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 2016-04-11

Tweet This Book!

Please help Bilgin Ibryam by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I'm reading #CamelDesignPatterns http://leanpub.com/camel-design-patterns by @bibryam

The suggested hashtag for this book is #CamelDesignPatterns.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#CamelDesignPatterns

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

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".

1. 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:

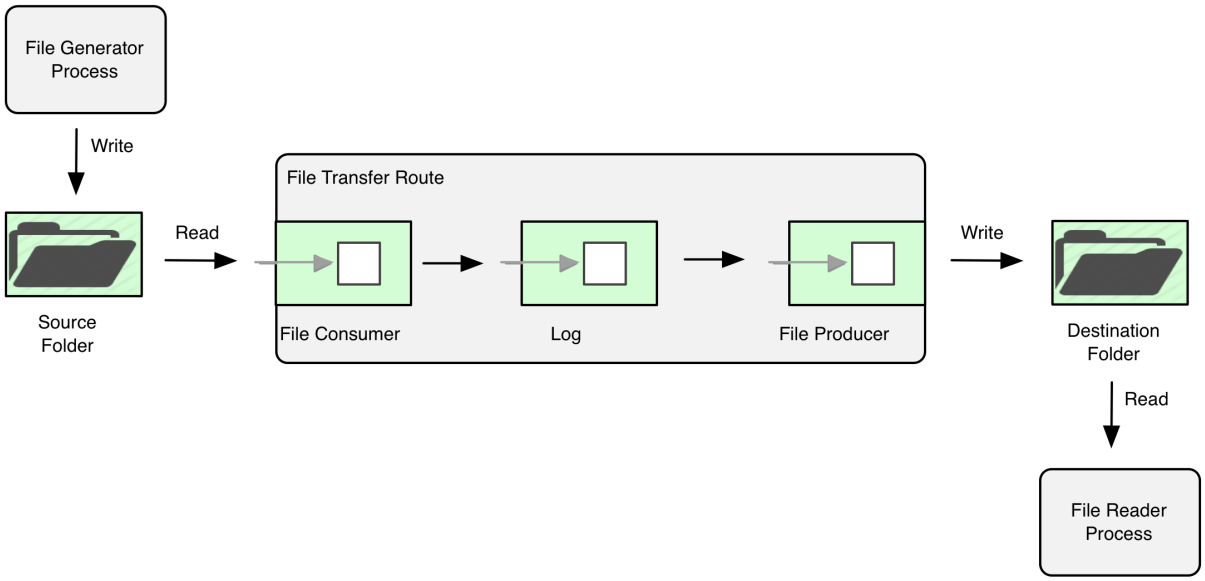
- Some systems **do not offer transactional behaviour** at all and the data consistency has to be ensured through additional mechanisms such as data locking before modifying, the committing of the changes when the operation completes successfully, or the undoing modifications when when there are failures.
- 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.

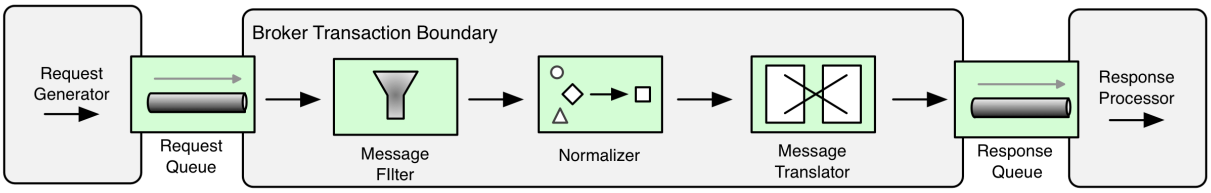
- The second important decision is concerned with how to make sure that the files we are writing to are not read by a File Reader Process before we finish writing to these files. The File Reader Process might not have all the fancy mechanisms that Camel has to detect when a file is ready for consumption (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.



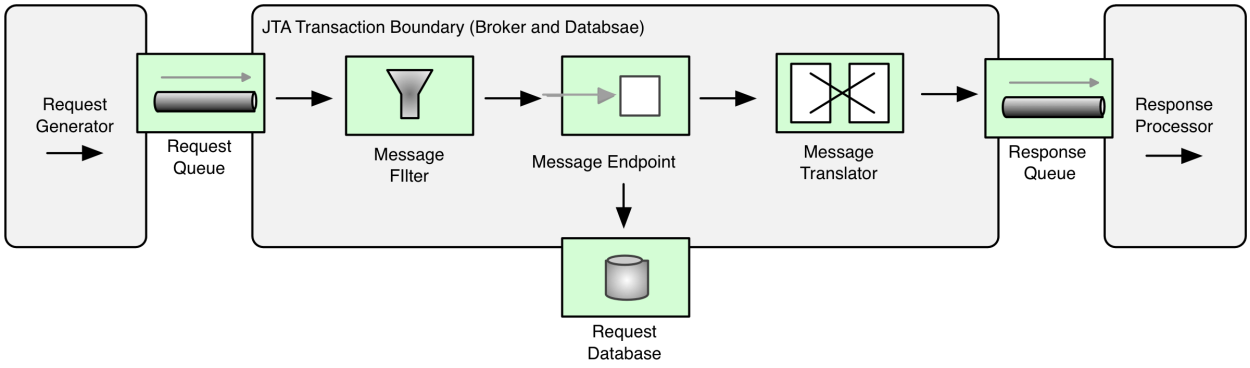
A service with local transactions

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

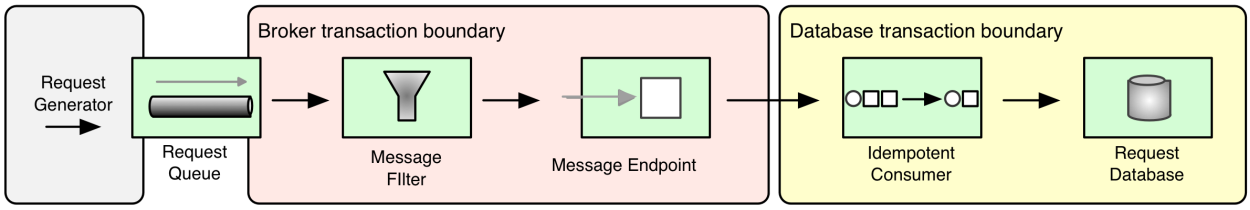
In this example, a Camel route receives messages from a request queue and writes to a relational database first. After this, it writes the result (let's assume that it is an ID generated by the database) to the result queue. The reason we need a global transaction here is because there is a chance that 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

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.

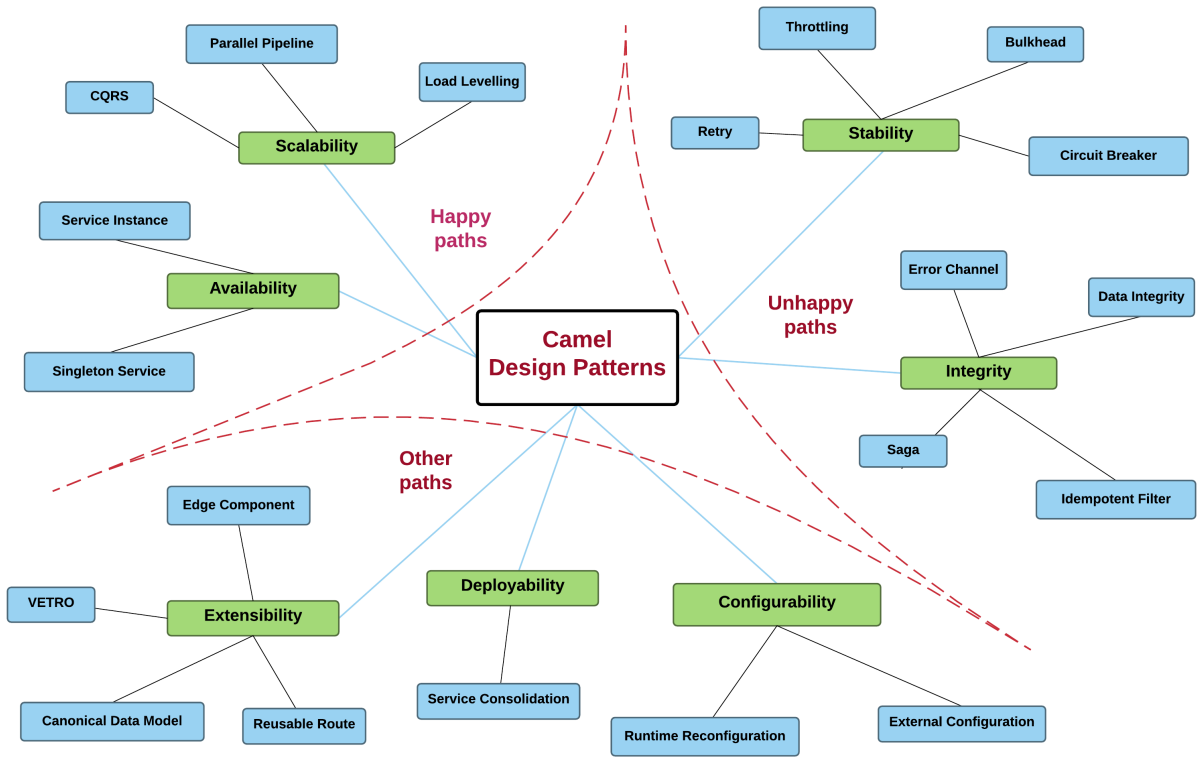
Patterns List

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_HA_Singleton.html

[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