ΠΑΝΕΠΙΣΤΗΜΙΟ
ΙΩΑΝΝΙΝΩΝ

Enterprise Integration Patterns
Building message-oriented middleware with Apache Camel

Odysseas Neslechanidis
Supervisor: Christos Gkogkos

September 6, 2022

Graduation Thesis

University of Ioannina
Department of Informatics and Telecommunications

Abstract

The term "Enterprise Integration Patterns (EIPs)" refers to a vocabulary of solutions to common problems in the integration of enteprise 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.

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

Examination Commitee

1. Christos Gkogkos
2. John Doe
3. John Smith

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

Contents

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

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 (Enteprise 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 enteprises, 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 organisational similarity of enterprises above a certain scale, switching costs (CITE), preservation of optionality in partnering with software vendors (CITE EIPbook p32), 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 applications'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".
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.

2 Introducing EAI in a organization

2.1 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. (CITE EIPbook p32)

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. (CITE EIPbook p32)

2.2 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 tha come as a consequence of, the prospective type of integration solution. The following categorization has been proposed:

2.2.1 Information Portals

Information portals serve to aggregate information from disparate systems within an organization with the aim of making it more accesible 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.

2.2.2 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.

2.2.3 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 (A service is a well-defined function that is universally available and responds to requests from "service consumers") 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 that is 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).

2.2.4 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 paricular 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 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.

2.2.5 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 Exchange (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 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.

3 The Evolution of Enterprise Application Integration

3.1 Islands of automation and the advent of EAI

The term "Islands of automation" was a popular term intoduced 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 organisations, and contribute to a higher barrier of transactional cost for cooperation across different enterprises. A major part of business operations requires coordination between multiple departments/organisations, 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. (CITE IBM, wikipedia).

In the early days of EAI, the development of custom middleware solutions begun spreading as a practice.

3.2 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 conjuction 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 nevertheless 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 software's requisite topological complexity can become onerous.

3.3 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.

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 events to be directed towards exactly one consumer, and to be kept to be resent in case there is no consumer available ("event queue").

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 the subscriber to process an event before sending the next one. 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 yet 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. (CITE wp Event-driven architecture)
- 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.(CITE archpat p245).
  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". Events groups/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. (CITE Hasan_DEBS_2012)

Enterprise-driven architecture is properly constructed with subscribers that are both stateless and context-free. Each event notification ought to contain just enough details to enable the event handlers to guide the business flow in the intended direction e.g. by selecting among running one of several stateless, functionally autonomous modules, that may or may not be event emitters themselves, or halting.

Event notifications should not provide any additional context. Also, they should not require any kind of dependencies on the in-memory session state of the connected 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 interactions between modules in vertical hierarchies, 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 edge-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.

It is also a point worth making separately, that provisions for reliability such as delivery acknowledgement, 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 messaging infrastructure such as the hub and event channel implementations, and the non-deterministic nature of parallel computation. (CITE archpat p.24, eipbook p.18)

3.4 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, all the while factoring in reliability provisions, is referred to as Service-Oriented Architecture, SOA for short.

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

Component-based architecture, or component-based software engineering, 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 emphasised. To realize this architectural style's potential, the promulgation of Web Service standards becomes instrumental. 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 by which they may be invoked.

Based on this, SOA can be defined as an architectural style that enables a dynamic collection of services to communicate with one another. If the conventions are observed dilligently, the need for an EAI hub and it's accompanying module-specific connectors, database drivers and protocol adapters is theoretically obviated. In common practice, however, and still in accordance with the loose definition given above, a federated hub solution is adopted.

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. It is equivalent to the "bus" design concept found in computer hardware architecture, in this case used to refer to the software infrastructure used to implement a model for communication among independent software services running within networks of disparate and independent computers.

In 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.

The primary aim of service management is, indeed, to facilitate service discovery

and exposure via a Service Registry- 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 describe the available APIs, enable potential users to issue keys automatically (self-subscribe) in order to use the APIs, and provision analytics for both users and providers of the APIs.
Many service management capabilities are, however, commonly relegated to a separate runtime, existing as a service known only to the gateway, called a 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, required.

De facto, therefore, the term "ESB" has 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, marketed as ESBs, commonly provide auxillary capabilities, which have proven to be essential additions for constructing more complex systems making use of said pattern. 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.
(TODO ESB)

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." CITE, TODO archpatbook p.28

Utilizing this approach, if the conventions are observed dilligently, the need for an EAI hub and it's accompanying module-specific connectors, database drivers and protocol adapters is theoretically obviated. Provision still has to be taken service discovery and negotiation),

(TODO eipbook p73, opengroup.org "Having a central mechanism by which all messages are exchanged facilitates monitoring, control, transformation, and security of messages." but why use messages/ asynchronous EDA features in soa in the first place? perhaps keep them separate for now, or make the distinction clear.)

(TODO eipbook p51 a Composed Message Processor is the combination of a Splitter, a Router and an aggregator. A router is a Camel Route, a thing that

takes in an event notification (from() and decides what service-server to invoke dynamically. a router is therefore at once an EDA decoupler (an event handler), and a chainer for SOA compliant message-interfaced services (with url() for web services, being wrapped automatically by camel into message interfaces, or with "processor" (CITE camelia p133) wrappers: esb-style connectors, a away to wrap multiple generic java components into message-interfaced SOA-services.

so camel routes expose event handler routing logic, see camelia p131

a camel route is fire-and-forget if eg the timer event causes no message to finally exit the route, the event handler run is just ineffectual)

(CITE archpatbook 25: SOA is the client/server architecture in which the server is a service: universally available, stateless and context-free, while traditionally in the client/server architecture the participants can be tightly coupled and maintain state.)

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, which is nevertheless efficiently bridged by another core feature of SOA, the Enterprise Service Bus.

-TODO LITMUS

and the establishment of the practice of environment definition via Web Service Litmus tests. (TODO litmus is about

http://deg.egov.bg/LP/soa.rup_soma/tasks/soa_service_qualification_E0D920A6.html

constraining the potential services to a subset that is reusable in the environment in which it is initially developed),

As mentioned previously, a distinctive feature of the SOA style is business centeredness, with components or interfaces aimed at fitting business functions, rules, or goals.

Service-oriented architecture can be implemented with web services or Microservices. This is done to make the functional building-blocks accessible over standard Internet protocols that are independent of platforms and programming languages.

(TODO domain driven design and microservices after loose coupling, reusability emphasis, ESBs. archpatbook p.32) . 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 approach is suitable for the design of middleware as it helps confer flexibility and extensibility to the resulting system, and reduces friction in the operations side, which, as mentioned previously, is an important factor when considering the adoption of integration solutions.

Each SOA system defines it's own environment, by which every service implementation must abide. This is ensured through the provision of a "Litmus Test", that determines whether a given service implementation is correct in it's particular SOA system's context.

A further recommendation of the SOA style is that open standards be used, their use being instrumental in realizing interoperability with different consumer implementations, and location transparency.

The related buzzword

https://en.wikipedia.org/wiki/Service-oriented_architecture#Defining%20concepts

https://en.wikipedia.org/wiki/Enterprise_service_bus

A further property service-orientation promotes is loose coupling between services. SaaS can be considered to have evolved from SOA.

Loose coupling, in addition to enabling the development of distributed architectures composed of programs developed by different teams at different times, allows for domain-driven design to be observed in its constituent parts, which is claimed to increase maintainability and creative cross-domain collaboration.

Domain-Specific-Langueges and Aspect-Oriented Programming can be used to manage the complexity produced by the increased need for isolation and encapsulation that Domain-Driven design necessitates.

https://en.wikipedia.org/wiki/Domain-driven_design

Loose coupling is achieved through transactions, queues provided by message-oriented middleware, and interoperability standards.

Transactions help ensure validity of exchanges, queues enable asynchronicity and load balancing in distributed systems, and interoperability standards provide a common target for the integration of legacy systems (often rendering them network-enabled in the process) and newly implemented services alike.

(TODO benefits of SOA https://en.wikipedia.org/wiki/Service-oriented_architecture#Organizational benefits)

In the messaging approach, provisions for asynchronicity (message buffers, brokers) and arbitrary consumer scaling are made.

(TODO Pure Messaging Integration approach https://en.wikipedia.org/wiki/Apache_ActiveMQ#Usage)

Hybrid Web Service - Messaging SOAs using MOMs is common practice.

(TODO benefits with Microservice-based SOA https://en.wikipedia.org/wiki/Service-oriented_architecture#Implementation approaches)

Microservices is a novel, implementation agnostic approach to SOA, that allows for domain-driven design to be observed (actually they are only loosely related and, in fact, operate at different scopes, as discussed here

https://www.ibm.com/cloud/learn/soa)

https://en.wikipedia.org/wiki/Service-oriented_architecture#Microservices

A service is defined TODO stability and reusability (time investment vs reusability, agility; lowers global cost by shielding consumer from changes to the backend)

services communicate via messages

-the partitioning of an application into a collection of dynamic application services facilitates the choice and the replication of one or more application components/services for scaling. divide-and-conquer

TODO describe SOA

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, in practice it has come to

serve as an essential complement in many EDA-stuctured enterprise systems, with SOA services commonly being wrapped as components triggered by event handlers. This fact, combined with the convenience (TODO duties of integration engines and convenience of them being centralized), has brought about a state of affairs where most productized ESB implementations have come to rely on distributed integration engines (often themselves confusingly being marketed as "ESBs"), introducing a federated hub as a common feature of present-day SOA systems.

(TODO archpat p238 soa vs eda and b2b

SOA is about:

-hierarchy, sequences, chaining: interaction between vertical hierarchical layers of functions packaged as services (synchronicity makes contract specification possible, and contracts are nice for that sort of thing)

(Definition: A contract bounds the service with schemas, a clear message exchange pattern, and policies. Policies define the QoS attributes, such as scalability, sustainability, security and so on.)

-request-reply: send question wait for answer; suitable for human-computer interfaces; made possible for reasons above

-transactionality; because information is passed through the response, transaction atomicity (complete in it's entirety or having no effect) can be ensured through tracking the sequence of changes

-testing: merely a matter of ensuring each chained interaction is performed successfully within contract's policy boundaries.

EDA is good for

-operations where human workflow is involved, as it's async and contract-free

-B2B: shifts much of the responsibility of control-flow away from the event source, distributes/delegates it to event handlers. have them possibly trigger a service and voila! you get a decoupled architecture, fit for B2B meddling and hybridizable with SOAs within (say, triggered service acts as a SOA client, sends callback/event notification when completed to event broker). Though there's still the semantic coupling issue and APIs rule the world now. but apis are not SOA-conforming services either (e.g. ignore Service statelessness principle).

-Testing of the systems with EDA is not easy due to the asynchronous nature of the processing (and lack of concommitant contracts).

-"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 event mediator topology, there exists a relatively tight coupling between the event mediator and event processor.

-performance: asynchronicity makes data parallelism-process cloning possible

-scalability: due to both of the above

BOTH EDA and SOA ensure agility, are meant to shield handlers/consumers from changes in the producer backend. EDA through thorough loose-coupling

with event mediator-processor coupling being the limiting factor, SOA conditional on honoring the contract-specified QoS policies (stability).

"So, the rule of thumb to be followed while designing architectures for organizations is use loose coupling whenever possible and use tightly coupled architectural options only if required."

"

TODO archpatbook p199 SOA

(TODO add to EDA pros: archpatbook p29

)

SOA is synchronous request-response as opposed to asynchronous fire-and-forget: pull information via request and demand their reception as opposed to push information and go home. SOA is puller-driven, client is a control-freak: selects exactly which services to invoke, waits for response, uses response

EDA is driven by each handler to whom the event notifications are unidirectionally distributed/delegated: trigger a service or choice, or do nothing: the B2B partner (perhaps) gets to decide.

there does exist async request-response, but is beyond the soa convention

SOA: easier to chain service invocations hierarchically. this produces systems with chained service sequences that are not really meant to be accessible by B2B partners as they forfeit much control, having to abide by contract-specified policies that are business-coupled and conceived to support the synchronous paradigm. QoS attributes, such as scalability, sustainability, security and so on.). it's also good for testing, as it's a matter of meeting the contract.

eda: the client is less involved: sends event, which trigger different services based on which handler catches them, the operation is asynchronous

event mediator (queue) better for b2b as it's inherently loosely coupled, clients need only act as emitters

TODO For ESB, see archpatbook p25, 351, 252, eda vs soa.pdf

the two above, combined in an integration runtime, with added logic for mediation/obscuring between the uniform exposure protocols and data formats, and the legacy formats and protocols (or informal specs) of the integrated applications, constitute an ESB.

The ESB is the integration middleware for any service environment, where the message is the basic unit of interaction between services. An ESB is lightweight compared with previous middleware solutions, such as the EAI hub. The ESB is lightweight because it obviates the need of using custom-made connectors, drivers, and adapters for integrating processes/applications, data sources, and UIs.

integration engines of today allow for the implementation of different patterns that are more lightweight and decentralized. no longer is runtime cloning of the entire integration engine required for scaling infrastructure, nor is there hard need for a dedicated team with deep expertise in the particular SOA solution. how this is achieved:

-

TODO where does oauth fit?

The gateways can be supplemented with web portals that describe the available APIs, enable potential users to issue keys automatically (self-subscribe) in order to use the APIs, and provision analytics for both users and providers of the APIs.

TODO "Increasingly, more modern systems of record already provided an HTTP-based interface that only needed controlled exposure using the exposure gateway. The integration runtime was only required when more complex integration took place, such as more unusual protocols, data formats, compositions of multiple requests—or perhaps in cases where transactionality was needed.

The introduction of an API management layer led to the obvious question: What now is the ESB? Many had come to see the integration runtime and the ESB pattern as one and the same. But in fact, if the ESB pattern is all about exposing services and APIs, then the boundaries of the pattern really include both the integration runtime and the exposure gateway, and in some cases just the gateway. However, due to the ESB's incorrect association with the integration runtime, we have to accept that this is not how the ESB term is typically used."

back-end-for-front-end (BFF) pattern (present in mobile apps/ SPAs) 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.

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. Services, in contrast, are generally designed with global cost and stability as the most important concerns. In the car analogy, the API is the race car designed for looks and consumption, but the service is the regular car designed for cost and mass production.

, a mobile developer just wants it to be simple for her particular app. On the other side, the back-end team wants everyone to use the same standardized service and data model.

APIs are controlled (proxy) views of the data and capabilities of a domain, optimized for the needs of API consumers. As long as it's dirt cheap to create and maintain proxy APIs, you can use them to render a domain in multiple forms, optimized for each group of API consumers. (After all, you probably want to give external partners a different view of your capabilities from the view your internal developers have

"SOA emerged as a means of shielding service consumers from changes in the

back end. But who protects the service providers from the churn of changing needs in omni-channel front-end solutions?"

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

"

from then on, public exposure of partner APIs has steadily become more common, providing the opportunity for collaboration among loosely related parties. The resultant market-like ecosystem has been dubbed the "API Economy"

3.5 The API economy

"lightweight and decentralized. no longer is runtime cloning of the entire integration engine required for scaling infrastructure," and integration can be performed wihout "hard need for a dedicated team with deep expertise in the particular SOA solution". APIs are no longer archaic hieroglyphs exposed by monolithic cobweb-ridden integration runtimes, left to accrue technical debt, they are part of outward facing platform offerings (API productization), that along with the -necessary to reduce maintenance complexity in todays's Agile software delivery model - API management capabilities, are core to the business strategy of many enterprises instead. orchestrated microservices and consumer-oriented partner APIs aimed at 3d party developers are the enablers of today's API economy.

"

The API economy emerges when APIs become part of the business model. Public and partner APIs have been strategic enablers for sev- eral online business models. For example, Twitter APIs easily have ten times more traffic than the Twitter website does. The com- pany's business model deliberately focuses on Tweet mediation, letting anyone who wants to do so provide the end-user experience. CHAPTER 1 Introducing APIs 7 These materials are © 2018 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.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 (internal or external), not for closed private net- works. Furthermore, the consumption models for APIs are standardized with a focus on ease of consumption rather than ease of creation.

"

TODO MENTION paypal, stripe as examples of API productization, Uber as example of basing a business on partner APIs

API layers (mentioned in us-fsi-api-economy.pdf): APIs exposed to accelerate new service development built upon them

The emergence of this last developmental trend arguably does credit to the promise of reusability the idea of web service orientation has borne since it's inception. TODO democratisation of industry, heightened pace of disruption (CITE forbes article)

TODO SaaS perhaps deserves a mention here

TODO check esiarchapproach As a technology, EAI typically refers to non-intrusive application integration techniques aimed at creating loosely coupled enterprise software systems. Message brokers, adapters, process automation tools, and similar modern products are hallmarks of the EAI technology.

)

4 Messaging in practice: Message Oriented Middleware

"The Java EE programming environment provides a standard API called JMS (Java Message Service), which is implemented by most MOM vendors and aims to hide the particular MOM API implementations; however, JMS does not define the format of the messages that are exchanged, so JMS systems are not interoperable." (semantic coupling)

(TODO EIPBook p72 on why messaging)

(TODO read EIPbook forewords on SOA, asynchronous messaging of self reliant systems being the point of integrated systems as opposed to n-tier codependent distributed systems, conferring benefits of request throttling and load balancing but increasing complexity; this fact makes asynchronous messaging a promising approach, and this has informed the direction of this thesis)

(TODO SOA is enabled via messaging. Time to get deep into it)

("The message queue paradigm is a sibling of the publisher/subscriber pattern" CITE wp Message Queue)

Apache Camel is a framework for building MOM middleware. More generally, it aspires to enable integrations designed around the Enterprise Integration Pattern (EIP) vocabulary. In addition to native support for ActiveMQ and other message brokers via JMS, 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 technologies supported as Extensions.

Examples from Java OOP, JMS messaging, Camel.

- Common architectural elements of MOM systems (request databases/message buffers, aggregators, api consumers, services).
- Intruduction to messaging, key problems it solves (separation of concerns, decoupling etc; use analogues from different domains e.g. URI barcodes etc) Identifier vs Locator disambiguation

Message passing implies URLs (Uniform Locator of Resources) Uniform Identifier: as in URI: 23-digit barcode form unique id of a thing whose location or

mode of access is not defined plus Resource Locator:

-Locator as in http:// (mode of access) which implies the rest is an address.

-Resource as in shared-nothing, volatile object: Evoke the same operation (e.g INSERT) twice, don't expect it to be the same function call because the message-passing abstraction hides (therefore isolates) underlying state changes.

Eg. a necessary rerouting of a request because a node is down does not concern the message sender. Drawing out the metaphor, URI = class interface, URL = address of volatile object

- Message buffer considerations: When full: - Block the sender (deadlock risk) - Drop future messages (producer-consumer problem; unreliability), Asynchronicity and concurrency: gotta have both! + friends
- Services as components of monoliths vs distributed systems. Messaging in OOP vs JMS topic, queue schema. Compare and contrast, justify differences by comparing problem domains.

  Pros: Shared--nothing, all the loose-coupling stuff

  Cons: overhead as arguments need to be copied and transmitted to the receiving object

  - https://en.wikipedia.org/wiki/Message_passing#Distributed_objects

Part II
The Enterprise Integration Patterns (EIP) vocabulary

("The commonalities between messaging systems (in terms of capabilities and architecture) have been captured in a platform-independent fashion as enterprise integration patterns (a.k.a. messaging patterns). [3] " CITE wp Enterprise Messaging System)

TODO esiarchapproach p23 on essential EAI technologies

TODO archpatbook p350 Intermediate routing pattern as an alternative to embedding routing logic in chained services

4.1 Messaging

TODO archpatbook p125

Message (and exchange), pipes and filters, router, translator, endpoint

4.1.1 Message channel

point2point, publish-subscribe, channel adapter, message bus

4.1.2 Router

filter, splitter, aggregator, broker

4.1.3 Message endpoint

consumers: polling, event-driven, message dispatcher

Part III
Apache Camel: EIPs in action

"In Camel, DSL means a fluent Java API that contains methods named for EIP terms."

"Camel isn't an enterprise service bus (ESB" (CITE camelia p102)

5 Introduction

Talk about Camel being a tool for working with EIPs, what it's built on, tooling eg IDEs and frameworks that support it, companies that invest on it and custom products eg JBoss

6 Terminology

A Component is a factory for creating Endpoint instances.

Processors are necessarily implementations of the Camel Processor interface. Apart from that, services differ from processors in that they are meant to be used procedurally, called directly from within other methods, whereas processors are integrated in a program's flow through message-based integration, which is what Camel is built for.

This makes services the proper abstraction for program-wide configuration injection (e.g. PartnerManagementService) and a feasible one for utilities for which there is no expected need for integration with remote components (e.g. RequestHandlingService).

- Server as persistence hub, interacting with client services via library calls with "direct:" route chaining.
- Split, choice and aggregator branching and joining.
- "from:"-driven abstract pollers with different implementations per instance. How can Camel help adapt towards a more scalable microservice variant?

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.

declarative DSLs/xml

(TODO https://martinfowler.com/dsl.html)

Examples from declarative vs procedural DSLs (e.g. shell vs guile scripts), Spring traditional remoting vs Camel, timer/from components introduced here, as an example of how Camel departs from older integration methods..

- Producer, consumer properties per component.
- Seamless remoting via ".to()" chaining. Direct ProducerTemplate calls.
- Declarative programming advantages, mention drag-and-drop services (e.g. redhat's integration product).
- Machine readable markup vs Camel's DSL hack.

7 Apache Camel field study

Examining a 3PL logistic company's middleware stack

References

[1] Christos Gogos, Angelos Dimitsas, Vasileios Nastos, and Christos Valouxis. Some insights about the uncapacitated examination timetabling problem. In *2021 6th South-East Europe Design Automation, Computer Engineering, Computer Networks and Social Media Conference (SEEDA-CECNSM)*, pages 1–7. IEEE, 2021.