ΠΑΝΕΠΙΣΤΗΜΙΟ
ΙΩΑΝΝΙΝΩΝ

Enterprise Integration Patterns
Building message-oriented middleware with Apache Camel

Odysseas Neslechanidis
Supervisor: Christos Gkogkos

October 10, 2022

Graduation Thesis

University of Ioannina
Department of Informatics and Telecommunications

Enterprise Integration Patterns

Building message-oriented middleware with Apache Camel

Author: Odysseas Neslechanidis

Supervisor: Christos Gkogkos

Graduation Thesis

University of Ioannina

Department of Informatics and Telecommunications

This thesis was approved by a three-person examination committee.

Examination Committee

1 Christos Gkogkos

2 .

3 .

Affidavit

I hereby affirm that this Bachelor's Thesis represents my own written work and that I have used no sources and aids other than those indicated. All passages quoted from publications or paraphrased from these sources are properly cited and attributed. The thesis was not submitted in the same or in a substantially similar version, not even partially, to another examination board and was not published elsewhere.

Signed,

Neslechanidis Odysseas

Abstract

The term "Enterprise Integration Patterns (EIPs)" refers to a vocabulary of solutions to common problems in the integration of enterprise systems. Of such vocabularies pattern languages may be constituted to allow complex business flows of diverse form to be described and handled in a uniform way.

Apache Camel is a framework that implements EIPs around a common interface based on Java Message Objects. Camel also provides an IDE-friendly declarative Domain Specific Language (DSL) oriented around this interface, which enables integration flows between disparate systems ("Camel routes") to be described neatly as Java Messages passed around between chained camel methods.

The specifics of the underlying communication protocols (FTP, http, ActiveMessageQueue etc) are abstracted away and the flow of information is cleanly described, leaving such considerations as availability, load balancing, validation, security as the primary factors influencing the middleware's architectural complexity.

In this thesis production deployments of Java Spring middleware utilizing Apache Camel will be studied. The most commonly used EIPs' Camel implementations will be inspected, and a comparison with more established integration tooling will be made when convenient, to ascertain the benefits of the Message-Oriented Middleware (MOM)-backed Camel DSL approach.

Keywords: Enterprise Integration Patterns, Apache Camel, Message-Oriented Middleware

Table of Contents

Abbreviations

API:	Application Programming Interface
EAI:	Enterprise Application Integration
EAS:	Enterprise Application Software
EDA:	Event-Driven Architecture
EIP:	Enterprise Integration Patterns
EMS:	Enterprise Messaging System
ESB:	Enterprise Service Bus
MEP:	Message Exchange Pattern
MOM:	Message-Oriented Middleware
QoS:	Quality of Service

List of figures

Part I Enterprise Application Integration (EAI): the why and the how

1 Introducing EAI in a organization

1.1 Introduction

Enterprise Application Software (EAS) is the term for computer programs used to satisfy the needs of an organization rather than individual users. Almost all business operations, at different points in time, have come to benefit from the proliferation of software in this space. Commonly used acronyms used to categorize such software include ERP (Enterprise Resource Planning), CRM (Customer Relationship Management), BI (Business Intelligence), CMS (Content Management System), WMS (Warehouse Management System). They serve to automate every business need of modern enterprises, from it's customer facing operations, to keeping track of warehouse inventory, calculating billing and taxes, observing regulations, and much more. While comprehensive enterprise software suites offering differing degrees of customizability have come to exist, owing to the organizational similarity of enterprises above a certain scale, switching costs, preservation of optionality in partnering with software vendors[7], as well as other adjoining business considerations, have hindered their more widespread adoption. Added to that, the employment of Domain-driven design, in recognition of the maintainability and extensibility benefits domain-expert input in the refining of an application's domain model confers, is a fact that has further complicated the effort of business software consolidation.

In this setting, the introduction of a "software glue" stack has come to be a very common business need, and much research in the space of EAI is aimed at providing insight for the development of better solutions in this class of software. The established term for such software is “middleware”[5, 15].

From the systematic study and development of solutions in this space, a particular subtype termed Message-Oriented Middleware, or MOM, has emerged as one the most promising. A vocabulary and a framework implementation for describing and building such middleware constitute the main topic of this thesis.

1.2 General challenges

Prior to engaging with the path-dependent and hard technical aspects of Enterprise Application Integration, it is necessary to consider a set of social and organizational features that the development and adoption of such solutions typically necessitate or bring about.

Enterprise Application Integration often requires a significant shift in corporate politics. By extension of Conway's law that postulates that “Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations.”, it appears that the consolidation of enterprise software tools serving business processes often necessitates a consolidation of the business units and IT departments involved in those same processes.[7]

Furthermore, owing to the wide scope of a middleware integration solution bringing together critical business functions, the novel risk of failure or misbehavior of such a system has to be internalized. The risk profile and magnitude of reorganization around such a single point of failure ought to be carefully considered.

Bordering the technical side, the feasibility of integrating systems by modifying them to better fit the integration architecture, rather than by having to design the integration architecture to work around the various systems' limitations and deficiencies, also often depends on political factors. In that vein, unsupported legacy systems still in operation, systems under proprietary licenses, and systems whose support is outsourced under more or less stringent long-term agreements can adversely influence the complexity of the final product.

In terms of standardization, it bears mentioning that despite the benefit of convergence around Web Services and a Service Oriented approach to middleware architecture (which will be expounded upon in later chapters), the proliferation of new extensions or interpretations of the standard, and most significantly the shift towards REST (and, more recently GraphQL) in lockstep with the mobile revolution, has created new challenges for integration engineers. REST, in particular, owing to it being an architectural style for software that expose http APIs rather than a protocol for web services per se, is frequently implemented partially and/or wrongly, often necessitating ad hoc code for the consumption of APIs exposed in this manner.

Finally, the operations aspect of utilizing middleware solutions presents a unique challenge, as maintenance, deployment, monitoring and troubleshooting of such heterogeneous, distributed systems commonly require mixes of skills which are not, as a matter of course, to be found in single individuals. To companies or organizations of sufficient scale as to already necessitate a formalized employee training regime, the overhead for the maintenance of such human capital might be lower.[7]

1.3 Types of integration

While the above challenges generally apply to every approach in the broader category of integration, many further issues have to be considered depending on the business aims that dictate, and the technical aspects that come as a consequence of, the prospective type of integration solution. The following categorization has been proposed:

1.3.1 Information Portals

Information portals serve to aggregate information from disparate systems within an organization with the aim of making it more accessible to humans. They often facilitate the collaboration between different departments and physical locations. They are also commonly used in business decision processing and data analysis. Common features include multi-window views serving information from different sources with automatic refresh of related windows during navigation, search, tagging and other categorization schemes.

Various other more advanced features are common, but being as they cater to particular business functions, employees roles or departments, no account of those will be attempted. Indeed, one of the common abstract features, or aims, of such systems, is the personalization of the displayed information, achieved through the profiling of users based on role, experience, competencies, habits and expressed preferences.

1.3.2 Data Replication

Many business systems require access to the same data, but are designed to utilize their own, separate datastores. The resulting data replication necessitates provisions for maintaining the data synchronized. Commonly utilized for those purposes are the replication features built into modern Database Management Systems, the file export and import functions supported by many Enterprise Software Systems, and message-oriented middleware automating transport of data via messages between arbitrary datastore solutions.

1.3.3 Shared Business Functions

Needless duplication can exist in code serving business functions as well as in data. Were supported, invocation of shared business functions implemented as services[1] can help avoid the native implementation of redundant functionality.

Were feasible, the need for data replication can also be circumvented via this approach by serving shared data as a service. In that vein, some criteria to be considered include the amount of control to be had over the systems (calling a shared function is usually more intrusive than loading data into the database) and the rate of change of the relevant data (service invocation is costlier than data access, therefore is less efficient for relatively frequently accessed, relatively static data).

1.3.4 Service-Oriented Architectures and Distributed Business Processes

Once an enterprise assembles a collection of useful services, managing the services becomes an important function.

Service Oriented Architecture is a proposed style of service design and orchestration that incorporates the best industry practices in structuring middleware solutions around services that correspond to business functions. This particular approach to middleware architecture shall be expounded upon in a later chapter.

A variant dubbed "Distributed Business Process", is also to be found in the bibliography. It concerns the design of management services that serve to coordinate the execution of the relevant business functions that are implemented natively in an integrated system's constituent

[1]A service is a contract-defined function that is universally available, and responds to requests from "service consumers" .

applications, in order to achieve each and every particular business process. Such schemes can exist within larger SOA-abiding systems, and the lines between the two terms often blur.

1.3.5 Business-to-Business Integration

In many cases, business functions may be available from outside suppliers or business partners. Business to Business (B2B) integration software provides the architecture needed to digitize information and route it through an organization's trading ecosystem (usually online platforms) using the Electronic Data Interchange (EDI) format appropriate for the application.

In the following chapter, the various technical approaches to Enterprise Application Integration will be discussed, beginning by retracing the historical contingencies defining the evolution of the EAI field, and culminating with a direction of focus towards the widely successful event-driven SOA approach and the message-oriented middleware used to facilitate it, a particular implementation of which will be the topic of the rest of this thesis.

2 The Evolution of Enterprise Application Integration

2.1 Islands of automation and the advent of EAI

The term "Islands of automation" was a popular term introduced in the 1980s to describe the status quo of automation systems existing within information silos. The rapid development and adoption of enterprise software systems during this time came to pass with little regard for the ability of those systems to communicate with one another.

Such fragmentation of automation systems turned out to significantly increase the cost of operations within organizations, and contribute to a higher barrier of transaction cost for cooperation across different enterprises. A major part of business operations requires coordination between multiple departments/organizations, each with their own system of

automation. In this state of affairs, manual intervention is required to keep information systems updated, human effort, data, infrastructure are often duplicated needlessly, and the risk of costly human error is introduced at multiple points.
The field of Enterprise Application Integration (EAI) is a field of study aiming to refine a framework for rectifying these inefficiencies. The shifting nature of the business landscape and of enterprises that operate within it, together with the continued innovation in, and expansion of, the EAS space, has resulted in it being a complicated problem to tackle. Enterprise software is adopted at different times, it is developed from different vendors, at different points in time, oriented towards different business needs.

As previously noted, the role of middleware is to to facilitate communication or connectivity between applications that were developed without such provisions, often through channels beyond those available from the operating system serving as the platform, or across distributed networks.[13, 3] In the early days of EAI, the development of custom middleware solutions begun spreading as a practice.

2.2 Point to point integration

The conceptually simplest way to perform integration is by connecting information systems directly in a point-to-point paradigm. In a common implementation, custom procedures are called on both ends targeting the native filesystem as the locus of communication between the systems, often in conjunction with a network file transfer protocol such as FTP. A system assuming the client role executes a reporting routine to extract data to a text file in a specified format. A routine is then run by the receiving application to import and process the data.

As similar point-to-point solutions begun to emerge, it nonetheless became apparent that the net cost of development and maintenance of such solutions stood high, and steeply increased with scale. This came as a result of the fact that in the point-to-point approach, the introduction of one new system typically requires many specialized connections to the existing systems, which in turn impose additional maintenance burden, reduce agility, and constitute additional potential points of failure.

Additionally, the tight coupling makes reliability a challenge, especially for real-time applications. For example, if the connection between two parties in a client-server connection is interrupted, the data supposed to be received by the client will be lost during this interruption, unless complex logic to deal with caching, session management and error recovery on the server side is implemented.

Moreover, the synchronous nature of the communication ties up resources to handle the interaction, which presents a bottleneck as the system scales both in service load and complexity.

In retrospect, this model of integration remains suitable when the software entities in the integrated system are relatively few, and/or the interactions are simple. It is in cases when there are many entities, which need to interact in multiple ways and in particular sequences, e.g. when the interactions are stateful, that the system's requisite topological complexity can become onerous.

2.3 Event-Driven Architecture and the hub-and-spoke pattern

One notable alternative architectural approach that serves to address the downsides of the P2P model first came to prominence as the hub-and-spoke pattern. Based on the concept of “events”, this system is built around a “hub”, that serves as the common target for the systems on either side, each assuming the role of either a “producer”, or a “consumer” of events. In the simplest implementation of the hub, which makes no provision for central orchestration of the events in transit, the hub's role is described as that of an “event broker”. The communication is multicast, with each event produced being "published" to the broker, and received by all consumers who have "subscribed" for receiving this event.

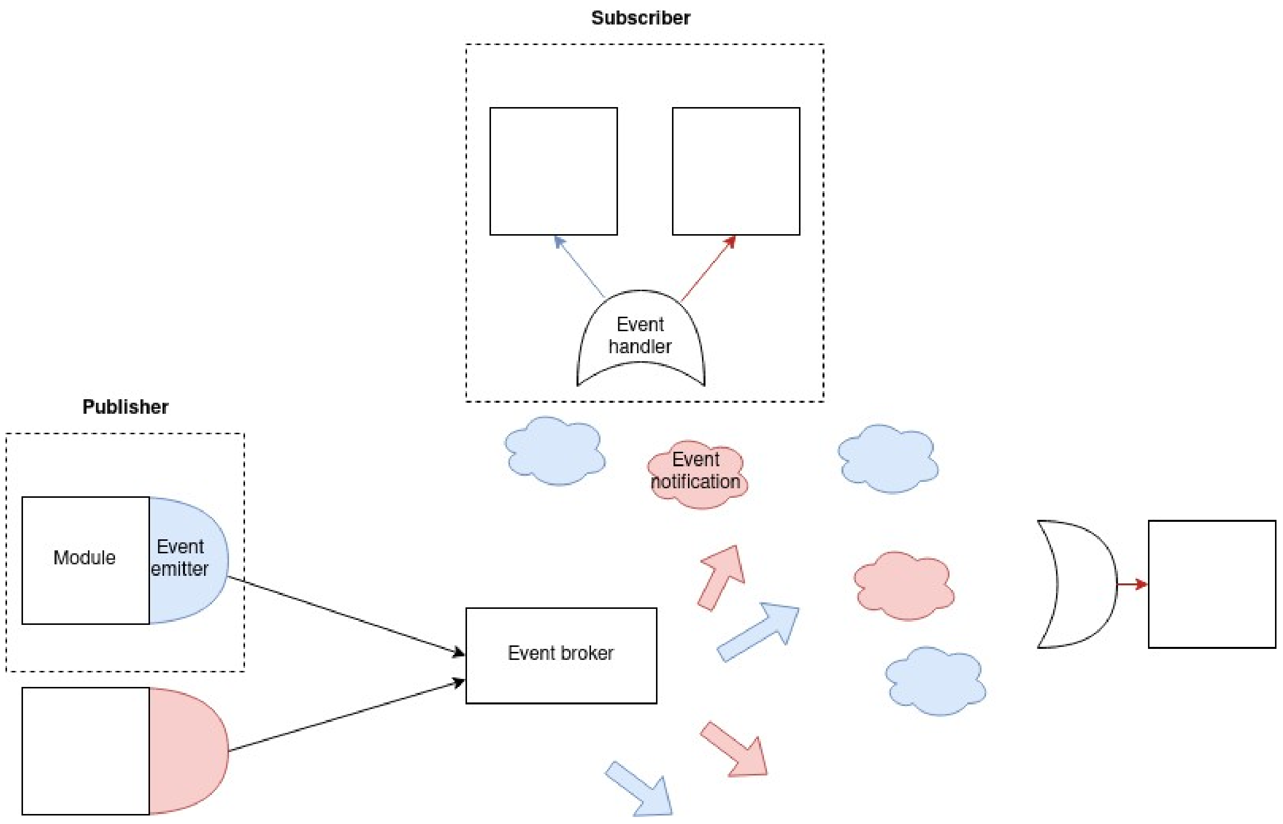

Figure 1 EDA, Broker topology

In a somewhat different topology, that requires a more complex hub implementation, the hub is meant to act as an "event mediator", centrally maintaining state regarding the event notifications. This positions the hub as the programmable orchestrator of the communication between systems, making more complex interactions possible and enabling it to act as a "load balancer”, by allowing event notifications to be directed towards exactly one consumer, and to be kept to be resent in case there is no consumer available (“event queue”).

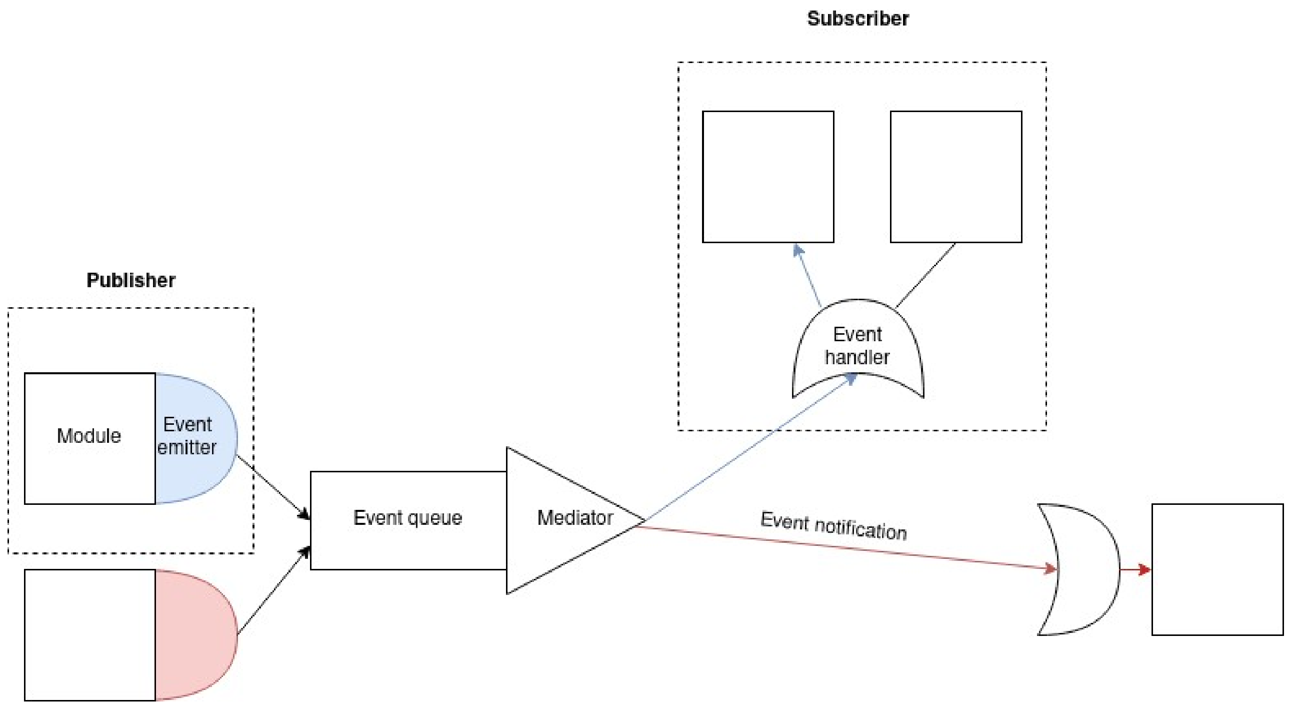

Figure 2 EDA, Mediator topology

The above variations of the same pattern, utilize, as they may, different semantics to describe their operation, evident in the terminology-laden paragraph above, they do nevertheless share a set of essential characteristics to differentiate them from the previously mentioned point-to-point pattern:

- Multicast communication: Each event can have more than one possible recipient-subscriber.

- Asynchronous communication: The publisher does not wait for a subscriber to process an event before sending a new event notification. Also called “fire-and-forget”.
- Loose coupling: Event publishers are not aware of how many, if any, subscribers to the event there are, nor are they informed of how any particular event's processing proceeds. Publishers may still be built to facilitate a stateful sequence of events, though it is often preferable for such sequences to be mediated centrally, by the hub, rather than by the participating applications.[10]
- Ontology: Event-driven systems commonly adopt a system-wide convention for prioritization and grouping of events. This allows subscribers to subscribe to entire categories of events or events that fall at some particular point in the hierarchical sequence of events regulating a business process. To indicate the distinction, subscribers are properly said to subscribe to “topics”, which can correspond to either particular events or categories thereof.[12]

A further point to be made on this property of hub-and-spoke, and event-driven architectures in general, is that ontologies produce what is called “semantic coupling”. Event groups or hierarchies are only meaningful within the context of a system adopting the particular ontology within which they are represented. This makes communication between systems implemented with different ontologies impossible, unless an intermediate semantic matching technique is employed. Research is currently active in this area.[6]

Event-driven architecture is properly constituted of subscribers that are both stateless and context-free. Each event notification ought to contain just enough details to enable the event handlers[1] to guide the business flow in the intended direction e.g. by selecting among running

[1]Alt. subscribers

one of several stateless, functionally autonomous modules, that may or may not be event emitters[2] themselves, or halting.

Event notifications should not provide any additional context. Also, the behavior they trigger should not depend in any way on the in-memory session state of the receiving applications.

All things considered, the Event-driven Architecture paradigm, confer as it may several benefits over the point–to-point model in integrations of scale, is nevertheless ill-suited to certain specifications commonly required of enterprise systems.

First of all, the ability to chain hierarchical interactions between modules, is only possible through defining routing rules at the event mediator. While the convenience aspect of this method due to the centralization of the more complex parts of the system is not to be discounted, the degree of control over such interactions, in particular with regards to Quality of Service considerations such as time-sensitivity, reliability etc is rather low.

Also, event mediation adoption comes at the price of relatively tight coupling between the prospective event handlers and the mediation-capable hub.

Ultimately, event mediation can be an appropriate solution for a number of special cases in the context of loosely-coupled IT infrastructure mirroring diffuse business process environments, but is far from a satisfactory way to handle vertical interactions among functionally autonomous modules.

For the reasons referenced above, event-driven systems, while highly performant, are not suitable ways to integrate applications with time-sensitive interactions, e.g. Human-Computer Interfaces in banking applications.

[2]Alt. publishers

It is also a point worth making separately, that provisions for reliability such as delivery acknowledgment, transaction atomicity, security etc are formally unsuited to EDA systems' design, and ad hoc interventions towards these ends can diminish EDA's inherent benefits.

Finally, the asynchronicity of EDA systems makes them more complex and harder to test, owing to the introduction of event communication infrastructure such as the hub and event channel implementations, and the non-deterministic nature of parallel computation.[10, 5]

2.4 Service Oriented architecture and the Enterprise Service Bus

A synchronous architecture meant to address the point-to-point paradigm's numerous drawbacks in terms of technical debt accretion, agility and complexity, while factoring in reliability provisions, is referred to as Service-Oriented Architecture, SOA for short.

Service Oriented Architecture is an evolution of predecessors such as component-based architecture and Object Oriented Analysis and Design of remote objects e.g. the CORBA standard

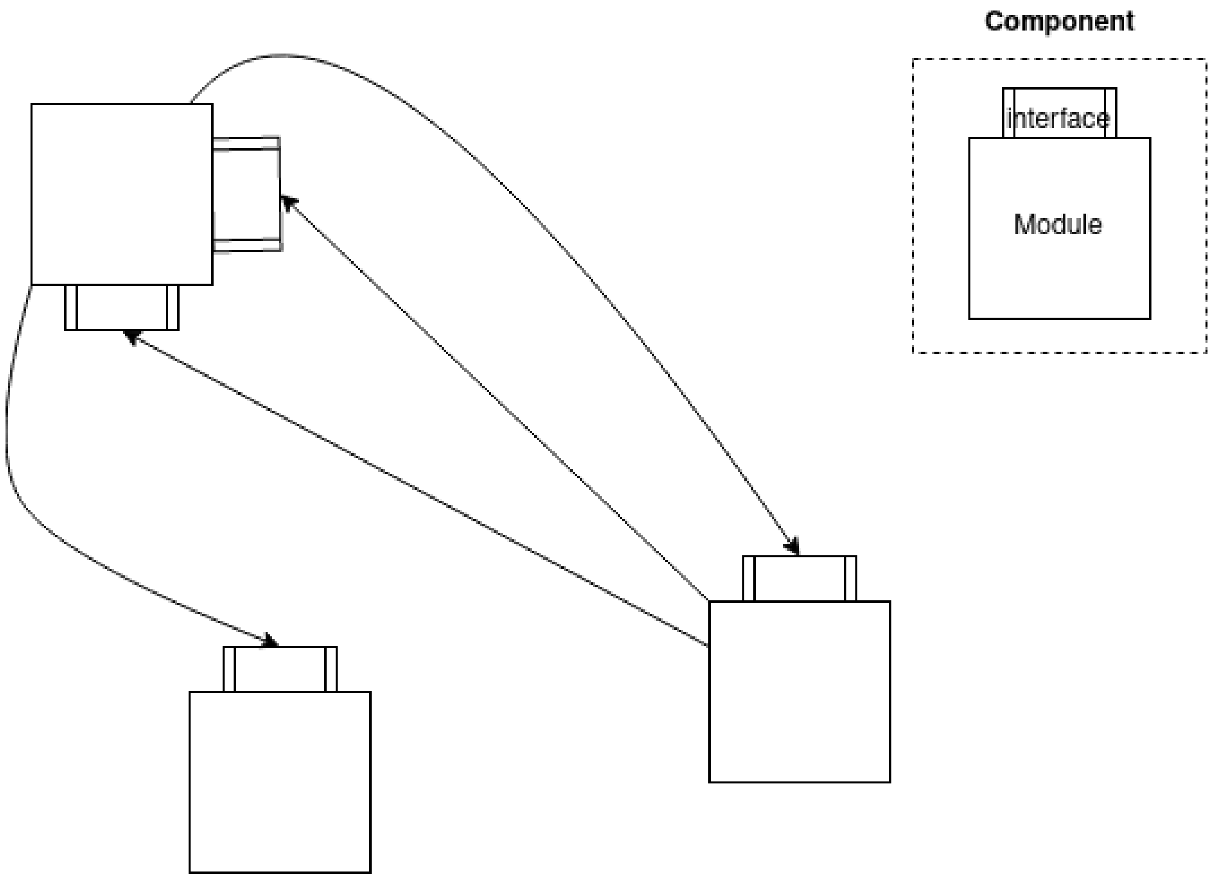

Figure 3 Component-based, point-to-point architecture

Component-based architecture emphasizes separation of concerns with respect to the various functions provided in a given software system. Components are commonly implemented around interfaces, that encapsulate the particulars of the components' implementation, and narrow the available surface-area for wiring together the various functionally autonomous modules. Cohesion is maintained by fitting additional modules onto the interfaces. The modules, which can be of arbitrary origin, are rendered into components by implementing their respective interfaces. The modules can exist as components locally within the same virtual or physical machine, or in the context of distributed systems such as networks (e.g. as web services or web resources).

In the SOA evolution of this approach, reusability and use in the context of distributed systems is emphasized. To realize this architectural style's potential, the promulgation of Web Service standards becomes instrumental.[1]

In this way it is ensured that networked software components can be developed as generic “Web Services”, or business function-specific components that are implemented without knowledge or regard for the multitude of systems in which they may become involved.

Based on this, SOA can be defined as an architectural style focused around designing a system as a dynamic collection of services capable of communicating with one another. If the conventions are observed diligently, the need for an EAI hub and it's accompanying module-specific connectors, database drivers and protocol adapters is theoretically obviated.

The Enterprise Service Bus (ESB) is the architectural feature of SOA systems enabling communication in a special variant of the more general client-server model, wherein any which service may behave as server or client. Universal availability and statelessness, both criteria met by proper services, but not by traditional server-client component couples, are prerequisites for the establishment of such as system.[12]

The ESB is equivalent to the “bus” design concept found in computer hardware architecture, in this case used to refer to the technological infrastructure used to implement a model for communication among independent, non-context aware software services running within networks of disparate and independent computers.

[1]The effort towards this end has borne results through the W3C Web Services specification, though nowadays the emergence of alternatives and the REST architectural style in particular has created a rift in the SOA ecosystem, whose bridging is often handled by integration middleware.

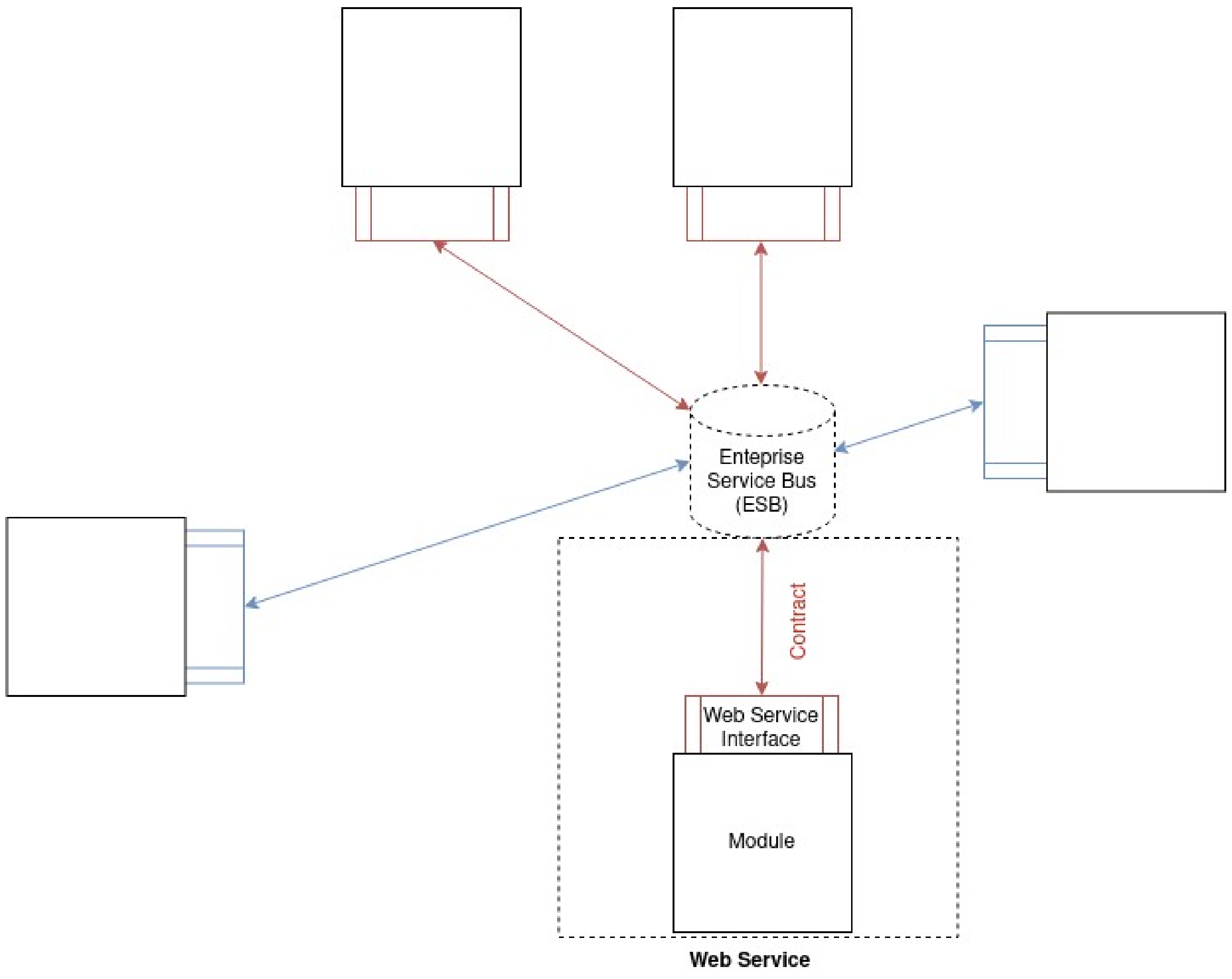

Figure 4 Generic SOA

When working with more modern enterprise systems, which provide a Web Service-abiding API interface, implementing the ESB pattern amounts to providing certain service management capabilities, such as a means of controlled exposure of said APIs, using an exposure gateway service.

The primary aim of service management is to facilitate service discovery and exposure via a database called a Service Registry[16] - in this particular scenario, it's function being partly substituted by the gateway - and an HTTP-accessible querying API. Reliance on human-maintained interface documentation e.g. Swagger or human-to-human interaction has proven a brittle strategy that erodes reusability, which, it is to be noted, is regarded as one of the main advantages of SOA.

More complete service management solutions deal with additional aspects, namely service negotiation i.e. the ability to set up a communication contract/connection with services, implementation of a security model with patterns for access control, e.g. with user roles/permission schemes, traffic control, encryption/redaction etc. Supplemental features can comprise configurable web portals that may describe the available APIs, enable potential users to issue keys automatically (self-subscribe) in order to use the APIs, provision analytics for both users and providers of the APIs, etc.

In systems requiring advanced service management capabilities, the related functions are commonly relegated to a separate database runtime known only to the gateway, introduced earlier as the Service Registry. Furthermore, in more complex integration scenarios, such as when unusual protocols or data formats are utilized, when compositions of multiple requests are called for, or perhaps in cases where transactionality needs to be implemented, the introduction of an integration engine existing as a separate runtime is, again, expedient. This hub-like arrangement is only one, albeit very common, of many possible topologies that are in accordance with the loose definition of ESB given above.

Even so, the term “ESB” has de facto come to refer to integration engine solutions adopting the architectural approach of a hub, often federated, whose main purpose is to facilitate a message-based communication model (termed Enterprise Message System, EMS) to be used within a particular SOA system's context. Such integration engines commonly provide auxiliary capabilities, which were found to be essential additions for constructing more complex systems in the service-oriented style. They commonly contain logic for the encapsulation of legacy formats, protocols (or informal specs) and APIs of the integrated applications into an EMS compatible format, incorporate a service registry, and sport numerous other features for message routing, mediation, transformation, enrichment, validation etc.

A summary of the archetypal SOA model's features in axes of comparison common with those of the aforementioned point-to-point and event-driven architectures could thus be:

- Unicast communication: Communication is established in provider-consumer pairs.
- Synchronous communication: A service consumer invokes a service provider through the network and has to wait until the completion of the operation on the provider's side, upon which a response is returned.
- Reusability/Interoperability: Services within a particular SOA context are defined by standardized service contracts, which include the interface, the schema, the communication protocol and various Quality of Service (QoS)[1] policies. Each service can have multiple contracts, aimed at supporting different consumers. While these requirements result in tight coupling of services to the particular SOA context, reusability and interoperability among contract-abiding services is high, and further increased through the introduction of a service registry to facilitate discovery and exposure.

[1]Quality of Service refers to the performance of a network service in multiple areas that commonly include scalability, security, reliability, lossyness, delay etc.

- Domain-driven design: Loose internal coupling allows Domain-driven design to be observed for a SOA system's constituent services. It's central idea is the creation of a Ubiquitous Language with the assistance of domain experts, that embeds domain terminology into the software components' naming and structure. This practice is claimed reduce friction in the operations side and increase maintainability and creative cross-domain collaboration.

- Business-centeredness: While many components within a SOA context may be developed in such a way as to constitute candidate services, only a subset of those have their service description exported, and perhaps published to a service registry, if there is one. Exposure in a controlled way is ensured via the provision of a fitness-for-purpose, or “litmus test”, that determines whether a given service implementation meets certain business alignment, composability, reusability and technical feasibility criteria.[11] This is facilitated greatly by the existence of accompanying service contracts, that allow the evaluation of the service's technical characteristics to remain separate from considerations concerning it's internal design.

In simpler scenarios, SOA systems are designed around a synchronous ESB, a property derived from the synchronous nature of their constituent services. A common approach is that of the service-oriented API gateway pattern, touched upon briefly in a previous section. It refers to business-coupled deployments consisting of components wrapped as Web Services, which are configured as to delegate service management to the system's gateway service. This service is charged with receiving requests from it's external API, accessible to clients outside the SOA system, aggregating the various services required to fulfill them, and returning the response in a synchronous manner.

2.5 SOA 2.0

While taking an antidiametrical approach to communication with regards to EDA (pull vs push - soliciting a response as opposed to publishing an event notification), thus evading many of it's shortfalls, SOA, in practice, has come to serve as a useful complement in many EDA-structured enterprise systems, with services commonly being wrapped as components triggered by event handlers.

For most applications, communication then has to be framed within an Enterprise Messaging System that enables information flow and routing through event-driven infrastructure, and into synchronous services.

The popularity of such hybrid systems, combined with the aforementioned benefits of dedicated SOA integration engines, has brought about a state of affairs where most productized ESB implementations have come to rely on distributed, message-oriented middleware, dubbed "ESBs", introducing a federated hub and event-driven messaging infrastructure as ubiquitous features of present-day SOA systems.[12]

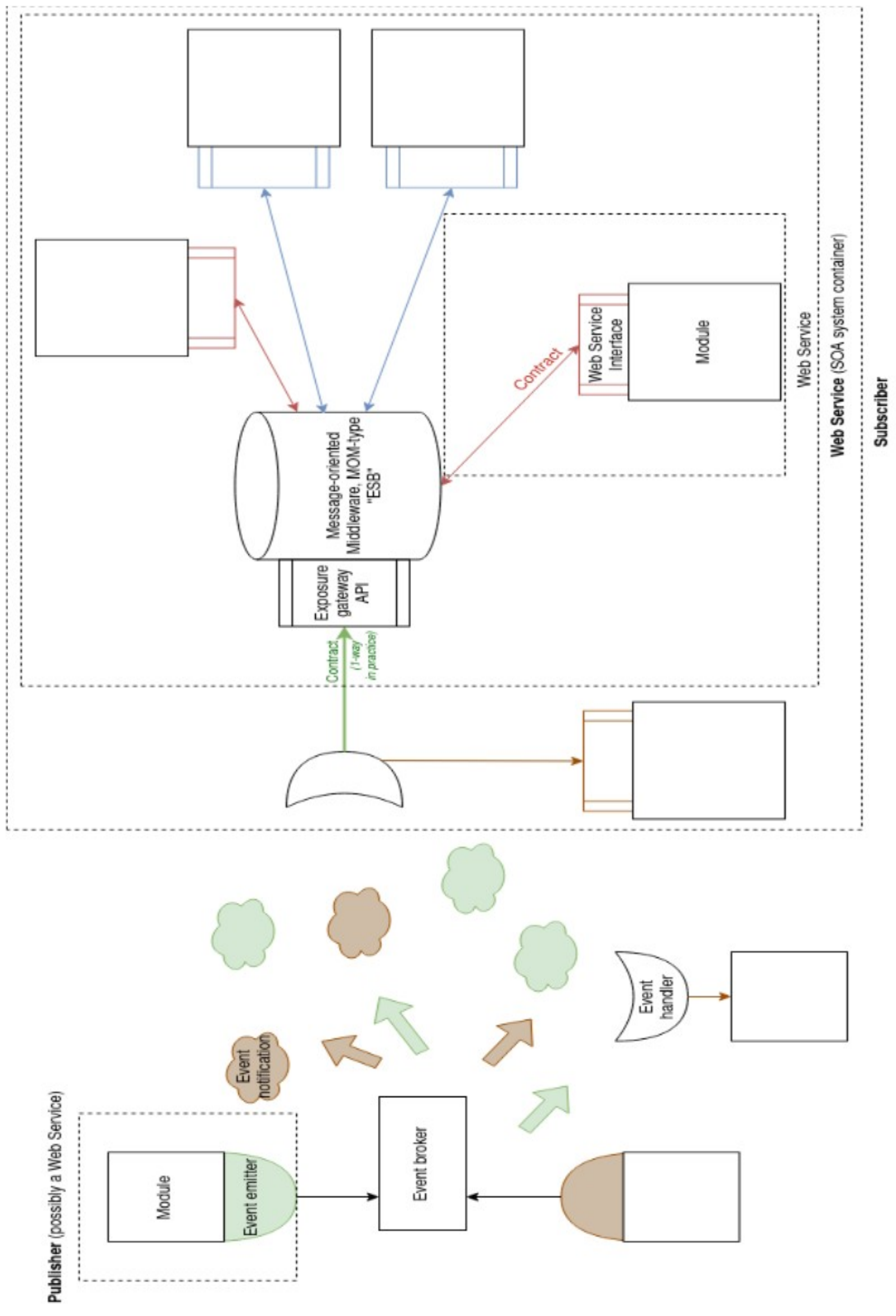
Publisher (possibly a Web Service)
Module
Event emitter
Event broker
Event notification
Event handler
Subscriber
Web Service (SOA system container)
Web Service
Module
Web Service Interface
Contract
Message-oriented Middleware, MOM-type "ESB"
Exposure gateway API
Contract (1-way in practice)

Figure 5: Event-driven SOA (SOA 2.0)

This model of SOA has been called next-generation SOA, or “SOA 2.0”. To elucidate the cause of the wide popularity of this hybrid, a pros-and-cons comparison of the two styles will be attempted.

Traditional SOA can be advantageous for:

- Service chaining: Operations that involve interaction between vertical hierarchical layers of functions packaged as services are common. The request-response model of communication, supplemented by polling mechanisms for interactions involving more than two services, ensures the interactions are performed in the correct order. The implementation of such chains is further simplified by the assured existence of contracts that specify the Quality-of-Service aspects of the modules involved.

- Human-Computer Interaction and other time-sensitive processes: The synchronous and contract-bound nature of communication is expedient for the design of real-time services, such as the decision-enablement and actuation capabilities that are expected from modern enterprise systems.

- Transactionality: Interaction between service pairs can be said to implement “pull” semantics. Information like response status codes can be propagated back down the call stack, through service chains of arbitrary depth, until the original service consumer receives their response. Mechanisms relying on this property can be put in place to guarantee transaction atomicity[1].

[1]Either all interactions occur successfully or no interaction occurs.

- Testing: Synchronicity makes contract specification possible, and contract-specified policy boundaries increase the tractability of operations involving multiple services.

On the con side of SOA, it could be noted that, despite the the high internal interoperability of systems designed in this manner, interfacing with external systems remains hard, as they would have to be wrapped as services abiding to the particular ESB setup, and establish connections bound to service-contracts composed often under unaccommodating assumptions.

Meanwhile, EDA exhibits it's own set of advantages in multiple areas:

- Business-to-business (B2B) integration: EDA implements “push” semantics. An event emitter's active involvement in communication ends once it publishes an event notification. Control of an operation's flow is shifted away from the event source, being distributed/delegated to event handlers. An enterprise partner can integrate with a preexisting EDA system with relative ease, by encapsulating their system's components as event emitters that conform to the system's particular event semantics. In case a partner already utilizes their own EDA system, a semantic matching solution can alternatively serve to directly bridge the two systems.

- Business workflow and other arbitrarily-halting processes: Many business processes involve human input in their workflow. A Warehouse Management System (WMS), for example, has to be updated with the status of the physical warehouse by warehouse employees, before moving forward with the processing of an inventory receipt. As EDA is asynchronous and contract-free by design, it enables such operations to be readily embeddable within an Enterprise Software-driven workflow.

- Ease of deployment: The EDA pattern is characterized by loose coupling which allows independent deployment and unhindered horizontal scalability, as there are no dependencies among the participating components. For solutions that require maximum ease of deployment, event broker topology is a better option than event mediator topology. This is due to the fact that in the event mediator topology, relatively tight event mediator - event handler coupling can exist .

- Performance: Asynchronicity makes data parallelism via multi-core processing possible

- Scalability: The above two points together, are essential elements for systems envisioned to be scalable.

On the con side, EDA system testing is not easy due to the asynchronous nature of the processing, and the concomitant lack of service contracts.

The introduction of an event-driven, message-oriented middleware at the core of a SOA-inspired system increases the base complexity of the design, but can, in theory, result in a SOA-EDA hybrid bestowed with advantages from both approaches while eliminating their respective drawbacks. A resulting system would benefit from a performant, loosely coupled, event-driven architecture, with a design amenable to Business-to-business connections and hybridizable with SOAs encapsulated as event emission-capable services triggered by event-handlers.

Part II Engineering concepts and tools

3 Apache Camel: a framework for constructing Message-Oriented Middleware

3.1 Introduction

The convergence around hybrid SOA architectures and Enterprise Messaging Systems as the default type of communication model for enterprise integration brought into focus the need for a common, platform-independent language to describe the common capabilities and architectural features of such systems. The resulting vocabulary is what is referred to as "Enterprise Integration Patterns" (EIPs).[14]

Apache Camel is a framework for building message-oriented middleware. More generally, it aspires to enable integrations designed around the Enterprise Integration Pattern (EIP) vocabulary. In addition to native support via JMS for ActiveMQ and other message brokers and infrastructure in the EDA paradigm, it provides features that enable most common SOA architectures, modern and legacy alike. Standard SOAP Web Services, RESTful http Web Services and more are natively supported, with Amazon Web Services, GraphQL and other modern service-oriented technologies supported as extensions.

3.2 Basic Concepts

Arguably the most important aspect of Camel is message routing, which is utilized to enable SOA-style service composition within the context of an event-driven integration system.

A Camel route begins from a consumer endpoint, which corresponds either to an event handler passively receiving inbound messages published on a particular event topic, or to a polling function synchronously monitoring a particular source by fetching messages at scheduled intervals.

Components are batteries-included Camel factories for creating Endpoint instances. They contain the protocol-specific logic for obtaining information from outside sources and packaging it as inbound messages.

The inbound messages' carrier through the service chain representing a Camel route is called a message exchange.

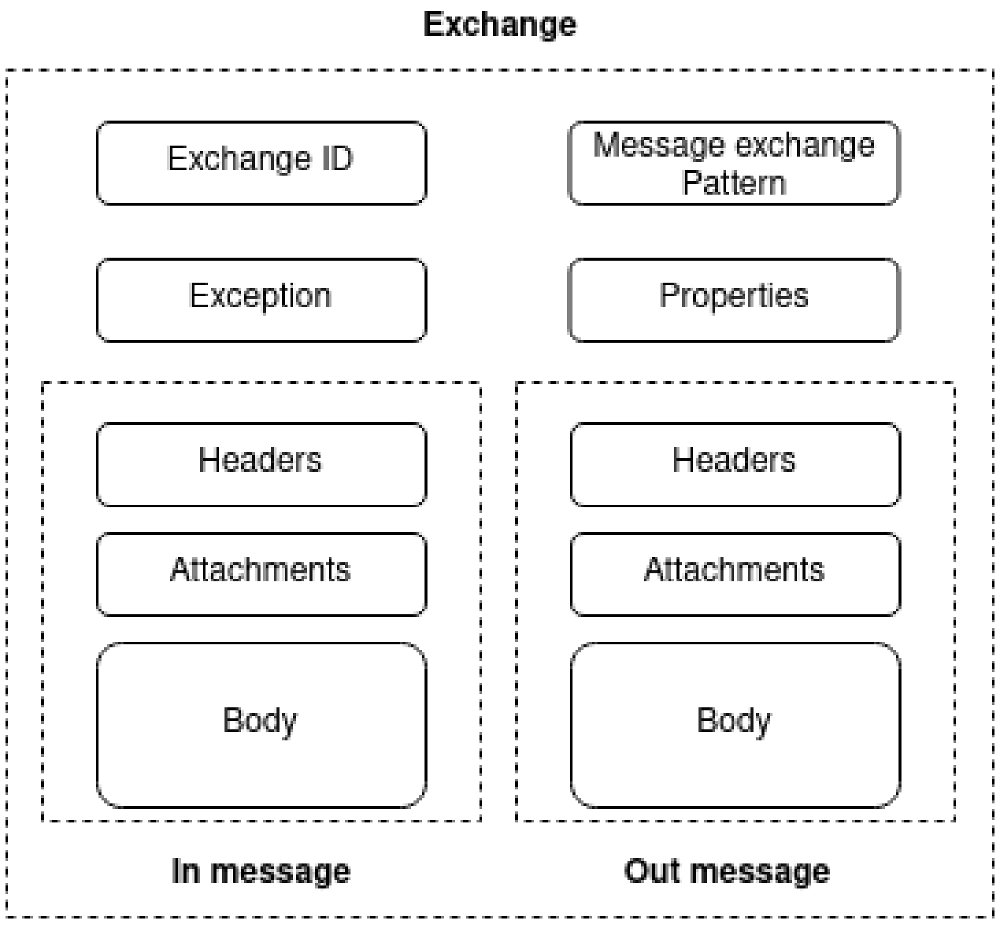

Figure 6 Camel Exchange

There are two types of Message Exchange Pattern (MEP). InOnly corresponds to operating as an event handler. The incoming Message is an event notification, being used in one-way communication where flow control is passed to the receiving module. InOut is a pattern used to emulate a SOA-style request-response interaction, and is useful for chaining local services with Web Services or other remote components.

The Camel Processor is the interface for incorporating custom logic, such as conventional system components or services, into a Camel route's exchange channel.

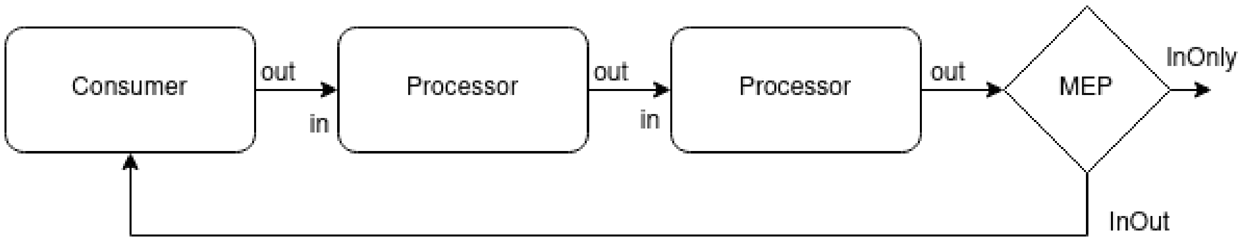

Figure 7: Camel Route

Finally, by Camel's offering of a Domain Specific language, which is implemented simply as a Java API that contains methods named after EIP terms, an integrated system's Camel Routes are rendered into a prime leverage point for inspection and control of the logical flow of integration.[8]

3.3 Example of a Camel Route

We maintain the middleware stack of a logistics company. One of our integration solution's tasks is contacting the various couriers' APIs and issuing shipment vouchers for the goods that need to be delivered, then updating the company's ERP with this knowledge through an http callback. Open requests for issuing such vouchers are stored in a database. This is the Camel route in full:

```
 from("timer://$Voucher?period=500000&repeatCount=-
1")            .log(LoggingLevel.INFO,logger,"Starting polling

                                        for open voucher requests")



.setHeader('courierCode').constant(configuration.courierCode)

.to("sql:classpath:sql/openVoucher.sql")

.log(configuration.name +'Voucher: Processing ${body.size}

                                        voucher requests')



.setProperty('totalRequests').simple('${body.size}')

.split(body(),voucherAggregation)
```

```
    .log(LoggingLevel.INFO,logger,'Posting for voucher

                                  for request: ${body}')



    .process("issueVoucherProcessor")

    .filter().simple('${body.voucherCallbackUrl} != null')

        .process('voucherCallbackProcessor')

    .end()

.end()

.log(LoggingLevel.INFO, logger, 'Successfully Updated

${property.successCount}/$ {property.totalRequests} requests')
```

The endpoint of the Camel route in question is a polling consumer firing periodically every 500000 seconds, The timer component is used just for the purpose of triggering the start of an exchange, yielding an inbound message with an empty body:

```
 from("timer://$Voucher?period=500000&repeatCount=-
1")            .log(LoggingLevel.INFO,logger,"Starting polling

                                  for open voucher requests")
```

The empty message is wrapped in an exchange, which is propagated to the .to() Camel EIP. The .to() EIP is an endpoint that, unlike .from(), which acts only as a message consumer, is also capable of producing messages. In this case, it executes an SQL script against a database whose address and credentials are stored in the system's configuration, and places the returned records in the exchange's outbound message's body:
The courierCode header value is injected into the sql query so as to draw open voucher records only for the specific courier.

```
.setHeader('courierCode').constant(configuration.courierCode)

.to("sql:classpath:sql/openVoucher.sql")
```

Next, the open voucher records are split, so that they may be consumed one by one by the issueVoucherProcessor. When the name of a Camel processor object is given as an argument to a .process() EIP, the processor's process() method is invoked. In our scenario, that method should be sending a POST request to a courier's API in order to receive the code of a new shipment voucher:

```
.log(configuration.name +'Voucher: Processing ${body.size}

                                          voucher requests')

.setProperty('totalRequests').simple('${body.size}')

.split(body(),voucherAggregation)

    .log(LoggingLevel.INFO,logger,'Posting for voucher

                                  for request: ${body}')
    .process("issueVoucherProcessor")

    .filter().simple('${body.voucherCallbackUrl} != null')



        .process('voucherCallbackProcessor')

    .end()

.end()

.log(LoggingLevel.INFO, logger, Successfully Updated

${property.successCount}/${property.totalRequests} requests')
```

A few more things are happening here. A callback is sent to the company's ERP via the voucherCallbackProcessor, to inform that a voucher has been issued. The relevant service is invoked only for those vouchers among the issued, who originally provided a callback url.

Finally, voucherAggregation is a function implementing the Aggregator EIP through the respective Camel interface. It's function is to run on each iteration of the .split() pattern. In this particular instance, it merely increments to the successCount property for each voucher successfully issued.

3.4 Debugging a Camel route

Traditional debuggers are incapable of inserting breakpoints on Camel components within Camel routes as they are laid out in the Java API DSL. The simplest way to insert a breakpoint at any given point within the Camel route as to, for example, inspect the exchange's contents, is to use a dummy processor for debugging purposes. This debugProcessor (the component instance is lowercase) would look like this:

```
package com.existanze.services.couriers.util

import org.apache.camel.Exchange

import org.apache.camel.Processor

import org.springframework.stereotype.Component

@Component

class DebugProcessor implements Processor{

   @Override

   void process(Exchange exchange) throws Exception {

      def x=10

   }

}
```

A breakpoint could then be set on def x = 10, whose scope is limited to within the Component and serves no other purpose than provide a statement for the debugger to break on.

4 Introducing Enterprise Integration Patterns as a pattern language

4.1 Messaging Terminology variants

So far an effort has been made to utilize the original terminology to describe the various architectural patterns. In the context of JMS, utilized by Camel, or other Messaging Systems, however, various different yet related terms are used. Here is a table, adapted from the Enterprise Integration Patterns book [7], that will aid in putting all those terms in their proper context:

Enterprise Integration Patterns	**Java Message Service (JMS)**	**Microsoft MSMQ**	**WebSphere MQ**
Message Channel	Destination	MessageQueue	Queue
Point-to-Point Channel	Queue	MessageQueue	Queue
Publish-Subscribe Channel	Topic	——	——
Message	Message	Message	Message
Message Endpoint	MessageProducer, MessageConsumer		

Enterprise Integration Patterns	**TIBCO**	**WebMethods**	**SeeBeyond**
Message Channel	Topic		Intelligent Queue
Point-to-Point Channel	Distributed Queue		Intelligent Queue
Publish-Subscribe Channel	Subject	——	IntelligentQueue
Message	Message	Document	Event
Message Endpoint	Publisher, Subscriber	Publisher, Subscriber	Publisher, Subscriber

Figure 7 EIP Terminology in various Messaging Systems

4.2 Structure of a pattern

A pattern language, such as that put forth in the Enterprise Integration Patterns book, is constituted of patterns. The individual patterns of this particular pattern language adopt the following structure:

- Name – An identifier indicative of what the pattern does

- Icon – Diagrams are frequently used to facilitate communication in the are of software architecture. A visual version of the verbal language is provided to accommodate this common practice.
- Context - The scenarios in which one might come upon a problem that can be solved with the particular pattern.
- Problem - A description of the problem faced, which enables the developer to quickly identify whether a pattern is relevant to their work.
- Forces – Here the –often conflicting -constraints that make the problem difficult to solve are analyzed.
- Solution – A template for pattern implementation.
- Sketch – An visual illustration of the solution template.
- Results – How the solution resolves the forces. New challenges that might be brought about as a result of the implementation of the pattern are also discussed here.
- Next – This section refers to other patterns to be considered after applying the current one.
- Sidebars – They contain relevant technical issues in more detail that strictly necessary for pattern implementation.
- Examples – One or a few examples of pattern application.

4.3 Patterns

In this chapter, a few select Enterprise Integration Patterns descriptions, drawn from Camel In Action and EIP books,, will be presented in a less structured format.

4.3.1 Endpoint

As covered in the first chapter of this part of the thesis, the Endpoint abstraction that models the end of a message channel through which a system can send or receive messages. An Endpoint in Camel is constructed through the use of a Camel component, selected by applying the appropriate locator prefix such as ftp:// or timer:// (seen in a previous chapter) to a camel endpoint's Uniform Resource Identifier (URI). Camel components thus act as endpoint factories.

4.3.2 Content-based Router

The Content-Based Router inspects the content of a message and routes it to another channel based on the content of the message. Using such a router enables the message producer to send messages to a single channel and leave it to the Content-Based Router to inspect messages and route them to the proper destination. This alleviates the sending application from this task and avoids coupling the message producer to specific destination channels.

4.3.3 Message Filter

The Message Filter is a special form of the Content-based Router. It only routes the incoming messages to another channel if certain conditions are met. An implementation of said pattern in Apache Camel is utilized in the Camel route code introduced in the seconds chapter of this part.

4.3.4 Splitter and Aggregator

These are also used in the Camel Router code example of the second chapter of this part of the thesis. A splitter may be used to split multi-line messages e.g. objects, returns from database queries, or lists, into single-line messages that can be processed individually.

An Aggregator may then be used to recombine the messages back into a single message, applying any custom logic that may be required in this process.

Unlike the other routing patterns, the Aggregator is stateful , as it has to store messages internally until their default or custom aggregation conditions are met.

4.3.5 Dynamic Router

In addition to generic Content-based Routers, which are static, dynamic version can be implemented. The core is to allow routing logic to be altered by sending control rules to a designated control port.

4.3.6 Composed Message Processor

A Composed Message Processor is the combination of Splitter, a Router and an Aggregator. Participants operate concurrently and reassemble the replies into a single message. We can say that these. patterns together manage the parallel routing of a message. A generic router in Camel (a Camel route that has not received any extra parameters) is at once an EDA decoupler (an event handler), and a mechanism to chain SOA compliant services via .to() producers. A composed message processor can be constructed in Camel through composition of the aforementioned patterns, an, example of which is presented in the second chapter of this part of the thesis.

5 Apache Camel Components

5.1 Core components

5.1.1 Log

The Log component logs message exchanges to the logging mechanism provided in the project's configuration (The default for Camel Springboot is SLF4J). An example using Camel's Java API DSL [8]:

```
                                              from("timer://${configuration.name}Voucher?period=$
{configuration.pollingPeriod}&repeatCount=-1")

          .log(LoggingLevel.INFO,logger,"${configuration.name} - Voucher: Starting polling
for open voucher requests")
```

An example with a the log component being specified by providing an explicit endpoint URI:

```
from("activemq:orders").to("log:com.mycompany.order?
level=DEBUG").to("bean:processOrder");
```

Example within Spring XML Camel route declaration:

```
<route>
        <from uri="activemq:orders"/>
        <to uri="log:com.mycompany.order?level=DEBUG"/>
        <to uri="bean:processOrder"/>
</route>
```

5.1.2 Bean

The term “beans” has been reused in several different contexts to refer to some kind of special Java object. They are certainly not Plain Old Java Objects (POJOs). Spring beans, in particular, are objects in the Spring Context stored within the JVM runtime, which represent instances of singleton classes.

The Bean Camel component then, in a Spring context, is used to bind such beans to Camel exchanges.

Here is an example of bean instantiation using this component in the Camel DSL style:

```
from("direct:start").bean(ExampleBean.class);
```

And here’s an example of an preexisting bean being invoked with Spring XML Came route declaration. Notice the fully qualified (absolute) classpath is used:

```
<route>
        <from uri="direct:start"/>
        <to uri="bean:com.foo.ExampleBean"/>
</route>
```

5.1.3 Direct

The Direct component provides direct, synchronous invocation of any consumers when a producer sends a message exchange.

This endpoint can be used to connect existing routes in the same camel context e.g. the same Java Virtual Machine.

Java DSL Example:

```
from("activemq:queue:order.in")

        .to("bean:orderServer?method=validate")

        .to("direct:processOrder");
```

Spring XML Example:

```
<route>

        <from uri="activemq:queue:order.in"/>

        <to uri="bean:orderService?method=validate"/>

        <to uri="direct:processOrder"/>

</route>
```

5.1.4 File

The File component provides access to file systems, allowing files to be processed by any other Camel Components or messages from other components to be saved to disk.

Java DSL example:

```
from("file:inbox?charset=utf-8")

  .to("file:outbox?charset=iso-8859-1")
```

Spring XML example:

```
<route>

        <from uri="bean:myBean"/>

        <to uri="file:/rootDirectory"/>

</route>
```

5.1.5 Timer

The Timer component is used to generate message exchanges when a timer fires. You can only consume events from this endpoint.

Multiple task scheduling components are available. They produce timer events that can be used to trigger recurring camel routes via consumer EIPs, or otherwise provide a means of time tracking for local or distributed tasks. The primary ones are scheduler (or it's simpler variant, timer) and quartz.

The scheduler component utilizes the host JDK's timer and is intended for locally tracked tasks that have no need for accuracy, as no provision is made against downtime.

The quartz component uses a database to store timer events and supports distributed timers, and is therefore fault tolerant and suitable for scheduling distributed tasks.

Java DSL example:

```
from("timer://foo?fixedRate=true&period=60000")

        .to("bean:myBean?method=someMethodName");
```

Spring XML example:

```
<route>

  <from uri="timer://foo?fixedRate=true&amp;period=60000"/>

  <to uri="bean:myBean?method=someMethodName"/>

</route>
```

Quartz is note a core component, however, in the interest of cohesion, an example of it's use through the Java DSL will be presented here:

```
from("quartz://myGroup/myTimerName?cron=0+0/5+12-18+?+*+MON-FRI")

   .to("activemq:Totally.Rocks");
```

5.1.6 Validator

The Validation component performs XML validation of the message body using the JAXP Validation API and based on any of the supported XML schema languages, which defaults to XML Schema.

Spring XML example:

```
 <route>

        <from uri="direct:startNullHeaderNoFail"/>

                          <to   uri="validator:org/apache/camel/component/validator/schema.xsd?
headerName=issueVoucherProcessor;failOnNullHeader=false"/>

        <to uri="mock:valid"/>

</route>
```

5.1.7 Other core components

- Browse: Inspect the messages received on endpoints supporting BrowsableEndpoint.
- Class: Invoke methods of Java beans specified by class name.

- Control Bus: Manage and monitor Camel routes.
- Data Format: Use a Camel Data Format as a regular Camel Component.
- Dataset: Provide data for load and soak testing of your Camel application.
- Dataset test: Extends the mock component by pulling messages from another endpoint on startup to set the expected message bodies.
- Direct VM: Call another endpoint from any Camel Context in the same JVM synchronously.
- Kamelet: To call Kamelets.
- Language: Execute scripts in any of the languages supported by Camel.
- Mock: Test routes and mediation rules using mocks.
- Ref: Route messages to an endpoint looked up dynamically by name in the Camel Registry.
- REST: Expose REST services or call external REST services.
- REST API: Expose OpenAPI Specification of the REST services defined using Camel REST DSL.
- Saga: Execute custom actions within a route using the Saga EIP.
- Scheduler: Generate messages in specified intervals using java.util.concurrent.ScheduledExecutorService.

- SEDA: Asynchronously call another endpoint from any Camel Context in the same JVM.

- Stub: Stub out any physical endpoints while in development or testing.

- VM: Call another endpoint in the same CamelContext asynchronously.

- XSLT: Transforms XML payload using an XSLT template.

- XSLT Saxon: Transform XML payloads using an XSLT template using Saxon

5.2 Protocol components and provider components

There are also hundreds of Camel components used to generate endpoints for communication, through specific protocols beyond File (http:, ftp:, jms:, amqp:, etc.), or with the APIs of different service providers (AWS, DropBox, GitHub, etc.).

Appendix: The API economy

SOA, as mentioned previously, was devised as a way to shield interface consumers from changes in the back end. But in a business landscape with an unrelenting drive towards opening up to Business-to-Business integration, and constant competition for driving down the cost through off-the-shelf integration solutions, how are the constantly changing needs of service-consuming partner frontend applications to be accommodated?

The Backends for Frontends (BFF) pattern, now present in mobile apps, Single Page Apps (SPAs) Progressive Web Apps (PWAs), and other modern solutions based on the browser platform in general, was the first foray outside enterprise: APIs perfectly suited to the needs of a prospective frontend, with rationalized data models, ideal granularity of operations, specialized security models etc. This developer-orientation also marked a wholesale departure from the idea of achieving API reusability through stability, a development that goes hand in hand with a bigger investment in API management to lower the maintenance overhead this new stance would entail.[1]

Modern API management solutions create APIs via configuration rather than coding, and the task of creating or changing an API usually takes only minutes. The nature of an easily managed API is simply that it is both defined and controlled by configuration. API management solutions, while complex in their implementation and often costly as proprietary offerings, render the maintenance of non-stable consumer-oriented APIs practical.

APIs are always designed to be attractive to the intended consumer, and they change as the needs of the consumer change. Service interfaces, in contrast, are generally designed with global cost and stability as the most important concerns. In a car analogy, the API is the race car designed for looks and consumption, and the service interface is the regular car designed for cost and mass production.

The interests of the parties involved in the development and maintenance of APIs are often conflicting. A mobile developer in the employ of an enterprise partner just wants the API consumption to be simple for their particular application. On the other side, the back-end team wants everyone to use the same standardized service interface and data model.

Services are the means by which providers codify the base capabilities of their domains. APIs are the way in which those capabilities (services) are repackaged, productized, and shared in an easy-to-use form.

APIs are controlled, proxy views of the data and capabilities of a domain, optimized for the needs of API consumers. As long as the cost of maintaining proxy APIs remains low, they can be used to render a domain in multiple forms, optimized for each group of API consumers. In big enterprises offering cloud services, a scheme comprising three tiers of APIs – public, partner, and internal – is very common. The availability of the first two tiers provide the opportunity for collaboration among loosely related parties. The resultant market-like ecosystem has been dubbed the “API Economy”[2].

Orchestrated microservices riding on container-based “serverless” cloud infrastructure, “Agile” software delivery models, and consumer-oriented partner APIs exposed through advanced, proprietary service management solutions, aimed at 3d party developers, are the enablers of today's API economy. These developments are all, however, beyond the scope of this thesis. For illustration purposes, a few examples of what the “API Economy” entails are quoted below[1]:

“The API economy emerges when APIs become part of the business model. Public and partner APIs have been strategic enablers for several online business models. For example, Twitter APIs easily have ten times more traffic than the Twitter website does. The company’s business model deliberately focuses on Tweet mediation, letting anyone who wants to do so provide the end-user experience. [...] Another example, Amazon, from the get-go, chose to be not only just an Internet retailer but also a ubiquitous merchant portal. Amazon’s merchant platform is deliberately built on APIs that allow easy onboarding of new merchants. APIs as business network enablers aren’t new. Banks have built payment infrastructures and clearinghouses based on well-defined APIs for decades. Modern APIs, however, are built explicitly for an open ecosystem[...].”

Paypal and Stripe are two other wildly successful examples of businesses whose entire business model rests on the API Economy. Uber can also be cited as example of a physical service, the majority of the software stack of which comprises of glued-together partner APIs

In closing, it can be argued that the emergence of this late developmental trend, while departing significantly from the original vision for SOA, nevertheless does credit to the promise of reusability the idea of web service orientation has borne since it's inception.

References

[1] Dennis Ashby, Claus T. Jensen: APIs for Dummies, 3rd IBM Limited Edition, pp. 5 - 9. John Wiley & Sons, Inc. (2018)

[2] George Collins, David Sisk: API Economy. Tech Trends 2015: The fusion of business and IT. Deloitte Consulting LLP (2015)

[3] David A Chappell: Enterprise Service Bus. O'Reilly Media (2004)

[4] Martin Fowler: Patterns of Enterprise Application Architecture. Addison-Wesley Professional (2002)

[5] Alexandros Gazis, Eleftheria Katsiri: Middleware 101, What to know now and in the future. In: ACM Queue, vol. 20 issue 1 (2022)

[6] Souleiman Hasan, Sean O'Riain, Edward Curry: Approximate Semantic Matching of Heterogeneous Events. ACM International Conference on Distributed and Event-based Systems, Berlin, Germany (2012)

[7] Gregor Hohpe, Bobby Woolf: Enteprise Integration Patterns, pp. 18, 32. Addison-Wesley Professional (2003)

[8] Claus Ibsen, Jonathan Anstey: Camel in Action. Manning Publications (2008)

[9] Pontus Johnson: Enterprise Software System Integration, An Architectural Perspective. Ph.D. thesis. Industrial Information and Control Systems KTH, Royal Institute of Technology, Stockholm, Sweden (2002)

[10] Brenda M. Michelson: Event-Driven Architecture Overview. Patricia Seybold Group (2006)

[11] Cécile Péraire, Mike Edwards, Angelo Fernandes, Enrico Mancin, Kathy Carroll: The IBM Rational Unified Process for System z, Service Specification. IBM Corporation (2007). URL http://deg.egov.bg/LP/soa.rup_soma/tasks/soa_service_qualification_E0D920A6.html

[12] Harihara Subramanian Pethuru Raj, Anupama Raman: Architectural Patterns, pp. 24, 25, 28, 210, 245. Packt Publishing (2017)

[13] Bobby Woolf: Event-Driven Architecture and Service-Oriented Architecture. IBM Software Services for WebSphere, IBM Corporation (2006)

[14] Olaf Zimmermann, Cesare Pautasso, Gregor Hohpe, Bobby Woolf: A Decade of Enterprise Integration Patterns: A Conversation with the Authors, pp. 13–19. IEEE Software, vol. 33 issue 1 (2016)

[15] IBM Cloud Education: What is Middleware? (2022). URL https://www.ibm.com/cloud/learn/middleware

[16] Redhat Technology Topics: What is a Service Registry? (2021). URL https://www.redhat.com/en/topics/integration/what-is-a-service-registry