A SURVEY ON THE OPEN SOURCE TOOLS FOR MODELLING AND IMPLEMENTING ENTERPRISE APPLICATION INTEGRATION SOLUTIONS

***Rafael Z. Frantz*[1]*, Rafael Corchuelo*[2] *and Fabricia Roos-Frantz*[1]**
[1]Unijuí University, Department of Exact Sciences and Engineering, Ijuí, Brazil
[2]University of Seville, Department of Computer Language and Systems, Seville, Spain

Abstract

Enterprise Application Integration aims to provide methodologies and tools to integrate the many heterogeneous applications of typical companies' software ecosystems. The reuse of these applications within the ecosystem contributes to reducing software development costs and deployment time. Studies have shown that the cost of integration is usually 5–20 times the cost of developing new functionalities. Many companies rely on Enterprise Service Buses (ESBs) to develop their integration solutions. The first generation of ESBs focused on providing many connectors and general-purpose integration languages whose focus is on communications, not on the integration problem being solved. The second generation of ESBs provides domain-specific languages inspired by enterprise integration patterns, which makes it clear that this generation is tailored to focus on the integration problem. In this chapter we provide a survey of Camel, Spring Integration, and Mule, which are the most successful open source second generation ESBs in the market. We report on them within a homogeneous framework that provides a clear overview of the three technologies.

PACS: 05.45-a, 52.35.Mw, 96.50.Fm

Keywords: enterprise application integration, open source ESBs, integration frameworks

1. Introduction

Typical companies run software ecosystems [1] that consist of many applications that support their business activities. Frequently, new business processes have to be supported by two or more applications, and the current business processes may need to be optimised, which requires interaction with other applications. Recurrent challenges are to make the applications inter-operate with each other to keep their data synchronised, offer new data views, or to create new functionalities [2].

Unfortunately, applications are not usually easy to integrate due to many reasons, e.g, the technologies on which they rely are different, their programming interfaces are not compatible from a semantic point of view, or they might not provide a programming interface at

all, which is the case of many web applications, legacy systems, and off-the-shelf software. Additionally, integration solutions must take four important constraints into account [3], namely: first, the resources and the programming interfaces of the applications being integrated should not be modified at all since a change might seriously affect or even break other business processes; second, they must keep running independently from each other since they were designed originally without taking integration concerns into account, i.e, no additional coupling must be introduced; third, the integration solution should interfere as less as possible with the normal behaviour of the integrated applications; finally, integration must be performed on demand, as new business requirements emerge and require support from Information Technology [4].

According to Weiss [5], the cost of integration is usually 5–20 times the cost of developing new functionalities. Software companies incur these high costs when they face their integration work using general-purpose languages and their accompanying workbenches, instead of using languages and tools that are specifically tailored towards solving integration problems. Software Engineers are responsible for devising these languages and tools. Domain-specific languages are intended to provide constructs by means of which a problem can be described at an abstraction level that is close to the conceptual level; later, the models expressed using these languages can be transformed automatically into low-level technologies.

Integration is expensive, but necessary as new applications sprout out. Usually, most of the functionalities and information involved in maintaining an existing business process or creating a new one can be found within a company's software ecosystem. The reuse of these resources within the ecosystem contributes to reducing software development costs and deployment time [6, 7, 8, 9].

Enterprise Application Integration (EAI) is a discipline whose focus is on providing methods and tools to integrate the many heterogeneous applications of typical companies' software ecosystems. With the sprout of the Service Oriented Architecture (SOA) [10] concept and supporting technologies, a new generation of enterprise application integration tools became available: the Enterprise Service Buses (ESBs). The first ESB generation focused on web services technologies. They provided many connectors that were configured by means of WSDL interfaces and a general-purpose integration language called BPEL [11]. Unfortunately, neither WSDL nor BPEL are domain-specific languages, which means that they provide a number of constructs that focus on communications, not on the integration problem being solved. In 2003, Hohpe and Woolf [2] compiled the first collection of enterprise integration patterns, which have became the defacto standard in this field. This has motivated some companies to work on second generation ESBs that provide domain-specific languages for integration. The most successful open source proposals are Camel [12], Spring Integration [13], and Mule [14].

Our purpose in this chapter is to provide a survey of Camel, Spring Integration, and Mule. We describe the architecture and the main concepts involved in each of these proposals. For the sake of clarity and comprehension, we use a case study that allows to compare these ESBs from a practical point of view. This survey of the technologies available presents the state of the art within a homogeneous framework in which their strong and weak points are highlighted. We further provide conceptual models of each technology, which may help readers to gain in-depth understanding of understand their design. As far as we know, no

account of such conceptual models has been found in the literature.

The rest of the chapter is organised as follows: Section 2 provides a background on the collection of integration patterns. Section 3 presents the Café case study, which we use as scenario to compare the open source ESBs. Camel, Spring Integration, and Mule are presented in Sections 4, 5, and 6, respectively. Finally, Section 7 reports on our main conclusions.

2. Background

An important contribution to the field of Enterprise Application Integration was done by Hohpe and Woolf [2] by means of their book on integration patterns. In this piece of work, the authors compiled several integration patterns that software engineers can use to develop their integration solutions. These patterns revolve around the concept of message, which is an abstraction of an envelop that can be used to transfer data from an application to another, and even to invoke its functionality. Integration solutions that are based on messaging allows for asynchronous communication between applications, which makes them loosely coupled.

The integration patterns documented by Hohpe and Woolf can be considered as the first-step to establish a common vocabulary within the Enterprise Application Integration community, which is expected to result in domain-specific languages. The patterns are described at a high conceptual level. Each one was given a name, a description of the context in which it can be used, and a description of how to solve a specific problem.

This catalogue of patterns has inspired Camel, Spring Integration, and Mule. In the following sections we, introduce the main categories of integration patterns.

2.1. Categories of Patterns

In their book, Hohpe and Woolf documented sixty five integration patterns that were classified into six categories, namely: message construction, messaging channels, message routing, message transformation, messaging endpoint, and system management. Integration solutions developed using these patterns also follow the architectural pattern Pipes and Filters [15]. The pipes are supported by messaging channels and the filters by the remaining categories of integration patterns. Below, we describe each category.

Message Construction. Messages are containers of data that flow inside an integration solution. Roughly speaking a message consists of two parts, namely: a header and a body. The header holds meta-data about the data that is carried in the body; it is the body that is expected to be modified, transformed, and routed through an integration solution.

The integration patterns in this category document the different types of messages that a software engineer may need to create, not only to transfer data amongst applications, but also to invoke functionalities, and send notifications. Furthermore, they document how to create messages to support request-reply communications and deal with situations in which a message must not be processed further since it can be considered stale.

Messaging Channels. Channels are part of the messaging infrastructure used to support the development of an integration solution, such as Java JMS [16] or Microsoft MSMQ [17]. They are used as resources to/from which messages can be written/read in total asynchrony. The writer and the reader can be either the applications being integrated or the integration solution. Simply put, a channel is a logical address that software engineers have to configure according to the adopted messaging infrastructure. A channel can be used by a single integration solution or can be shared by two or more solutions. Each messaging infrastructure may provide different types of channels and different configurations.

The integration patterns in this category document the use of channels for one-to-one and one-to-many communications, the setting up of a request-reply communication, how to restrict the type of messages a channels can receive, how to connect an application to the messaging system, how to deal with invalid messages or messages that have no readers, how to connect different channels in different messaging infrastructures, and so on.

Message Routing. Message routing comprises a set of integration patterns that allow to change the route of a message within an integration solution. The decision to which route a message has to go is usually made according to its contents. For this reason, the integration patterns have to inspect the body of a message; however, depending on the needs they can inspect the header as well. Some patterns can be configured with external contents, which are used to perform the routing of a message as well. An important characteristic of this kind of integration patterns is that they do not modify the contents of any messages.

The integration patterns in this category document how to route a message to a single or multiple destinations, how to define fixed or dynamic routing policies, how to process individually each element from a list hold by a message, how to combine the results of individual processing of related messages so that they can be processed as a whole, how to remove unwanted messages from the workflow of an integration solution, and so on.

Message Transformation. When integrating applications, it is not usual that they use the same data model. Thus, the differences in data models usually require to transform the contents of messages from one format into another, so that they can be understood and processed by the applications that receive them. In addition to these application-specific data models, integration solutions may involve other applications that adopt standardised formats that are independent from an specific application, such as RosettaNet [18], HL7 [19], SWIFT [20], and HIPPA [21].

The integration patterns in this category document how applications that have different data models can be integrated, how data from one application can be sent to another application if the original message does not contain the required data, how to simplify the contents of a message, how to process messages that have equivalent contents but are represented in different formats, how to minimise dependencies when integrating applications that use different data models, how to create message formats that are independent from any specific application, and so on.

Messaging Endpoint. Since the applications in a software ecosystem are not usually designed with integration concerns in mind, it is not likely that they can send and/or receive messages. Therefore, software engineers have to develop messaging endpoints, which are

pieces of code that interface an application and the integration solution, so that both can exchange messages. This piece of code has to be external to the application, since software engineers should preserve the applications unmodified.

The integration patterns in this category document several ways to interface an application and the integration solution to support communicating with one another. This may include interfaces that allow to compete for reading data from an application, to be selective when reading data, to provide an event-driven or a polling communication type, to provide transactional support to map objects onto messages, and so on.

System Management. Operating an integration solution that is running in production is a challenging task. An integration solution may process thousands or even millions of messages exchanged amongst several applications that may have their state changed by every message. Furthermore, there can be performance bottlenecks not only in the integration solution, but also in the applications being integrated due to the communication with the integration solution. To make things even more challenging, the parts involved in an integration solution communicate asynchronously, may be distributed within the software ecosystem, and may fail.

The integration patterns in this category document different ways to manage an integration solution. They document how to detect if a building block is failing, how to debug them, how to inspect a message without affecting its regular processing, how to track messages, and so on.

3. A Case Study

The Café case study has become the de facto standard to compare proposals from a practical point of view [22], cf. Figure 1. In this section we describe this case study, which we use in the following sections by modelling an EAI solution to it using Camel, Mule and Spring Integration.

The associated workflow describes how customer orders are processed in a coffee shop. Roughly speaking, it starts when a customer places an order to the cashier, who then adds the order to a queue. An order may include entries for hot and cold drinks, which are prepared by different baristas. When all drinks corresponding to the same order have being prepared, they are ready to be delivered by a waiter, who, in turn, delivers it to the original customer. Every order has a tray associated to it, which is used to deliver the order to the customer. Note that, the cashier is decoupled from the baristas, since orders taken from customers are placed in a queue from which the baristas take them. This allows the cashier to keep taking orders from customers even when the baristas are backed up. Baristas do not have a complete view of the whole set of drinks in an order, they receive individual drink requests and when a drink is prepared the barista places it to the corresponding tray. Every drink request carries the identifier of the order to which it belongs.

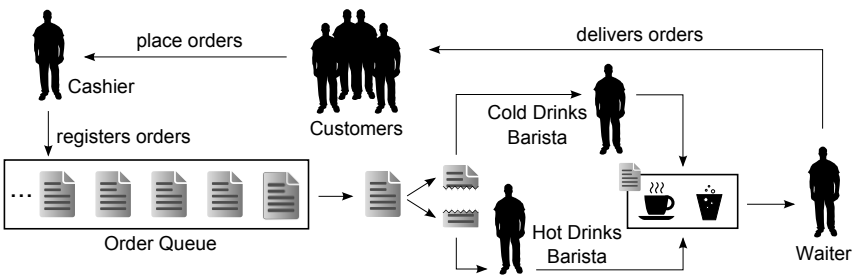

Figure 1. An abstract view for the Café solution.

4. Camel

Camel [12] is a Java-based software tool that aims to provide an integration framework with a fluent API [23] to support the design and implementation of EAI solutions based on integration patterns. It was designed to be used by means of a Java- or a Scala-based domain-specific language, or by means of declarative XML Spring-based configuration files. The Java-based domain-specific language approach is the most popular in the Camel community. Camel is an open source tool that is hosted by the Apache Software Foundation. FuseSource is the company that provides products based on Camel, which includes a commercial version of Camel, a web-based graphical editor, and an Eclipse-based IDE with a graphical editor.

Central to the Camel architecture are the concepts of exchange, endpoint, processor, and route. The conceptual model in Figure 2 shows these concepts and their relationships. Exchanges are containers of messages. They flow inside an EAI solution and carry messages from one processor to another. The messages contain data that endpoints read/write from/to the applications available inside a software ecosystem, from one processor to another. Processors execute atomic integration tasks on messages and are chained in routes, which represent complex integration tasks.

4.1. Exchanges

Exchanges are building blocks that wrap inbound and outbound messages. Every exchange must be set to a message exchange pattern, which can be either one-way or request-response. The former indicates that the EAI solution does not produce a response at the end of the workflow. On the contrary, the latter pattern indicates that the EAI solution returns a response. Processors consider the inbound message in the exchange for their processing. The result of the processing can be stored back in the inbound message or as an outbound message. If a processor stores a message in the outbound message of an exchange, Camel transfers the outbound message to the inbound message before passing the exchange to the next processor in the workflow. At the end of the workflow, if an exchange holds an outbound message and it is set with a request-response pattern, then Camel uses this outbound message to produce a response; otherwise, if the pattern is set to one-way, the outbound message is thrown away.

Messages have a header, a body, and attachments. The header contains meta-data in-

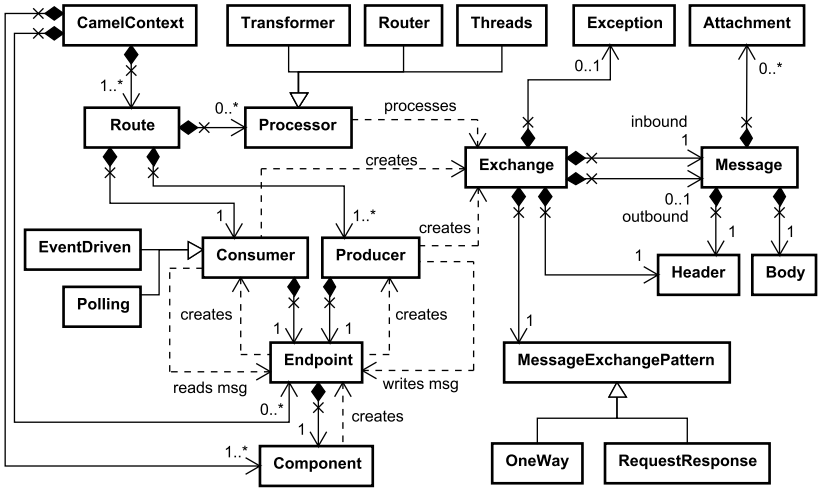

Figure 2. Conceptual model of Camel.

formation that is associated with the message, which is an arbitrary piece of data that can be used during the processing of a message. Headers are implemented as a map that stores data in the form of name-value pairs, which are referred to as attributes. The body allows to store the main data contents of a message. Both, header and body can be read or modified at any time during the workflow. Attachments allow messages to carry additional data that goes through the solution without further processing.

Similarly to messages, exchanges also have a header. The difference with regard to a message header is that it aims to store global-level data. This information is available to all processors in the EAI solution, independently from the inbound and outbound messages an exchange wraps. Camel uses this header to store information about the protocol being used to read the data in the corresponding message from an application, such as the encoding type, address, security permissions, and data that is related to service-level agreements. Note that when a processor creates a message, it does not contain the headers or the body of the message from which it originates unless the software engineer copies them explicitly. We provide additional details about processors in Section 4.3. Any `Exception` that occurs during the processing of a message is captured by Camel and stored in the exchange, so that information is available to be used in error recovery.

4.2. Endpoints

Endpoints are building blocks used at the beginning and the endings of the EAI solution workflow. They are used to connect applications from the software ecosystem to the EAI solution. Endpoints are created from components, which are responsible for implementing the low-level transport protocol necessary to read/write data from/to a particular resource. Camel provides an extensive list with more than 80 different types of components, including components for files, databases, e-mail systems, queues, enterprise java beans, remote method invocations, Amazon's simple storage service, HL7, LDAP, RSS, HTTP, and SIP.

Every endpoint provides an interface that allows to create consumers and producers of messages from/to endpoints. They provide a high-level interface that software engineers can use to perform read/write operations; however the semantics of these operations depend on the type of component used to create the endpoint.

When a consumer uses an endpoint to read from an application, it creates an exchange to wrap a message that contains the input data, adds information about the resource to the header of the exchange, and feeds the exchange into the route. There are two types of consumers, namely: event-driven, which provide an interface on which clients can invoke methods, and polling consumers, which are consumers that have to poll an application periodically to gather data from it, e.g, a folder or a database. A producer, receives an exchange and writes the inbound message to the application.

4.3. Processors

Processors represent the processing units inside the workflow of an EAI solution. They are building blocks that can transform and route exchanges in a workflow. Transformers are processors that change the payload from one format to another. Routers are applied to change the trajectory of messages in the workflow based on a user-defined criterion.

Threads are a particular type of processor. They are used to define a pool of threads in a certain point in the workflow to enable concurrency from this point onwards. The processors after this point are executed using threads from this thread pool. Camel allows software engineers to use this strategy to speed up the performance of the EAI solution in those parts of the workflow that consume more system processing. Unfortunately, using a threads processor breaks transaction boundaries, i.e, if such a processor is used, then transactions are not preserved.

4.4. Routes

Routes represent workflows inside integration solutions. Roughly speaking, a route is composed of a *consumer*, zero or more *chained processors*, and one or more *producers*. Every exchange a consumer creates is processed by the chain of processors preceding the target producers. If a route does not include any processor, then the route implements a simple bridge pattern [2] that reads data from an application and writes it to other(s).

The Camel context has a global view of the types of components available, the endpoints and routes that are created, and it is responsible for managing the execution of routes, i.e, it acts as the Runtime System of Camel. Every route involved in an EAI solution has to be registered to the context, differently from the endpoints and components, which are automatically managed by the context. Although an EAI solution can have several routes, every route is independent from each other, which means they can only exchange data by means of endpoints. In this case a producer of a route writes to an endpoint from which a consumer of another route reads.

Consumers are executed with their own pool of threads provided by Camel. By default, the same thread that is allocated to execute a consumer executes the whole route into which it feeds messages. In this scenario, at the end of the route, if the exchange has a request-response message exchange pattern, a response is returned to the application that has activated the consumer. This response is given back using the same thread that has

executed the consumer and consequently the whole route is executed synchronously. In scenarios in which a route includes a threads processor, when the execution of the workflow reaches it, the remaining execution can be as follows: a) if the current exchange has a one-way message exchange pattern, the current thread is released and the execution follows with a new thread; b) if the current exchange has a request-response message exchange pattern, the current thread remains blocked until the new thread finishes the processing of the remaining route. When this happens, the blocked thread is used to return a response to the particular application that has activated the consumer, unless the consumer's endpoint allows for asynchronous request-response.

4.5. Error Detection

In Camel, when an exchange cannot be processed, an exception is raised. Camel provides software engineers with two mechanisms to detect errors. The first uses try-catch constructors, which have to surround the code that can potentially fail. The second is more sophisticated and allows to configure an error handler based on a redelivery strategy and a dead letter channel, to which exchanges that have failed and cannot be redelivered are moved. By default, a dead letter channel is just a logger of errors, but it can be configured as a queue that stores exchanges that have failed, so that they can latter be read from this queue. An error handler can be configured on a global or per-route basis. At the global level, it gets the exceptions from every route, applies the same redelivery strategy, and uses the same dead letter channel independently from the route. Contrarily, if configured at the route level, the error handler allows for different redelivery strategies and dead letter channels.

4.6. The Café Integration Solution

Figure 3 shows the design of the Café case study using Camel's graphical notation.

The EAI solution was implemented using five routes that communicate by means of internal queues. The workflow starts at route (a). In this route, endpoint (1) is used to read orders from queue `direct:orders`. Every order is passed on to splitter (2), which breaks them up and generates new messages for every drink item in the order. The new messages that contain the drinks are written to the internal queue `direct:drinks` by means of endpoint (3). Route (b) was designed to read messages from `direct:drinks`, and, based on their contents, its recipient list routes messages whether to `seda:coldDrinks` or `seda:hotDrinks` internal queues. Recall that every route has always a producer endpoint, however in this route they are not shown in the graphical notation because the Java beans that implement the recipient list routing policy also act as producers by writing their response directly to the `seda:coldDrinks` and `seda:hotDrinks` queues. Routes (c) and (d) execute in parallel. They read messages from the `seda:coldDrinks` and `seda:hotDrinks` queues, respectively. The communication with the cold and hot baristas is done by means of endpoints (4) and (5), respectively. Their responses are sent to the `direct:preparedDrinks` internal queue, which is then the input for route (e). This route has an aggregator that aggregates all drink items from the same order into a single list, which is sent to endpoint (6). This endpoint builds a delivery message for the list of items, which is sent to endpoint (7), so that delivery messages are written to an application.

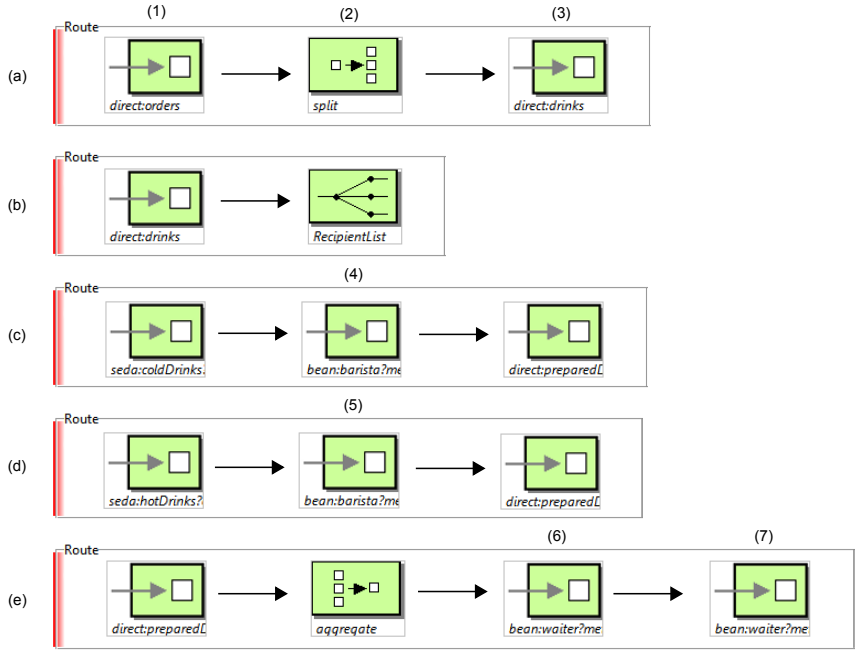

Figure 3. Café integration solution designed with Camel.

5. Mule

Mule [14] is a Java-based software tool whose architecture is inspired by the concept of enterprise service bus. It aims to support the design and implementation of Enterprise Application Integration solutions based on integration patterns. It was designed to be used by means of a command-query API [23] or declarative XML Spring-based configuration files. The latter seems to be the most popular and recommended approach by the Mule community. Mule is open source and provides a community version that includes an Eclipse-based IDE with a graphical editor. A commercial enterprise version is also available and maintained by MuleSoft Inc., which supports the Mule project.

Central to the Mule architecture are the concepts of message, endpoint, processor, and flow. Figure 4 shows a conceptual model that describes the relationships amongst these concepts and other elements around them. Messages encapsulate the data that endpoints read/write from/to the applications available within a software ecosystem. On reading data, the corresponding message can be processed by a series of processors that, in the end, write them to one or more applications.

5.1. Messages

Data that flows in a Mule EAI solution are wrapped into messages. Every message has a header that allows software engineers to add/read meta-data information associated with the message. The header is implemented as a map that stores data in the form of name-value pairs that are referred to as attributes. There is not a limit to the number of attributes neither a limit to the size of the meta-data stored in an attribute. The main data contents of

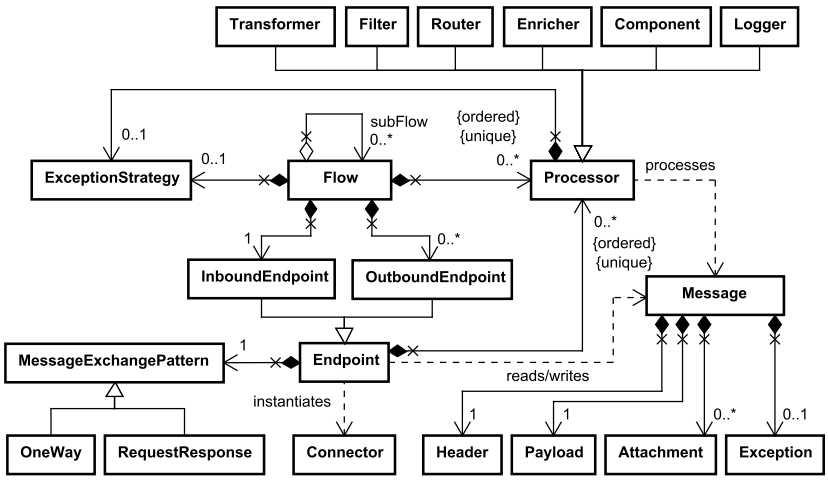

Figure 4. Conceptual model of Mule.

a message is stored in its payload, which holds an `Object`. Different from the header, data in the payload can be read and modified during the processing of a message in a workflow. Additional data that is not usually intended to be processed, but needs to be kept in order to produce an output message, should be carried as attachments. Messages also define an exception element to hold an exception that occurred during the processing of the message, so that this information is available to be used in error recovery.

5.2. Endpoints

Endpoints represent inbound and outbound points in an integration process. They correspond to a specific instance of a connector. Connectors abstract away from the technical details to deal with low-level transport protocols, which carry out the interactions with a particular type of resource. The most common types of transport protocols are supported by Mule, which provides a list of connectors that range from connectors to files, databases, queues, web services, to connectors for social networks, cloud infrastructures, and business process management systems. Endpoints provide software engineers with a unique interface to read/write messages from/to a variety of applications. By default, Mule creates a pool of threads for every endpoint, so that an endpoint can handle several read/write operations at the same time.

Both endpoints and connectors have properties that software engineers can use to configure them. Generally, this configuration is a tradeoff. A connector can be configured mixing properties regarding reading or writing operations, so that it is not necessary to have different connectors for each operation. Thus the kind of use (read/write) depends on the endpoint that uses it. On the contrary, it is necessary to configure an independent endpoint for each operation, although they can share the same connector. Although this way of configuration seems intuitive, it constraints the reuse of a connector. The reason is that if the connector is configured for both operations, it must have information about the applications from which it has to read or write and is tightly coupled with the applications.

Thus, to make connectors more reusable, it is necessary to configure independent connectors for each operation and take the information about the applications to the corresponding endpoints.

Software engineers can also configure processors to execute specific tasks inside endpoints. We provide additional details in Section 5.3. These tasks are intended to transform a message from a resource-specific format to a canonical format that is specific to an integration process, or to filter unwanted messages. In situations like these, the use of such processors aims to separate the wrapping logic for a particular resource from the integration process logic.

Endpoints support two kinds of message exchange patterns, namely one-way and request-response. The former indicates that the endpoint does not return a response when the message that has been read or written has been completely processed by the integration process. On the contrary, the latter pattern indicates that the endpoint returns a copy of the current outbound message. Not every endpoint supports both kinds of patterns, instead the support is constrained by the type of the connector that is used by the endpoint. For example, file connectors only support the one-way message exchange pattern, however HTTP connectors support both message exchange patterns (request-response by default).

5.3. Processors

Processors are building blocks that receive messages and do some processing with them. Every processor implements a small, atomic integration task taking into account the header, the payload, and/or the attachments of a message. Processors can be chained together to implement complex tasks that require several different types of processing on a message.

The processors supported by Mule can be organised into: transformers, filters, routers, enrichers, components, and loggers. Transformers are processors that change the payload from one format to another; filters can selectively filter some messages out of a workflow; routers are applied to change the trajectory of messages in a workflow based on a user-defined criterion; enrichers are used to add contents from external sources to a message; components allow to wrap `Objects` to re-use some functionality; finally, loggers write messages to a log system. Every category also provides a general implementation that can be extended and customised by software engineers according to their needs.

Transformers and filters are quite common in integration solutions, which is the reason why Mule allows to configure global transformers and filters. This is interesting in situations in which the same kind of transformation or filter can occur several times in the same or even in different workflows.

There are many processor types that are fully-configured by default. This is common for very simple tasks chiefly in the transformers category, such as the transformation from an `Object` into its XML representation, from a byte array to an `Object`, or from `String` to `Object`. However, other processors in these categories as well as in the filters, routers, and enrichers categories, can have their integration logic modified by means of scripting languages. The language used depends on the type of message: one can use XPath for XML messages, or OGNL, JXPath or Groovy for Java objects.

5.4. Flows

Flows in Mule are used to implement integration processes. They chain together endpoints, processors, and other sub-flows. A flow includes one inbound endpoint, zero or more processors, zero or more outbound endpoints, and an optional exception strategy. Flows that do not include a processor, implement a pass-through integration process that simply moves data from a source to a target (in this case, we assume that the flow includes an outbound endpoint). If a flow does not include an outbound endpoint, then the inbound endpoint must be configured with a request-response message exchange pattern or use a component processor that interacts with an external resource to write messages out of the flow. The exception strategy receives messages whose processing has failed.

As soon as a read operation terminates in an endpoint, the thread on which it runs is released. Then, the inbound message is made available to the first processor in the flow by means of an internal queue. By default, messages are processed synchronously in the chain of processors that compose a flow; therefore Mule defines a pool of threads to be used by these processors, as well. Software engineers can change this default processing model used in flows by defining asynchronous scopes that embrace all the flow or only part of it. Asynchronous scopes are sub-chains in which every processor in the chain runs in a different thread. Similarly to inbound endpoints, messages are made available to outbound endpoints by means of an internal queue.

Sub-flows are re-usable flows; the key is that they do not include endpoints; when they are invoked, the calling flow passes the current message to the sub-flow, waits for the response from it and then resumes processing. If the calling flow is executed synchronously, then the same thread running the calling flow runs the processors of the sub-flow for that message; otherwise the calling flow thread pool is shared with the sub-flow.

5.5. Error Detection

In Mule, if a message cannot be processed, an exception is raised. Mule allows to configure an exception strategy object at the processor and/or flow levels. Thus, when Mule detects an exception, it logs it, adds it to the message that has failed, and forwards the message to the exception strategy object. An exception strategy is configured to use an `OutboundEndpoint`, so that the message can be stored into a resource, such as a queue or database.

5.6. The Café Integration Solution

Figure 5 shows the design of the Café case study using Mule's graphical notation. As of the time of writing this chapter, the graphical editor is in beta version and does not provide support for some integration patterns.

The workflow of this EAI solution starts at file endpoint (1), which reads orders. Orders taken by processor (2) are split and generate new messages for every drink item. Then, the dispatcher processor (3) inspects every message in order to route them either to the `Barista Cold Drinks` (4) or to the `Barista Hot Drinks` (5). In this EAI solution, processors (4) and (5) are interfaces that allow to invoke the business logic that implements the baristas. The outbound messages from these processors represent drinks

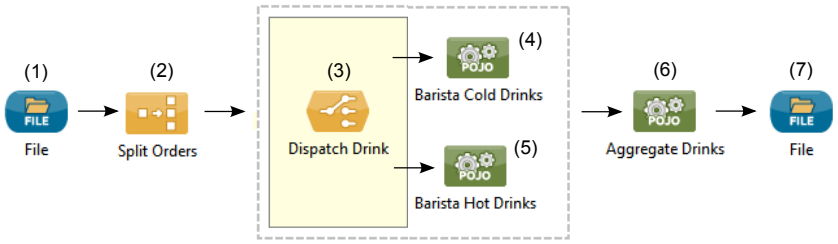

Figure 5. The Café integration solution designed with Mule.

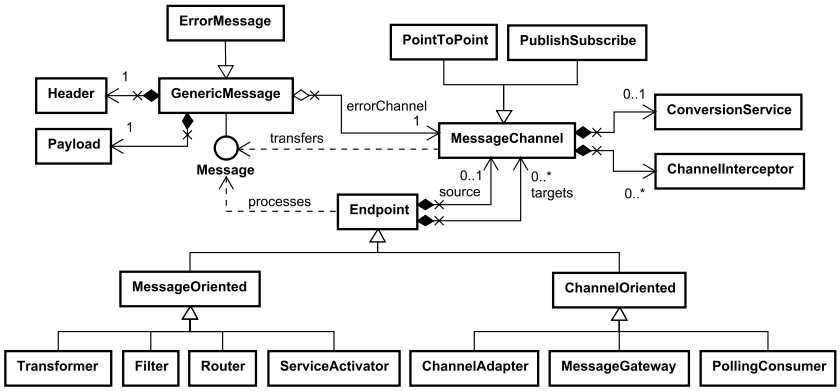

Figure 6. Conceptual model of Spring Integration.

that are prepared and then can be aggregated back into an order to which they correspond. Since the current version of the graphical editor does not support aggregators, we provide a ready-to-use aggregator, we have implemented the aggregation business logic in a separate Java class that is interfaced using processor (6). Finally, file endpoint (7) is used to deliver messages.

6. Spring Integration

Spring Integration [13] is a Java-based software tool built on top of the Spring Framework container. It aims to extend this framework to support the design and implementation of Enterprise Application Integration solutions. Following the philosophy of Spring Framework, Spring Integration promotes the use of XML Spring-based files to configure integration solutions, although it is also possible to use Spring Integration as a command-query API [23]. Spring Integration is an open source tool that includes an Eclipse-based IDE with a graphical editor. The tool is led and supported by SpringSource, a division of company VMware Inc. VMware does not commercialise an enterprise version of Spring Integration, instead they use individual Spring Integration components in their commercial tools, such as vFabric RabbitMQ and vCenter Orchestrator.

The architecture of integration solutions implemented with Spring Integration have to follow the Pipes-and-Filters design pattern [15]. In this pattern, messages flow through

several independent processing units (filters) that are communicated by means of channels (pipes). Messages are implemented with a building block with the same name, filters are implemented with endpoints, and pipes are implemented with message channels, cf. Figure 6.

6.1. Messages

Messages wrap data that flows and is processed in an EAI solution. Spring Integration defines a general interface for messages that aims to provide access to the header and the payload of a message. The header allows software engineers to add/read meta-data information associated with the message, and is implemented as a map that stores data in the form of name-value pairs, which are referred to as attributes. There is not a limit for the number of attributes neither a limit for the size of the meta-data stored in an attribute; however, once a message has been created its header cannot be changed, since it is immutable. The payload allows to store the contents of a message, which can be read and modified within the workflow. Although the API of Spring Integration is based on the command-query style, it provides a fluent API [23] to create messages.

There are two implementations for the message interface, namely: generic message and an error message. The former represents regular messages that flow in an EAI solution in normal conditions, whereas the latter represents messages that are created by Spring Integration when an error occurs during the processing of a regular message. To report eventual errors, generic messages are configured, by default, with a general error channel to which error messages are sent. This configuration can be changed by software engineers, so that error messages can be redirected to a different channel. We provide additional details about channel types in Section 6.3. Error messages are only created by Spring Integration and the difference between this type of message and a generic message is that the payload of the former must have an object of class `Throwable`, whereas the latter may have an arbitrary object of an arbitrary type.

6.2. Endpoints

Endpoints are building blocks that read, process, and write messages. They must always be connected to at least a source or a target channel. Roughly speaking endpoints can be grouped into message or channel-oriented endpoints. Message-oriented endpoints focus on performing a task on a message, possibly changing its contents. Endpoints in this group can be classified as transformers, filters, routers, or service activators.

Transformer endpoints aim to change an inbound message by transforming its payload from one format into another (e.g, from an XML document into a String), or by adding or removing contents to/from it. Filters apply a filtering policy, usually taking into account attributes in the header or the body, to evaluate whether a message can continue in the workflow of an EAI solution or it has to be dropped. Routers are used to decide to what channel(s) an inbound message should be written, to aggregate or split messages. Service activators are a very generic type of endpoint. They aim to wrap an arbitrary object as a service, so that messages in the workflow can be arbitrarily processed. A source channel is used to send messages to the service, and if the service returns a value, this is done by means of a target channel.

Channel-oriented endpoints focus on providing support to bridge the communication between applications and integration solutions, or provide functionality to access the internal channels of an EAI solution. Endpoints in this group can be classified as channel adapters, message gateways, or polling consumers. Channel adapters are responsible for reading/writing data from/to a particular type of resource. Their interface provides software engineers with a layer of abstraction on top of the low-level transport protocols necessary to read/write. Spring Integration provides several types of channel adapters, including files, databases, queues, web services, FTP servers, remote procedure calls, remote objects, HTTP servers, instant messaging systems, and social networks protocols. Message gateways aim to communicate with applications, however, they are used to provide a proxy that applications can use to push data to the EAI solution. This endpoints enable clients to work with objects instead of messages, since they can push objects to the endpoint and the message gateway is responsible for wrapping them into messages and write the results to the appropriate channels, or vice-versa.

Endpoints can write messages to a target channel, independently from the type of channel and how messages are transferred by the channel. Contrarily, reading messages depends on how messages are transferred. Roughly speaking, they can be transferred synchronously or asynchronously. Channels transfer messages synchronously by default, which means that messages are read by an endpoint as they are written by the previous endpoint. Polling consumers come into the picture when a channel that transfers messages asynchronously is used. In this case a polling consumer endpoint is necessary to check the channel for new messages. As long as endpoints communicate by synchronous channels, they are executed in the same thread; contrarily, endpoints that communicate by means of asynchronous channels can execute on different threads. In the latter case, the thread associated with the endpoint that writes the message to the asynchronous channel is released immediately after the endpoint completes the writing operation.

6.3. Message Channels

Channels are responsible for transferring messages between endpoints. By default channels do not put any restriction on the messages they transfer, however they can be configured to accept messages with only certain type(s) of payload. If a message with another type is received, then Spring Integration attempts to convert the payload to an acceptable type using a conversion service, either built-in or user-defined. If no conversion service is configured or the conversion fails, an `Exception` is thrown. Every channel can also define zero or more interceptors. Channel interceptors allow to intercept messages that are read or written from/to a channel without altering the workflow. This is an interesting approach for debugging and monitoring integration solutions.

Message channels can be classified along two axes: whether they deliver messages to a unique endpoint or not, and whether they are synchronous or not. Depending on the number of readers, message channels can be classified into point-to-point channels, in which there is a unique reader, and publish-subscribe channels, in which there can be an arbitrary number of readers. Synchronous channels require a writer and a reader to be ready simultaneously so that a message can be transferred through them; depending on whether the writer and the reader execute on the same thread or not they can be further classified into direct channels

and rendezvous channels. Asynchronous channels, on the contrary, decouple the thread that writes a message to them from the thread that reads a message from them. Asynchronous channels may be unbounded or bounded, and they can optionally deliver their messages using a user-defined priority criterion.

6.4. Error Detection

Endpoints can raise an exception during the processing of a message. Spring Integration allows software engineers to configure an `error-channel` attribute to an endpoint. Thus, when an exception is raised, the Spring Integration detects this exception, wraps it with an `ErrorMessage`, and sends the `ErrorMessage` to a channel configured to receive the errors. If there is not such channel, then an exception is thrown in the code and software engineers have to capture them with a traditional Java try-catch block in the Java source code.

6.5. The Café Integration Solution

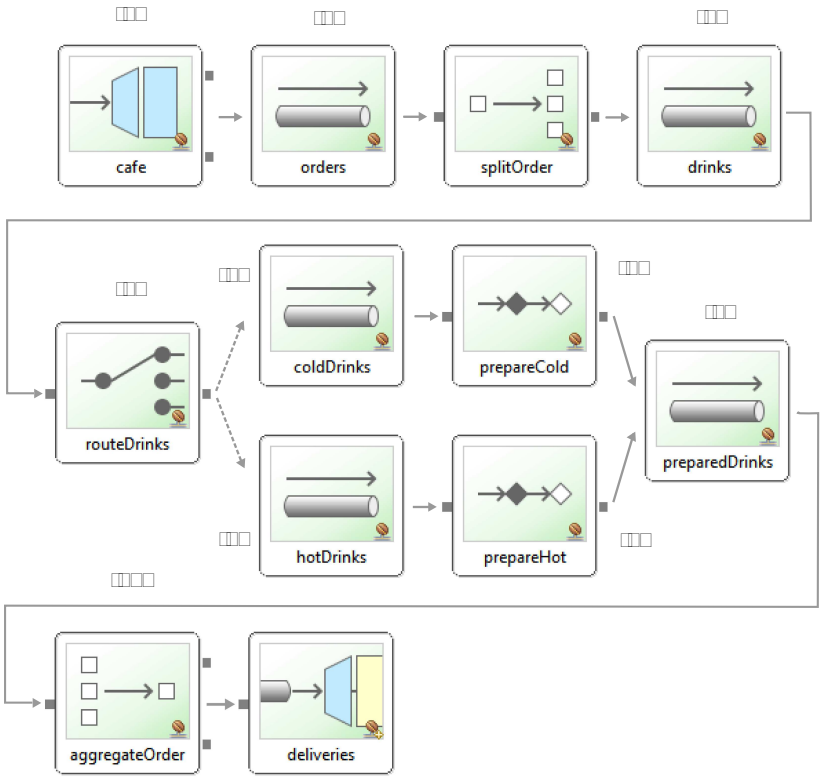

Figure 7. The Café integration solution designed with Spring Integration.

Figure 7 shows the design of the Café case study using Spring Integration's graphical notation.

The workflow of this EAI solution starts at message gateway (1). This endpoint allows to receive orders from external resources and writes them to channel (2), which is used to transfer the orders to the next endpoint, a splitter. The splitter breaks them up and generates new messages for every drink item in the order. Channel drinks (3) is used to transfer these new messages to router (4), which has to inspect every message in order to route them either to the cold drinks channel (5) or to the hot drinks channel (6). These channels are used to communicate with service activators (7) and (8), which then interact with external Java beans that implement the baristas that are responsible for preparing the cold and hot drinks. The responses from the baristas are sent to channel (9), which acts as a merger for the flow. Endpoint (10) is an aggregator that builds deliveries by aggregating all drink items from the same order into a new message. The last endpoint is a channel adapter used to write messages to an external resource.

7. Conclusion

Enterprise Application Integration (EAI) focuses on providing methodologies and tools to integrate the many heterogeneous applications of typical companies' software ecosystems. The first generation of Enterprise Service Buses (ESBs) focused on providing connectors that were used to integrate applications using general-purpose integration languages like BPEL [11]. Such languages provide constructs that focus on communications, not on the integration problem being solved. The catalog of patterns compiled by Hohpe and Woolf [2] inspired some organisations to work on a second generation of ESBs that provide domain-specific languages for integration. The most successful open source proposals in the market are Camel, Spring Integration, and Mule.

The lack of a common vocabulary to define the concepts involved in the context of EAI still represent a challenge for the communication amongst software engineers when designing and implementing integration solutions. The work published by Hohpe and Woolf [2] can be considered a first-step towards a common vocabulary. The DSLs provided by Camel, Mule and Spring Integration help to raise the level of abstraction when designing EAI solutions, however we have noticed that the beginning and ends of an integration flow in models designed using the DSL of Camel cannot be easily identified without the help of the corresponding source code. Camel and Spring Integration are the tools that largely support the integration patterns proposed by Hohpe and Woolf [2]. Mule, in its current version, still provides little support, however the concrete syntax of the DSL provided by Mule and Spring Integration are more intuitive than the one provided by Camel. The error detection mechanism in Camel, Mule and Spring Integration is mostly carried out by means of try-catch blocks. None of these tools include in the concrete syntax of its DSL building blocks to allow the detection and mitigation of errors.

References

[1] D. Messerschmitt and C. Szyperski, *Software EcoSystemm: Understanding an Indispensable Technology and Industry*. MIT Press, 2003.

[2] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003.

[3] F. B. Vernadat, "Enterprise integration and interoperability," in *Springer Handbook of Automation*. Springer, 2009, pp. 1529–1538.

[4] B. A. Christudas, *Service-Oriented Java Business Integration*. Packt, 2008.

[5] J. Weiss, "Aligning relationships: Optimizing the value of strategic outsourcing," IBM, *Tech. Rep.*, 2005.

[6] J. Li, R. Conradi, C. Bunse, M. Torchiano, O. P. N. Slyngstad, and M. Morisio, "Development with off-the-shelf components: 10 facts," *IEEE Software*, vol. 26, no. 2, pp. 80–87, 2009.

[7] T. G. Baker, "Lessons learned integrating COTS into systems," in *ICCBSS*, 2002, pp. 21–30.

[8] L. D. Balk and A. Kedia, "PPT: a COTS integration case study," in *ICSE*, 2000, pp. 42–49.

[9] T. Pfarr and J. E. Reis, "The integration of COTS/GOTS within NASA's HST command and control system," in *ICCBSS*, 2002, pp. 209–221.

[10] M. P. Papazoglou and W.-J. van den Heuvel, "Service oriented architectures: approaches, technologies and research issues," *VLDB J.*, vol. 16, no. 3, pp. 389–415, 2007.

[11] "*Web Services Business Process Execution Language Version 2.0*," 2007. [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

[12] C. Ibsen and J. Anstey, *Camel in Action*. Manning, 2010.

[13] M. Fisher, J. Partner, M. Bogoevici, and I. Fuld, *Spring Integration in Action*. Manning, 2010.

[14] D. Dossot and J. D'Emic, *Mule in Action*. Manning, 2009.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[16] M. Richards, R. Monson-Haefel, and D. A. Chappell, *Java Message Service*. O'Reilly, 2009.

[17] A. Redkar, C. Walzer, S. Boyd, R. Costall, K. Rabold, and T. Redkar, *Pro MSMQ: Microsoft Message Queue Programming*. Apress, 2004.

[18] "*RosettaNet home*," 2014. [Online]. Available: http://www.rosettanet.org

[19] "*Health level seven international home*," 2014. [Online]. Available: http://www.hl7.org

[20] "*Society for worldwide interbank financial telecommunication home*," 2014. [Online]. Available: http://www.swift.com

[21] "*Health insurance portability and accountability act home*," 2014. [Online]. Available: http://www.hipaa.com/

[22] G. Hohpe, "Your coffee shop doesn't use two-phase commit," *IEEE Software*, vol. 22, no. 2, pp. 64–66, 2005.

[23] M. Fowler, *Domain-Specific Languages*. Addison-Wesley, 2010.

Reviewed by

Dr. Vitor Manuel Basto Fernandes (vitor.fernandes@ipleiria.pt) - Polytechnic Institute of Leiria, Portugal and

Dr. Iryna Yevseyeva (iryna.yevseyeva@newcastle.ac.uk) - Newcastle University, United Kingdom.