Programowanie wielowątkowe w Javie

Łukasz Andrzejewski

2 Czym jest wątek?

- Przepływ sekwencji sterowania w programie, posiadający własny stos wywołań
- Wątki w ramach procesu mają dostęp do wspólnej przestrzeni adresowej
- Aplikacja rozpoczyna działanie w wątku głównym (ang. main thread), który może stworzyć i uruchomić kolejne wątki
- Wątki konkurują o dostęp do fizycznych zasobów komputera, dając możliwość i/lub iluzję pracy równoległej

3 Tworzenie wątku

- W celu uruchomienia nowego wątku należy:
  - zaimplementować interfejs Runnable - instrukcje umieszczone w metodzie run() będą wykonywane w czasie pracy wątku
  - stworzyć instancję wątku (klasa Thread) i zainicjajizować ją obiektem typu Runnable
  - uruchomić wątek przy użyciu metody start()

4 Wykonywanie wątków

Programowanie wielowątkowe w Javie - wybrane zagadnienia

- Model wywłaszczeniowy zakłada, że wątek jest w stanie Running, dopóki mechanizm szeregowania wątków nie zdecyduje o tym, że inny wątek powinien uzyskać dostęp do zasobów
- Java udostępnia metody statyczne pozwalające na sterowanie wykonaniem wątków:
  - yield() - aktualny wątek zrzeka się zasobów, Scheduler wybiera z puli wątków typu Runnable inny wątek, który zajmie jego miejsce
  - sleep() – uśpienie aktualnego wątku na określony czas

5 Metody sterujące wątkami

Programowanie wielowątkowe w Javie - wybrane zagadnienia

- join() – powoduje zatrzymanie wykonania bieżącego wątku do czasu zakończenia pracy innego wątku lub upływu określonego okresu czasu
- setPriority() – pozwala na ustalenie priorytetu wątku
- setDeamon() – powoduje przejście wątku do stanu demonicznego
- isAlive() – zwraca informację czy wykonano metodę start() i czy wątek nie zakończył jeszcze swojego działania
- stop() – nagłe zatrzymanie pracy wątku - niezalecane
- interrupt() – bezpieczne zatrzymanie wątku oparte o ustawienie flagi

6 Ochrona współdzielonych danych

Programowanie wielowątkowe w Javie - wybrane zagadnienia

- W celu ochrony integralności danych można wykorzystać wbudowany mechanizm monitora do obiektu
- Monitor to znacznik, który w danej chwili może być przejęty tylko przez jeden wątek (poprzez realizację bloku lub metody synchronizowanej)
- Jeśli wątek posiadający monitor zostanie wywłaszczony nie oddaje on monitora
- Należy ograniczać liczbę instrukcji synchronizowanych - nie są one wykonywane współbieżnie, dlatego mogą stanowić "wąskie gardło" aplikacji. Dodatkowo powstają narzuty związane z przejęciem/zwrotem monitora

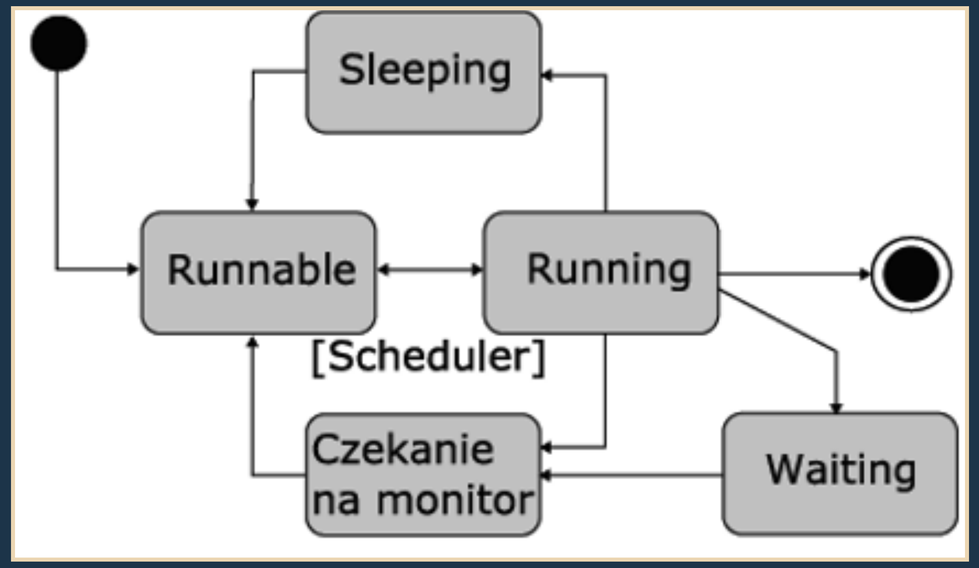
7 Zakleszczenie

- Występuje gdy wątek posiadający monitor do obiektu oczekuje dostępu do obiektu, którego monitor jest zajęty przez inny wątek, który oczekuje na dostęp do monitora posiadanego przez wątek pierwszy
- Sytuacja zakleszczenia często jest trudna do wykrycia, ponieważ może występować tylko przy pewnej zależności czasowej
- Poszukując potencjalnego miejsca zakleszczenia należy rozpatrzeć wszystkie wywołania wymagające pobrania więcej niż jednego monitora

8

Koordynacja wątków

Programowanie wielowątkowe w Javie - wybrane zagadnienia



9 Java Memory Model

Programowanie wielowątkowe w Javie - wybrane zagadnienia

- określa sposób dostępu i wykorzystania rzeczywistej pamięci RAM m.in. mówi o tym jak i kiedy różne wątki mają dostęp do wartości zapisywanych przez wątki

Internal Java Memory Model - Stack

- każdy wątek ma własny stos zawierający informacje o wywołanych metodach (call stack) oraz o zmiennych lokalnych utworzonych w ramach ich wywołań
- wątek ma dostęp wyłącznie do własnych zmiennych (nawet jeśli różne wątki uruchamiają ten sam kod)

Internal Java Memory Model - Heap

- sterta zawiera wszystkie obiekty utworzone w ramach aplikacji (niezależnie od tego jaki wątek je utworzył lub tego czy jest to zmienna instancyjna czy lokalna dla metody)
- sterta jest dostępna dla wszystkich wątków (mogą współdzielić dostęp do obiektów i ich zmiennych instancyjnych oraz statycznych, ale nie lokalnych)

Internal Java Memory Model - lokalizacja w pamięci

- obiekty zapisywane są na stercie
- lokalne zmienne typów prymitywnych i referencje są przechowywane na stosie i kopiowane w czasie przekazywania do innego wątku (każdy działa na własnej kopii)
- zmienne instancyjne oraz zmienne statyczne przechowywane są na stercie wraz z obiektem (nawet jeśli są one prymitywne) i definicją klasy

13 Klasyczny, sprzętowy model pamięci

- komputery mogą posiadać więcej niż jeden procesor, z których każdy może mieć wiele rdzeni co umożliwia realne działanie wielu wątków
- każdy procesor zawiera zbiór rejestrów (pamięć CPU) oraz własną pamięć podręczną (często wielopoziomową)
- operacje na rejestrach są dużo bardziej wydajne niż operacje na pamięci głównej (RAM)
- operacje na pamięci podręcznej CPU są wolniejsze niż na rejestrach, ale szybsze niż na pamięci głównej (RAM)
- dostęp do danych w RAM odbywa się najczęściej pośrednio (przez pamięć cache i rejestry)

Sprzętowy model pamięci vs. Java Memory Model

- sprzętowy model pamięci nie rozróżnia stosu i sterty (oba obszary są częścią RAM)
- fragmenty stosu i sterty mogą być przechowywane na poziomie rejestrów i pamięci podręcznej CPU
- przechowywanie zmiennych i obiektów w różnych obszarach pamięci może prowadzić do problemów
  - widoczność zmian stanu współdzielonych zmiennych
  - race condition przy odczycie, wykorzystaniu i zapisie współdzielonych zmiennych

Słowo kluczowe volatile

- jeśli jeden wątek zmienia wartość zmiennej balance, a drugi dokonuje jej odczytu, volatile gwarantuje spójność (widoczność zmian). W przypadku zmiany wartości przez większą ilość wątków wymagana jest synchronizacja

```
public class Account {

   volatile int balance = 0;

}
```

Słowo kluczowe volatile w Java 5+

- jeśli wątek A pisze do zmiennej oznaczonej volatile, a wątek B ją czyta, wtedy wszystkie zmienne widoczne dla wątku A przed zapisem będą także widoczne dla wątku B (synchronizacja z „pamięcią główną)
- jeśli wątek A czyta zmienną oznaczoną volatile, wtedy wszystkie zmienne widoczne dla wątku A będą odświeżone z „pamięci głównej”

17 Słowo kluczowe volatile vs. synchronizacja

- volatile może być stosowane na poziomie zmiennych prymitywnych, synchronizacja (synchronized, typy atomowe) może odnosić się jedynie do typów referencyjnych
- volatile nie jest związane z blokadą (zamkiem), w związku z tym nie nadaje się do operacji typu atomic read-update-write

Spójność na poziomie aplikacji wielowątkowych

- Mutual Exclusion - tylko jeden wątek może wykonywać sekcję krytyczną
- Visibility - zmiany realizowane przez jeden wątek są widoczne dla innych wątków
- synchronizacja z użyciem blokad spełnia oba warunki kosztem wydajności
- volatile - spełnia tylko drugi warunek, koszty wydajnościowe są mniejsze ale nadal występują (praca na „pamięci głównej”, brak wybranych optymalizacji)

Słowo kluczowe volatile, a model pamięci

- Java gwarantuje atomowość operacji odczytu i zapisu zmiennych dla typów o rozmiarze do 32-bitów
- w przypadku typów 64-bitowych, takich jak long odczyt/zapis zmiennej może zostać podzielony na dwie operacje 32-bitowe, co z kolei może wpłynąć na wynik działania programu współbieżnego
- zastosowanie modyfikatora volatile gwarantuje, że do tego nie dojdzie

Słowo kluczowe volatile, a optymalizacje

- JVM oraz CPU mogą dokonywać zmian w kolejności wykonywania instrukcji kodu w celu jego optymalizacji (jeśli ich sens/semantyka nie ulega zmianie)
- w celu zagwarantowania bezpieczeństwa tych optymalizacji Java stosuje zasadę happens-before
  - zapis/odczyt do zmiennych nie może być przeniesiony tak aby następował po zapisie do zmiennej volatile jeśli wcześniej występował przed nim (co nie wyklucza sytuacji odwrotnej)
  - zapis/odczyt do zmiennych nie może być przeniesiony tak aby następował przed odczytem zmiennej volatile jeśli wcześniej występował po nim (co nie wyklucza sytuacji odwrotnej)

Priorytet wątku

- scheduler w Javie (menadżer wątków) przydziela zasoby procesora bazując na priorytecie wątku stosując jednocześnie time slicing
- każdy wątek w Javie posiada priorytet w zakresie 1-10 (standardowo 5)
- każdy nowo utworzony wątek dziedziczy priorytet po wątku który go stworzył
- priorytet wątku może zostać zmieniony programistycznie w dowolnym momencie działania programu
- priorytety wątków w Javie są mapowane na priorytety w systemie operacyjnym, których gradacja może się różnić

Priority inversion

- zachodzi kiedy wątek o niższym priorytecie blokuje (synchronizacja, semafory itd.) dostęp do zasobu potrzebnego do działania wątku o wyższym priorytecie przy jednoczesnej „walce” o zasoby procesora z wątkami o średnim priorytecie

Priority Inversion, higher priority task (H) ends up waiting for middle priority task (M) when H is sharing critical section with lower priority task (L) and L is already in critical section. Effectively, H waiting for M results in inverted priority i.e. Priority Inversion

https://www.geeksforgeeks.org/priority-inversion-what-the-heck/
https://www.embedded.com/how-to-use-priority-inheritance/

Priority inheritance jako rozwiązanie Priority inversion

- tymczasowe podniesienie priorytetu wątku o niskim priorytecie, na czas wykonywania sekcji krytycznej tak, aby wątki o średnim priorytecie nie powodowały dodatkowego opóźnienia jej realizacji

Priority Inheritance, when L is in critical section, L inherits priority of H at the time when H starts pending for critical section. By doing so, M doesn't interrupt L and H doesn't wait for M to finish. Please note that inheriting of priority is done temporarily i.e. L goes back to its old priority when L comes out of critical section.

https://www.geeksforgeeks.org/priority-inversion-what-the-heck/

Kolekcje synchronizowane

- większość kolekcji w języku Java nie gwarantuje bezpieczeństwa bez użycia zewnętrznej synchronizacji
- w przypadku prostych/pojedynczych operacji kolekcje można opakować stosując statyczne metody z Collections np. synchronizedList(), synchronizedMap(). W rezultacie dostęp do metod kolekcji jest synchronizowany (obiektem synchronizacji jest instancja kolekcji)
- w przypadku operacji złożonych takich jak jednoczesny odczyt i zapis należy użyć innych mechanizmów np. blokady jawne lub niejawne

Kolekcje współbieżne

- synchronizacja kolekcji może znacząco wpływać na wydajność realizowanych operacji (zwłaszcza jeśli są one złożone np. iteracja po wszystkich elementach)
- kolekcje współbieżne są bezpieczne wielowątkowo, ale jednocześnie nie ograniczają możliwości wykonywania operacji przez wiele wątków np.
  - copy on write collections np. CopyOnWriteArrayList, CopyOnWriteArraySet
  - compare and swap (CAS) np. ConcurrentLinkedQueue, ConcurrentSkipListMap
  - oparte o blokady jawne np. ConcurrentHashMap, BlockingQueue
- powyższe kolekcje nie rzucają wyjątkiem ConcurrentModificationException

Wielowątkowość na poziomie GUI (Swing)

- operacje związane obsługą UI powinny realizowane się w specjalnym wątku (event dispatch thread - EDT), który gwarantuje bezpieczeństwo w środowisku wielowątkowym
- operacje związane z logiką biznesową, przetwarzaniem powinny być realizowane w tle (worker threads), zwykle z użyciem SwingWorker

Źródła spadku wydajności programów wielowątkowych

- nadmierna/niewłaściwa synchronizacja
- niewłaściwy sposób zarządzania wątkami (tworzenie, wykorzystanie, niszczenie)
- dobór niewłaściwych mechanizmów/implementacji w kontekście rozwiązywanego problemu
- nadmierna/niepoprawna/nieświadoma optymalizacja kodu
- niewłaściwe zarządzanie priorytetami

GC - strategie usuwania nieosiągalnych obiektów

- Mark-Sweep - oznaczanie obszarów zajmowanych przez nieosiągalne obiekty jako gotowych do alokacji
- Mark-Sweep-Compact - jak wyżej, z dodatkowym kompaktowaniem sterty
- Mark-Copy - kopiowanie osiągalnych obiektów do nowego miejsca na stercie

Algorytmy GC

- JVM oferuje standardowo 4 mechanizmy działania Garbage Collectora
  - Serial
  - Paraller
  - Mostly-Concurrent-Mark-And-Sweep (CMS)
  - G1

https://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html

Serial GC

- stosuje strategię Mark-Copy przy czyszczeniu młodej generacji oraz Mark-Sweep-Compact dla generacji starej
- czyszczenie starej generacji poprzedzone jest zawsze czyszczeniem młodej generacji (full gc)
- całość wykonywana jest na jednym wątki i powoduje wstrzymanie całej aplikacji
- sprawdza się dla aplikacji z niewielką stertą

Paraller GC

- działa jak Serial z tą różnicą że wykorzystuje wiele wątków
- czyszczenie starej generacji poprzedzone jest zawsze czyszczeniem młodej generacji (full gc)
- pozwala określić cele - maksymalny czas zatrzymania aplikacji, stosunek czasu pracy gc do pracy aplikacji oraz jaki powinien być max. rozmiar sterty
- sprawdza się dla aplikacji szybko zapełniających stertę, dla których możliwe są krótkie, sporadyczne przerwy w działaniu

Mostly-Concurrent-Mark-And-Sweep GC (deprecated)

- minimalizuje czas trwania pauz w działaniu programu kosztem użycia procesora
- stosuje strategię Mark-Copy przy czyszczeniu młodej generacji
- dla starej generacji wykonywane są następujące etapy
  - Initial Mark - wstrzymuje działanie aplikacji, identyfikuje GC roots pozwalających na rozpoczęcie procesu oznaczania żywych obiektów
  - Concurrent Mark - przeszukiwanie wcześniej oznaczonych drzew i oznaczanie obiektów jako żywych
  - Concurrent Preclean - przeszukiwanie drzew referencji w obiektach które uległy zmianie od czasu poprzedniej fazy
  - Concurrent Abortable Preclean - wydłużenie poprzedniej fazy, aby rozpocząć kolejną mniej więcej w połowie czasu między kolejnymi czystkami w edenie
  - Final Remark - wstrzymuje działanie aplikacji, ostateczne zlokalizowanie i oznaczenie żywych obiektów w starszej generacji
  - Concurrent Sweep - oznaczenie obszarów wolnych do ponownej alokacji

G1 GC

- lepsze wsparcie dla dużych stert (powyżej 4GB)
- wykorzystuje wiele wątków do skanowania sterty podzielonej wcześniej na mniejsze obszary w celu znalezienia wolnych wolnych obszarów (zwracanie dłużej ilości miejsca)

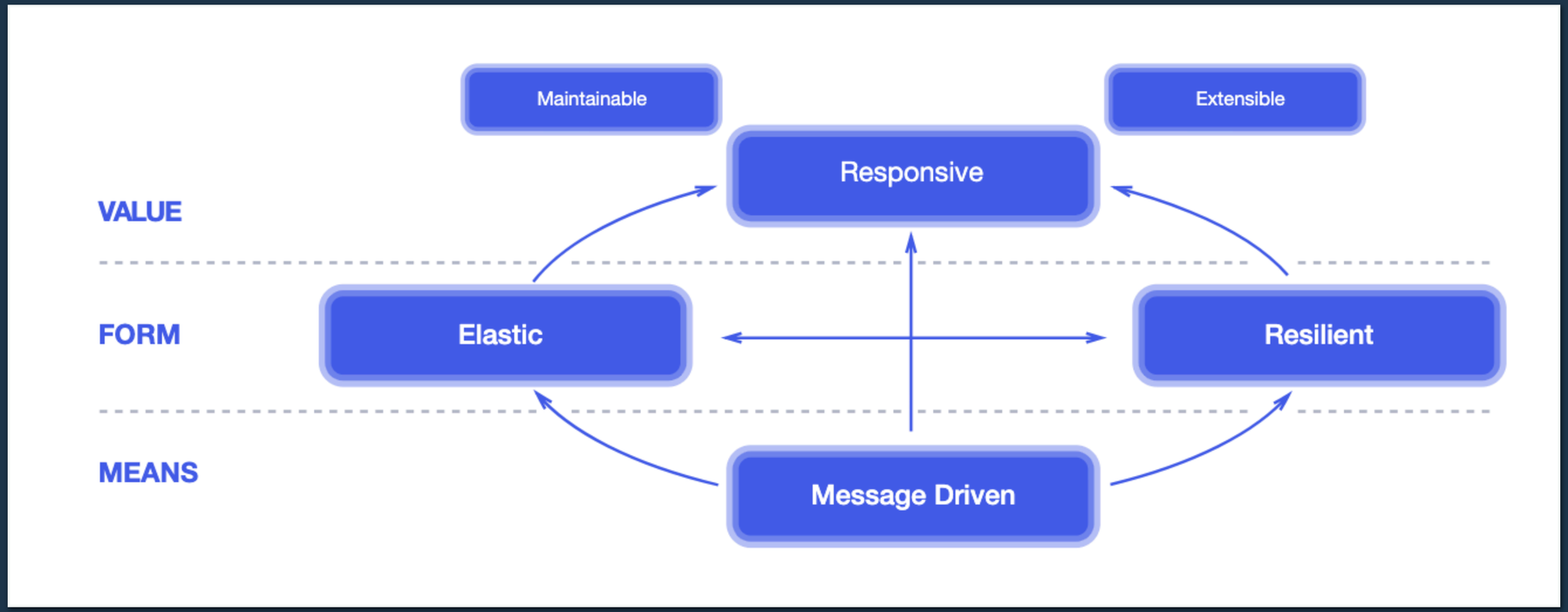
https://www.oracle.com/technetwork/tutorials/tutorials-1876574.html

Co oznacza termin Reactive?

- Reactive systems (architecture and design)
- Functional reactive programming (FRP)
- Reactive programming (declarative event-based)

Reactive systems (architecture and design)

Stwórz pierwszą aplikację z WebFlux / Reactor



https://www.reactivemanifesto.org

Functional reactive programming (FRP)

Stwórz pierwszą aplikację z WebFlux / Reactor

- Zdefiniowane ponad 20 lat temu w ramach pracy „Functional Reactive Animation” (Conal Elliott i Paul Hudak)
- FRP działa na wartościach, które zmieniają się w czasie w sposób ciągły, a w przypadku programowania reaktywnego działamy na dyskretnych wartościach emitowanych w czasie

http://conal.net/papers/icfp97

https://www.oreilly.com/radar/reactive-programming-vs-reactive-systems

http://vindum.io/blog/behaviors-and-streams-why-both

https://itnext.io/demystifying-functional-reactive-programming-67767dbe520b

Reactive programming (declarative event-based)

- Paradygmat programowania, który promuje asynchroniczne, nieblokujące, oparte na zdarzeniach podejście do przetwarzania danych

Reactive streams

„Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure. This encompasses efforts aimed at runtime environments as well as network protocols.”

https://www.reactive-streams.org

39

Reactive streams

Stwórz pierwszą aplikację z WebFlux / Reactor

```
public interface Publisher<T> {
    public void subscribe(Subscriber<? super T> s);
}
```

```
public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
```

```
public interface Subscription {
    public void request(long n);
    public void cancel();
}
```

```
public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {
}
```

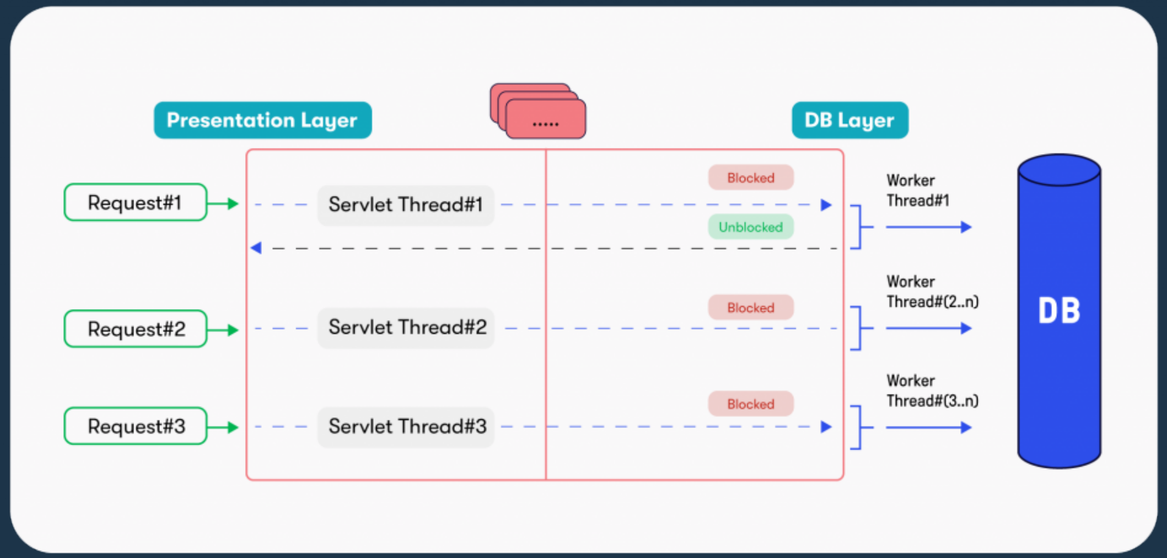
Przykłady implementacji: Reactor 3, RxJava 2, Akka Streams, Vert.x, Ratpack

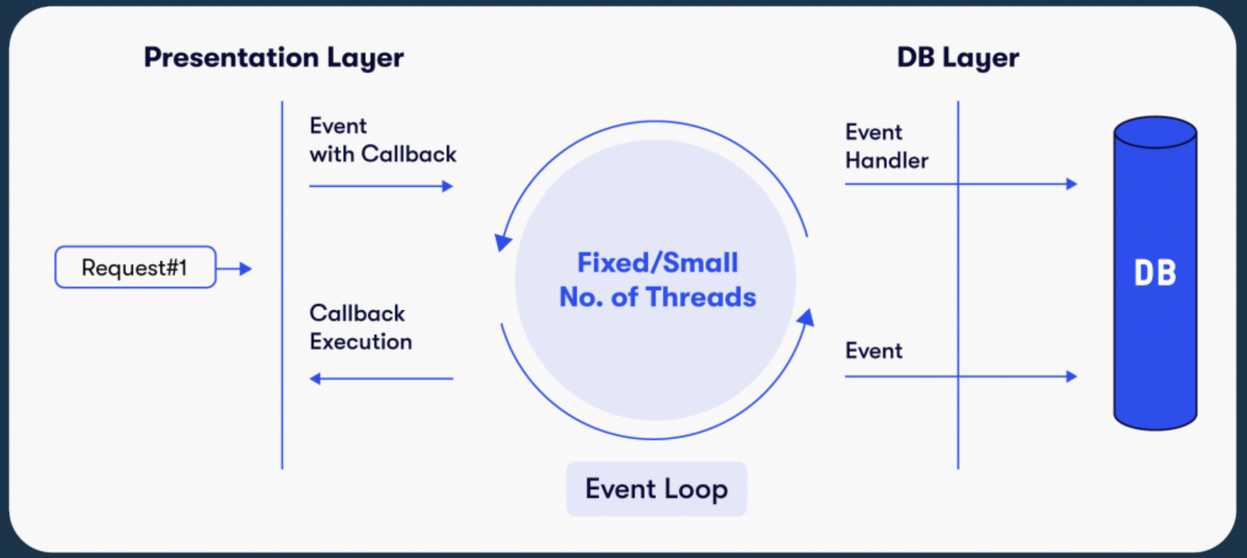
https://github.com/reactive-streams/reactive-streams-jvm/blob/v1.0.3/README.md

40

Blocking vs. non-blocking request processing

Stwórz pierwszą aplikację z WebFlux / Reactor





https://blog.moove-it.com/web-reactive-programming-spring-webflux