Notatki z Programowania Sieciowego

1. Wielowątkowość w Javie (Runnable i Callable)

1.1 Wstęp

Wielowątkowość polega na jednoczesnym wykonywaniu wielu zadań (wątków) w obrębie jednej aplikacji. Dzięki temu możliwe jest efektywniejsze wykorzystanie zasobów procesora i przyspieszenie działania programu – zwłaszcza gdy jeden z wątków jest blokowany (np. przez operacje I/O), w tym czasie inne wątki mogą się wykonywać.

1.2 Interfejs `Runnable`

- **Definicja**: `Runnable` to interfejs funkcyjny (posiadający jedną metodę abstrakcyjną `run()`).
- **Zasada działania**:
  - W klasie implementującej `Runnable` umieszczamy logikę w metodzie `run()`.
  - Następnie tworzymy obiekt klasy `Thread` przekazując w konstruktorze obiekt `Runnable`.
  - Wywołując `start()` na obiekcie `Thread`, faktycznie uruchamiamy nowy wątek.

Przykład kodu z `Runnable`:

```
class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("Wątek działa (Runnable)!");
    }
}

public class RunnableExample {
    public static void main(String[] args) {
        Thread thread = new Thread(new MyRunnable());
        thread.start();

        System.out.println("Metoda main została zakończona.");
    }
}
```

Schemat działania z wykorzystaniem `Runnable`:

```
+---------------+       +-----------------+
| MyRunnable    |       | Thread          |
| (implements   |       |                 |
| Runnable)     |       |                 |
+-------+-------+       +--------+--------+
        |                        |
        | run()                  |
        | - logika wątku         |
```

```
        |_____________________|
        |
        | Tworzenie wątku
        v
+-------+--------+
| new Thread(...)|
| .start()       |
+----------------+
        |
        v
 Uruchomienie nowego wątku
```

1.3 Interfejs `Callable`

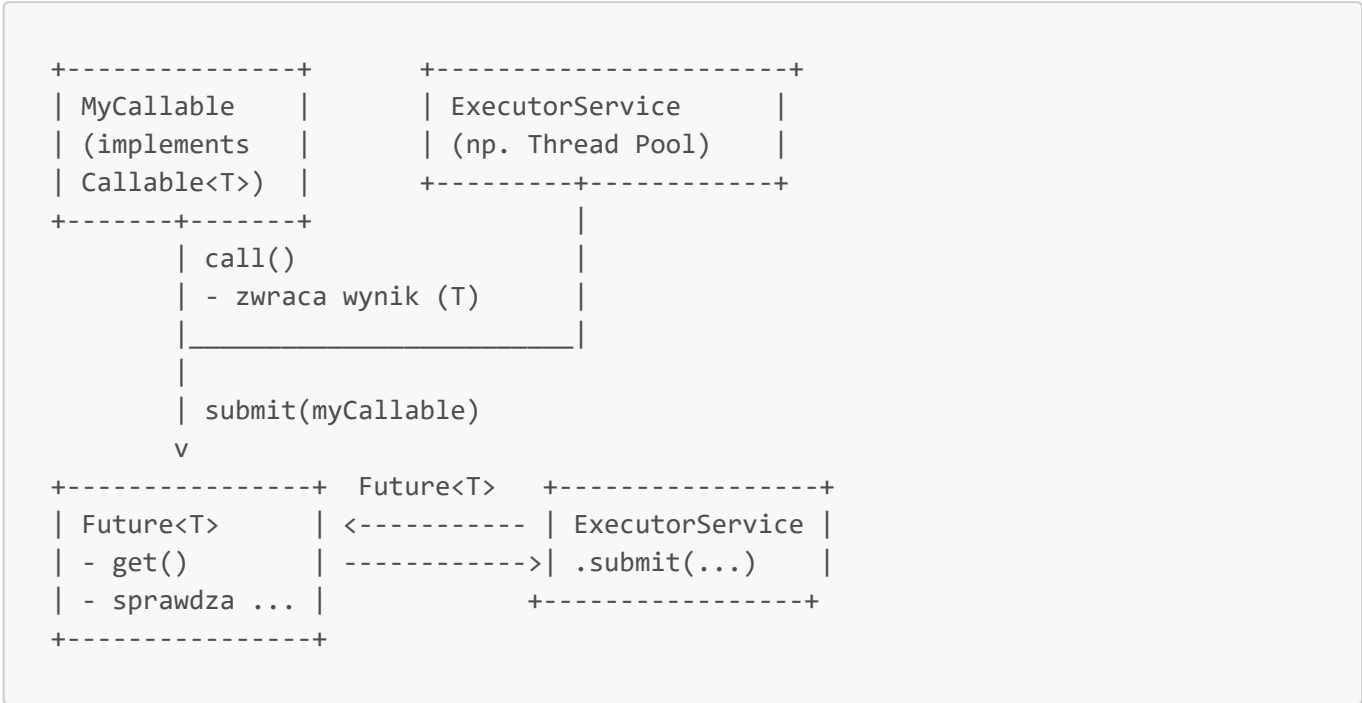
- **Definicja**: `Callable<T>` to interfejs przypominający `Runnable`, ale w odróżnieniu od niego zwraca wartość typu `T` i może rzucać wyjątki.
- **Zasada działania**:
  - Implementujemy metodę `call()`, która zwraca wynik obliczeń (wartość typu `T`).
  - Aby uruchomić obiekt `Callable`, stosujemy `ExecutorService` (np. `ThreadPoolExecutor`) i metodę `submit()`, która zwraca obiekt typu `Future<T>`.
  - Za pomocą `Future` możemy pobrać wynik obliczeń metodą `get()` (zwraca wartość typu `T`).

Przykład kodu z `Callable`:

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

class MyCallable implements Callable<Integer> {
    @Override
    public Integer call() throws Exception {
        // Symulacja obliczeń
        Thread.sleep(1000);
        return 42;
    }
}

public class CallableExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newSingleThreadExecutor();

        Future<Integer> futureResult = executor.submit(new MyCallable());

        try {
            Integer result = futureResult.get(); // Blokuje do czasu zakończenia
wątku
            System.out.println("Wynik z wątku: " + result);
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
```

```
        }

        executor.shutdown();
    }
}
```

Schemat działania z wykorzystaniem `Callable` i `Future`:



1.4 Różnice między `Runnable` a `Callable`

- `Runnable`:
  - Nie zwraca wartości.
  - Nie rzuca wyjątku (poza RuntimeException).
  - Posiada metodę `run()`.
- `Callable<T>`:
  - Zwraca wartość typu `T`.
  - Może rzucać wyjątki (checked exceptions).
  - Posiada metodę `call()`.

1.5 Przykładowe zastosowania

- **`Runnable`**:
  - Proste zadania, np. logowanie, obsługa zdarzeń w tle.
  - Gdy nie potrzebujemy wyniku obliczeń, tylko wykonywanie czynności w tle.
- **`Callable`**:
  - Pobieranie danych z różnych źródeł (np. z bazy), przetwarzanie równoległe i zbieranie wyników.
  - Zastosowanie, gdy istotny jest zwracany rezultat lub obsługa wyjątków.

1.6 Najważniejsze punkty (w pigułce)

- Wątek tworzymy, gdy chcemy pracować równolegle (np. operacje sieciowe, I/O).
- `Runnable` jest prostszy, ale nie zwraca wartości i nie rzuca checked exceptions.
- `Callable` zwraca wartość, obsługuje wyjątki, uruchamiany jest przez `ExecutorService` (zwraca `Future`).

---

2. Aplikacje klient – serwer

2.1 Model klient-serwer – wstęp

Model klient-serwer to architektura, w której:

- **Klient** (np. aplikacja desktopowa, przeglądarka WWW, aplikacja mobilna) wysyła żądania do serwera.
- **Serwer** (np. serwer HTTP, serwer aplikacji) przetwarza żądania i odsyła odpowiedzi.

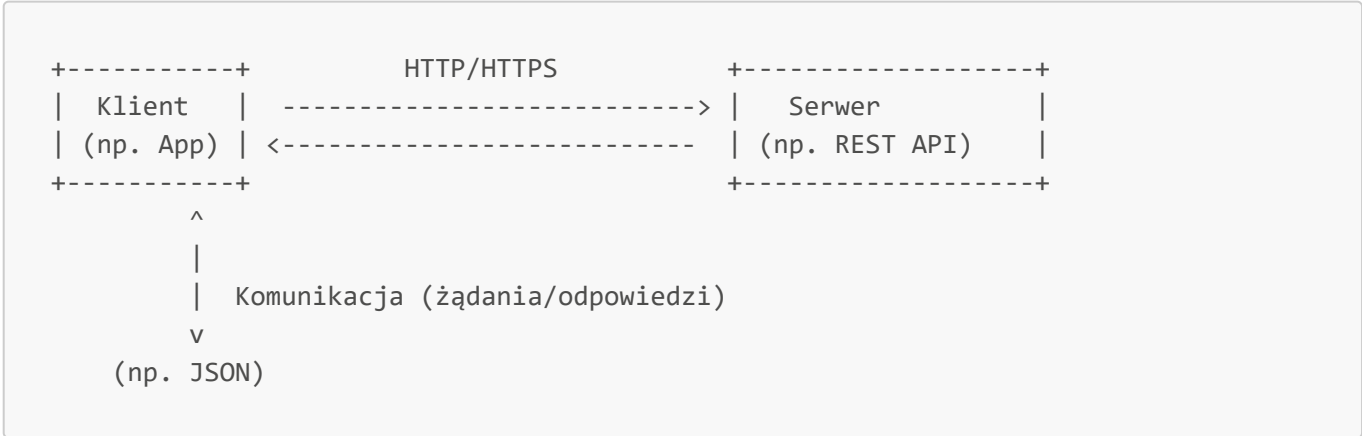
2.2 Rodzaje aplikacji klient-serwer

1. **Klient lokalny – serwer zdalny**:
   - Klient uruchomiony na komputerze użytkownika.
   - Serwer (np. w chmurze) otrzymuje żądania przez sieć.
2. **Klient webowy – serwer webowy**:
   - Klientem jest przeglądarka.
   - Serwerem jest serwer www obsługujący protokół HTTP.
3. **Aplikacje mobilne – serwer w chmurze**:
   - Klientem jest aplikacja mobilna.
   - Serwer w chmurze (np. AWS, Azure) obsługuje dane przez REST API, gRPC itp.

2.3 Komunikacja

- **Protokół** – najczęściej HTTP/HTTPS (w aplikacjach webowych). Możliwe też TCP/UDP (np. w grach sieciowych).
- **Format danych** – JSON, XML, protokoły binarne (Protobuf, gRPC).

2.4 Przykładowy schemat



2.5 Przykładowy kod (serwer HTTP w Javie – uproszczony)

Poniższy przykład wykorzystuje bibliotekę Jetty lub prosty serwer:

```
import com.sun.net.httpserver.HttpServer;
import com.sun.net.httpserver.HttpHandler;
import com.sun.net.httpserver.HttpExchange;
import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;

public class SimpleHttpServer {
    public static void main(String[] args) throws IOException {
        HttpServer server = HttpServer.create(new InetSocketAddress(8080), 0);

        server.createContext("/hello", new MyHandler());
        server.setExecutor(null); // domyślny executor
        server.start();

        System.out.println("Serwer wystartował na porcie 8080");
    }

    static class MyHandler implements HttpHandler {
        @Override
        public void handle(HttpExchange exchange) throws IOException {
            String response = "Witaj, kliencie!";
            exchange.sendResponseHeaders(200, response.getBytes().length);
            try (OutputStream os = exchange.getResponseBody()) {
                os.write(response.getBytes());
            }
        }
    }
}
```

- **Uruchomienie serwera**: Po uruchomieniu, serwer słucha na porcie `8080`.
- **Odpowiedź**: Dla żądania na adres `http://localhost:8080/hello` zwraca tekst „Witaj, kliencie!”.

---

3. Wybrane mikro usługi i sposoby ich działania

3.1 Definicja mikroserwisów

- **Mikroserwisy** to styl architektoniczny, w którym aplikacja składa się z wielu małych, niezależnie wdrażanych i skalowalnych usług.
- Każda usługa odpowiada za konkretną część logiki biznesowej (np. moduł zamówień, płatności, użytkowników).

3.2 Cechy charakterystyczne

1. **Niezależność wdrożeń**: Każdy mikroserwis można wdrażać, aktualizować i skalować niezależnie.
2. **Komunikacja przez sieć**: Usługi komunikują się najczęściej za pomocą protokołów takich jak HTTP (REST), gRPC lub komunikatów (RabbitMQ, Kafka).

3. **Luźne powiązanie** (ang. loose coupling): Zmiana w jednej usłudze nie powinna wpływać na resztę systemu, o ile nie zmienia się interfejs komunikacji.
4. **Autonomiczność**: Każda usługa może mieć własną bazę danych i własny cykl rozwoju.

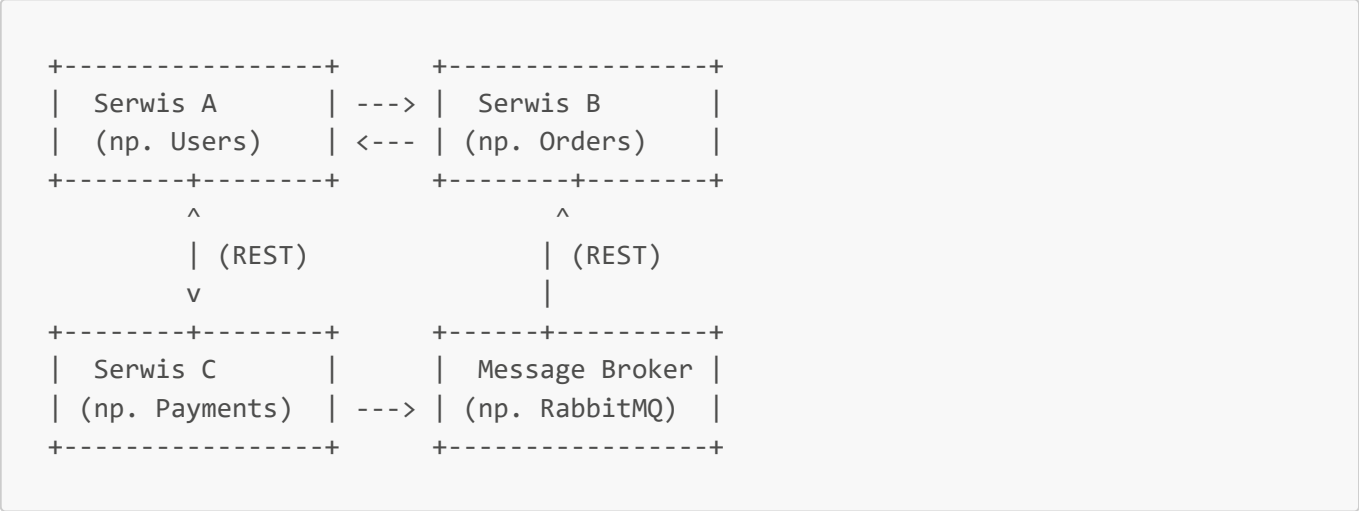
3.3 Przykłady zastosowań

- **Netflix**: pionier architektury mikroserwisów (np. usługa personalizacji, usługa obsługi strumieniowania wideo).
- **Amazon**: każda funkcjonalność (np. obsługa płatności, obsługa zamówień) jest oddzielną usługą.
- **Uber**: usługi do zarządzania trasami, płatnościami, obsługi kierowców itp.

3.4 Wzorce komunikacji

1. **REST** (HTTP + JSON) – najpopularniejszy wzorzec.
2. **gRPC** (HTTP/2 + Protokoły binarne) – szybki i wydajny, często używany w komunikacji wewnętrznej.
3. **Messaging** (RabbitMQ, Apache Kafka) – asynchroniczne przesyłanie komunikatów.

3.5 Przykładowa architektura mikroserwisowa



- Każdy serwis ma swoje API.
- Komunikacja między serwisami może odbywać się synchronicznie (REST, gRPC) lub asynchronicznie (Message Broker).
- Serwisy mogą być skalowane niezależnie, np. `Serwis B` może mieć 3 repliki, jeśli jest najbardziej obciążony.

Pytania i Odpowiedzi – Sprawdź się!

1. Pytania do sekcji: Wielowątkowość w Javie (Runnable i Callable)

P1: Jaka jest podstawowa różnica między `Runnable` a `Callable`?
Odpowiedź:
`Runnable` nie zwraca wyniku ani nie rzuca checked exceptions, natomiast `Callable<T>` zwraca wartość typu `T` i może rzucać wyjątki.

P2: W jaki sposób uzyskujemy wynik z wątku utworzonego za pomocą `Callable`?
Odpowiedź:
Wynik pobieramy za pomocą obiektu `Future<T>` zwróconego przez metodę `submit()`. Wywołujemy `future.get()`, aby odebrać rezultat.

P3: Jak wygląda uruchomienie wątku za pomocą `Runnable`?
Odpowiedź:
Tworzymy obiekt klasy implementującej `Runnable`, następnie przekazujemy go do konstruktora `Thread` i wywołujemy `start()`.

---

2. Pytania do sekcji: Aplikacje klient – serwer

P1: Na czym polega model klient-serwer?
Odpowiedź:
W modelu klient-serwer aplikacja kliencka wysyła żądania do serwera, a serwer je przetwarza i odsyła odpowiedzi.

P2: Jakie są przykładowe protokoły i formaty danych używane w komunikacji klient-serwer?
Odpowiedź:
Najczęściej HTTP/HTTPS jako protokół, a JSON lub XML jako format danych. Możliwe są też protokoły binarne (np. gRPC).

P3: Podaj prosty przykład adresu żądania do serwera HTTP stworzonego w Javie.
Odpowiedź:
`http://localhost:8080/hello` – jeśli serwer nasłuchuje na porcie 8080 i posiada endpoint `/hello`.

---

3. Pytania do sekcji: Mikro usługi

P1: Co to są mikroserwisy?
Odpowiedź:
To niewielkie, niezależnie wdrażane usługi, z których każda odpowiada za określoną funkcjonalność i komunikuje się z innymi usługami po sieci.

P2: W jaki sposób mikroserwisy komunikują się między sobą?
Odpowiedź:
Najczęściej przez protokół HTTP (REST), gRPC lub asynchronicznie poprzez kolejkowanie komunikatów (np. RabbitMQ, Kafka).

P3: Dlaczego mikroserwisy są łatwiejsze do skalowania?
Odpowiedź:
Ponieważ każdą usługę można skalować niezależnie od innych (np. zwiększyć liczbę replik tylko tam, gdzie jest największe obciążenie).