Would you mind having a little less?

JMS, just lighter

Java Message Service, the Java EE standard for messaging, is often used in enterprise environments in order to communicate asynchronously and transactional between services. Version 1.1 was standardized in 2003 [1], but not updated since. On the one hand, this proves its technical maturity, but on the other hand it shows signs of age, as it has not yet taken the step to become lightweight, as e.g. EJB has taken between 2.1 and 3.0. This step is taken by the project "Message-API" that is available from java.net under the Apache License 2.0.

by Rüdiger zu Dohna

Big systems, that are developed by multiple teams, multiple departments, and maybe even at multiple locations, a.k.a. enterprise systems, have special non-functional requirements. In this kind of environment, responsibilities are taken by services that run on separate servers that separate teams take account for. In order to organize such services they are organized into layers and orchestrated into processes and subprocesses. It's common for remote calls to pass many layers, making for hearty call cascades.

First, performance suffers badly when e.g. a front end accesses various databases via three remote intermediate layers with series of services arranged into subprocesses. Second, availability is seriously affected, as a single service failing drags down the whole chain.

Thirdly, when calling services that are not read-only, you'll have to take good care not to miss any calls or execute the same call twice. Status changes of the sender and the receiver always must be in sync, but distributing transactions over so many layers is hardly practical with any technology. So all clients have to repeat all calls that failed and services must recognize such repetitions and ignore them, i.e. they must be idempotent. That's the only way to make sure, e.g. after a timeout, that a call has arrived but is not executed multiple times.

All of this causes quite some stress on the developers, which can be mitigated considerably by making the calls asynchronous. Even when the receiver is unavailable for a moment, is too slow to react within a reasonable time, or runs on an internal error: The client should not suffer from it. If the state of a client and a service do not have to be in sync immediately, you can trade a little consistency for a lot of availability and scalability. This is basically the same tradeoff that is done in cloud architectures [4].

JMS

Let's start with an example of sending a simple message via JMS to create a customer with first and last name (Listing 1).

The code does not show how to set any of the various delivery attributes. In some details that code would not have been 100% portable between different JMS implementations, as it can even lead to dropped messages in some error situations, e.g. when done in the wrong order. And there is no error handling shown. Even without that, the point gets clear good enough: Using JMS requires *a lot* of boiler plate code.

Refactoring

Much worse than the sheer amount of code is the commixture of different levels of abstraction. Various aspects superimpose each other, making it difficult to tell them apart. This problem can be mitigated by proper refactoring. So in order to keep

How about EJB 3.1 @Asynchronous and CDI Events?

Sometimes JMS was used for *local* asynchronous calls, too. You send yourself a JMS in order to process it asynchronously. With Java EE 6 this has become much easier with use of the @Asynchronous annotation and CDI Events [3]. But that's not persistent transactional, as the receiver will not be restarted, if the system fails in the meantime.

the business code clean [5] you would start by extracting the code that does the actual sending of the message into a separate method with a pure business signature, i.e. a signature that reveals absolutely no technical details:

```
void createCustomer(String first, String lastName);
```

Please note that this adapter method returns void and declares no exceptions, as the real work – creating a customer – will be carried out asynchronously. At runtime, only local, technical problems can occur.

Along the way we have gained a certain level of type safety. This is especially useful when values have to be converted, e.g. a *Date* object.

If more types of messages have to be sent to the same queue, you'll write more adapter methods. Now the envy by these methods of the *Connection* and *Queue* become clear (cf. Feature Envy, [6], page 80): These methods are the only ones that require those fields. So you'd extract all adapter methods together with the fields into a separate *CustomerService* class. The name of the *Queue* and other options can easily be made configurable, so these parameters for the container that the application runs in can be configured when deploying the application and not when writing the code.

This separation as well facilitates writing unit tests for the business code, as it's easy to mock the *CustomerService* class. You'd probably extract a *CustomerService* interface and name the implementation *JmsCustomerService*, although this isn't really necessary any more for current mocking frameworks like Mockito [7]. In theory it would even be possible to address a completely different messaging solution instead.

The receiver needs a similar adapter for unpacking the message and calling the actual business method. These two adapter classes encapsulate the precise, technical format of the messages. You'd probably pack them together with the interface into an API jar and distribute it to the developers of the client and the service. In this way a single developer can make sure that these two adapters match, and you don't have to make the laborious and error-prone detour of writing documentation.

Now we have reached quite something by doing this refactoring: The code is clean and maintainable. Sender and receiver only have to agree on a clear business interface. All code stays on a single level of abstraction. On the logical level, the business code sends a message to the receiver – asynchronously and transactional. Technically, the call takes the indirection of an adapter for sending, JMS as the infrastructure, and an adapter for receiving.

Listing 1

```
public class SenderBusinessClass {
  @Resource(mappedName = “jms/ConnectionFactory”)
  private ConnectionFactory connectionFactory;

  @Resource(mappedName = “jms/Queue”)
  private Queue queue;

  private Connection connection = null;

  @PostConstruct
  public void openConnection() {
    this.connection =
       connectionFactory.createConnection();
  }

  @PreDestroy
  public void closeConnection() {
    if (connection != null) {
      connection.close();
    }
  }

  public void businessMethod() {
    ...

    Session session = connection.createSession(true, 0);
    MessageProducer producer = session.createProducer(queue);
    MapMessage message = session.createMapMessage();

    message.setStringProperty(“MESSAGE_TYPE”,“createCustomer”);
    message.setStringProperty(“MESSAGE_VERSION”, “1.0”);
    message.setString(“firstName”, firstName);
    message.setString(“lastName”, lastName);

    producer.send(message);

    ...
  }
}
```

Generic Adapters

When the parameters of the adapter methods get more complex, the weakness of this approach starts to show: It gets more and more cumbersome to store the parameters into a *MapMessage*. Entries of a list have to be counted somehow and structures have to be expressed as paths in some way. Let's take an order that has several positions that all have a quantity. The quantity of order item 3 could, e.g., be keyed as *item(3)/quantity*, if you want to follow the syntax of XPath. But this can get messy very quickly.

Unfortunately it's generally neither a good idea to just serialize the parameters, as the various tools within the messaging system don't know the classes they have to transport; i.e. you can't inspect the messages, let alone build things like a content based router [8].

One established technology stack to transport structured data in a generic and platform independent way, is XML. But in order to easily marshal multiple parameters (and the method name) into an XML document, we'll need an additional step: The parameters of the adapter method are stored into a POJO that's annotated properly for JAXB to do the marshaling. Then it's easy to store it into a *TextMessage*. Our sample method would be converted into the POJO of Listing 2. These POJOs too are packed into the API jar, so the receiver can use them to unmarshal the document again.

Interestingly the adapter methods can now be implemented in a generic way. For the sender adapter a java proxy can take the method name and the parameters of the method call and write them into an instance of the POJO. The receiver adapter can go the opposite direction and use reflection to pass the fields of the POJO as parameters to the target business method. The code for this is slightly beyond this article, as you'd have to find the proper POJO constructor or the proper method to call.

Message-API

In the Message-API project mentioned above, of course you'll find implementations for those generic adapters. But there still remains the considerable amount of code to be written for the POJOs. The Message-API provides an annotation processor that automates this work: You just annotate the business interface as *@MessageApi* and the annotation processor generates matching POJOs for every method within. You can run the processor with your normal *javac* call, in Maven, or even directly within Eclipse. It checks that all of your business methods return *void* and don't declare any exceptions. Eclipse displays these errors just as compile errors directly in your editor.

It's hard to believe, but of all the code that you had to write in order to send a JMS, only one annotation and a call to a factory method remain (Listing 3). The name of the queue can be configured, e.g. in an xml file (Listing 4).

On the receiver side, you only have to write your business code implementing the interface (Listing 5). At the moment you still have to write an MDB and annotate it for the correct queue (Listing 6). This MDB just forwards the message to the generic receiver adapter that unmarshals the xml back into an instance of the POJO, finds the matching method from the interface, and calls your code with the parameters from the message. But we work on making even this MDB superfluous.

In order to extend a Java EE application that is built with Maven to sending JMS messages, you only need these three steps:

1. Create a new module from the maven archetype *net.java.messageapi:api-archetype* and add your business interface. Maven already generates the POJOs and packages everything as an API jar.
2. Declare a dependency from your application to *net.java.messageapi:adapter* and call the factory method *MessageSender.of* in your business code.
3. Add the config file with the settings for the queue.

Communicating with "legacy" systems that send or receive e.g. *MapMessage*s, is just a change to the

Listing 2

```
@XmlRootElement
public class CreateCustomer {
  @XmlElement(required = true)
  private final String firstName;

  @XmlElement(required = true)
  private final String lastName;

  ...
}
```

Listing 3

```
@MessageApi
public interface CustomerService {
  public void createCustomer(String first, String lastName);
}

public class SenderBusinessCode {
  private CustomerService customerService =
       MessageSender.of(CustomerService.class);

  public void businessMethod() {
    ...
    customerService.createCustomer(firstName, lastName);
    ...
  }
}
```

Listing 4

```
<jmsSenderFactory api="CustomerService">
  <destination name="CustomerQueue">
    <factory>ConnectionFactory</factory>
    <user>guest</user>
    <pass>guest</pass>
    <transacted>true</transacted>
  </destination>
  <xmlJmsPayloadHandler/>
</jmsSenderFactory>
```

Listing 5

```
public class MyCustomerService implements CustomerService {
  @Overrides
  public void createCustomer(String first, String lastName){
    ...
  }
}
```

Listing 6

```
@MessageDriven(mappedName = "CustomerQueue")
public class CustomerServiceMdb
       extends XmlMessageDecoder<CustomerService> {
  public CustomerServiceMdb() {
    super(CustomerService.class, new MyCustomerService());
  }
}
```

configuration. By default the field names are derived from the parameter names, e.g. the parameter name *firstName* is mapped to the message field *FIRST_NAME*; but you can also configure the name of every field, just as you can configure converters. So it's easy to introduce the Message-API one-sided for only the sender or the receiver, which is often necessary when development is distributed.

Status

The Message-API is in productive use at 1&1 since august 2010. Since the adapters sit on top of JMS and don't reach into any internals, no problems have shown up, even in high load situations. Version 1.1 is available under the Apache 2.0 license since december 2010 [9] and we work on making everything available from a public Maven repository.

The solution of the Message-API seems so obvious to me, that I expect that at any time someone laughs at me saying: "This was available for years in project XXX". Namely Spring [10] is a project that I would have expected to have something like this, but neither the JMS Templates nor the message driven POJOs come even close – both mitigate the hassle with JMS but don't lift it to the business level. Other solutions like [11] start with the message format, but not with the business interface.

Currently Oracle seems to think about specifying version 2.0 of the JMS [12]. This would be a great chance to take a comparable step. Spring, too, would be a great candidate that is yet somewhat weak here... competition stimulates the business. Whether the Message-API becomes the reference implementation or not is insignificant – I just want to *easily* communicate safely and asynchronously.

Rüdiger zu Dohna is Expert Software Architecture at 1&1 in the department for DSL- and mobile order and delivery systems.

Prospect: Registry

Until now, the name of the *Queue* and the server that it runs on have to be configured by hand. In order to make this more easy, we work on a registry, so all configuration becomes optional.

We use the fully qualified name of the interface that is annotated as *@MessageApi*, and its version, that can be established from its package, i.e. from the *Specification-Version* in the manifest of the jar where the interface is packaged. When deploying a service, the container registers it with these coordinates. If the *Queue* does not exist yet, it is created on the fly. On the client side, a lookup with the same coordinates retrieves the name of the *Queue* and other configuration options.

So manual configuration is left only for special situations.

Links & Literature

1. http://jcp.org/en/jsr/summary?id=914
2. http://download.oracle.com/javaee/6/api/javax/ejb/Asynchronous.html or chapter 4.5 of the EJB 3.1 spec (jsr-318); not yet available in the Java EE tutorial.
3. http://docs.jboss.org/weld/reference/1.1.0.Final/en-US/html/events.html
4. http://www.allthingsdistributed.com/2008/12/eventually_consistent.html
5. Martin, Robert C.: "Clean Code", Prentice Hall, 2008
6. Fowler, Martin: "Refactoring", Addison Wesley, 1999
7. http://mockito.org
8. http://www.enterpriseintegrationpatterns.com/ContentBasedRouter.html
9. http://messageapi.java.net/pages/Home
10. http://static.springsource.org/spring/docs/3.0.5.RELEASE/reference/jms.html
11. http://www.w3.org/TR/soapjms
12. http://kenai.com/downloads/timeforjms2/JMS_BOF_Slides.pdf