

REST API

REST – Representational State Transfer – styl architektury oprogramowania, opierający się o zbiór wcześniej określonych reguł opisujących jak definiowane są zasoby, a także umożliwiających dostęp do nich.

Podczas tworzenia REST API, do komunikacji wykorzystuje się następujące metody HTTP:

- GET
- POST
- PUT
- DELETE
- CONNECT
- OPTIONS
- TRACE
- PATCH
- HEAD

REST API

- GET - odczyt danych
- POST - zapis nowych danych
- PUT - aktualizacja istniejących danych
- DELETE - usunięcie danych

1. Klient przygotowuje zapytanie (request) i przesyła je pod odpowiedni adres (tzw. endpoint).
2. System otrzymuje zapytanie klienta i przygotowuje odpowiedź (response).
3. System zwraca odpowiedź na zapytanie klienta.
4. Klient otrzymuje i przetwarza odpowiedź.

REST API

2 rodzaje adnotacji dla kontrolerów:

@Controller

Nie ma żadnych założeń co do stylu aplikacji, którą budujemy. Nie ma znaczenia, czy budujemy REST API, czy tradycyjną aplikację w stylu MVC.


@RestController

Przeznaczony dla REST API. W takim przypadku zazwyczaj chcemy, aby nasze odpowiedzi zawierały dane (obiekty DTO). W tym celu używa się adnotacji `@ResponseBody`. Jednak `@RestController` zawiera już w sobie adnotację `@ResponseBody` i nie trzeba jej dodawać ręcznie.

REST API

Adnotacja @RequestMapping

Ta adnotacja używana jest globalnie dla całego kontrolera i wskazuje na początek adresu (endpointa), który będzie używany dla wszystkich metod znajdujących się w tym kontrolerze.

```
@RestController
@RequestMapping(value =  "/projects")
public class ProjectController {

}
```

REST API

Adnotacje Springowe dla poszczególnych typów żądań HTTP

- @GetMapping
- @PostMapping
- @PutMapping
- @DeleteMapping

Tymi adnotacjami oznaczamy poszczególne metody kontrolera.

REST API

Przykład metody typu GET

```
@Override  
public Optional<Project> findById(Long id) {  
    return projectRepository.findById(id);  
}
```

```
@GetMapping(value = "/{id}")  
public Project findOne(@PathVariable Long id) {  
    return this.projectService.findById(id).get();  
}
```

REST API

Przykład metody typu GET

`http://localhost:8080/projects/1`

```
{"id":1,"name":"Project 1","startDate":"2021-08-15"}
```

Obiekty DTO (Data Transport Object)

W komunikacji REST API nie należy stosować bezpośrednio obiektów domenowych (encji), ale konwertować je na obiekty typu DTO.

Dlaczego?

- Encje nie są zasobami. Nie zawsze wszystkie dane encji powinny być udostępniane klientowi.
- Pobieranie wszystkich danych może prowadzić do problemów wydajnościowych.
- Brak odseparowania warstwy dostępu do danych od warstwy kontrolerów.

Obiekty DTO (Data Transport Object)

```
public class ProjectDto {  
  
    private Long id;  
  
    private String name;  
  
    private LocalDate dateCreated;  
  
    private Set<TaskDto> tasks;  
  
    public ProjectDto(Long id, String name, LocalDate dateCreated) {  
        this.id = id;  
        this.name = name;  
        this.dateCreated = dateCreated;  
    }  
}
```

Mapper

```
public ProjectDto convertToDto(Project entity) {  
    ProjectDto dto = new ProjectDto(entity.getId(), entity.getName(), entity.getStartDate());  
    dto.setTasks(entity.getTasks().stream().map(t -> convertTaskToDto(t)).collect(Collectors.toSet()));  
    return dto;  
}  
  
public Project convertToEntity(ProjectDto dto) {  
    Project project = new Project(dto.getName(), dto.getDateCreated());  
    project.setId(dto.getId());  
    return project;  
}
```

Alternatywnie gotowe narzędzia takie jak ModelMapper, MapStruct

Użycie mappera i DTO

```
@GetMapping(value = "{id}")
public ProjectDto findOne(@PathVariable Long id) {
    ProjectMapper mapper = new ProjectMapper();
    Project entity = projectService.findById(id).orElseThrow(
        () -> new ResponseStatusException(HttpStatus.NOT_FOUND));
    return mapper.convertToDto(entity);
}
```

```
@PostMapping
public void create(ProjectDto newProject) {
    ProjectMapper mapper = new ProjectMapper();
    Project entity = mapper.convertToEntity(newProject);
    this.projectService.save(entity);
}
```

Adnotacja @RequestParam

Adnotacja @RequestParam służy do mapowania parametrów zapytania pobieranych z adresu URL.

Przykład zapytania z parametrem name:

http://localhost:8080/projects?name=Project 2

```
@GetMapping  
public List<ProjectDto> findProjects(@RequestParam("name") String name) {  
    ProjectMapper mapper = new ProjectMapper();  
    return this.projectService.findByName(name).stream().map(project -> mapper.convertToDto(project)).collect(Collectors.toList());  
}
```

REST API - obsługa błędów

2 możliwości

- *ResponseStatusException*
- *@ControllerAdvice* i *@ExceptionHandler*

ResponseStatusException

```
@GetMapping(value = "/{id}")
public ProjectDto findOne(@PathVariable Long id) {
    ProjectMapper mapper = new ProjectMapper();
    Project entity = projectService.findById(id).orElseThrow(
        () -> new ResponseStatusException(HttpStatus.NOT_FOUND));
    return mapper.convertToDto(entity);
}
```

ResponseStatusException zapewnia prosty sposób na określenie dokładnie, jaki kod HTTP odeślemy klientowi w danym przypadku.

Standardowy błąd 500

```
@DeleteMapping(value="/{id}")  
public void delete(@PathVariable Long id) {  
    this.projectService.delete(id);  
}
```

```
1 {  
2     "timestamp": "2021-08-18T08:22:13.604+00:00",  
3     "status": 500,  
4     "error": "Internal Server Error",  
5     "path": "/projects/2"  
6 }
```

@ControllerAdvice i @ExceptionHandler

```
@ControllerAdvice
public class ControllersAdvice {

    @ExceptionHandler(EmptyResultDataAccessException.class)
    public ResponseEntity<ErrorMessage> handleDataRetrievalException(EmptyResultDataAccessException ex) {
        return new ResponseEntity<ErrorMessage>(new ErrorMessage(ex.getMessage()), HttpStatus.NOT_FOUND);
    }

    class ErrorMessage{
        String message;

        public ErrorMessage(String message) {
            this.message = message;
        }

        public String getMessage() {
            return message;
        }
    }
}
```


Konsumowanie webserwisów

```
private RestTemplate restTemplate = new RestTemplate();  
private static final String baseUrl = "http://localhost:8080/projects/";
```

```
ResponseEntity<ProjectDto> response = restTemplate.getForEntity(url: baseUrl + "1", ProjectDto.class);
```

Konsumowanie webserwisów

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.DEFINED_PORT)
```

```
public ProjectDto() {  
}
```

Konsumowanie webserwisów

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.DEFINED_PORT)
public class ProjectRestAPILiveTest {

    private RestTemplate restTemplate = new RestTemplate();
    private static final String baseUrl = "http://localhost:8080/projects/";

    @Test
    public void givenProjectExists_whenGet_thenSuccess() {
        ResponseEntity<ProjectDto> response = restTemplate.getForEntity(url: baseUrl + "1", ProjectDto.class);

        assertEquals(response.getStatusCodeValue(), Matchers.equalTo(operand: 200));
        assertNotNull(response.getBody());
    }

    @Test
    public void givenNewProject_whenCreated_thenSuccess() {
        ProjectDto newProject = new ProjectDto(new Random().nextLong(), name: "First Project", LocalDate.now());
        ResponseEntity<Void> response = restTemplate.postForEntity(baseUrl, newProject, Void.class);

        assertTrue(condition: response.getStatusCode() == HttpStatus.OK);
    }
}
```

Programowanie aspektowe

Mówiąc najprościej, pozwala nam izolować funkcje, które dotyczą większej części kodu lub funkcji systemu (tzw. cross-cutting concerns).

Nowe zachowania definiujemy w separacji wobec istniejącego kodu za pomocą aspektów.

Programowanie aspektowe

Mówiąc najprościej, pozwala nam izolować funkcje, które dotyczą większej części kodu lub funkcji systemu (tzw. cross-cutting concerns).

Nowe zachowania definiujemy w separacji wobec istniejącego kodu za pomocą aspektów.

Pojęcia

Aspekt - to logika/zachowanie, którą chcemy dodać

Join Point - punkt w programie w którym następuje wykonanie metody.

Point Cut - określa miejsca w programie np. zestaw klas dla których porada ma zostać uruchomiona.

Advice - Moment działania aspektu, który określony jest przez punkt złączenia (around, before, after).

Przykład

Advice PointCut



```
@Aspect
@Component
public class ProjectServiceAspect {

    private static final Logger LOG = LoggerFactory.getLogger(ProjectServiceAspect.class);

    @Before("execution(* pl.fis.spring1.service.ProjectServiceImpl1.findById(Long))")
    public void before(JoinPoint joinPoint) {
        LOG.info("Searching Project with Id {}", joinPoint.getArgs()[0]);
    }
}
```

Aspekt

JoinPoint

Advice @Around

```
@Around("execution(* pl.fis.spring1.service.ProjectServiceImpl.save(*))")
public Object aroundSave(ProceedingJoinPoint joinPoint) {
    Object val = joinPoint.getArgs()[0];
    try {
        LOG.info("saving project : {}", val);
        val = joinPoint.proceed();
        LOG.info("project saved successfully !!");
    } catch (Throwable e) {
        LOG.error("error while saving project: ", e);
    }
    return val;
}
```


Zdarzenia (events)

- Zdarzenia pozwalają nam pisać luźno powiązane komponenty. Czasami zamiast wiązać obiekty np. poprzez wstrzykiwanie zależności lepiej obsłużyć komunikację poprzez wysyłanie i subskrybowanie zdarzeń.
- Zdarzenia Springowe są całkowicie synchroniczne, ponieważ są wysyłane i przetwarzane w tym samym wątku.
- Całe rozwiązanie nie opuszcza maszyny wirtualnej Java, na którym działa. Oznacza to, że nie jest to broker komunikatów ani magistrała komunikacyjna, a zatem nie jest to dobre rozwiązanie dla systemów rozproszonych.

Przykład eventu

```
public class ProjectCreatedEvent
{
    private Long projectId;

    public Long getProjectId() {
        return projectId;
    }

    public void setProjectId(Long projectId) {
        this.projectId = projectId;
    }
}
```

|

Publikowanie eventu

`@Autowired`

```
private ApplicationEventPublisher publisher;
```

`@PostMapping` 

```
public void create(ProjectDto newProject) {  
    publisher.publishEvent(new ProjectCreatedEvent(newProject.getId()));  
    ProjectMapper mapper = new ProjectMapper();  
    Project entity = mapper.convertToEntity(newProject);  
    this.projectService.save(entity);  
}
```

Nasłuchiwanie na event

```
@Component
public class ProjectCreatedEventListener {

    private static final Logger LOG = LoggerFactory.getLogger(ProjectCreatedEventListener.class);

    @EventListener
    public void handleProjectCreatedEvent(ProjectCreatedEvent projectCreatedEvent) {
        LOG.info("New Project Created with Id {}", projectCreatedEvent.getProjectId());
    }
}
```

Spring Security

Spring Security to rozbudowany framework, który zapewnia funkcje uwierzytelniania i autoryzacji dla projektów.

Dzięki temu narzędziu możemy chronić nasze aplikacje przed różnymi atakami, takimi jak utrwalanie sesji, clickjacking, CSRF itp.

Dodanie zależności

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-data-jpa</artifactId>  
</dependency>
```

Konfiguracja w application.properties

```
spring.security.user.name=fisadmin  
spring.security.user.password=fisadmin
```

Logowanie

Please sign in

Sign in

Konfiguracja

```
@Configuration
@EnableWebSecurity

public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests() ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
            .antMatchers( ...antPatterns: "/login*") ExpressionUrlAuthorizationConfigurer<...>.AuthorizedUrl
            .permitAll() ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
            .anyRequest() ExpressionUrlAuthorizationConfigurer<...>.AuthorizedUrl
            .authenticated() ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
            .and() HttpSecurity
            .formLogin();
    }
}
```


Konfiguracja użytkowników

1

```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication() InMemoryUserDetailsManagerConfigurer<AuthenticationManagerBuilder>
        .withUser( username: "user") UserDetailsManagerConfigurer<...>.UserDetailsBuilder
        .password(encoder().encode( charSequence: "user"))
        .roles("USER");
}
```

```
@Bean
public PasswordEncoder encoder() {
    return new BCryptPasswordEncoder();
}
```

Dostęp do Security Context

```
@GetMapping("/testSecurity")
public String testSecurity() {
    SecurityContext context = SecurityContextHolder.getContext();
    Authentication authentication = context.getAuthentication();
    LOG.info(authentication.getName());
    authentication.getAuthorities().stream().forEach(
        grantedAuthority -> LOG.info(grantedAuthority.getAuthority())
    );
    return "security test";
}
```