Wielowątkowość

dzień 3

v3.0

Plan

1. Wielowątkowość w spring
2. Executor
3. Wykorzystanie @Async
4. Scheduled
5. MVC i @Async

Wielowątkowość w Spring

@Async

Najprostszym sposobem na wprowadzenie wielowątkowości w aplikacjach Spring jest oznaczenie **publicznej** metody za pomocą adnotacji **`@Async`** spowoduje to wykonanie jej w osobnym wątku. Innymi słowy wywołujący nie będzie czekał na zakończenie działania wywołanej metody

```
@Async
public void simpleMethod() {
  System.out.println("Simple method name: " + Thread.currentThread().getName());
}
```

@EnableAsync

Aby zacząć korzystać z adnotacji @Async należy umożliwić wielowątkowe wykonywanie kodu poprzez adnotację **@EnableAsync** w klasie konfiguracyjnej. Za pomocą tej adnotacji Spring automatycznie wykryje wszystkie metody oznaczone adnotacją @Async i umożliwi im pracę w trybie wielowątkowym.

```
@Configuration
@EnableAsync
public class SpringAsyncConfig {
  // class body
}
```

Metody nie zwracające wartości

Podczas tworzenia metod które działają w środowisku wielowątkowym w Spring metody nie muszą oczywiście zwracać wartości a jedynie wykonywać pewne operacje:

```
@Async
public void asyncMethodWithVoidReturnType() {
  System.out.println("Method name: " + Thread.currentThread().getName());
}
```

Metody zwracające wartość

Natomiast nic nie stoi na przeszkodzie aby metody zwracały pewne wartości, w tym celu użyjemy poznanej już konstrukcji Future:

```
@Async
public Future<String> asyncMethodWithReturnType() {
  System.out.println("Running method: " + Thread.currentThread().getName());
  try {
    Thread.sleep(5000);
    return new AsyncResult<String>("Some text");
  } catch (InterruptedException e) {
    // to do
  }
}
```

Metody zwracające wartość

Natomiast nic nie stoi na przeszkodzie aby metody zwracały pewne wartości, w tym celu użyjemy poznanej już konstrukcji Future:

```
@Async
public Future<String> asyncMethodWithReturnType() {
  System.out.println("Running method: " + Thread.currentThread().getName());
  try {
    Thread.sleep(5000);
    return new AsyncResult<String>("Some text");
  } catch (InterruptedException e) {
    // to do
  }
}
```

Wykonywana praca i zwracany jest obiekt AsyncResult typu String

AsyncResult

Future to interfejs który reprezentuje zwrócony wynik po wykonaniu danej metody. Długo działające metody to dobrzy kandydaci aby w ten sposób je wykonywać, umożliwia to wykonywanie innych zadań w czasie gdy oczekujemy na zwrócenie wyniku przez inną metodę. Wartość zwracana przez metodę jest opakowana w implementację interfejsu Future czyli klasę **`AsyncResult`**

Dla przypomnienia przydatnymi metodami podczas pracy z interfejsem Future jest **`get()`** - metoda pobierająca wynik zwrócony po wykonaniu zadania, oraz metoda **`isDone()`** która sprawdza czy metoda oznaczona jako Future zakończyła już swoje działanie

Zadania

Executor

Executor

Domyślnie Spring używa klasy **SimpleAsyncTaskExecutor** aby uruchamiać zadania w formie wielowątkowej

Klasa ta jest możliwa do nadpisania w Spring na dwa sposoby:

- na poziomie aplikacji
- na poziomie konkretnej metody

Nadpisanie Executor-a na poziomie metody

W celu nadpisania Spring Executor-a na poziomie metody należy zadeklarować Executor w klasie konfiguracyjnej:

```
@Configuration
@EnableAsync
public class SpringAsyncConfig {
  @Bean(name = "threadPoolTaskExecutor")
  public Executor threadPoolTaskExecutor() {
    return new ThreadPoolTaskExecutor();
  }
}
```

Nadpisanie Executor-a na poziomie metody

W celu nadpisania Spring Executor-a na poziomie metody należy zadeklarować Executor w klasie konfiguracyjnej:

```
@Configuration
@EnableAsync
public class SpringAsyncConfig {
  @Bean(name = "threadPoolTaskExecutor")
  public Executor threadPoolTaskExecutor() {
    return new ThreadPoolTaskExecutor();
  }
}
```

Informacja o tym jaki będzie alias nowego Executor-a

Nadpisanie Executor-a na poziomie metody

W celu nadpisania Spring Executor-a na poziomie metody należy zadeklarować Executor w klasie konfiguracyjnej:

```
@Configuration
@EnableAsync
public class SpringAsyncConfig {
  @Bean(name = "threadPoolTaskExecutor")
  public Executor threadPoolTaskExecutor() {
    return new ThreadPoolTaskExecutor();
  }
}
```

Utworzenie metody zwracającej obiekt ThreadPoolTaskExecutor

Nadpisanie Executor-a na poziomie metody

Po konfiguracji nowego Executor-a w metodzie która ma z niego korzystać należy w adnotacji @Async podać jego nazwę określoną wcześniej poprzez **`@Bean(name = "threadPoolTaskExecutor")`**:

```
@Async("threadPoolTaskExecutor")
public void methodWithCustomExecutor() {
  System.out.println("Method name: " + Thread.currentThread().getName());
}
```

Nadpisanie Executor-a na poziomie metody

Po konfiguracji nowego Executor-a w metodzie która ma z niego korzystać należy w adnotacji @Async podać jego nazwę określoną wcześniej poprzez **`@Bean(name = "threadPoolTaskExecutor")`**:

```
@Async("threadPoolTaskExecutor")
public void methodWithCustomExecutor() {
  System.out.println("Method name: " + Thread.currentThread().getName());
}
```

Użycie wcześniej stworzonej konfiguracji Executor-a

Nadpisywanie Executor-a na poziomie aplikacji

Drugim sposobem nadpisania domyślnego Executor-a w Spring jest nadpisanie go na poziomie całej aplikacji. W tym celu klasa konfiguracyjna powinna implementować interfejs **`AsyncConfigurer`** i nadpisywać jego metodę **`getAsyncExecutor()`**:

```
@Configuration
@EnableAsync
public class SpringAsyncConfig implements AsyncConfigurer {
  @Override
  public Executor getAsyncExecutor() {
    return new ThreadPoolTaskExecutor();
  }
}
```

Nadpisywanie Executor-a na poziomie aplikacji

Drugim sposobem nadpisania domyślnego Executor-a w Spring jest nadpisanie go na poziomie całej aplikacji. W tym celu klasa konfiguracyjna powinna implementować interfejs **`AsyncConfigurer`** i nadpisywać jego metodę **`getAsyncExecutor()`**:

```
@Configuration
@EnableAsync
public class SpringAsyncConfig implements AsyncConfigurer {
  @Override
  public Executor getAsyncExecutor() {
    return new ThreadPoolTaskExecutor();
  }
}
```

Implementacja interfejsu AsyncConfigurer

Nadpisywanie Executor-a na poziomie aplikacji

Drugim sposobem nadpisania domyślnego Executor-a w Spring jest nadpisanie go na poziomie całej aplikacji. W tym celu klasa konfiguracyjna powinna implementować interfejs **`AsyncConfigurer`** i nadpisywać jego metodę **`getAsyncExecutor()`**:

```
@Configuration
@EnableAsync
public class SpringAsyncConfig implements AsyncConfigurer {
  @Override
  public Executor getAsyncExecutor() {
    return new ThreadPoolTaskExecutor();
  }
}
```

Nadpisanie metody getAsyncExecutor zwracającej obiekt ThreadPoolTaskExecutor

Zadania

Wykorzystanie @Async

Praktyczny przykład @Async

Aby pokazać, jak w praktyczny sposób adnotacji **@Async** i tym samym użyć wielowątkowości w Spring wykonamy prosty przykład

Nasz przykładowy projekt zostanie utworzony w SpringBoot, w którym będziemy używać żądań GET do adresu URL obsługiwanego przez konkretny kontroler z pomocą którego wykona się nasz kod

W kodzie zostanie wykonanych 5 zadań a każde z nich będzie trwało około 1 sekundy ze względu na zastosowanie poznanej już wcześniej metody **`Thread.sleep()`**

Praktyczny przykład @Async

W naszym projekcie Spring Boot użyjemy przydatnej adnotacji **@SpringBootApplication** która automatycznie zastępuje wszystkie poniższe adnotacje (opisane dla przypomnienia):

- **`@Configuration`** - oznacza klasę jako źródło definicji dla kontekstu aplikacji
- **`@EnableAutoConfiguration`** - umożliwia automatyczną konfigurację i dodawanie tzw. "beans" na podstawie określonej ścieżki
- **`@EnableWebMvc`**(jeżeli korzystamy z niego) - jeśli w ścieżce aplikacji Spring Boot znajdzie spring-webmvc wtedy doda tą adnotację automatycznie
- **`@ComponentScan`** - informuje Spring aby wyszukał inne komponenty, konfiguracje, i serwisy w danym pakiecie

Praktyczny przykład @Async

```
@SpringBootApplication
@EnableAsync
public class Application implements AsyncConfigurer {
  public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
  }
```

Praktyczny przykład @Async

```
@SpringBootApplication
@EnableAsync
public class Application implements AsyncConfigurer {
  public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
  }
```

Implementacja interfejsu AsyncConfigurer w naszej klasie

Praktyczny przykład @Async

W naszym przykładzie chcemy nadpisać domyślny dla Spring Executor i zastąpimy go poznanym wcześniej **`ThreadPoolTaskExecutor`**

W tym celu w klasie implementującej interfejs **`AsyncConfigurer`** musimy nadpisać metodę **`getAsyncExecutor()`**, która zwróci obiekt **`ThreadPoolTaskExecutor`**. Poznaliśmy już tą metodę wcześniej i będzie to nadpisanie Executor-a na poziomie aplikacji

Praktyczny przykład @Async

```
@Override
 public Executor getAsyncExecutor() {
   ThreadPoolTaskExecutor taskExecutor = new ThreadPoolTaskExecutor();
   taskExecutor.setMaxPoolSize(10);
   taskExecutor.setThreadNamePrefix("CustomExecutor-");
   taskExecutor.initialize();
   return taskExecutor;
 }
```

Praktyczny przykład @Async

```
@Override
 public Executor getAsyncExecutor() {
   ThreadPoolTaskExecutor taskExecutor = new ThreadPoolTaskExecutor();
   taskExecutor.setMaxPoolSize(10);
   taskExecutor.setThreadNamePrefix("CustomExecutor-");
   taskExecutor.initialize();
   return taskExecutor;
 }
```

Nadpisanie parametrów obiektu klasy ThreadPoolTaskExecutor

Praktyczny przykład @Async

Dodatkową rzeczą którą należy wykonać to poinformować Spring jak ma obsługiwać ewentualne błędy wynikające z kodu wielowątkowego. Na szczęście nie jest to problematyczne i poradzimy sobie z tym nadpisując metodę **`getAsyncUncaughtExceptionHandler()`** w klasie implementującej interfejs **`AsyncConfigurer`**

Tym sposobem metoda getAsyncUncaughtExceptionHandler() zwróci obiekt klasy **`SimpleAsyncUncaughtExceptionHandler`** który obsłuży ewentualne błędy

Praktyczny przykład @Async

```
 @Override
  public AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler() {
    return new SimpleAsyncUncaughtExceptionHandler();
  }
}
```

Praktyczny przykład @Async

Po ukończeniu konfiguracji aplikacji przechodzimy do utworzenia klasy która będzie wykonywała określone zadanie.

Naszą klasę nazwiemy **`MySampleService`** i jedyną metodą która będzie wykonywana w niej to metoda którą nazwiemy **`callAsync()`**

Utworzona metoda **`callAsync()`** oznaczona będzie poznaną już adnotacją **`@Async`** i będzie zwracała typ **`Future`**, natomiast dla uproszczenia zamiast zadania wykonywanego w ciele naszej metody wywołamy metodę **`sleep()`** dla symulacji czasu potrzebnego na wykonanie określonych czynności w metodzie

Ostatecznie utworzona przez nas metoda callAsync() zwróci nowy obiekt typu **`AsyncResult`** który jest w tym przypadku wymagany

Praktyczny przykład @Async

```
@Component
public class MySampleService {
  @Async
  public Future<Long> callAsync(int taskNum) throws InterruptedException {
    long start = System.currentTimeMillis();
    System.out.println("Task: " + taskNum + " is starting");
    Thread.sleep(500);
    long end = (System.currentTimeMillis() - start)
    System.out.println("Elapsed time: " + end);
    return new AsyncResult<Long>(end);
  }
}
```

Praktyczny przykład @Async

Ostatnim krokiem do uruchomienia naszego przykładu będzie utworzenie kontrolera który nazwiemy **`MyAsyncController`**

W metodzie **`executeTasks()`** naszego kontrolera wywołamy trzykrotnie metodę **`callAsync()`** klasy **`MySampleService`** przekazując w argumencie metody kolejny numer zadania

Po zakończeniu działania pobierzemy wynik działania metody callAsync() za pomocą metody **`get()`**

W naszym przykładzie na samym końcu pobierzemy również całkowity czas wykonywania metody callAsync(). Mimo wywołania trzech metod gdzie każda z nich będzie działała przez około 500ms, wykonanie ich w wielu wątkach jednocześnie sprawi że całkowity czas wykonania metody executeTasks() naszego kontrolera również powinien wynieść około 500ms

Praktyczny przykład @Async

```
@RestController
public class MyAsyncController {
  @Autowired private MySampleService mySampleService;
  @RequestMapping(value = "/executeTasks", method = RequestMethod.GET)
  public String executeTasks() throws InterruptedException, ExecutionException {
    long start = System.currentTimeMillis();
    Future<Long> result1 = mySampleService.callAsync(1);
    Future<Long> result2 = mySampleService.callAsync(2);
    Future<Long> result3 = mySampleService.callAsync(3);
    System.out.println("Result 1 got: " + result1.get());
    System.out.println("Result 2 got: " + result2.get());
    System.out.println("Result 3 got: " + result3.get());
    long end = System.currentTimeMillis() - start;
    return "Time to finish tasks: " + end;
  }
}
```

Praktyczny przykład @Async

```
@RestController
public class MyAsyncController {
  @Autowired private MySampleService mySampleService;
  @RequestMapping(value = "/executeTasks", method = RequestMethod.GET)
  public String executeTasks() throws InterruptedException, ExecutionException {
    long start = System.currentTimeMillis();
    Future<Long> result1 = mySampleService.callAsync(1);
    Future<Long> result2 = mySampleService.callAsync(2);
    Future<Long> result3 = mySampleService.callAsync(3);
    System.out.println("Result 1 got: " + result1.get());
    System.out.println("Result 2 got: " + result2.get());
    System.out.println("Result 3 got: " + result3.get());
    long end = System.currentTimeMillis() - start;
    return "Time to finish tasks: " + end;
  }
}
```

@Autowired - odpowiedzialne za Dependency Injection

Zadania

@Scheduled

@Scheduled

Za pomocą adnotacji **@Scheduled** oznaczamy metody za pomocą których będziemy mogli planować zadania

Przy użyciu @Scheduled należy pamiętać o tym że metoda oznaczona tą adnotacją:

- powinna mieć ustawiony typ void jako return type
- nie może przyjmować żadnych parametrów

Włączenie wsparcia dla @Scheduled

Aby możliwe było wykorzystanie adnotacji **@Scheduled** w naszych aplikacjach Spring należy użyć adnotacji **@EnableScheduling** podobnie jak to było w przypadku adnotacji **@EnableAsync**

```
@Configuration
@EnableScheduling
public class SpringConfig {
  // class body
}
```

Planowanie zadań z @Scheduled i fixedDelay

Adnotację **@Scheduled** oraz parametr **fixedDelay** możemy wykorzystać do ustawienia opóźnienia pomiędzy zakończeniem wykonania jednego zadania a rozpoczęciem kolejnego

Zadanie, które ma zostać uruchomione, zawsze będzie czekało aż poprzednie zadanie się zakończy. Rozwiązanie to powinno się stosować właśnie, gdy konieczne jest zakończenie jednego zadania, zanim kolejne zostanie uruchomione

```
@Scheduled(fixedDelay = 2000)
public void scheduleTaskWithFixedDelay() {
  SimpleDateFormat dateFormat = new SimpleDateFormat("HH:mm:ss");
  System.out.println("Working task - " + dateFormat.format(new Date()));
  try {
    TimeUnit.SECONDS.sleep(5);
  } catch (InterruptedException ex) {
    throw new IllegalStateException(ex);
  }
```

Planowanie zadań z @Scheduled i fixedDelay

Uruchamiając stworzoną przez nas metodę **`scheduleTaskWithFixedDelay()`** każde zadanie wykona się w 5 sekund a od zakończenia jednego zadania do rozpoczęcia drugiego muszą minąć 2 sekundy

Tym sposobem różnica między kolejnymi wykonaniami zadań będzie wynosiła 7 sekund

```
# Sample result
Working task - 10:30:01
Working task - 10:30:08
Working task - 10:30:15
....
....
```

Planowanie zadań z @Scheduled i fixedRate

Planując wykonywanie zadań z adnotacją **`@Scheduled`** i parametrem **`fixedRate`** należy zauważyć że rozpoczęcie kolejnego zadania nie jest uzależnione od zakończenia poprzedniego zadania. Wykorzystanie takiej metody powinno być używane gdy każde wykonanie zadania jest niezależne od siebie

```
@Scheduled(fixedRate = 2000)
public void scheduleTaskWithFixedRate() {
  SimpleDateFormat dateFormat = new SimpleDateFormat("HH:mm:ss");
  System.out.println("Working task - " + dateFormat.format(new Date()));
}
```

```
# Sample result
Working task - 10:26:58
Working task - 10:27:00
Working task - 10:27:02
....
```

Planowanie zadań z @Scheduled i initialDelay

Planując wykonywanie zadań korzystając z adnotacji **`@Scheduled`** i parametru **`initialDelay`** zadanie będzie wykonane po czasie określonym w parametrze initialDelay

Użycie parametru initialDelay jest pomocne gdy początkowo musi zostać wykonana jakaś konfiguracja lub inna operacja dla zadania które ma zostać uruchomione

```
@Scheduled(fixedRate = 2000, initialDelay = 5000)
public void scheduleTaskWithInitialDelay() {
  SimpleDateFormat dateFormat = new SimpleDateFormat("HH:mm:ss");
  System.out.println("Working task - " + dateFormat.format(new Date()));
}
```

```
# Sample result (Server started at 10:48:46)
Working task - 10:48:51
Working task - 10:48:53
Working task - 10:48:55
....
```

Planowanie zadań z użyciem wyrażeń Cron

Czasami metody opisane wcześniej nie są wystarczające do zaplanowania wykonywania zadań dlatego wtedy można skorzystać z elastyczności wyrażeń Cron dlatego warto zapoznać się dokładniej z dokumentacją tego mechanizmu

```
@Scheduled(cron = "0 * * * * ?")
public void scheduleTaskWithCronExpression() {
  SimpleDateFormat dateFormat = new SimpleDateFormat("HH:mm:ss");
  System.out.println("Working task - " + dateFormat.format(new Date()));
}
```

```
# Sample result
Working task - 11:03:00
Working task - 11:04:00
Working task - 11:05:00
...
```

Zmiana domyślnego ScheduledExecutor

Domyślnie wszystkie metody oznaczone przez **`@Scheduled`** wykonywane są w zbiorze wątków o wielkości jeden co może być czasami mało efektywny

Można utworzyć swój własny zbiór wątków poprzez implementację interfejsu **`SchedulingConfigurer`** a ostatnią rzeczą do zrobienia jest nadpisanie megody **`configureTasks()`**

Zmiana domyślnego ScheduledExecutor

Przykład implementacji interfejsu **`SchedulingConfigurer`** i nadpisania metody **`configureTasks()`**

```
@Configuration
public class SchedulerConfig implements SchedulingConfigurer {
  @Override
  public void configureTasks(ScheduledTaskRegistrar scheduledTaskRegistrar) {
    ThreadPoolTaskScheduler thPoolTaskScheduler = new ThreadPoolTaskScheduler();
    thPoolTaskScheduler.setPoolSize(10);
    thPoolTaskScheduler.setThreadNamePrefix("my-scheduled-task-pool-");
    thPoolTaskScheduler.initialize();
    scheduledTaskRegistrar.setTaskScheduler(thPoolTaskScheduler);
  }
}
```

Zadania

MVC i @Async

MVC i @Async

Częstym przypadkiem który występuje podczas pracy z MVC to wykonywanie pewnych operacji które zajmują dużo czasu jak np. pobieranie czy zapisywanie danych.

Z pomocą przychodzi wielowątkowość, za pomocą której możemy przyspieszyć wykonywanie poszczególnych czynności i samo wyświetlanie wyników na stronie.

W naszym przypadku wykonamy aplikację, która wykona operacje symulujące długotrwałe działanie, a dzięki wielowątkowości zobaczymy, jak można osiągnąć oczekiwany rezultat.

MVC i @Async

Na początek tworzymy standardową aplikację w Spring Boot oraz dodajemy poznaną wcześniej adnotację **@EnableAsync**:

```
@SpringBootApplication
@EnableAsync
public class DemoMvcApplication {
  public static void main(String[] args) {
    SpringApplication.run(DemoMvcApplication.class, args);
  }
}
```

MVC i @Async

Tworzymy klasę której metoda **`longRunning()`** symuluje długie działanie, np. zapisywanie do bazy danych lub łączenie z jakimś zasobem:

```
@Component
public class ExampleService {
  @Async
  public void longRunning() {
    long time = System.currentTimeMillis();
    try {
      Thread.sleep(2000);
    } catch (InterruptedException e) { /* TO DO */ }
    long end = System.currentTimeMillis() - time;
    System.out.println(LocalDateTime.now() +
        " - Task completed after: " + end + " ms");
  }
}
```

MVC i @Async

Na koniec pozostaje utworzyć kontroler który trzykrotnie wykonuje metodę **`longRunning()`**

```
@Controller
public class SampleAsyncController {
        @Autowired
        public ExampleService exampleService;
        @RequestMapping(value = "/controller", method = RequestMethod.GET)
        @ResponseBody
        public String foo() {
                long time = System.currentTimeMillis();
                exampleService.longRunning();
                exampleService.longRunning();
                exampleService.longRunning();
                long end = System.currentTimeMillis() - time;
                return LocalDateTime.now() + " - Response after: " + end + " ms";
        }
}
```

MVC i @Async

Po uruchomieniu całej aplikacji uzyskamy wynik bardzo podobny do poniższego - warto zwrócić uwagę na czas wykonania poszczególnych zadań:

```
2017-12-07 00:21:44.277 - Response after: 20 ms
2017-12-07 00:21:46.274 - Task completed after: 2000 ms
2017-12-07 00:21:46.274 - Task completed after: 2000 ms
2017-12-07 00:21:46.275 - Task completed after: 2001 ms
```

Jak widzimy najpierw praktycznie od razu(po 20ms) pojawił się napis "Response after" który również zostanie wypisany w przeglądarce po wejściu na adres kontrolera.

Cała aplikacja wykonała się w około 2 sekundy mimo, że zostały wywołane trzy każda trwająca po 2 sekundy metody. Powodem tak szybkiego wykonania działania aplikacji jest wykorzystanie wielowątkowości.

MVC i @Async

Klasyczny przykład nie wykorzystujący wielowątkowości miałby podobny kod do naszej aktualnej aplikacji z wyjątkiem adnotacji **`@Async`** w komponencie oraz adnotacji **`@EnableAsync`**.

Tak uruchomiony kod byłby o wiele mniej wydajny. Poniżej przedstawione są rezultaty wykonania aplikacji po wspomnianych zmianach:

```
2017-12-07 00:20:55.429 - Task completed after: 2007 ms
2017-12-07 00:20:57.431 - Task completed after: 2000 ms
2017-12-07 00:20:59.431 - Task completed after: 2000 ms
2017-12-07 00:20:59.431 - Response after: 6007 ms
```

MVC i @Async

W przeciwieństwie do poprzedniego przypadku na wyświetlenie napisu "Response after" a tym samym na przekazanie komunikatu do przeglądarki musielibyśmy czekać prawie 6 sekund.

Powodem tego jest fakt, że każde z zadań wywoływanych w kontrolerze wykonywałoby się jedno po drugim bez użycia wielowątkowości.

Tym samym widzimy jak niewielka zmiana może dać wielką różnicę w działaniu. Poza szybszym wykonywaniem samej aplikacji również użytkownik szybciej otrzyma w przeglądarce wyniki których oczekuje.

Zadania

KONIEC