Ja definiuję beany merkury – jako integration – widok je czyta na podstawie składanki wysyła json,

W backend na podstawie json-połaczenia i bean – komponenty – generuje kontekst

- odpalany na kontekście merkury i konfiguracja json ladowana do bazy

- oprócz tego wszystkie message tez do bazy – z czego duże obciążenie może być

http://docs.spring.io/spring-integration/reference/htmlsingle/

- daje jako mbean, można obserwowac co przechodzilo, co się wywalilo itd.

* The framework has extensive JMX support
* Exposing framework components as MBeans
* Adapters to obtain attributes from MBeans, invoke operations, send/receive notifications

- <**int:management default-logging-enabled="false" default-counts-enabled="false" default-stats-enabled="false"**/>

- doczytać

- własna metoda SpEL, ktora odwoluje się do obiektu i jego metody statycznej

<int:spel-function id="xpath"

class="com.foo.test.XPathUtils" method="evaluate(java.lang.String, java.lang.Object)"/>

<int:transformer input-channel="in" output-channel="out"

expression="#xpath('//foo/@bar', payload)" />

- funkcje dziedziczone z parenta, moga być nadpisane w podrzednych contextach

<transformer expression="#jsonPath(payload, '$.store.book[0].author')"/>

<filter expression="#jsonPath(payload,'$..book[2].isbn') matches '\d-\d{3}-\d{5}-\d'"/>

<splitter expression="#jsonPath(payload, '$.store.book')"/>

- payload to object, ma timestamp, id , corelation id,

- header to klucz-wartośc, dane o source, pliku, można też o przeznaczeniu (dane mail), mozna też własne pary dodac do header

- The components responsible for these connections are Message Endpoints

-- Just as a Controller handles HTTP requests, the Message Endpoint handles Messages. Just as Controllers are mapped to URL patterns, Message Endpoints are mapped to Message Channels. The goal is the same in both cases: isolate application code from the infrastructure

- service activator

-- If no output channel has been configured, then the reply will be sent to the channel specified in the Message’s "return address" if available.

- inbound message channel adapter – pollable or Message Driven

- mam MessageChannel – send – rozszerzony przez SubscribeChannel i pollableChannel - receive

- PSC

any subscriber must be a MessageHandler itself, and the subscriber’s handleMessage(Message) method will be invoked in turn.

- QueueChannel – point-to-point I kolejkowanie, Priority to szczeólny przypadek, bo po priority nie FIFO

- RANDEZOUS – sender blokuje az nie przyjdzie receiver I odwrotnie, wiadomo, że wiadomość dotarła

- można do channel persystencje dodać – message-store attribute or JMS-backed

- mozna wlasną load balancing strategy – bean

<int:dispatcher load-balancer-ref="lb"/>

- Keep in mind that load-balancing and failover only apply when a channel has more than one subscribed Message Handler. When using the namespace support, this means that more than one endpoint shares the same channel reference in the "input-channel" attribute.

- ExecutorChannel – point-to-point ale async, moze blokowac, jak queue executora zapchane

- ScopedChannel, można własne, np. Dla danego thread, że jak wraca to tez tylko ten watek czyta powrotne

- do channel mozna dodac interceptor – pre/post send/receive I completion

-- send dla direct, receive dla poolable

The methods that return a Message instance can be used for transforming the Message or can return null to prevent further processing (of course, any of the methods can throw a RuntimeException). Also, the preReceive method can return *false* to prevent the receive operation from proceeding.

- DateType channel, może akceptowac tylko payload wybranego typu

<int:channel id="stringOrNumberChannel" datatype="java.lang.String,java.lang.Number"/>

-- jak inny typ to ex, albo definiujemy integrationConversionService by konwertowac na docelowy typ – to jako Converter

- teraz szuka konwertera w zdefiniowanych beanach, ale mozna na channel atrybut message-converter

- store queue channel by nie tracić messages

<int:channel id="dbBackedChannel">

<int:queue message-store="channelStore"/>

</int:channel>

<bean id="channelStore" class="o.s.i.jdbc.store.JdbcChannelMessageStore">

<property name="dataSource" ref="dataSource"/>

<property name="channelMessageStoreQueryProvider" ref="queryProvider"/>

</bean>

- mozna w channel PriorityQueue z comparatorem

<int:priority-queue comparator="widgetComparator"

capacity="10"/>

- jest message store dla channel

Implementations of the PriorityCapableChannelMessageStore are currently provided for Redis

- mozna grlobal interceptors na dany pattern channel, z bean wewnętrznym bądź jako referencja, order mówi który interceptor w kolejności odpalony na channel, jak na chanel bezposrednie to positiv order za nimi a negative przed nimi

<int:channel-interceptor ref="myInterceptor" pattern="input\*, bar\*, foo" order="3"/>

-- default order 0 a pattern \*

- wire tap, dobry do debugowania, kopiuje massage w inny kanal

<int:interceptors>

<int:wire-tap channel="logger"/>

<int:logging-channel-adapter id="logger" level="DEBUG"/>

-- tu expression – SpEL

-- default logowane payload, ale można log-full-message na true

- wire-tap – to wariacja bridge, ale wewnątrz channel, zazwyczaj sync, zależy od channel do, czasem transakcja np. potrzebna

- selector or selector-expression mówią, czy message ma isć na channel

- Mozna globalny wire-tap na pattern

<int:wire-tap pattern

- 2 default global channels in context - errorChannel and nullChannel.

- we can poll internal services and external sources

- global poller

<int:poller max-messages-per-poll="1" fixed-rate="1000"/>

- default -1, execute poll all the time

- the expression for an <inbound-channel-adapter> will generate a message anytime the evaluation result is a non-null value, while the expression for an <outbound-channel-adapter> must be the equivalent of a voidreturning method invocation.

- message nie mutowalny, zarowno header I payload, ale wewnetrzne dane header mozna modyfikowac,

- header predefined – id, timestamp, reply\_channel, error\_channel

- dodatkowe : CORRELATION\_ID, SEQUENCE\_NUMBER, SEQUENCE\_SIZE, EXPIRATION\_DATE, PRIORITY, DUPLICATE\_MESSAGE

- tworzenie message

Message<String> message1 = MessageBuilder.withPayload("test")

.setHeader("foo", "bar")

.build();

Message<String> message2 = MessageBuilder.fromMessage(message1).build();

- mozna nowy payload, ale kopiowac header

- priority w header dla PriorityQueue

- payload type router – po typie kolejny channel

- header-value-router – na podstawie klucza z header – jak value a to do a itd.

-- jak nie zdefiniowane to channel nazywa się jak header value

<int:header-value-router input-channel="routingChannel" header-name="testHeader"/>

-- Normally, messages for which the header value is not explicitly mapped to a channel will be sent to the default-output-channel

- do listy zdefiniowanych –

<int:recipient-list-router

-- mogą być dodatkowe selectory dla channel

<int:recipient channel="channel1" selector-expression="payload.equals('foo')"/>

- route error handling

- Router z bean w srodku, bądx namiar ref na bean I metodą bądź odwołanie do klasy z metoda z adnonotacją

- mozna prosty SpEL do wyboru docelowego channel

<int:router input-channel="inChannel" expression="payload.paymentType">

<int:mapping value="CASH" channel="cashPaymentChannel"/>

<int:mapping value="CREDIT" channel="authorizePaymentChannel"/>

<int:mapping value="DEBIT" channel="authorizePaymentChannel"/>

</int:router>

<int:router input-channel="inChannel" expression="payload + 'Channel'"/>

- wiadomosć może być dostarczona do wielu channels na raz, jak router zwróci liste :

<int:router input-channel="inChannel" expression="headers.channels"/>

You may also find Collection Projection and Collection Selection expressions useful to select multiple channels

- mozna routing na xpath, mozna na adnotacji I message/header jako param

- The [Dynamic Router](http://www.eaipatterns.com/DynamicRouter.html) pattern describes the mechanisms by which one can change/configure routers dynamically without bringing down the system or individual routers.

-- dodajemy dynamikę w mapowaniu pomiędzy message identifier I channel – to pomiędzy

-- mozna dynamicznie rutowanie, np. Przez\ jmx

-- mozna to np. Przez control buss – set/replace/removeChannelMappings

- souting slip – wybiera kolejny channel, gdy nie ma outputChannel zdefiniowanego

- w Filtr moze zjeść wiadmomośc, na discard channel lub rzucić ex

-- mozna dynamicz\nie za pomocą expression

- aggregator koreluje message po id, zbiera grupę I ją wydaje

- resequencer – nie zmienia wiadomości, ale zbiera je I sortuje wg sequence\_number, uzywa też correlation\_Id

- dodanie wartości do header- taki specyficzny transformer

<int:header-enricher>

<int:header name="foo" value="bar"/>

</int:header-enricher>

- mozna wywołanie innego łańcuch na koncu/ze srodka łańcucha

## - Scatter-Gather pozwala zawołac kilku subskrybentów I zebrac informacje od nich I na ich postawie wybrac najlepsza/podjąć decyzję

- barrier – robi wstrzymanie na pewien czas, az przyjdzie wiadomość zwrotna z channel

<int:barrier id="barrier1" input-channel="in" output-channel="out"

correlation-strategy-expression="headers['myHeader']"

output-processor="myOutputProcessor"

timeout="10000">

</int:barrier>

-- output processor łączy wiadomości po correlation

- Transformacja z SpEL

<int:transformer input-channel="inChannel"

output-channel="outChannel"

expression="payload.toUpperCase() + '- [' + T(java.lang.System).currentTimeMillis() + ']'"/>

- są zdefiniowane transformer

<int:object-to-string-transformer input-channel="in" output-channel="out"/>

- <int:payload-serializing-transformer input-channel="objectsIn" output-channel="bytesOut"/>

- object to mam, map to object

<int:object-to-map-transformer input-channel="directInput" output-channel="output" flatten="false"/>

Map-to-Object

<int:map-to-object-transformer input-channel="input"

                       output-channel="output"

                        type="org.foo.Person"/>

-

<int:stream-transformer id="withCharset" charset="UTF-8"

input-channel="charsetChannel" output-channel="output"/>

<int:object-to-json-transformer input-channel="objectMapperInput"/>

<int:json-to-object-transformer input-channel="objectMapperInput"

type="foo.MyDomainObject"/>

we can add here

object-mapper="customObjectMapper"

- wyrzucanie specyficznych key z header

<int:header-filter input-channel="inputChannel"

output-channel="outputChannel" header-names="lastName, state"/>

- enricher wzbogadza header, daje wartości, zmienia specyficzne header

<int:header-enricher input-channel="in" output-channel="out">

<int:error-channel ref="applicationErrorChannel"/>

<int:reply-channel ref="quoteReplyChannel"/>

<int:correlation-id value="123"/>

<int:priority value="HIGHEST"/>

<routing-slip value="channel1; routingSlipRoutingStrategy; request.headers[myRoutingSlipChannel]"/>

<int:header name="bar" ref="someBean"/>

</int:header-enricher>

- można też namiar na bean co dynamicznie value oblicza

<int:header-enricher input-channel="in" output-channel="out">

<int:header name="foo" method="computeValue" ref="myBean"/>

</int:header-enricher>

computeValue(String payload)

- można też to z groovy I SpEL

<int:header-enricher input-channel="inputChannel" output-channel="outputChannel">

<int:header name="some\_header">

<int-groovy:script location="org/SampleGroovyHeaderEnricher.groovy"/>

<int:header name="foo" expression="payload.toUpperCase() + '\_US'"/>

- podobnie payload enricher

<int:enricher id="userEnricher"

input-channel="input">

<int:property name="user.updateDate" expression="new java.util.Date()"/>

<int:property name="user.firstName" value="foo"/>

<int:property name="user.lastName" value="bar"/>

<int:property name="user.age" value="42"/>

</int:enricher>

- ukrywanie części wiadomości np. Jak duzy payload

The [Claim Check](http://www.eaipatterns.com/StoreInLibrary.html) pattern describes a mechanism that allows you to store data in a well known place while only maintaining a pointer (Claim Check) to where that data is located

- Codec – jest to alternatywa dla serializacji mozna do byte [] i z powrotem

- pobieranie z poolable channel, może tez byc cron

<int:transformer input-channel="pollable"

ref="transformer"

output-channel="output">

<int:poller fixed-rate="1000"/>

</int:transformer>

Jak channel poolable I nie ma pooler to ex, a jak jest pooler a channel nie poolable to też ex

-- by nie lecialo ex – default pooler

<int:poller id="defaultPoller" default="true" max-messages-per-poll="5" fixed-rate="3000"/>

- możemy zmienić pooling rate na runetime

- konwerter z typu na typ

*@Component*

*@IntegrationConverter*

**public** **class** TestConverter **implements** Converter<Boolean, Number> {

**public** Number convert(Boolean source) {

**return** source ? 1 : 0;

}

}

<int:converter ref="sampleConverter"/>

<bean id="sampleConverter" class="foo.bar.TestConverter"/>

- pooler może być asynch z executorem

<int:channel id="publishChannel">

<int:queue />

</int:channel>

<int:service-activator input-channel="publishChannel" ref="myService">

<int:poller receive-timeout="5000" task-executor="taskExecutor" fixed-rate="50" />

</int:service-activator>

<task:executor id="taskExecutor" pool-size="20" />

- Roles allow endpoints to be started and stopped as a group; this is particularly useful when using leadership election where a set of endpoints can be started or stopped when leadership is granted or revoked respectively.

- mozna startować stopowac w grupach – life cycle

Any object implementing SmartLifecycle can be programmatically added, not just endpoints.

roleController.addSmartLifeCycleToRole("cluster", someEndpoint);

- mozna odpalac jednego I start, mozna podzielić na grupy

- gateway to od kodu prosty interface, a w bean definiujemy z jakich kolejek korzysta

<int:gateway id="cafeService"

service-interface="org.cafeteria.Cafe"

default-request-channel="requestChannel"

default-reply-timeout="10000"

default-reply-channel="replyChannel"/>

- mozna ustawic róznych channel/response type dla roznej metody gateway

<int:method name="getAllLoanQuotes" request-channel="loanBrokerPreProcessingChannel">

<int:header name="RESPONSE\_TYPE" value="ALL"/>

</int:method>

- mozna w rózny sposób mapowac na payload i header, mozna adnotacje, może być mylne do przemapowania, trzeba w dobry sposób

- mozna na interface i na rózne sposoby expression dla róznych metod, przy uzyciu argumentów, metod i innych beanów

<int:gateway id="myGateway" service-interface="org.foo.bar.MyGateway">

<int:method name="send1" payload-expression="#args[0] + 'bar'"/>

<int:method name="send2" payload-expression="@someBean.sum(#args[0])"/>

<int:method name="send3" payload-expression="#method"/>

<int:method name="send4">

<int:header name="foo" expression="#args[2].toUpperCase()"/>

</int:method>

</int:gateway>

- mozna przetworzyć xml na adnotacje na interfejsie – klasa i metoda

- mozna tylko wysyłac, wtedy mozna uzyc paylowad – jakaś stała, wywołanie metody z bean, etc.

@Payload("new java.util.Date()")

List<Order> retrieveOpenOrders();

- asynchroniczny gateway zwraca Future

- LitenableFuture – nowe, ma callback wywołany gdy odpowiedx wróci

ListenableFuture<String> result = **this**.asyncGateway.async("foo");

result.addCallback(**new** ListenableFutureCallback<String>() {

*@Override*

**public** **void** onSuccess(String result) {

...

}

*@Override*

**public** **void** onFailure(Throwable t) {

...

}

});

- do wykonania dodatkowej operacji na zwróconym response

CompletableFuture

- Promise podobne do future, mozna wartośc pobrac od razu, albo dodac consumer, coby zkonsumowal odpowiedź jak przyjdzie

promise.consume(**new** Consumer<Invoice>() {

*@Override*

**public** **void** accept(Invoice invoice) {

handleInvoice(invoice);

}

})

.flush();

- jest kilka sposobów obsługi jak gateway nie otrzyma zwrotki

- service activator z SpEL

<int:service-activator input-channel="in" output-channel="out"

expression="@accountService.processAccount(payload, headers.accountId)"/>

<bean id="accountService" class="foo.bar.Account"/>

- delayer zapewnie proste opóxnienie w przepływie wiadomości

<int:delayer id="delayer" input-channel="input"

default-delay="3000" output-channel="output"/>

- wiadomości przetrzymywane w message store

- mozna uzywac ruby/groovy lub js zamiast SpEL dla skryptowania w komponentach

-- mozna inline w CDATA lub odwołanie do resource gdzie skrypt

<int:filter input-channel="referencedScriptInput">

<int-script:script lang="ruby" location="some/path/to/ruby/script/RubyFilterTests.rb"/>

</int:filter>

<int:filter input-channel="inlineScriptInput">

<int-script:script lang="groovy">

<![CDATA[

return payload == 'good'

]]>

</int-script:script>

</int:filter>

-- support scripting include router, service-activator, transformer, and splitter.

-- uzyteczne jest odświerzanie skryptu, jak wartośc 0, to zaraz po zmianie

refresh-check-delay="5000"

-- mozna bindowac variable dla skryptu

<script:variable name="foo" value="foo"/>

<script:script lang="ruby" variables="foo=FOO, date-ref=dateBean">

-- mozna też variable generator

<int-script:script location="foo/bar/MyScript.groovy"

script-variable-generator="variableGenerator"/>

<bean id="variableGenerator" class="foo.bar.MyScriptVariableGenerator"/>

- filter groovy

<int:filter input-channel="inlineScriptInput">

<int-groovy:script><![CDATA[

return payload == 'good'

]]></int-groovy:script>

</int:filter>

Mozna adnotacjami

*@groovy.transform.CompileStatic*

- control bus, pobiera z channel, przetwarza i wywołuje message, z groovy to jako skrypt po przetworzeniu

<int-groovy:control-bus input-channel="input"

output-channel="output"

customizer="groovyCustomizer"/>

- advice support – mówi np. Co zrobic jak we flow jakiś niespodziewany problem i np. Retry by sie przydał, można wywołać dodatkową akcje na flow

- logging adapter

<int:logging-channel-adapter

channel="" 1

level="INFO" 2

expression="" 3 // mowi co logowane jak full=false, default payload

log-full-message="false" 4

logger-name="" /> 5

- dla metrics jest management tag

<int:management

default-logging-enabled="false" 1

default-counts-enabled="false" 2

default-stats-enabled="false" 3

counts-enabled-patterns="foo, !baz, ba\*" 4

stats-enabled-patterns="fiz, buz" 5

metrics-factory="myMetricsFactory" /> 6

- metryki channel zależą od typu, ile wysłal/odebrał, error, buffer in queue...

- są channel adaptery dla JMX

-- notifications listening channel adapter – podłącza się na dane mbean i nasłuchuje na niego, event-driven

-- notifications publishibg channel adapter, jak przychodzi cos na kanał to tworzona notyfikacja przy uzyciu mbean-exporter

- message history – trakowanie message przez dodawanie informacji do headera, właczamy przez

<int:message-history/>

- kazdy co ma id będzie trakowany List<Properties>, dostep przez

message.getHeaders().get(MessageHistory.HEADER\_NAME, MessageHistory.**class**).iterator()

mozna wybrać element do trakowania

<int:message-history tracked-components="\*Gateway, sample\*, foo"/>

- tam gdzie buforowane element – np. Aggregator/queue – mozemy dodac message store

<int:channel id="myQueueChannel">

<int:queue message-store="refToMessageStore"/>

-- default w pamięci, ale mozna rózne data-stores podłączyć, relacyjne, redis, mongo, gemfire

- metadata store – for example for idempotent receiver

- mozliwośc stopowania przez JMX

- mozna pobrac integration graph w JSON

-- mozna graph jako REST wystawić w webówce

- jest Redis message store to zapisu message w redis

-

-

-

-

-

-

-