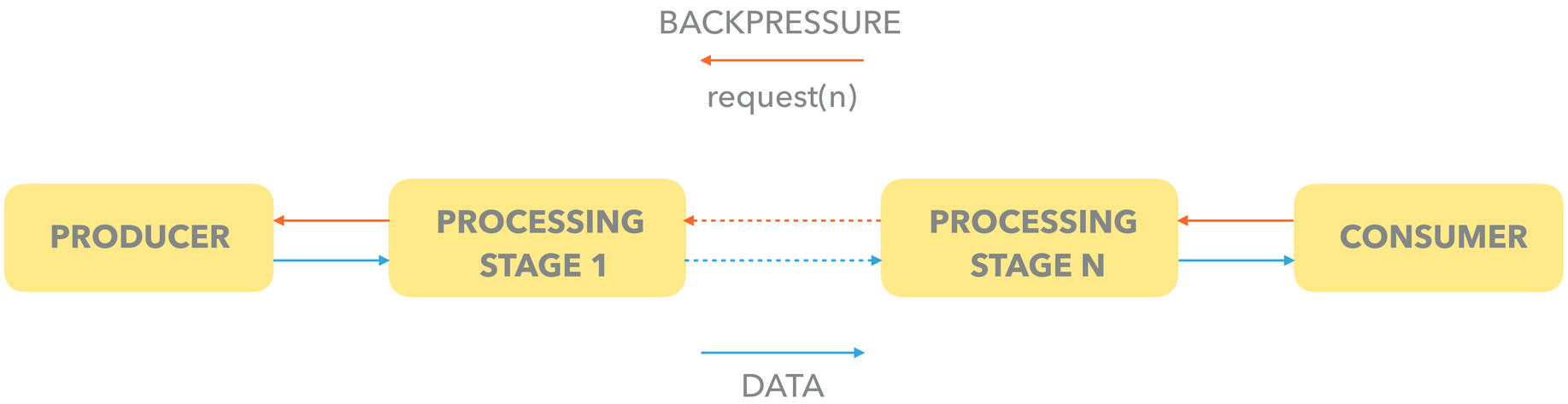
JACEK KUNICKI
@rucek

SOFTWAREMILL

JAK (NIE) UŻYWAĆ REACTIVE STREAMS W JAVIE 9+

PRZETWARZANIE STRUMIENIOWE



Publisher **Processor** **Processor** **Subscriber**

REACTIVE STREAMS

- asynchroniczność
- backpressure jest nieblokujący
  - zwalniamy zasoby gdy tylko nie są potrzebne
- model domeny uwzględnia wolnych konsumentów, np.
  - Twitter API - informuje o tym, że za wolno pobieramy
  - `conflate()` w Akka Streams - agreguje gdy downstream nie nadąża

java.util.concurrent.Flow

`j.u.c.Flow.Publisher<T>`

- emituje elementy typu T, które będą przetwarzać subscriberzy
- subscriberzy rejestrują się przez `subscribe(Subscriber<? super T>)`

`j.u.c.Flow.Subscriber<T>`

- podłącza się do publishera, żeby otrzymywać:
  - potwierdzenie subskrypcji przez `onSubscribe(Subscription)`
  - dane przez `onNext(T)`
  - błędy przez `onError(Throwable)`
  - sygnalizację zakończenia przez `onComplete()`

`j.u.c.Flow.Subscription`

- reprezentuje połączenie publisher-subscriber, umożliwia:
  - backpressure przez `request(long)`
  - anulowanie przetwarzania przez `cancel()`

`j.u.c.Flow.Processor<T, R>`

- połączenie `Subscriber<T>` i `Publisher<R>`

```
publisher.subscribe(subscriber)


onSubscribe
onNext*
(onComplete | onError)?
```


Talk is cheap
show me the
CODE
SOFTWAREMILL

DALSZE WYZWANIA

- nieskończona rekurencja: `request() -> onNext() -> request() -> …`
- obsługa nieskończonego zapotrzebowania
  - gdy mamy `MAX_VALUE` elementów, to wywoływanie dla każdego `onNext()` spowoduje wyczerpanie puli wątków
  - `long demand` + inkrementacja nie wystarczy - overflow

http://www.rhfleet.org/sites/default/files/images/RHF_DTTAH_Clogo_DEC14.jpg

SPI

SERVICE PROVIDER INTERFACE

ISTNIEJĄCE ABSTRAKCJE

- `java.io.InputStream / OutputStream`
- `java.util.Iterator`
- `java.nio.channels.*`
- `javax.servlet.ReadListener/WriteListener`
- `java.sql.ResultSet`
- `java.util.Stream`
- `java.util.concurrent.Flow.*`

```
publisher.subscribe(subscriber)
```

PODSTAWOWE OPERACJE

- obecnie mamy tylko interfejsy
- brak standardowych operatorów typu `filter`, `map` itp.
- https://github.com/lightbend/reactive-streams-utils
- część operacji wbudowana, część dołączana `-Djava.flow.provider=akka`

ale polityka:  vs 

HTTP

- Servlet IO - asynchroniczne od 3.1
- klient HTTP w JDK 9+ udostępnia `POST(Publisher<ByteBuffer>)`
- gdyby `HttpServletRequest` udostępniał publishera z request body, przesyłanie plików sprowadzałoby się do:

```
POST(BodyPublisher.fromPublisher(req.getPublisher()))
```

BAZY DANYCH

- ADBA (Asynchronous Database Access API)
- R2DBC SPI (https://github.com/r2dbc/r2dbc-spi) + Spring Data
- istniejące asynchroniczne sterowniki (ale vendor-specific)
- JPA - co by było, gdyby...

```
Publisher<User> users = entityManager
  .createQuery("select u from users")
  .getResultPublisher()
```

ITD.

- reactive file IO (w stylu `Publisher<Byte>`)
- JMS
- websockets
- AWS - pół na pół
- Alpakka?

DEMO - PROSTA INTEGRACJA

- publisher: `Flux` (Project Reactor)
- processor: `Flow` (Akka Streams)
- subscriber: RxJava

TL;DR

- nie jest to pełna implementacja Reactive Streams
- dostajemy SPI, które ujednolica komunikację pomiędzy różnymi implementacjami
- implementacja na własną rękę nie jest trywialna
  - bez TCK ani rusz

MATERIAŁY

- pluggable runtime: https://github.com/lightbend/reactive-streams-utils
- TCK: https://github.com/reactive-streams/reactive-streams-jvm#specification
- ADBA: https://blogs.oracle.com/java/jdbc-next:-a-new-asynchronous-api-for-connecting-to-a-database
- R2DBC SPI: https://github.com/r2dbc/r2dbc-spi
- Advanced Reactive Java: http://akarnokd.blogspot.com/

cv@softwaremill.com

DZIĘKI!

@rucek

SOFTWAREMILL