



**ΠΑΝΕΠΙΣΤΗΜΙΟ
ΙΩΑΝΝΙΝΩΝ**

Enterprise Integration Patterns Building message-oriented middleware with Apache Camel

Odysseas Neslechanidis
Supervisor: Christos Gkogkos

May 20, 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 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.

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

Examination Committee

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

All Rights Reserved ©

Contents

I	The language of Enterprise Application Integration (EAI)	5
1	Introduction	5
2	Islands of automation and the advent of EAI	5
3	Challenges and approaches to integration	6
4	Integration in a Service-Oriented Architecture	7
5	Messaging in practice: Message Oriented Middleware	8
6	Enterprise Integration Patterns	9
II	Apache Camel: EIPs in action	9
7	Introduction	9
8	Terminology	9
8.1	Flow-based programming,	10
8.2	declarative DSLs/xml	10
9	Apache Camel field study	10

Part I

The language of Enterprise Application Integration (EAI)

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), CMS (Content Management System), BI (Business Intelligence), WMS (Warehouse Management System) (TODO Streamline acronyms: some are software-specific, others not so) and serve to automate every business need of modern enterprises, from its 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 (TODO add footnote), preservation of optionality in partnering with software vendors (EIPbook p32), as well as other adjoining business considerations, have hindered their more widespread adoption. This has necessitated the systematic study and development of solutions for Enterprise Application Integration, among which message-oriented middleware has stood out as one the most promising.

2 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.

(TODO INNEFICIENCIES of isolation and drawbacks of adhoc point2point solutions)

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.

The employment of Domain-driven design, in recognition of the maintainability and extensibility benefits domain-expert input in the refining of the software's domain model confers, is a fact further complicating the effort of business software consolidation.

To this day, the introduction of a (TODO define middleware) middleware stack remains a very common business need.

3 Challenges and approaches to integration

Prior to engaging with the technical aspects of Enterprise Application Integration, it is necessary to consider a set of social, organizational and [TODO] that the development and adoption of such a solution might 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. (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.

(Integration challenges p.32)

(TODO resolve fuzziness of the term integration as described in the EIP-book)

(TODO read EIPbook forewords on SOA, asynchronous messaging of self-reliant systems being the point of integrated systems as opposed to n-tier code-dependent 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 EIPbook p64 on Integration Approaches, with messaging as the most promising one)

4 Integration in a Service-Oriented Architecture

(TODO p34 on SOA and other types of integration, p37 Distributed business process vs SOA is the existence in the first case of a business process management component that manages the execution of a business function across multiple existing systems, has the business process encoded inside of it, e.g. serverx. ServerX in turn communicates in a soa manner with the exn_api (exn_db), lv_api and odoo api services. Also, since the introduction of courier service accessed directly from clients we have no longer a single process business process management service, so perhaps it could again be called SOA.)

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. CORBA.

(TODO write about older approaches: https://en.wikipedia.org/wiki/Service-oriented_architecture#Organizational_benefits)

The related buzzword

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

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

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-Languages 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)

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.

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

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

5 Messaging in practice: Message Oriented Middleware

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

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).
- Introduction 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

6 Enterprise Integration Patterns

le jus de cette affaire

Part II

Apache Camel: EIPs in action

7 Introduction

not again!

8 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.

8.1 Flow-based programming,

8.2 declarative DSLs/xml

Examples from declarative vs procedural DSLs (e.g. shell vs guile scripts), Spring traditional remoting vs Camel, timer/from components introduced here.

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

9 Apache Camel field study

Examining a 3PL logistic company's middleware stack

References

- [1] Christos Gogos, Angelos Dimitzas, 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.