

## Ćwiczenie: Spring Retry na endpointcie

Spodzielając się dużego ruchu na tym endpointcie, spodziewamy się również kolizji Optimistic Lockingu.

Obecny kod kontrolera PointOfInterestController wygląda następująco:

```
@GetMapping(value = "/poi/{id}")  
  
public Point get(@PathVariable UUID id) {  
    return service.findById(id)  
        .orElseThrow(NotExists::new);  
}
```

Endpoint narażony na częste kolizje

```
@PatchMapping(value = "/poi/{id}")  
  
public Point patch(@PathVariable UUID id,  
    @RequestBody @Valid PointOfInterestUpdate update) {  
    return service.update(id, update);  
}
```

By zmniejszyć ilość błędów widocznych dla klientów endpointu i konieczności powtórzeń ze strony klientów:

- Dodaj adnotację @Retryable dla wyjątku OptimisticLockException nad metodą patch
- Włącz spring retry poprzez adnotację @EnableRetry w klasie AppRunner

W kolejnym ćwiczeniu zaimplementujesz dodatkowy test weryfikujący funkcjonalność retry.

### Przygotuj odpowiedź na poniższe pytania:

- Czy w tym kontekście @Retryable ma sens dla każdego wyjątku?
- Dla jakich wyjątków warto powtarzać transakcję?

## Ćwiczenie: WebMvcTest weryfikacja Spring Retry na endpointcie

Zaimplementuj dodatkowy test patchWithRetry w klasie PointOfInterestControllerMockMvcTest

Możesz posłużyć się istniejącym testem patchExisting:

@Test

```
void patchExisting() throws Exception {  
    Point givenPoint = PointsFixture.evb().setLocation(rooseveltlaanInGent());  
    whenUpdating().thenReturn(givenPoint);  
    mockMvc.perform(  
        patch("/poi/{id}", givenId)  
            .contentType(MediaType.APPLICATION_JSON)  
            .content(jsonOf(rooseveltlaanInGent()))  
            .accept(MediaType.APPLICATION_JSON)  
            .andExpect(status().isOk())  
            .andExpect(content().contentType(MediaType.APPLICATION_JSON))  
            .andExpect(content().json(jsonOf(givenPoint)));  
    Mockito.verify(service, Mockito.times(1))  
        .update(givenId, location(rooseveltlaanInGent()));  
}
```

Konfigurowanie mocka serwisu

Weryfikacja ilości powtórzeń

W nowym teście wykonaj dwie zmiany:

- 1) Skonfiguruj mocka serwisu tak by przy pierwszym wywołaniu rzucił wyjątek OptimisticLockException, a przy drugim wywołaniu zwrócił prawidłowy wynik.
- 2) Zweryfikuj dwa powtórzenia wywołania metody update serwisu.

### Pond to:

Upewnij się, że dla innych wyjątków retry nie powtarza operacji.

Zwróć uwagę że nowo dodany test działa znacznie dłużej od pozostałych.

Czy dla wyjątku OptimisticLockException warto czekać pomiędzy powtórzeniami?

### Przygotuj odpowiedź na poniższe pytania:

- Jak inaczej zwrócić status 404 NOT\_FOUND z kontrolera MVC?
- Czy testy PointOfInterestControllerMockMvcTest weryfikują kontrakt REST API, to znaczy czy przypadkowe złamanie kompatybilności API zostanie wyłapane przez któryś z testów?

## Ćwiczenie: Resilience4j Retry do klienta RestTemplate

W klasie LocationUpdateClient jest wywołanie klienta RestTemplate:

```
void sendLocationUpdate(UUID id, PointLocation location) {  
    HttpEntity<LocationUpdateFields> payload =  
        payload(LocationUpdateFields.from(location));  
  
    restTemplate.exchange(url + "/locations/{id}",  
        HttpMethod.POST, payload, Void.class,  
        Map.of("id", id)  
    );  
}
```

Skorzystaj z klasy Retry z pakietu io.github.resilience4j.retry

```
Retry.ofDefaults(„location-update”).executeRunnable( ... );
```

by powtórzyć to wywołanie w razie problemów z komunikacją.

Przyjrzyj klasę RetryConfig, zweryfikuj domyślne ustawienia dla retry i jakie są możliwości jego konfiguracji.

Poeksperymentuj z:

```
Retry.of(„location-update”, RetryConfig.custom()  
...  
    .retryExceptions(...)   
    .retryOnResult(result -> ...)   
...  
    .build()  
)
```

zaproponuj najlepsze według Ciebie ustawienia.

## Ćwiczenie: RestClientTest weryfikacja Resilience4j Retry na kliencie

Zaimplementuj dodatkowe testy w klasie LocationUpdateClientTest:

- **successRequestAfterRetry** - weryfikujący zachowanie retry w scenariuszu: po pierwszym requeście zakończonym wyjątkiem połączenia oczekiwany jest kolejny identyczny request zakończony sukcesem.
- **persistentServerError** - weryfikujący zachowanie retry w scenariuszu: kilkukrotne wywołanie serwisu konsekwentnie kończy się wyjątkiem połączenia.

Możesz posłużyć się istniejącymi testami successfulRequest oraz internalServerError:

@Test

```
void successfulRequest() {  
    expectRequest()  
        .andExpect(content().json(expectedJson()))  
        .andRespond(withSuccess());
```

Konfigurowanie odpowiedzi

```
    client.sendLocationUpdate(givenId, givenLocation());  
    server.verify();  
}
```

@Test

```
void internalServerError() {  
    expectRequest()  
        .andExpect(content().json(expectedJson()))  
        .andRespond(withStatus(HttpStatus.INTERNAL_SERVER_ERROR));
```

Weryfikacja propagowanego wyjątku

```
    assertThatThrownBy(() -> client.sendLocationUpdate(givenId, givenLocation()))  
        .isExactlyInstanceOf(HttpServerErrorException.InternalServerError.class)  
        .hasMessageStartingWith("500 Internal Server Error");  
    server.verify();  
}
```

By zweryfikować dwa lub więcej powtórzonych wywołań serwisu możesz kilkukrotnie wywołać instrukcję:

```
expectRequest()  
    .andExpect(content().json(expectedJson()))  
    .andRespond(...);
```

By zasymulować wywołanie zakończonym wyjątkiem połączenia zdefiniuj oczekiwaną odpowiedź jako:

```
.andRespond(withException(new ConnectException("Connection reset")));
```

Kiedy Retry nie zakończy się sukcesem propaguje wyjątek, zweryfikuj go w teście.

## Ćwiczenie: Rozróżnianie wyjątków klienta RestTemplate

Przyjrzyj się hierarchii wyjątków rzucanych z RestTemplate:

```
RestClientException
    UnknownContentTypeException
    RestClientResponseException
        HttpStatusCodeException
            HttpClientErrorException
                Conflict in HttpClientErrorException
                Forbidden in HttpClientErrorException
                Unauthorized in HttpClientErrorException
                BadRequest in HttpClientErrorException
                MethodNotAllowed in HttpClientErrorException
                UnprocessableEntity in HttpClientErrorException
                Gone in HttpClientErrorException
                NotAcceptable in HttpClientErrorException
                UnsupportedMediaType in HttpClientErrorException
                TooManyRequests in HttpClientErrorException
                NotFound in HttpClientErrorException
            HttpServerErrorException
                InternalServerError in HttpServerErrorException
                NotImplemented in HttpServerErrorException
                BadGateway in HttpServerErrorException
                GatewayTimeout in HttpServerErrorException
                ServiceUnavailable in HttpServerErrorException
            UnknownHttpStatusCodeException
    ResourceAccessException
```

Wybierz z pośród nich te wyjątki po których powinno się ponawiać wywołanie.

Dostosuj odpowiednio klasy LocationUpdateClient, LocationUpdateClientTest.

**Ćwiczenie: Wykonanie żądanie REST po transakcji bazodanowej**

Obecna implementacja metody update serwisu PointOfInterestService wywołuje serwis REST jeszcze przed potwierdzeniem zapisu w bazie danych. Wiadomym jest, że szereg czynników może spowodować wycofanie (Rollback) transakcji:

`@Transactional`

Aspekt transakcji

```
public Point update(UUID id, PointOfInterestUpdate update) {  
    Point point = repository.findById(id).orElseThrow(NotExists::new);  
    if (update.getOpeningHours() != null) {  
        point.setOpeningHours(update.getOpeningHours());  
    }  
    if (update.getLocation() != null  
        && !Objects.equals(point.getLocation(), update.getLocation())) {  
        point.setLocation(update.getLocation());  
        client.sendLocationUpdate(id, PointLocation.from(point.getLocation()));  
    }  
    repository.save(point);  
    return point;  
}
```

Wywołanie klienta REST

Obecna implementacja sprawia, iż istnieje możliwość poinformowania serwisu o dokonanej zmianie lokalizacji, kiedy w rzeczywistości ona nie zaszła z powodów np. constraintu czy długotrwałych problemów z bazą.

**Dokonaj zmiany:**

Zamiast bezpośredniego wywołania klienta REST użyj metody statycznej

**TransactionSynchronizationManager.registerSynchronization**, przekazując do niej własną implementację nadpisującą klasę abstrakcyjną

**TransactionSynchronizationAdapter**.

Nadpisz wyłącznie metodę **afterCommit**, wywołaj w niej klienta REST jak poprzednio.

## Ćwiczenie: Spring Events + obsługa eventu po transakcji

Alternatywą metodą wywołania logiki po transakcji bazodanowej dla **TransactionSynchronizationManager.registerSynchronization** jest skorzystanie ze Spring Events.

1) W klasie PointOfInterestService **zamiast** bezpośrednio wywoływać serwis REST skonstruuj reprezentację eventu dziedzicznego:

```
new LocationUpdated(  
    id, PointLocation.from(point.getLocation())  
);
```

Stwórz nową klasę LocationUpdated, klasa PointLocation już istnieje.

2) Opublikuj event metodą ApplicationEventPublisher.publishEvent(Object), publisher musi być wstrzyknięty do serwisu (zamiast klienta REST).

3) Do klasy LocationUpdateClient dodaj metodę słuchającą na event LocationUpdated i wyślij w niej żądanie REST:


```
@EventListener  
  
public void handle(LocationUpdated event) {  
    ...  
}
```

4) Relacją obsługi eventu z transakcją bazodanową można sterować za pomocą adnotacji:

```
@TransactionalEventListener(phase = TransactionPhase.???)
```

Przejrzyj dostępne fazy transakcji do wyboru

Zwróć uwagę na ikonę ziarenka w słuchawkach w IntelliJ-u przy publikacji eventu:

 publisher.publishEvent

## Ćwiczenie: Outbox Pattern for REST

Outbox pattern stosujemy by:

- zagwarantować wysłanie żądań po transakcji nawet w sytuacji, kiedy aplikacja była restartowana
- mieć zapis żądań, które nie dotarły do klienta (by np. ponowiły się po naprawie usterki)

### UWAGA:

To zadanie należy zacząć od implementacji serwisu i klienta działających synchronicznie w ramach jednej transakcji. Zadanie proszę zrealizować w nowej klasie klienta

### **LocationUpdateOutboxClient.**

1) Zapoznaj się z klasami:

- **LocationUpdateOutboxClient** szkielet klienta, który trzeba dokończyć
- **OutboxRequest** encja, która pozwala zapisać żądanie przed wysyłką wiadomości
- **OutboxRequestRepository**, repozytorium dokonujące zapisu i odczytu encji
- **LocationUpdateOutboxClientTest**, test weryfikujący zachowanie klienta (wymaga uruchomionego lokalnie docker-a)

2) W klasie **LocationUpdateOutboxClient** zaimplementuj metodę **sendLocationUpdate** wykonującą następujące kroki:

- skonstruuj instancję **OutboxRequest** z niezbędnymi informacjami dotyczącymi żądania
- zapisz żądanie w repozytorium
- po pozytywnym zakończeniu transakcji:
  - prześlij żądanie
  - usuń zapisane żądanie z repozytorium

By zarejestrować czynności po zakończeniu transakcji skorzystaj z metody **TransactionSynchronizationManager.registerSynchronization**

3) Zweryfikuj poprawność za pomocą testu **LocationUpdateOutboxClientTest**.

Pytania dodatkowe:

- Czy serwis wywołujący nowego klienta czeka na pozytywną wysyłkę żądania?
- Czy serwis wywołujący nowego klienta zostanie poinformowany o wyjątku przy wysyłce żądania?
- Jak twoim zdaniem powinno być (odnośnie czekania i wyjątków)?
- Jak tym sterować - jakie zmiany należy dokonać?
- Jak automatycznie wysłać żądania po restarcie / naprawie długotrwałej usterki?
- Czy każdy wyjątek powinien pozostawiać żądanie w kolejce do wysyłki?