>
<plugin>
<groupId>org.openapitools</groupId>
<artifactId>openapi-generator-maven-plugin</artifactId>
<version>${openapi-generator-maven-plugin.version}</version>
<configuration>
<generatorName>java</generatorName>
<library>webclient</library>
<generateApiDocumentation>false</generateApiDocumentation>
<generateApiTests>false</generateApiTests>
<generateModelTests>false</generateModelTests>
<generateModelDocumentation>false</generateModelDocumentation>
<configOptions>
<serializableModel>true</serializableModel>
<dateLibrary>java8</dateLibrary>
<serializationLibrary>jackson</serializationLibrary>
</configOptions>
<output>target/generated-sources/openapi</output>
</configuration>
<executions>
<execution>
<id>cepik-client-generation</id>
<goals>
<goal>generate</goal>
</goals>
</configuration>

```

```

    <inputSpec>${project.basedir}/src/main/resources/petstorev3.0
    <apiPackage>com.example.infrastructure.petstore.api</apiPackage>
<modelPackage>com.example.infrastructure.petstore.model</modelPackage>
    </configuration>
  </execution>
</executions>
</plugin>
</plugins>
</build>
</project>

```

Nasze klasy, które ręcznie implementowały API zamieniamy na 2 poniższe:

```

@Service
@AllArgsConstructor
public class PetClientImpl implements PetDAO {
    private final PetApi petApi;
    private final PetMapper petMapper;
    @Override
    public Optional<Pet> getPet(final Long petId) {
        try {
            final var available = petApi.findPetsByStatusWithHttpInfo("available")
                .getBody();
            return Optional.ofNullable(petApi.getPetById(petId).block())
                .map(petMapper::map);
        } catch (Exception e) {
            return Optional.empty();
        }
    }
}

```

```

import org.springframework.stereotype.Component;
import java.util.Optional;
@Component
public class PetMapper {


```

```

public Pet map(com.example.springrest.infrastructure.petstore
return Pet.builder()
    .id(pet.getId())
    .name(pet.getName())
    .status(Optional.ofNullable(pet.getStatus()).map(status -> sta
    .build());
}
}

```

## HTTP Clients - RestTemplate vs RestClient vs WebClient

 [New in Spring Framework 6.1: RestClient](#) - Maciej Walkowiak

[Getting Started with the Web Client in Spring Boot & Writing Tests](#) - Dan Vega

### **RestClient vs. WebClient vs RestTemplate: Choosing the right library to call REST API in Spring Boot - Digma**

- **RestTemplate:** To starsza, synchroniczna biblioteka, używana głównie w aplikacjach opartych na Spring MVC. Obsługuje blokujące wywołania HTTP, co oznacza, że każde żądanie oczekuje na odpowiedź, zanim wątek może kontynuować pracę. Jest to efektywne przy małej liczbie żądań, ale nie radzi sobie dobrze z większą skalą operacji lub złożonymi aplikacjami, gdzie potrzebna jest obsługa wielu jednoczesnych żądań(). Spring odradza jej stosowanie w nowych projektach na rzecz bardziej nowoczesnych rozwiązań.
- **RestClient:** Nowy klient HTTP wprowadzony w **Spring 6.1** i **Spring Boot 3.2**, który łączy elementy infrastruktury RestTemplate z API WebClient. Jest synchroniczny, ale oferuje bardziej elastyczny i przyjazny interfejs podobny do WebClienta. Dzięki temu, RestClient może być używany w projektach, gdzie potrzebne są blokujące operacje, bez potrzeby dodawania zależności do Spring WebFlux().
- **WebClient:** Wprowadzony w **Spring 5** jako część Spring WebFlux, WebClient obsługuje zarówno operacje synchroniczne, jak i asynchroniczne, korzystając



z reaktywnego modelu. WebClient oferuje nieblokującą obsługę żądań HTTP, co czyni go idealnym do aplikacji o dużym obciążeniu, gdzie wielu użytkowników jednocześnie wykonuje operacje HTTP. Pozwala na przetwarzanie większej liczby żądań przy mniejszym zużyciu zasobów, dzięki czemu jest bardziej wydajny niż RestTemplate().

## Swagger Codegen vs Generator OpenAPI - narzędzia służące do automatycznego generowania kodu klienta i serwera na podstawie specyfikacji API OpenAPI/Swagger

**Swagger Codegen:** Dobre narzędzie, zwłaszcza jeśli pracujesz nad starszym projektem lub korzystasz z ograniczonego zestawu technologii.

- Dokumentacja: [Swagger Codegen Documentation](#)
- Repozytorium GitHub: [Swagger Codegen GitHub](#)

**OpenAPI Generator:** Bardziej nowoczesne i elastyczne narzędzie, szczególnie przydatne w nowych projektach oraz tam, gdzie potrzebne jest wsparcie dla wielu języków i nowszych frameworków.

- Dokumentacja: [OpenAPI Generator Documentation](#)
- Repozytorium GitHub: [OpenAPI Generator GitHub](#)

<https://spring.academy/courses/building-a-rest-api-with-spring-boot/lessons/introduction> - Additional Course