Camel Design Patterns

Patterns, Principles, and Practices
for Designing Apache Camel Applications

Bilgin Ibryam

Camel Design Patterns

Patterns, Principles, and Practices for Designing Apache Camel Applications

Bilgin Ibryam

This book is for sale at http://leanpub.com/camel-design-patterns

This version was published on 2016-03-16

* * * * *

* * * * *

In memory of my father, without whom I would not be who I am today.

Table of Contents

Foreword

It has been an awesome journey to witness first-hand how the Apache Camel project has grown and established itself as the preferred integration library in the Java ecosystem. I started my Camel ride when the project was still new and was amazed how Apache Camel was able to put the EIP patterns in the driving seat using a clear and concise DSL that developers and architects alike could comprehend.

At the time of writing, the EIP patterns book was published over a decade ago, and Apache Camel will hit 10 years in 2017. All of us who are working in the IT industry are aware of how fast things changes. With the rise of new technologies such as cloud, containers, big data, IoT, social, and whatnot, there is a growing need for any business to adapt. And this requires being able to integrate all these systems faster and smarter.

As a seasoned consultant, Bilgin is out there in the field every day and witnesses first-hand all the good, bad, and ugly that takes place in the world. In this book, Bilgin is giving us a treasure with 20 modern integration patterns. Any architects or Camel developers who are building integration solutions can learn a lot from diving into this book and reading patterns of interest (hopefully all of them as the book is reasonably sized). Bilgin does not take the easy road and spare us of any of the harder parts of integration. I personally enjoyed reading all the wisdom from the error handling patterns.

As a closing remark I want to circle back to the EIP book that was the inspiration for the creation of Apache Camel. Now a decade later, it is thanks to Bilgin that we have a new set of EIP patterns to talk about in the Apache Camel community.

Thank you, Bilgin. Now dear reader, I certainly expect your Camel ride will be much better with these new patterns in your tool belt.

Claus Ibsen
Principal Software Engineer at Red Hat
Co-author of the Camel in Action books
http://www.davsclaus.com / https://twitter.com/davsclaus

Introduction

Regardless of whether you program in Java or .NET, create SOAP or REST endpoints, implement SOA or mircroservices, or deploy to cloud-based or on premise infrastructure, there are common patterns and principles used for designing and implementing integration applications. Over the years these have been named as messaging patterns, SOA patterns, cloud patterns, microservices patterns, resiliency patterns, etc. You can even classify some of them as principles or recipes. But it is important to be aware of them, understand the cost and the benefits, and make informed design decisions.

One aspect of patterns is that they evolve. With the advance of technology, hardware gets faster, compilers get more intelligent, and new languages and paradigms get created. The business wants to go faster and adapt more rapidly to the changing world. All of these forces are catalysts for new languages, frameworks, patterns, and best practices. A GoF pattern today can become an anti-pattern tomorrow. Think of the Singleton pattern and the Double-checked locking pattern; we hardly use them today as the Singleton creates hard-to-unit test classes, and there are better performant ways for concurrent programming than the Double-checked locking pattern nowadays. I am a pragmatist and believe that there are no best practices, only good practices in a context. This line of thinking should be applied to the patterns in this book too. Read through these patterns and before using one, think whether it adds more value than the effort it requires. If it doesn’t, do not be afraid of creating your own pattern.

The patterns listed here are not new; they are all over the internet, described many times under various categories and names. Here I describe them from an application integration point of view with emphasis on implementing these patterns in Apache Camel. The way I came up with the list of patterns was not by picking up cool patterns and trying to implement them in Camel, but rather the opposite. While working on tens of Camel projects, I saw the repetitive solutions used in each context without realizing they were well-known patterns. I have mapped these real world project experiences into existing patterns from various software domains, and backed them up with some context and samples. So the patterns in this book are described more from a practical point of view, rather than an academical pattern definition.

How this Book is Structured

It is worth setting your expectations right from the very beginning by saying that this book does not follow any specific pattern description language. It has been written in a relaxed style; it is similar to a series of essays, but with a consistent structure. English is not my first language, nor my second; as a result, the book is nothing like Martin Fowler's style, so bear with me. Each chapter has the following structure:

- **Name:** pattern name(s);
- **Intent:** a short description of the pattern;
- **Context and Problem:** when and how the problem manifests itself;
- **Forces and Solution:** how the pattern solves the problem;
- **Mechanics:** Camel-specific details related to the pattern;
- **More Information:** other information sources related to the topic.

As much as is practical, diagrams are used to depict the essence of the use cases and patterns. Rather than relying on Camel DSL, pseudo code or UML, Enterprise Integration Pattern icons are used as the primary notation in the diagrams.

Who this Book is for

Even though the title of this book is Camel Design Patterns, there is not one line of Camel code within it. There are plenty of good Camel books covering Camel syntax and the framework details at varying degrees (including my Camel starter book: Instant Apache Camel Message Routing). This book is intended for developers who are somewhat familiar with Camel but are perhaps lacking the wider application integration experience. It is based on use cases and lessons learnt from real world projects with the intention of making the reader think and become inspired to create better integration designs. There is no syntax in this book, only practical theory backed up by countless hours of riding Apache Camel.

I hope you find this book useful.

About the Author

Bilgin Ibryam is a software craftsman based in London, a senior integration architect at Red Hat, and a committer for Apache Camel and Apache OFBiz projects. He is an open source fanatic, and is passionate about distributed systems, messaging, and enterprise integration patterns, and application integration in general. His expertise includes Apache Camel, Apache ActiveMQ, Apache Karaf, Apache CXF, JBoss Fuse, JBoss Infinispan, etc. In his spare time, he enjoys contributing to open source projects and blogging. Follow Bilgin using any of the channels below:

Twitter: https://twitter.com/bibryam
Blog: http://www.ofbizian.com/
Github: https://github.com/bibryam/
Linkedin: https://uk.linkedin.com/in/bibryam

Technical Reviewers

Paolo Antinori is an ex-consultant and a current Software Engineer at Red Hat, working primarily on the JBoss Fuse product family. He likes to challenge himself with technology-related problems at any level of the software stack, from the operating system to the user interface. He finds particular joy when he invents “hacks” to solve or bypass problems that were not supposed to be solved.

Andrea Tarocchi is an open source and GNU/Linux enthusiast who now works at Red Hat. Computer-passionate since he was eight years old, he has studied Computer Engineering in Florence and has worked mainly in the application integration domain. If you would like to put a face to the name, check out his https://about.me/andrea.tarocchi profile.

Proofreader

Pete Gentry is a fully-trained professional proofreader/proof-editor, and the owner of Pete Gentry Editorial. He helps self-publishers, as well as a variety of other individuals and organizations, to make their texts as error-free, effective, and consistent as possible. If you would like to find out more about him, visit http://www.petegentryeditorial.com.

Cover Image

Bridges are engineering marvels used for centuries to bypass physical obstacles. They provide a passage over these obstacles without affecting the system underneath, whether that is water, road, or anything else. This is much like Apache Camel, which is used to connect difficult-to-integrate applications in a non-intrusive way.

The cover image is a photo from the Netherlands taken by **Christian Schoorlemmer** at 9:24 a.m., 25th December, 2005. Credits go to “FreeImages.com/Vormgeven-33659”.

I Foundational Patterns

A pattern does not always fit into one category. Depending on the intent, the same pattern may have various implications and contribute to multiple categories. The three pattern categories in this book are loosely defined, mainly to provide structure for the book.

The patterns in this category are used for structuring and organizing the integration flow, primarily for the happy path scenarios.

1. VETRO Pattern

The VETRO pattern has many variations. One such variation is the VETO pattern *[ESB VETO]*, which excludes the route step.

Intent

This combines multiple sequential actions taken on a message, into a consistent structure with well-defined responsibilities.

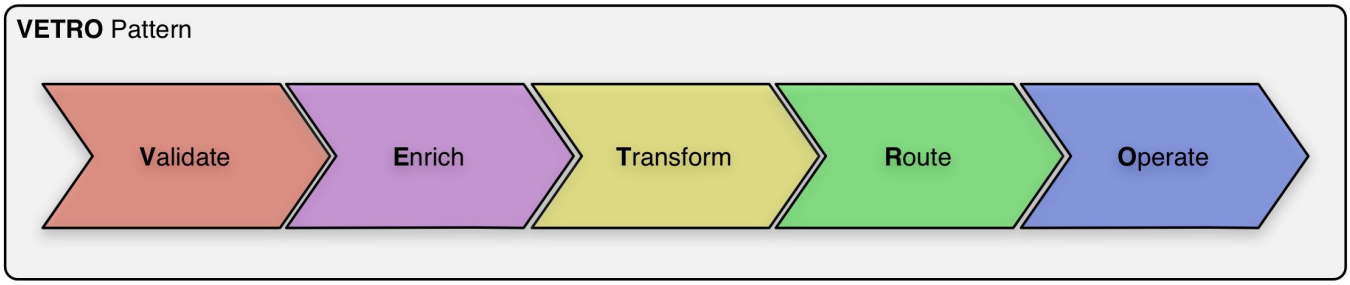
Context and Problem

Camel is a lightweight integration framework. In the same way that it can scale down and be used for ad-hoc protocol bridging only, Camel can also scale up and be part of a fully fledged ESB with hundreds of routes. When the number of Camel routes is that large, it is a different kind of challenge for all parties involved. The architects have to design new services with manageable complexities, the development team must develop, test, and debug multiple new routes in a short space of time, and the (dev)ops team must be able to understand and provide 24/7 support for all these processing flows in production. In such an environment, having a common structure for the Camel routes, following a strict naming convention, and having project-wide guidelines and principles becomes essential. Understanding a common structure will help you interpret the majority of the message flows; the common naming convention means that you will be able to reason about the purpose of a route by its ID, reason about the implementation of a `Processor` by its name, etc. Having project-wide guidelines and principles will ensure the team is not wasting time choosing from among many different kinds of DSLs, components, error handling policies, logging strategies, etc.

Forces and Solution

In a large project, if you analyze the Camel routes, you will notice that the majority have a similar structure and perform similar operations. Perhaps they differ in the transport protocol, the message format, the persistent store, and the target system, but the overall routing flow is quite common. Usually the message is received from the source system and validated; it is then enriched with some contextual information, transformed into the desired format, and acted upon immediately or routed to another system. Some flows may have more elaborate validation if the message comes from an untrusted system, and some may not require validation; some messages may require enrichment from external data sources, and some may not require any additional information at all. Some flows may have the data in the right format and some may require transformation; some messages

may require routing to other systems, and for some it might be the end of the flow.

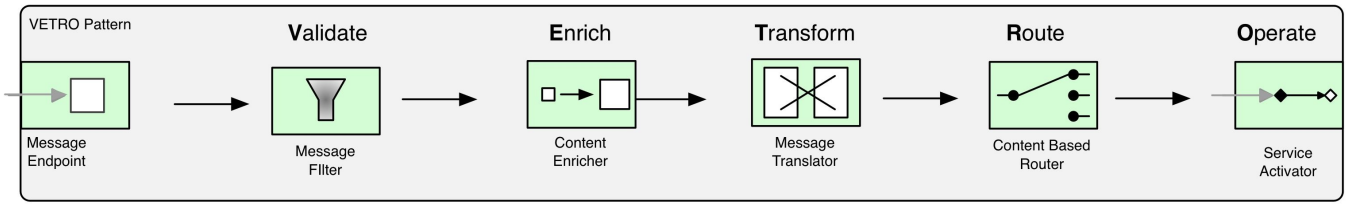


Whatever the specifics in your particular use case, the overall structure remains the same. These steps for processing messages are known under the acronym VET(R)O - Validate, Enrich, Transform, Route, and Operate. Let’s look in more details at each step:

- **Validate:** when a candidate message is received by a processing flow, it must first match certain criteria and qualify for processing. This is performed by the validate step. If a message does not qualify it is rejected and the rest of the processing flow is not invoked. Validate is an optional step that can be skipped if the messages are coming from a trusted source (another processing flow owned by the same team, for example) that is guaranteed to send valid messages. When validation is performed, it can be from a very lightweight format validation to a full-blown validation using reference data from a data source. If a message is not valid, it is either rejected with an exception thrown and propagated back to the caller, or it is moved to an invalid message channel. The first approach is preferred for synchronous Request-Response interaction where the message producer can receive back the rejected messages, and the latter approach is preferred for asynchronous Fire-and-Forget interactions where returning the error back to the message producer is not possible.
- **Enrich:** a message that has passed the validation step may not always contain all the necessary information required by the target system or by the current processing flow. Sometimes the incoming messages contain only an ID and this has to be used for looking up more data, and sometimes simply adding service-specific contextual information is enough to enrich a message. Enrich is an optional step that is used to give more meaning to the incoming data in order to enable its further processing.
- **Transform:** this step is used to convert the incoming message into a format that is easier to work with, or into a format that is necessary for the target system. Usually the incoming message is validated and enriched with more information so that it can be transformed into a format desired by the target system.
- **Route:** this pattern is also known as VETO pattern where the routing step is omitted. Indeed the route step can be omitted here or it can happen before the Transform or Enrich steps. This step gives some flexibility to the processing flow and allows routing of the incoming message to different paths usually driven by the message content (through Content Based Routing EIP).
- **Operate:** the last step is the core of the processing flow. All the previous steps prepare the incoming message for this step that performs an action. This can either take an action (for example, persist the message to a database) or deliver the message to another system. While the previous steps alter the in-memory copy of the message, Operate persists the message or passes it to another system.

Mechanics

Having understood the intention of each step, let's see how each of these steps is typically implemented with Camel.



VETRO Pattern expressed with EIPs

- **Validate:** as mentioned earlier, not every step of VETRO is mandatory and not every step needs to be performed explicitly. Sometimes certain steps are performed implicitly by the endpoint itself or as part of the routing process. For example, an Apache CXF-based SOAP consumer will do schema validation (and even message translation) as part of the message receiving process. But most of the other components cannot perform validation as part of the consuming process and the validation has to be an explicit step in the Camel route. Receiving a message from a message queue will not validate the message, nor will reading a file from a folder. If the data source is not trusted (for example, it is not an internal queue used to connect processing flows), messages have to be validated explicitly. Common components used for the Validate steps are:
    - *The validation component* for XML document validation through XSD schema validation.
    - *The schematron component* for more complex validations of XML documents. An XML schema can describe the structure of an XML document by specifying the valid elements, their order, and certain constraints on these. But XSD will not let you specify constraints such as the choice of attributes (when one of two attributes are required, for example), co-occurrence constraints (for example, when the value of an attribute has certain value then the element should have a child element; in other cases it should have a different child element). For these and other dependency validations between XML values, the schematron component can be useful.
    - *MSV and Jing components* can also perform XML document validation through RelaxNG XML syntax.
    - *The Java bean validation component* is a Hibernate implementation of the JSR 303 specification for validating Java beans.
    - If none of the available components fit with your message format, it is always possible to plug in a POJO and perform custom validation.
- **Enrich:** this step can be as simple as setting a hard-coded value to a header, or using `enrich and pollEnrich` Camel DSL constructs. `enrich and pollEnrich` can invoke a Camel endpoint to enrich the current exchange with more data. `enrich` uses a producer to invoke the endpoint (for example, a web service call), and `pollEnrich` uses a Polling Consumer to obtain data through the consumer (for example, it polls a folder for a specific file). Both of these constructs have limitations that prevent them from using data of the current Exchange (for example, they cannot use a message

header as the file name to poll), but that has been improved from Camel version 2.15 onwards. For older versions, a workaround for these limitations is to use a Recipient List with a custom Aggregation Strategy to enrich the current Exchange. Recipient Lists can be used for single or multiple endpoints and it supports dynamic endpoints (based on exchange data) too.

- **Transform:** Camel has an extensive set of transformation capabilities. There are tens of Data Formats (for message marshalling/unmarshalling), templating components(FreeMarker, Velocity, XQuery, XSLT, Mustache, Chunk, MVEL, String Template), Camel Simple Expression Language, Bean binding, etc.
- **Route:** this is another area where Camel DSL is shining. Using constructs such as Message Router, Dynamic Router, Recipient List, etc., the routing step can be expressed in a human readable fashion. This raises the question of which DLS flavour to choose: XML or Java? Both DSLs are equal in terms of capability and produce the same runtime model of the Camel flows. It is more a question about maintenance and operational support of the Camel routes. Java DSL is more powerful as Java language itself allows the use of inner classes and conditions. But with Java DSLs it is easy to create complex and hard-to-read integration logic. When the size of the development team is large, and there are many different parties interested in understanding the integration flow, implementing it in XML is the more feasible choice.
- **Operate:** this is the essence of the integration flow. Every integration flow has a business reason for its existence and usually that is expressed in the Operate step. One of the nice features of Camel is the ability to embed POJOs as part of the integration flows. Camel is a non-intrusive framework and the POJOs can become part of the integration flow without implementing any specific interfaces and having any annotation. Camel can do the hard jobs of data transformation and mapping the data to a Java bean method for you. So favour using beans without any dependency on Camel API (rather than implementing a `Processor`, for example), as these will be easier to unit test in isolation.

The key takeaway from this chapter is that you should strive to achieve a common structure for the Camel routes, and follow common naming conventions and coding practices in order to scale and manage the code base without creating complex integration flows with the powerful Camel DSLs. This begins from using common components, and having common route structure, common project modules, and application structure. When creating many services with a similar structure and a similar set of dependencies, using Maven archetypes can be helpful. Camel has a number of Maven archetypes *[CAMEL MVN]* used to create the most popular types of project structures. But creating you own Maven archetype based on your project structure and dependencies is easy as well. Then these custom Maven archetypes can be used to speed up the creation of new services and modules.

More Information

[ESB VETO] *The VETO Pattern - Enterprise Service Bus by David A Chappell*
[CAMEL MVN] *Maven Archetypes - Apache Camel*

2. Canonical Data Model Pattern

This is also known as the Common Data Model.

Intent

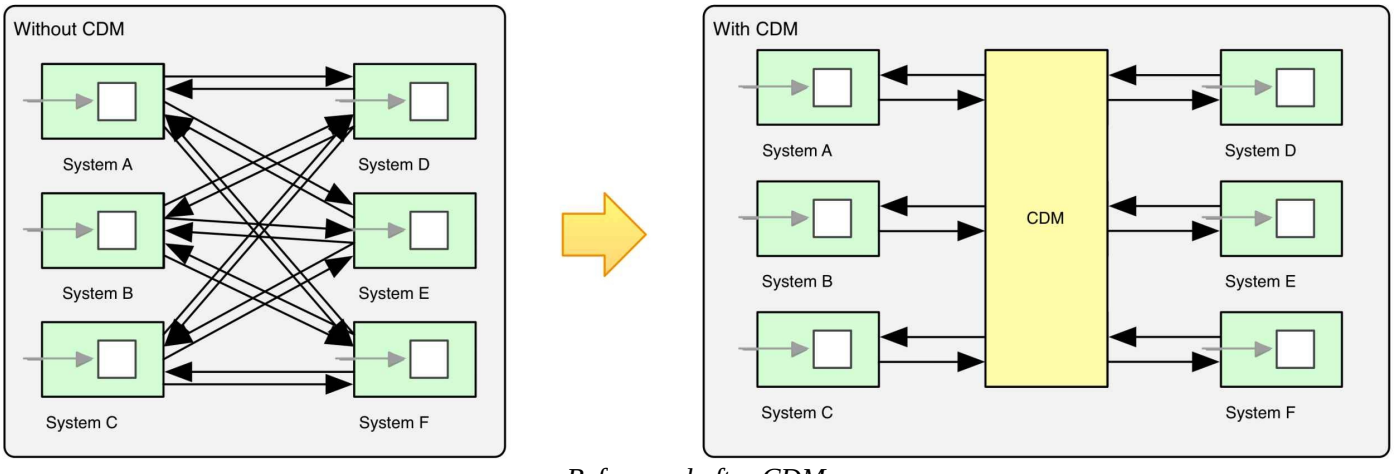
This pattern allows the minimization of dependencies between applications that use different data formats through an additional level of data format indirection.

Context and Problem

Applications written by different teams at different times, quite often using different programming languages, inevitably end up creating different formats and representations of the same data model. Middleware frameworks are perfectly equipped to solve the integration challenges imposed by such applications. Messaging channels provide temporal decoupling, the routing layer ensures location transparency, and message translators help convert data from the source to the target format. However, if there are a large number of applications that have to communicate with each other, the number of message translators can grow exponentially and make it difficult to add more data formats.

Forces and Solution

The Canonical Data Model *[EIP]* is the definition of all entities and the relationships that they have, designed and implemented in an application-independent way. Creating and maintaining a CDM that multiple teams have to agree on is a challenging process on its own and is outside the scope of this book. The CDM is independent from any specific application format but at the same time generic enough to cover all of them. By using a CDM we have a single language and model that all teams in the company can use, facilitating conversations and a common understanding of the data model.



Before and after CDM

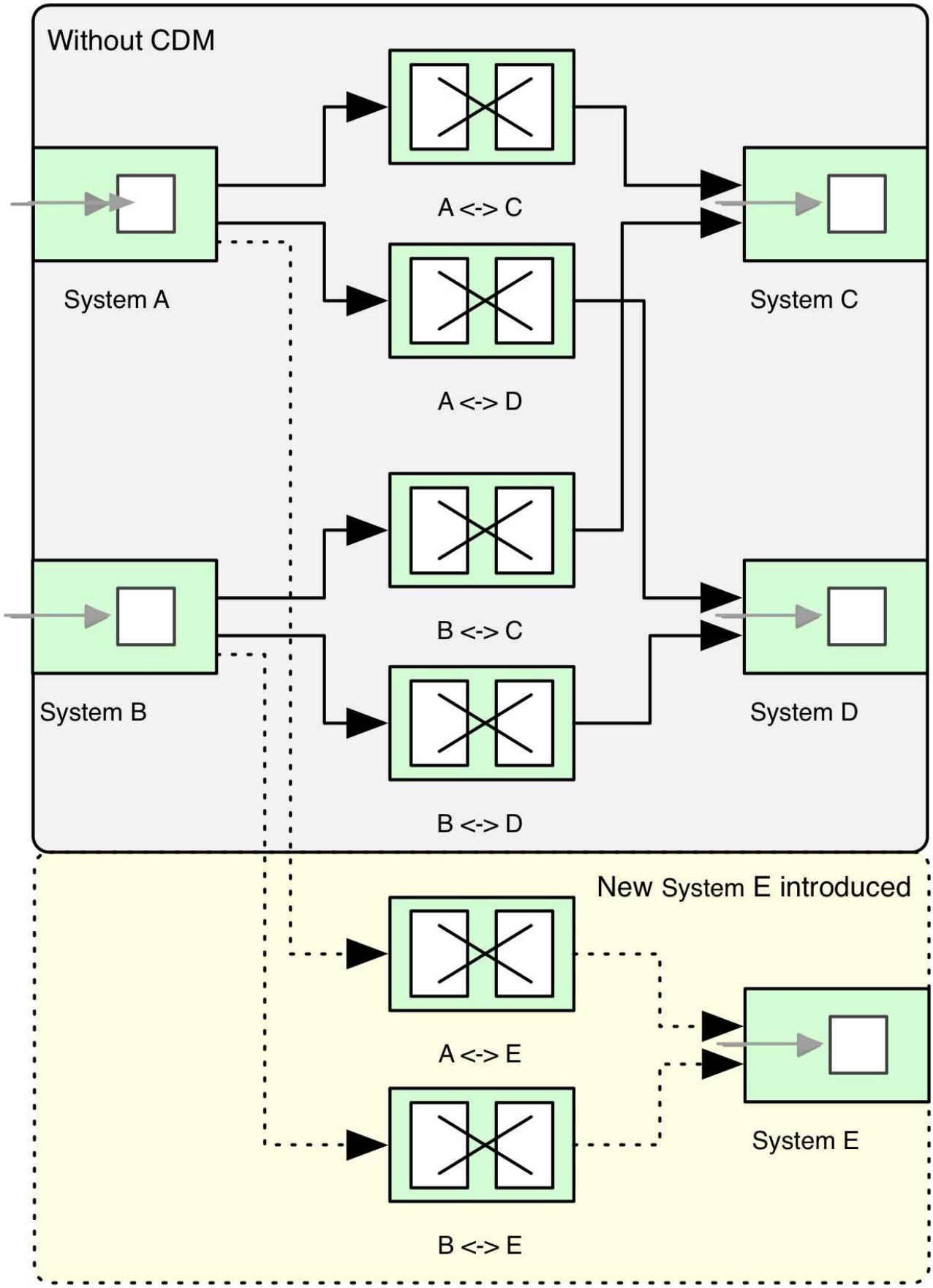
With this additional level of indirection between an application's custom data formats, when a new application is added to the mix, only transformation between the Canonical

Data Model has to be created regardless of the number of existing applications. This indirection of data formats also comes with a price. Adding one more translation layer introduces some latency caused by the extra processing steps. While having the CDM for small applications is a clear overkill, for larger applications it pays off. Thanks to the CDM, the application becomes easier to maintain and extend with more endpoints, but with the cost of extra latency.

The CDM is the ESB of the data layer. The diagram above with the CDM layer added looks very similar to diagrams visualizing an ESB. An ESB provides location independence through message routing, and transport independence through adapters and message channels. Similarly the CDM provides data format independence through a common data format.

Mechanics

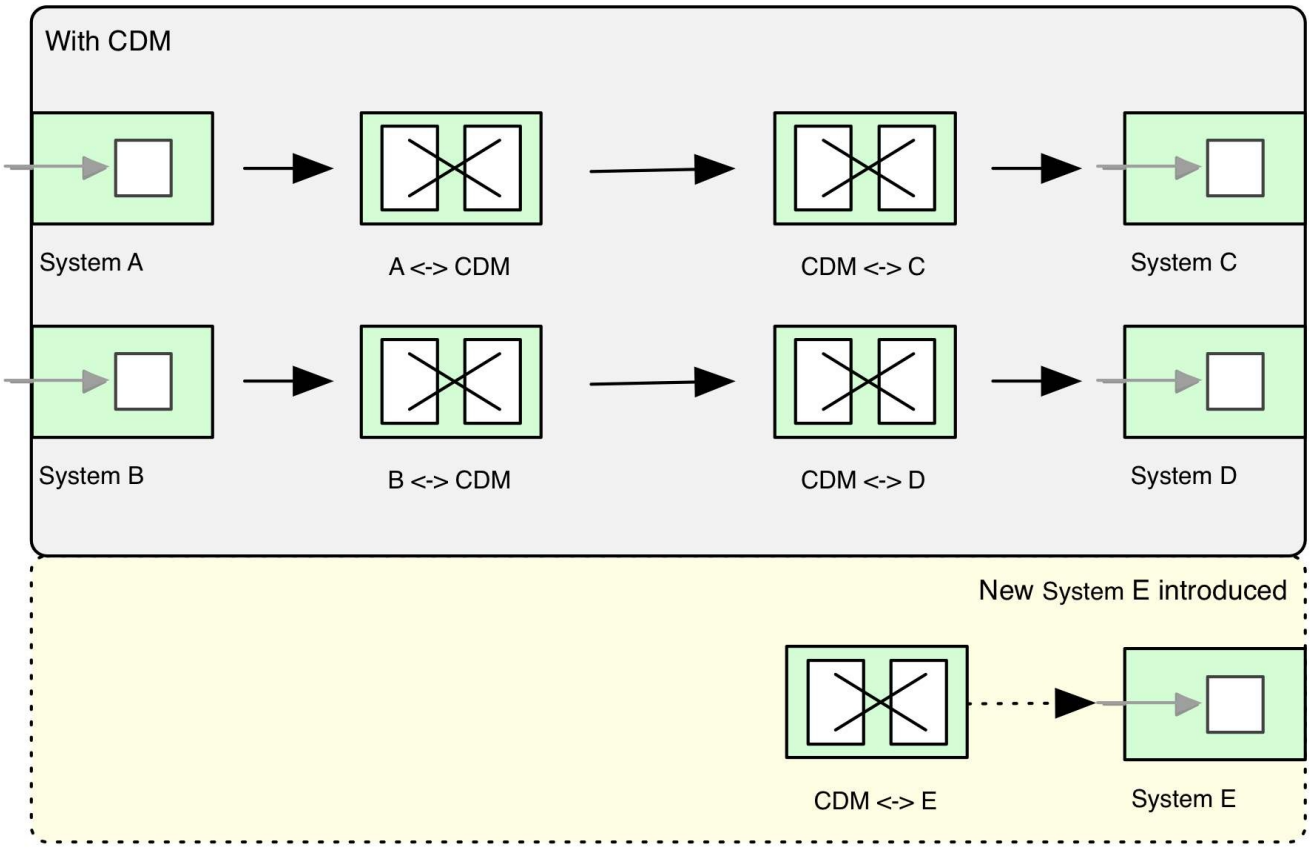
In the following diagram, we have an example where each source application (A and B) needs a message translator for each target system (C and D).



Introducing a new endpoint without CDM

If we decide to add another target endpoint (E) to the mix, it will require a message translator for each source system. While a Message Translator Pattern allows different source and target formats, it is tightly coupled to these formats across all endpoints using

it. A change in one data format (A) will require a change in all places where the data format is used (A <-> C, A <-> D, A <-> E). To break down this direct coupling between data formats an additional intermediary data format needs to be introduced.



Introducing a new endpoint with CDM

Once we have the CDM defined, the mappings between the individual application data formats are not needed any more. Instead, we map between the application and the CDM, making the addition of new endpoints a single mapping effort.
From a practical point of view, expressing the CDM as XSD schema works seamlessly. XSD is cross-language and can be shared by multiple teams in the organization, regardless of the preferred programming language. In the Java world, there are plug-ins (such as the `cxf-codegen-plugin` Maven plug-in) that can generate Java classes based on the schema. The schema can be used as part of WSDL for endpoint definition and also for message validation (for example, through a Camel validator component).
One way to implement the message translation from one model to another can be through a Camel Dozer Type Converter. Dozer allows a declarative way for defining the type conversion from one Java type to another. Camel also has many data formats (such as JAXB, XStream, Protobuf, and many others) to allow messages to be marshalled to and from binary or text formats. One data format that is a little bit different to the others is XML JSON Data Format (`camel-xmljson`) which can perform XML- and JSON-related conversion directly without requiring a POJO model. These are some of the tools commonly used in routes with a CDM.

More Information

[EIP] *Canonical Data Model - Enterprise Integration Patterns by Gregor Hohpe and Bobby Woolf*

3. Edge Component Pattern

This is also known as Multiple Service Contracts.

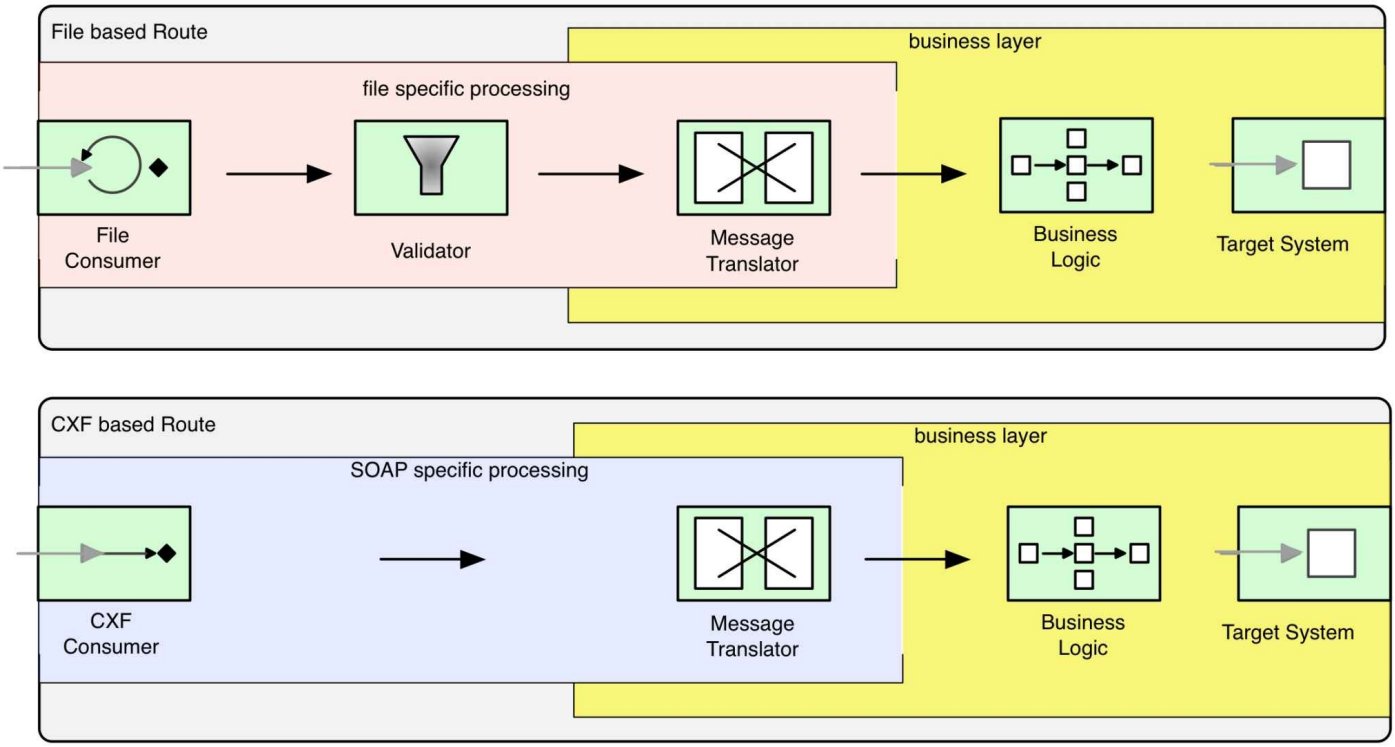
Intent

This encapsulates endpoint-specific details and prevents them from leaking into the business logic of an integration flow.

Context and Problem

Integration applications based on the Pipes and Filters architectural style divide a complex integration task into smaller independent tasks connected by channels. For small flows it is not important how the processing steps are organized, but for large and complex flows, the flexibility offered by the Pipes and Filters style can lead to a messy spaghetti structure that is hard to maintain and change in the long term. The Edge Component Pattern *[SOA PATTERNS]* helps us achieve a consistency and a reusability of the integration flow by isolating the business logic from the various transport endpoints.

A messaging channel receives messages from other systems through endpoints. Endpoints are the glue between disparate systems and they support a variety of technologies. Message endpoints are technology-specific and when they pass a message to a channel, the message and the surrounding metadata are also technology-specific. For example, a message generated by a CXF consumer (Apache CXF is an open source web services framework that is named after the original two projects Celtix and XFire) will contain HTTP specific headers; a message received from a JMS consumer will contain JMS headers. The problem arises when these endpoints and technology-specific data leak further into the business layer and couple business logic with transport-specific data. Such coupling of endpoint-specific data with the business logic limits the extensibility of a flow by mixing the different responsibilities; thus, of the transport layer and the business logic.



One business service with two input channels

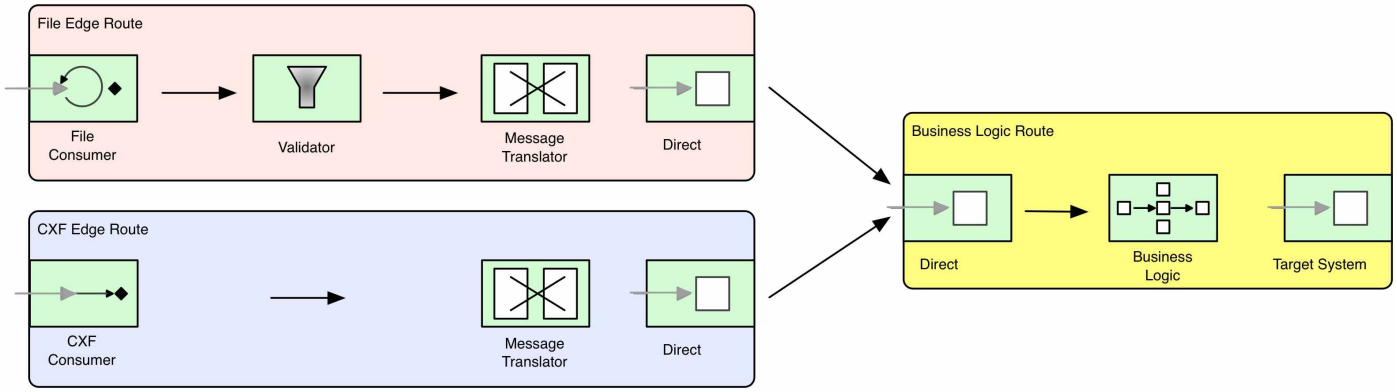
In the routes above, we can see how transport-specific processing has leaked into the business logic. When this happens, in order to add another endpoint, we have to duplicate the business logic for the new endpoint. Let’s see how we can fix this with the Edge Component Pattern.

Forces and Solution

The solution for the above scenario is to apply the “Separation of Concerns Principle”, or the “Single Responsibility Principle”, from the object-oriented world. In the integration and SOA world, one application of this principle is through the Edge Component Pattern. The idea is to have everything related to the transport and the endpoint isolated from the business logic of the flow. This is a way of organizing related processing steps in groups that are easier to understand and reuse. So operations related to the transport mechanism are carried out at the beginning of the flow, and the business logic is done in the next section of the processing flow. With this separation, the business logic is self-contained, and is easier to test and reuse. It also allows us to change the transport mechanism or add a new one without modifying the processing steps implementing the business logic.

Mechanics

Here is the previous example of Camel flow, refactored into two edge component routes and one business logic route. The edge routes carry out processing related to the message endpoint and the transport mechanism, and then pass the message to the second route that does some business-logic-related processing.



Service with two input flows implementing Edge Component

With this organization of the processing steps, it is possible to unit test the route that does the business logic in isolation, and regardless of the transport mechanism of the message endpoint. The code is also easy to extend; replacing the source endpoint and adding another (JMS, for example) message endpoint will not require change in the business logic flow. In a sense, it is similar to a layered application architecture where the types (such as `HttpServletRequest`) specific to the presentation layer are used, validated, but not passed to the business layer directly. In the Camel world, this means using message headers such as `CamelHttpMethod, CamelHttpUri, JMSCorrelationID, or JMSXGroupID` in the edge routes, and keeping business logic routes agnostic of the transport endpoints.

Typically one edge component will be represented by one route (rather than just the consumer). Based on the transport protocol and the consumer capabilities, there might be a need for validation and data transformation. For example, an edge component based on an Apache CXF consumer can do the validation with the schema and carry out payload transformation from SOAP to Java classes only with the consumer declaration and CXF endpoint configuration. But an edge component that is based on a JMS consumer cannot do validation and data transformation, so it will require a couple of additional processing steps. In addition the JMS-based transport is asynchronous and it is not possible to reject invalid messages as in HTTP-based CXF. So you would probably also have an Invalid Message Channel for a JMS-based edge component.

More Information

[SOA PATTERNS] Edge Component Pattern - SOA Patterns by Arnon Rotem-Gal-Oz

4. Command Query Responsibility Segregation Pattern

This is also known as Read versus Write Services *[SOA PRACTICE]*, or the Command Query Separation Pattern. The Pure Attribute Class in the .NET framework is a similar concept as it is used to indicate that a type or method is pure and does not make any visible state changes.

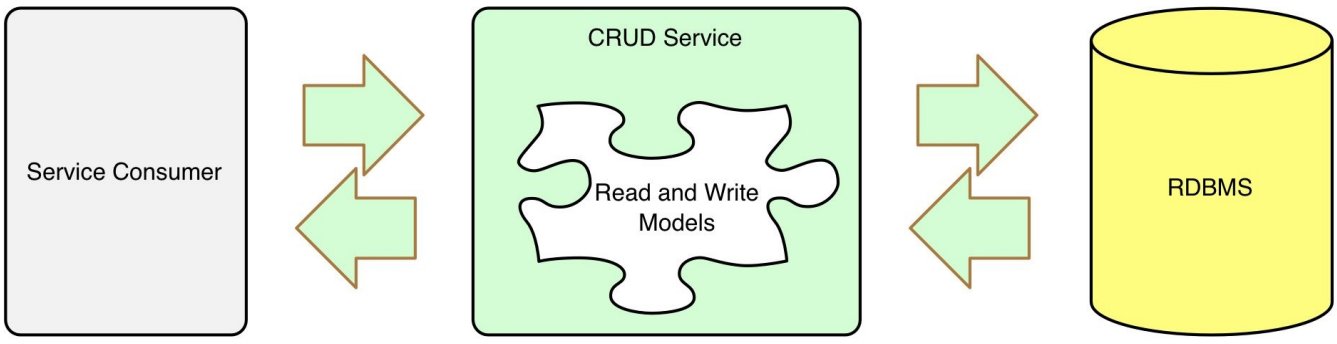
Intent

This decouples read from write operations to allow them to evolve independently.

Context and Problem

Quite often, services that deal with persistence are designed to provide CRUD operations. Regardless of the transport mechanism or technology, a service would have one data model and provide, create, read, update, and delete operations for that model. The operations can then be easily mapped to a REST style interaction, for example, and the persistence can be implemented with very little coding using an ORM tool such as Hibernate. This approach is easy to start with and allows creation of CRUD services quickly, but there are two areas which can become problematic in the long term:

- With the application becoming more and more sophisticated, one model representing both read and write data sets might not be feasible. Read models might have to represent data from virtual views which collapse data from multiple records. Write models may have to represent data that must be in a certain form in order to pass the validation rules. An API with two operations, which have started life with one data model representing the data from the persistent store, after a couple of iterations may need separate models to evolve independently.
- Depending on the application nature, different operations may require different quality attributes. For example, a service may be read intensive and require read operation to be scaled up on demand and be highly available.

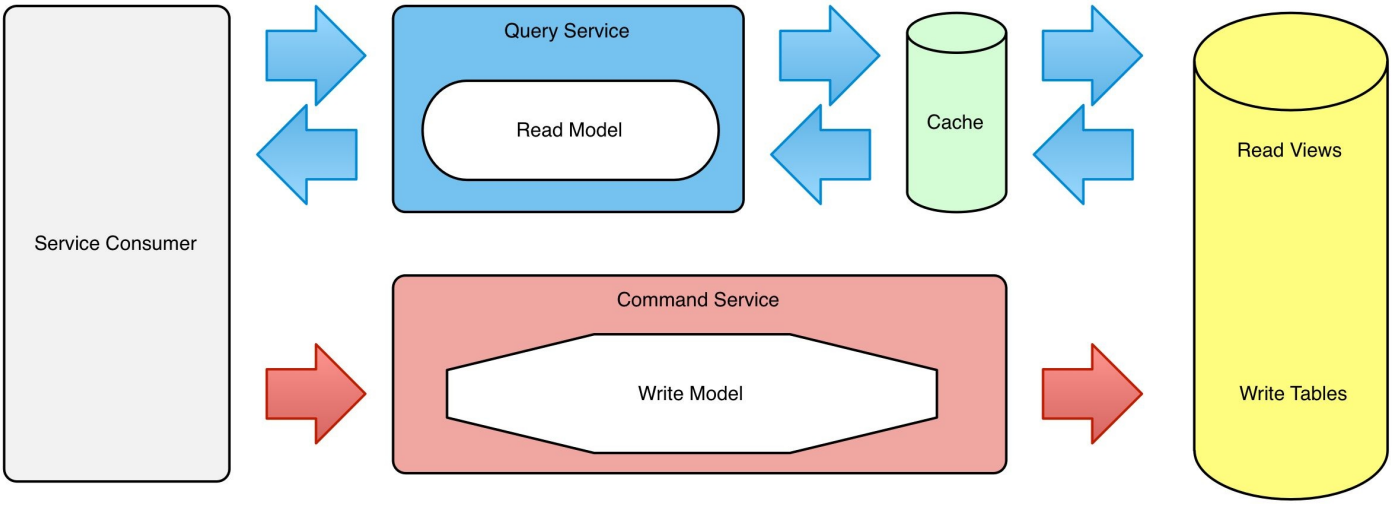


CRUD application with one model

A service with read and write operations tightly coupled at design time becomes difficult to evolve, and the scaling of the operations independently at runtime is also troublesome. To address these concerns the Command Query Separation principle can be used.

Forces and Solution

The Command Query Separation design principle is coined by Bertrand Meyer in his “Object-Oriented Software Construction” book. The main idea of this principle is that every method of an object should either be a Command to perform an action, or a Query to return data, but not both. The principle applied to the object-oriented programming model allows us to clearly separate methods that change state from those that do not. The Command Query Responsibility Segregation (CQRS) Pattern *[MSDN CQRS]*, coined by Greg Young, extends the CQS principle and applies it at architectural level. This pattern suggests the use of a separate model for mutating data and another for reading data. With this separation, the models, and the operations (which become services) using the models, can run in different processes on separate hosts. This allows services to evolve independently and be scaled on their own.



CQRS with read and write operations separated

Separating reading services from writing services is also gaining popularity with microservices. Reading services are less problematic as they are inherently idempotent and do not require transactions or compensating actions when they fail. They are easier to scale or improve the performance of through caching. Writing services on the other hand have different business and technical constraints which makes them different from the reading counterpart, regardless of the shared data model.

Mechanics

There are many long articles, training courses, and even books for implementing a full CQRS architecture. This pattern relies heavily on domain-driven design development methodology, the Event Sourcing Pattern, and the Materialized View Pattern. From an integration application point of view, the most useful principle of this architecture is the separating of the read and write operations. So we will look only at this aspect of CQRS. There is no specific framework feature or component in Camel that directly contributes to the isolating of read and write operations. Instead, it is a more general project-structuring

exercise driven by CQRS principles. When starting a project, it is tempting to use one artefact for both read and write operations. This can be the sharing of the same endpoint, the same component instance, the same `CamelContext`, or the same module. But before doing so, it is important to consider whether the operations have to evolve separately (from a development perspective), and whether the operations have to scale independently (from a runtime perspective).

- Scaling independently (or some other non-functional requirement) requires the creation of separate deployment modules (if using Maven, that is, Maven modules). So each operation has to be in isolated `CamelContext`, even the isolated Maven project. This will allow the operation to be deployed and scaled independently.
- Evolving independently requires even further separation. This effectively means creating a microservice for each operation (separate source code repository, separate build pipeline, etc.), so the operation can be treated as an independent project with its own life cycle.
- The read operation can benefit from a distributed cache (Camel supports many such connectors) that will allow horizontal scaling, be idempotent, and use materialized views.
- The write operation will most likely be transactional, and keep the cache updated when the data source is mutated.

More Information

[SOA PRACTICE] *Reading Versus Writing Services - SOA in Practice by Nicolai M. Josuttis*
[MSDN CQRS] *CQRS in MSDN*

5. Reusable Route Pattern

Reusability *[SOA PRINCIPLES]* is one of the main tenets of object-oriented and functional programming paradigms. It is also one of the main promises of component-based and service-oriented architectures. In this chapter we will look at this principle from a Camel application point of view, where it expresses itself as the Reusable Route Pattern.

Intent

To allow an agnostic business logic to be repeatedly used in different service contexts.

Context and Problem

Reuse is a common and overrated goal in many software projects used to avoid redundancy. It can be implemented at many levels, starting from methods and class level reuse, to complex processing flow reuse, and complete service reuse (as advertised by SOA and microservices usually). When implementing a complex system composed of multiple services, there might be bits of logic that are repeated in multiple places, in the same service, or different services. That repetition can vary from small endpoint configuration to complex sets of processing steps. Rather than duplicating the implementation (through copy and paste), and making it harder to maintain in the long term, it is better to reuse a single implementation in multiple places.

Forces and Solution

There are two main strategies for reusing an implementation in different places:

Development Time Reuse

With this approach, we need access to the reusable logic (whether that is the source code or the compiled binary artefact) at build time. At the end of the build process, the reusable logic becomes part of the using service itself. With this mechanism a business logic can be included and reused in multiple services. It is not always the case that the reusable implementation is needed at build time. It is possible to use some kind of abstraction such as a Java interface rather than the concrete implementing class, and have the dependency provided at deployment time rather than at build time. But the principle remains the same: an instance of the reusable logic is included in the main service and becomes part of it at development or deployment time.

Runtime Reuse

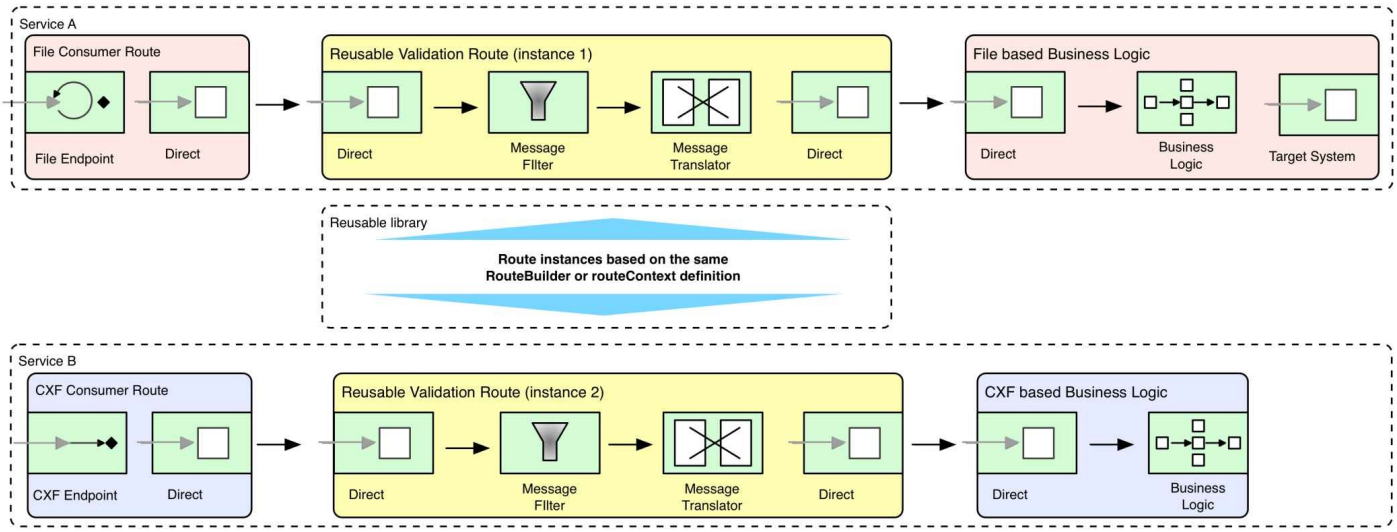
With this approach, there is no direct dependency to the reusable logic at build or deployment time. But at runtime, multiple services access and reuse the same service-agnostic business logic.

Mechanics

In the Camel world, the business logic is implemented in the form of Camel routes. So we will look at the different ways for reusing routes primarily.

Development Time Reuse

This is a quite common way of reusing Camel routes. It is very similar to having a Java class with the routing logic implementation and then instantiating it (either directly or by extending it) multiple times for different `CamelContexts`. Let's have a look at an example where we have two services which implement different business logic, but have to perform the same validation steps. And the validation is not a single component (such as Filter EIP or Validator), but rather consists of a couple of steps. We can encapsulate the validation logic in a Camel route and reuse it across both services.



Reuse of a route definition in multiple services

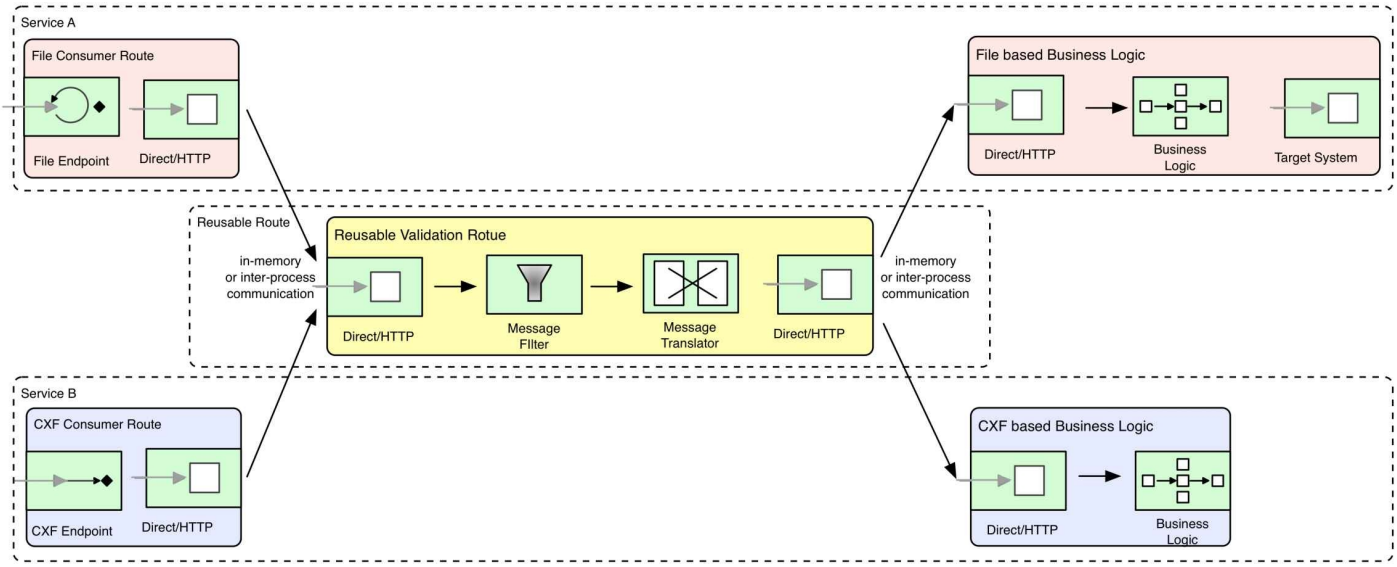
If Java DSL is used, the reusable route would be a `RouteBuilder` class that is instantiated in multiple services. If XML DSL is preferred, the equivalent of `RouteBuilder` would be the `routeContext` element *[CAMEL ROUTE CONTEXT]*. Remember, these two constructs are just the skeletons, i.e. the route definitions only. The actual runtime route instance will be created based on these definitions when they are referenced from a `CamelContext` (for example, `<camelContext> <routeContextRef ref="reusableXMLRoute"/> </camelContext>`). It is also possible to mix different DSLs; for example, if your main DSL is XML-based, then the reusable route can be based on Java DSL and included in the XML DSL through the routeBuilder ref (rather than routeContextRef being used to reference an XML route definition as following `<camelContext> <routeBuilder ref="reusableJavaRoute" /> </camelContext>`). Actually, it is better to have the reusable routes in Java DSL rather than XML DSL which has some limitations with regards to accessing onException, Intercept, and other global `CamelContext` constructs.

The advantage of the static reuse is that, based on a single implementation, we can create separate instances of the reused route at runtime. And that allows us to customize the behaviour of each reused route. For example, you can parametrize the endpoints and other options in the route, and before instantiating the route set those options that are specific for each service. In our example the File Consumer Service could customize the Reusable

Validation Route (instance one), for example, to do file-specific validation, and the Apache CXF Consumer Service could customize the Reusable Validation Route (instance two) for its CXF-specific validation. The downside here is that there are two separate instances of the same Camel route at runtime, and if, for example, both services share the same JVM, you will end up with two instances of the reused Camel route, and this is runtime duplication. If, for example, the reused route has an error handler with separate thread pools, or some other resource consuming elements, they will not be shared, but multiplied.

Runtime Reuse

The idea of this approach is to have only one instance of the reusable logic, and access that instance from different routes and services. Let's visualize the same two services from the previous example, but this time reusing the routing logic as a single instance.



Reuse of a single route instance

With this approach, we cannot customize the reused route for each service as the route instance is shared, but on the positive side, it will consume fewer resources. It is still possible to perform different behaviours per message basis by passing message headers to customize the behaviour of the reused route, but ultimately, there is only one route instance. Depending on the transport mechanism used to call a route from another one, there are different options:

- **Same CamelContext:** when all routes are in the same `CamelContext`, Direct and Seda components can be used to connect the routes. This simple mechanism allows developers to split a complex integration flow into smaller reusable routes with focused functionality. The reason for such a split might be functional (if the route requires a specific error handler, route policy, or other life cycle related customization), or simply organizational (to group related operations into separate routes for easier maintenance). You can imagine a `CamelContext` being like a Java class, and every route as a method in that class that can be called by other methods to reuse its functionality. This level of micro-reuse is a very common practice with Camel and usually driven by developer considerations.

- **Same JVM:** this is the case when Camel routes that belong to different `CamelContexts`, but are located in the same JVM, have to call each other. This may happen when a service is composed of multiple CamelContexts, or when different services (with different CamelContexts) share the same JVM. Sometimes, to reduce the latency and improve the performance of a system, it is possible to collocate a couple of services in the same JVM. In such a case, you can use the VM component, which is an extension of the Seda component, and acts as a JVM-wide registry for connecting routes. Notice that with these in-memory calling mechanisms there is no contract or boundary between the routes, and any route in a JVM can access any other route in the same JVM. This reuse mechanism is not that common as it introduces coupling between services that are not explicit, but it can be useful in some high latency scenarios.
- **Separate processes:** in the previous two scenarios, the reusable route was in the same JVM as the calling route, which simplified the connecting of them through in-memory calls. But it is also possible to implement a generic reusable functionality and deploy it on a separate JVM. This effectively turns the business functionality into a reusable service. Along with service autonomy and service discoverability, service reusability is one of the fundamental principles in SOA. As such, this method of reuse is common, but not specific for Camel applications. Accessing such a reusable route would usually be through HTTP for synchronous Request-Response style interactions, or through messaging for asynchronous interactions.

When there is only one instance of the reusable route that is accessed by multiple other routes, the route needs to know how to return the result to the different callers. This is where the Return Address EIP Pattern becomes useful. The idea of this is that, as part the request, the caller also specifies the return address to where the response should be sent. This can be a header that specifies the JMSReplyTo channel for a JMS interaction, the HTTP endpoint for an HTTP interaction, or simply the Direct component's endpoint name for an in-memory interaction. To dynamically build the target endpoint based on the message header, the reused route can use a recipientList or, from Camel 2.16 onwards, a dynamic endpoint through a toD construct (such as `<toD uri="${header.targetEndpoint}"/>`). When an in-memory interaction is used to call a route, there is not even a need to use a Return Address Pattern. Since there is only one Camel exchange instance passed from one route to another, the reused route can simply modify the exchange object, and when that route reaches its last step, the calling route will see the modifications done to the exchange. This simplifies the reuse of routes in the same JVM (usually in the same `CamelContext`), and allows the creation of reusable routes for repetitive tasks (such as setting or cleaning up headers, converting message type, validation, etc.).

You can reuse other constructs as well as routes too. The reusing of endpoints, error handlers, redelivery policies, and route policies is through the usual Java bean creation and reuse process.
In conclusion, remember that any reuse (other than reusing routes belonging to the same `CamelContext`) introduces a degree of coupling. Static reuse couples services to a common artefact at compile time, and dynamic reuse introduces a runtime dependency to a common service instance. When that common service is a different process, this also

introduces a performance hit caused by network latency and data serialization/deserialization overhead.

More Information

[SOA PRINCIPLES] Service Reusability - SOA Principles of Service Design by Thomas Erl
[CAMEL ROUTE CONTEXT] Route Context - Apache Camel

6. Runtime Reconfiguration Pattern

This is also known as the Runtime Variability Pattern.

Intent

This allows the externalizing and runtime variability of behaviour without requiring an application redeployment.

Context and Problem

A business service can be composed of multiple short processing flows connected together. It is possible to change the internal behaviour of a service without changing its public interface by rearranging the internal processing steps. In the Reusable Route Pattern we have seen how to reuse Camel routes by linking them together at deployment time. From an object-oriented point of view, that approach is similar to the structural Bridge Design Pattern where an abstraction (API in the integration world) and its implementation (the actual processing flow in the integration world) can vary independently. But sometimes, the structure and the processing steps of a service do not change; instead, the behaviour of the individual processing elements must change at runtime. Perhaps a Message Filter EIP will have to use a regularly updating blacklist, or there are dynamically changing destinations for a Content Based Router (CBR), or certain configuration parameters affecting the processing flow will have to be regularly updated by the business. In similar situations, we need the ability to alter and reconfigure the behaviour of a service without making code modifications or restarting the application every time.

Forces and Solution

The solution for the above challenges is similar to the Strategy Design Pattern from the object-oriented world. The Strategy Pattern lets you encapsulate behaviour and vary it independently from the clients at runtime. Bridge and Strategy patterns have a similar structure but differ in the problem they solve. Bridge is a structural pattern that decouples the abstraction from the implementation at compile time, whereas Strategy is a behavioural pattern that allows the changing of the behaviour of an object at runtime. The key for a runtime variability of behaviour is to encapsulate and externalize the varying behaviours from the main processing flow through an extension point. This allows the processing flow and the dynamic behaviour to have independent life cycles and interact through this extension point.

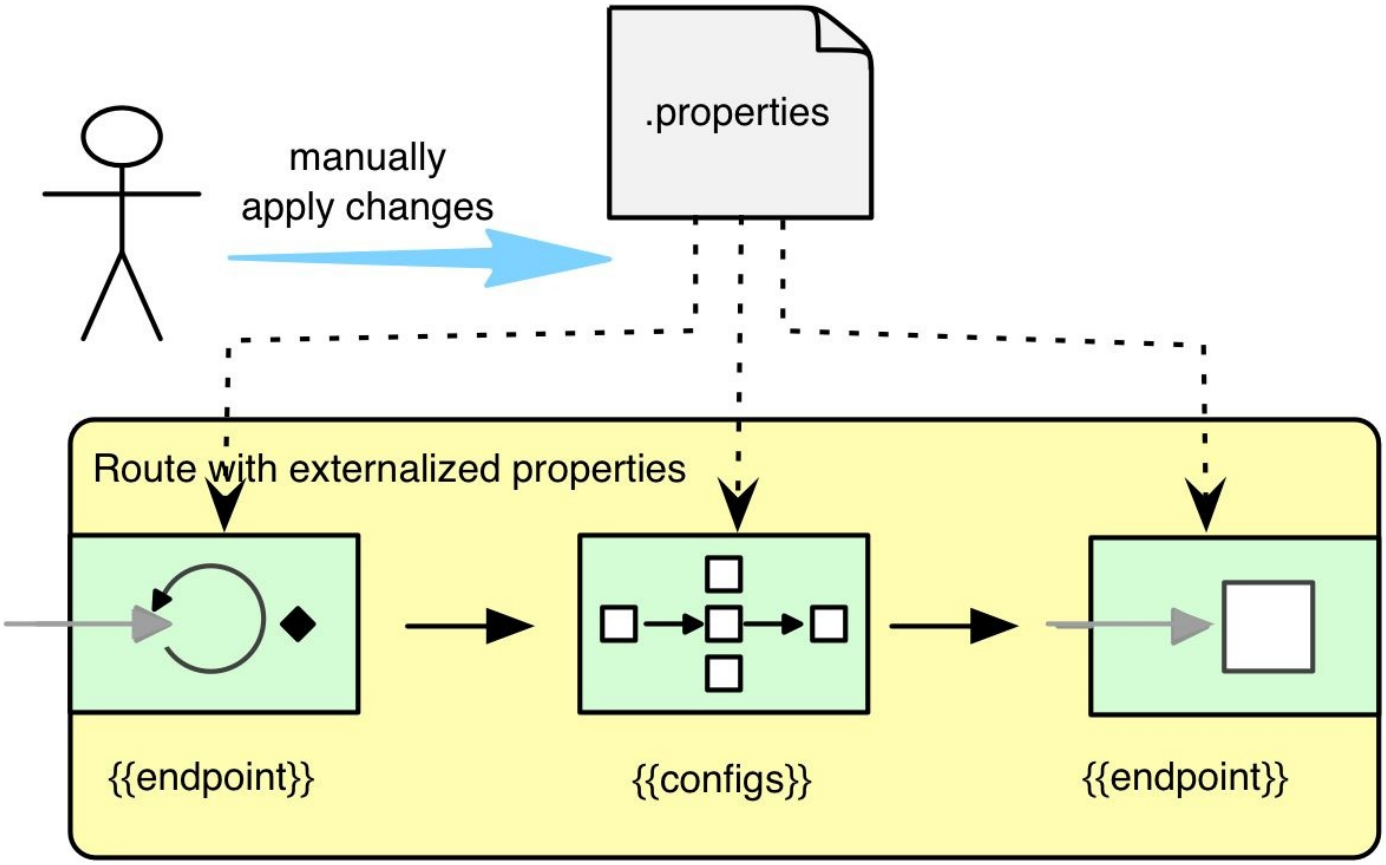
Mechanics

Let’s see a few concrete examples that demonstrate different ways of achieving runtime behaviour variability, i.e. the Runtime Reconfiguration Pattern *[MSDN RRP]*. We will

start from simple approaches such as reconfiguration through property files, and then go on to full dynamic runtime through the usage of a rule engine as part of the routing process.

Configuration-Based Variability

This category is a self-explanatory and obvious one, but we will mention it here for completeness. As described in the External Configuration Pattern, it is a good practice to use the Properties component to decouple endpoints and EIP configurations from Camel routes.



Properties based runtime variability

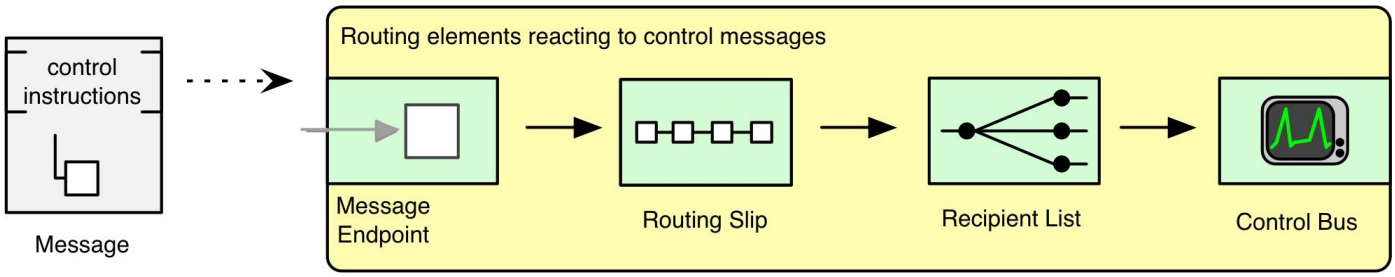
This allows the changing of these properties from environment to environment or from deployment to deployment without modifying the routing code base. A typical need for externalizing properties is when you move from your in-house test environment to the production environment that lives in the cloud. Without extracting all environment-specific elements into properties, such a move would be impossible.
Some Java containers such as Apache Karaf allow the updating of properties at runtime, and can even apply those changes without restarting or redeploying the whole application. That dynamism of Apache Karaf and OSGI combined with the distributed capabilities of Apache ZooKeeper is in the heart of the JBoss Fuse project, which is a perfect example of reconfiguring Camel applications at runtime with minimal downtime and no restart.
Another way for manually configuring a Camel-based application is through JMX. Camel has extensive JMX support and allows monitoring and management of the Camel runtime with a JMX client. There are MBeans for `CamelContext`, `Component`, `Endpoint`, `Consumer`, `Producer`, `Route`, `Processor`, `Tracer`, `Service`, etc., that can be used to

retrieve metrics and also to change their runtime behaviour by updating certain attributes. Using a JMX client to connect to remote JVM instances may not sometimes be possible because of security constraints, and secure tunnelling may be used. An easier solution here is to expose all MBeans over HTTP through Jolokia *[JOLOKIA]*. Jolokia is a JMX-HTTP bridge that gives access to all MBeans over a REST interface. This option is very often used for monitoring Camel applications through external tools which perform HTTP calls and gather metrics. Keep in mind that configurations performed through MBeans (whether that is directly through the JMX or the HTTP interface) is not persisted and will not survive a JVM restart. But it is a handy tool to use for urgent runtime tweaking and tuning while waiting for the next application release to happen.

Message-Driven Variability

Sometimes the routing steps in a flow vary widely depending on the message type and cannot be defined at design time. For example, depending on the message source, some messages may require additional validation logic or a different target destination. This is a well-known use case and there are EIPs to handle it:

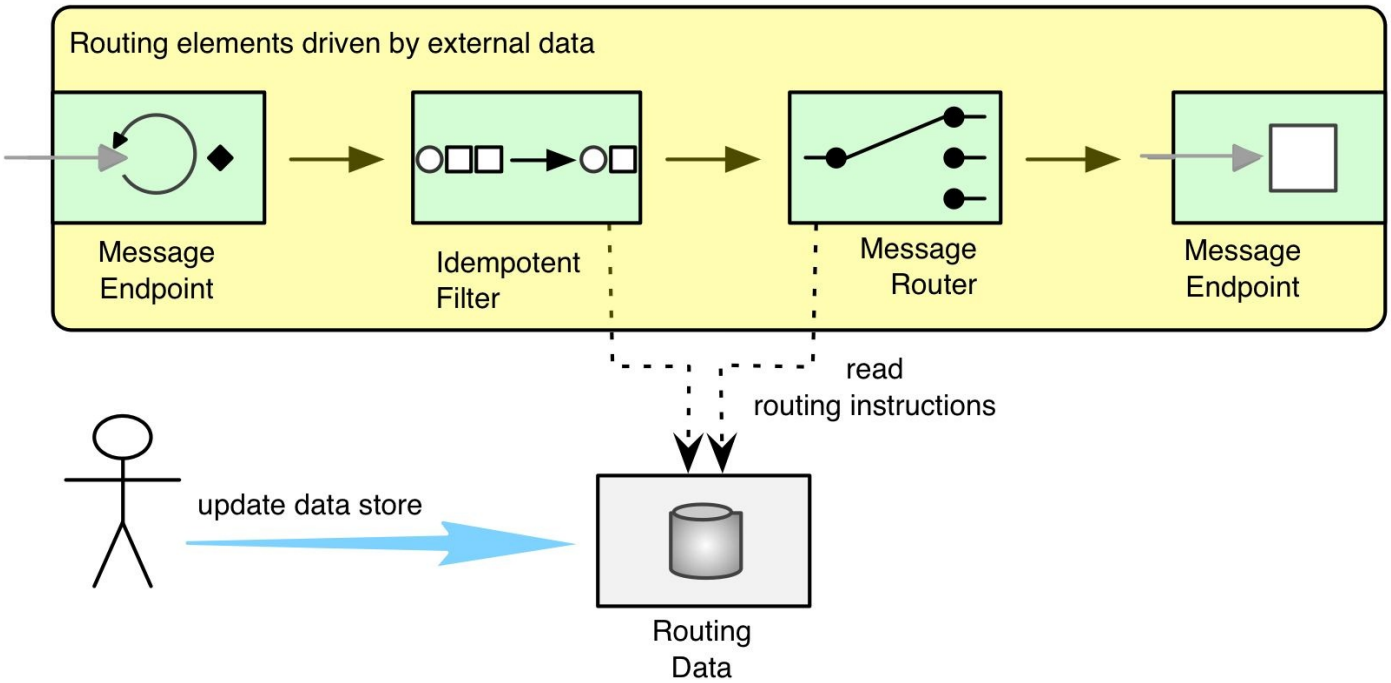
- **Routing Slip:** allows the routing of a message consecutively through a series of processing steps which are specified in the message itself. So rather than the route, it is the message that dictates which processing steps it should go through. In practical terms, it is implemented by sending a Camel exchange with a header containing a comma-delimited list of endpoints the exchange should be routed through. Then the Routing Slip will make sure that the exchange is routed through the list of endpoints as if it were a usual Camel route.
- **Recipient List:** this is another message-driven EIP that allows the routing of a message to a number of dynamically-specified recipients. In some respects, this EIP is similar to a Publish-Subscribe Channel where a publisher sends the same message to a number of consumers. But here the difference is that it is not the subscribers showing interest and subscribing to receive the message, it is the publisher who specifies the message receivers as part of the message. In the Camel world, `recipientList` is the dynamic counterpart of the `multicast` construct. With multicast the recipients have to be known in advance and hard-coded in the Camel route, whereas `recipientList` can calculate the recipients at runtime per message basis.
- **ControlBus:** this EIP uses the same infrastructure as the application data but separate channels to control and administer the processing flows. In the Camel world, this means using Camel routes to manage other Camel routes at runtime. ControlBus allows you to alter the runtime behaviour of a flow by starting and stopping other Camel routes.



In addition to these EIPs which are specifically designed to do dynamic routing, there are **other EIPs** in Camel that are also implemented in a way that allows them to be configured at runtime. Patterns such as Throttler, Delayer, or Detour can also evaluate the exchange data and alter their behaviour.
The common theme around all the patterns mentioned in this section is that they all allow runtime variability of the processing steps driven by message data. So the messages used to pass application data also carry some control instructions and alter the routing behaviour at runtime.

External Data-Driven Routing

Sometimes the routing flow is heavily dependent on data that is constantly changing. In this instance it is not feasible to constantly update configurations to keep up with change. Instead it is easier to use EIPs and Camel components that can use external data stores to drive their behaviour. This allows the external data store to be updated at a different pace to the application life cycle and the changes to take immediate effect on the routing flows. The most obvious example here is the Dynamic Router EIP *[CAMEL DYNAMIC ROUTER]*, which is kind of a CBR, but rather than having the routing rules hard-coded, it uses a data store that can be modified by other channels at runtime and alter the routing destination. An example would be to have a Content Based Router that looks up an external key-value store to decide where to route every message.



External data-driven runtime variability

Since Camel allows you to use Java beans in many EIPs such as CBR, Filter, Message Translator, Splitter, Aggregator, Content Enricher, Content Filter, Detour, Claim Check, etc., it is possible to make any of these use an external data store and change the runtime behaviour by updating the data. Typical use cases here are looking up a data store from a Message Filter for invalid requests (blacklisting or expired tokens), doing CBR using

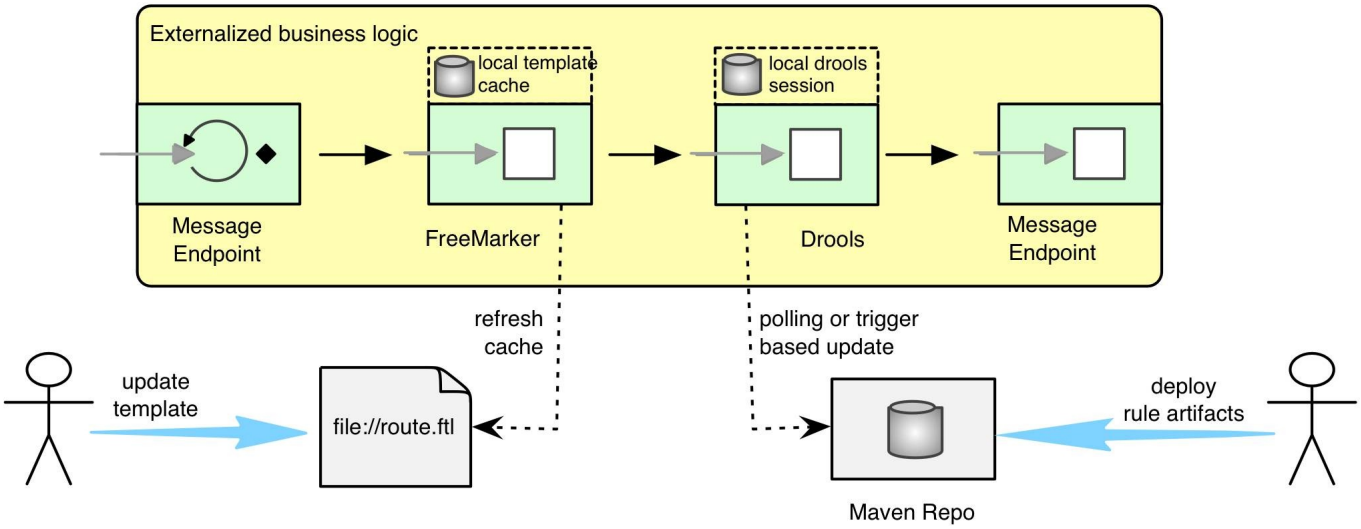
dynamic routing data, Message Translation and Content Enrichment from external data stores, etc.

Business Logic Variability

In the data-driven routing section we have seen how changing the data can alter the behaviour of a processing step in a flow. In some circumstances you need even more dynamism when the data change is not enough, but also the data model and the algorithm using the model has to change at runtime. This can be achieved through specialized Camel components:

- **Camel and Drools:** JBoss Drools is a business rules management system and reasoning engine based on JSR-94. The Drools engine combined with Camel is pretty awesome and offers great runtime logic dynamism. As illustrated in the diagram below, the Drools engine can be instantiated in a processing step as part of a Camel route and reason, and manipulate Camel exchanges. The rule engine uses rules which are usually part of a separate deployment unit, not embedded in the Camel application. The newer versions of Drools have simplified managing the rules, and the domain models used in those rules, by relying on Apache Maven primarily. So rules and the corresponding Java models are packed as Maven artifacts and loaded at runtime by the rule engine to reason over incoming data. In our hypothetical example, there is a Drools-based processor which creates a Drools session using a specific version of the validation rules. Then every incoming Camel exchange is inserted into the session and validated. Now let's imagine that the business has decided to update the validation rules and has released a new version of these to a Maven repository (such as Nexus). When this happens, the Drools engine running in the Camel route can detect that there is a new version of the rules; it will download it, update the runtime session, and evaluate new incoming exchanges using updated rules. There is no need to touch the Camel JVM at all; all of the work is done by the Drools engine. The Drools engine has an agent called `KieScanner` (previously known as `KnowledgeAgent`) that can continuously monitor Maven repositories and check for new releases of specific Maven artifacts. The agent can be configured to run with a fixed time interval (polling mode) or it can be asked on demand (the push model) to check for new Maven artifacts. Whenever `KieScanner` finds a newer version, it downloads the artifact and updates the runtime model (`KieContainer`) to ensure newer sessions are based on the updated rules. Usually the decision of whether you choose the polling or the push model is primarily driven by change control and release management processes in place. But there are also a few technical considerations to keep in mind:
    - *Poll:* if polling is used, as soon as the rule update is released to the Maven repository, the changes will be automatically detected and applied to the running application. That behaviour may or may not be desirable. As with any polling mechanism, if the polling frequency is too high, it will incur an additional load on the system, but it will detect changes quicker and apply them faster. If the polling frequency is too low, there might be too much lag between a change and its application to the runtime, leading to undesirable side effects.

- *Push:* with this model, the application has to be explicitly triggered to check for updates. This approach is more scalable and is closer to real time, but it requires an additional communication channel for control purposes back to the Drools engine to trigger the scan.
  Covering the basics of Drools is outside the scope of this book, and hopefully the short example was enough to demonstrate the basic principles. When the business logic is complex and highly varying at runtime, Drools is a natural fit and worth checking out.



Business logic variability at runtime

- **Templating/scripting engines:** Camel supports a number of scripting languages (such as Groovy, Ruby, and BeanShell) as part of JSR 223 in Java 6. So some steps of the routing flow can be written in a scripting language different to the regular Camel DSL and Java beans. The scripting resources can be located at an external location (classpath, file, or HTTP) and altered without changing the Camel route itself. Unfortunately at the time of writing, there is not an easy way to reload the scripting resource at runtime without restarting the Camel route itself. Hopefully that will change in future versions of Camel and scripts can be updated to alter the runtime behaviour of a route without restarting the route.
  In this category I've also included the template components such as FreeMarker, Velocity, XQuery, XSLT, and StringTemplate, which can also alter the content of an exchange by evaluating the template. These engines can also access external resources to load the templates (similarly to the scripting engines), but an added benefit is that these template components will reload the template regularly. Depending on the configuration, they may read the template resource for every message (which will have a big performance impact) or they can cache it for a certain time. In both cases, it allows the updating of the template file externally and the changing of the processing algorithm in the template at runtime without touching the Camel application. This flexibility can be very handy in certain situations where you have to do changes to a running Camel route. But act very cautiously as untested changes may cause errors at runtime or may even not compile.

More Information

[MSDN RRP] Runtime Reconfiguration Pattern in MSDN
[JOLOKIA] Jolokia
[CAMEL DYNAMIC ROUTER] Dynamic Router EIP - Apache Camel

7. External Configuration Pattern

This is also known as the External Configuration Store *[MSDN ECSP]*.

Intent

To parametrize the configuration information of an application and externalize it from the deployment archive.

Context and Problem

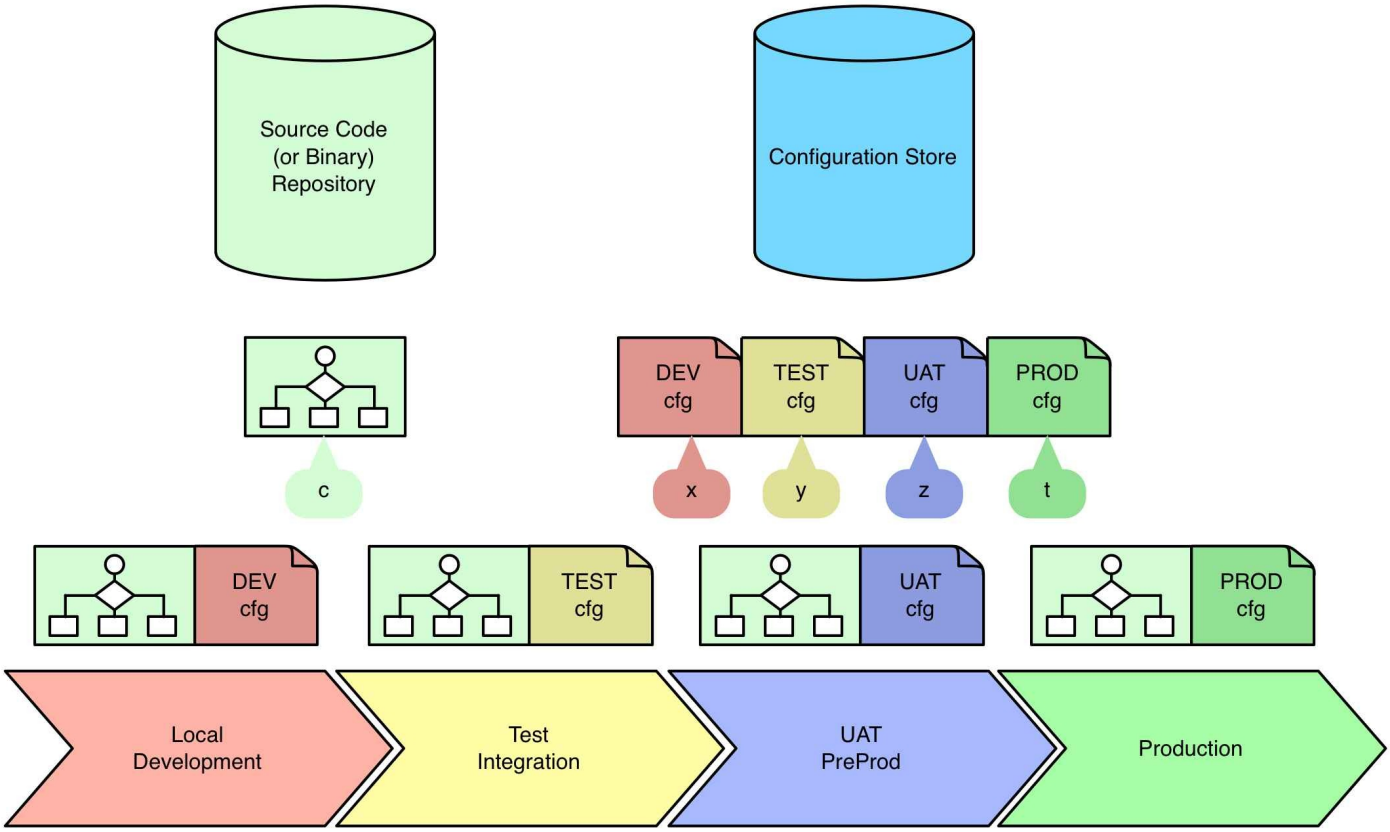
A very simplified view of an integration application is that it moves data from source to target. And while doing so, the application splits messages, aggregates them, does data transformation, handles errors, and may perform a number of retries. While developing and deploying this application from development up to production environments, the endpoint details differ, and other configuration options and tuning details need to vary too. During the lifetime of the application, the team gets more insight into the behaviour of the application and further tuning is needed. Configuring the same binaries to run in different environments, and tuning them during production maintenance windows without changing the actual binaries, requires externalizing multiple configuration options, even those that might not seem likely to change.

Forces and Solution

By their very nature integration applications deal with a multitude of systems, some of which might be outside of our control. To adapt to the changing environment, it is important to identify possible points of change in the application and externalize them. There are two main categories of configuration option that are good to have externalized:

- **Environment properties:** these are well-known environment-specific properties such as IP addresses, port numbers, file paths, various credentials, etc. Usually identification of the properties happens earlier in the software development life cycle - when the same code base has to run in an environment different than the developer machine. So this category is less problematic to identify.
- **Tuning properties:** these are harder to identify at early stages of a software life cycle as the software works fine with the existing default values until the load on the application reaches a certain limit. The need for such tuning usually arises during a performance test cycle (load and stress tests mainly), or when the application reaches the production environment and runs for a while. Another common reason for tuning the application is a change in the deployment topology. An application may run fine in a development and test environment with a small number of nodes, but behave unsatisfactorily when deployed in a production environment where there are many more nodes. The common theme for these properties is that the application would work fine with default values during the early stages of development and prevent the

identification of these configuration spots. Identifying these will help to tune the application at later stages without going through code change and the whole release cycle process.



Application binaries and configurations separated

Once all the configuration values for an application have been identified, the next step is managing these in an external store with appropriate tools and procedures. Fortunately for me, configuration management is a discipline on its own and outside the scope of this book. But it is classified as a "known problem area" as it requires the understanding and managing of properties by different teams. Rather than creating scripts and tools to further isolate the Dev and Ops teams, adopting a DevOps culture, where one team owns the development and deployment of the application to all environments, seems to show better results.

Mechanics

The Camel Properties component allows us to use property placeholders when defining endpoint URIs. It provides a nice abstraction and a unified access to properties from various locations (JVM system properties, environments variables, file, or classpath) regardless of the bootstrap mechanism (standalone Java process, Spring application, Apache Karaf and OSGI, or Docker).
Camel URIs are a big part of the routing code base. The Camel DSL defines the skeleton of the routing flow, but the URIs, with all the options, define fully the endpoint behaviour. If the business requirements do not change, there is no reason to change the routing flow (described though the DSL), but the endpoint URIs will change from environment to environment, and some options may be added or removed through the lifetime of the application. That's why it is helpful to externalize all URIs fully, rather than selecting URIs and choosing options from them. The only exception to this rule would be when a `CamelContext` is broken down into smaller routes and these routes are connected through the Direct component. Since the way in which the routes are connected will not change (it is usually an implementation detail and up to the developer to break down and connect

routes). Externalizing individual options will allow you only to change these options, but having the full URI externalized will allow you to add more options and further tuning (overriding default values) on newer environments or when the external constraints change. If you still want to expose individual options separately so that teams unfamiliar with the Camel URI syntax can change certain values, it is possible to have properties embedded in other properties (such as `<to uri="{{destination}}"/>` and then have `destination=direct?keyOne={{valueOne}}`). This technique will allow you to change one value only (`{{valueOne}}`) in a controlled manner, but still having the ability to modify the full URI if needed (`{{destination}}`).
One downside of externalizing full URIs is that the Camel routes become less readable and harder to understand, as the DSL will not give you any clue about the actual used endpoints. But the ability of reconfiguring and changing the behaviour of the integration flow only through configurations (in environments like Karaf even without restarting the JVM) is quite cool.

Here are few candidates for externalizing in a Camel application:

- All Camel URIs, except those that reference other routes or beans in the same `CamelContext` or module.
- Numbers and enumerated values that are hard-coded through the Camel DSL.
- Error Handler- and Exception Clause-related configurations, such as redelivery behaviour, delays, etc.
- Camel does allow the use of beans to configure certain aspects of the routing process. Route policies, various strategies, and profiles can also use the same property replacement mechanism (but rather using the `${key}` notion, instead of `{{key}}`).
- `CxfEndpoint`-related configurations, such as ConnectionTimeout, and ReceiveTimeout.
- ActiveMQ related connection configurations, and everything related to ActiveMQConnectionFactory.

More Information

[MSDN ECSP] *External Configuration Store Pattern in MSDN*

II Error Handling Patterns

The patterns under this category are focused around error handling, error prevention, and recovery from errors. These are the patterns and principles used for designing the integration flow for the unhappy paths.

8. Data Integrity Pattern

This is also known as the Transactional Service Pattern, and the Atomic Service Transactions.

Intent

To maintain the data consistency and business integrity of a system comprised of dispersed data sources in the case of a processing failure.

Context and Problem

Integration applications typically process data from disparate data sources such as a relational database, key value store, message queue, or file system. Maintaining data integrity in an environment where each data source has a different transaction model can be challenging. If a service has to mutate two or more data sources, there is a chance that these data sources will get out of sync in the case of a failure of the mutating service.

Forces and Solution

Depending on the data consistency requirements, there are different approaches for dealing with failure scenarios:

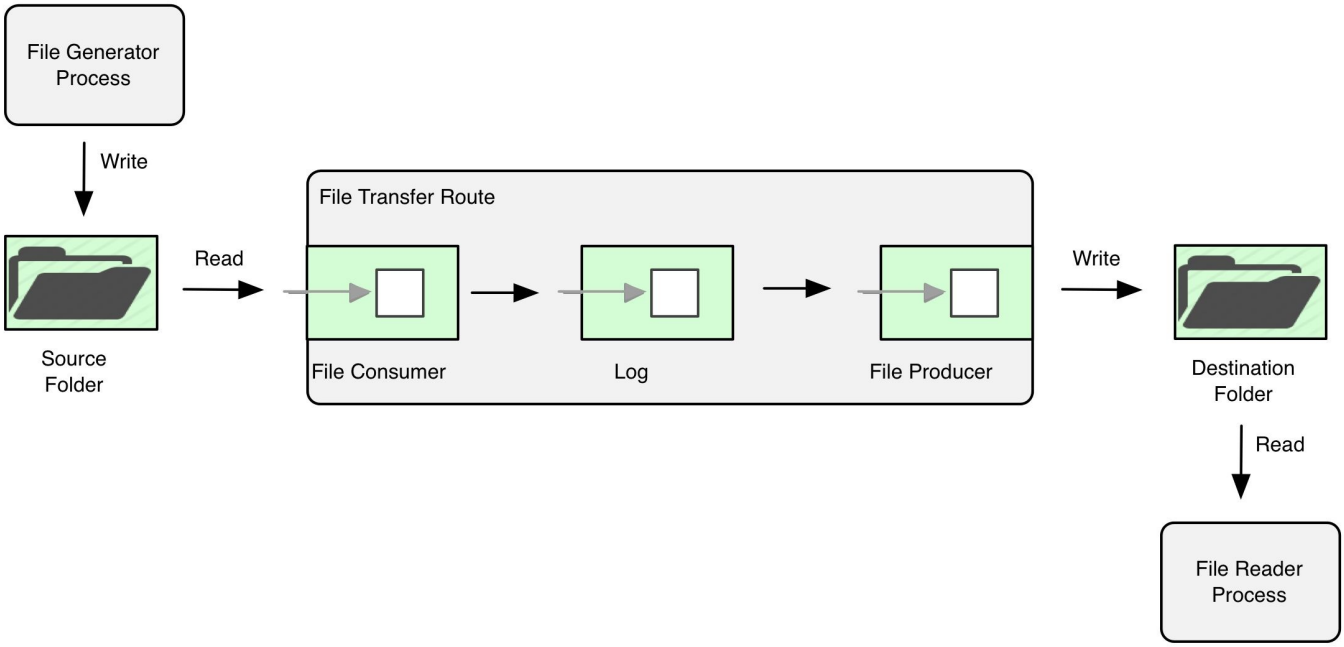
- Some systems **do not offer transactional behaviour** at all and the data consistency has to be ensured through additional mechanisms such as data locking before modifying, the committing of the changes when the operation completes successfully, or the undoing modifications when when there are failures.
- In a **strongly consistent system**, all changes are atomic (indivisible), consistent, isolated, and durable (ACID). This model relies on using locks within a single data source and as such impacts on availability and scalability of the system. If a service has to modify multiple data sources, it has to be wrapped in a global transaction, and then this transaction will ensure that either all changes are successful, or that no changes are carried out.
- In an **eventually consistent model**, the data sources are partitioned across processes and it is not possible to use locking to ensure consistency. If a service has to modify multiple data sources, it requires additional coordination logic to guarantee the overall system integrity.

In this chapter we will look at a non-transactional example and a few strongly consistent use cases. The eventually consistent model will be covered as part of the Saga Pattern.

Mechanics

No Transactions

The main goal of the Data Integrity Pattern is to ensure that the data that is managed by a service is mutated and persisted in accordance with ACID principles. But integration applications do support a variety of protocols and not all of them support transactions. In these situations, whichever means are available for mutating data in an ACID compliant way, these are used. Let’s have a look at a simple and still popular way of exchanging information, that is the file transfer, and the potential issues with it.



Non-transactional system

In this example we have an external File Write Process that writes files to a folder (this process can be a Camel route or anything else). Our Camel-based route copies files from the source to the destination folder and also does some logging. There is also another external process that reads the copied files from the destination folder and does further processing. In this simple looking-at-a-first-sight example there are a couple of potentially problematic areas:

- The first one is how will the Camel route know that the File Generator Process has completed writing the files, and that these files can be read and processed? You do not want to read half an XML file, or CSV file, when some lines are missing. One of the simplest techniques that comes to mind to solve this is to check for exclusive read-lock on the file. But not all file systems support read-locks; for example, some mounted/shared file systems or FTPs. Luckily Camel do provide various `readLock` strategies *[CAMEL FILE]* and all you have to do is choose the right one for your case. You can use marker files, check whether the file size is changing over a time period, or try to rename the file to make sure there is no other lock on it. All of these mechanisms are there to ensure that the file can be read and that its content will not be modified by other processes at the same time. Choose wisely.
- The second important decision is concerned with how to make sure that the files we are writing to are not read by a File Reader Process before we finish writing to these files. The File Reader Process might not have all the fancy mechanisms that Camel has to detect when a file is ready for consumption (whether that is reading only or involving the moving/deleting of the file too does not matter). This is a no-brainer; the most reliable way is to use a different temporary file name while writing and once the writing is finished, to rename the file its expected final name. Since moving is an atomic (remember ACID) operation, the file cannot be read while moving - only after it is fully moved. But be careful with this option too as some file systems may give you `create` but not `move` permissions. Another quite common mechanism used for writing files is to create a `done` marker file once the writing of the original file is

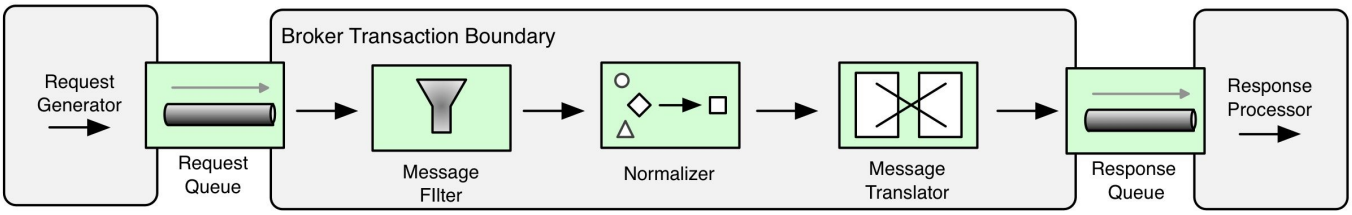
complete. This marker file can be used as an indicator for the other processes that the original file is ready for consumption. This option assumes that the other processes can work with `done` files, though. Camel consumer can work with `done` files in addition to all the `readLock` strategies just fine. So if you are writing and reading files from Camel-based applications, `done` file markers can be used too without any additional effort.

- The third potentially problematic area is to do with streaming. If the files are large, instead of reading the whole file into application memory and then writing it to the target location, we can enable streaming which will allow us to stream the file content from source to destination byte by byte. We could even split the file using a splitter in a streaming mode if we need to. But in streaming mode, it is crucial that we do not read the stream accidentally during processing as streams can be consumed only once, and if the stream is consumed during processing, an empty file will be created in the destination folder as the content is no longer available. We also mentioned that our Camel route will do logging. Imagine what can happen if we start logging the message body to debug a problem. In certain environments with detailed logging enabled, we might get empty files in the destination folder and log files might get filled up with the transferred file content (and in some other environments, files copied properly). Camel has thought about this one too. In order to read streams multiple times, you can enable stream caching, which will read the stream and store it in a structure that allows multiple reads. By default, for smaller streams (less than 128kb), Camel will store them in-memory which defeats the purpose of using streams in the first place, but for larger streams, Camel will use the `temp` directory as a spool location; so there is still a value in using streams.

File component is only one example of a non-transactional system interaction. There are many other use cases in this category, and for dealing with these, Camel offers some synchronization mechanisms. Camel internally uses the concept of `UnitOfWork` to represent something like a transactional boundary for a group of tasks. In short, every exchange has a `UnitOfWork` which allows you to register callbacks implementing the `Synchronization` interface. When the exchange has completed routing, all the callbacks are called. The callback (`Synchronization` interface) has `onComplete` and `onFailure` methods which allow you to implement a kind of commit or rollback behaviour. This mechanism allows the registering of callbacks to the exchange with custom logic (to manage the transactional behaviour manually) that will be executed when the exchange is completed. A higher level abstraction of this behaviour is the so-called onCompletion *[CAMEL ONCOMPLETION]*. This construct lets you register not Java beans implementing the `Synchronization` interface, but Camel routes as callbacks. So whenever the exchange has completed routing, another route can be triggered to perform, commit, or roll back different kinds of tasks. Under the hood, `onCompletion` uses `Synchronization` to register itself, but it also offers additional features such as asynchronous behaviour through a different thread pool, predicates, or the ability to execute before or after the consumer returns the result (just before the exchange is complete). By the way, this is the same mechanism used by File component to delete or move consumed files when the exchange completes successfully, or to move the file to a failed folder when the exchange has failed. So it is a quite reliable callback mechanism.

Local Transactions

Many data sources such as message queues, relational databases, and key-value stores support transactions. Let’s have a look at an example where we have two queues that belong to the same message broker and we interact with the broker though a transaction manager.

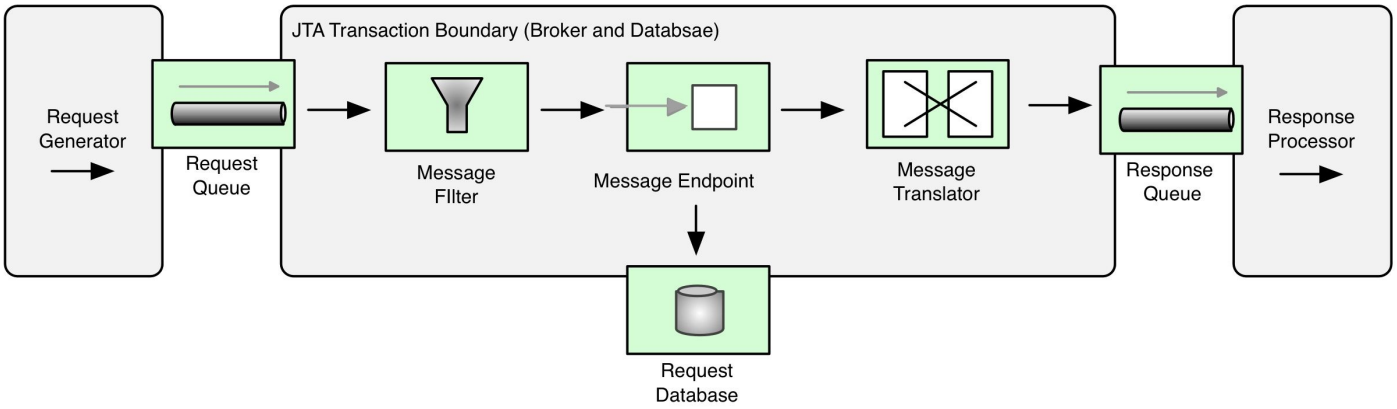


A service with local transactions

We can see that there is a request generator process that writes messages to the request queue and this is where the transaction boundary of that process reaches up to. Then a Camel route reads the messages and performs a number of transformations. During this time, the message is in flight in the message broker so it is not visible to other message consumers, but the message is not removed from the queue either as the transaction has not committed yet. If the Camel application crashes at this stage, the transaction will time out and the message will become available for reading to other consumers rather than being lost. When the message is written to the response queue, then the transaction is committed and the messages will be removed/consumed from the request queue. Another possibility is that during the processing an exception may be thrown; this will roll back the transaction and the broker will move the message from the request queue to a DLQ. Before assuming a message for a poison pill, by default the broker will perform a number of redeliveries. Only when these redeliveries are exhausted, the message will be moved the a DLQ. The important point here is that every request message will end up either in the response queue or in the DLQ, and no messages will disappear in the case of catastrophic events during processing. This is an example of a Transactional Service Pattern *[SOA PATTERNS]*, and the behaviour is ensured by the transaction manager and its ACID capabilities. This is also a so-called local transaction as there is only one transaction manager involved (Spring `JmsTransactionManager` in our example) and only one transactional resource (the ActiveMQ broker). Transaction managers use the thread context to keep transaction-related information, so when processing messages with Camel, make sure the exchange is using one thread only throughout the routing path. If Threads DSL is used, it will pass the exchange to a different thread; if SEDA is used for connecting routes (rather than Direct) it will use different threads for different routes. Any of these (or some other parallel behaviour) will leak the exchange outside of the transaction boundary and lead to unexpected behaviour.

Global Transactions

A more complicated scenario is when there are different data sources with different transaction managers. In this situation we need a global transaction manager that can span multiple resources and coordinate the transactions across these resources.



A service with global transactions

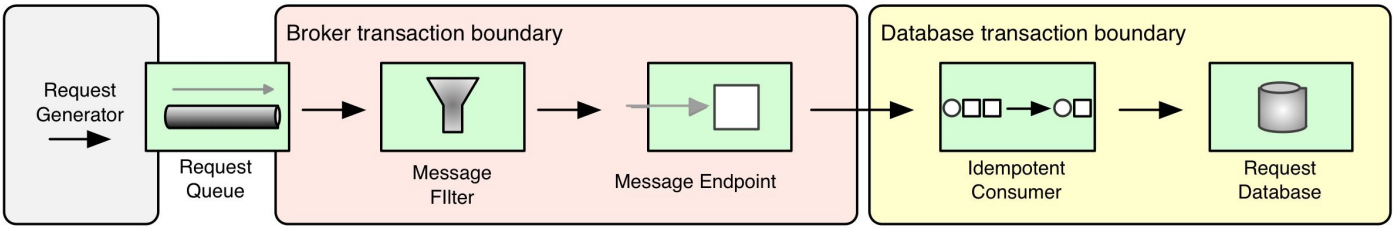
In this example, a Camel route receives messages from a request queue and writes to a relational database first. After this, it writes the result (let's assume that it is an ID generated by the database) to the result queue. The reason we need a global transaction here is because there is a chance that after the data has been persisted to the database, the processing will fail, for example, during message translation or during writing to the result queue. In this situation, the transaction from the broker will roll back and the message will be moved to a DLQ, but the database would still have the inserted value. A global transaction can roll back the changes in the database and also roll back the transaction in the broker, and ensure that both resources are in a consistent state (either both transactions are committed or both are rolled back).
Keep in mind that not all resources support global transactions and having these is very costly in terms of speed as the transaction manager has to coordinate the transaction with two or more processes. The global transactions use the XA protocol for resource coordination (the resources have to be XA compliant in order to participate in such a transaction), which in the Java world is represented by Java Transaction API (JTA). Containers such as Apache Karaf and JBoss WildFly do provide JTA-compliant transaction managers (Apache Aries and Narayana respectively), but if you want to use a different runtime, you may have to use a different transaction manager (such as Atomikos).

Distributed transactions are also available for Web services. Through WS-Atomic Transaction standard, it is possible to call two or more separate Web services and have them participate in a transaction. If a web service call fails, the WS-Atomic implementation will make sure to roll back the previous calls and ensure all service calls are in a consistent state. When a transaction boundary spans across multiple services/processes, another strategy is to use the Saga Pattern (i.e. compensating transactions). Saga does not require a two phase commit and it is a more scalable solution.

Avoiding Global Transactions through Idempotency

Here is a bonus section. In certain situations, there is a way to avoid global transactions even if there are two resources involved. The following Camel route reads messages off the queue, performs some filtering, does a database insert and transformation, and then puts the result into a response queue. If the database operation is idempotent by nature, there is no need for an additional idempotent filter. But for our example I have added an idempotent filter element to emphasize that the database operation must be idempotent.

The idempotent filter and the database operation use the same database and as such they both can be part of the same local transaction. Notice also that there is no global transaction that spans both resources (the broker and the database), but two independent local transactions.



Two isolated local transactions

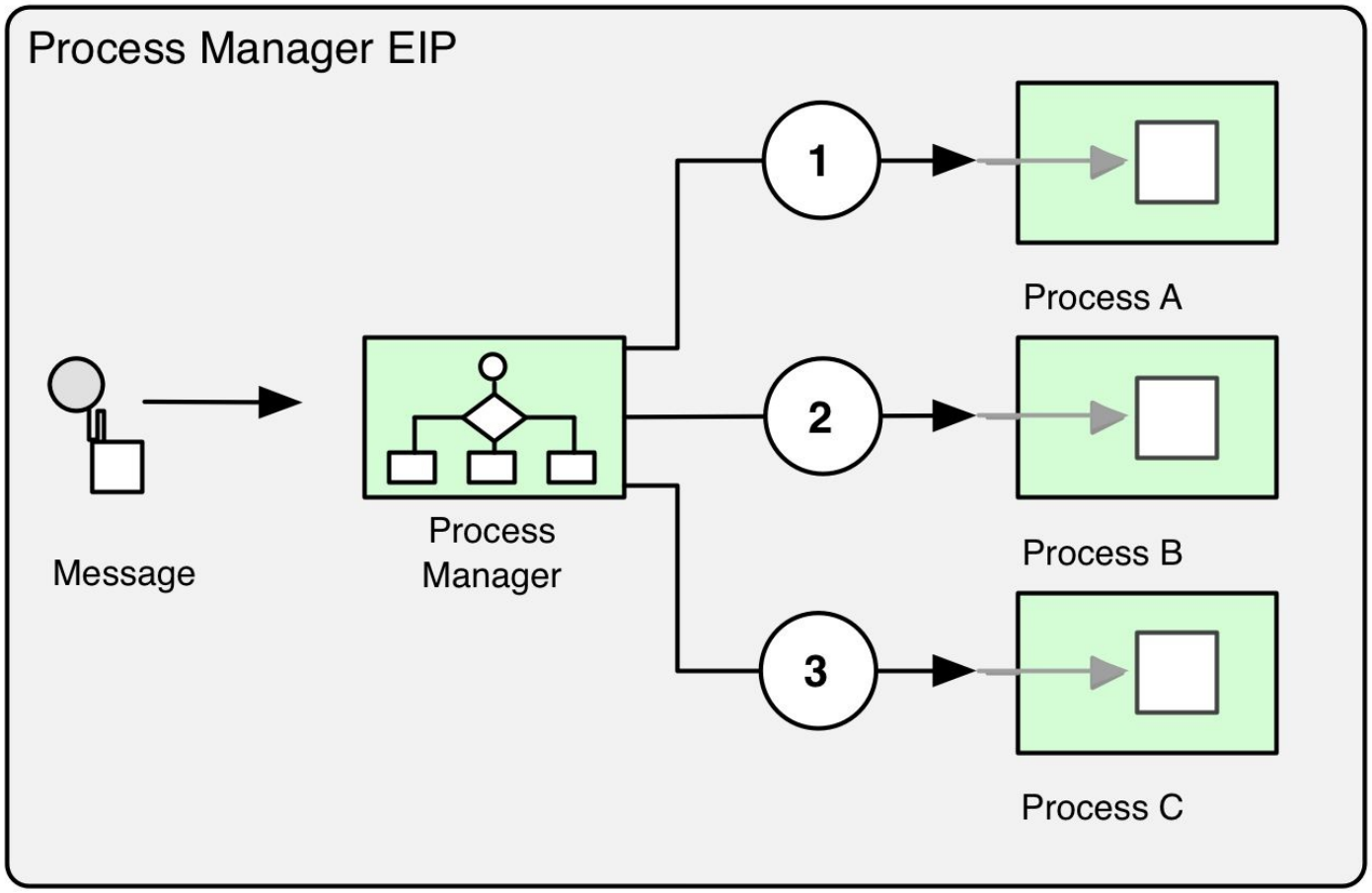
With this layout of the message flow, starting from a message queue, going into an idempotent database operation, and no further state modifications as part of the processing, it is possible to avoid using distributed transactions. Let's see the possible scenarios.
If there is an exception before writing to the database, the broker transaction will be rolled back and the message will be moved to a DLQ. Both the queue and the database will remain in a clean state.
If there is an exception during the database insert, the database will roll back the transaction (the idempotent filter is part of the same transaction too), and then the broker will roll back its transaction and both resources will be in a clean state again.
But why do we need the database operation to be idempotent? That is, for the case when the database transaction is committed, but the broker transaction is not. Perhaps the Camel process was killed straight after the database commit, but before the broker commit could happen. In this unlikely scenario, the message will remain in the message broker. But when the Camel route starts again, the message will be consumed, and since our database operation is idempotent, even if we process the same message again, there will not be any side effects on the database, so we are all good and consistent.

More Information

[CAMEL FILE] File Component - Apache Camel
[CAMEL ONCOMPLETION] OnCompletion - Apache Camel
[SOA PATTERNS] Transactional Service Pattern - SOA Patterns by Arnon Rotem-Gal-Oz

9. Saga Pattern

In recent years the term Saga has been used to refer to the implementation of workflow and state machines. This is the case with the NServiceBus framework and the CQRS architecture. In the EIP world, workflow or state machines are represented by the Process Manager Pattern. The Process Manager is defined as a central processing unit that maintains the state of the sequence and determines the next processing step based on intermediate results.



Process Manager EIP

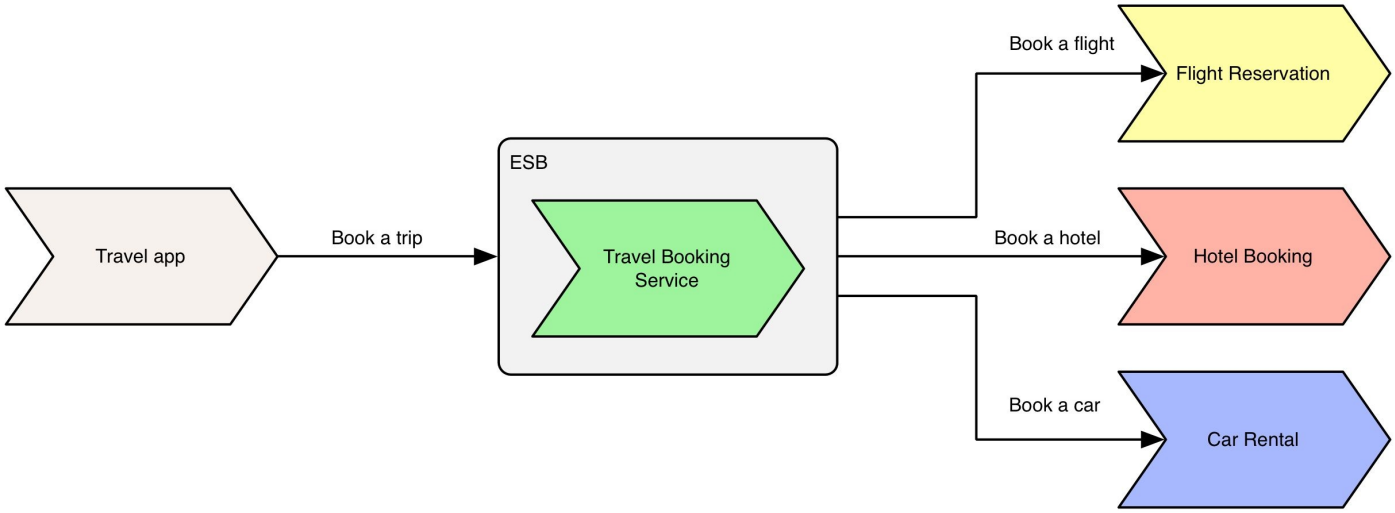
As described by the authors of the pattern, Process Manager is a catch-all term used for business process management systems such as jBPM and Activiti. In this chapter we will be looking at the Saga Pattern *[SAGAS]* as it was originally defined by Hector Garcia-Molina and Kenneth Salem as an alternative to using a distributed transaction for managing a long-running business process. We will discuss the Saga Pattern as a failure management technique only.
Other names describing the same concept as the Saga Pattern *[SOA PATTERNS] [SOC]* are the Compensating Transaction Pattern and the Long-Running Transaction Pattern *[MSDN CTP] [NARAYANA]*.

Intent

This pattern removes the need for a distributed transaction by ensuring that the transaction at each step of the business process has a defined compensating transaction. In this way, if the business process encounters an error condition and is unable to continue, it can execute the compensating transactions for the steps that have already been completed. This undoes the work completed so far in the business process and maintains the consistency of the system.

Context and Problem

To illustrate the Saga Pattern we will look at a traditional use case for distributed systems. Let's assume we have to implement a holiday booking service that can book a car, a hotel room, and a flight. The service has to book all three items in one go; partial bookings are not allowed.

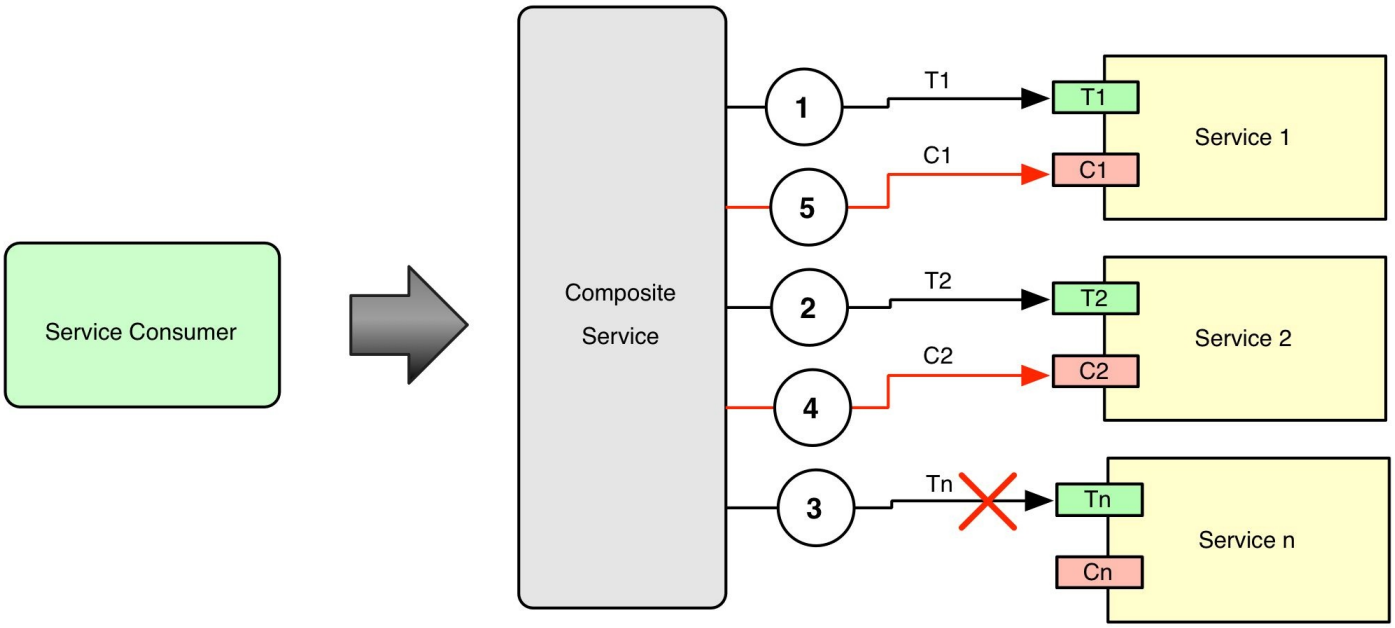


A composite service for holiday booking

Regardless of what sequence the three services are executed in (or in parallel) there is a chance that one or more of the bookings will fail due to an availability change. This is a situation that our service should not fall into as this is an inconsistent state with partial success we are trying to avoid. One way to avoid this problem would be to use the classic distributed transactions with holding locks in all three services for the whole duration of the transaction, and perform a two phase commit. WS-Atomic specification is designed for this purpose. But this is a non-scalable and non-performant solution in the modern world of REST services and cloud computing. As you may have guessed already, one solution for this problem is to use the Saga Pattern.

Forces and Solution

The Saga Pattern *[APPLYING SAGA] [SOA PATTERNS] [SOC]* splits a long-lived distributed transaction into individual sets of transactions whose effects can be reversed using compensating actions. With that arrangement, a Saga can complete either by committing all individual transactions, or by compensating the partially committed transaction. In either case, the system remains in a consistent state: all transactions are committed or partially committed with a corrective action to compensate.



Compensating actions in use after a failed operation

To implement our holiday booking service using the Saga Pattern, each of the booking services (car booking, room booking, and flight booking) has to provide an operation to reverse the respective booking action. Notice that we are deliberately not using the terms "rollback or "delete" as these actions are not always possible. Instead we use the term "compensate" to describe an action that semantically undoes the committed transaction. In the happy path scenario, when all three bookings succeed, Saga completes successfully. But let's say the car and room booking succeeded, but the flight booking failed. Then Saga will execute corrective actions, cancelling the car and room booking and returning failure indicating no bookings have been made.
As this is not exactly a transactional system, it is possible for the compensating actions to fail. That's why it is important to have them as idempotent operations in order to retry the failing compensations. It is also possible to organize the operations in a way that minimizes the need for compensating actions; for example, by running the operation most likely to fail first, or performing the action hardest to revert last. But that is still not a guarantee for a successful completion. While an ACID compliant system is always in a consistent state, a system relying on the Saga Pattern will be in an inconsistent state during the time window during which a Saga starts and completes (or even for longer if a compensation fails permanently).

Mechanics

Form a Camel application point of view, there is no component or framework feature specific to the Saga Pattern. The pattern has to be implemented with error handling and conditional routing, and compensating logic for every sub-action.
In a Java application, errors are dealt with by `try-catch-finally` blocks. Camel also offers a similar experience through `doTry-doCatch-doFinally` constructs, but its true power is with error handlers and exception policies. Error handlers can be applied to the whole `CamelContext` or reused and applied to individual routes. They offer a declarative approach for dealing with errors without leaking the error processing logic into the happy path flow. A typical implementation for the Saga Pattern could be the following. Each sub-action has its dedicated route, with an error handler specifically configured (with

redelivery, delays, etc.) for the external endpoint. If the external operation succeeds, the route will populate the exchange with a property that stores the compensating action endpoint.
Then a main route will orchestrate the sequence of calling each action by calling the individual routes synchronously (through the Direct component). When a route with an operation is completed, the orchestrating route calls the next route/operation in the sequence. If an operation fails, the error handler in that route will kick in and handle the exception. Then the orchestrator detects that the last operation has failed, and deviates from the happy path flow and starts the rollback procedure. The rollback procedure can be implemented with recipientList or any other mechanism that can invoke the compensating operations recorded in the exchange by each successfully completed action so far. Each compensating action can also be implemented as a separate route with an error handler. If a compensating action fails permanently (even after retries), before returning the failure from the orchestrator service it is important to persist the compensation needed to a queue so that it can be compensated later.

Implementing this pattern requires a solid understanding of Camel Error Handler and Exception clauses, as the business integrity of the service will be primarily guaranteed by the unhappy path implementation of the processing flows.

More Information

[SAGAS] *Sagas by Hector Garcaa-Molrna and Kenneth Salem*
[MSDN CTP] *Compensating Transaction Pattern in MSDN*
[NARAYANA] *Compensating Transactions by JBoss Narayana*
[APPLYING SAGA] *Applying the Saga Pattern by Caitie McCaffrey*
[SOA PATTERNS] *Saga Pattern - SOA Patterns by Arnon Rotem-Gal-Oz*
[SOC] *Sagas - Service-Oriented Computing by Munindar P. Singh, Michael N. Huhns*

10. Idempotent Filter Pattern

This is also known as the Idempotent Receiver EIP *[EIP]*.

Intent

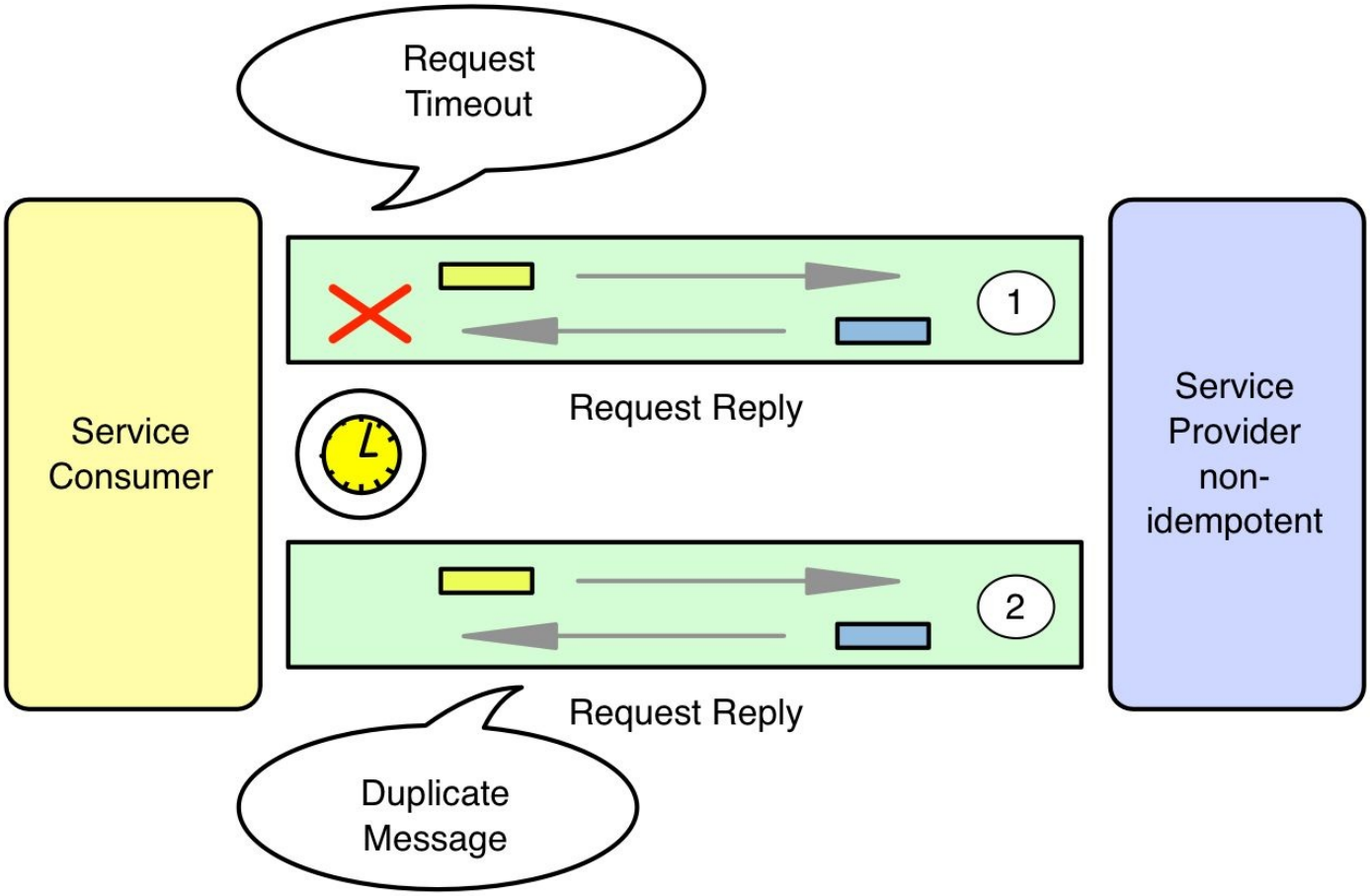
This filters out duplicate messages and ensures only unique messages are passed through.

Context and Problem

Idempotency *[SOA PRACTICE]* is the property of certain operations that allows them to process the same request multiple times without causing any side effects. Some operations are idempotent by nature and it is safe to execute them multiple times without any side effects. Read-only operations (retrieving a value or executing a complex query) do not change the system state and are idempotent. There are other operations that do change the system state but they are also idempotent. For example, delete or update/set operations that do not depend on the original state of the system but set a new value with each request are idempotent too; deleting a bank account or updating its status with the same value multiple times will have the same effect as doing it once.
But there are other operations which are not safe to be invoked multiple times. These are operations that create new values with each invocation and ones that depend on the current state of the system. For example, an operation that creates a new bank account or one that adds an amount to an existing account cannot be safely invoked multiple times without side effects. The Idempotent Filter Pattern can turn a non-idempotent operation into an idempotent one.

Forces and Solution

Let’s imagine we have invoked a remote service and the request timed out. In such a situation, it is not clear whether the request succeeded or failed and a natural way to handle this failure would be to retry the same request.



A service consumer retrying a timed out request

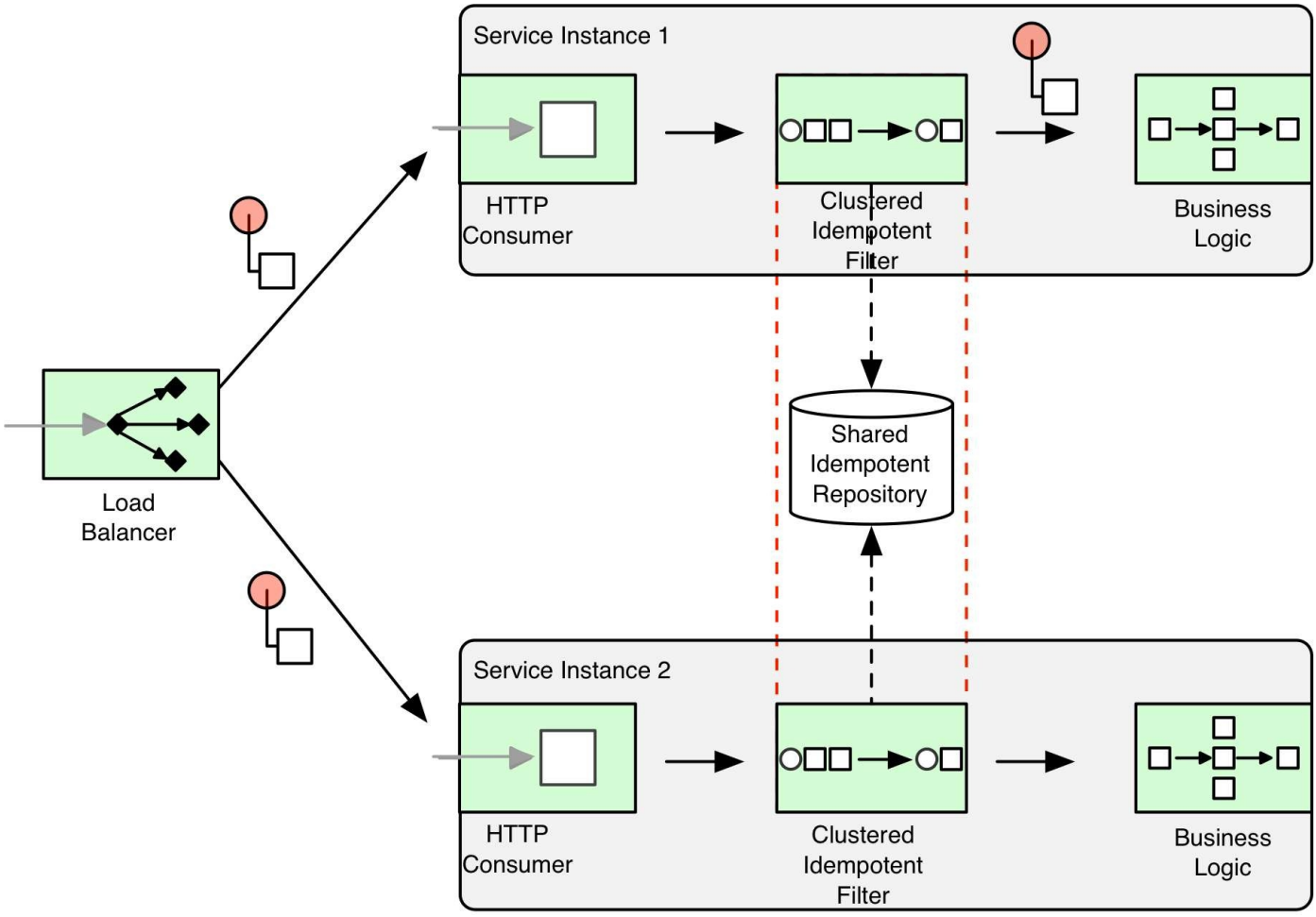
But an operation must be idempotent in order to allow retries. An idempotent filter turns a naturally non-idempotent operation into an idempotent one with some technical support. Turning an unsafe operation into an idempotent one requires additional logic to do the de-duping (the removal of duplicate messages). There are two decisions to make when turning an operation into an idempotent one:

- **Key selection:** when the service consumer sends a request, the service provider needs a way to detect duplicate requests. This is done through a unique ID sent by the service consumer though each request. The ID can be either a business key that is naturally part of the request, or a generated GUID. If the consumer decides to retry the operation, it should use the same ID for the message.
- **Repository selection:** before processing a request, the service provider checks if the given request ID is already present. If so, the request is rejected. While the key is used to identify the requests, the repository is used to remember the IDs. The type of repository will define the idempotency characteristics. The repository can be in-memory or persisted to survive service restarts; local or clustered if you want to use the Service Instance Pattern and have multiple instances of the same service to be idempotent; limited in time; for example, to have the repository store the IDs for the last ten minutes only, and ensure there will not be duplicates in a ten minute window; or limited in size; for example, to ensure uniqueness in a certain number of requests only.

Mechanics

In Camel, the Idempotent Filter Pattern is implemented as an Idempotent Consumer EIP through the DSL. Using it in the routing process involves specifying the key expression for uniquely identifying every message, configuring the EIP as to whether or not to eagerly add the message key to the repository even before the message has completed routing, whether to filter out or just flag up duplicate messages, etc. As we know from the previous section, the most important choice when using this pattern is the repository. There is a good selection of idempotent repositories available: in-memory, file-based, JPA and JDBC-based for relational databases, and popular NoSQL stores such as Cassandra, Hazelcast, Infinispan, HBase, Redis, etc. Having implemented the Infinispan and Redis idempotent repositories myself, I can assure you it is easy to add your own implementation if you need to. Each of these repositories offers a different set of capabilities and best suits for slightly different use cases. Depending on your application requirements, you have to choose the repository that best fits your needs and tune it accordingly.

Here is an example Camel flow that receives requests from an HTTP endpoint and executes some business logic. We can then apply the Service Instance Pattern and start two instances of the service (for HA and scaling purposes) with a load balancer in front of these.



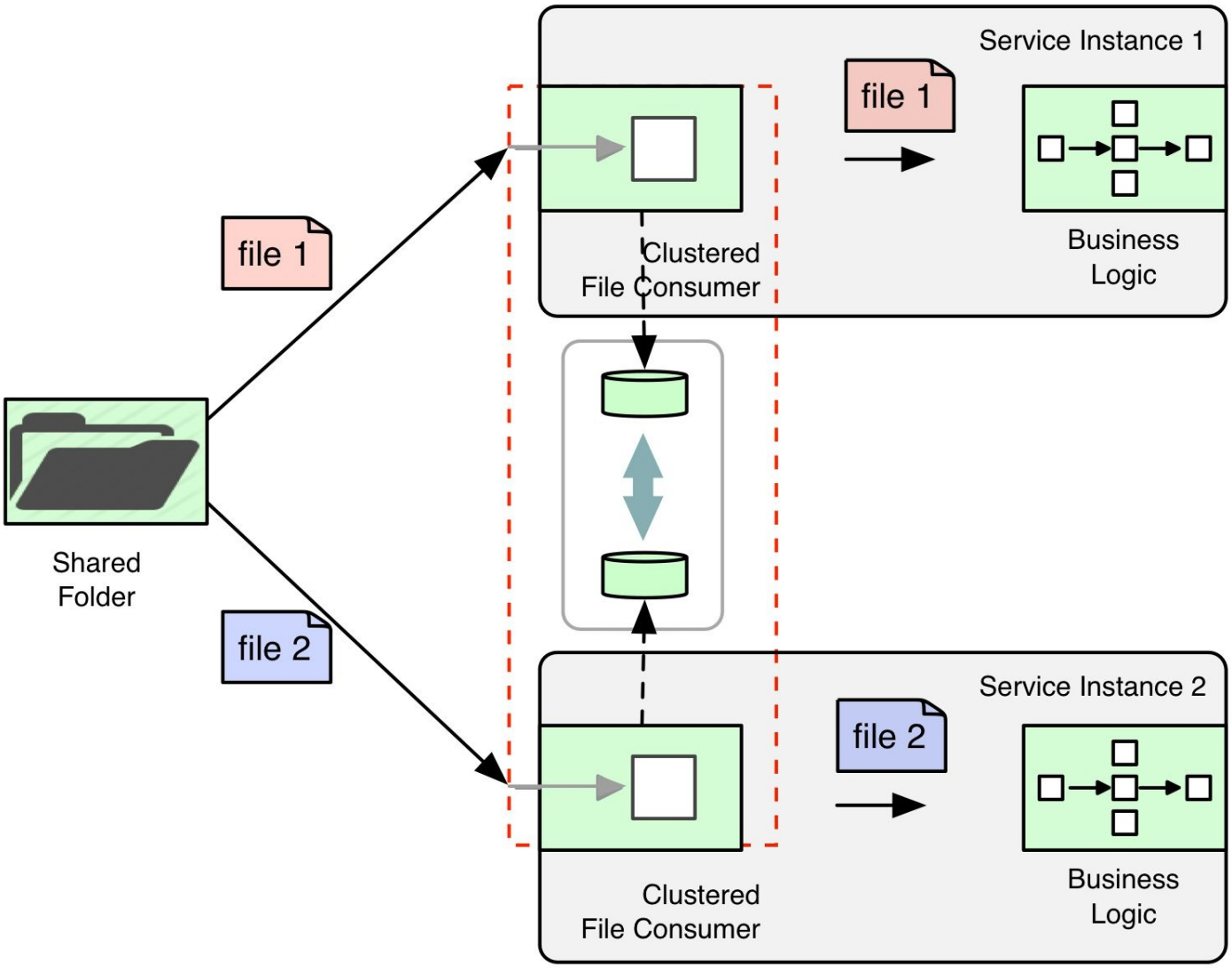
Distributed Idempotent Filter through shared data store

If we want to make the service idempotent and also support clustering, we need a repository that can be clustered or a shared repository (like a relational database) that can be used by both service instances. With this setup, even if the load balancer distributes the duplicate requests to the two separate service instances, only one request will pass the clustered idempotent filter and reach the business layer.

In addition to being a standalone EIP offered by the Camel DSL, idempotency is also used in the File component. To be more precise, as of this writing, there are three possible idempotent repositories that can be used for file consumption. Let's briefly look at each of them:

- **inProgressRepository:** this is used to keep the current in-progress files that are being consumed. By default a memory-based repository is used, but any of the other implementations can be plugged in. This repository is used internally by the consumer to skip files which are currently in progress.
- **idempotentRepository:** if the consumer is configured to be idempotent, then any processed file will be recorded to this repository and not consumed again. Again it is using an in-memory repository be default, but it can accept any of the other implementations too.
- **readLockIdempotentRepository:** in the Data Integrity chapter, we covered various readLock strategies supported by the File consumer, but intentionally skipped idempotent `readLock`. With an idempotent strategy, the File consumer tries to get an exclusive `readLock` from the idempotent repository before consuming a file. So in

this component, the idempotent repository is used only as a central locking mechanism that prevents concurrent access to the same file. This is also the recommended way for consuming files in a clustered environment where multiple Camel consumers are reading files from the same shared folder. Of course, in such a setup, it is necessary to have either a clustered or a shared repository.



A clustered file consumer using a distributed cache for a shared lock

In the diagram above, we see two instances of the same service consuming files from a shared folder. Since the idempotent behaviour is embedded in the File consumer itself, there is no need for a standalone idempotent filter as part of the processing flow. In order to function properly, the File consumer needs to use idempotent repositories which support clustering (rather than local or in-memory repositories). In our example we have chosen a clustered repository (such as Infinispan or Hazelcast) where the nodes local to each service instance synchronize the state of the repository. With such a setup, each file will be consumed only once (thanks to the clustered `idempotentRepository`) and non-concurrent access to the same file will occur (thanks to the clustered `readLockIdempotentRepository`). To improve performance by preventing choosing in-progress files, you can also use a distributed `inProgressRepository` rather that the default in-memory one.

More Information

[EIP] *Idempotent Receiver - Enterprise Integration Patterns by Gregor Hohpe and Bobby Woolf*
[SOA PRACTICE] *Idempotency - SOA in Practice by Nicolai M. Josuttis*

11. Retry Pattern

This is also known as the Transient Fault Handling Pattern.

Intent

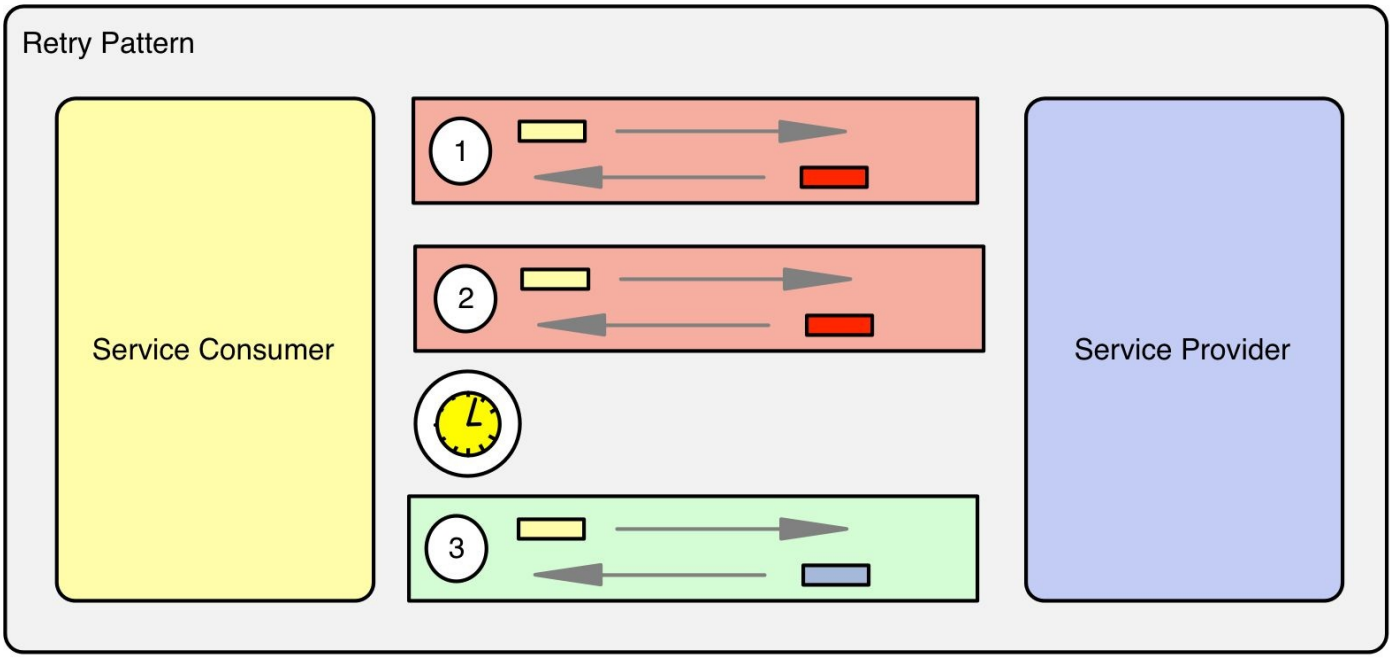
To enable applications to handle anticipated transient failures by transparently retrying *[MSDN RP]* a failed operation with an expectation that it will be successful.

Context and Problem

By their very nature integration applications have to interact with other systems over the network. With dynamic cloud-based environments becoming the norm, and the microservices architectural style partitioning applications into more granular services, the successful service communication has become a fundamental prerequisite for many distributed applications. Services that communicate with other services must be able to handle transient failures that can occur in downstream systems transparently, and continue operating without any disruption. As a transient failure can be considered an infrastructure-level fault, a loss of network connectivity, timeouts and throttling applied by busy services, etc. These conditions occur infrequently and they are typically self-correcting, and usually retrying an operation succeeds.

Forces and Solution

Reproducing and explaining transient failures can be a difficult task as these might be caused by a combination of factors happening irregularly and related to external systems. Tools such as Chaos Monkey *[CHAOS MONKEY]* can be used to simulate unpredictable system outages and let you test the application resiliency if needed. A good strategy for dealing with transient failures is to retry the operation and hope that it will succeed (if the error is truly transient, it will succeed; just keep calm and keep retrying).



A service consumer retrying the failing request

To implement a “retry” logic there are a few areas to consider:

- **Which failures to retry:** certain service operations, such as HTTP calls and relational database interactions, are potential candidates for a retry logic, but further analysis is needed before implementing it. A relational database may reject a connection attempt because it is throttling against excessive resource usage, or reject an SQL insert operation because of concurrent modification. Retrying in these situations could be successful. But if an relational database rejects a connection because of wrong credentials, or an SQL insert operation has failed because of foreign key constraints, retrying the operation will not help. Similarly with HTTP calls, retrying a connection timeout or response timeout may help, but retrying a SOAP `Fault` caused by a business error does not make any sense. So choose your retries carefully.
- **How often to retry:** once a retry necessity has been identified, the specific retry policy should be tuned to satisfy the nature of both applications: the service consumer with the retry logic and the service provider with the transient failure. For example, if a real time integration service fails to process a request, it might be allowed to do only few retry attempts with short delays before returning a response, whereas a batch-based asynchronous service may be able to afford to do more retries with longer delays and exponential back off. The retry strategy should also consider other factors such as the service consumption contracts and the SLAs of the service provider. For example, a very aggressive retry strategy may cause further throttling and even a blacklisting of a service consumer, or it can fully overload and degrade a busy service and prevent it from recovering at all. Some APIs may give you an indication of the remaining request count for a time period and blacklisting information in the response, but some may not. So a retry strategy defines how often to retry and for how long before you should accept the fact that it is a non-transient failure and give up.
- **Idempotency:** when retrying an operation, consider the possible side effects on that operation. A service operation that will be consumed with retry logic should be

designed and implemented as idempotent. Retrying the same operation with the same data input should not have any side effects. Imagine a request that has processed successfully, but the response has not reached back. The service consumer may assume that the request has failed and retry the same operation which may have some unexpected side effects.

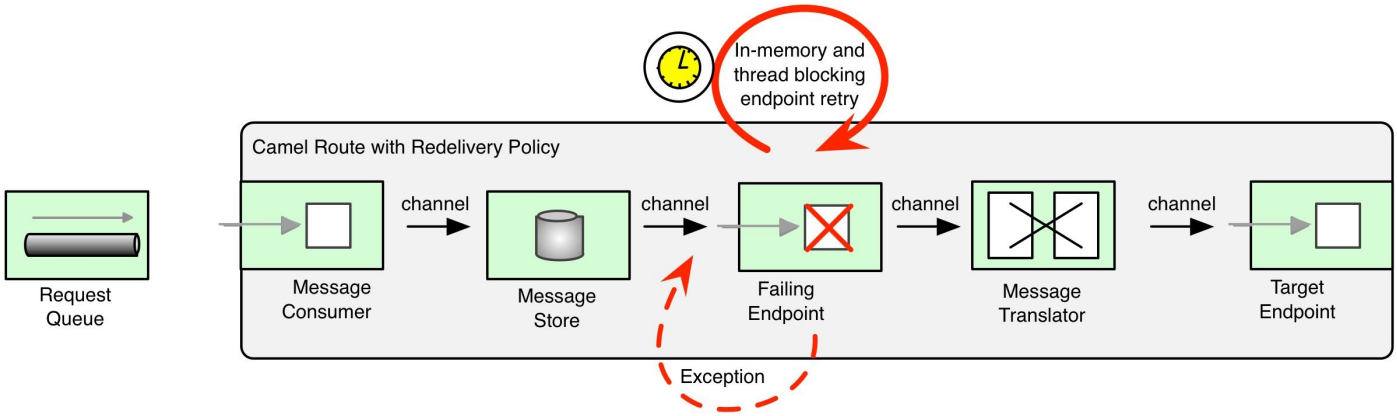
- **Monitoring:** tracking and reporting retries is important too. If certain operations are constantly retried before succeeding or they are retried too many times before failing, these have to be identified and fixed. Since retries in a service are supposed to be transparent to the service consumer, without proper monitoring in place, they may remain undetected and affect the stability and the performance of the whole system in a negative way.
- **Timeouts and SLAs:** when transient failures happen in the downstream systems and the retry logic kicks in, the overall processing time of the retrying service will increase significantly. Rather than thinking about the retry parameters from the perspective of the number of retries and delays, it is important to drive these values from the perspective of service SLAs and service consumer timeouts. So take the maximum amount of time allowed to handle the request, and determine the maximum number of retries and delays (including the processing time) that can be squeezed into that time frame.

Mechanics

There are a few different ways of performing retries with Camel and ActiveMQ.

Camel RedeliveryPolicy

This is the most popular and generic way of doing retries in a Camel. A redelivery policy *[CAMEL REDELIVERY]* defines the retry rules (such as the number of retries and delays, whether to use collision avoidance and an exponential backoff multiplier, and logging) which can then be applied to multiple `errorHandler` and `onException` blocks of the processing flow. Whenever an exception is thrown up, the rules in the redelivery policy will be applied.



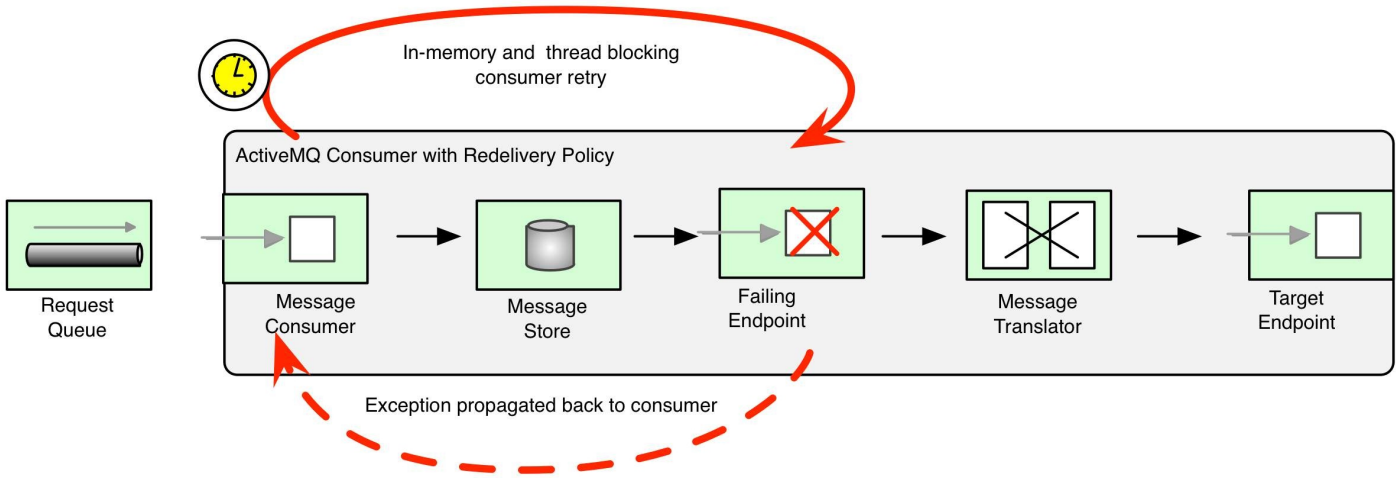
Camel RedeliveryPolicy example

The key differentiator of the retry mechanism is that Camel error handling logic will not retry the whole route, but it will retry only the failed endpoint in the processing flow. This is achieved thanks to the channels that connect the endpoints in the Camel route.

Whenever an exception is thrown up by the processing node, it is propagated back and caught by the channel, which can then apply various error handling policies. Another important difference here is that Camel-based error handling and redelivery logic is in-memory, and it blocks a thread during retries, which has consequences. You may run out of threads if all threads are blocked and waiting to do retries. The owner of the threads may be the consumer, or some parallel processing construct with a thread pool from the route (such as a parallel splitter, recipient list, or Threads DSL). If, for example, we have an HTTP consumer with ten request processing threads, a database that is busy and rejects connections, and a `RedeliveryPolicy` with exponential backoff, after ten requests all the threads will end up waiting to do retries and no thread will be available to handle new requests. A solution for this blocking of threads problem is opting for `asyncDelayedRedelivery` where Camel will use a thread pool and schedule the redelivery asynchronously. But the thread pool stores the redelivery requests in an internal queue, so this option can consume all of the heap very quickly. Also keep in mind that there is one thread pool for all error handlers and redeliveries for a `CamelContext`, so unless you configure a specific thread pool for long-lasting redelivery, the pool can be exhausted in one route and block threads in another. Another implication is that because of the in-memory nature of the retry logic, restarting the application will lose the retry state, and there will be no way of distributing or persisting this state.
Overall, this Camel retry mechanism is good for short-lived local retries, and to overcome network glitches or short locks on resources. For longer-lasting delays, it is a better option to redesign the application with persistent redeliveries that are clustered and non-thread-blocking (such a solution is described below).

ActiveMQ consumer RedeliveryPolicy

When consuming messages from ActiveMQ, the consumer can be configured with a `RedeliveryPolicy` *[ACTIVEMQ CRP]* (this is a class from ActiveMQ, not related to Camel `RedeliveryPolicy` other than sharing the same name) on `ActiveMQConnectionFactory` or `ActiveMQConnection`. This retry mechanism is offered by an ActiveMQ consumer component, rather than Camel error handling logic, to redeliver the failing (rolled back) messages. For example, if we have a Camel route consuming message from ActiveMQ, and an exception is thrown up during processing, the error will propagate back to the ActiveMQ consumer. If the consumer has a `RedeliveryPolicy`, the error will be caught and the consumer will attempt to process the message again.

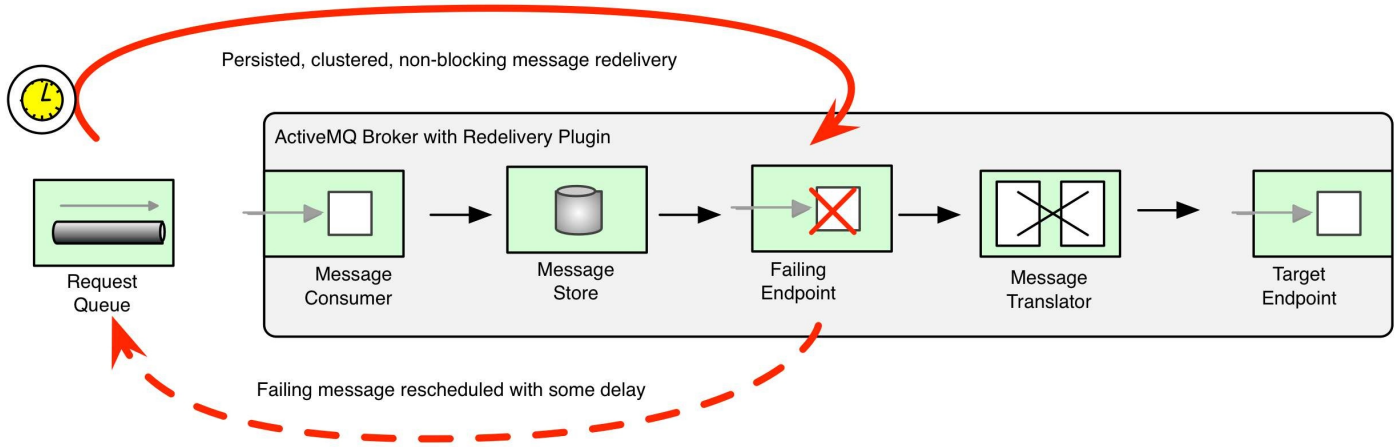


ActiveMQ consumer RedeliveryPolicy example

Again the important aspect to keep in mind is that the retry is managed by the consumer. From the broker perspective the message is delivered to the consumer and it is still in flight. The broker has no idea that the message is offered by the consumer to the processing route multiple times. This approach has similar characteristics in terms of threading and the state to Camel `RedeliveryPolicy`. It is also local to the message consumer and typically used when the local message order needs to be preserved.

ActiveMQ Broker Redelivery

This retry mechanism has different characteristics to the previous two since it is managed by the broker itself (rather than the message consumer or the Camel routing engine). ActiveMQ has the ability to deliver messages with delays thanks to its scheduler. This functionality is the base for the broker redelivery plug-in *[ACTIVEMQ BRP]*. The redelivery plug-in can intercept dead letter processing and reschedule the failing messages for redelivery. Rather than being delivered to a DLQ, a failing message is scheduled to go to the tail of the original queue and redelivered to a message consumer. This is useful when the total message order is not important and when throughput and load distribution among consumers is.



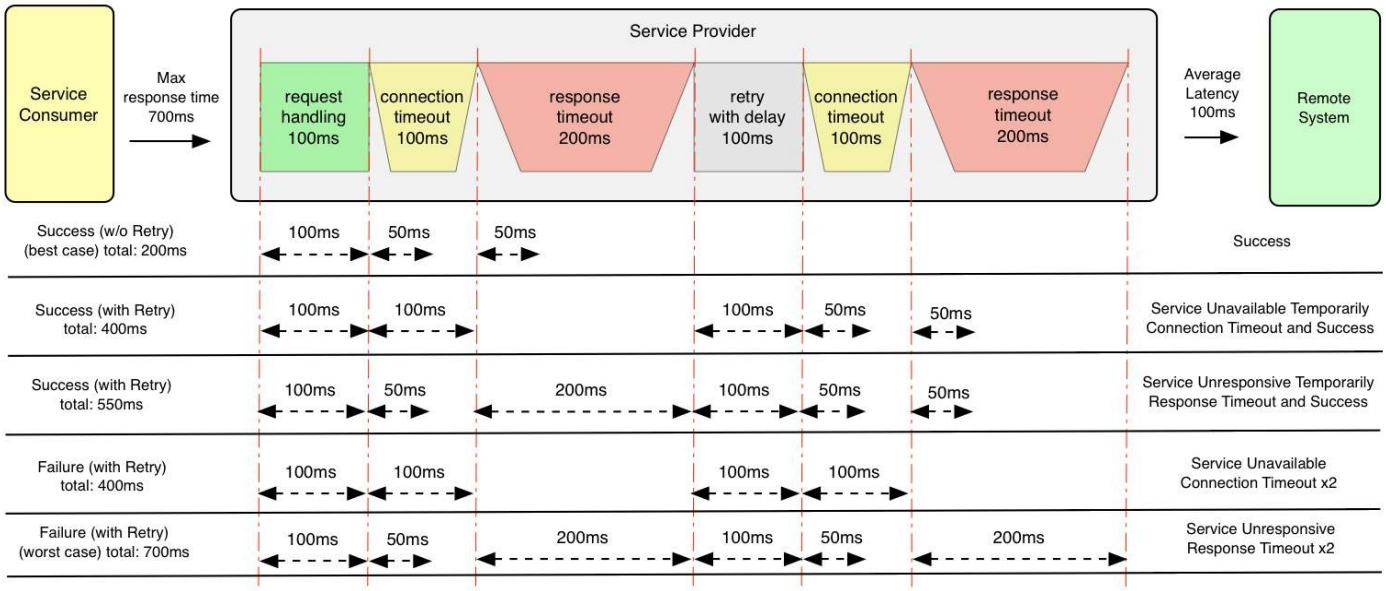
ActiveMQ broker redelivery example

The difference to the previous approaches is that the message is persistent in the broker message store and it would survive broker or Camel route restart without affecting the redelivery timings. Another advantage is that there is no thread blocked for each retried

message. Since the message is returned back to the broker, the Competing Consumers Pattern can be used to deliver the message to a different consumer. But the side effect is that the message order is lost as the message will be put at the tail of the message queue. Also, running the broker with a scheduler has some performance impact. This retry mechanism is useful for long-delayed retries where you cannot afford to have a blocked thread for every failing message. It is also useful when you want the message to be persisted and clustered for the redelivery.
Notice that it is easy to implement the broker redelivery logic manually rather than by using the broker redelivery plug-in. All you have to do is catch the exception and send the message with an `AMQ_SCHEDULED_DELAY` header to an intermediary queue. Once the delay has passed, the message will be consumed and the same operation will be retried. You can reschedule and process the same message multiple times until giving up and putting the message in a backoff or dead letter queue.

Timeouts and SLAs

One thing to consider with Retry is that it will increase the overall service response time significantly. To illustrate the problem, let's analyze this example service that performs retries:



Redelivery implications on service SLAs

We have a service provider that requires around 100ms for its request handling and another 100ms to call a remote service over HTTP. So an average successful call to our service provider takes around 200ms.
The average latency of the remote service is 100ms per SLA, so we have configured our HTTP connector options with a connection timeout of 100ms and a response timeout of 200ms. In the case of any failure, our service performs one retry with a 100ms delay. With this setup in place, we have the following possible scenarios:

1. The base case scenario is when the remote system replies as per SLA. Then our service can reply in 200ms.
2. If there is a network glitch, and our service fails to connect, we wait 100ms and retry. And if the remote service replies as per SLA, our service will reply in 400ms in total.

3. If we can reach the remote service, but it is overloaded and does not respond in time, our response timeout will be reached and the request will fail. Then our retry logic will kick in again (after 100ms) and if the remote service replies per SLA, we will produce a response in 550ms.
4. If the remote system is not accessible at all, we will get connection timeout, wait, and then retry to get a second connection timeout. Such a request will be processed in around 400ms.
5. And the worst case scenario will be when the remote service is overloaded and it allows connections but cannot produce a response in time; it will then take around 700ms for our service to respond.

As we can see, without a retry, a successful call takes around 200ms. A failure without a retry would take around 200ms or 300ms depending on whether we get connection refused or response timeout errors. But with the retry logic in place, some successful requests can take around 400ms to 550ms, and a failing request up to 700ms. So the service consumer now has to set its maximum response timeout to 700ms if it wants to always get a response back. Having timeouts on IO and thread blocking operations is always important for the stability of the system, but when redeliveries are used, timeouts become even more important as redelivery will multiply the effect of these timeouts.

More Information

[MSDN RP] *Retry Pattern in MSDN*
[CHAOS MONKEY] *Chaos Monkey by Netflix*
[CAMEL REDELIVERY] *Redelivery Policy - Apache Camel*
[ACTIVEMQ CRP] *Consumer Redelivery Policy - Apache ActiveMQ*
[ACTIVEMQ BRP] *Broker Redelivery Plugin - Apache ActiveMQ*

12. Throttling Pattern

This is also known as the Back-Pressure Mechanism (in reactive systems).

Intent

This controls the throughput of a processing flow to meet the service level agreements, and prevents the overloading of other systems.

Context and Problem

Typically, the non-functional requirements of a service are concerned with scalability and performance. An application needs to satisfy certain throughput and latency requirements, and also be able to scale further when the load increases. Less often, an application will need to handle an increasing load not by increasing capacity but by throttling it. For example, an application may have an instant load burst at certain hours of the day, certain days of the week, or perhaps at the end of the month, or the holiday season. But some downstream systems may have a fixed capacity that cannot handle such a burst in activity. Or it may be that our application is dependent on external services and there are agreed SLAs about a maximum rate of calls. In these and similar circumstances, we can use the Load Levelling Pattern or scale up our service to handle the increased incoming message load, but this will not help us to control the outgoing message rate. To set a limit on the message processing rate in a service, we need to apply the Throttling Pattern *[MSDN TP]*.

Forces and Solution

The Throttling Pattern works by keeping track of the system load and reacting when a certain limit is reached. In a pipes-and-filters-based application, this is centred on the number of requests that are processed in a sliding time window, and reacting when the limit is reached. The way the Throttling Pattern reacts when the limit is reached depends on the use case; most often, it uses one of the following throttling strategies:

- **Reject:** this is the simplest strategy to implement from the throttler point of view. If a throttling limit is reached, the service starts rejecting new requests until the request rate is again within the limit. It is the responsibility of the service consumer to deal with the rejected request (for example, by retrying the operation at a later point in time).
- **Block:** when the limit is reached, rather than rejecting the new request, this strategy blocks (or uses) the calling thread until the request rate is again within the limit.
- **Delay:** this option does not reject the request or block the calling thread; instead, it puts the request into an in-memory queue (or thread pool) for a delayed execution by honouring the enforced rate. The calling thread is released, but the request is held and delayed in a queue.

- **Degrade:** the idea of this approach is to provide an alternative path for the requests when a rate limit is reached.

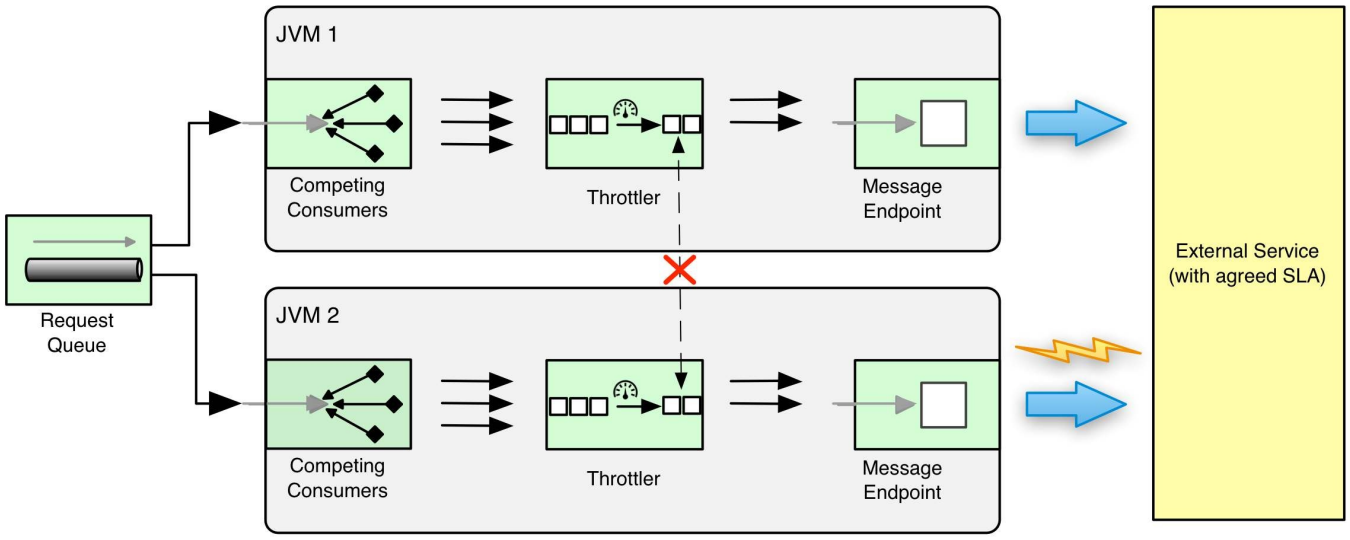
This pattern is ideal for handling an external request burst, a single tenant monopoly, meeting SLAs, or simply for cost optimizing and resource limiting.

Mechanics

This pattern is easy in theory, but in practice it can be implemented at various levels and it is quite hard to implement in a clustered environment.

Throttler EIP

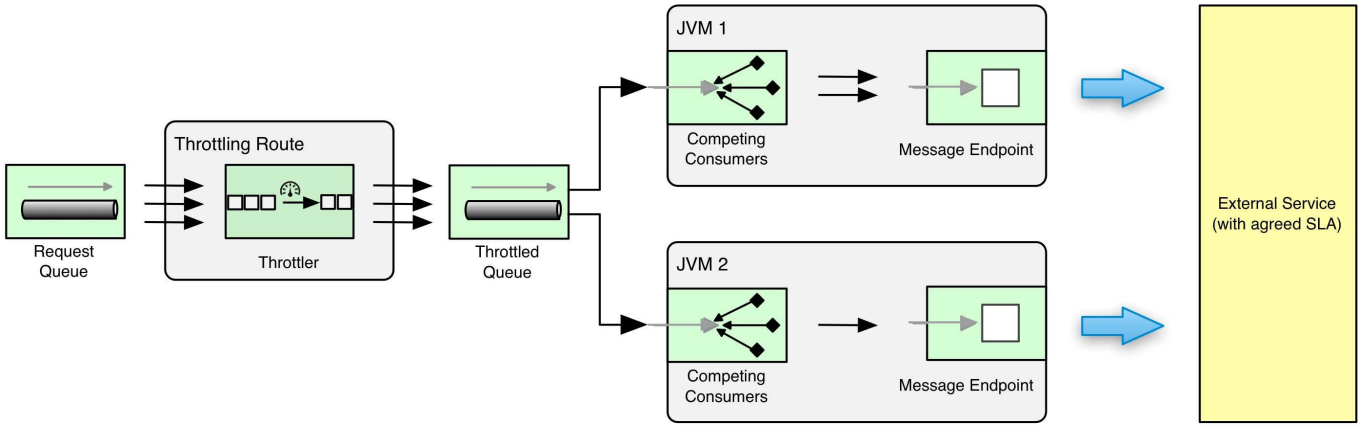
The obvious choice for implementing this pattern is to use the Camel Throttler EIP *[CAMEL THROTTLER]*. But one very big limitation of this implementation is that the Camel Throttler EIP is a stateful pattern and currently there is no way to externalize the state. Each instance of the Throttler EIP has an internal state which keeps track of the number of requests for the configured time window. Since the internal state is not shared, it is very easy to exceed the rate limit when there are multiple instances of Camel routes on separate JVMs with throttlers. To illustrate the problem let's have a look at an example route. Let's assume we have an SLA with a maximum of three requests per second. Once we have configured our Throttler EIP, we may need to scale our Camel route by starting multiple instances of it on separate JVMs (the Service Instance Pattern).



Non-clustered Throttler EIP example

With this setup both throttlers will individually try to prevent the requests from exceeding the rate limit, but since they are not synchronized, it is possible for the overall request rate from both service instances to exceed the limit. Even if we set a lower rate of two requests per second on both throttlers, it is still possible for both instances combined to reach four requests per second and exceed the limit. Notice that this limitation happens only when the throttler instances are on separate JVMs. If the throttler constructs are deployed to the same JVM, even if they are on separate Camel routes, or separate `CamelContexts`, you can reuse the same throttler instance by calling the single throttling route through direct or direct-vm components. This technique can be used to have a single throttler instance and

avoid issues faced by multiple non-clustered throttler instances. One not so great workaround for cross-JVM throttling could be the following. If you need a global throttler, then you can first let one consumer read the messages off the queue, throttle the messages as per SLA, then resend them to an intermediary throttled queue for further processing by the competing consumers.



A single Throttler EIP instance

This approach only works if there are no messages accumulating in the throttled queue. If the messages do accumulate here, then it is possible again for the competing consumers to temporarily exceed the SLAs while catching up with the message backlog. It is not a great solution, but it may work in a very specific context.

Throttling Route Policy

We have seen how the Camel Throttler EIP can limit the number of exchanges passing through a Camel route for a given time window. But there is also another throttling mechanism available in Camel called the `ThrottlingInflightRoutePolicy` *[CAMEL THROTTLING]*. This throttler is more coarse-grained and it is based on a number of `inflight` exchanges for a given `CamelContext` or Camel route. When the number of exchanges that are still being processed reaches a certain limit, the `ThrottlingInflightRoutePolicy` suspends the Camel routes, and resumes them when the exchange count is back in the limit. This throttler can be used to avoid fast message consumption by stopping the consuming Camel routes, or to prevent having a large number of total exchanges (for example, if these are with huge payloads and take a large amount of the heap).
If we compare both throttling mechanisms, the Throttler EIP counts messages passing though the throttler as part of the Camel route, whereas the `ThrottlingInflightRoutePolicy` counts all `inflight` messages belonging to a Camel route or `CamelContext`, regardless of their position in the processing flow.

Other Mechanisms

There are also other ways to control the throughput of an application. Let’s see a few more examples:

- ActiveMQ has a feature called **Producer Flow Control** *[ACTIVEMQ PFC]* where if the broker detects that the memory limit for the destination has been exceeded, then

the flow of messages can be slowed down. The message producer will be either blocked or rejected until resources are available again. This feature is essential for a message broker to prevent it from becoming flooded when there are fast producers and slow consumers.

- Similarly the in-memory asynchronous **SEDA component**, where the exchanges are kept in a blocking queue, can throttle too. When the SEDA queue is full, it can either block the SEDA producer, or throw an exception and reject the exchange until new messages can be accepted.
- HTTP, for example, the **Jetty server**, can be configured with a fixed maximum number of connections rather than the default unbounded queue *[JETTY]*, or activate a low resources monitor in Jetty that can take actions when the Jetty Server is running low on resources.
- You can go even further and use the **TCP flow control** by tuning the operating system's TCP buffer sizes, queue sizes, file descriptors, etc.
- **A custom RoutePolicy** in Camel is used to control route(s) at runtime by starting (resuming) and stopping (suspending) them. This is a natural extension point in Camel where a custom logic can control the route life cycle. The existing route policies, `SimpleScheduledRoutePolicy` and `CronScheduledRoutePolicy`, can also be used to control the route state based on a time dimension rather than a message load.

More Information

[MSDN TP] *Throttling Pattern in MSDN*
[CAMEL THROTTLER] *Camel Throttler EIP - Apache Camel*
[CAMEL THROTTLING] *Route Throttling Example - Apache Camel*
[ACTIVEMQ PFC] *Producer Flow Control - Apache ActiveMQ*
[JETTY] *Limiting Load - Jetty*

13. Circuit Breaker Pattern

This is known only under this name so far.

Intent

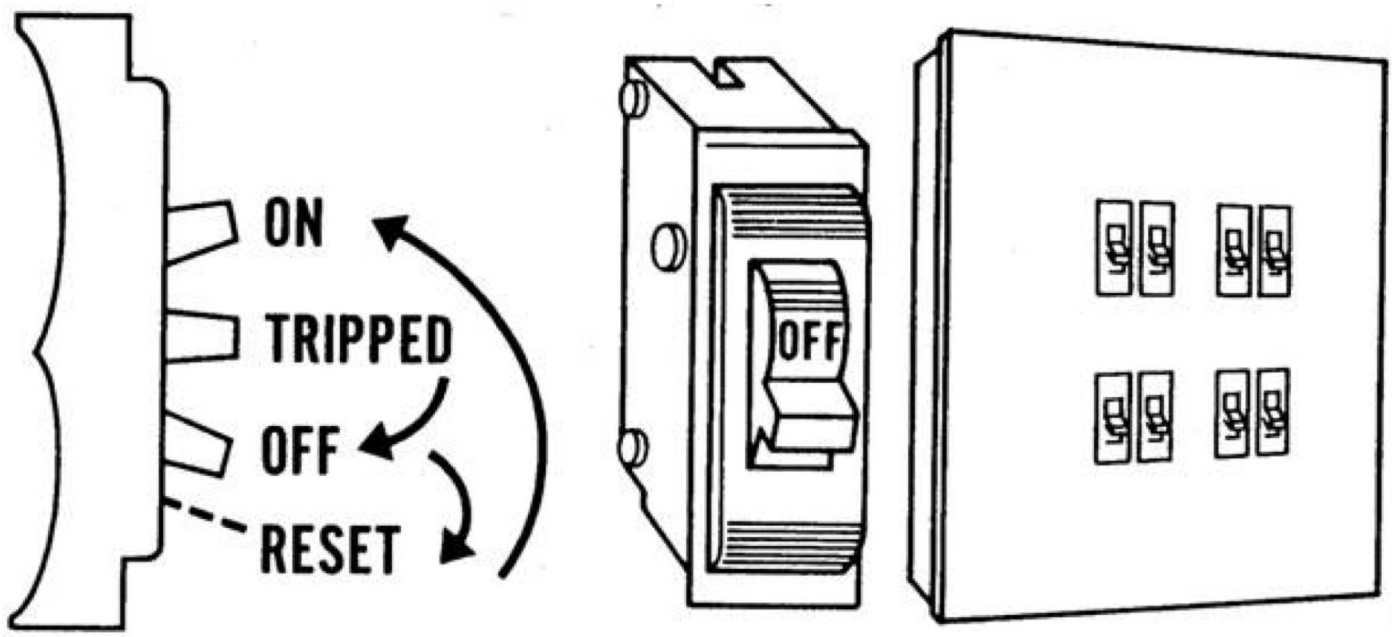
This is to improve the stability and the resilience of a system by guarding integration points from cascading failures and slow responses from external systems.

Context and Problem

We have seen how the Retry Pattern can help us to overcome transient failures, such as slow network connections or temporarily overloaded services, by retrying the same operation. But sometimes a failure may require human interaction to rectify it or its recovery can take a long time. In similar situations there is no benefit in retrying the same operation repeatedly and putting additional load on an already struggling system. Instead the application should detect that the remote service is unavailable and deal with it in a graceful manner until the service is back and healthy again. As an example, if we have a service that interacts with a remote Web service, and this third party facility suddenly starts acting very slowly, this will affect our service too. Having many threads waiting on responses from the remote service could soon exhaust the resources on our system and prevent the handling of new requests. Having a shorter timeout can prevent hanging requests and cascading failures but may cause the invocations to time out even if the remote service finished successfully later. Shorter timeouts can also not prevent having concurrent requests waiting on the remote service and potentially taking up resources. We need a way to detect failures and to fail fast in order to avoid calling the remote service until it fully recovers.

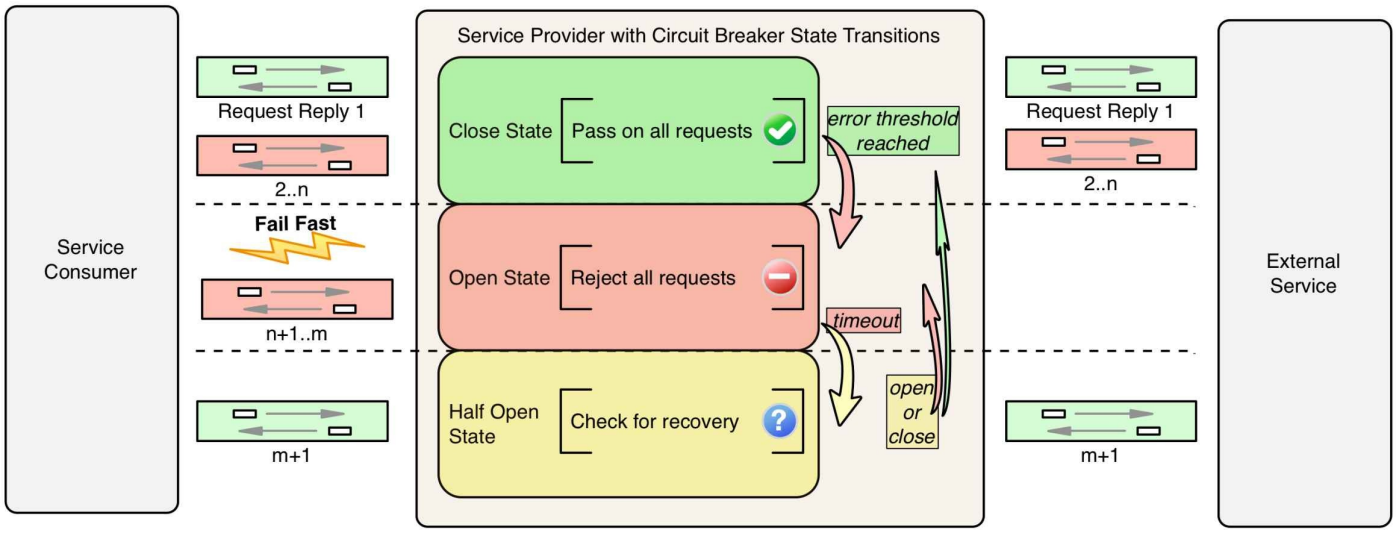
Forces and Solution

The solution for the problems is the Circuit Breaker Pattern *[RELEASE IT] [MSDN CBP]* as described by Michael Nygard in his *Release it!* book. It is inspired by the real world electrical circuit breaker systems which detect excessive current draw and fail fast to protect electrical equipment. Once the fault is fixed, the circuit breaker can be reset to restore the current to the entire system.



Electrical Circuit Breaker

The software-based Circuit Breaker Pattern works on the same principle. It wraps the unreliable operation and monitors it for failures. Once the failure rate reaches a certain threshold, the Circuit Breaker trips and avoids further calls to the failing operation. In this state, the Circuit Breaker returns an error without invoking the protected operation and it avoids putting additional load on the struggling service. Not calling a slow or failing operation also reduces resource leakage and improves service stability. After a short period, the software-based Circuit Breaker can also probe and detect if the protected resource has recovered, and then resume the normal processing.

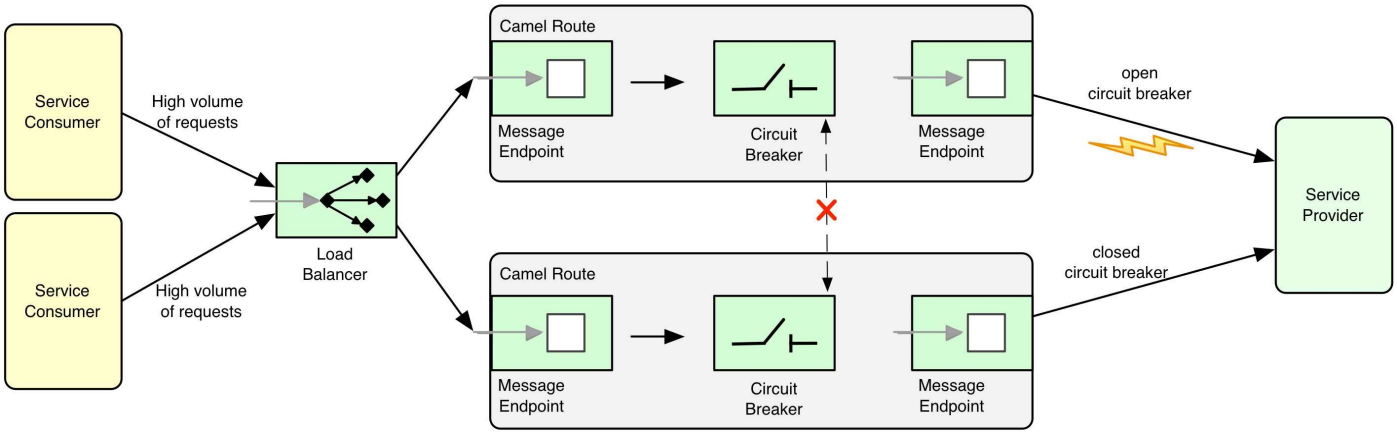


Circuit Breaker state transitions

At first sight, the Retry Pattern and the Circuit Breaker Pattern seem similar as both make a service resilient to failures in external systems. But the difference is that the Circuit Breaker Pattern avoids executing an operation that is likely to fail, whereas the Retry Pattern invokes the same operation again hoping it will succeed. The first one is good for dealing with permanent and longer-lasting failures and the latter for overcoming transient failures.

Mechanics

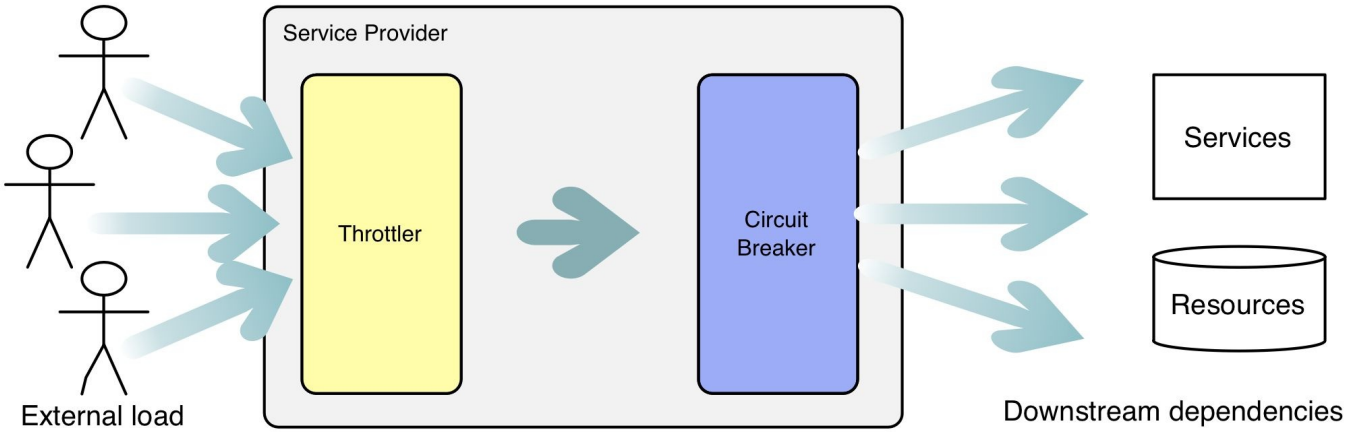
Using the Circuit Breaker Pattern in Camel is straightforward, but keep in mind that it is a stateful pattern that does not externalize its state. Other patterns such as the Idempotent Consumer EIP are stateful too, but it is possible to use an external persistent store and have multiple filters share it.



Non-clustered Circuit Breaker example

With the Circuit Breaker Pattern, the number of failures is stored in every instance and it is not possible to have a clustered behaviour. So, if you apply the Service Instance Pattern and have multiple instances of a Camel route with Circuit Breakers, each one will be tracking the number of failures separately, which might not be what you intended in the first place. With this setup, it is possible to end up with a circuit breaker that is in a closed state and another one that is in an open state, regardless of the fact that both point to the same target system.

The Circuit Breaker and Throttler patterns are similar in some regards: they both protect the system from external influences. But there is an important difference, which is visualized in the diagram below:



Throttler and Circuit Breaker compared

The throttler protects the system from an incoming external load, whereas the circuit breaker protects the system from external dependencies that are not available.

More Information

[RELEASE IT] *Circuit Breaker - Release It! by Michael T. Nygard*
[MSDN CBP] *Circuit Breaker Pattern in MSDN*

14. Error Channel Pattern

This is also known as the Dead Letter Channel and the Invalid Message Channel *[EIP]*.

Intent

These are a set of patterns and principles used for error handling in integration applications with different conversation styles.

Context and Problem

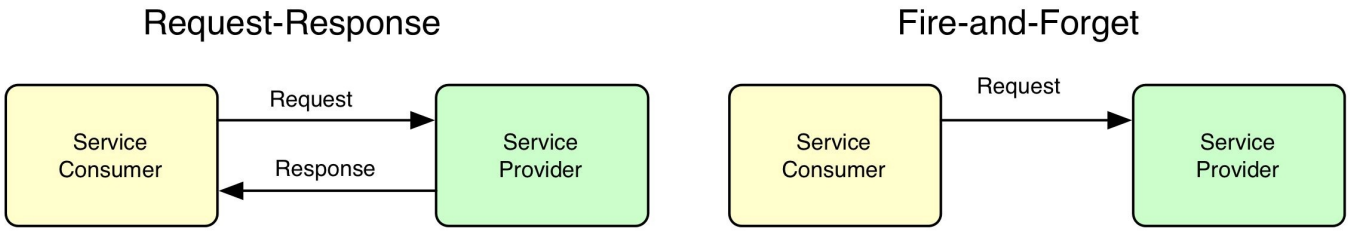
Distributed systems run on multiple processes and interact with some kind of inter-process communication mechanism. Different interaction styles require different ways of handling and dealing with error conditions. Following some principles and patterns make designing and implementing the error conditions easier.

Forces and Solution

Depending on the number of participants, the interaction can be one-to-one or one-to-many. Depending on whether the service consumer expects a response or not, the interaction is classified as synchronous or asynchronous. Based on these dimensions, there is the following combination of interaction styles:

Interaction Style	One-to-One	One-to-Many
Synchronous	Request-Response	N/A
Asynchronous	Fire-and-Forget	Publish-Subscribe

There are many aspects of service interactions. “Conversation Patterns” by Gregor Hohpe *[CONVERSATIONS]* is a great article (and hopefully a future book) covering the interactions between loosely-coupled services. In this chapter, we will not look at service interaction mechanisms, but focus instead on designing the service provider from an error handling point of view. The two styles we will look at are Request-Response and Fire-and-Forget.



Request-Response and Fire-and-Forget interaction styles

Request-Response

This is a conversation style where information is transmitted in both directions. An integration application may have to do a Request-Response through different kinds of data exchange mechanism:

- The most widely used Request-Response interaction is over HTTP. An HTTP interaction involves exchanging two messages over a single TCP/IP connection.
- Another popular Request-Response interaction mechanism is through messaging. Messaging channels transfer data only in a single direction, but using two separate channels it is possible to do a Request-Response. Since there are two separate channels the service consumer needs a way to match the request and the response; and this is achieved through a Correlation Identifier Pattern. When the service consumer sends a request, it also provides a unique request ID. The service provider can use this request ID for some processing (for example, to make the operation idempotent in case the request is retried), but also to add it to the response message as a correlation ID. The service consumer can then match each response to the originating request with matching IDs.

Regardless of the transport mechanism, from the perspective of this chapter, the important point of Request-Response is that there is a way for the service provider to return the result of the request. So for this kind of interaction, the most logical way to deal with errors happening while processing a request is to propagate the error back to the service consumer. It is then the responsibility of the service consumer to interpret the error and take adequate action, such as retrying the request, trying a different request, or giving up. Notice that it is not important whether the error is caused by the service consumer (by sending an incorrect request) or by the service provider (a third party service or a resource not being available). In either case the error is propagated back to the caller.
While propagating the error to the caller, it is important to apply the Exception Shielding Pattern and sanitize unsafe exceptions. Leaking information such as connection strings, SQL queries, or XPath commands, can become a potential security threat.

Fire-and-Forget

This is the simplest conversation style where information is transmitted only in one direction. The most popular one-way interaction is through messaging. Another Fire-and-Forget style of interaction is, for example, through file transfer. If you think about it, writing files to a folder is not any different to sending messages to a queue. And reading files from a folder, processing them, and then deleting them is not different to consuming a message from a queue.
Fire-and-Forget is highly scalable as the service consumer does not have to wait for a response, but it has one drawback (by design) in that there is no way for the service provider to return the result. So if a request has failed, there is no way of telling this to the service consumer (the service consumer might not care, or might not be equipped to deal with errors either). There are other patterns and conventions that can help us with Fire-and-Forget error conditioning:

- **Invalid Message Channel (IMC):** as we have seen with the VETRO Pattern, one of the very first actions performed by a service is to validate the request. Even if there is

no explicit validation, some of the processing steps will try to interpret the request and find out if it is valid or not. A message may be invalid if its data format is different, or a mandatory header is missing, or for any other reason that the message consumer decides. If the message is not good enough for processing it is moved to the Invalid Message Channel. So rather than keeping the message in the queue, rolling back the transaction and moving the message to a DLQ, or retrying to consume the message, it is moved to an IMC. An IMC is like an error log for messages. It contains messages which were delivered to the service, but not fully processed because of issues with the message itself.

- **Back Out Channel (BOC):** message backouts are used in IBM WebSphere MQ to remove the so-called poison messages from the queue. The Back Out Channel is similar to the more generic concept of the Dead Letter Channel, but with a difference in that the Dead Letter Channel is used by the messaging system when it cannot deliver a message, whereas the BOC is used by the application to collect failing messages. When a message cannot be processed successfully due to a failure in a third party system, after a couple of retries the message is backed out to the BOC for a later processing. Then someone can review the BOC and replay the messages to the original source channel for reprocessing.
- **Dead Letter Channel (DLC):** in contrast to the IMC and the BOC, which are used by the message consumer application to move failing messages, the DLC is used by the messaging system itself to deliver messages that couldn't reach their designated destination. The reason for such a failure can vary from a misconfiguration of the channels, the producing application not having permission for a certain channel, to missing headers on the message. A message with TTL after expiry should not be delivered to the consumer, so the DLC is the logical destination for it. The DLC is also a catch-all channel for all failures. If a consumer service does not have the concept of the IMC and the BOC, and simply rolls back the transaction, the message will eventually end up in the DLC.

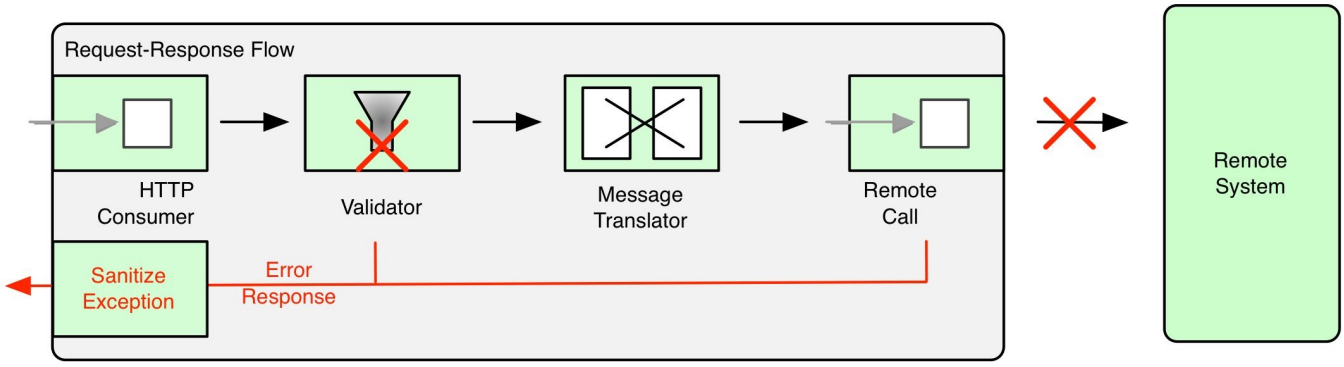
All of these different types of channels are just concepts to give semantic meaning to different kinds of failures: validation failures that should not be retried, application failures that can be retried by replaying messages, and messaging system failures that are worth investigating. Other than the semantic differences, there is no difference between a queue (or even a folder) used for IMC, BOC, or DLC.

Message Channel	Implementation layer
Invalid Message Channel (IMC)	Application layer
Back Out Channel (BOC)	Application layer (or messaging subsystem in WebSphere MQ)
Dead Letter Channel (DLC)	Messaging subsystem

Mechanics

Request-Response

To illustrate the error handling for this message exchange pattern, let's have a look at the following Camel route which receives HTTP requests and calls an external system. The preferred way for dealing with unrecoverable errors (that is, after retries, for example) in this situation is to propagate the error back to the caller.

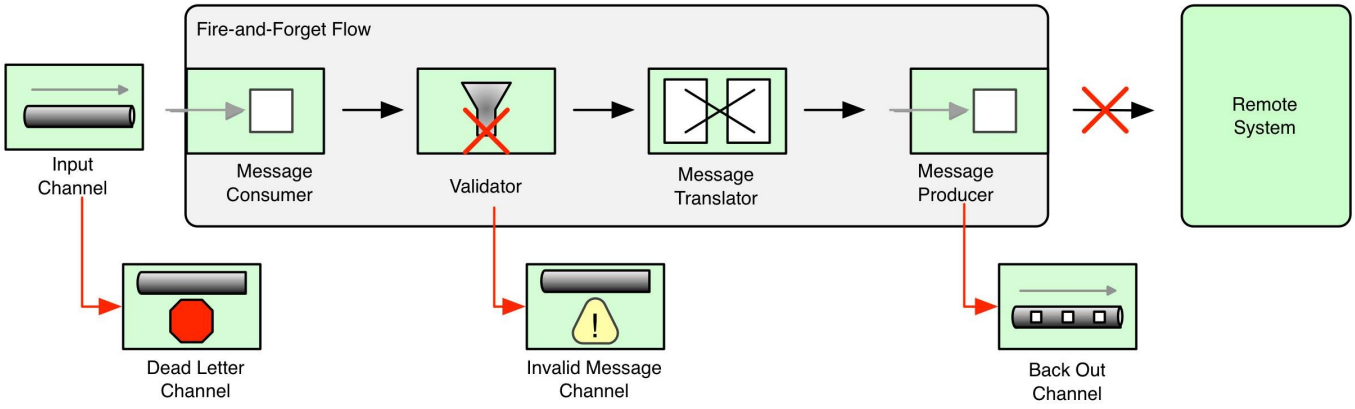


Synchronous Error Channel

The default error handler in Camel will do the job, but it is important also to sanitize the response before returning it. That's why having a global error handler that will catch any exception and sanitize it might be the safest option. Also, depending on the consumer endpoint that is in use, you may have to clean the message headers to prevent the leaking out of Camel-specific information in the response. For example, some components support the `headerFilterStrategy` option that can control which incoming and outgoing headers to pass and which to not.

Fire-and-Forget

For this conversation style, the first step is to configure the messaging system (if messaging is used). ActiveMQ by default will have one default DLC for the whole broker (named `ActiveMQ.DLQ`), but it might be easier to track messaging failures by configuring ActiveMQ to use a separate DLC for each queue. This is usually following the convention `DLQ.ORIGINAL_QUEUE_NAME`, but it can be customized if a different naming convention is preferred. Once the DLC is configured, optionally you may decide to use IMC and BOC. If the queue is internal, you may skip the validation step totally, but for external queues, where the message publisher is controlled by other teams, validation is a good idea.



Asynchronous Error Channels

Validation is performed at the beginning of the flow and if it fails, the message can be moved to an IMQ without any revalidation. Populating the failed message with some extra information before sending the IMC can be very helpful during the investigation of failures. You can populate the message with headers about the error message, the full stack trace, the original message that was received and anything else that might help to identify the reason for the failure. To enrich the message with more information before sending it to an IMC, you can assign a processor to the error handler through the `onPrepareFailureRef` construct, then access exception-related information from exchange properties such as `Exchange.EXCEPTION_CAUGHT` and `FAILURE_ENDPOINT`.

When a message has passed the validation step, but fails due to a failure in other systems or resources not being available, then it is moved to a BOC. Depending on the nature of the failure, the operation might be retried before sending it to a BOC. Camel has the notion of “exception policies” that allows the further customizing of the error handling behaviour per exception basis. The exception policies (used through the `onException` construct) allow you to specify error handling behaviour per exception type. Camel can do exception matching through a clever gap analysis method, and perform a redelivery specific for the exception that has occurred. You can tune this behaviour even further through the `onWhen` predicate. This filter lets you access the exchange for writing predicate logic to decide whether the error handler should be triggered or not. So on top of the exception-type match, you can perform further logic analysis (for example, to check a specific HTTP header value), and decide which error handler to trigger and how many redeliveries to perform.

More Information

[EIP] *Dead Letter Channel - Enterprise Integration Patterns by Gregor Hohpe and Bobby Woolf*
[EIP] *Invalid Message Channel - Enterprise Integration Patterns by Gregor Hohpe and Bobby Woolf*
[CONVERSATIONS] *Conversation Patterns by Gregor Hohpe*

III Deployment Patterns

Having designed integration flows considering the happy and unhappy paths, the next step is to take a higher-level view and consider how these integration flows are interacting with each other and why.
The patterns in this category are not focused on the structure of individual services, but rather on the structure of the distributed system as a whole.

15. Service Instance Pattern

This is also known as Service Load Balancing [CLOUD PATTERNS].

Intent

This accommodates increasing workloads by distributing the loads on multiple service instances.

Context and Problem

The quality attributes of a software system are non-functional properties that are driven by the architectural constraints of the system. There are many quality attributes and their priorities vary largely from project to project, but in the integration world there are a few that are almost always present: availability and scalability:

- Availability is the proportion of time that all components of the system are fully functional and operational. It is measured as a percentage of the total system downtime over a fixed period.
- Scalability is the ability of a system to handle increases in the load without impacting on the performance.

An application can be scaled vertically or horizontally. Vertical scaling (scaling up) is achieved by adding more resources to a single node in a system to give greater capacity to this node. This way of scaling is limited by the amount of CPU/RAM that can be added to a single host. There is another way of increasing the capacity of a system that can be achieved with commodity hardware (rather than supercomputers) called horizontal scaling. Vertical scaling increases the capacity of a single node, but it does not decrease the overall load on the nodes. Horizontal scaling (scaling out), on the other hand, increases the number of nodes in a cluster and reduces the load on the individual nodes. In this setup the capacity of each node stays the same, but the load on the nodes is decreased as it is spread across more nodes. While vertical scaling is easier, it is more expensive and with a limited reach. On the other hand, horizontal scaling is cheaper (from a hardware point of view) but harder to implement and sometimes not even possible. The Service Instance Pattern *[SOA PATTERNS]* is the result of applying horizontal scaling principles to the application layer.

Forces and Solution

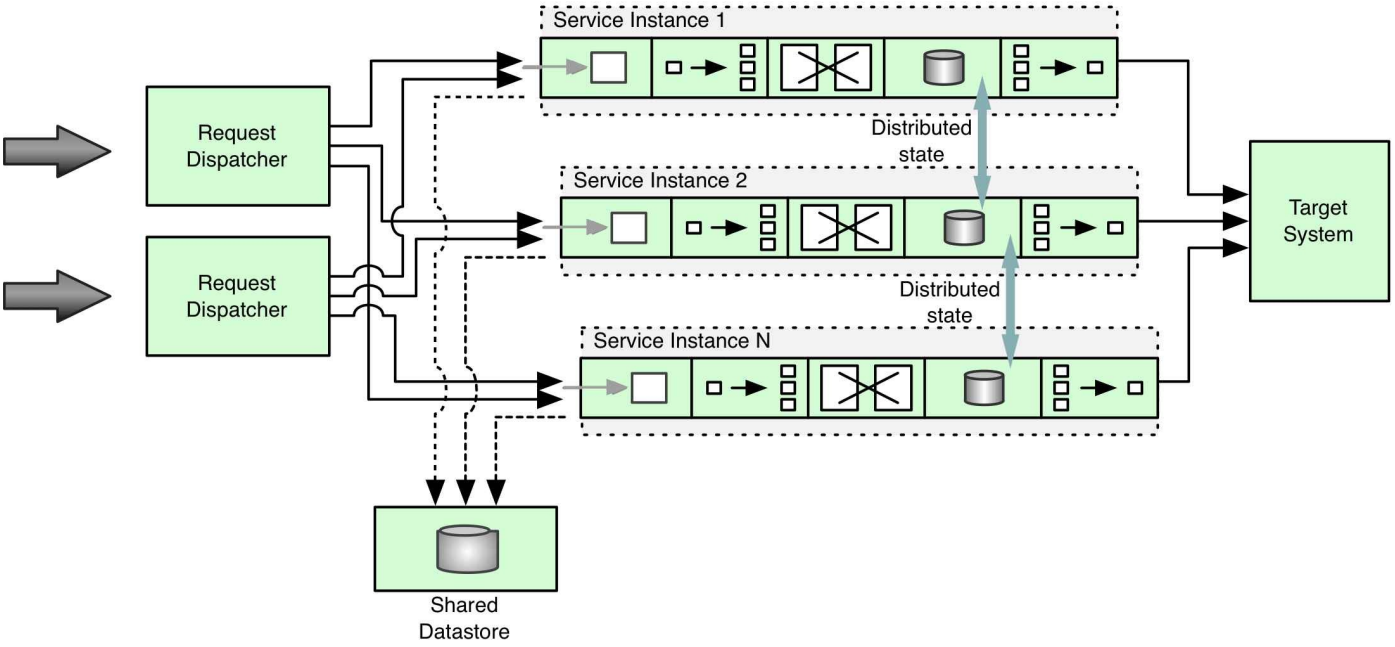
Starting multiple instances of a service and distributing the load to all instances allows us to increase the overall capacity of the system and achieve high availability through redundancy and avoiding single point of failure. There are few areas to consider before implementing the Service Instance Pattern that we will look at this section.

- **Service state:** in the SOA and integration world, ideally we want to have stateless services whenever the business requirements allow it. But what exactly is a stateless service *[SOA PRACTICE]* in this context? This is not a service that does not interact with an external state (such as a database) or a service that does not have any local state (local variables for processing a request), but rather a service that does not maintain any state between different service calls; that is, if all the service data used to process a request (temporarily-created local data, thread context, etc.) are discarded or persisted to an external service data store. In such a case, if the subsequent service calls go to a different thread or process instance running the service, the request is processed successfully.
  The main benefit of stateless services is that they are easier to load balance and fail over through the Service Instance Pattern. If the throughput of a service is not high enough, it is easy to start more instances and scale linearly. If a service fails, it is easy to start another instance or fail over to other running service instances.
  Stateful services on the other hand, are client-session-bound and require resource allocation that spans multiple service invocations. In addition, applying the Service Instance Pattern requires session affinity (i.e. sticky sessions) and/or session clustering across service instances. Sticky sessions require a state (that is, client to service instance mapping) that is kept either with the client side (through a cookie) or with the request dispatcher (the load balancer maps clients to service instances). Session clustering also requires sophisticated clustering technologies, all of which makes scaling stateful services difficult.
- **Request dispatcher:** this is responsible for distributing the load to the service instances. Certain endpoints will require an external request dispatcher (like an HTTP load balancer) and some will support request dispatching out of the box (such as message queues with competing consumers, or reading files from a folder). If the services are stateful, the request dispatcher may have to also provide the session stickiness. One of the motivations and benefits of the Service Instance Pattern is the HA. But for true HA it is important to avoid any single point of failure, including the one in the request dispatcher.
- **Message ordering:** depending on the business logic, it is possible that a service will be sensitive to message/request ordering. While this might not be obvious when there is only a single instance of the service, it might break the business logic when there are multiple service instances processing the requests in parallel, and those requests are getting out of order depending on the available resources of each instance. If this is the case, you may have to keep certain services as singletons (through the Singleton Service Pattern) or implement some kind of message sequencing logic to discard out of order messages.
- **Resource contention:** services in integration applications may use all kinds of endpoints and some are very sensitive to concurrent usage. Before running multiple instances of a service, it is important to check each endpoint for possible side effects. We will see a few examples below.
- **Resource coupling:** most services depend on other services and resources. And sometimes these dependencies happen to be part of the same process, running on the same container as the service itself. Usually for ease of use (and using performance as a pretext) it is possible to access the dependencies through an in-memory call in

the process itself. This creates issues for applying the Service Instance Pattern, as the service expects to be running in the same process as the dependencies.
- **Singleton service:** certain services (such as batch jobs) can benefit from the Service Instance Pattern but only for HA purposes through active/passive clustering. Because of the business logic in these services, there can be only a single instance that is active at a time. This use case is described in detail in the Singleton Service Pattern chapter.

Mechanics

Here we will see example use cases from each category, mainly covering Camel and related technologies. In the diagram below we have a service that is started by three instances. The services are accessing the same data source and also have a distributed embedded cache. As such, these services are stateless and can benefit from the Service Instance Pattern.



Clustered service instances with shared data store and embedded distributed cache

Service state

You may expect that multiple service instances will behave logically as one service when clustered. But there are a few Camel EIPs that are stateful and that may cause unexpected behaviour when the Service Instance Pattern is applied:

- **Load balancer:** this EIP supports a number of different strategies:
  - *Round-Robin strategy* - this has an internal counter of the last processor used.
  - *Weighted Round-Robin strategy* - again, this has an internal counter of the last processor used.
  - *Sticky strategy* - this internally relies on the Round-Robin strategy for the initial mapping of the session key to processors.
    A possible side effect when the above strategies are used in a multi-instance setup is that every service will do its own Round-Robin distribution starting from the first endpoint. So if there are three services, the first request reaching

each service will be forwarded to the first processor, and rather than having the three requests distributed in Round-Robin order, they will be sent three times to the same processor - not exactly Round-Robin.
    - *Circuit breaker strategy* - this has an internal state used for keeping the `lastFailure` time stamp. Based on this field, the circuit breaker state transitions are calculated. A possible side effect could be that the circuit breakers will be in different states in different service instances. So some requests will hit a closed circuit breaker and be allowed to pass, and some requests will hit an open one and be rejected.
- **Resequencer:** this has an internal queue with exchanges to keep them for reordering. A side effect will be that each service instance will perform its own resequencing. As a result, it is not possible to have a clustered global resequencing that spans service instances.
- **Sampling throttler:** this has internal fields to keep a `currentMessageCount` and a `timeOfLastExchange`. As a result, sampling will be performed per service instance rather than clustered.
- **Throttler:** as explained in the Throttler Pattern chapter, throttling will happen per service instance rather than per global throughput rate. As such, it is possible to exceed SLA rates.
- **Delayer:** similarly to Throttler, it is not possible to have a clustered delayer applied; instead, each service instance will apply its own delays.

  The next few EIPs are also stateful, and by default they will use in-memory persistence. But the good thing about these patterns is that their implementation do allow the use of different persistence implementations that can be shared or clustered:

- **Idempotent consumer:** this is a well-described use case in the Idempotent Filter Pattern chapter. `MessageID` repositories such as the `InfinispanIdempotentRepository` and the `HazelcastIdempotentRepository` are great for clustering and linear scalability of the Idempotent Consumer EIP state. Using a shared database for all service instances with the `JpaMessageIdRepository` or the `JdbcMessageIdRepository` will also work fine.
- **Aggregator:** when multiple service instances are aggregating messages, the default in-memory `aggregationRepository` used by the Aggregator EIP will not suffice. Repositories based on Hazelcast, Cassandra, or JDBC will do the job, though.

Camel also has components that do help clustering and stateless services such as Infinispan, Hazelcast, Redis, JGroups, and many others. Do check these components before implementing the Service Instance Pattern.

Request Dispatcher

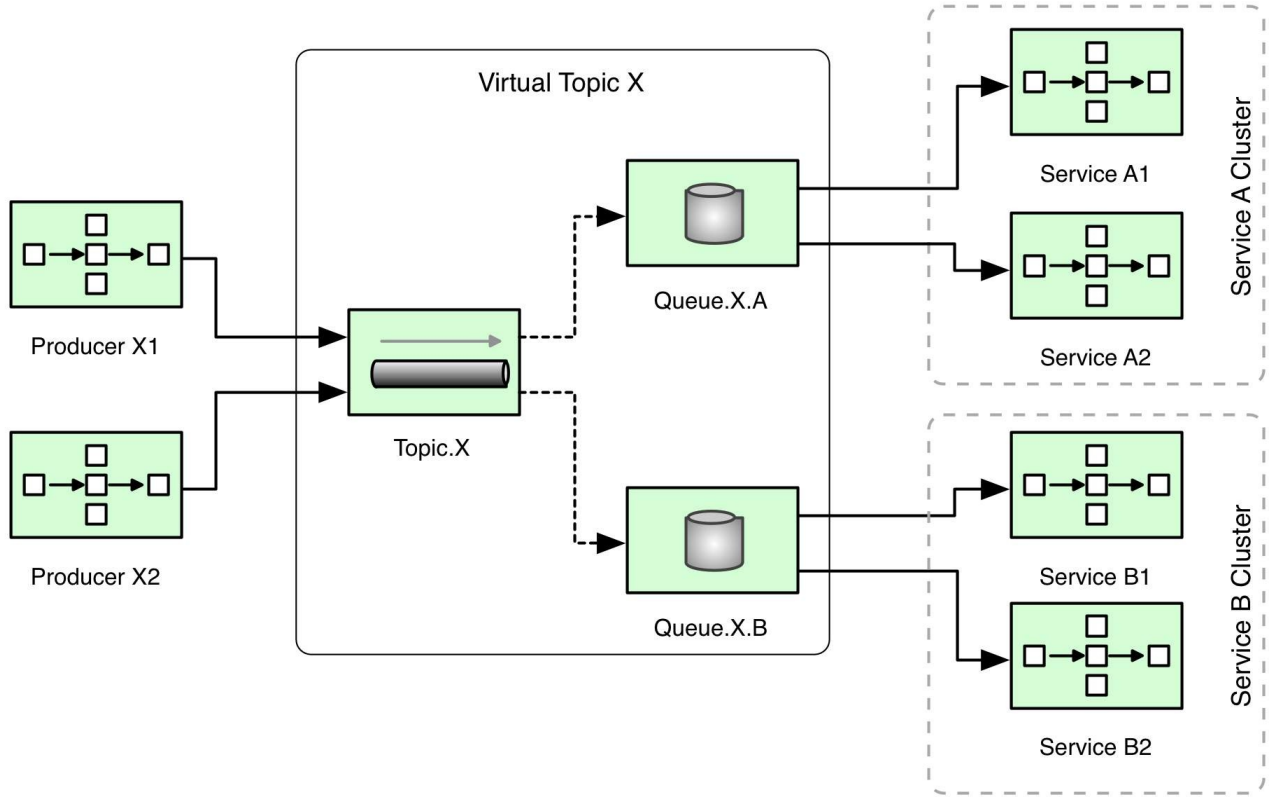
Implementing the Service Instance Pattern requires a mechanism that will distribute the load among all service instances. This mechanism is endpoint-specific and for an integration application that supports a multitude of endpoints, it will vary widely. Messaging endpoints are the easiest to implement as message queues do support load balancing and the failover of clients through the Competing Consumers Pattern. HTTP endpoints will require an external load balancer (such as the HA proxy, F5's BIG-IP, or the

Fuse HTTP Gateway) that does the load balancing and failover. In addition, the Fuse HTTP Gateway performs service discovery too, a very useful capability in a JBoss Fuse environment.
Picking up files from a shared folder does not require a dispatching mechanism, but requires coordination/clustering of file consumers. Using a distributed `idempotentRepository` and `inProgressRepository` will ensure that file consumers do not pick up the same file, and using an `idempotent readLock` ensures a distributed read lock.

Message Ordering

Sometimes the business logic of the service dictates that the messages/requests should be processed in the order they are generated, processed sequentially, or maybe the same service instance should be stuck to, etc. If this is the case, there are a number of useful features in different endpoints:

- **ActiveMQ exclusive consumer:** the broker chooses a single message consumer to get all the messages for a queue and ensure message ordering. If this consumer fails, the broker fails over and chooses another consumer. This feature ensures singularity at a message consumer level with a failover capability.
- **ActiveMQ message groups:** this is an enhancement of the exclusive consumer feature to provide kind of a parallel exclusive consumers. So rather than dispatching all messages to a single (exclusive) consumer, the broker dispatches a group of messages (identified by the JMS header `JMSXGroupID`) to a single consumer. This feature makes certain that a group of messages will be dispatched to a single consumer, ensuring an ordering as per a group of messages. So in a sense it provides a sticky session capability to a messaging system.
- **ActiveMQ consumer priority:** the broker orders consumers by priority and dispatches messages to them with the highest priority first until its prefetch buffer is filled up, before moving to the consumer with the next highest priority.
- **ActiveMQ message priority:** this allows high priority messages to be consumed before low priority messages. For it to work, the message channel should have `prioritizedMessages` enabled, and the message priority specified with the `JMSPriority` header for every message.
- **ActiveMQ virtual topics:** if you are using queue semantics for interacting between services, starting multiple services will not have any side effects (other than loosening the message order). But if you need topic semantics, then starting multiple service instances as consumers will be problematic as each service instance will get a copy of the message from the topic. To illustrate the problem, let’s assume that we have a message producer that sends the same messages to service A and service B through topic X. If we apply the Service Instance Pattern, we will have multiple consumers from service A and service B subscribed to topic X. This is not what we want as every message from the topic will be delivered to all service A and B instances. Instead what we want is to have every message from the topic to be delivered to one Service A and one Service B instance. This is kind of a mixture of topic and queue behaviour; to be more precise, topic behaviour for the producer, and queue behaviour for each service.



One solution is to use Camel and do a simple routing from topic X to a queue for every service (queue.X.A and queue.X.B). Then the service instances can benefit from the Competing Consumers Pattern by subscribing to the queues. This is not a bad solution but every message from topic X has to be consumed from a Camel JVM and then delivered back to a Broker JVM, to queue.X.A, and to queue.X.B, which is unnecessary network overhead.
A better solution would be to run the Camel route in the ActiveMQ JVM, which is possible through the Broker Camel Component plug-in. This plug-in allows the embedding of Camel routes in the broker configuration and the intercepting of messages moving through the broker.
And an even better solution is to do all of the above without any coding, but declaratively. This is possible through the ActiveMQ Virtual Topics feature which allows the creation of topics which have queues subscribed to them just by following some naming conventions:

- **Resequencer EIP:** Camel can also be used for the reordering of out of order messages. It will collect a number of messages and sort them before dispatching further as part of the flow. The main drawback of this component is that it cannot be clustered.
- **Session affinity:** this technique is popular mainly with the HTTP protocol and allows service consumers to stick to the same service instance.

Resource Contention

Whenever the Service Instance Pattern is applied, you have to check every resource accessed by the service and ensure it can be accessed concurrently. For example, if you are consuming from a folder, make sure to use `idempotentreadLock` to prevent concurrent file

reads, or if writing to a database table, ensure that concurrent updates will not be blocked, etc.

Resource Coupling

Typically in Camel, JVM-wide coupling happens through VM and Direct-VM components which rely on the producer and consumer being on the same JVM. If these components are used for service communication, then the Service Instance Pattern cannot be applied for individual services.
Another example of where this kind of dependency exists is when the services rely on some kind of resource embedded in the same JVM, such as a cache or a search engine. Embedding such resources is problematic, as a scaling of the service will also require a scaling of the resources, but usually the resources are stateful and not easy to scale. So try to avoid direct in-memory access to resources as this creates runtime coupling that is difficult to break.

Singleton Services

This is a way of limiting the Service Instance Pattern to a single instance (with auto failover capability usually) and the techniques are described in the Singleton Service Pattern chapter.

More Information

[CLOUD PATTERNS] *Service Load Balancing by Erl Naserpour*
[SOA PATTERNS] *Service Instance Pattern - SOA Patterns by Arnon Rotem-Gal-Oz*
[SOA PRACTICE] *Services and State - SOA in Practice by Nicolai M. Josuttis*

16. Singleton Service Pattern

In the clustered JBoss EAP, the same concept is known as the HA Singleton *[JBOSS]*.

Intent

This is to ensure that only a single instance of a service can be active at a time.

Context and Problem

We have seen how the Service Instance Pattern can help us to improve the performance and availability of a service. By running multiple instances of the same service and having a request dispatcher capable of load balancing the requests to all available service instances, the system has increased throughput and availability. The availability of the system increases as well because if an instance of a service becomes unavailable, the request dispatcher detects this and forwards future requests to running instances. But there are certain use cases where only one instance of a service is allowed to run at a time. For example, if we have a scheduled job (a trigger-based service) and run multiple instances of the same job, every instance would trigger at the scheduled intervals rather than having only one trigger fired as expected by the business. Another example would be if the service performs polling on certain resources (a file system or database) and you want to ensure that only a single instance and maybe even a single thread performs the polling and processing. In all these and similar situations, you need some kind of control over how many instances (usually it is only one) of a service are active at a time, regardless of how many instances have been started. The Service Instance Pattern still can be useful to achieve high availability in these cases, but there is an extra capability needed to ensure that only one instance of a service is active at a given moment.

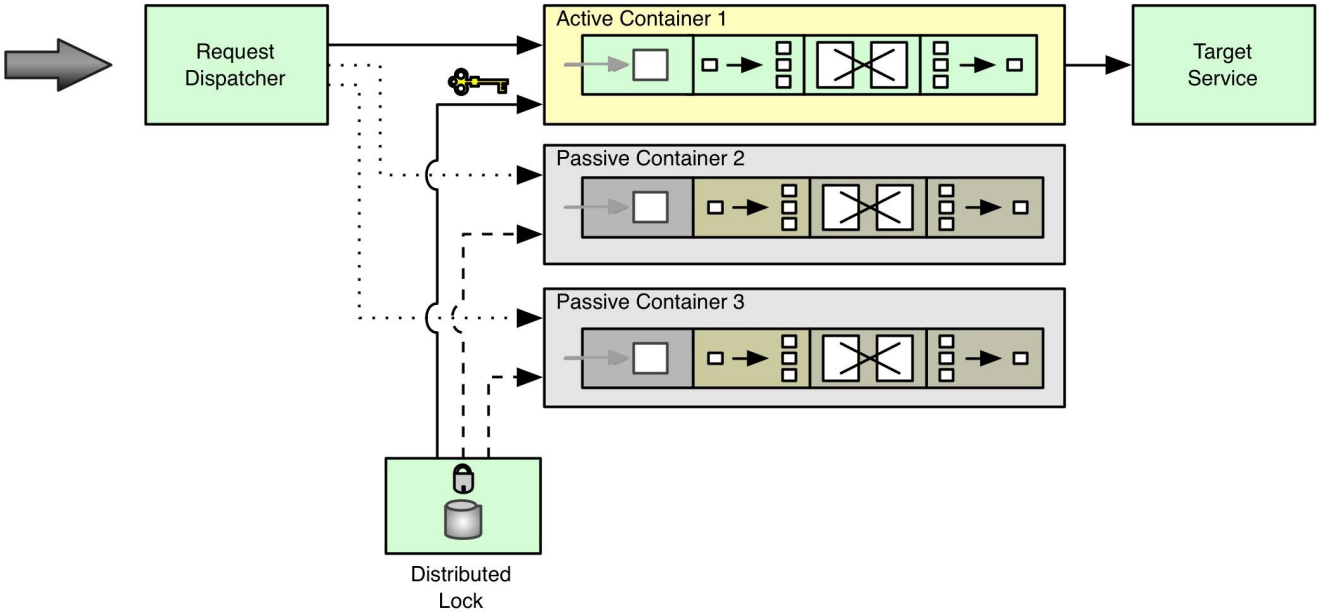
Forces and Solution

The Service Instance Pattern creates an active/active topology where all instances of a service are active and sharing the system load. What we need is an active/passive (or master/slave) topology where only one instance is active and all the other instances are passive. In a distributed system we can have control over the service instances through a distributed lock. Whenever a service instance is started, it will try to acquire the lock, and if it succeeds, then the service will become active. Any subsequent service that fails to acquire the lock will wait and continuously try to get it in case the current active service releases this lock. This mechanism is widely used by many existing frameworks to achieve high availability and resilience. For example, Apache ActiveMQ can run in a master/slave topology where the data source is the distributed lock. The first ActiveMQ instance that starts up acquires the lock and becomes the master, and other instances become slaves and wait for the lock to be released. Apache Karaf can also run in master/slave topology with a shared lock to achieve high availability.

Mechanics

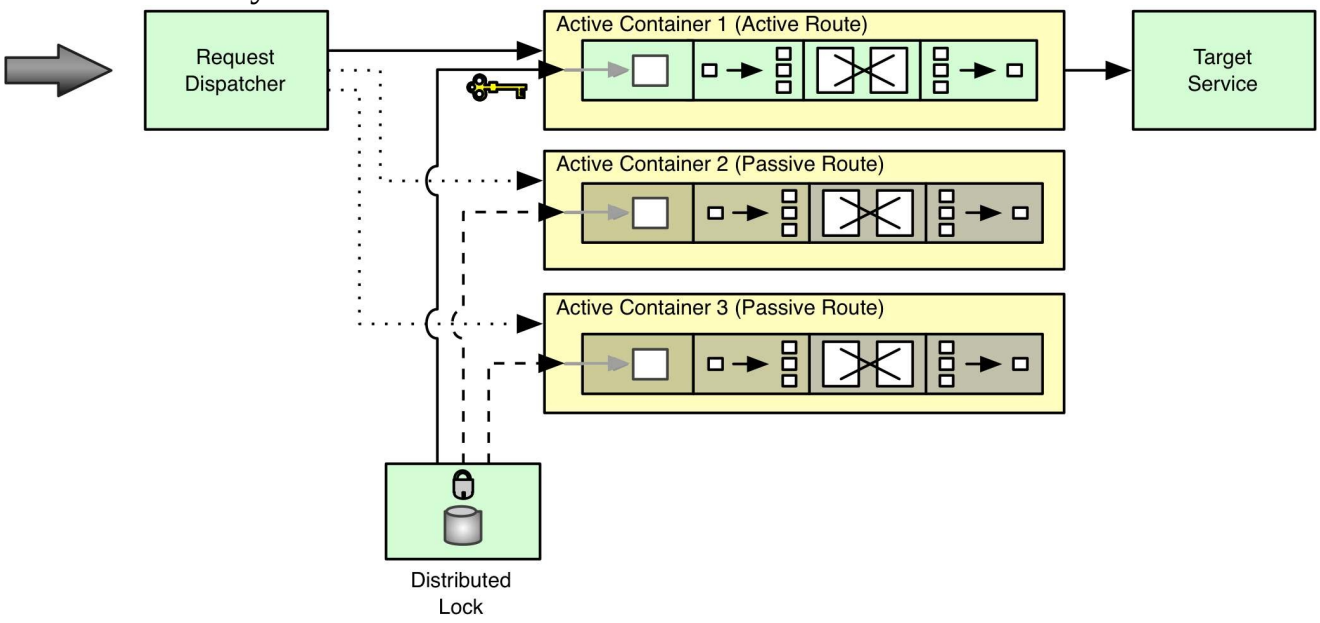
Camel does not have any mechanism at framework level for clustering, high availability, or singleton instances (yet). As such, clustering with singleton services has to be done in one of two possible layers: container (or process) or Camel route (through a component or `RoutePolicy`):

- **Container/process-based Camel singletons:** as the name suggests, this mechanism relies on the actual container or the container managing process to ensure that only a single instance of the Camel application is running. The Camel code itself is not aware that it is intended and will be run as a single instance. From this perspective it is similar to having a POJO that is created as a singleton from the managing container (such as Spring Framework).



  An example of a container-based Camel singleton is Apache Karaf. Two or more Karaf instances can be started in a failover active/passive mode, making sure that Camel routes are started only in the active instances.
  Another example can be having Camel routes deployed on a few clustered JBoss Wildfly servers, but activated only in one instance of Wildfly using JBoss HA Singleton beans. JBoss HA Singleton relies on JGroups as a central locking mechanism and lets you have active beans on one server only in the whole cluster.
  A third example would be relying on Kubernetes *[KUBERNETES]* to ensure that only a single instance of a docker container, with a Camel application deployed in it, is running. Even if this topology is not exactly active/passive (there is no passive instance), it has the same effect, as Kubernetes will ensure that there is only a single instance of the container running at a time.
- **Camel component-based singletons:** this mechanism has the Singleton behaviour embedded in the Camel route. From a code perspective, it would be similar to the original Singleton Design Pattern from GoF where the class has a private constructor and a public static field that exposes the only instance of the class. So the class is written in a way that does not allow creation of multiple instances in a JVM. Similarly, with Camel component-based singletons, the Camel route itself is written in a way that does not allow any more than one active instance at a time, regardless of the number of `CamelContext` instances that are started. To achieve this, the route

may use a component or a `RoutePolicy`, and it can even span multiple JVMs, but there will be only one active Camel route at a time.



There are a number of components in Camel offering this functionality with slight differences in behaviour:

1. **The Camel Quartz component** performs job scheduling and supports clustering when used with a persistent storage. If you are using quartz consumers in the clustered mode, you can have only one of the routes triggered at a time. Or if a quartz-based `CronScheduledRoutePolicy` is used in the clustered mode, only one of the routes will be started/stopped ensuring Singleton behaviour. Both of these options rely on having quartz to be configured with a data source that is shared among all the routes in the cluster.
2. **The Camel ZooKeeper component** offers a `RoutePolicy` that can start/stop routes in a master/slave fashion. The first route that gets the lock will be started where the remaining routes will be waiting to get this lock. One advantage of this component is that it can be configured to have more than one master running if needed.
3. **The Camel JGroups component** also offers master/slave capability using `JGroupsFilters`.
4. **The JBoss Fuse Master component** is probably the easiest way to have singleton behaviour in a JBoss Fuse environment. Internally it relies on Apache ZooKeeper's ephemeral nodes which exist as long as there is a client session, and get deleted as soon as the session ends. As you may have guessed, the first Camel route (actually the consumer in the route) that starts up initiates a session in the ZooKeeper server and creates an ephemeral node to become the master. All other routes from the same cluster become slaves and start waiting for the ephemeral node to be released. This is how this component wraps the message consumer, and makes sure there is only one active consumer instance in the whole cluster, ensuring a master/slave failover behaviour.
5. **The database as a lock:** Christian Schneider has demonstrated how to achieve "Standby failover for Apache Camel routes" using Camel RoutePolicy-based implementation and a database as a lock.
   All of the component-based mechanisms still rely on a central system to provide

the distributed lock functionality. The difference to container-based singletons is that component-based singletons expect `CamelContext` to be started, and it is the component or the `RoutePolicy` that controls whether the route or the consumer is started or not in a singleton mode. So the handling of the singleton logic is carried out as part of the route itself.

Sometimes, the reason for having singleton services in a cluster is to ensure message ordering (which cannot be guaranteed when Competing Consumers Pattern is applied with multiple active services). If this is the case, you might not need singleton services, as Apache ActiveMQ offers such a capability called exclusive consumers *[ACTIVEMQ EC]*. ActiveMQ's exclusive consumer can choose a single message consumer as a receiver of all the messages in a queue to ensure ordering. If this consumer fails, the broker will fail over and choose another consumer. This feature in effect makes sure that there is singleton message consumer for a queue that receives the messages, regardless of the actual active consumers. Awesome, is it not?

Here is a summary of all the singleton mechanisms listed in the chapter:

Mechanism	Central Lock	Notes
Karaf Instance	Relational database or file system	Container level failover
JBoss HA Singleton	JGroups leader election	Used in JBoss EAP
Kubernetes	etcd (a distributed key-value store)	Can have only active node(s)
Camel ZooKeeper Component	ZooKeeper with sequence and ephemeral znodes	Can be configured to have more than one master
Camel JGroups Component	JGroups leader election	Not widely used
JBoss Fuse Master Component	ZooKeeper ephemeral znodes	The preferred method in JBoss Fuse stack
Database Route Policy	Relational database	A very simple implementation
ActiveMQ Exclusive Consumers	ActiveMQ instance	Limited only to message consumers

More Information

[JBOSS] *Implement an HA Singleton by JBoss*
[KUBERNETES] *Kubernetes by Google*
[ACTIVEMQ EC] *Exclusive Consumers - Apache ActiveMQ*

17. Load Levelling Pattern

This is also known as the Decoupled Invocation Pattern *[SOA PATTERNS]*.

Intent

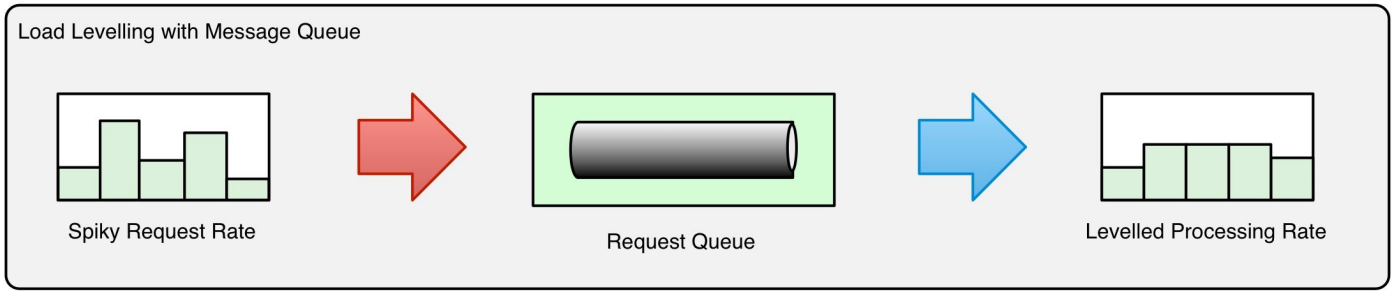
This allows the handling of peak loads and slow running tasks by introducing temporal decoupling between consumers and producers using message queues.

Context and Problem

Depending on the nature of an application, the request load pattern will vary. Unless the application is a batch-based system where the processing flows are triggered on regular intervals with a predictable load, the load on the application will not be evenly distributed. Regardless of whether the traffic pattern is publicity-driven, market-driven, or seasonal, there are certain periods, such as early evening hours, weekends, ends of the quarters, or the holiday season, where the load on the system will reach its peak. One approach to cope with this instant burst of load and spike in the resource utilization is to add more hardware to cover the peak loads. But this is not a cost-effective approach as the resources are not utilized during the rest of the time when the load is not at its peak.

Forces and Solution

A better approach to cope with such unpredictable peak loads is through load levelling with queues *[MSDN QBLLP]*. With this approach a queue is introduced to store requests and decouple the execution from the request submission.



Load Levelling with queues

The queue splits the processing flow into two separate flows: the first that handles and validates requests, and puts them in a queue, and the second that receives requests from the queue and does the actual processing. This way, whenever a new request is received by an endpoint, it does not have to be executed fully to return a result. Instead, the request is stored to a queue and acknowledgement is returned to the requestor that their request has been received successfully. The aim is to keep the processing flow as light and fast as possible, so all requests during a peak load period can still be handled and stored in the queue. Then requests can be processed from the queue in a throughput that is based on the

processing speed, independent from the request rate. Notice that if a processing flow depends on other services, then the throughput of your processing flow will depend on the external services throughput, but this will not affect the request handler flow throughput as the handler is delivering messages to the queue and not to external services.
Using queues as part of a processing pipeline has a number of advantages:

- **Deferred processing and temporal decoupling:** the sender and the receiver do not have to be running at the same time. If a receiver is not running or catching up with messages, new messages will be delayed until off-peak hours.
- **Load balancing and load levelling:** message queues can help distribute the load across multiple servers to improve throughput using the Competing Consumers Pattern *[MSDN CCP]*. Message queues can also act as a buffer to handle a sudden burst of activity by senders through the Load Levelling Pattern.
- **Reliable and resilient messaging:** message brokers can act in a transactional context and prevent messages from being lost.
- **Other interesting capabilities:** priority queues can reorder messages based on priority. Message expiration can limit the lifetime of a message. Message scheduling can temporarily postpone the processing of a message.

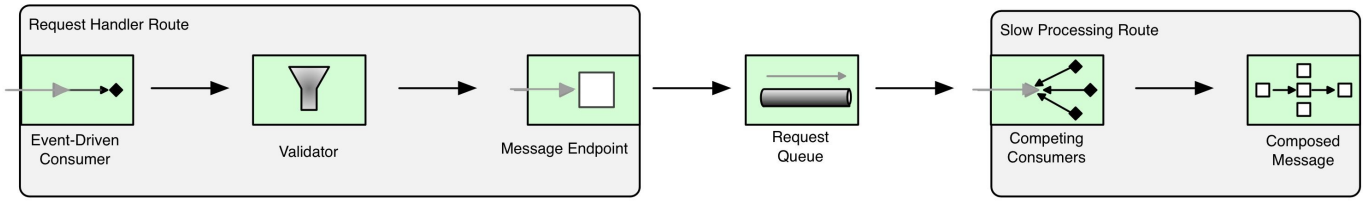
There are also some downsides of using queues:

- **Message ordering** may not be guaranteed. Even if the queueing mechanism offers FIFO order, when competing consumers are used, the message order cannot be guaranteed.
- **The Request-Reply Pattern** requires an additional mechanism to send a response. Usually this is through another response queue and correlation identifier in every response message.

Mechanics

Once a messaging system is in place to provide a queue, implementing this pattern in Camel is fairly straightforward. The few things to keep in mind for both parts of the flow are as follows:

- **Request handling route:** the aim of this flow is to handle requests as fast as possible in order to cope with peak loads. As such, the implementation should be as light as possible, preferably only with some form of validation. If there is a way to validate the request and reject invalid ones, this will give faster feedback to the service consumer and reduce the load on the queue with invalid messages.
  Another important consideration for this flow is that it is a request-response interaction. The response, with the acknowledgement of the request receipt, should be returned after the message has been delivered successfully to the broker. If in-memory queues or thread pools such as SEDA and Threads DSL are used as part of the routing process, the request might not have been persisted to the broker before returning a response to the service consumer; so be careful.



Load Levelling example with EIPs

- **Request processing route:** this part of the flow contains the business logic of the service. Since this is decoupled from the request handling flow, it can scale independently, or catch up with the message backlog slowly.
  This pattern is suited for services that are subject to overloading and do not require low latency, request-response style interactions.

More Information

[SOA PATTERNS] *Decoupled Invocation Pattern - SOA Patterns by Arnon Rotem-Gal-Oz*
[MSDN QBLLP] *Queue-Based Load Leveling Pattern in MSDN*
[MSDN CCP] *Competing Consumers Pattern in MSDN*

18. Parallel Pipeline Pattern

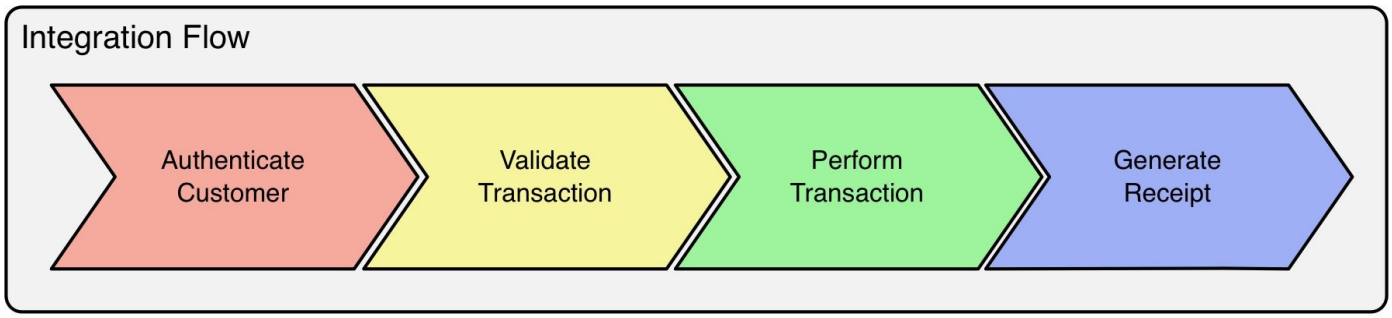
This is also known as the Pipeline Pattern *[MSDN PPL]*.

Intent

This pattern allows concurrent execution of the steps of a multistep process by maintaining the message order.

Context and Problem

The Free Performance Lunch *[FREE LUNCH]* has been over for more than ten years now. The CPU clock speed has stopped increasing in favour of a number of cores. Software that is written for sequential execution is not getting faster on newer CPUs. To gain an advantage of the increasing number of cores, the software has to be built for parallel processing. To illustrate the problem, let's have a look at the following banking example with an ATM processing flow that consists of four steps:

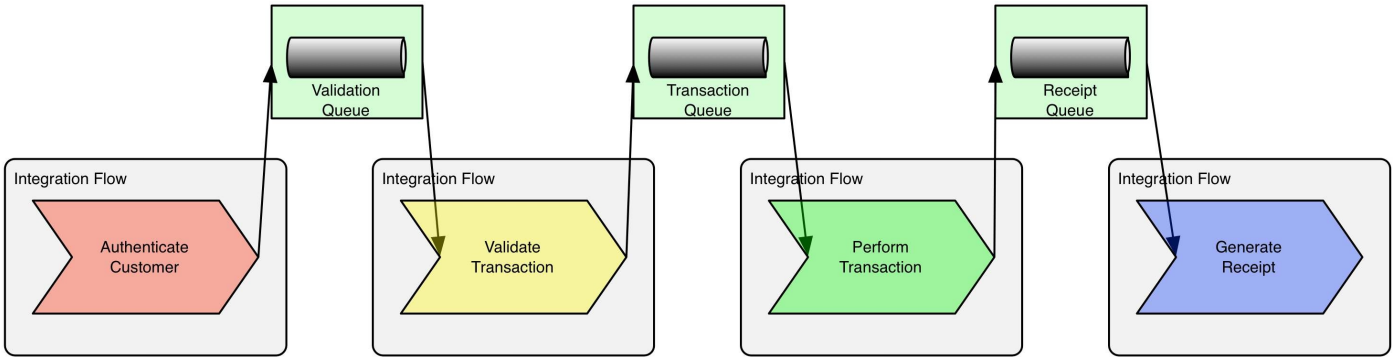


A flow with synchronous sequential processing steps - Pipeline

The steps are tightly coupled together and executed sequentially one after another. The processing thread must execute the complete flow before handling the next request, regardless of whether a step is CPU-intensive, IO-intensive (and the thread is waiting), requires a connection to an external system, requires error handling or retries, etc. Scaling such a service is challenging, let alone testing and running it.

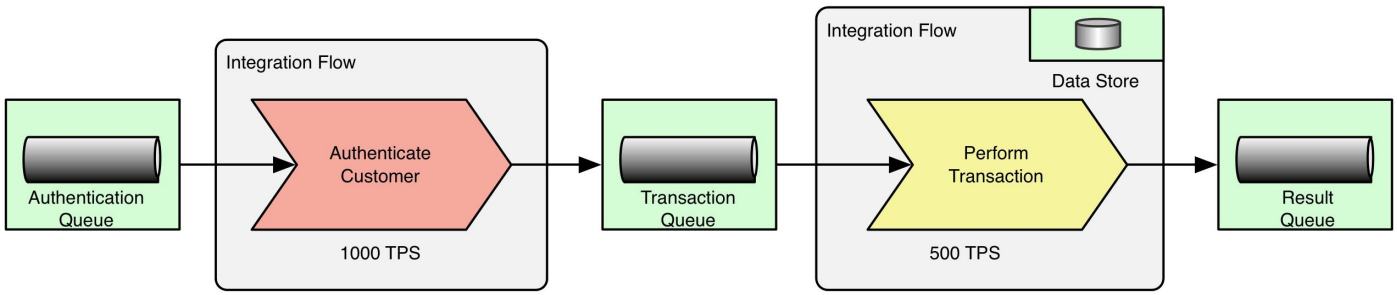
Forces and Solution

The main concept of the Parallel Pipeline Pattern *[SOA PATTERNS]* is to divide a processing flow into a series of steps, some of which can run in parallel either locally or remotely. The pattern is similar to assembly lines in a factory, where each item is processed in stages and passed from one stage to another. Each step from the pipeline uses queues to receive and pass around messages which act as buffers with load balancing capabilities and preserve the order.



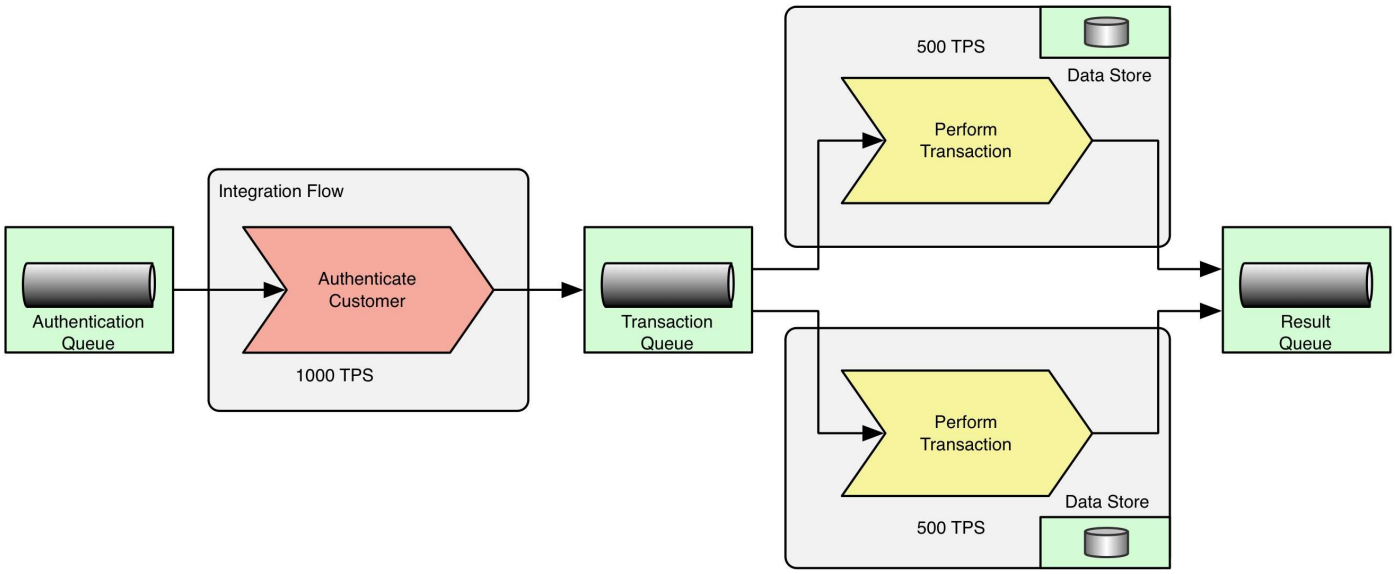
Parallel Pipeline example

You can partition an application process into discrete steps based on technical or business considerations. One of the main advantages of this pattern is that it allows the scaling of slow steps by distributing them to multiple nodes. If let's say we had the following capacity indicators for each of the stages of the pipeline, we could easily identify where the bottleneck is:



Parallel Pipeline with bottleneck in the database

The well-balanced flow has no bottlenecks *[PIPELINES]*. To balance the above flow and increase the overall capacity, we can distribute the slow state into two flows running in parallel.



Balanced Parallel Pipeline

This is a very simplified view. In a real project you have to make sure that individual flows are allowed to run in parallel. For example, the flow has to be stateless (or with a distributed state), and does not require a strict message order. If there are two instances of a flow using Competing Consumers Pattern, the message order cannot be guaranteed. Quite commonly, the processing flow is not the bottleneck, but other systems and transactional resources that the flow is interacting with are. In our example flow, it would be the database instance that would have to be sharded. Once the database is distributed through some sharding key, then we can use a content-based router pattern on the producer side, or message filters on the consumer side, to direct the flow to the right database shard.

Mechanics

This is another pattern that does not map to a specific Camel feature, but to a specific processing flow structure. The two main tasks involved in implementing this pattern are: partitioning the flow into subflows and connecting these partitions.

Partitioning a Flow

Breaking an existing complex flow into subflows, or creating a multistep pipeline from scratch, is not easy to get right the first time. If a business process is partitioned into too many fine-grained steps, it will become difficult to manage and will add too much latency. If the steps are too coarse-grained, there will not be much benefit from applying this pattern. Finding the right balance may require a little bit of experimentation. One approach for partitioning is to follow the business needs and partition the flow based on the boundaries of the business process steps. This is what is demonstrated in the previous diagrams. Another approach is to partition based on technical considerations such as: the transaction boundary, the process bottleneck, unstable parts of the flow, dependency on a problematic resource/service, etc. Such a partitioning is fine as long as the intermediate queues and subflows remain internal and do not turn into public interfaces. Separating part of the flow based on technical considerations will allow you to solve the technical difficulty in isolation, for example, by horizontal or vertical scaling, by adding extra monitoring, etc. Also consuming and producing messages through transactional boundaries will ensure that there are no messages lost in any subflow.

Connecting Subflows

This pattern relies heavily on messaging concepts such as the Fire-and-Forget style of interactions, transactional support to prevent message loss, Competing Consumers pattern for scalability and high availability, etc. As such, using JMS and ActiveMQ is the natural choice for connecting the subflows. If a Request-Response style of interaction is needed, ActiveMQ's Camel connectors (or you can say Camel's ActiveMQ connectors) makes this easy too. When the message exchange pattern is `IN_OUT`, the ActiveMQ producer will create a temporary queue and wait for a response on this queue behind the scenes. The downside of using a message broker for this pattern is that it will add extra latency due to networking and message marshalling/unmarshalling.
An alternative for using JMS and inter-process communication is the use of in-memory queues to connect the subflows. This can be done with an embedded broker instance or other components such as SEDA and VM. Using an in-memory messaging system has a serious downside though: scalability will be limited to only one JVM.

More Information

[MSDN PPL] *Pipelines in MSDN*
[FREE LUNCH] *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software By Herb Sutter*
[SOA PATTERNS] *Parallel Pipeline Pattern - SOA Patterns by Arnon Rotem-Gal-Oz*
[PIPELINES] *Software Pipelines and SOA by Cory Isaacson*

19. Bulkhead Pattern

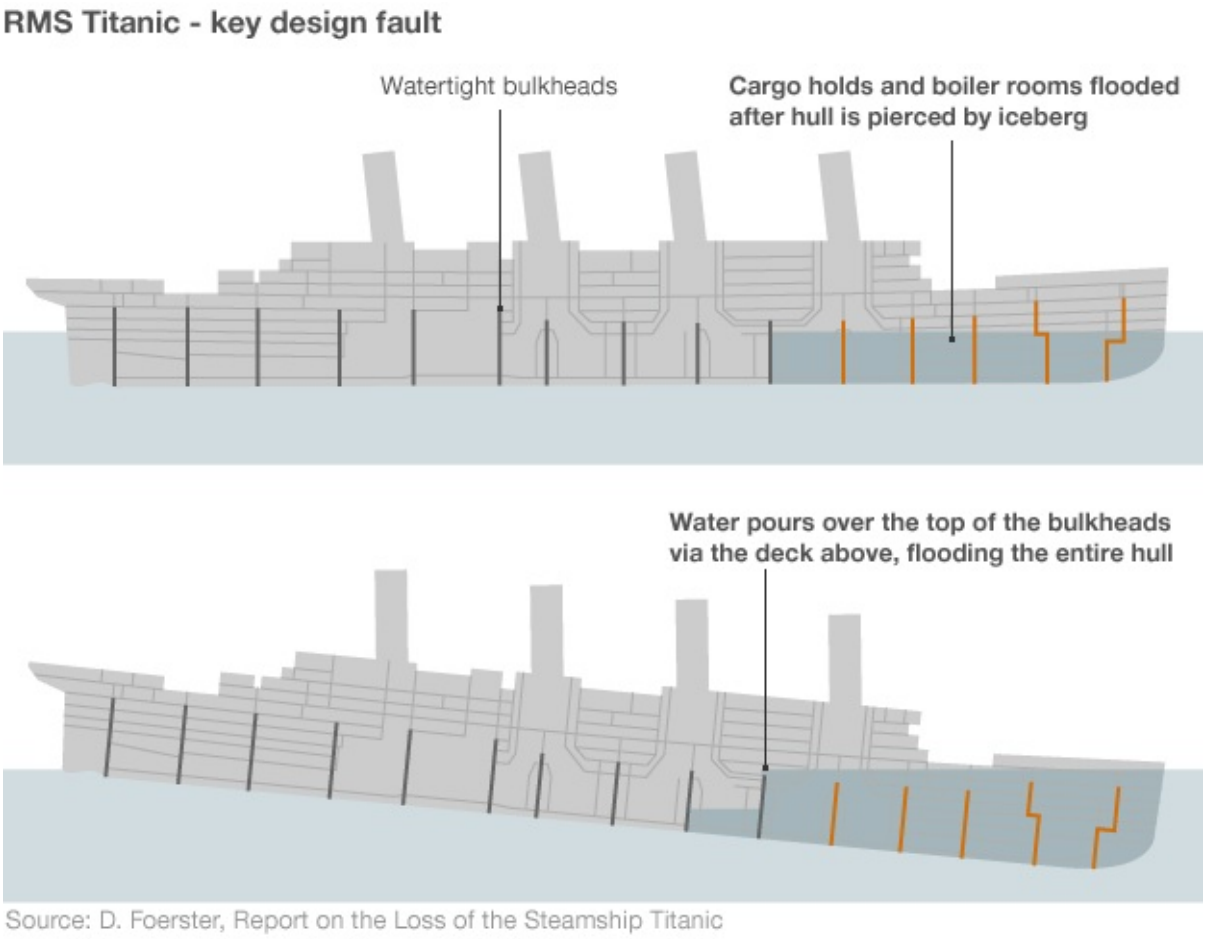
This is also known as the Failure Containment Principle and the Damage Control Principle.

Intent

This enforces resource partitioning and damage containment in order to preserve partial functionality in the case of a failure.

Context and Problem

The Titanic disaster has been well studied over the years and there are many lessons we can learn from it in the IT industry. Among the many reasons why it sank, we can find design flaws (watertight compartments did not reach high enough to give more living space in first class), implementation/construction faults (the three million rivets used to hold different parts of the Titanic together were found to be made from substandard quality iron, and they were badly impacted from the collision with the iceberg), and operational failures (the iceberg notice was given too late and the ship was travelling too fast to react to any warning). We can identify similar issues in software projects today too, but luckily they have not cost so many human lives so far, and as a consequence we learn more slowly than other industries. But that does not mean software failures have no consequences; there are plenty of examples (such as the Therac-25 machine *[THERAC-25]*) that caused human death because of software defects.

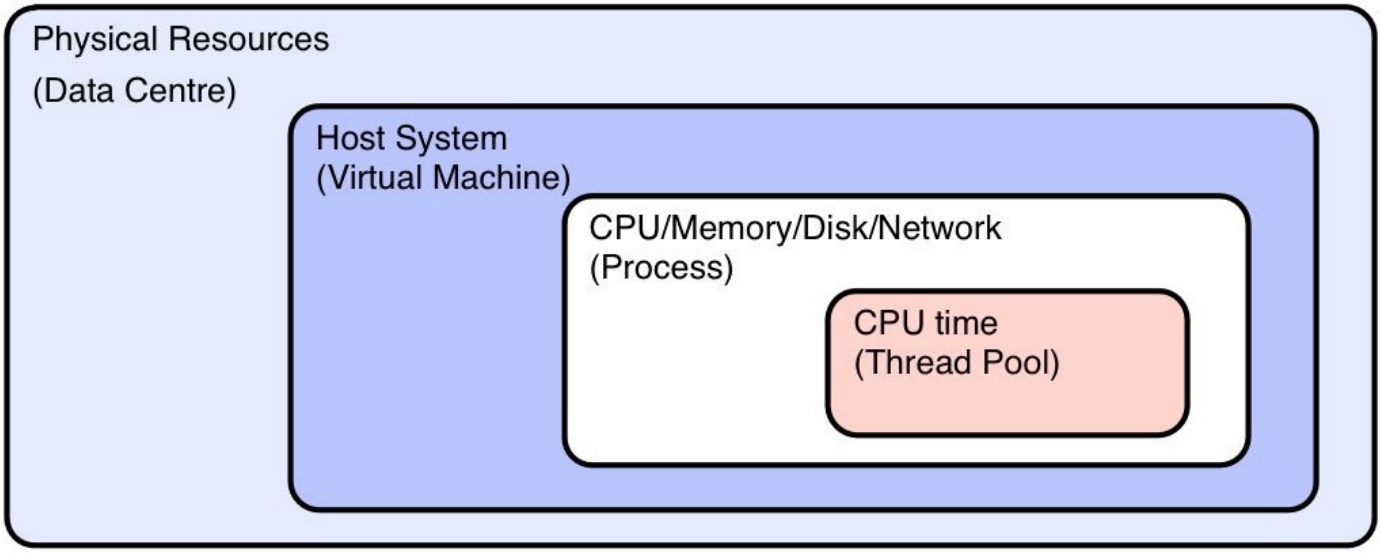


Titanic disaster and bulkheads

From an architectural point of view, the ship was designed to cope with four compartments being flooded, but the bulkheads that isolated the compartments did not reach to the deck above and weren't watertight. As a result the water spread to more compartments and sunk the ship. Similar cascading failures exist in software systems too. A failure in one component can exhaust all available resources and it can spread to other components until the whole system is down. Avoiding similar failure scenarios requires resource partitioning and capacity isolation at all levels, starting from data centres going down to individual thread pools.

Forces and Solution

The Bulkhead Pattern *[RELEASE IT]* in software systems works by the same principle as in ships. By partitioning a system into separate components and isolating the resources, failures cannot cascade and bring the whole system down. This pattern enforces the damage containment principle and improves the system resilience. Implementing the Bulkhead Pattern can be done at many different granularity levels depending on the type of faults you want to protect the system from.



Possible bulkhead levels

- The most common way to apply the Bulkhead Pattern is through **physical redundancy**. In this age of cloud computing and virtual machines, the only way to make sure that two hosts are on separate hardware (compute, storage, or networking) is by making sure that they are on two separate data centres (for example, AWS Availability Zones). Having this separation hardware failure in one processing unit cannot affect the other.
- Virtualization is still the most common way for resource isolation and scalability. So the next level for applying the Bulkhead Pattern is to use **redundant hosts** (Virtual Machines).
- The next level of the Bulkhead Pattern is having **isolated processes** on the hosts. The processes can be CPU-bound with technologies such as cGroups or Docker containers. It is also possible to dedicate a certain amount of memory, limit the disk size, and put constraints on other resources to each process and prevent a resource monopoly and contention.
- At application level, the Bulkhead Pattern can be applied to ensure **thread pool isolation**. So rather than having a single thread pool for all tasks which can be exhausted, using separate thread pools for critical tasks can ensure thread leakage containment. Threads are the most granular software primitives that map to hardware resources (CPU time). Ensuring isolated thread pools for critical parts of the application is essential.
  Each of these isolation boundaries are addressing different concerns and mitigating a different set of risks. Thread pool isolation will ensure partial functionality and graceful degradation of different parts of the application. For example, it can make sure that admin functionality is still working, or the request handling endpoint is still functioning, while the database connection pool is exhausted. Having multiple service instances as separate processes will ensure that a hung process does not bring down all of these instances. Using a separate host will provide further capacity isolation and easier scalability. And physical redundancy will allow disaster recovery or transparent failover as a last resort.

Mechanics

The Bulkhead Pattern is one implementation of the failure containment principle. Applying it requires the use of many other patterns and principles at different levels. Let's see what should be considered to implement this pattern when looking from an application development point of view.

Hardware redundancy

There is not much to consider for this category. The Java platform with its "Write once, run anywhere (WORA)" promise makes sure the applications are behaving the same way even on different hardware. But still, keep in mind that there might be some differences if the underlying hardware architecture is different. For example, a 32-bit architecture will allow only a 2GB effective heap, whereas a 64-bit one will consume more memory and hit the Java garbage collector harder. Also having a distributed system deployed on hardware that is geographically dispersed will increase the chances of potential issues caused by any of the Eight Distributed Computing Fallacies *[FALLACIES]*.

Host redundancy

This category is also mostly taken care of by the Java platform's ability to run on different operating systems, but there are a few more things to be cautious about. Different hosts may run different operating systems, and as such, when writing an application, do not make assumptions about the line endings (Windows will use two symbols, whereas Unix will use only one), file separators, disk structures, file locking capabilities, or command line calls. If you have made any assumptions about these, or about the Java version, or assumed localhost to be the only host name when implementing a service, it will limit the available hosts to deploy a service.

Process isolation

This principle relies on having multiple instances of a service as separate processes (whether that is on the same host or not is irrelevant). And implementing the Service Instance Pattern requires many considerations, which are described in the chapter with the same name. For example, if you have used an in-process communication mechanism, such as Camel VM or Direct-VM components, for service communication, it will not be possible to split these two services into separate processes without refactoring both services.
Using multiple processes for the same service is also popular in messaging. There is a scaling technique in ActiveMQ called client side partitioning. With this technique, rather than having one message broker instance for all services, the message queues are distributed into multiple message broker instances. And the messaging clients are configured to connect to the right broker instances - hence the name of the pattern. Then you can locate queues for critical services in a broker instance with more redundancy, or allocate a separate broker instance for important customers. This way, if the broker process dies, it will only affect the services using that broker instance.

Thread pool isolation

At this most granular level, applying the Bulkhead Pattern is a part of the individual service design and development. Assuming that failures will happen, the aim of this

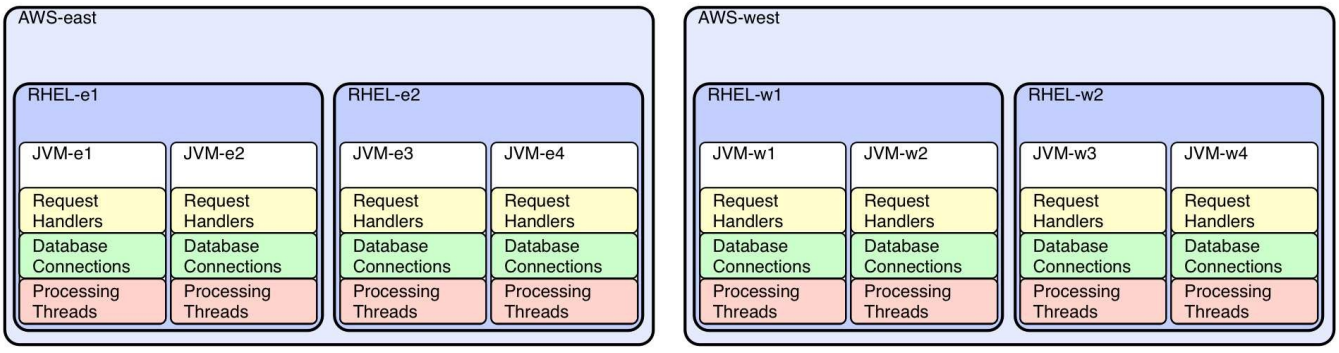
pattern is to try to isolate and control failures in part of the application from spreading and affecting the whole application. To achieve this, you can use whatever patterns, principles, and practices that are available. Let's see a few that are very common:

- **Parallel Pipeline Pattern:** this allows the breaking of a processing flow into independent subflows. When the interaction style is Fire-and-Forget where no response is expected, a failure in one subflow will not affect the rest of the pipeline. The other subflows can still continue accepting and processing incoming requests.
- **Load Levelling Pattern:** using queues in general for temporal decoupling will help isolate different parts of the application.
- **Other related patterns:** using the following stability patterns does not contribute to the implementation of bulkheads, but these patterns will improve the stability of the system, making it less likely to fail and need bulkheads. These are patterns such as throttling, circuit breakers, timeouts, even CQRS.
- **Thread pool segregation:** this is the most popular way to implement bulkheads at application level. Let's briefly cover some key areas to configure in Camel with regards to thread segregation: Camel uses multi-threading to perform concurrent tasks at many different levels. There are components, EIPs, and error handlers that offer parallel processing features. Usually whenever a parallelism is offered by Camel, it is also possible to tune and customize it with a custom thread pool. By default Camel thread pools are based on the implementations provided by the `java.util.concurrent` package and described through something called Camel Thread Pool Profiles. Profiles are a declarative way of describing the various thread pool configurations and behaviours (such as `poolSize, maxPoolSize, keepAliveTime, maxQueueSize, and rejectedPolicy`) through configuration. By default there is a profile (called the default thread pool profile) used as a base for creating thread pools. For the record, as of this writing it has the following default values: `poolSize=10, maxPoolSize=20, keepAliveTime=60s, maxQueueSize=1000, rejectedPolicy=CallerRuns`. So if you do not customize the default profile and do not use a different profile, all thread pools will have the aforementioned configuration, which might not be ideal for your application. There are some exceptions - internal thread pools created for aggregating messages in Recipient List, Splitter, and Multicast ignore profiles and create unbounded cached thread pools to ensure the aggregate on-the-fly task will always have assigned a thread as soon as it is submitted. All other thread pools in Camel will be based on the default thread pool profile.
- **Consumers:** usually polling consumers (as opposed to event-driven consumers) use a separate thread to perform the polling. Some of the components (such as JMS, SEDA, Kestrel, Hazelcast, and Scheduler) provide `concurrentConsumers` the option to specify the number of threads used for polling. So before using a component, do check whether the component offers any parallelism, and if so whether the default thread pool configurations are feasible.
- **Single threaded components:** some components such as the File consumer are single-threaded. If the only worker thread gets blocked, the whole route will stop functioning. To make it a little more resilient, and to scale such a consumer, you can use the Threads DSL in Camel *[CAMEL THREADS]* to process files in parallel.

Threads DSL can turn a synchronous route into an asynchronous one by processing the messages using new threads from the Threads DSL point forwards.

- **Custom components:** when creating a custom component that uses threads, rather than creating a custom thread pool manually, use `ExecutorServiceManager` from Camel. This class will make sure that the thread profiles are honoured and also do some resource management for you. Similar concepts also exist in other projects like JBoss WildFly, where the whole thread management is delegated to the application server.
- **Multi threaded EIPs:** the following EIPs support parallel processing and allow custom thread pools. When an EIP is configured for parallel processing, it creates a thread pool based on the default thread pool profile (so bulkhead is already applied), but make sure the default configuration makes sense for this use case. The EIPs are: Delayer, Multicast, Recipient List, Splitter, Threads, Throttler, Wire Tap, Polling Consumer, ProducerTemplate, and OnCompletion.
- **Error handler:** the error handler in Camel can also perform asynchronous tasks, like delayed retries. But if no thread pool is provided, all the error handlers in the `CamelContext` will share the same thread pool. So it is a potential area where asynchronous redelivery in one route consumes the threads and blocks the redelivery thread in another route, as the thread pool is shared for the whole `CamelContext`. Make sure to either tune the global `ErrorHandlerExecutorService` or specify separate thread pools for some error handlers.
- **Camel Hystrix Component:** Hystrix *[HYSTRIX]* is a fault tolerance library designed to stop cascading failures and enable resilience in distributed systems. As of this writing there is no official Camel connector for Hystrix yet, but it is a work in progress and should be available soon (I will commit my implementation in a couple of days - as soon as I finish working on this book). The Hystrix component can also act as a bulkhead by providing thread and semaphore isolation.

This is an example application with the Bulkhead Pattern applied that is fully symmetrical. In the real world, you should analyse the loss of a capability and its impact to the business and let this drive the isolation boundary.



An example with bulkheads applied at all possible levels

If the solution is technically feasible to implement, then protect the capability with a financially achievable redundancy. In the simplest form, this means dedicating hardware and ensuring redundancy for the mission-critical services. In a more complex scenario, it requires various trade- offs as described in the Service Consolidation chapter.

More Information

[THERAC-25] Therac-25 machine by Wikipedia
[RELEASE IT] Bulkheads - Release It! by Michael T. Nygard
[FALLACIES] Fallacies of Distributed Computing Explained by Arnon Rotem-Gal-Oz
[CAMEL THREADS] Threads DSL - Apache Camel
[HYSTRIX] Hystrix by Netflix

20. Service Consolidation Pattern

This is also known as Service-to-Host Mapping *[MICROSERVICES]*, and the Service Partitioning/Segregation Pattern *[MSDN CPG] [MSDN CPG]*.

Intent

This provides guidelines for the grouping of services together, or the isolating of them from one another, for deployment purposes driven by constraints.

Context and Problem

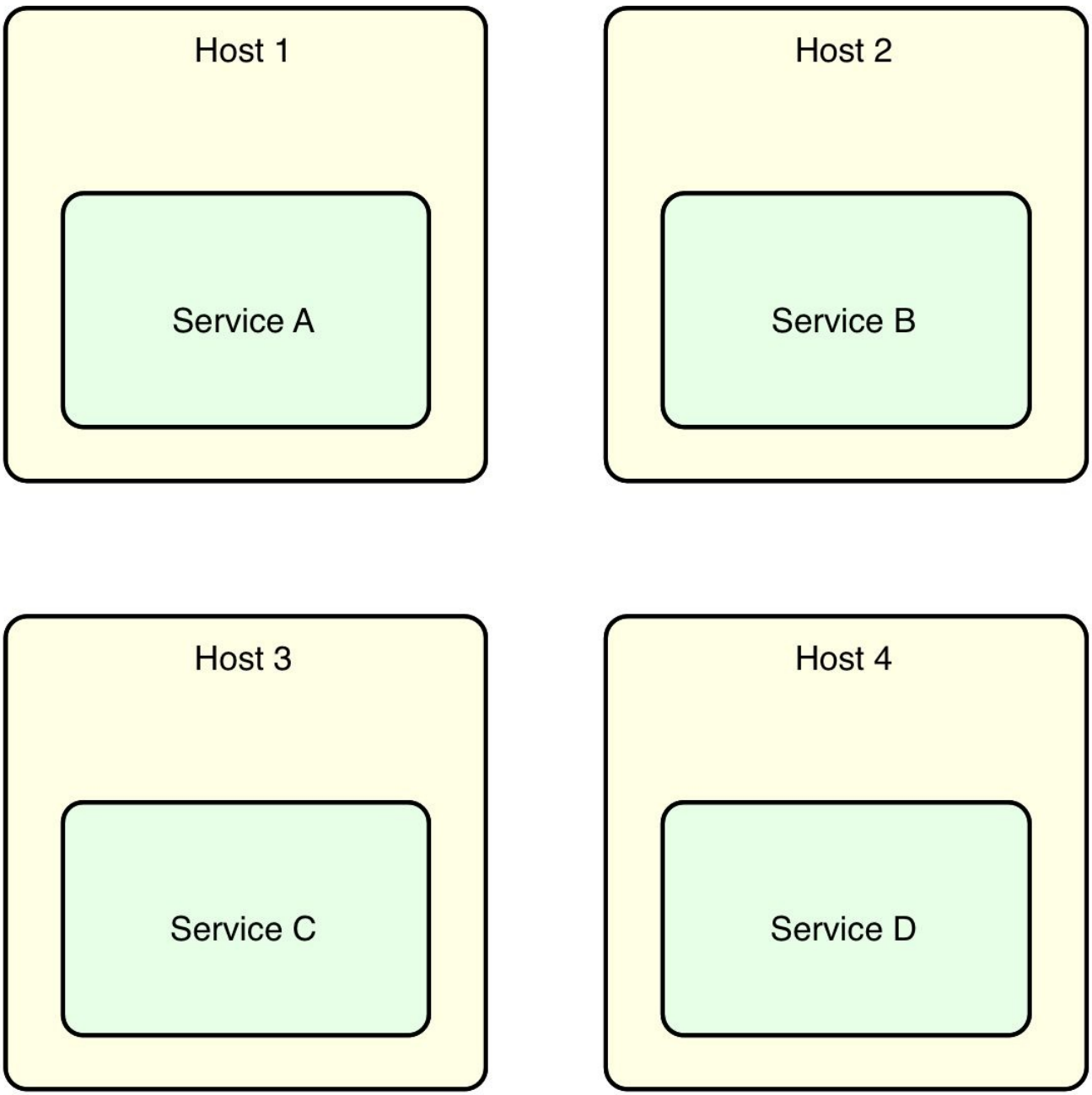
Let’s assume we have written our service and it is ready to be deployed. The service may be composed of sub-modules such as contracts, routes, domain model, persistence layer, configurations, and other layers needed for it to function as a whole. It can be a very granular microservice with one operation or a monolithic application with a large number of endpoints and operations. All of these artefacts represent the static build time structure of the application and not the runtime view. At runtime this service may be deployed together with other services in one container (JVM), or in an isolated container with other containers on the same host (VM), or in a totally isolated host. In addition the same service may be deployed differently in different environments: from heavily consolidated in dev environments, to fully segregated in a production environment. In certain situations deployment of a service may vary temporarily (for example, scaling up or down a service based on the load), or be based on tenancy considerations (a different number of instances deployed for different tenants). All of these deployment models require the application to be written in a way that allows deployment time flexibility and some analyses of the runtime constraints.

Forces and Solution

There are a few well-known deployment models with clear benefits and drawbacks, but finding the right balance for your situation will require the analysing of the application’s runtime behaviour and other constraints. First we will look at the different deployment options and then the driving forces behind each of them.

Single Service per Host

With the term host we will be referring to an OS regardless of whether it is physical or virtual, on the cloud or on-premise. With this model there is only one service running per host. This is the preferred model for the microservices architecture and there are many benefits of using this.



Single service per host

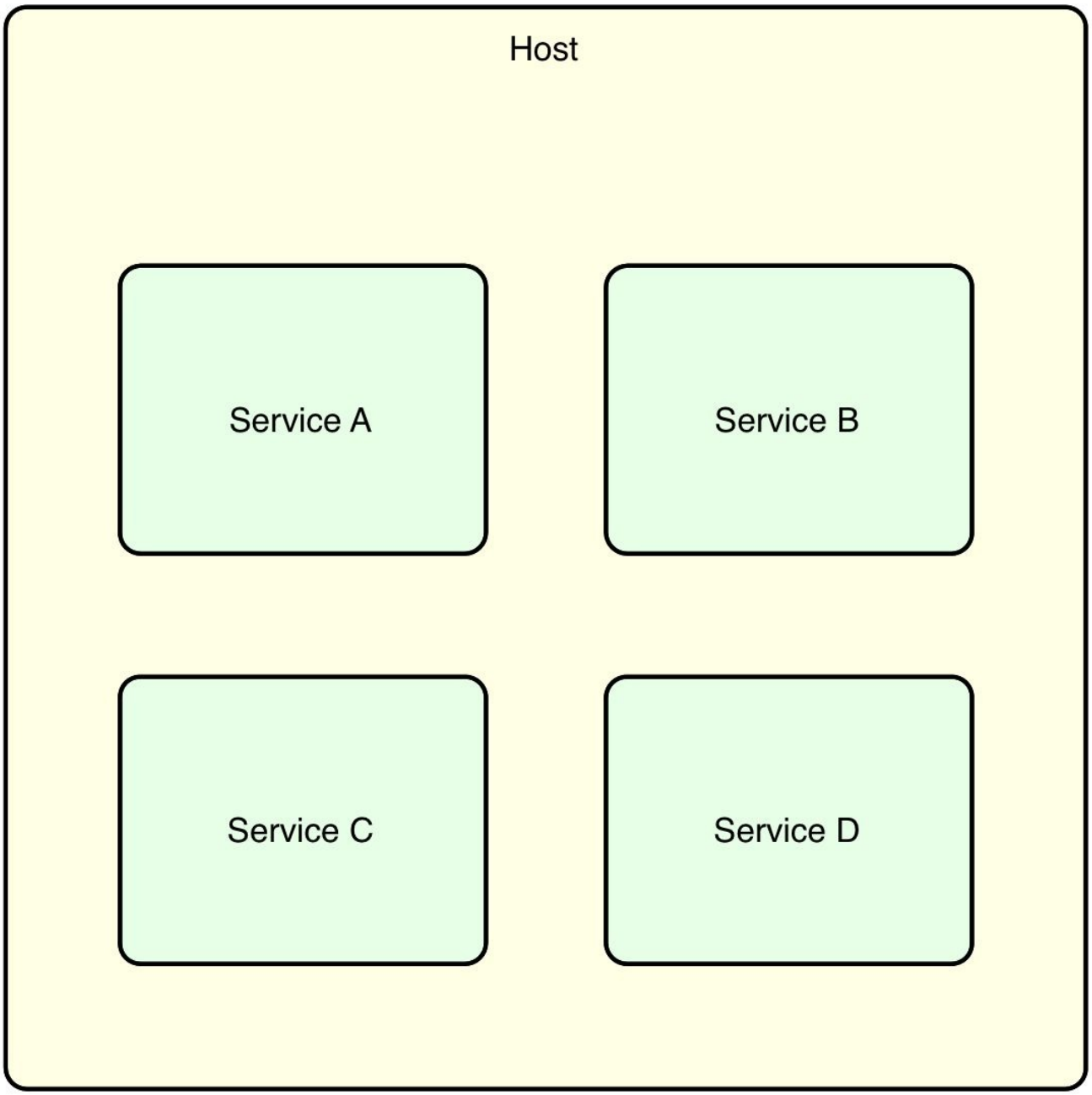
Thanks to the service isolation and one-to-one mapping between the service and the host, there are great monitoring and manageability benefits. If there are increased CPU cycles or there is a high memory utilization, these can be assumed to be caused by the only service on the host. From a management and maintenance point of view, it becomes easier to prepare a host with all the prerequisites for a single service deployment and future releases. This model also simplifies creating auto scaling and fault-tolerant systems. The scaling up and down of a service can be achieved easily with services such as AWS ELB and auto scaling, which work on a per host basis.

The main drawback of this model is its high cost due to a small resource utilization. Usually the dynamic service resource utilization profile will not match up to the static host resources and there will be lots of idle time. Managing a large number of hosts with regular patching, monitoring, and security hardening also requires a significant effort.

Also worth mentioning is the fact that when the service is baked into and deployed as a virtual image to the host, the deployment time is quite high due to a full host deployment.

Multiple Services per Host

The other extreme deployment model is to have multiple services on a single host.



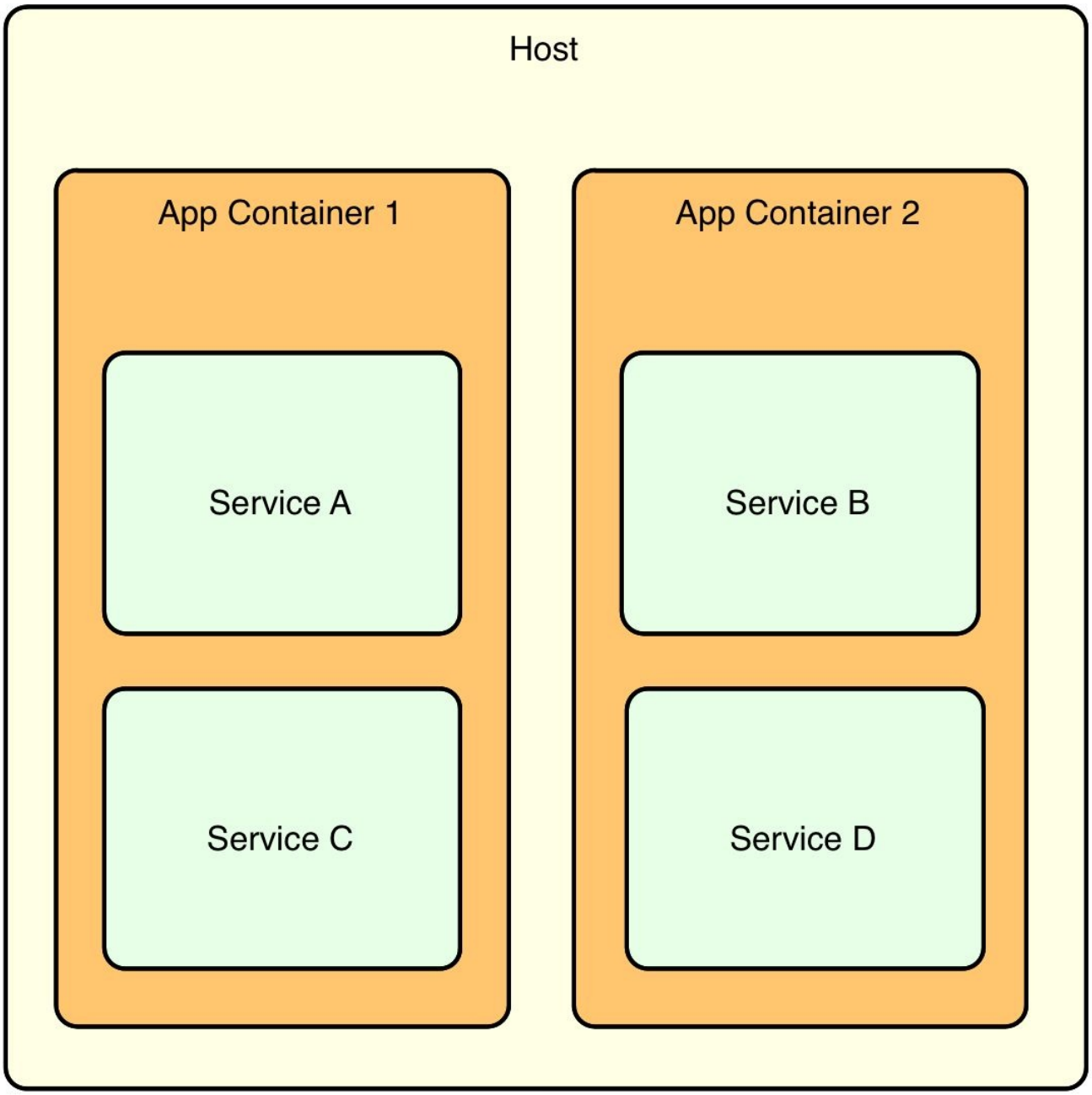
Multiple services per host

When multiple services are packed on to a single host, the resource utilization increases and brings down the cost. Since the number of services is not proportionate to the number of hosts, increasing the number of services does not increase the number of hosts directly. This simplifies the life of the operations team as they have to manage and provision a smaller number of hosts. The main drawback of this model is caused by the fact that the services are sharing the host system which introduces a runtime coupling. Preparing a host system that satisfies the prerequisites of all services and keeping that up to date for future services becomes challenging. Deploying a new service to a host with existing services also becomes tricky as it can affect these existing services in an unexpected way. Scaling a service for performance or high availability becomes harder too as the service is sharing the host with other services. Vertical scaling will affect other services as well, and horizontal scaling will require a different deployment model for the target service. Monitoring also has to be done at a more granular level, as the host-level metrics will

show the aggregate of all services running on the host. Overall, this model offers cost saving from a resource utilization point of view, but significantly increases the maintenance overhead.

Application Container(s) per Host.

When you have one service per process (JVM), whether it is a standalone application (Java `main()`, for example), or an application container such as OSGI-based Apache Karaf, or a full-blown Java EE application server (such as JBoss Wildfly), the service life cycle is mapped to the container life cycle. Similarly the service can be run with a docker container which offers greater resource limiting and isolation capabilities than a JVM. The services in the previous two deployment models can be run in any kind of process, but the difference is that there is only one process per service. In the application container(s) per host model, multiple services do share the same process as their runtime container. That is, for example, when a service is packaged as an OSGI bundle and deployed to Apache Karaf with other services, or a service is packaged as WAR and deployed to Apache Tomcat with other WAR files. Similarly to the multiple services per host model, there can be multiple application containers with multiple services deployed on a single host.



Multiple application containers (with multiple services each) per host

Consolidating multiple services on a single application container helps with the management, monitoring, and deployment of services. These containers do have dynamic deployment capabilities where a single service can be deployed, upgraded, or undeployed without affecting other services or stopping the whole container. Containers do also have improved monitoring, logging, transactional, and clustering capabilities. The main drawback comes from the fact that the service shares the same process at runtime which makes managing individual services harder. It becomes more difficult to monitor and manage the resource consumption of a service. Because of the shared process, it becomes harder to scale a service separately and sometimes one unstable service may bring the whole container down.

Consolidation Criteria

Finding the right balance between service consolidation and isolation requires the consideration of various factors and it is an ongoing process throughout the life of an application. To identify the groups and plan the physical deployment into a single host or multiple compute hosts we must determine the requirements of the individual services. Services with similar characteristics can be grouped into the same partition whether that is an application container or a host. Let's see a few categories of decision points that apply to most project deployments:

Non-functional requirements

- **Scalability and performance:** one of the main considerations for consolidating services is based on performance and throughput requirements. Services with common requirements can share the same host or be deployed on isolated hosts, making it easier to scale and satisfy performance requirements by starting multiple instances. Grouping services with conflicting requirements on the same container or host will make it difficult to benefit from infrastructure elasticity.
- **Availability and fault tolerance:** decomposing services based on availability requirements are common too. Customer-facing or business-critical services with strict SLAs can go on separate hosts making them easier to scale and improve availability. Services with lower availability requirements can share a single host.
- **Security:** services sharing the same security context can share the same container or the same host. But if there is no high degree of trust between services, for example, they belong to different tenants, you may consider splitting them into separate hosts. Considering security boundaries of the application and the attack surface of each service may require changes to the planned deployment model.

Life cycle

- **Lifetime:** an application with a short lifespan, such as a microservice for an ad campaign, should be segregated from long-lived services executing long-running processes. Deploying and undeploying short-lived processes can affect the stability of long-lived applications.
- **Release cadence:** some services have their implementations change very frequently or they require configuration changes periodically. There are application containers (such as Apache Karaf) that can deploy and undeploy services without restarting the whole container. But other containers will need to be stopped and started again, or some changes may require the restarting of the host machine (for some environmental changes). Sharing the same application container or the same host for services with different release cadence usually requires extra management overhead.

Resource profile

- **Resource utilization:** different services will have different requirements for CPU, memory, and network bandwidth. Consolidating services based on resource requirements to match the host profile is a common starting point. For example, a batch-based service can go to a less powerful host, whereas a real-time service with fluctuating demand can go to a host with a larger capacity.

- **Resource contention:** while the previous criteria tries to match service resource requirements to the host profile, the resource contention principle tries to avoid services competing for the same resource. For example, having two memory-intensive services with the same request load pattern (a request burst at the same time) on the same host may not be a good idea if the host does not have enough memory to accommodate both services at their peak load. But combining a CPU-intensive and a memory-intensive service on the same host will reduce resource contention and both services may perform better. Keep in mind that overcommitting *[OPENSTACK]* is a very common practice used in virtualized environments (such as OpenStack) and the resources visible to the host system are much fewer in reality.

Runtime dependencies

- **Environmental dependencies:** some services may have dependencies to resources on the host system. These are typically legacy systems with very specific access restrictions. The only way to access such a system could be through co-locating a service and the resource.
- **Service dependencies:** modern integration applications consist of multiple services communicating over HTTP or some kind of messaging which does not require service co-location. But sometimes to simplify deployment, minimize latency, or improve throughput, services can be collocated on the same host or even on the same container. For example, the Camel direct-vm endpoint allows services to communicate through direct in-memory calls. Similarly, other software systems such as cache or text search engines can share the same container and benefit from direct in-memory calls. Keep in mind that having embedded in-process service dependencies is difficult to scale and may require some specific clustering technology.

Maintenance

- **Management, maintenance and cost:** segregating services onto separate containers and separate hosts will require extra effort for managing, monitoring, and maintaining the hosts. Automating some of these tasks, and using an elastic infrastructure with auto scaling capabilities, may help reduce this cost.
- **Complexity:** on the other hand, consolidating multiple services on the same container and the same host will increase the management and maintenance complexity. Finding the right balance between these two conflicting characteristics is the whole point.

Mechanics

This is another pattern that is not specific to Camel and there is not much to consider from a Camel routing point of view to implement it. The two principles I followed during development that directly contribute to this pattern are the Smallest Deployment Unit and The Last Responsible Moment.

Smallest Deployment Unit

With microservices architecture becoming more and more popular, applying this principle becomes easier as there are more and more books and guidelines to help. But this was not the case when Camel was used to create a microservices-like application before all the buzz. The smallest deployment unit is usually a couple of Camel routes in a `CamelContext` bundled as a jar file. It can also be more than one `CamelContext` in a jar file. The smallest deployment unit may provide a couple of operations for a bounded context, or perhaps the read and write operations are separate deployment units following the CQRS design pattern. Ideally the deployment unit should not make any assumptions about other deployment units, whether they are in the same JVM, or the same host. The smallest deployment unit should also allow clustering (applying the Service Instance Pattern). So the smallest deployment unit should run fine as a single instance or multiple instances. If the business logic does not allow multiple service instances, the deployment unit should be developed by implementing the Singleton Service Pattern (or it should be made clear that it expects such a behaviour from the container). If the services are stateful, make sure to use some kind of distributed cache (embedded or external) so that the service can be deployed as a single instance or multiple instances, and on a single host or separate hosts without a change of the implementation. The boundaries of the smallest deployment unit may be driven by business requirements, non-functional requirements, the life cycle, the resource profile, runtime dependencies, maintenance, etc. Nowadays, the smallest deployment unit is more or less a microservice, so the principle suggests using microservices mainly.

The Last Responsible Moment

I have seen projects where the physical design and deployment model has to be created way before the requirements are clarified and the services are designed. Sometimes it is necessary to order the physical hardware in advance as it takes a long time to be provisioned. But when designing and implementing an integration application, it is much better if you apply the Last Responsible Moment *[CODING HORROR]* principle and delay making commitments about the deployment model. Instead write the service without making any assumptions about where and how the services will be deployed. Identifying the smallest deployment unit and creating isolated services which do not make assumptions about other services will give you much more flexibility during deployment time. Also keep in mind that the deployment models will change over the lifetime of the application. It is quite common to start collocating the first few services on the same JVM or the same host, and later to start segregating and consolidating these services to different hosts. For example, if you have made the assumption that two services will always be located on the same JVM and have used Camel VM and Direct-VM components, it will not be possible to separate these services on different hosts without refactoring the code base (you may need to add marshalling/unmarshalling to the route before replacing VM/Direct-VM with messaging, for example). Another issue with VM/Direct-VM communication is that it does not provide any kind of isolation; any service can access any other service in the same JVM as long as it knows the endpoint name, which might be problematic for multi-tenant deployments. One of the strong sides of Camel is that it is easy to refactor existing routes. Camel DSL is very powerful and usually changing only a few lines of code is enough to achieve the desired effect.

More Information

[MICROSERVICES] Service-to-Host Mapping - Building Microservices By Sam Newman
[MSDN CRCP] Compute Resource Consolidation Pattern in MSDN
[MSDN CPG] Compute Partitioning Guidance in MSDN
[OPENSTACK] Overcommitting in OpenStack
[CODING HORROR] The Last Responsible Moment by Jeff Atwood

Summary

This book contains twenty design patterns useful for designing Apache Camel-based applications. The following list is a quick reference with a short description for each pattern, followed by a Mind Map visualizing the patterns.

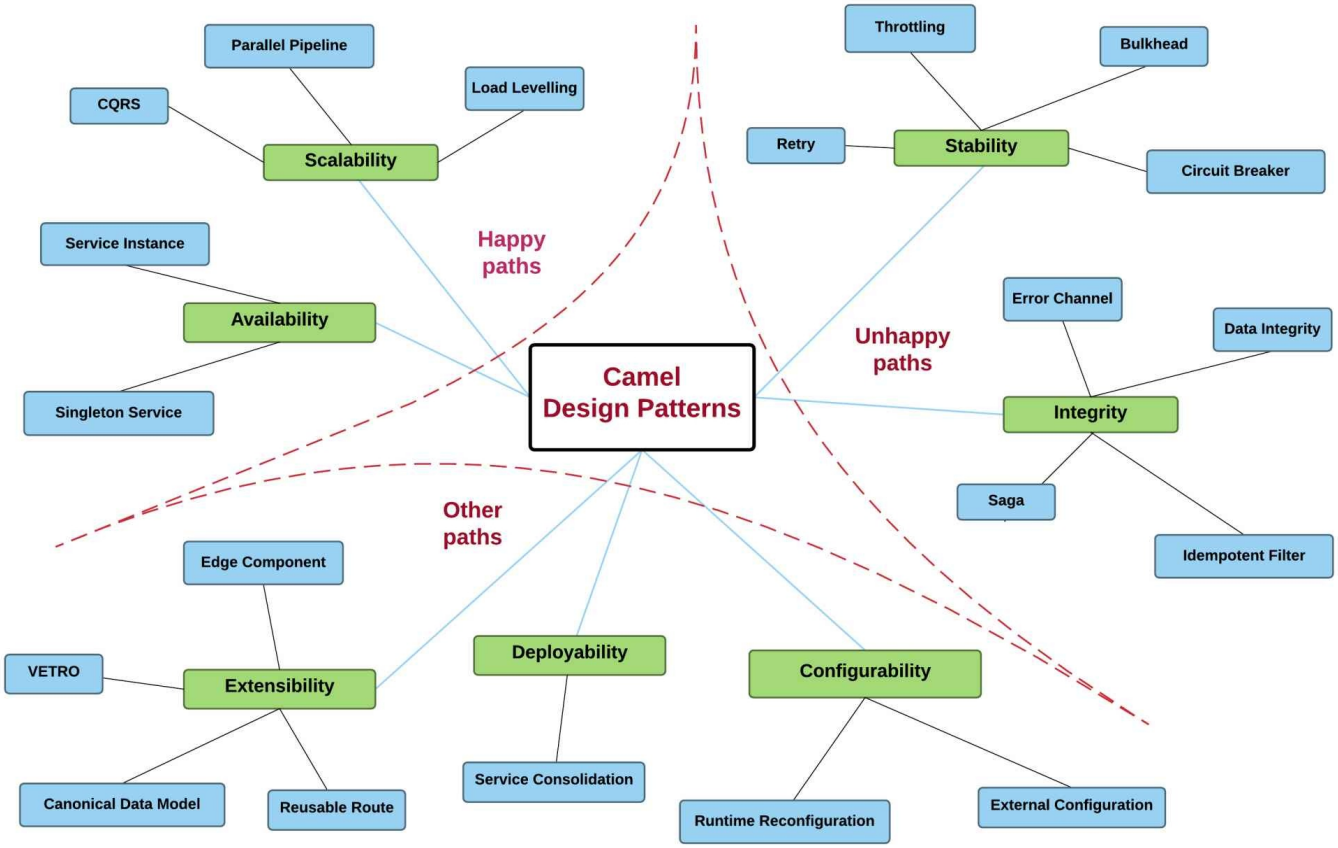
Patterns List

1. **VETRO Pattern:** Combines multiple sequential actions taken on a message into a consistent structure and well defined responsibilities.
2. **Canonical Data Model Pattern:** Allows the minimization of dependencies between applications that use different data formats through an additional level of data format indirection.
3. **Edge Component Pattern:** Encapsulates endpoint-specific details and prevents them from leaking into the business logic of an integration flow.
4. **Command Query Responsibility Segregation Pattern:** Decouples read from write operations to allow them to evolve independently.
5. **Reusable Route Pattern:** Allows an agnostic business logic to be repeatedly used in different service contexts.
6. **Runtime Reconfiguration Pattern:** Allows externalizing and runtime variability of behaviour without requiring an application redeployment.
7. **External Configuration Pattern:** Parametrizes the configuration information of an application and externalizes it from the deployment archive.
8. **Data Integrity Pattern:** Maintains the data consistency and business integrity of a system comprised of dispersed data sources in the case of a processing failure.
9. **Saga Pattern:** Removes the need for distributed transactions by ensuring that the transaction at each step of the business process has a defined compensating transaction to undo the work completed in the case of partial failures.
10. **Idempotent Filter Pattern:** Filters out duplicate messages and ensures that only unique messages are passed through.
11. **Retry Pattern:** Enables applications to handle anticipated transient failures by transparently retrying a failed operation with an expectation that it will be successful.
12. **Throttling Pattern:** Controls the throughput of a processing flow to meet the service-level agreements, and prevents the overloading of other systems.
13. **Circuit Breaker Pattern:** Improves the stability and the resilience of a system by guarding integration points from cascading failures and slow responses from external systems.
14. **Error Channel Pattern:** Set of patterns and principles used for error handling in integration applications with different conversation styles.
15. **Service Instance Pattern:** Accommodates increasing workloads by distributing the loads on multiple service instances.

16. **Singleton Service Pattern:** Makes sure that only a single instance of a service is active at a time.
17. **Load Levelling Pattern:** Allows the handling of peak loads and slow running tasks by introducing temporal decoupling between consumers and producers using message queues.
18. **Parallel Pipeline Pattern:** Allows concurrent execution of the steps of a multistep process by maintaining the message order.
19. **Bulkhead Pattern:** Enforces resource partitioning and damage containment in order to preserve partial functionality in the case of a failure.
20. **Service Consolidation Pattern:** Provides guidelines for the grouping of services together or the isolation of them from one another for a deployment purpose driven by constraints.

Patterns Mind Map

Every pattern has implications to multiple non-functional requirements (NFR), and can contribute to a variety of scenarios. The following mind map is a simplified view showing the main NFR that every pattern contributes to and the category it belongs to.



Camel Design Patterns mind map in relation to NFRs and use cases

Bibliography

[ACTIVEMQ BRP] Broker Redelivery Plugin - Apache ActiveMQ
http://activemq.apache.org/message-redelivery-and-dlq-handling.html

[ACTIVEMQ CRP] Consumer Redelivery Policy - Apache ActiveMQ
http://activemq.apache.org/redelivery-policy.html

[ACTIVEMQ EC] Exclusive Consumers - Apache ActiveMQ
http://activemq.apache.org/exclusive-consumer.html

[ACTIVEMQ PFC] Producer Flow Control - Apache ActiveMQ
http://activemq.apache.org/producer-flow-control.html

[APPLYING SAGA] Applying the Saga Pattern by Caitie McCaffrey
http://gotocon.com/chicago-2015/presentation/Applying%20the%20Saga%20Pattern

[MICROSERVICES] Service-to-Host Mapping - Building Microservices By Sam Newman
http://samnewman.io/books/building_microservices

[CAMEL DYNAMIC ROUTER] Dynamic Router EIP - Apache Camel
http://camel.apache.org/dynamic-router.html

[CAMEL FILE] File Component - Apache Camel
http://camel.apache.org/file2.html

[CAMEL MVN] Maven Archetypes - Apache Camel
http://camel.apache.org/camel-maven-archetypes.html

[CAMEL ONCOMPLETION] OnCompletion - Apache Camel
http://camel.apache.org/oncompletion.html

[CAMEL REDELIVERY] Redelivery Policy - Apache Camel
http://camel.apache.org/redeliverypolicy.html

[CAMEL ROUTE CONTEXT] Route Context - Apache Camel
http://camel.apache.org/how-do-i-import-routes-from-other-xml-files.html

[CAMEL THREADS] Threads DSL - Apache Camel
http://camel.apache.org/async.html

[CAMEL THROTTLER] Camel Throttler EIP - Apache Camel
http://camel.apache.org/throttler.html

[CAMEL THROTTLING] Route Throttling Example - Apache Camel
http://camel.apache.org/route-throttling-example.html

[CHAOS MONKEY] Chaos Monkey by Netflix
https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey

[CLOUD PATTERNS] Service Load Balancing by Erl Naserpour
http://cloudpatterns.org/design_patterns/service_load_balancing

[CODING HORROR] The Last Responsible Moment by Jeff Atwood
http://blog.codinghorror.com/the-last-responsible-moment

[FREE LUNCH] The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software By Herb Sutter
http://www.gotw.ca/publications/concurrency-ddj.htm

[CONVERSATIONS] Conversation Patterns by Gregor Hohpe
http://www.enterpriseintegrationpatterns.com/docs/ConversationsEuroPlopFinal.pdf

[EIP] Enterprise Integration Patterns by Gregor Hohpe and Bobby Woolf
http://www.enterpriseintegrationpatterns.com/patterns/messaging/toc.html

[ESB VETO] The VETO Pattern - Enterprise Service Bus by David A Chappell
http://shop.oreilly.com/product/9780596006754.do

[FALLACIES] Fallacies of Distributed Computing Explained by Arnon Rotem-Gal-Oz
http://www.rgoarchitects.com/Files/fallacies.pdf

[HYSTRIX] Hystrix by Netflix
https://github.com/Netflix/Hystrix

[JBOSS] Implement an HA Singleton by JBoss
https://access.redhat.com/documentation/en-US/JBoss_Enterprise_Application_Platform/6/html/Development_Guide/Implement_an_H

[JETTY] Limiting Load - Jetty
http://www.eclipse.org/jetty/documentation/current/limit-load.html

[JOLOKIA] Jolokia
https://jolokia.org/

[KUBERNETES] Kubernetes by Google
http://kubernetes.io

[MSDN ECSP] External Configuration Store Pattern in MSDN
https://msdn.microsoft.com/en-us/library/dn589803.aspx

[MSDN RRP] Runtime Reconfiguration Pattern in MSDN
https://msdn.microsoft.com/en-us/library/dn589785.aspx

[MSDN CTP] Compensating Transaction Pattern in MSDN
https://msdn.microsoft.com/en-us/library/dn589804.aspx

[MSDN CBP] Circuit Breaker Pattern in MSDN
https://msdn.microsoft.com/en-gb/library/dn589784.aspx

[MSDN TP] Throttling Pattern in MSDN
https://msdn.microsoft.com/en-us/library/dn589798.aspx

[MSDN CQRS] CQRS in MSDN
https://msdn.microsoft.com/en-us/library/dn568103.aspx

[MSDN RP] Retry Pattern in MSDN
https://msdn.microsoft.com/en-us/library/dn589788.aspx

[MSDN CRCP] Compute Resource Consolidation Pattern in MSDN
https://msdn.microsoft.com/en-us/library/dn589778.aspx

[MSDN CPG] Compute Partitioning Guidance in MSDN
https://msdn.microsoft.com/en-us/library/dn589773.aspx

[MSDN PPL] Pipelines in MSDN
https://msdn.microsoft.com/en-gb/library/ff963548.aspx

[MSDN QBLLP] Queue-Based Load Leveling Pattern in MSDN
https://msdn.microsoft.com/en-gb/library/dn589783.aspx

[MSDN CCP] Competing Consumers Pattern in MSDN
https://msdn.microsoft.com/en-gb/library/dn568101.aspx

[NARAYANA] Compensating Transactions by JBoss Narayana
http://jbossts.blogspot.it/2013/05/compensating-transactions-when-acid-is.html

[OPENSTACK] Overcommitting in OpenStack
http://docs.openstack.org/openstack-ops/content/compute_nodes.html#overcommit

[PIPELINES] Software Pipelines and SOA by Cory Isaacson
http://www.pearsoned.co.uk/bookshop/detail.asp?item=100000000274818

[RELEASE IT] Release It! by Michael T. Nygard
https://pragprog.com/book/mnee/release-it

[SAGAS] Sagas by Hector Garcaa-Molrna and Kenneth Salem
http://www.amundsen.com/downloads/sagas.pdf

[SOA PRACTICE] SOA in Practice by Nicolai M. Josuttis
http://www.soa-in-practice.com

[SOA PATTERNS] SOA Patterns by Arnon Rotem-Gal-Oz
https://www.manning.com/books/soa-patterns

[SOA PRINCIPLES] Service Reusability - SOA Principles of Service Design by Thomas Erl
http://www.amazon.co.uk/Principles-Service-Prentice-Service-Oriented-Computing/dp/0132344823

[SOC] Sagas - Service-Oriented Computing by Munindar P. Singh, Michael N. Huhns
http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470091487.html

[THERAC-25] Therac-25 machine by Wikipedia
https://en.wikipedia.org/wiki/Therac-25