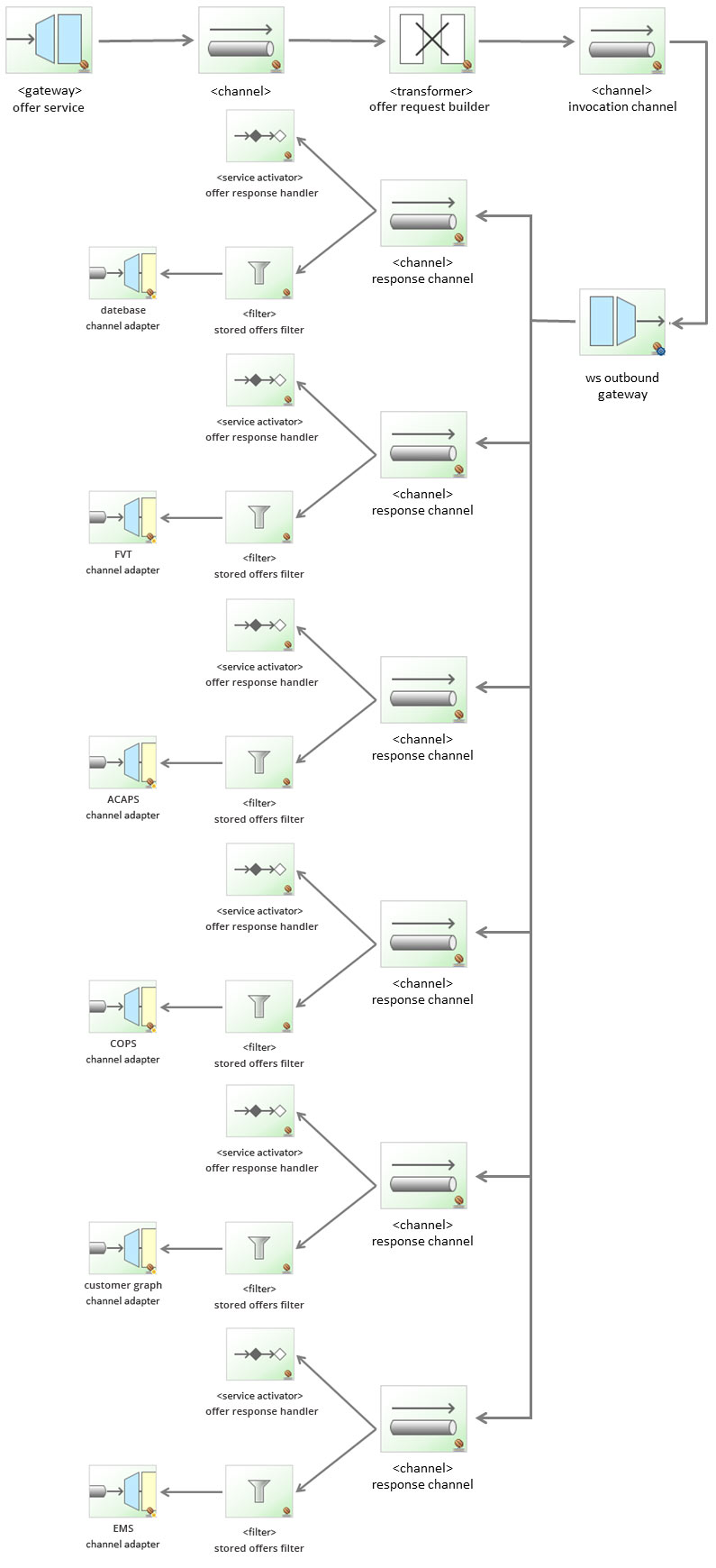
overview of the flow

This use case shows how to configure several messaging and integration endpoints. The user asks for a offer by specifying the offer Id. The flow will invoke a web service and return the response to the user. Additionally, some type of offers will be stored to a database.

The flow is as follows:

* An integration [gateway](http://docs.spring.io/spring-integration/docs/4.0.0.RELEASE/reference/html/messaging-endpoints-chapter.html#gateway)(offer service) serves as the entry to the messaging system.
* A [transformer](http://docs.spring.io/spring-integration/docs/4.0.0.RELEASE/reference/html/messaging-transformation-chapter.html#transformer)builds the request message from the user specified offer Id.
* A web service [outbound gateway](http://docs.spring.io/spring-integration/docs/4.0.0.RELEASE/reference/html/ws.html#webservices-outbound) sends the request to a web service and waits for a response.
* A [service activator](http://docs.spring.io/spring-integration/docs/4.0.0.RELEASE/reference/html/messaging-endpoints-chapter.html#service-activator) is subscribed to the response channel in order to return the offer name to the user.
* A [filter](http://docs.spring.io/spring-integration/docs/4.0.0.RELEASE/reference/html/messaging-routing-chapter.html#filter)is also subscribed to the response channel. This filter will send some types of offers to a database [channel adapter](http://docs.spring.io/spring-integration/docs/4.0.0.RELEASE/reference/html/mongodb.html#mongodb-outbound-channel-adapter) in order to store the response to a database.

The following diagram better shows how the flow is structured:



3 Spring configuration

As discussed in the introduction section, the entire configuration is defined with JavaConfig. This configuration is split into three files: infrastructure, web service and database configuration. Let’s check it out:

3.1 Infrastructure configuration

This configuration file only contains the definition of message channels. The messaging endpoints (transformer, filter, etc…) are configured with annotations.

**InfrastructureConfiguration.java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | *@Configuration*  *@ComponentScan*("xpadro.spring.integration.endpoint") //@Component  *@IntegrationComponentScan*("xpadro.spring.integration.gateway") //@MessagingGateway  *@EnableIntegration*  *@Import*({MongoDBConfiguration.class, WebServiceConfiguration.class})  public class InfrastructureConfiguration {    *@Bean*  *@Description*("Entry to the messaging system through the gateway.")      public MessageChannel requestChannel() {          return new DirectChannel();      }    *@Bean*  *@Description*("Sends request messages to the web service outbound gateway")      public MessageChannel invocationChannel() {          return new DirectChannel();      }    *@Bean*  *@Description*("Sends web service responses to both the client and a database")      public MessageChannel responseChannel() {          return new PublishSubscribeChannel();      }    *@Bean*  *@Description*("Stores non filtered messages to the database")      public MessageChannel storeChannel() {          return new DirectChannel();      }  } |

The @ComponentScan annotation searches for @Component annotated classes, which are our defined messaging endpoints; the filter, the transformer and the service activator.

The @IntegrationComponentScan annotation searches for specific integration annotations. In our example, it will scan the entry gateway which is annotated with @MessagingGateway.

The @EnableIntegration annotation enables integration configuration. For example, method level annotations like @Transformer or @Filter.

3.2 Web service configuration

This configuration file configures the web service outbound gateway and its required marshaller.

**WebServiceConfiguration.java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | *@Configuration*  public class WebServiceConfiguration {    *@Bean*  *@ServiceActivator*(inputChannel = "invocationChannel")      public MessageHandler wsOutboundGateway() {          MarshallingWebServiceOutboundGateway gw = new MarshallingWebServiceOutboundGateway("http://localhost:8080/spring-ws-offers/offers", jaxb2Marshaller());          gw.setOutputChannelName("responseChannel");            return gw;      }    *@Bean*      public Jaxb2Marshaller jaxb2Marshaller() {          Jaxb2Marshaller marshaller = new Jaxb2Marshaller();          marshaller.setContextPath("xpadro.spring.integration.ws.types");            return marshaller;      }  } |

The gateway allows us to define its output channel but not the input channel. We need to annotate the adapter with @ServiceActivator in order to subscribe it to the invocation channel and avoid having to autowire it in the message channel bean definition.

3.3 Database configuration

This configuration file defines all necessary beans to set up [mongoDB](https://www.mongodb.org/). It also defines the mongoDB outbound channel adapter.

**MongoDBConfiguration.java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | *@Configuration*  public class MongoDBConfiguration {    *@Bean*      public MongoDbFactory mongoDbFactory() throws Exception {          return new SimpleMongoDbFactory(new MongoClient(), "si4Db");      }    *@Bean*  *@ServiceActivator*(inputChannel = "storeChannel")      public MessageHandler mongodbAdapter() throws Exception {          MongoDbStoringMessageHandler adapter = new MongoDbStoringMessageHandler(mongoDbFactory());          adapter.setCollectionNameExpression(new LiteralExpression("offers"));            return adapter;      }  } |

Like the web service gateway, we can’t set the input channel to the adapter. I also have done that by specifying the input channel in the @ServiceActivator annotation.

4 Detail of the endpoints

The first endpoint of the flow is the integration gateway, which will put the argument (offerId) into the payload of a message and send it to the request channel.

|  |  |
| --- | --- |
| 1  2  3  4  5 | *@MessagingGateway*(name = "entryGateway", defaultRequestChannel = "requestChannel")  public interface OfferService {        public String findOffer(String offerId);  } |

The message containing the offer id will reach the transformer. This endpoint will build the request object that the web service is expecting:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | *@Component*  public class OfferRequestBuilder {      private Logger logger = LoggerFactory.getLogger(this.getClass());    *@Transformer*(inputChannel="requestChannel", outputChannel="invocationChannel")      public GetOfferRequest buildRequest(Message<String> msg) {          logger.info("Building request for offer [{}]", msg.getPayload());          GetOfferRequest request = new GetOfferRequest();          request.setOfferId(msg.getPayload());            return request;      }  } |

Subscribed to the response channel, which is the channel where the web service reply will be sent, there’s a service activator that will receive the response message and deliver the offer name to the client:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | *@Component*  public class OfferResponseHandler {      private Logger logger = LoggerFactory.getLogger(this.getClass());    *@ServiceActivator*(inputChannel="responseChannel")      public String getResponse(Message<GetOfferResponse> msg) {          GetOfferResponse offer = msg.getPayload();          logger.info("Offer with ID [{}] received: {}", offer.getOfferId(), offer.getName());            return offer.getName();      }  } |

Also subscribed to the response channel, a filter will decide based on its type, if the offer is required to be stored to a database:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | *@Component*  public class StoredOffersFilter {      private Logger logger = LoggerFactory.getLogger(this.getClass());    *@Filter*(inputChannel="responseChannel", outputChannel="storeChannel")      public boolean filterOffer(Message<GetOfferResponse> msg) {          if (!msg.getPayload().getOfferId().startsWith("BC-")) {              logger.info("Offer [{}] filtered. Not a BF offer", msg.getPayload().getOfferId());              return false;          }            logger.info("Offer [{}] validated. Storing to database", msg.getPayload().getOfferId());          return true;      }  } |

5 Testing the entire flow

The following client will send two requests; a BC type offer request that will be stored to the database and a DF type offer that will be finally filtered:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | *@RunWith*(SpringJUnit4ClassRunner.class)  *@ContextConfiguration*(classes={InfrastructureConfiguration.class})  public class TestApp {  *@Autowired*      OfferService service;    *@Test*      public void testFlow() {          String offerName = service.findOffer("BC-45");          assertNotNull(offerName);          assertEquals("Introduction to Java", offerName);            offerName = service.findOffer("DF-21");          assertNotNull(offerName);          assertEquals("Functional Programming Principles in Scala", offerName);  }  } |