Programowanie reaktywne

Vert.x

Programowanie reaktywne

Przetwarzanie danych

- Asynchroniczne
- Nieblokujące

Programowanie reaktywne

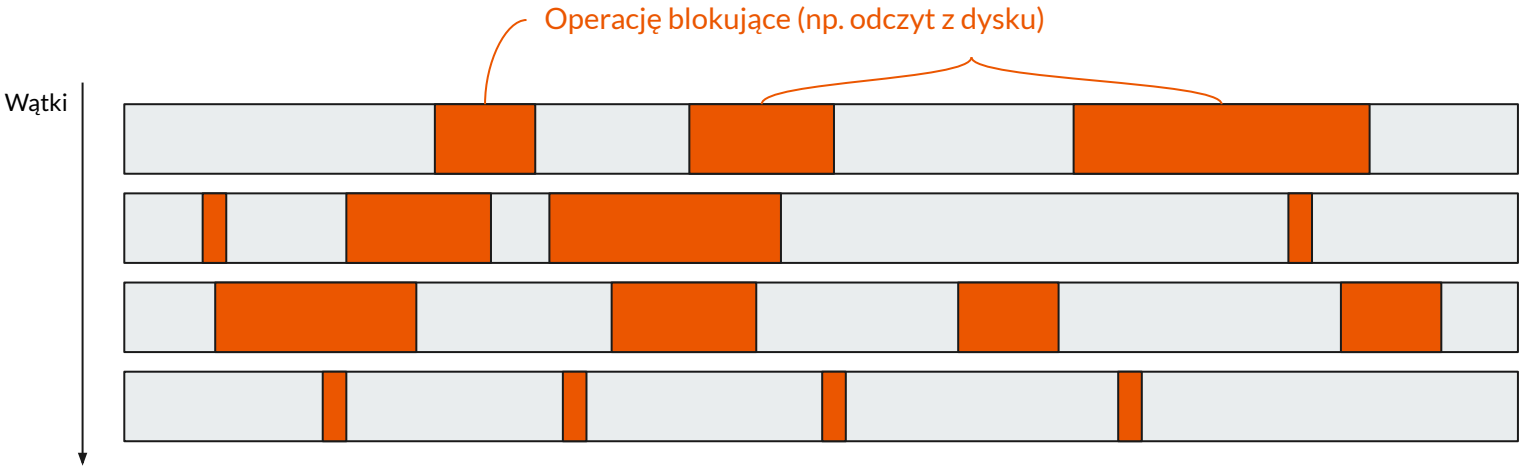
- Bazuje na zdarzeniach (*events*)
- Zdarzenia generowane są przez ***publisher***
- A obsługiwane przez ***handlery***

Przykłady zdarzeń

- Koniec odliczania minutnika
  - Uruchom kod co jakiś czas
- Dane zostały przesłane do *socketu*
  - Procesowanie zdarzeń systemowych
- Dane zostały odczytane z dysku
  - Logowanie odczytów plików
- Wystąpił wyjątek w działaniu programu
  - Wysyłanie powiadomień do administratorów
- Serwer HTTP otrzymał żądanie (*request*)
  - Generowanie odpowiedzi (*response*)

Dlaczego?

- Blokujące operacje powodują tworzenie wielu *stosunkowo* zasobożernych wątków



Co to jest operacja blokująca?

W porównaniu do szybkości działania procesora (ilości obliczeń w jednostce czasu) dostęp do dysku twardego (a nawet pamięci RAM) jest **bardzo** wolny.

Każda operacja wymagająca odczytania/zapisy danych (np. w bazie danych, lub z pliku) powoduje **zatrzymanie** wykonywania operacji na procesorze przez wątek czekający na zakończenie operacji blokującej (blocking I/O)

Jeśli chcemy wykonać jakąś akcję w momencie czekania na zakończenie operacji blokującej musimy stworzyć **nowy wątek**

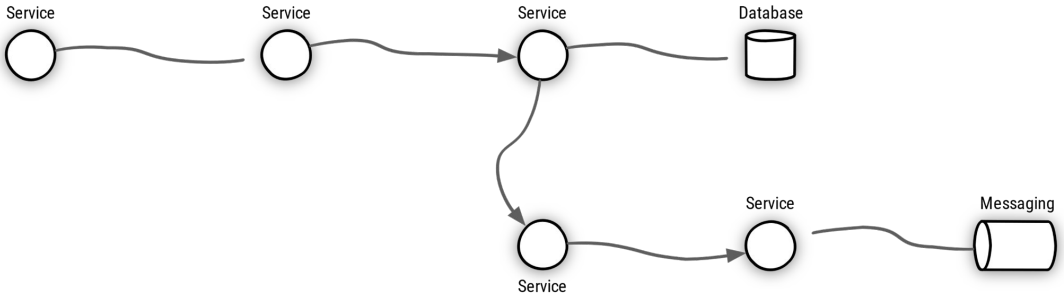
Dlaczego cd .

- Tworzenie wątku jest stosunkowo kosztowne: trwa kilka milisekund i powoduje zużycie ~1MB RAMu
- Każdy wątek musi być obsłużony przez **Scheduler**, który wybiera kiedy i który wątek dostanie czas procesora
- Każdy pracujący system przetwarza tysiące wątków
- Wątek, który czeka zakończenie operacji blokującej efektywnie nie robi nic - a mógłby przecież robić coś innego
- Zmiana kontekstu na inny wątek również trwa stosunkowo długo

Reaguj, nie czekaj!

Aplikacje reaktywne łatwo się **skalują** oraz są bardzo **elastyczne** jeśli pojawiają się awarie. Takie aplikacje w optymalny sposób wykorzystują zasoby systemowe oraz utrzymują wspaniałe **latancy** (w porównaniu do normalnych blokujących aplikacji)

Upraszczając, standardowe aplikacje poradzą sobie z dziesiątkami tysięcy wątków, zaś te reaktywne mogą obsłużyć setki i miliony współbieżnych połączeń



Vert.x

Co daje nam Vert.x?

Możliwość przetworzenia **większej** ilości zapytań i mniejszego zużycia zasobów w stosunku do standardowych frameworków (np. Spring)

Pozwala po prostu skupić się na pisaniu kodu - nie ma określonych reguł jak to robią **frameworki**

Jest biblioteką, a nie **frameworkiem** - łatwo wbudować go w istniejące już aplikacji

Wspiera wiele języków programowania: Java, JavaScript, Ruby, Groovy, Kotlin, Python, Scala...

Operacje nieblokujące w Vert.x

Jeśli wynik działania żądania może być zwrócony natychmiast to jest po prostu zwracany.

W przeciwnym przypadku (np. odczyt danych z dysku) zwraca tzw. **handler**, który pozwala dostać rezultat, gdy ten już będzie dostępny.

To znacząco zmniejsza ilość wątków, które potrzebne są do zapewnienia współbieżności (np. obsługi dużej ilości zapytań)

Fluent API

Jest to sposób pisania kodu i tworzenia łańcuchów wywołań kolejnych metod.

```
users.stream()
     .filter(user -> user.age() > 18)
     .filter(user -> user.name().startsWith("A"))
     .map(user -> String.format("Mam na imię %s i jestem pełnoletni(a)", user.name()))
     .forEach(System.out::println);
```

Vert.x wykorzystuje **fluent API** co jest dość przyjemne w tworzeniu skomplikowanego kodu

Jak to działa?

Pobierz informacje o pliku (to generuje jego odczyt - operacja blokująca)

Nasz kod nie blokuje wywołania, a po prostu definiuje handler, który obsłuży, gdy informacje o pliku są dostępne

```
FileSystem fs = vertx.fileSystem();

Future<FileProps> future = fs.props("pom.xml ");

future.onComplete((AsyncResult<FileProps> ar) -> {
  if (ar.succeeded()) {
    FileProps props = ar.result();
    System.out.println("File size = " + props.size());
  } else {
    System.out.println("Failure: " + ar.cause().getMessage());
  }
});
```

Zaczynamy: **obiekt** Vertx

Główny obiekt, od którego właściwie wszystko się zaczyna. Pozwala na tworzenie:

- klientów
- serwerów
- event bus
- wszystko inne…

```
Vertx vertx = Vertx.vertx();
```

Ważne: w zdecydowanej większości przypadków wystarczy tylko jeden obiekt Vertx na całą aplikację

Verticles

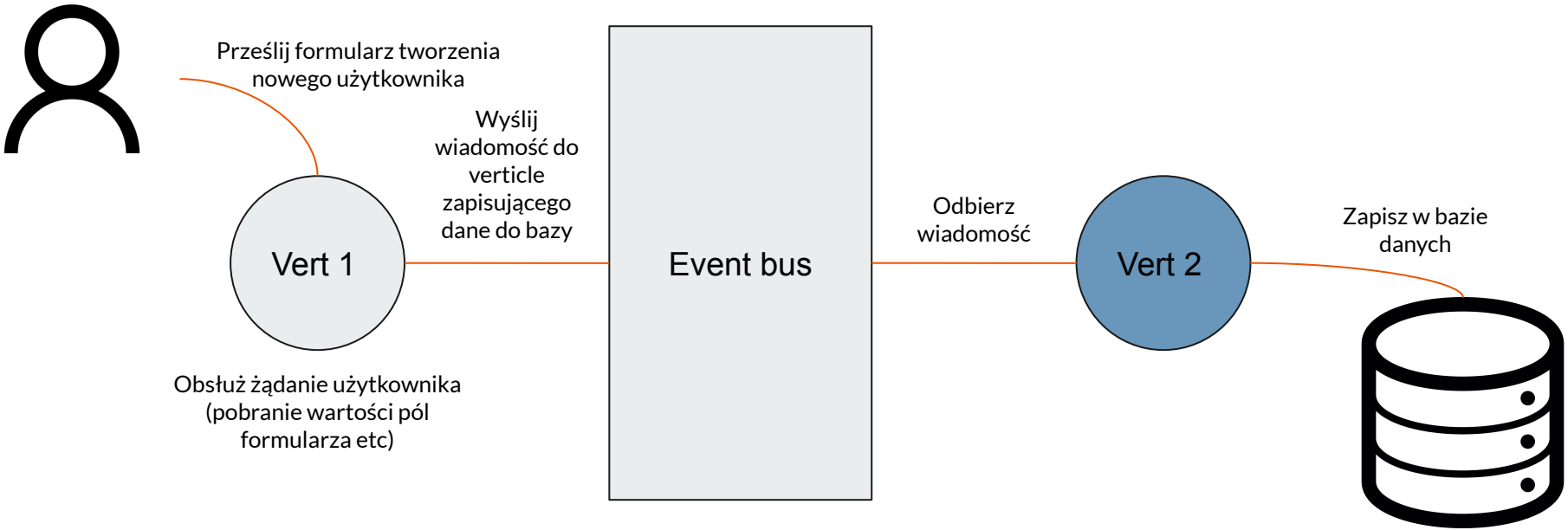
Vert.x pozwala w łatwy sposób **skalować** za pomocą tzw. **verticles**. Innymi słowy, są one klockami budującymi naszą aplikację.

Każdy Verticle odpowiedzialny jest za jakąś część aplikacji. Może być obsługa zapytań HTTP (serwer HTTP), zapis informacji do pliku, odczyt/zapis danych do bazy.

Verticle porozumiewają się ze sobą za pomocą **event bus**. Przykład: verticle obsługujący zapytanie użytkownika, może wysłać wiadomość do verticle zapisującego informacje w bazie danych

Verticle mogą być pisane we wszystkich obsługiwanych językach

Koncept


Prześlij formularz tworzenia nowego użytkownika
Wyślij wiadomość do verticle zapisującego dane do bazy
Vert 1
Obsłuż żądanie użytkownika (pobranie wartości pól formularza etc)
Event bus
Odbierz wiadomość
Vert 2
Zapisz w bazie danych

Tworzymy Verticle

Każdy Verticle rozszerza klasę **AbstractVerticle**

```
public class MyVerticle extends AbstractVerticle {

    // Called when verticle is deployed
    public void start() {
    }

    // Optional - called when verticle is undeployed
    public void stop() {
    }
}
```

Typy Verticli

Standardowe

To takie, które wykonywane są w głównej pętli programu. Są asynchroniczne i nie powinny wykonywać operacji blokujących bo zatrzymują wykonanie wszystkich innych verticli

Workery

Wykonywane są poza główną pętlą programu (jako oddzielne wątki). Dzięki temu powinny być używane, gdy już musimy wykonać operacją blokującą.

Dodawanie Verticli do aplikacji

Stworzenie odpowiedniej klasy to jedno. Aby faktycznie użyć naszego nowego Verticla musimy go wdrożyć (*deploy*).

```
Verticle myVerticle = new MyVerticle();
vertx.deployVerticle(myVerticle);

vertx.deployVerticle("com.mycompany.MyOrderProcessorVerticle");

// Deploy a JavaScript verticle
vertx.deployVerticle("verticles/myverticle.js");

// Deploy a Ruby verticle verticle
vertx.deployVerticle("verticles/my_verticle.rb");
```

Zadanie 1

Prosty serwer HTTP

Uruchom klasę **SimpleWebServer** z GitHuba:
https://github.com/dzon2000/wse/tree/main/vertx

Otwórz przeglądarkę i wejdź pod adres:
http:/localhost:8080

Zadanie a:
Zmień port na 9999

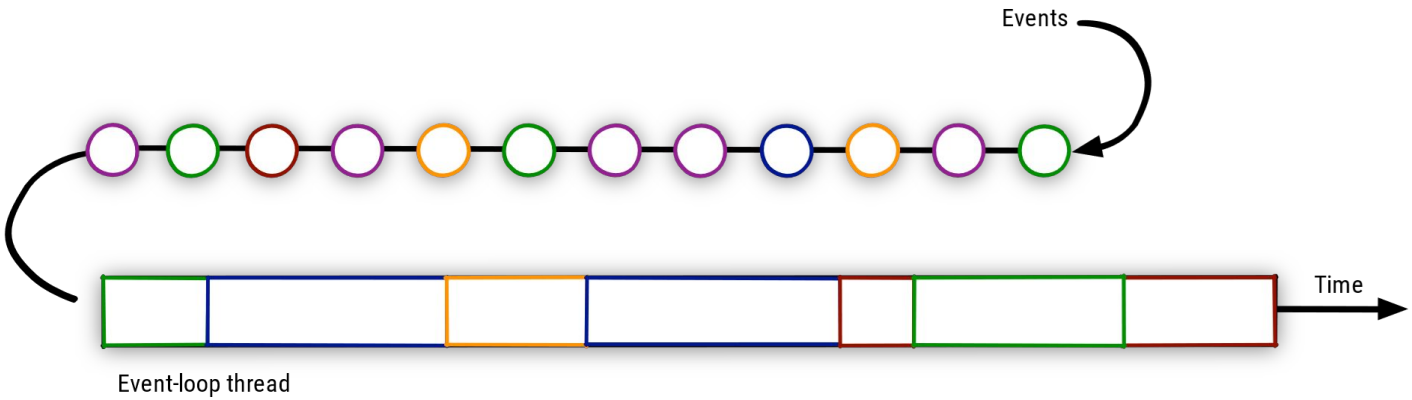
Zadanie b:

Obsłuż path **/wse** i wyświetl obecną godzinę

Podpowiedź:

```
if (req.path().equals("/wse"))
```

Event Bus



Obsługa asynchroniczna

Współbieżna obsługa żądań możliwa jest dzięki operacjom **asynchronicznym**. Zamiast blokować wątek podczas operacji wejścia/wyjścia (I/O), zostanie on użyty do wykonania kolejnego zadania czekającego w kolejce.

W takim przypadku możemy obsłużyć wiele zadań jednocześnie, bo wątki czekające na operacje blokujące, czy rezultat długiego procesu mogą być użyte do obsługi innych zadań.

Event Bus

Pozwala na komunikację pomiędzy różnymi częściami aplikacji. Niezależnie od wybranego języka (np. możemy wysłać wiadomość z Verticle napisanego w Javie, do takiego napisanego w JavaScript) oraz instancji Vert.x. Działa na zasadzie

- **publish/subscribe** - wiadomość od nadawcy trafi do wszystkich, którzy zapisali się na dany adres (podobnie jak RSS, lub protokół MQTT)
- **point-to-point** - wysłanie konkretnej wiadomości od nadawcy do określonego odbiorcy
- **request-response** - dodatkowo możemy wysłać odpowiedź

Adresy

Wiadomości w **event bus** wysyłane są na **adresy**

Adres jest Stringiem, ale powinien być nazwą domenową (wynika to z konwencji, a nie ograniczeń Vert.x)

```
var address = "v1.core.httpserver.healtcheck";
```

Handlery

Wiadomości obsługiwane są przez **handlery**. Możemy zarejestrować **handler** pod **adres**

Wiele **handlerów** może być zapisanych do jednego **adresu**

Jeden **handler** może być zapisany do wielu **adresów**

```
EventBus eb = vertx.eventBus();

eb.consumer("car.hud.notifications", message -> {
  System.out.println("New notification: " + message.body());
});
```

Wysyłanie wiadomości

Przesyłanie wiadomości na **event bus** odbywa się na kilka sposobów:

- publish - wyślij do wszystkich subskrybentów

```
eventBus.publish("car.hud.notifications", "Change Oil!");
```

- send - wyślij do jednego. Wybrany zostanie jeden subskrybent na zasadzie algorytmu round-robin

```
eventBus.send("car.hud.notifications", "Change Oil!");
```

Żądania

Jeśli subskrybent może wysłać odpowiedź, handler może takową obsłużyć.

```
eventBus.request("car.engine.params", "oil-level", reply -> {
    System.out.println((String) reply.result().body());
});
```

Event bus - podsumowanie