AsyncMDSL: a domain-specific language for modeling message-based systems

Candidate: GIACOMO DE LIBERALI

Supervisor: Prof. Dr. OLAF ZIMMERMANN
University of Applied Sciences of Eastern Switzerland
(HSR FHO), Rapperswil, Switzerland

Co-supervisor: Prof. Dr. ANTONIO BROGI

A thesis presented for the degree of
MSc in Computer Science

UNIVERSITÀ DI PISA

Department of Computer Science
University of Pisa
24 July 2020

Abstract

Different solutions exist to standardize how RESTful APIs are described, such as OpenAPI or RAML. Those solutions do not fit, however, in describing message-driven systems. AsyncAPI is an emerging specification — started as an adaptation of OpenAPI — that can model asynchronous APIs and is thus suitable for describing message-based systems. We intend to present a new, more expressive domain-specific language that derives its abstract syntax from the state of the art patterns and concepts described in the Enterprise Integration Patterns book by Gregor Hohpe and Bobby Woolf. AsyncMDSL, presented in this project, aims at modeling message-driven systems using a human-friendly language, allowing a concise yet expressive representation. A converter that produces enriched AsyncAPI documents starting from AsyncMDSL is also presented, which ensures that AsyncMDSL can effectively generate scaffolding code exploiting existing tools available on the market.

Contents

Chapter 1

Introduction

1.1 Context

Event processing is becoming the paradigm of choice in many enterprise and reactive applications. Enterprise-grade solutions benefit from the loose coupling asynchronous event-driven architectures offer, and reactive systems rely on messages to react to changes. Standardize how to describe asynchronous systems denotes a critical factor in promoting style guidelines and team cooperation. Every framework proposes its language, often incompatible, to face similar underlying problems from a different perspective. A modeling language that abstracts product-specific details, such as data representation formats or transport protocols, can be used to describe event-driven systems in a consistent set of rules, yet offering the possibility to target different frameworks. A system's description created with a common modeling language can be fed to a transformation algorithm that produces a product-specific representation, using the base model as a single source of truth. In this thesis, we focus on creating a modeling language to describe asynchronous APIs, favoring a design-first approach, which helps in determining an effective design. Delivering APIs which share common behaviors, patterns, and consistent interfaces ease the work of both software architects and APIs consumers. The text-based modeling language we propose, in form of a domain-specific-language, extends the Microservice Domain-Specific Language (MDSL) [59] [29], used to describe synchronous APIs. It enjoys an abstraction both over data formats and transport protocols, offering a sound basement upon which an extension to model asynchronous APIs can be built. Existing specification languages, like OpenAPI [26] or RAML [38], employ well-known formats (such as JSON or YAML) but are bound to RESTful

services, and are not suitable to describe asynchronous systems. A modeling language that targets those systems is AsyncAPI [24], an emerging specification inspired by OpenAPI. AsyncAPI favors simplicity, offering a plain interface to model APIs, which, in an asynchronous event-based context, take the form of message channels. Message channels are one of the pillars of event-driven and message-driven architectures. The Enterprise Integration Patterns (EIPs) book [23] offers an established set of recurring solutions that can be employed in designing message-driven systems. It also introduces a naming convention for components and a clear description of how they interact. A modeling language suitable for describing asynchronous APIs can exploit these patterns as language concepts, as they are common to all frameworks dealing with message-driven systems. AsyncAPI targets event-driven systems that slightly differ from message-based systems adopted by Enterprise Integration Patterns, preventing their proper representation. The Reactive Manifesto does a great job defining the difference between the event-driven approach chosen by AsyncAPI and the message-based architecture lying under EIPs:

> [...] A message is an item of data that is sent to a specific destination. An event is a signal emitted by a component upon reaching a given state. In a message-driven system addressable recipients await the arrival of messages and react to them, otherwise lying dormant. In an event-driven system notification listeners are attached to the sources of events such that they are invoked when the event is emitted. This means that an event-driven system focuses on addressable event sources while a message-driven system concentrates on addressable recipients. [54]

Our domain-specific language targets the modeling of message-based systems, focusing on creating common models that can subsequently be used to generate product-specific representations. The transformation phase presented in this thesis focuses on generating AsyncAPI compliant documents, enriching them with patterns that can not be modeled using that specification, allowing us to benefit from the available tools and community that AsyncAPI has gained over the last couple of years, yet exploiting the expressiveness and features our new language offers.

1.2 Vision

In this thesis, we present a domain-specific modeling language designed to describe message-based systems. AsyncMDSL, our proposal, is characterized by an expressive abstract syntax inspired by the Enterprise Integration Patterns, allowing an integration architect to find a close correspondence between the concepts defined in [23] and AsyncMDSL's language constructs. The value of a modeling language, especially in a design-first approach, goes beyond the mere system description. Once a system's model is available, one would expect some tools that permit the execution of the model, or that can extract some documentation to illustrate the available interfaces. In our asynchronous APIs context, this execution could be realized by generating scaffolding code that targets a technology stack (such as Node.js or Java and a concrete protocol, such as AMQP), and that contains all the information represented in the model. Given that the automatic generation of code or documentation is out of the scope of this work, and that several tools already exist, we chose to exploit AsyncAPI's available tooling to achieve the objective. Instead of taking care of generator templates, once a model created with our language has been modeled, our conversion algorithm will create the corresponding AsyncAPI compliant file, which will be used by AsyncAPI's tools to generate scaffolding code and documentation.

After the consolidation of some terminology, in the following chapter, we provide an overview of related work in the field of modeling integration solutions with Enterprise Integration Patterns [23] and a brief introduction to existing modeling frameworks and specifications as Apache Camel and AsyncAPI. We then proceed listing the requirements that our domain-specific language has to fulfill in order to model message-based systems effectively. Once requirements have been introduced, we dive into the DSL's technical implementation, providing some background information on the base language that we extended, and we preset the features supported by our new language. Finally, we showcase a classic example first described in the original EIPs book, and we demonstrate how it can be modeled with our language. We later conclude by specifying missing features and critical points of our proposal and the future work that can derive from it.

1.3 Terminology

We introduce a set of terms used all across this thesis to fix concepts unambiguously.

- *Application*: an application is any computer program capable of both producing or consuming a message. It may be written in different programming languages as long as they support the protocol used by the broker.
- *Message*: a message is a piece of data exchanged via a channel between brokers and applications. A message contains a payload and may also contain headers. The payload contains application-specific data that must be serialized into a format (such as JSON).
- *Message Broker*: a message broker is a piece of infrastructure in charge of receiving messages and delivering them to those who have shown interest, exploiting a protocol.
- *Message Producer*: a message producer (or publisher) is an application, connected to a broker, that is creating messages and addressing them to channels. A producer may be publishing to multiple channels depending on the server and the protocol.
- *Message Consumer*: a message consumer (receiver, or subscriber) is an application that connects to a broker via a supported protocol and consumes messages from one or more channels, depending on the broker and the protocol.
- *Channel*: a channel is an addressable component, exposed by a broker, for the organization of messages. Producers send messages to channels, and consumers consume messages from channels. A broker may support many channel instances allowing messages with different content to be addressed to different channels.
- *Protocol*: a protocol is the mechanism by which messages are exchanged between an application and a channel. Example protocols are AMQP, MQTT or Kafka.

Chapter 2

Related Work

2.1 Academic Literature

Parameterisable EAI patterns

[45] and [42] introduce the notion of Parameterisable EAI pattern (PEP). They allow integration architects to reuse the guidelines captured through EIPs and configure (i.e., parameterize) them so that they fit a specific integration problem, after passing a transformation phase. In the transformation algorithm proposed in [43], PEPs act as a platform-independent model that serves as a base for generating executable artifacts. GENIUS, a visual editor for modeling PEPs, is used to model an integration solution that can eventually be transformed into various target platforms (such as Apache Camel).

EIPs to BPEL

[58] exploits PEPs — extending [42] — to support other types of patterns categories, such as Message Routing and System Management. It leverages a mapping between Enterprise Integration Patterns and BPEL processes [13] and illustrates an example of how this mapping can be accomplished to finally generate a BPEL model that can be deployed and executed on the ActiveBPEL engine. [39] presents another example of formal mapping between integration semantics represented by EIPs and BPMN syntax and its execution semantics.

Modeling examples

[31] evaluates EIPs' implementation inside Apache Camel [3] and proposes an approach able to generate executable Java code from the modeled integration scenario. [44] presents an example of how EIPs can be realized inside WebSphere, exploiting the built-in XML visual editor.

Even if visual representations could help define components interactions, this benefit slowly vanishes as the model complexity grows. A graphical representation of a system should be eventually generated starting from a specification document, rather than the other way. A visual modeler should, instead, be combined with the underlying language.

2.2 Existing Modeling Frameworks

Several solutions currently exist to model RESTful APIs, such as OpenAPI [26] (formerly Swagger [51]) or RAML [38], but none of them are designed to describe message-based systems. Our AsyncMDSL language is not the first attempt to model message-driven systems, but rather an alternative proposal of what the market offers: AsyncAPI, which has overlapping but different design goals. AsyncAPI aims at defining models highly compatible with OpenAPI, favoring simplicity over expressiveness. AsyncMDSL, instead, focuses on expressiveness exploiting the domain's native components, EIPs, as language concepts.
Other relevant projects, such as Apache Camel [5] or Mule ESB [27], exist; they are not specification languages to describe messaging systems, but integration frameworks based on known Enterprise Integration Patterns (EIPs) [23]. Both products indeed offer interfaces for EIPs, providing commonly needed implementations and connectivity to different transport APIs. Camel also proposes a domain-specific language to wire patterns and transports together. We decided to present Apache Camel rather than other alternatives products as it is open-source and its fluent API design influenced the syntax of AsyncMDSL's grammar.

2.2.1 Apache Camel

Apache Camel is an Java-based open-source integration framework based on the Enterprise Integration Patterns [23]. It can run as a standalone application, embedded as a library in a Spring [48] project or run natively in a Kubernetes [18] cluster. In a Camel-based application, one can create endpoints and connect these endpoints with routes. An endpoint is an addressable component provided by Camel. Some examples of the supported endpoint technologies are JMS queues, web services, files or FTP servers. The routing engine is core part of Camel: routes contain the flow and logic of integration between different systems. A route is a step-by-step movement of a message from an input queue, through arbitrary types of decision making — such as filters and routers — to a destination queue. Camel provides different ways for an application developer to specify routes. One can, for example, specify route information in an XML file or through *Java DSL*. The Camel's *Java DSL* is a library that offers fluent APIs that simulate the experience of a domain-specific language, yet having a Java syntactic baggage.

```
1  public void configure() {
2    from("queue:c").choice()
3      .when(header("foo")
4        .isEqualTo("bar"))
5        .to("queue:d")
6      .when(header("foo")
7        .isEqualTo("cheese"))
8        .to("queue:e")
9      .otherwise()
10       .to("queue:f");
11 }
```

Listing 1: Example of Camel's *Java DSL*

The Camel documentation compares *Java DSL* favorably against the alternative of configuring routes and endpoints in an XML-based configuration file. In particular, *Java DSL* is less verbose than its XML counterpart. Besides, many integrated development environments (IDEs) provide an autocompletion feature in their editors, thereby easing developers' work. Defining how messages are produced, consumed or routed to endpoints can also be represented exploiting annotations in Java beans (Listing 2),

```
public class Foo {

  @Consume(uri = "activemq:my.queue")
  @RecipientList
  public Bar doSomething(
    @Header("JMSCorrelationID") String correlationID,
    @Body String body) {

    // process the inbound message here
  }
}
```

Listing 2: Example of Camel's bean integration

or by using a REST styled DSL that allows clients to interact with a message broker using standard HTTP verbs such as GET or POST.

2.2.2 AsyncAPI

AsyncAPI is an emerging specification language for defining asynchronous APIs. It was initially developed due to the lack of tooling in the message-driven space [7], where none of existing standards such as [26] and [38] were suitable. This was leading every company that had to deal with message-based systems to create custom solutions to keep code and documentation in sync. AsyncAPI exploits JavaScript Object Notation (JSON) format as specification language and provides a set of tools that range from code to documentation generation. The language emerged thanks to its simplicity; the learning curve is low for a developer coming from other specifications such as OpenAPI, and the popular JSON format is already familiar to a considerable number of developers. The possibility to define a platform-agnostic model is, in part, the same factor that helped the diffusion of OpenAPI, from which AsyncAPI took inspiration. Furthermore, AsyncAPI allows the definition of protocol-agnostic models, avoiding the specification-protocol coupling, as it is happening with OpenAPI and HTTP. As is [26], AsyncAPI clients can understand and consume services without the knowledge of server implementation or server code access. Given a model, indeed, tools provided by AsyncAPI ecosystem allow generation of documentation in different formats, such a browsable HTML website, as well as boilerplate code to quickly get started with a new project.

The specification is capable of grasping several aspects of a message-driven system; it is possible to define:

- channels
- operations available in channels
- Data Transfer Objects (DTOs) schemas
- protocol-specific binding
- servers definitions
- server security policies

It does not allow, however, the definition of endpoints — message produces and message consumers — , which have a radical influence on the behavior of the system. The impossibility to express endpoints, for example stating which are idempotent [1] and for which some sort of state must be kept, could lower the entire model's expressiveness, resulting in a lack of information for users of the services. Another critical factor to consider when adopting a specification language is the ability to define models that use constructs derived from the context where the specification will be applied, in our case messaging solutions. As often happens when designing systems, indeed, the design of a message-driven system benefits from a set of community accepted patterns and conventions. The reference point for message-driven systems is the Enterprise Integration Patterns [23] book, where a set of design patterns for common situations that arise in message-driven designs are defined, together with a naming convention. Those patterns remind the GoF's Design Patterns [21] for what concerns the object-oriented programming. The conjunction between patterns and naming convention uniquely identifies components of a system that can, in this way, rapidly be shared unambiguously. AsyncAPI, however, lacks the expressiveness EIPs can offer, and adopts an abstract syntax that does recall an event-driven system rather than a message-based system, preventing the proper representation of EIPs.

[1] A message consumer is idempotent if it can handle duplicate messages.

Example

AsyncAPI documents are represented using JSON; YAML (YAML Ain't Markup Language [57]), being a superset of JSON, can be used as well to represent any AsyncAPI document. Examples provided in this thesis use YAML as it is more readable and it supports block literals and comments.

```
1  channels:
2    tasks/new:
3      publish:
4        # this description uses block literals
5        description: |
6            Emits when a new task is available to be computed.
7        message:
8          payload:
9            type: object
10           properties:
11             id:
12               type: integer
13               description: The task identifier.
```

Listing 3: AsyncAPI channel definition example

In Listing 3 we define a new channel with the AsyncAPI specification. We declare that exists the channel `tasks/new` (line 2) that produces a message (line 7) which payload is an object with a property of type integer named `id` (lines 8-13). The information contained in this model allow a message consumer to infer only the type of the payload it will eventually receive. A message consumer can not understand whether it will be competing with other message consumers, nor if the received message would require a reply. The semantic of a channel is, indeed, not expressed, and consumers miss crucial information on the proper usage of available interfaces.

Some of the EIPs, such as Correlation Identifier and Message Expiration, can be represented with this specification, while some others, such as Invalid Message Channel or Request-Reply Channel, can not or can be only partially represented. Some patterns can be modeled exploiting the AsyncAPI so-called "bindings". Even if it is protocol-agnostic, AsyncAPI offers a mechanism — a binding — that aims defining protocol-specific information. This allows users to define a general model that can be enriched with additional metadata to make AsyncAPI aware of the actual protocol and/or software topology.

```
1 channels:
2   user/signup:
3     publish:
4       bindings:
5         amqp:
6           expiration: 100000
7           replyTo: user.signedup
```

Listing 4: AsyncAPI AMQP binding

If we consider another example (Listing 4), we notice how we can define a channel with a Message Expiration (line 6) and a Message Reply (line 7). Those information are contained in an AMQP [49] binding object, which means that the current channel is made available by an AMQP broker. If we consider the Request-Reply Message pattern, we can rapidly notice that a Message Consumer interested a message coming from the channel can not understand what the message reply should look like; it only knows that it has to send the reply message through the channel `user.signedup`. Given this situation, the missing information must be reported somewhere, more likely in the `user.signedup` channel's documentation. In Table 1 we provide a summary of the EIPs that can expressed using AsyncAPI and AsyncMDSL.

Table 1: AsyncAPI vs AsyncMDSL EIPs support

EIP patterns		**AsyncAPI support**		**AsyncMDSL support**	
		Model	Semantic	Model	Semantic
Messaging Channels	Point-to-Point Channel	YES	NO	YES	YES
	Publish-Subscribe Channel	YES	NO	YES	YES
	Datatype Channel	YES	YES	YES	YES
	Invalid Message Channel	Protocol-specific		YES	YES
	Dead Letter Channel	Protocol-specific		YES	YES
	Guaranteed Delivery Channel	Protocol-specific		YES	YES
Message Construction	Command Message	NO	NO	YES	YES
	Document Message	NO	NO	YES	YES
	Event Message	NO	NO	YES	YES
	Request-Reply	Partial		YES	YES
	Return Address	Protocol-specific		YES	YES
	Correlation Identifier	YES	YES	YES	YES
	Message Sequence	NO	NO	YES	YES
	Message Expiration	Protocol-specific		YES	YES
Messaging Endpoints	Competing Consumers	NO	NO	YES	YES
	Polling Consumer	NO	NO	YES	YES
	Event-Driven Consumer	NO	NO	YES	YES
	Selective consumer	NO	NO	YES	YES
	Durable subscriber	NO	NO	YES	YES

SWOT analysis

In the next table we also provide a Strengths, Weaknesses, Opportunities, and Threats (SWOT) overview of AsyncAPI. Born in 2017, and primarily maintained by two developers, it is gaining more attention from the community thanks to the similarity it shares with OpenAPI and some big companies supporting it, such as Slack, that mentioned AsyncAPI in the description of some of its APIs[2].

	Positive	Negative
Internal	**Strenghts** • Protocol agnostic • Language agnostic • Powerful tooling • Open source	**Weaknesses** • Can not model API endpoints • Can not model well-known patterns • Does not enforce channels/messages semantics • Readibility
External	**Opportunities** • De-facto community standard • Familiar specification (similar to OpenAPI) and format (JSON/YAML) • Backed by companies (eg. Slack and Salesforce)	**Threats** • Not yet mature and with few competitors • Not supported yet by message oriented middleware

Figure 1: AsyncAPI SWOT analysis

[2]https://github.com/slackapi/slack-api-specs/blob/master/events-api/slack_events_api_async_v1.json

Although it is gaining some popularity, AsyncAPI still suffers from low readability, provoked by the chosen format, as well as a lack of expressiveness: APIs miss some semantic information that could be very useful for their clients. A channel or message purpose needs to be deducted either from comments or domain knowledge. AsyncAPI ecosystem is also not as vast as rival products' ones, such as OpenAPI. Indeed only a few message-oriented middlewares support it (e.g., MuleSoft [27]), and no different real alternatives exist.

The requirements described in the next chapter are focused on highlighting the semantic constructs that our domain-specific language should be able to express, as well as non-functional requirements defined to assess the delivered project's intrinsic properties, such as the specification complexity or its license.

Chapter 3

Requirements

This chapter discusses the requirements of the Microservice Domain-specific Language (MDSL) extension, presented in this project, should cover. Besides the functional requirements described as User Stories, the chapter also presents the non-functional requirements AsyncMDSL has to fulfill. Requirements come from the need for modeling common patterns for asynchronous, message-based systems. They are derived by analyzing AsyncAPI, selected Enterprise Integration Patterns (EIPs), and messaging platforms such as Apache Kafka and RabbitMQ.

3.1 User Stories

User stories (US) [1] — usually employed in agile methodologies [34] — are a popular method for representing requirements using a simple template. Each user story expresses a single functional behavior of a product, and once implemented, it should contribute to its value, independently of the order of implementation. Different templates can be used to denote stories, but the more popular [32] is the original one [1]:

> As a ⟨role⟩, I want ⟨goal⟩, [so that ⟨benefit⟩]

and we as well will be using this template to express our requirements.

3.1.1 US-1: Model a message-based system

As messaging integration architect, I want to specify the API contracts of my asynchronous message-based system, so that message producers and message

consumers are loosely coupled in time dimension and can exchange messages in an interoperable manner.

Variant

As a service provider, I want to make consumers of my message-based system aware of the available interfaces, so that they would autonomously be able to interact with it.

3.1.2 US-2: Integrate with AsyncAPI

As a messaging integration architect, I want the APIs contract of a messaging system described with AsyncMDSL to be converted into AsyncAPI, while maintaining the expressiveness that AsyncMDSL provides. Patterns not representable in AsyncAPI will be inserted as comments in a generated AsyncAPI document. The conversion must target AsyncAPI version 2.0.0.

Variant

As a project manager, I want to exploit existing tooling available on the market to generate code and documentation, so that developers will take less time to build the system.

3.1.3 US-3: Message Channels

As a messaging integration architect, I want to model a Message Channel. Since message channels tend to be static and defined at design time, I need a proper way to make applications that produce shared data have a way to communicate with those that wish to consume it. A message channel could be one, or a combination of, the following channel types:

- Point-to-Point Channel
- Publish-Subscribe Channel
- Datatype Channel
- Invalid Message Channel
- Dead Letter Channel
- Guaranteed Delivery Channel

Each channel type is described as separate user story in the following sections.

US-3.1: Point-to-Point Channel

As a message integration architect, I want to model a Point-to-Point Channel so that the producer knows that each message it will send over this channel type is received by only a single consumer.

US-3.2: Publish-Subscribe Channel

As a messaging integration architect, I want to model a Publish-Subscribe Channel so that producer knows that each message it will send over this channel type is received as a copy by all interested consumers.

US-3.3: Datatype Channel

As a messaging integration architect, I want to model a Datatype Channel so that producers and consumers know the type of the messages' payload that flows through this channel. Messages could contain headers that must be, as well, possible to model.

US-3.4: Invalid Message Channel

As a messaging system administrator, I want to model an Invalid Message Channel, a special channel for messages that could not be processed by their receivers, so that receivers can handle messages they do not consider valid and put them into a monitored channel.

US-3.5: Dead Letter Channel

As a messaging system administrator, I want to model a Dead Letter Channel, a special channel for messages that could not be delivered by the messaging system, so that messages will not be silently dropped, but instead put into a monitored channel.

US-3.6: Guaranteed Delivery Channel

As a messaging integration architect, I want to model a Guaranteed Delivery Channel so that messages are always stored on disk until they are successfully delivered and acknowledged by consumers.

3.1.4 US-4: Messages

As a messaging integration architect, I want to model the messages that will flow through channels so that the semantic of each message is explicit. A message could be one of the following types:

- Command Message
- Document Message
- Event Message
- Request Message
- Reply Message

Each message type is described as separate user story in the following sections.

US-4.1: Command Message

As a messaging integration architect, I want to model a Command Message so that producer knows that it is explicitly invoking a known procedure in another application.

US-4.2: Document Message

As a messaging integration architect, I want to model a Document Message so that producer explicitly transfers a data structure between applications.

US-4.3: Event Message

As a messaging integration architect, I want to model an Event Message so that producer explicitly communicates the occurrence of an event to consumers.

US-4.4: Request-Reply Message

As a messaging integration architect, I want to model Request-Reply messages so that an explicit flow of messages is defined between requester and replier. A pair of Request-Reply messages are used to create a bidirectional data transfer: the Request Message should contain a ReturnAddress to tell the replier where to send the reply. The Reply Message should contain a Correlation Identifier that specifies which request this reply is for.

3.1.5 US-5: Return Address

As a messaging integration architect, I want to model the return address of a message so that a replier can send back the reply to the requester.

3.1.6 US-6: Correlation Identifier

As a messaging integration architect, I want to model the correlation between messages so that a receiver can infer a chain of related messages.

3.1.7 US-7: Message Sequence

As a messaging integration architect, I want to model a message sequence so that I can break data into a message-size chunk. Each message chunk should contain a sequence identification to allow a later reconstruction.

3.1.8 US-8: Message Expiration

As a messaging integration architect, I want to model message expiration, so that once the time for which a message is viable passes, and the message still has not been consumed, then the message will expire and the messaging system's consumers will ignore it.

3.1.9 US-9: Message Endpoints

As a messaging integration architect, I want to model one or more Message Endpoints, so that I can describe clients of a messaging system and their interactions with available Message Brokers.

3.1.10 US-10: Competing Consumers

As a messaging integration architect, I want to identify a channel as capable of managing competing consumers, so that subscribers are explicitly informed. A channel with this attribute is a Point-to-Point Channel with multiple consumers, where only one of them will receive a particular message.

3.1.11 US-11: Polling Consumer

As a messaging integration architect, I want to model a Polling Message Consumer, so that the consumer knows that it has to explicitly make a call when it wants to receive a message.

3.1.12 US-12: Event-Driven Consumer

As a messaging integration architect, I want to model an Event-Driven Message Consumer, so that the consumer knows that it will handle messages as soon as they are delivered to the channel.

3.1.13 US-13: Selective Consumer

As a messaging integration architect, I want to specify the conditions under which a Message Consumer will receive a message, so that it only receives messages that match its criteria.

3.1.14 US-14: Durable Subscriber

As a messaging integration architect, I want to identify a Message Consumer as durable, so that it will avoid missing messages while it is not listening for them. The messaging system will save messages and deliver them as soon as the consumer is listening.

3.1.15 US-15: Message Brokers

As a messaging integration architect, I want to model one or more Message Brokers, specify their protocol (e.g., AMQP, Kafka, or MQTT) and the address under which they will expose the Message Channels, so that clients know how to exchange messages.

3.1.16 US-16: Specify protocol-specific information

As a messaging system administrator, I want to specify protocol-specific information so that the model can be enriched with information about the protocol and/or the software topology. For instance, knowing that the Message Broker will expose channels through Kafka, I could specify the name of the consumer group a Message Consumer belongs to.

3.1.17 US-17: Server security

As a messaging system administrator, I want to specify, for each Message Broker, the authentication mechanism, so that Message Endpoints know how to authenticate. AsyncMDSL will support user-password and API key authentications.

3.2 Non-functional Requirements

Non-functional requirements (NFRs) express criteria used to assess the operation of a system rather than specific behaviors [22], which are instead defined as user stories in above the section 3.1.

3.2.1 NFR-1: Usability

A messaging system architect should be able to understand examples written in AsyncMDSL within 15 to 30 minutes. Following the provided examples and tutorials, he/she should be ready to start creating its model within one hour.

3.2.2 NFR-2: Expressiveness

AsyncMDSL's grammar should represent EIPs using an easy to understand, self-explanatory, syntax, reducing the need for further code documentation.

3.2.3 NFR-3: Reliability

The provided tools should have no crashes nor data losses during the conversion between AsyncMDSL and AsyncAPI. Any grammar constructs that could result in an invalid AsyncAPI document generation must be statically checked to inform the user and provide a possible fix. In case an error occurs during the generation, the user must be informed.

3.2.4 NFR-4: Specification's complexity

The complexity of a specification can be roughly estimated by the number of its grammar rules. The number of AsyncMDSL's grammar rules, considering also the standard MDSL, should not exceed 100 rules.

3.2.5 NFR-5: AsyncAPI conversion time

The conversion between AsyncMDSL and AsyncAPI should last at most two second per channel. The conversion of the AsyncMDSL equivalent of the demo project[1] should complete within two seconds.

3.2.6 NFR-6: Maintainability and supportability

The project should have clear setup instructions and mechanisms to support the maintainability of the code. This mechanisms could include static code analysis (linters), continuous integration and development pipelines, a well-defined git strategy (Gitflow) and tools to generate documentation (eg. Javadoc). The code should be clean and understandable and follow a well-known style convention (eg. Google Java Style Guide) as well as be properly documented.

3.2.7 NFR-7: License

Since the standard MDSL is distributed under the Apache License 2.0, its extension grammar also has to adopt the same license.

[1] https://www.asyncapi.com/docs/tutorials/streetlights

Chapter 4

Language Design and Tool Implementation

AsyncMDSL[1], the language proposed in this thesis, extends the Microservice Domain-Specific Language (MDSL), adding the possibility to model asynchronous message-based systems. From the homepage of the project we can read that:

> [...] MDSL's syntax is inspired and driven by the domain model and concepts of Microservice API Patterns, featuring endpoints, operations, and data representation elements. [59]

Considering the context where AsyncMDSL will be adopted, some of MDSL's concepts needed to shift to best fit message-driven naming convention defined in [23], and avoid terminology confusions. Leveling AsyncMDSL's terminology with the concepts defined in [23], integration architects will find a close correspondence between EIPs and our language constructs.

4.1 Background: Standard MDSL Language

The standard MDSL language, the base grammar of the extension proposed in this thesis, has been developed using Ecplise Xtext [14], a framework for development of programming languages and domain-specific languages. The framework uses ANTLR [36] for generating the parser and provides features such as a typed Abstract Syntax Tree (AST), scoping, unparsing[2] and validation. Xtext uses in-memory EMF [50] models for representing the generated

[1]The full grammar will be soon available at
https://github.com/Microservice-API-Patterns/MDSL-Specification

[2]The process of constructing text from an AST

AST, and we will use Ecore diagrams [19], as well as code examples, to introduce the language structure. Once some background context on the standard MDSL language has been provided, we will introduce the AsyncMDSL grammar extension.

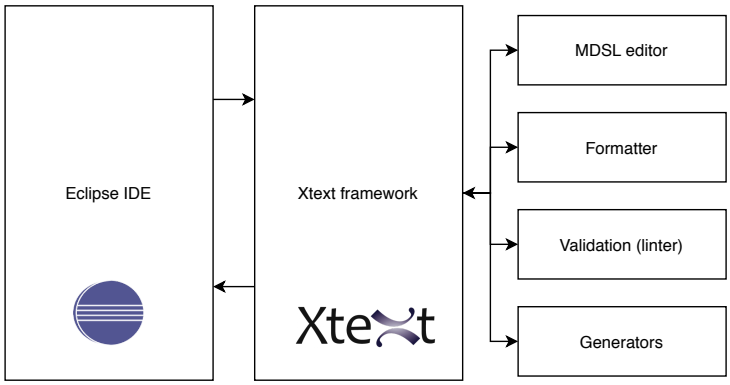

Figure 2: MDSL is distributed as an Eclipse plugin that provides an editor, a linter and a generator (under development) that produces OpenAPI documents.

4.1.1 Language elements

MDSL defines a base structure upon which AsyncMDSL is built on top. It supports the API Description pattern [61], exploiting the concepts of Endpoint, Operation, Client and Provider. An API description features one or more endpoints which, in turn, expose operations that either expect or deliver messages. A message carries a header and a payload.

In the introductory example[3] of MDSL (listing 5), we can already notice many features of the language. We are modeling an API with a single endpoint *HelloWorldEndpoint* that exposes a single operation called *SayHello*. This operation accepts a single scalar string value `"D<string>"` as input, and returns a Data Transfer Object (DTO) called *SampleDTO* as output. The character `"D"` that comes before the type `"string"` represents one of the possible Element Stereotypes defined in [61], and indicates the role of the parameter content: `"D"` data, `"ID"` identifier, `"L"` link or `"MD"` metadata. In addition to the endpoint type *HelloWorldEndpoint*, an API client and an

[3]https://microservice-api-patterns.github.io/MDSL-Specification

API provider are abstractly defined. Note that no protocol information are supplied, since MDSL is not bound to any transport protocol.

```
API description HelloWorldAPI

  data type SampleDTO { "id": ID<int>, "message": D<string> }

  endpoint type HelloWorldEndpoint
  exposes
    operation SayHello
    expecting payload D<string>
    delivering payload SampleDTO

  API provider HelloWorldAPIProvider
    offers HelloWorldEndpoint
    at endpoint location "https://microservice-api-patterns.org"
    via protocol RESTful_HTTP

  API client HelloWorldAPIClient
    consumes HelloWorldEndpoint
```

Listing 5: MDSL basic example

In this basic example, we can understand the structure of any MDSL document, a single root object containing different components:

- datatypes, the definitions of Data Transfer Objects (DTOs)
- endpoints
- providers
- and finally clients.

The concrete syntax is inspired from the Microservice API Patterns defined in [61]. A minimal MDSL document should include endpoint addresses, operation names, structure and meaning of the request and response message representations.

4.1.2 Data types

MDSL provides a JSON-like syntax to express rich data models, either as anonymous objects or as named entities that can be reused. The Structure Patterns from [61] compose the base of the type system. The Representation Elements [61] supported are:

- Atomic Parameter: a single, primitive data element (eg. a scalar value)

- Atomic Parameter List: a representation of multiple primitive data elements
- Parameter Tree: a hierarchical data structure that can define nested Representation Elements
- Parameter Forest: one or more nested data structures that cannot be represented well in a single Parameter Tree

As introduced in [29], the Identifier-Role-Type (IRT) triples `"name": D<string>` defines a single, primitive data element, the AtomicParameter pattern, where:

- the identifier `"name"` corresponds to a variable name
- the role `"D"` can be any element stereotype defined in [61]: `"D"` data, `"ID"` identifier, `"MD"` metadata or `"L"` link. Others stereotypes, such as *RequestBundle* or *Pagination*, can be specified prefixing the data type with `<<Stereotype>>`, e.g.:

  ```
  <<Embedded_Entity>> "customerId": ID<int>
  ```

- the type `<string>` is either a basic type, such as `string`, `int`, `long`, `double`, or a nested structure.

Nesting is expressed in a block-like syntax: $\{\ldots\{\ldots\}\}$ and constitutes the Parameter Tree pattern. Since MDSL is designed to support agile modeling practices [34], it offers partial specification as first-class language concept [59]. If a type of an object is indeed unknown, a generic parameter can be used, as shown in line 2 of Listing 6. Items collections are represented by appending some modifiers as $*$ or $+$ to a type definition. The $*$ modifier turns a type definition into a collection of zero or more elements, while $+$ into a collection with at least one element. Finally, parameters' optionality can be modeled by ? modifier, that indicates the parameter is optional. Parameters are mandatory by default, but the ! modifier can be used to explicitly ensure that a value will be present.

```
1 data type SingleNodeParameter "atomicParameter": D<string>
2 data type GenericSingleNodeParameter "genericParamater": P
3 data type SingleNodeParameterRef "typeRef": SingleNodeParameter
4 data type SingleNodeParameterAlias SingleNodeParameter
```

Listing 6: SingleParameterNode example

```
data type AtomicParameterList (
  "identifier": ID<int>,
  "listOfFloats": D<float>*
)
```

Listing 7: AtomicParameterList example

```
data type ParameterTree {
  "level1": {
    "level2": {
      "listOfFloats": D<float>+,
      "typeReference": SingleNodeParameter
    }
  }?
}
```

Listing 8: ParameterTree example

```
data type ParameterForest [
  {
    "prop1": D<string>,
    "list": (
      "ref1": AnotherDatatype, // reference existing data types
      "myNumber": D<int>
    ),
    "subTree": {
        "identifier": ID<int>
    }
  };

  "namedTree": {
    "treeStringAttribute": D<string>
  }
]
```

Listing 9: ParameterForest example

4.1.3 Endpoint skeleton

A MDSL's endpoint, a description of a system's interface, follows the following skeleton (from [59]):

```
endpoint type <name>
version x.y.z // semantic versioning information (optional)
serves as <role_enum> // MAP tag(s) (optional)
exposes
  operation <name>
  with responsibility <resp_enum> // MAP tag (optional)
  expecting
    headers [...] // optional
    payload [...] // mandatory
  delivering
    headers [...] // optional
    payload [...] // mandatory
  reporting
    [...] // error handling such as fault elements
          // or response codes
```

Listing 10: MDSL endpont skeleton

AsyncMDSL channels definitions follow a similar structure, trying to mimic the experience of using standard MDSL. Providing a consistent structure across the whole grammar is an important aspect to consider when designing an extension.

4.1.4 Provider skeleton

An API provider offers one or more endpoint contracts, specifying at which address location it will expose them and under which protocol. A basic provider follow the skeleton:

```
API provider <name>
offers <serviceSpecification> // reference
at endpoint location <endpointAddress>
via protocol <endpointProtocol>
under conditions [...] // optional
provider governance <evolutionPattern> // Evolution MAP, optional
```

4.1.5 Client skeleton

API Clients are endpoints' consumers, and they need only to specify which endpoints they interact with and under which protocol.

```
API client <name>
consumes <endpointRef> // reference
from <providerRef> // reference, optional
via protocol <endpointProtocol> // optional
```

4.2 AsyncMDSL Language

The AsyncMDSL document structure is the same as the original MDSL's one, excepts for the components' names. To align MDSL's components with the naming convention in [23], AsyncMDSL proposes a new terminology:

- endpoint type → message channel
- provider → message broker
- client → message endpoint

This naming guarantees a tight correspondence between EIPs names and language constructs, preventing an overlapping between unrelated concepts, such as Message Endpoints in AsyncMDSL and endpoints as operations container in standard MDSL.

Every AsyncMDSL document has thus a root object, *serviceSpecification*, which represents a description of a single message-based system. A *service-Specification* contains a list of Message Channels, of Message Brokers and finally a list of Message Endpoints. A simplified AsyncMDSL's Ecore class diagram is depicted in Figure 3.

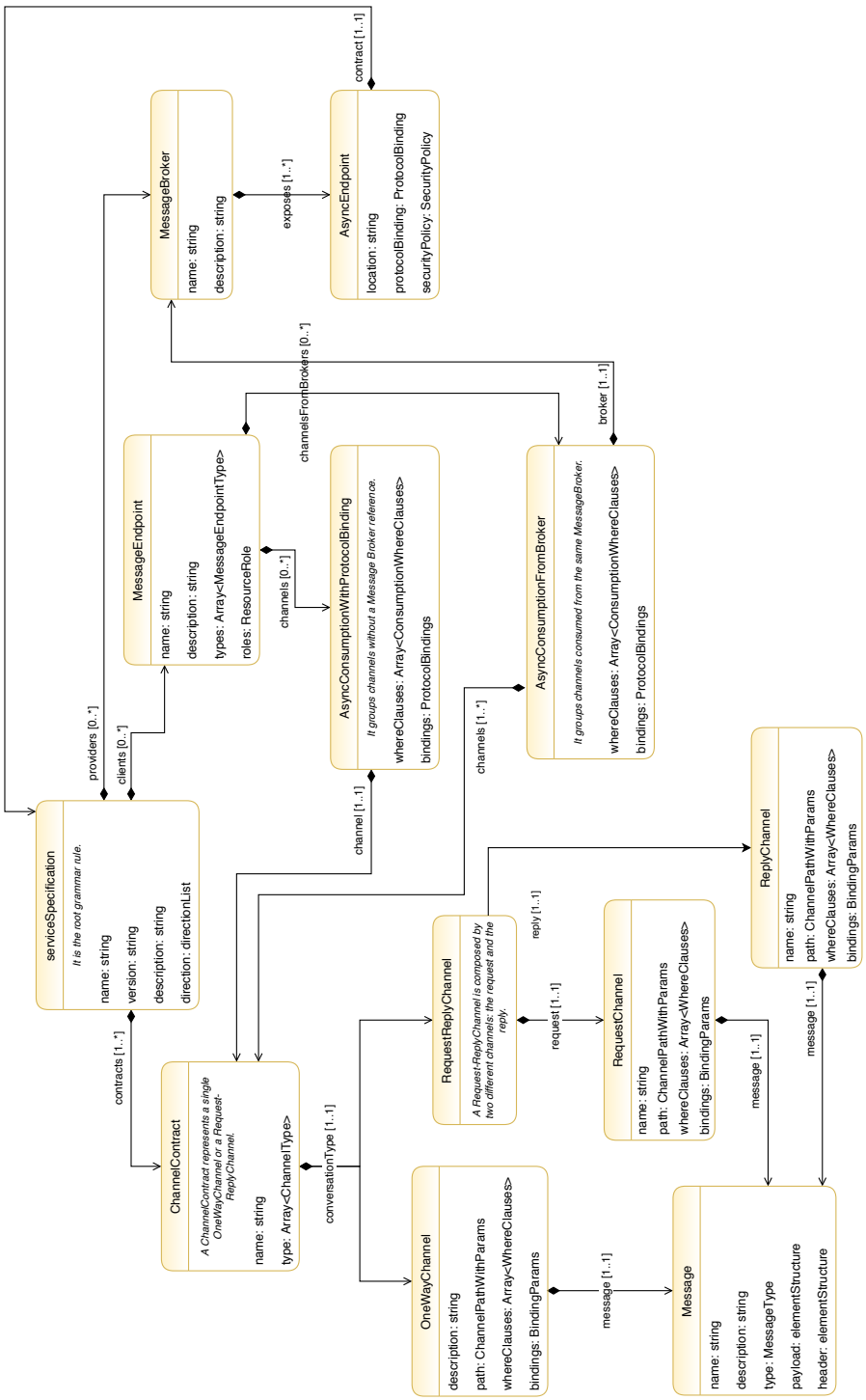

Figure 3: Simplified AsyncMDSL's Ecore class diagram

4.2.1 AsyncMDSL example

The Example 5 applied in a message-based context, and converted in AsyncMDSL, is reported in the next listing.

```
API description HelloWorldAsyncAPI

data type SampleDTO { "id": ID<int>, "message": D<string> }

channel SayHello
  of type PUBLISH_SUBSCRIBE
  on path "/public/sayHello"
  produces message HelloMessage
  delivering payload SampleDTO

message broker HelloWorldAmqpProvider
exposes HelloWorldAsyncAPI
at endpoint "amqp.example.com"
via protocol AMQP

message endpoint HelloWorldAmqpClient
uses from HelloWorldAmqpProvider:
      SayHello
```

Listing 11: AsyncMDSL example

In Listing 11 we define an API which has a single channel, *SayHello* to which many Message Endpoints can consume from. The channel is a Publish-Subscribe Channel and delivers a *SampleDTO* as output to subscribers. A Message Broker exposes the *SayHello* channel via AMQP protocol, and a Message Endpoint uses the channel from this message provider. A Message Endpoint using a channel means that if that channel is producing messages (as in this case), the Message Endpoint will act as a consumer of that channel, subscribing to it, and waiting for messages.

4.3 AsyncMDSL Language Features

4.3.1 Extending a grammar

To extend the base MDSL language we need to extend its grammar. The Xtext-based grammar is composed by several rules that can be modified. To support a new syntax, we can either directly extend existing rules to support

it, or we can add new alternative rules that can recognize the new syntax. For instance, suppose having a rule like the following:

```
provider:
  'API' 'provider' name=ID
  ('contained' 'in' parent=[provider])?
  // [...]
;
```

Listing 12: Base rule that we want to extend

If the new syntax requires

1. the introduction of new keywords for invoking the rule (eg. allowing to instantiate a provider with 'message broker' keywords instead of 'API provider')
2. the elimination of the parent provider
3. the backwards compatibility of the specification

we can adopt different approaches to support this new requirements.

Directly extending rules

The first approach could be to modify the existing rule to support both the 'API provider' and 'message broker' keywords. After that, to ensure that parent provider is used only if the instance is initialized with the 'API provider' keyword, we would need to create a verification rule that will be invoked at runtime by the framework. A runtime rule is a non-other that a linter rule that statically analyzes the source code.

```
provider:
  (apiProvider='API' 'provider'
    | messageBroker='message' 'broker') name=ID
  ('contained' 'in' parent=[provider])?
  // [...]
;
```

Listing 13: Directly extended rule

Even if this approach could actually work, it does not scale to complex rules and requires lots of validation checks at runtime, which is not something a grammar should go for. A grammar should indeed statically force to write meaningful language constructs. To statically enforce meaningful constructs we can adopt the extension proposal described next.

Adding new alternative rules

Instead of directly modifying rules to support the new syntax, we can add a new rule that represents new syntax's requirements,

```
MessageBroker:
  'message' 'broker'
  // note also the removal of the parent provider
  // [...]
;
```

Listing 14: New rule for new syntax

and apply a modification to the parent rule. The previous *provider* rule is indeed contained in another rule.

```
serviceSpecification:
  // [...]
  providers+=provider*
  // [...]
;
```

Listing 15: *provider* parent rule

Editing the parent *serviceSpecification* rule as the following

```
serviceSpecification:
  // [...]
  providers+=(provider | MessageBroker)*
  // [...]
;
```

Listing 16: New parent rule that supports both syntaxes

ensures that at the same time the backwards compatibility will be guaranteed, and that the new syntax will be recognized. This approach allows defining a scalable grammar that enforces statically meaningful constructs (meaning that the new syntax rules can not be mixed in unwanted ways).

4.3.2 ServiceSpecification

A *serviceSpecification* is the only root object of any AsyncMDSL document. It represent a whole message-based system or a logically connected portion of it. It should contain all the information that a service provider intend to offer to its clients. In the message-driven context, such information comprise

Message Channels and Message Brokers. AsyncMDSL has been developed taking into account the possibility to model entire systems, and entire systems are composed also by Message Endpoints. Hence, a *serviceSpecification* contains also the definition of Message Endpoints that interact with the exposed channels from Message Brokers.

```
serviceSpecification:
  'API' 'description' name=ID
  ('version' svi=semanticVersioningIdentifier)?
  ('description' description=STRING)?
  ('usage' 'context' reach=visibility
      'for' direction+=directionList)?
  types+=dataContract*
  contracts+=(endpointContract | ChannelContract)+
  providers+=(provider | MessageBroker)*
  clients+=(client | MessageEndpoint)*
;
```

Listing 17: Simplified AsyncMDSL root grammar rule

Listing 18 shows the *serviceSpecification* language concept in action, which is defined in the above Listing 17 and covers — once it has been populated with all the children components — the user story US-1: Model a message-based system. The *serviceSpecification* rule is provided by standard MDSL, and it has been extended to support our new requirements.

```
API description AsyncMDSLServiceSpecification
version "1.0.0"
description "
  This preamble represent a description of a system.
"
  // A preamble is followed by the definitions of:
  // - datatypes
  // - channels
  // - brokers
  // - endpoints
```

Listing 18: AsyncMDSL *serviceSpecification* example

4.3.3 ChannelContract

ChannelContract components contained in the *serviceSpecification* represent a Message Channel. Each *ChannelContract* can model either a single Message

Channel or a Request-Reply Message flow. Each *ChannelContract* has one or more types, that comprise:

- Point to Point: exactly one receiver will receive the message sent over this channel
- Publish-Subscribe: the message sent over this channel is broadcasted to all receivers subscribed to it
- Datatype: all messages sent over this channel have the same payload and header structures
- Invalid Message: messages sent over this channel are messages which receivers were not be able to handle
- Dead Letter: messages sent over this channel are messages that the messaging system was not able to deliver
- Guaranteed Delivery: messages sent over this channel are ensured to be persisted, preventing their loss if the messaging system fails.

Each of these types represents the corresponding Message Channel pattern described in section 3.1.3.

```
ChannelContract:
  'channel' name=ID
  ('of' 'type' types+=ChannelType (',' types+=ChannelType)*)?
  conversationType=(RequestReplyChannel | OneWayChannel)
;
```

Listing 19: AsyncMDSL *ChannelContract* grammar rule

The grammar rule allows the definition of multiple types for a single *ChannelContract*, without taking into account that not all values are a valid combination. For instance, if we declare a Message Channel as a Point-to-Point Channel, we could potentially mark it as also as a Publish-Subscribe Channel, creating a syntactically valid structure that is logically incorrect. This type of validation is thus required, but performed at runtime, exploiting Xtext semantic checks, described in section 4.4 Static Verification Rules (linter).

```
channel MyAwsomeChannel
of type POINT_TO_POINT, DATA_TYPE, GUARANTEED_DELIVERY
// OneWayChannel or RequestReplyChannel
```

Listing 20: AsyncMDSL *ChannelContract*

Listing 20 shows the common shared structure between Message Channels and Request-Reply Channels. The inner content of the *ChannelContract* rule

is delegated to *OneWayChannel* or *RequestReplyChannel* rules respectively defined in section 4.3.3 and section 4.3.3. This rule fulfills the US-3: Message Channels.

One-way channel

In Listing 21 we define the grammar rule for a single Message Channel. Each Message Channel has a path on which the broker expects the input or delivers the output. This path could also contain parameters that need to be determined at runtime (described in below section 4.3.3). Each Message Channel can produce messages, consume messages, or both, even if consuming from and producing to the same channel is not a best practice. Finally, a Message Channel can contain some other information, such as the Message Expiration policy of a message. This kind of information is represented inside the list of *WhereClauses*.

```
OneWayChannel:
  (('description' description=STRING)? &
  'on' path=ChannelPathWithParams)
  (subscribe?='accepts' | publish?='produces') message=Message
  ('where' whereClauses+=WhereClauses
      (',' whereClauses+=WhereClauses)*)?
  ('bindings' 'for' protocol=TransportProtocol
      bindings=BindingParams)?
;
```

Listing 21: AsyncMDSL *OneWayChannel* grammar rule

```
channel MyAwsomeChannel
of type POINT_TO_POINT, DATA_TYPE, GUARANTEED_DELIVERY
on path "channel-logical-path"
produces message MyAwsomeMessage
  delivering payload {
    "myAwsomeProperty": D<string>
  }
where
  MESSAGE_EXPIRES in 60s
bindings for AMQP {
  "queue": {
    "name": "my-queue-name"
  }
}
```

Listing 22: AsyncMDSL *OneWayChannel*

Channel path

The logical channel path, a parameterizable string, is used to identify a channel inside a Message Broker. It is possible to define a channel path inside the channel definition or the Message Broker. If the path were defined in the Message Broker, it would have been specified as a property of the relation between a broker and a channel, meaning that different brokers expose the same channel under different logical paths. If brokers need to expose the same channel under the same path, they would need to specify the channel path multiple times. Given that a logical path can be considered equivalent to a RESTful API's URI, we decided that it had to be a property of the channel rather than a property of a broker's relation with it. An application that uses a channel, indeed, references it by its logical path. Even if two channels, with the same input/output schemas, can be considered equal from the application's perspective, if their logical path is different, they would no longer be interchangeable.

```
ChannelPathWithParams:
'path' path=STRING
('with'
params+=BasicParameterWithDescription
(',' params+=BasicParameterWithDescription)?
)?
;
```

Listing 23: AsyncMDSL channel path grammar rule

```
channel BanksLoansInsightsChannel
of type PUBLISH_SUBSCRIBE
on path "banks/${bankId}/loans" // channel path with parameter
  with bankId: int, "The bank from which the loan
                     has been requested."
description "Subscribe to be notified when
             a new loan request happens."
produces message NewLoanRequested
  delivering payload LoanNotificationDto
```

Listing 24: AsyncMDSL channel path with parameters example

Parameters are embedded in the path using the syntax `${parameterName}`, and a semantic verification rule will notify the lack of parameter definitions in the path, if needed.

Message's payload and header

Each channel either expects (consumes) or delivers (produces) a message. A message is a representation of the data that will be carried through a channel, and it is composed of payload, header, and type. A message type represents one of the following user stories:

- US-4.1: Command Message
- US-4.2: Document Message
- US-4.3: Event Message

while the possibility to model message payload and header covers US-3.3: Datatype Channel.

```
Message:
  'message' name=ID
  ('description' description=STRING)?
        (deliveringPayload?='delivering' |
  expectingPayload?='expecting')
  payload=Payload
;

Payload:
  schema=dataTransferRepresentation
  ('as' messageType=MessageType)?
;

dataTransferRepresentation:
  ('headers' headers=elementStructure)?
  'payload' payload=elementStructure
;

enum MessageType:
  COMMAND_MESSAGE | EVENT_MESSAGE | DOCUMENT_MESSAGE
;
```

Listing 25: AsyncMDSL message grammar rules

```
channel MyChannel
of type DATA_TYPE
on path "channel-logical-path"
accepts message MyIncomingMessage
  expecting
    headers CommonHeaders
    payload MyDocumentDto as DOCUMENT_MESSAGE
```

Listing 26: AsyncMDSL Message example

Request-Reply channel

Request-Reply messages require distinct logical channels to communicate: one channel for the request and one channel for the reply. A *RequestReplyChannel* allows the definition of both logical channels, where each of them contains the payload they expect/deliver. Also, in this type of communication, it might be useful to specify further information, such as the Correlation Identifier of a message. As per the *OneWayChannel*, this information is represented into the list of *WhereClauses*.

```
RequestReplyChannel:
  request=RequestChannel
  reply=ReplyChannel
;

RequestChannel:
  'request' 'message' name=ID
  (('description' description=STRING)? &
  'on' path=ChannelPathWithParams)
  'expecting' payload=Payload
  ('where' whereClauses+=WhereClauses
      (',' whereClauses+=WhereClauses)*)?
  ('bindings' 'for' protocol=TransportProtocol
      bindings=BindingParams)?
;

  ReplyChannel:
  'reply' 'message' name=ID
  (('description' description=STRING)? &
  'on' path=ChannelPathWithParams)
  'delivering' payload=Payload
  ('where' whereClauses+=WhereClauses
      (',' whereClauses+=WhereClauses)*)?
  ('bindings' 'for' protocol=TransportProtocol
      bindings=BindingParams)?
;
```

Listing 27: AsyncMDSL *Request-Reply Messages* grammar rule

RequestReplyChannel covers US-4.4: Request-Reply Message.

```
data type WakeUpDto {
  "deviceId": ID<int>,
  "wakeUp": D<bool>
}

channel RequestReplyMessageChannel
of type POINT_TO_POINT
  request message WakeUpChannelRequest
    on path "devices/wakeup"
    expecting payload WakeUpDto as COMMAND_MESSAGE
  reply message WakeUpChannelReply
    on path "devices/${deviceId}/wakeup/reply"
      with deviceId: int, "The id of the device
                           that received the command"
    delivering payload { "success": D<bool> }
```

Listing 28: AsyncMDSL Request-Reply Message flow

Where clauses

The *WhereClauses* construct allows to specify further conditions on Message Channels. It permits to represent Message Expiration, Message Sequence and Correlation Identifier EIPs, thus covering user stories defined in sections 3.1.6, 3.1.7 and 3.1.8.

```
WhereClauses:
  MessageExpireWhereClause
    | SequenceIdWhereClause
    | CorrelationIdWhereClause
;
MessageExpireWhereClause:
  'MESSAGE_EXPIRES' 'in' messageExpire=INT
          messageExpireUnit=MessageExpireUnit
;
SequenceIdWhereClause:
  'SEQUENCE_ID' 'is' source=STRING
;
CorrelationIdWhereClause:
  'CORRELATION_ID' 'is' source=STRING
;
```

Listing 29: AsyncMDSL *WhereClauses* grammar rules

Correlation Identifier and Message Sequence indicate a property of a message that can be respectively used to trace the chain of messages or understand messages' order. The property is specified following the JSON Pointer [17] specification. It expects a root value, in our case the message payload or header, and then it specifies a list of reference tokens used to reconstruct the path of the property inside the root value. In our DSL it follows the template:

```
$message.(payload | header)#/path/to/the/target/property
```

WhereClauses can be seen in action in the following listing.

```
channel LoanBrokerChannel
of type GUARANTEED_DELIVERY, POINT_TO_POINT
request message LoanRequest
  on path "loan-broker/request"
  expecting
    payload { requestId: ID<int>, content: P }
reply message LoanReply

  on path "loan-broker/reply"
  delivering
    headers CommonHeaders
    payload LoanReplyDto
  where
    CORRELATION_ID is "$message.payload#/requestId",
    MESSAGE_EXPIRES in 60m
```

Listing 30: AsyncMDSL *WhereClauses* in action

4.3.4 Message Brokers

A Message Broker, that can be seen as a service provider, is in charge of defining which channels it offers and under which conditions. A Message broker can offer different *serviceSpecification*s at different locations, under different protocols and with different Service Level Agreement (SLA). For each *serviceSpecification* component offered by a Message Broker, protocol-specific information — bindings — can be also specified, as well as some basic information about the security policy.

```
MessageBroker:
  'message' 'broker' name=ID
  ('description' description=STRING)?
  'exposes' epl+=AsyncEndpoint+ (',' epl+=AsyncEndpoint)?
;

AsyncEndpoint:
  contract=[serviceSpecification]
  'at' 'location' location=STRING
  pb=ProtocolBinding
  ('bindings' bindings=BindingParams)?
  ('policy' name=ID 'realized' 'using'
     (securityPolicy=OASSecurity | other=STRING)
     ('in' securityPolicyExpression=STRING)?)?
;
```

Listing 31: AsyncMDSL Message Broker grammar rules

In the following example we are defining a Message Broker which exposes the same *serviceSpecification* under different locations and protocols. Furthermore, the service exposed under MQTT contains some protocol-specific information and a security policy.

```
message broker MyMessageBroker
exposes
  MyServicesSpecification
    at location "tcp://mqtt.myapp.com:1883"
    via protocol MQTT
    bindings {
      "lastWill": {
        "qos": 2,
          "retain": false
        }
    }
    policy AuthenticatedUsersOnly
      realized using API_KEY in "$message.header#/apiKey",

  MyServicesSpecification
    at location "amqp://my.app.com"
    via protocol AMQP
```

Listing 32: AsyncMDSL Request-Reply Message flows

The Message Broker grammar rule defined above allows to cover US-15: Message Brokers and US-16: Specify protocol-specific information.

4.3.5 Message Endpoints

Message Endpoints are clients that connect to Message Brokers. Message Endpoints can use one or more Message Brokers, and for each one of them, they can specify individual Message Channel configurations. Such channel configurations can consist of specifying under which condition a subscription is invoked (e.g., a Selective Consumer) or if a subscription is durable or not (e.g., Durable Subscriber). A Message Endpoint can also be abstractly defined by specifying which channel it will use, without any needs to specify also the broker from which it will use those channels.

```
MessageEndpoint:
  'message' 'endpoint' name=ID
  ('of' 'type' types+=MessageEndpointType
      (',' types+=MessageEndpointType)*)?
  ('serves' 'as' primaryRole=ResourceRole
      ('and' otherRoles+=ResourceRole)* 'role'?)?
  ('description' description=STRING)?
  'uses'
  ('channels' ':'
    channelsNoBroker+=AsyncConsumptionWithProtocolBinding (','
        channelsNoBroker+=AsyncConsumptionWithProtocolBinding)*
  )?
  (channels+=AsyncConsumptionFromBroker
      (',' channels+=AsyncConsumptionFromBroker)*)?
;
```

Listing 33: AsyncMDSL Message Endpoints grammar rules

```
message endpoint MyDeviceEndpoint
of type EVENT_DRIVEN_CONSUMER, DURABLE_SUBSCRIBER
serves as PROCESSING_RESOURCE
uses
  channels:
    MyChannelWithoutBroker1,
    MyChannelWithoutBroker2

  from MessageBroker:
    WakeUpChannelRequest
      where consumed if "$message.payload#/deviceId" == 42,

  from OtherMessageBroker:
    ChannelExposedByOtherMessageBroker
```

Listing 34: AsyncMDSL Message Endpoint example

Indicating the type of a Message Endpoint allows us to cover:

- US-10: Competing Consumers
- US-11: Polling Consumer
- US-12: Event-Driven Consumer
- US-14: Durable Subscriber

while the *ConsumptionWhereClauses* (Listing 34, line 11) to cover US-13: Selective Consumer. The *ConsumptionWhereClauses*' left expression follows the same JSON Pointer format described in section 4.3.3.

4.4 Static Verification Rules (linter)

A linter [28] is a software that examines source code to detect programming errors, bugs, stylistic inconsistencies, and suspicious constructs. Xtext provides a way to define custom verification rules that will statically validate the source code of the DSL, in this case of AsyncMDSL. This rules play a fundamental role in enforcing meaningful constructs since grammar rules alone can not represent all the semantic we need. The currently implemented semantic checks are listed in table 2.

Rule	Type	Motivation
Message Channel names must be unique	error	Message Channel are identified and referenced by their name, that thus requires to be unique in each AsyncMDSL document.
Message Channel paths must be unique	error	Message Channel using the same path might cause unwanted behavior in the message-based system. Message Channels with the same path are likely to be a design error.
Request-Reply Channel types can not be Publish-Subscribe, Invalid Message or Dead Letter	error	A Request-Reply Channel is a Point-to-Point channel by design, and thus it can not be of type Publish-Subscribe, Invalid Message Channel or Dead Letter Channel.
Message Channel types must be compatible each other	error	A Message Channel defined with a type can not contain types that conflict with the previously defined one. For example if a Message Channel is declared as Dead Letter Channel, it can not be also of type Invalid Message Channel.
Message Channel types can not contain duplicates	warning	It is useless to define a duplicate type for a Message Channel.

Message Endpoint should contain a Message Channel reference	warning	A Message Endpoint not containing Message Channel references does not represent any information.
Message Channel direction must match payload direction	warning	If a Message Channel produces a message it will deliver a payload, and it will never expect a payload. The same applies reversed.
Message Channel direction must match header direction	warning	If a Message Channel produces a message it will deliver headers, and it will never expect headers. The same applies reversed.
Runtime expression must be compliant to the JSON Pointer format [17]	error	The expression must reference a valid field of the message or it will not parsed properly.
Parameters in the path of a channel must be described	error	Each parameter that appears in the path of a channel must have a type.
Broker security policy must specify API_KEY path	warning	If a Message Broker is using API_KEY as security policy, it must specify where to find the API_KEY in the message.
Broker security policy must specify API_KEY path in correct format	error	The expression must reference a valid field of the message or it will not parsed properly.

Table 2: Implemented semantic checks

4.5 Generating AsyncAPI

One of the project requirements is the possibility of generating AsyncAPI from an AsyncMDSL document to exploit the existing tooling that AsyncAPI's team has developed. One of those existing tools is the *async-api-generator*[4],

[4]https://github.com/asyncapi/generator/releases/tag/v1.0.0-rc.4

a Node.js [35] package that takes as input an AsyncAPI document and produces as output a project skeleton based on the input specification and the selected target template. The generator supports different templates, allowing to reduce the initial configuration of a new project by providing boilerplate code. The current version (*v1.0.0-rc.4*) of the generator supports the following official templates:

- asyncapi/nodejs-template: a Node.js template with support for AMQP, MQTT, Kafka and WebSockets
- asyncapi/nodejs-ws-template: another Node.js template with support only for WebSockets
- asyncapi/java-spring-template: a Java Spring Boot template with support only for Kafka
- java-spring-cloud-stream-template: a Java Spring Cloud Stream template with support for Kafka and RabbitMQ
- asyncapi/python-paho-template: a Python template with support for MQTT
- asyncapi/html-template: a browsable static website that can be used for documentation purposes
- asyncapi/markdown-template: an alternative representation of the HTML version that instead uses markdown[5]

Our language extension must be converted into AsyncAPI to exploit this generator. The conversion can be accomplished either by:

- mapping our Abstract Syntax Tree (AST) to the AsyncAPI's AST
- exploiting a template-based approach.

Eclipse Xtext provides the AST of AsyncMDSL, but at the time of writing, there is no object-oriented representation of AsyncAPI's AST. So we opted for a template-based approach, where we define a template that matches the AsyncAPI specification structure, and we populate it with a model. This approach is often referred to as the MVC (Model View Controller) pattern [11].
Different template engine exists, such as [6], [16] or [4]. The initial choice was Apache Freemarker, but it turned out that it was not the best approach to accomplish our goal. Three major feature were missing:

[5]https://commonmark.org/

- it does not provide code completion for anything but template language[6]
- a template can not contain Java expression
- every method that does not belong to the model object must be individually injected.

We then decided to use a tool suggested by the Xtext framework: Xtend [14]. Xtend is a dialect of Java that compiles into readable Java 8 compatible source code. As soon as we generate Xtext artifacts for the grammar, a code generator stub is put into the DSL's runtime project. Writing a generator for AsyncAPI is a matter of implementing that abstract class to produce valuable output. One of the main advantages of using Xtend to implement the generator is that it supports feature like multiline strings, native Java expressions invocations and smart handling of white space in the template output, resulting in readable templates as well as nicely formatted output.

```
private def compile(serviceSpecification serviceSpecificationInstance) '''
    asyncapi: '2.0.0'
    info:
        title: «serviceSpecificationInstance.fullyQualifiedName»
        version: «getValueOrDefault(serviceSpecificationInstance.svi, "Not defined")»
        description: |
            «getValueOrDefault(serviceSpecificationInstance.description, "No description specified")»
    «IF(serviceSpecificationInstance.providers?.filter(MessageBroker).length > 0)»
        servers:
            «FOR broker : AsyncApiGeneratorHelper.getBrokers(serviceSpecificationInstance)»
                «broker.name»:
                    url: «broker.url»
                    protocol: «broker.protocol»
                    description: «broker.description»
                    «insertBinding(broker.bindings, broker.protocol)»
            «ENDFOR»
    «ENDIF»
    channels:
        «FOR contract : serviceSpecificationInstance.contracts.filter(ChannelContract)»
            «contract.compile»
        «ENDFOR»
```

Figure 4: Xtend smart white space support in Eclipse

The code snippet shown in Figure 4 represent an example of Xtend's smart white space handling. All blue characters highlighted in gray are the one that will be directed to output, while other characters, such the ones that indent the inner for loop, are ignored.

Using Xtend to create the generator allows to employ Java features, carefully made available though a syntax designed to define templates (e.g., by

[6] An Eclipse Freemarker editor has been deprecated by Red Hat, but it is still available at https://github.com/jbosstools/jbosstools-freemarker.

offering a safe navigator operator, Figure 5). On the other side, one huge drawback of using a template approach rather that mapping Abstract Syntax Trees is that any error in the template result in an invalid AsyncAPI document.

```
def compile(elementStructure dto) '''
    «««« Single Parameter Node
    «dto.np?.compile»
    «««« Parameter Tree
    «dto.pt?.compile»
    «««« Parameter Forest
    «dto.pf?.compile»
    «««« Atomic Parameter List
    «dto.apl?.compile»
'''
```

Figure 5: Xtend example: arguments that would be passed to the method *compile* are only evaluated if the method will be invoked.

4.5.1 MDSL data types to JSON Schema specification

All features defined into an AsyncMDSL document must be converted into an AsyncAPI document. Documents describing an event-driven system in accordance with AsyncAPI specification are represented as JSON objects and conform to the JSON standards. YAML [57], being a superset of JSON, can be used as well to represent any AsyncAPI specification file and will target this format. MDSL datatypes must be converted as well to YAML format, and Table 3 illustrates the mapping between Structure Patterns available in MDSL and the corresponding YAML structures.

Structure Pattern [61]	**YAML**
Atomic Parameter	Scalar Node
Atomic Parameter List	Sequence Node of Scalar Nodes
Parameter Tree	Mapping Node
Parameter Forest	Sequence Node of Mapping Nodes

Table 3: AsyncMDSL data types to YAML mapping

Even if a mapping between MDSL data types and YAML format exists, we actually need to convert data types in what AsyncAPI's teams call Schema

Object[7]. A Schema Object allows the definition of input and output data types for the exposed channels, and it is a superset of JSON Schema [46] specification. [46] describes data formats providing human readable documentation and can be used to automatically validate data. An example of an AsyncAPI's Schema Object can be found in Listing 35. The same information represented in MDSL is in Listing 36.

```
TypeReferenceDemo:
  type: object
  required:
    -  referenceId
  properties:
    referenceId:
      type: number
ParameterTreeDemo:
  type: object
  required:
    -  listOfKeyValues
  properties:
    listOfKeyValues:
      type: array
        items:
        type: object
        required:
          -  key
        properties:
          key:
            type: number
          value:
            type: object
            properties:
            additionalProperties:
              type: null
    reference:
      $ref: '#/components/schemas/TypeReferenceDemo'
```

Listing 35: AsyncAPI Schema Object example

```
data type TypeReferenceDemo "referenceId": ID<int>

data type ParameterTreeDemo {
  "listOfKeyValues": {
    "key": D<int>,
```

[7]https://www.asyncapi.com/docs/specifications/2.0.0/#schemaObject

```
    "value": { P }? // optional value of unknown shape
  }+, // note the + to define a non-empty list
  "reference": TypeReferenceDemo?
}
```

Listing 36: MDSL equivalent of the above Schema Object

In the conversion to the JSON Schema representation Parameter Forest — which are list of items of any shape — are converted into tuples, sequences of fixed length where each item may have a different schema. Labels of Parameter Forest are ignored during the conversion.

```
data type ParameterForestDemo [
  "x": { // "x" is the label of the first tuple item
    "tupleFirstItem": D<string>
  };

  "y": {
    "tupleSecondItem": TypeReferenceDemo
  };

  "z": {
    "tupleThirdItem": P
  }
]
```

Listing 37: Parameter Forests are seen as tuples in AsyncMDSL

```yaml
ParameterForestDemo:
  type: array
  items:
    -
      type: object
      required:
        - tupleFirstItem
      properties:
        tupleFirstItem:
          type: string
    -
      type: object
      required:
        - tupleSecondItem
      properties:
        tupleSecondItem:
          $ref: '#/components/schemas/TypeReferenceDemo'
    -
      type: object
      required:
        - tupleThirdItem
      properties:
        tupleThirdItem:
          type: object
          additionalProperties:
            type: null
```

Listing 38: ParameterForest are mapped to tuples in JSON Schema specification

4.5.2 AsyncMDSL to AsyncAPI mapping

Each AsyncMDSL concept is mapped to one, or a combination of more AsyncAPI's specification objects. Table 4 provides a mapping to such objects.

AsyncMDSL	AsyncAPI	Description
Service Specification	AsyncAPI Object, AsyncAPI Version String, AsyncAPI Info Object and Components Object	Each serviceSpecification represent a group of channels, and contains global available information such as the version and a description, and furthermore it is the container of all the other components.
OneWay Channel	Operation Object	An operation represent a channel, with path, parameters and message schema. In AsyncAPI channels' names are their paths.
Request-Reply Channel	A pair of Channel Item Object	A Request-Reply Channel is mapped into a pair of separate channels, each with its own path.
Channel parameter	Parameter Object	Each object describes a single parameter included in a channel path.
Message	Message Object	Represent a message that flows through a channel. It provides a place to document how and why messages are produces and consumed.
Channel Binding	Operation Binding Object	Allows the definition of protocol-specific parameters for a Message Channel.
Broker Bindings	Server Bindings Object	Allows the definition of protocol-specific parameters for a Message Broker.
Datatypes	Schema Object	Allows the definition of input and output data types.

Correlation Id Where Clause	Correlation ID Object	Specifies an identifier at design time that can used for message tracing and correlation.
Datatype Type Reference	Reference Object	Allows referencing other data types in the specification.
Message Broker	Server Object	Represents a Message Broker using a single protocol. If a Message Broker supports multiple protocols, a copy of it will be created for each distinct protocol.
Endpoints		AsyncAPI does not support Message Endpoints.

Table 4: AsyncMDSL concepts mapped to AsyncAPI's specification objects

Chapter 5

Loan Broker Example

5.1 Modeling a scenario

The Loan Broker is a well-known example described in [23] to discuss how EIPs can be applied. Even if it does not cover all of our user stories, it is a good example to see AsyncMDSL in action. The language easily models this scenario, and even if not shown in this example, it can cover more use cases derived from our requirements.

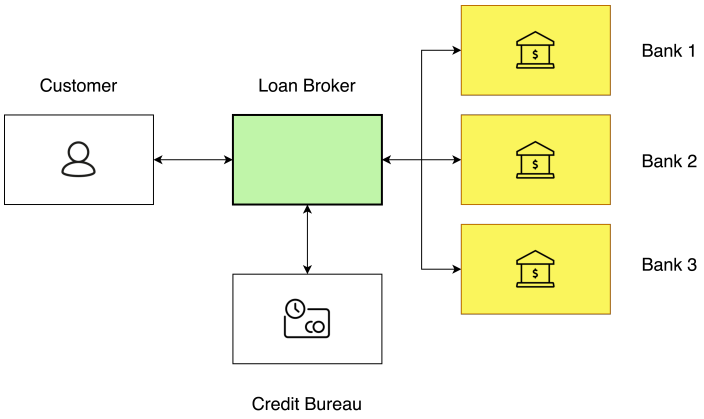

Figure 6: Overview of the example scenario

A customer need to obtain a loan, and provides some personal information to a loan broker. The loan broker needs this data to determine the best interest rate it could obtain from the banks. Before contacting the banks, the loan broker interacts with a credit bureau to get the credit worthiness of the customer. Once obtained this value the loan broker proceeds contacting

banks and determining the best quote to offer to its customer. The loan broker requires the following inputs from the customer:

- social security number, to uniquely identity the requester
- loan amount
- loan term,

while the customer will receive in response from the broker:

- the interest rate
- a quote identifier for future references.

For the sake of simplicity, we assume that all entities already employ a message-based architecture or some Channel Adapters [23]. Every AsyncMDSL document start with a preamble

```
API description LoanBrokerExample
version "1.0.0"
```

Listing 39: AsyncMDSL preamble

where we define name, version and an optional description of the overall system design. We then proceed to declare all the Data Transfer Objects (DTOs) that our system will use. Defining all DTOs is not required, as they can also appear as anonymous objects where needed, but it explicitly separates data representation and system's architecture, furthermore they can be reused multiple times.

```
data type LoanRequestDto {
  "socialSecurityNumber": ID<int>,
  "amount": D<double>,
  "termInMonths": D<int>,
  "requestId": ID<int>
}

data type LoanReplyDto {
  "quoteId": ID<int>,
  "interestRate": D<double>,
  "requestId": ID<int>
}
```

Listing 40: Loan broker input/ouput parameters

```
data type CreditBureauRequestDto {
  "socialSecurityNumber": ID<int> ,
  "requestId": ID<int>
}

data type CreditBureauReplyDto {
  "socialSecurityNumber": ID<int>,
  "creditScore": D<double>,
  "creditHistory": { P }*,
  "requestId": ID<int>
}

data type BankLoanRequest {
  "creditScore": D<int>,
  "creditHistoryLength": D<int>,
  "requestId": ID<int>
}

data type BankLoanReply {
  "quoteId": D<int>,
  "interestRate": D<double>,
  "requestId": ID<int>
}

data type CommonHeaders {
  "brokerId": ID<int>
}
```

Listing 41: Data Transfer Objects (DTOs) emploied in channels

Once data types are defined, they can be referenced inside channels. The first channel we model is the one that will be used by the customer to make a loan request and receive a response (*LoanBrokerChannel*). All types of channels, in this example, expect a request-reply message flow. Thus they are represented as Request-Reply Channels. Furthermore, they are also Guaranteed Delivery Channels, meaning that no message will be lost due to network or message broker failures.

```
channel LoanBrokerChannel
  of type GUARANTEED_DELIVERY, POINT_TO_POINT
  request message LoanRequest
    description "This channel is used by a
                 customer to make a request."
    on path "loan-broker/request"
```

```
7       expecting
8         payload LoanRequestDto as DOCUMENT_MESSAGE
9     reply message LoanReply
10      description "The loan broker will reply to the customer
11                   after having contacted all the banks and
12                   found the best quote."
13      on path "loan-broker/reply"
14      delivering
15        headers CommonHeaders
16        payload LoanReplyDto
17      where
18        CORRELATION_ID is "$message.payload#/requestId",
19        MESSAGE_EXPIRES in 60m
```

Listing 42: Loan broker channels

Every Request-Reply Channel is composed of two different channels on which the request and the reply will flow. Request and reply channels have a different logical path, and different input/output data models. The reply message must somehow be bound to the request message to understand which reply is for which request. We can express this relation by exploiting the Correlation Identifier pattern, as shown in line 15 of Listing 42, where we declare that the field that has to be used to link the reply is its payload's property `requestId`.

```
1  channel CreditBureauChannel
2    of type GUARANTEED_DELIVERY, POINT_TO_POINT
3    request message CreditScoreRequest
4      description "Request the credit score and customer history."
5      on path "credit-bureau/request"
6      expecting
7        headers CommonHeaders
8        payload CreditBureauRequestDto
9    reply message CreditScoreReply
10     description "Return the credit score and customer history"
11     on path "credit-bureau/reply"
12     delivering payload CreditBureauReplyDto
13     where
14       CORRELATION_ID is "$message.payload#/requestId"
```

Listing 43: Credit bureau channels

Channels can also optionally expect/deliver protocol-specific headers (e.g., AMQP, JMS, or Kafka message headers), and their path can contain parameters, as in the *BanksChannel*, where, based on the `bankId`, a different bank

will take care of satisfying the request.

```
channel BanksChannel
  of type GUARANTEED_DELIVERY, POINT_TO_POINT
  request message LoanProposalRequest
    description "Request a loan proposal"
    on path "banks/${bankId}/loans/request"
      with bankId: int, "The identifier of the bank to contact"
    expecting payload BankLoanRequest
  reply message LoanProposalReply
    description "The loan proposal for the given customer"
    on path "banks/${bankId}/loans/reply"
      with bankId: int, "The identifier of the bank that replied"
    delivering payload BankLoanReply
```

Listing 44: Banks channels

Once all channels have been defined, we can list Message Brokers, the providers that will expose the channels under a concrete protocol.

```
message broker LoanBrokerAmqpProvider
  exposes LoanBrokerExample
  at location "amqp.loanbroker.com"
  via protocol AMQP
```

Listing 45: Loan broker definition

Finally, we define all the Message Endpoints that will use channels defined above to consume or to produce messages. Message Endpoints can use channels from a Message Broker or can be abstractly defined by specifying the list of channels they will use without mentioning the actual Message Broker.

```
message endpoint Customer
  uses channels:
    LoanBrokerChannel

message endpoint LoanBroker
  uses from LoanBrokerAmqpProvider:
    LoanBrokerChannel,
    LoanRequest,
    LoanReply

message endpoint Bank1
  uses channels:
    LoanProposalRequest
      where
        consumed if "$message.payload#/creditScore" > 80,
```

```
16      LoanProposalReply
17
18  message endpoint Bank2
19    of type DURABLE_SUBSCRIBER
20    uses channels:
21      BanksChannel
22
23  message endpoint Bank3
24    uses channels:
25      BanksChannel
```

Listing 46: Message endpoints

Message Endpoints can have their type as well (e.g., they can be Durable Subscribers, as in line 19 of Listing 46), and specify under which conditions they will consume a message from a channel (and act as a Selective Consumer, as in line 15 of Listing 46).

The generated AsyncAPI specification file from this example can be found in appendix B.

Chapter 6

Discussion

6.1 AsyncMDSL and AsyncAPI

The AsyncMDSL language presented in this work can be used to model message-based system, providing an alternative specification to AsyncAPI. Even if both specifications have the same final scope, they use different design goals. AsyncAPI started as an adaptation of the OpenAPI specification [25], and one of the design principles was to have as much compatibility as possible with it. In this way some components of OpenAPI can be directly imported into an AsyncAPI document, and vice-versa. This approach, on one side, is convenient for messaging integration architects because it is similar to existing products and code reuse can be exploited. While on the other side this very same approach prevented some specification design decisions from being radically changed, maybe to best fit some new concepts. AsyncMDSL, on the other hand, has been developed entirely focusing on expressiveness in modeling messaging-based systems. As per this decision, it features lots of components directly inspired from state of the art solutions in [23].

6.1.1 Missing features

AsyncMDSL does not yet offer all features AsyncAPI offers, as the AsyncAPI specification is quite complex and AsyncMDSL is a novelty.

Security mechanisms

AsyncAPI allows the definition of a variety of security mechanisms to describe how message brokers deal with authentication. AsyncAPI currently supports:

- User/Password
- API key
- X509 certificate
- End-to-end encryption (either symmetric or asymmetric)
- HTTP authentication
- Some OAuth2's common flows
- OpenID Connect Discovery

AsyncMDSL does not yet provide a complete representation of the security mechanisms that a message broker would need.

Component traits

AsyncAPI allows the definition of a base component, a trait, that can be used to extend other components reducing code duplication. In Listing 47 we are defining a message channel where some information are inherited by the `BaseMessageHeaders`. Supposing `BaseMessageHeaders` defines payload's headers, also `users/register` would deliver the same headers.

```
users/register:
  publish:
    operationId: registerUser
    summary: Action to sign a user up.
    message:
      payload:
        $ref: "#/components/schemas/RegisterDto
      traits:
        - $ref: "#/components/operationTraits/BaseMessageHeaders"
```

Listing 47: AsyncAPI message trit example

To achieve a similar result with AsyncMDSL one would need to define a `data type CommonHeaders` and for each channel specify the headers referencing the shared base model. An AsyncAPI's `trait` indeed is more useful when used to specify multiple common properties of a channel, for example headers and bindings for a specific protocol.

Bindings

> The AsyncAPI specification does not assume any kind of software topology, architecture or pattern. Therefore, a server may be a message broker, a web server or any other kind of computer program capable of sending and/or receiving data. However, AsyncAPI offers a mechanism called "bindings" that aims to help with more specific information about the protocol and/or the topology. [24]

AsyncAPI's documentation describes "bindings" as a way to enrich the definition of a component, such as a channel or a broker. Bindings allow to define protocol-specific properties, for example the exchange name in AMQP or the group identifier in Kafka. AsyncMDSL allows the definition of bindings both at broker and at channel level, but the values are not validated, but rather just transcribed in the generation phase.

```
channel MyAMQPBroker
  on path "messages/new"
  produces message IncomingMessage
  delivering payload MessageDto
  bindings for AMQP {
    "is": "routingKey",
    "queue": {
      "name": "my-messages-queue-name",
      "durable": true
  }
}
```

Listing 48: AsyncMDSL channel binding example

Bindings contained in Listing 48 are just reported in the generated YAML file. This means that the AsyncAPI document will result valid if and only if binding properties follow the format that AsyncAPI expects. In a future version of the grammar, bindings will be directly supported, thus providing structure validation.

6.1.2 Specifications comparison

The following table summarizes some quick facts that derive from the comparison between our DSL and AsyncAPI. It features different aspects of the languages, such as their syntax, the size of their specifications, or their maturity.

Criterion	AsyncMDSL	AsyncAPI
Concrete syntax	DSL (made with Xtext)	YAML, JSON
Abstract syntax	EIPs	event-driven systems
Main use cases	agile modeling, contract first	code first, testing and scaffolding generation
Bindings	same as AsyncAPI[1]	AMQP, Kafka, WebSocket and many more[2]
Size of specification	18 pages (less complete)	52 pages
Size of *Loan Broker* example	347 words 3,211 characters	610 words 7,646 characters
Tools	few (editor, AsyncAPI generator)	many [3]
Maturity	will soon be open sourced	since 2017
Licence	Apache License 2.0	Apache License 2.0

Table 5: Comparison between AsyncMDSL and AsyncAPI

One of the facts that stands out is the considerable difference in the number of words and characters needed to model the Loan Broker Example. Our DSL indeed exploits the captured patterns knowledge to create comments in the generated AsyncAPI document. Those comments enrich the specification file, making it possible for clients to derive similar information as they would have done directly using our AsyncMDSL model. Another fact to consider when comparing the languages is that AsyncMDSL is less comprehensive than the corresponding counterpart: it does not allow, for example, the proper

modeling of the message broker's security policies, as previously pointed out.

Expressiveness is another important feature where specifications diverge. AsyncMDSL, being a domain-specific language for message-based systems, is designed to cover scenarios that arise in this domain. This means that it supports domain components as first class citizens. In a specification based on YAML/JSON, as AsyncAPI, domain components are instead non-native concepts that need to be represented exploiting the chosen format. Moreover, AsyncAPI highly exploits JSON Pointer to link components together, resulting in a less understandable structure.

```
devices/wakeUp:
  subscribe:
    operationId: wakeUpCommand
    message:
      $ref: '#/components/messages/WakeUp'
```

Listing 49: AsyncAPI channel definition

Comparing listing 49 and 50, we can notice how the AsyncAPI channel definition is not self-explanatory, and the reference chain needs to be manually followed in order to infer even some basic information that in the AsyncMDSL counterpart are straight available. The AsyncAPI channel in the first listing does not include information like the type of the channel (are consumers of the `WakeUp` message one or more than one?), neither information on the semantic of the channel itself (which is the purpose of the message?). This missing data must be reported as comments, making it challenging to understand the channel's roles, and keeping consistent documentation.

```
channel MyDatatypeChannel
  of type DATA_TYPE, PUBLISH_SUBSCRIBE
  on path "devices/wakeUp"
  accepts message WakeUp
  expecting
    payload WakeUpDto as COMMAND_MESSAGE
    where MESSAGE_EXPIRES in 15m
```

Listing 50: AsyncMDSL channel definition

If we consider the concrete syntax of AsyncAPI, either JSON or YAML, we can stand out that it is not designed to be explicative, as JSON is a

[1] AsyncMDSL just forwards the provided bindings to AsyncAPI generator.
[2] https://github.com/asyncapi/bindings
[3] https://www.asyncapi.com/docs/tooling

data-interchange format, and YAML a superset of it. On the other side, our DSL offers clear keywords — not properties names — that are valid under a specific context. This means that the autocompletion suggestions that our editor proposes ease the work of integration architects, as it is simpler to pick from a small number of suggestions than reading a specification document which lists all the properties available at the current object depth or path.

6.1.3 AsyncMDSL design

AsyncMDSL has been designed based on the patterns described in [23]. Given that EIPs are widely accepted and used, a language that defines those patterns as first-class concepts would express familiar notions to software architects. The language, as MDSL, promotes readability over parsing efficiency, support partial specifications — that can be refined iteratively — and it is not bound to any specific protocol or message exchange format.

6.2 Requirements Evaluation

6.2.1 Requirements coverage

We provided, for each section presenting a language feature, traceability to requirements. To summarize, out of 17 user stories representing a functional requirement, 16 are fully satisfied, and the last one, US-17: Server security, is supported by the grammar but not yet available in the generated AsyncAPI document. Table 6 recaps where users stories have been covered, while non-functional requirements, for which we didn't provide traceability yet, are described next. NFRs can be divided in two categories: the ones that are determined by our implementation, such as NFR-5: AsyncAPI conversion time, and ones that demand evaluators' feedback, such as NFR-1: Usability.

Requirement	Covered in
US-1: Model a message-based system	ServiceSpecification
US-2: Integrate with AsyncAPI	Generating AsyncAPI
US-3: Message Channels	ChannelContract
US-3.1: Point-to-Point Channel	One-way channel
US-3.2: Publish-Subscribe Channel	ChannelContract
US-3.3: Datatype Channel	ChannelContract, Message's payload and header
US-4: Messages	Message's payload and header
US-4.1: Command Message	Message's payload and header
US-4.2: Document Message	Message's payload and header
US-4.3: Event Message	Message's payload and header
US-4.4: Request-Reply Message	Request-Reply channel
US-5: Return Address	Request-Reply channel
US-6: Correlation Identifier	Where clauses
US-7: Message Sequence	Where clauses
US-8: Message Expiration	Where clauses
US-9: Message Endpoints	Message Endpoints
US-10: Competing Consumers	Message Endpoints
US-11: Polling Consumer	Message Endpoints
US-12: Event-Driven Consumer	Message Endpoints
US-13: Selective Consumer	Message Endpoints
US-14: Durable Subscriber	Message Endpoints
US-15: Message Brokers	Message Brokers
US-16: Specify protocol-specific information	Message Brokers, One-way channel and Request-Reply channel
US-17: Server security	Partially in Message Brokers

Table 6: Requirements mapping to language features

NFRs fulfillment by implementation

In the following bullet list, we present NFRs which demand no evaluators' feedback, while in section 6.2.1 we present how we collected evaluators feedback and how they affected the fulfillment of the remaining NFRs.

- NFR-4: Specification's complexity. The MDSL standard grammar is composed by 67 rules, and the AsyncMDSL extension adds 31 new rules, for a total of 98 rules, which is slightly below our requirements limit.
- NFR-5: AsyncAPI conversion time. The example modeled in section 5.1, which contains more channels than the demo project[4] lasts, in average, below a second.
- NFR-6: Maintainability and supportability. The project has clear setup instruction, provided in form of markdown *readmes*, it exploits eclipse built-in linter and it is properly documented following the Javadoc convention.
- NFR-7: License. The project will be distributed under the Apache 2.0 License.

NFRs fulfillment by validation

We have not conducted a representative user test to measure the fulfillment of the NFRs analytically. However, AsyncMDSL has been used by the project's supervisor and students from our university, allowing us to collect initial feedback[5]. Our test users, even if a small number, include both senior software architects (the supervisor) and different master and bachelor students (seven in total). The Table 7 summarizes the impressions and feedback we received. Considering those feedbacks, we can conclude that also the remaining NFRs have been fulfilled:

- NFR-1: Usability. Most of our evaluators confirmed that 15 to 30 minutes are enough to understand examples written in AsyncMDSL.
- NFR-2: Expressiveness. All the evaluators also affirmed that the syntax is easy to read and to understand, provided that it is properly formatted.

[4]https://www.asyncapi.com/docs/tutorials/streetlights

[5]Feedbacks have been collected individually by a direct communication with evaluators, since they did not submitted a validation survey.

Topic	Feedbacks
Syntax	Users found the syntax self-describing and almost always clear. However, we received a few critical feedbacks as well, precisely on the absence of parenthesis to logically identify blocks and scopes and on a missing common syntax for data types, channel path parameters, and protocol bindings. Furthermore, users were expecting autocompletion feature for the correlation identifier path.
Examples	Users mentioned that it is possible to understand examples written in the DSL within 15 to 20 minutes, given that a brief intruduction to message-based architectures is provided.
Features and Tools	A frequent remark is the IDE support. Users are not willing to install Eclipse if it is not their primary IDE. The absence of a formatter has also been pointed out several times.
General	In summary, the feedback regarding the syntax was satisfying, taking into AsyncMDSL is not mature yet. Providing high quality documentation and detailed examples are key factors in being quickly productive.

Table 7: AsyncMDSL evaluators feedbacks

Chapter 7

Conclusions

This thesis introduced a domain-specific language (DSL) featuring some Enterprise Integration Patterns as first-class concepts to allow the creation of message-based system's models. While the different patterns have already been discussed in the literature [23] [20] [52] , this work aims to define how they can be employed to describe an evolving system iteratively. AsyncMDSL can offer a contract-first design approach and support for agile modeling practices [34] as well as generate rich documentation exploiting existing tools. The targeted AsyncAPI specification used to generate documentation or code scaffolding is enriched, allowing the description of patterns that the specification alone can not model, such as Point-to-Point or Request-Reply Channels, allowing for keeping a consistent style also in the documentation.

The implemented DSL provides a way to describe systems, and the models are written in a form that can be processed. Once message routing patterns will be included, automatic tools could systematically identify the architectural smells that possibly violate the design principles, as [10] is doing for microservice-based architectures.

Given that this project's critical requirements and goals were fulfilled, and besides concrete deliverables, this project offered the opportunity to strengthen personal knowledge about message-driven architectures and integration design patterns. Even if EIPs were introduced in the early 2000s, message-driven solutions are spreading more and more thanks to the always increasing diffusion of the Internet of Things (IoT) devices that exploit light asynchronous protocols like MQTT. A specification language that grasps the context's key factors plays a fundamental role in helping both system designers and consumers achieving their goals.

7.1 Future Work

AsyncMDSL is not a full-fledged product and it is still missing crucial features that a modeling language should include in its ecosystem. AsyncMDSL does not cover Message Routing, Message Translation nor System Management patterns defined in [23]. One of the next steps that this project should take is adding the possibility to model routing. The possibility to model messages flow, and thus business logic, could be used inside frameworks as [5], with a similar experience as with [47].

Others features regards visual representation, code formatting and IDE support. Grammar refinement derived from feebacks is also subject of grammar's next releases. They can comprise topics described in Table 7 as well as others targeting the base MDSL language.

Visual representation, in the form of components interactions' diagrams, are an effective mean for software architects. [23] defines a comprehensive notation [55] which covers all aspects of an integration solution. This notation is handy as a precise visual description of a system that can serve as the basis for code generation as part of model-driven architecture [30]. An example of visual modeler with support for EIPs is [15].

The last point AsyncMDSL should take into account is the IDE support. The DSL's editor should also be available in IDE different from Eclipse, such as the popular Visual Studio Code [56], that supports the Language Server Protocol (LSP) [33]. Formatting, rearranging the text in a document to improve the readability without changing the semantic, should as well be supported in the editor.

List of Figures

List of Tables

Listings

Bibliography

[1] Agile Alliance. *User Stories*. agilealliance.org. Dec. 2015. URL: https://www.agilealliance.org/glossary/user-stories (visited on 07/01/2020).

[2] Agile Alliance. *User Story Template for Agile*. agilealliance.org. Dec. 2015. URL: https://www.agilealliance.org/glossary/user-story-template/ (visited on 07/01/2020).

[3] Apache. *Apache Camel*. Apache.org. URL: https://camel.apache.org.

[4] Apache. *Apache FreeMarker*. Apache FreeMarker. 2000. URL: https://freemarker.apache.org/ (visited on 06/09/2020).

[5] Apache. *Java DSL - Apache Camel*. Apache.org. 2020. URL: https://camel.apache.org/manual/latest/java-dsl.html (visited on 05/21/2020).

[6] Apache. *The Apache Velocity Project*. Apache.org. 2003. URL: http://velocity.apache.org/ (visited on 06/11/2020).

[7] Nordic APIs. *AsyncAPI vs OpenAPI: What's The Difference?* Nordic APIs. Sept. 2019. URL: https://nordicapis.com/asyncapi-vs-openapi-whats-the-difference/ (visited on 05/21/2020).

[8] AsyncAPI. *asyncapi/bindings*. GitHub. Aug. 2019. URL: https://github.com/asyncapi/bindings/tree/master/amqp (visited on 05/22/2020).

[9] Guruduth Banavar et al. "A Case for Message Oriented Middleware." In: *Lecture Notes in Computer Science* (1999), pp. 1–17. DOI: 10.1007/3-540-48169-9_1.

[10] Antonio Brogi, Davide Neri, and Jacopo Soldani. "Freshening the Air in Microservices: Resolving Architectural Smells via Refactoring." In: *Lecture Notes in Computer Science* (2020), pp. 17–29. DOI: 10.1007/978-3-030-45989-5_2. (Visited on 06/26/2020).

[11] Frank Buschmann and Et Al. *Pattern-oriented software architecture*. J. Wiley, 2000.

[12] Mike Cohn and Kent Beck. *User stories applied: for agile software development*. Addison-Wesley, , Cop, 2011.

[13] Bettina Druckenmüller and Frank Leymann Betreuer. "Parametrisierung von EAI Patterns (In German)." In: (Feb. 2007).

[14] Sven Efftinge and Miro Spoenemann. *Xtext - Language Engineering Made Easy!* Eclipse.org. 2020. URL: https://www.eclipse.org/Xtext/ (visited on 06/04/2020).

[15] *Enterprise Integration Patterns Diagram Tool*. Visual-paradigm.com. 2020. URL: https://www.visual-paradigm.com/features/enterprise-integration-patterns-diagram-tool/ (visited on 06/26/2020).

[16] Daniel Fernández. *Thymeleaf*. Thymeleaf.org. 2018. URL: https://www.thymeleaf.org/ (visited on 06/11/2020).

[17] Internet Engineering Task Force. *JavaScript Object Notation (JSON) Pointer*. Ietf.org. 2013. URL: https://tools.ietf.org/html/rfc6901 (visited on 06/19/2020).

[18] Cloud Native Computing Foundation. *Production-Grade Container Orchestration*. Kubernetes.io. 2014. URL: https://kubernetes.io/ (visited on 06/11/2020).

[19] The Eclipse Foundation. *EcoreTools - Graphical Modeling for Ecore*. www.eclipse.org. URL: https://www.eclipse.org/ecoretools/ (visited on 07/01/2020).

[20] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley, 2015.

[21] Erich Gamma et al. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, 1994.

[22] Martin Glinz. "A Risk-Based, Value-Oriented Approach to Quality Requirements." In: *IEEE Software* 25 (Mar. 2008), pp. 34–41. DOI: 10.1109/MS.2008.31.

[23] Gregor Hohpe and Bobby Woolf. *Enterprise integration patterns: designing, building and deploying messaging solutions*. Boston Addison-Wesley, 2015.

[24] Async API Initative. *AsyncAPI*. www.asyncapi.com. May 17. URL: https://www.asyncapi.com/ (visited on 05/21/2020).

[25] AsyncAPI Initiative. *Coming from OpenAPI — AsyncAPI Initiative*. Asyncapi.com. 2019. URL: https://www.asyncapi.com/docs/getting-started/coming-from-openapi/ (visited on 06/19/2020).

[26] OpenAPI Initiative. *OpenAPI Specification*. OpenAPI Initiative. URL: https://www.openapis.org/ (visited on 05/21/2020).

[27] *Integration Platform*. MuleSoft. 2020. URL: https://www.mulesoft.com/ (visited on 07/01/2020).

[28] S. C. Johnson. "Lint, a C Program Checker." In: (1978), pp. 78–1273.

[29] Stefan Kapferer and Olaf Zimmermann. "Domain-driven Service Design – Context Modeling, Model Refactoring and Contract Generation." In: *Proc. of the 14th Symposium and Summer School On Service-Oriented Computing - SummerSoC (September 13-19, 2020)*. Springer Communications in Computer and Information Science (CCIS), to appear.

[30] Anneke G Kleppe, Jos B Warmer, and Wim Bast. *MDA explained : the model driven architecture : practice and promise*. Addison-Wesley, 2003.

[31] Pascal Kolb. "Realization of EAI Patterns with Apache Camel." In: (2007).

[32] Garm Lucassen et al. "The Use and Effectiveness of User Stories in Practice." In: *Requirements Engineering: Foundation for Software Quality* (2016), pp. 205–222. DOI: 10.1007/978-3-319-30282-9_14. URL: https://link.springer.com/chapter/10.1007%2F978-3-319-30282-9_14.

[33] Microsoft. *Language Server Protocol Specification*. microsoft.github.io. 2017. URL: https://microsoft.github.io/language-server-protocol/specifications/specification-current/ (visited on 07/01/2020).

[34] Agile Modeling. *Agile Modeling: Effective Practices for Modeling and Documentation*. Agile Modeling. 2020. URL: http://agilemodeling.com/ (visited on 06/26/2020).

[35] Node.js. *Docs — Node.js*. Node.js. 2020. URL: https://nodejs.org/en/docs/ (visited on 06/09/2020).

[36] Terence Parr. *ANTLR*. ANTLR. 1989. URL: https://www.antlr.org/ (visited on 06/08/2020).

[37] Cesare Pautasso and Olaf Zimmermann. "The Web as a Software Connector: Integration Resting on Linked Resources." In: *IEEE Software* 35 (Jan. 2018), pp. 93–98. DOI: 10.1109/ms.2017.4541049.

[38] RAML. *RESTful API Modeling Language*. RAML. URL: https://raml.org/ (visited on 05/21/2020).

[39] Daniel Ritter. "Experiences with Business Process Model and Notation for Modeling Integration Patterns." In: *Modelling Foundations and Applications* (2014), pp. 254–266. DOI: 10.1007/978-3-319-09195-2_17.

[40] Kristopher Sandoval. *7 Protocols Good For Documenting With AsyncAPI — Nordic APIs —*. Nordic APIs, Jan. 2019. URL: https://nordicapis.com/7-protocols-good-for-documenting-with-asyncapi/ (visited on 05/21/2020).

[41] Kristopher Sandoval. *AsyncAPI: 2020's Industry Standard For Messaging APIs? — Nordic APIs —*. Nordic APIs. Jan. 2020. URL: https://nordicapis.com/asyncapi-2020s-industry-standard-for-messaging-apis/ (visited on 05/22/2020).

[42] Thorsten Scheibler and Frank Leymann. "A Framework for Executable Enterprise Application Integration Patterns." In: *Enterprise Interoperability III* (2008), pp. 485–497. DOI: 10.1007/978-1-84800-221-0_38. (Visited on 07/03/2020).

[43] Thorsten Scheibler and Frank Leymann. "From Modelling to Execution of Enterprise Integration Scenarios: The GENIUS Tool." In: *Kommunikation in Verteilten Systemen (KiVS)* (2009), pp. 241–252. DOI: 10.1007/978-3-540-92666-5_20. (Visited on 07/03/2020).

[44] Thorsten Scheibler and Frank Leymann. "Realizing Enterprise Integration Patterns in WebSphere." In: (2005).

[45] Thorsten Scheibler, Ralph Mietzner, and Frank Leymann. "EMod: platform independent modelling, description and enactment of parameterisable EAI patterns." In: *Enterprise Information Systems* 3 (Aug. 2009), pp. 299–317. DOI: 10.1080/17517570903042770. (Visited on 07/03/2020).

[46] JSON Schema. *JSON Schema*. JSON Schema. 2019. URL: http://json-schema.org/ (visited on 06/09/2020).

[47] Spring. *Java DSL*. Spring.io. 2020. URL: https://docs.spring.io/spring-integration/docs/5.1.0.M1/reference/html/java-dsl.html (visited on 05/22/2020).

[48] Spring. *Spring Framework Core*. 2002. URL: https://docs.spring.io/spring/docs/current/spring-framework-reference/pdf/core.pdf (visited on 06/30/2020).

[49] Open standard. *AMQP - Advanced Message Queuing Protocol*. Amqp.org. 2014. URL: https://www.amqp.org/ (visited on 05/23/2020).

[50] Dave Steinberg. *EMF - Eclipse modeling framework*. Addison-Wesley, 2011.

[51] Swagger.io. *Swagger Tools — Swagger*. Swagger.io. 2019. URL: https://swagger.io/.

[52] Sasu Tarkoma. *Publish/subscribe systems : design and principles*. Wiley, 2012.

[53] Mariah Tauer. *TIBCO Teams Up With AsyncAPI to Advance Modern Event-Driven Apps — The TIBCO Blog*. The TIBCO Blog. Aug. 2019. URL: https://www.tibco.com/blog/2019/08/21/tibco-teams-up-with-asyncapi-to-advance-modern-event-driven-apps/ (visited on 05/21/2020).

[54] *The Reactive Manifesto*. Reactivemanifesto.org. 2014. URL: https://www.reactivemanifesto.org/glossary#Message-Driven (visited on 06/29/2020).

[55] *UML Profile for Enterprise Application Integration Specification*. www.omg.org. Mar. 2004. URL: https://www.omg.org/spec/EAI/About-EAI/ (visited on 06/26/2020).

[56] *Visual Studio Code*. Visualstudio.com. Apr. 2016. URL: https://code.visualstudio.com/ (visited on 06/26/2020).

[57] *YAML Ain't Markup Language (YAML™) Version 1.2*. Yaml.org. 2020. URL: https://yaml.org/spec/1.2/spec.html (visited on 06/08/2020).

[58] Xin Yuan. "Prototype for executable EAI patterns." In: (2017). (Visited on 07/11/2020).

[59] Olaf Zimmermann. *Microservice Domain-Specific Language (MDSL) Homepage*. Microservice DSL (MDSL). 2018. URL: https://github.com/Microservice-API-Patterns/MDSL-Specification (visited on 06/18/2020).

[60] Olaf Zimmermann et al. "A Decade of Enterprise Integration Patterns: A Conversation with the Authors." In: *IEEE Software* 33 (Jan. 2016), pp. 13–19. DOI: 10.1109/MS.2016.11.

[61] Olaf Zimmermann et al. *Microservice API Patterns*. microservice-api-patterns.org. 2016. URL: https://microservice-api-patterns.org/ (visited on 06/03/2020).

Appendix A

Language Reference

This appendix lists all the supported patterns AsyncMDSL can model, and their corresponding syntax. The standard MDSL syntax will only be reported as context information where needed.

AsyncMDSL *serviceSpecification*

AsyncMDSL supports only a single root object, a *serviceSpecification*, that represents the description of a message-based system.

```
API description MyServicesSpecification
version "1.0.0"
usage context PUBLIC_API for FRONTEND_INTEGRATION
description "
  This preamble represent a description of a system.
"

```

Listing 51: AsyncMDSL preamble

Properties *version*, *usage context* and *description* are optional.

Version

The `version`, an optional string value, represents an identifier for the version of the current system's specification. The suggested format follows the semantic versioning[1].

[1] https://semver.org/

Foundation Patterns

The *usage context* — inherited from MDSL — represents the Foundation Pattern[2], and follows the template:

```
usage context <ctx> [for <direction>]
```

where `ctx` can assume one of the following values:

- `PUBLIC_API`
- `COMMUNITY_API`
- `SOLUTION_INTERNAL_API`
- or any string value

and `direction`

- `FRONTEND_INTEGRATION`
- `BACKEND_INTEGRATION`
- or any string value.

Description

The `description`, an optional string value that supports markdown syntax when converted to AsyncAPI, is a generic overview of the whole system's purpose.

Data types

Standard MDSL's datatypes, representing Structure Patterns[3], can be defined as children of a *serviceSpecification*, making it possible to reuse them in all the subsequently modeled channels. MDSL datatypes are introduced in section 4.1.2.

ChannelContract

Channel contracts represent a single communication channel (a Message Channel). The shared syntax for Message Channels and Request-Reply Message Channels is characterized by the template:

[2]https://microservice-api-patterns.org/patterns/foundation/
[3]https://microservice-api-patterns.org/patterns/structure/

```
channel <channelName> [of type <channelType> [, <channelType>]*]
```

where `channelName` is a unique name (of type string) that represents the channel in the current system, and `channelType` expresses one of the Message Channel types presented in section 3.1.3. The property `channelType` can assume one of the following values:

- `POINT_TO_POINT`
- `PUBLISH_SUBSCRIBE`
- `DATA_TYPE`
- `INVALID_MESSAGE`
- `DEAD_LETTER`
- `GUARANTEED_DELIVERY`

If a channel is of type `DEAD_LETTER` or `INVALID_MESSAGE`, other types can not be supplied. If a channel is `POINT_TO_POINT`, `PUBLISH_SUBSCRIBE` can not be specified, and same applies for the reverse. Others combinations are considered valid.

OneWayChannel

A *OneWayChannel* represents a single Message Channel. The syntax matches the following schema:

```
channel <channelName>
[of type <channelType> [, <channelType>]*]
[description <channelDescription>]
on path <channelPath>
  [with <parameterName: <parameterType>, <parameterDescription>]
(accepts | produces | accepts and produces)
  message <messageName>
  [description <messageDescription>]
  (delivering | expecting)
    [headers <messageHeaders>]
    payload <messagePayload> [as <messageType>]
  [where <whereClauses>]
  [bindings for <channelProtocol> <bindingsObject>]

```

Listing 52: OneWayChannel syntax

A *OneWayChannel*'s `channelPath` is a string that represents its logical path. It uniquely identifies a channel inside a message-based system (and thus channels can not have duplicates paths). Parameters can be specified in the path by surrounding their name with `${paramName}`, providing a type and a description (exploiting the `with` construct). The type of a parameter (`parameterType`) can assume one of the following values: `bool`, `int`, `long`, `string` or `raw`.

The channel direction, defined in line 6, indicates if a Message Channel will be consuming data (`accepts`), producing data (`produces`) or both (`accepts and produces`). Producing and consuming data in the same channel is possible but not considered a best practice.

The message direction, defined in line 9, is derived from the above `accepts` and `produces` keywords, and it must be specified accordingly (e.g., if a channel `accepts` a message, it will be `expecting` headers and payload, not `delivering` them).

`messageHeader` and `messagePayload` (lines 10 and 11) are MDSL's datatypes that can be referenced (if already defined) or created as anonymous objects. The `messageType` (line 11) can assume one of the following value: `DOCUMENT_MESSAGE`, `EVENT_MESSAGE` or `COMMAND_MESSAGE`, to express different message purposes.

The `where` construct allows us to represent patterns such as Correlation Identifier or Message Sequence. Inside this construct three different clauses can be specified:

- `MESSAGE_EXPIRES in <expirationValue> <expirationUnit>`, for representing Message Expiration. `expirationValue` is an integer value, while the `expirationUnit` can assume `m` for minutes or `s` for seconds
- `SEQUENCE_ID is <expression>`, for representing the Message Sequence. `expression` is a string that must follow the JSON Pointer specification, as described in section 4.3.3.
- `CORRELATION_ID is <expression>`, for representing Correlation Identifier. Also in this case the `expression` is a string that must follow the JSON Pointer specification.

Bindings are the last element of a Message Channel, and allow to specify protocol-specific information, such as the group identifier in Kafka or the exchange name in AMQP. Their syntax highly recalls the one from JSON. The `channelProtocol` can assume one of the following values:

- `MQTT`
- `Kafka`
- `AMQP`
- `STOMP`
- `HTTP`
- `JMS_ActiveMQ`
- or a string value.

Note that the protocol must be supported by AsyncAPI for a successful specification generation. A `bindingsObject` follows the syntax:

```
{
  "<fieldName>": <type>
}

```

where `fieldName` is a string representing the name of the property of the object, and its `type` can be:

- a string value (wrapped between double quotes)
- a boolean value (`true` or `false`)
- another nested `bindingsObject`

Request-Reply Channel

A *Request-Reply Channel* represents a pair of Message Channels, one where a request message is sent, and another where the reply for the request is sent. The syntax is:

```
channel <channelName>
[of type <channelType> [, <channelType>]*]
[description <channelDescription>]
request message <requestMessageName>
  [description <requestMessageDescription>]
  on path <requestChannelPath>
    [with <parameterName: <parameterType>, <parameterDescription>]
  expecting
    [headers <messageHeaders>]
```

```
    payload <messagePayload> [as <messageType>]
    [where <whereClauses>]
    [bindings for <channelProtocol> <bindingsObject>]
reply message <replyMessageName>
  [description <replyMessageDescription>]
  on path <replyChannelPath>
    [with <parameterName: <parameterType>, <parameterDescription>]
  delivering
    [headers <messageHeaders>]
    payload <messagePayload> [as <messageType>]
    [where <whereClauses>]
    [bindings for <channelProtocol> <bindingsObject>]

```

Listing 53: Request-Reply Channel syntax

Properties inside the `request message` and `reply message` are the same as the ones for a *OneWayChannel*, presented above.

Message Brokers

Message Brokers indicate which *serviceSpecification* they intend to offer to their clients. A Message Broker follows the syntax:

```
message broker <brokerName>
[description <brokerDescription>]
exposes
  <serviceSpecification>
  at location <brokerLocation>
  via protocol <brokerProtocol>
  [bindings <bindingsObject>]
  [policy <securityPolicyName> realized using <securityPolicy>
    [in <securityPolicyExpression>]]

```

Listing 54: Message Broker syntax

The `serviceSpecification` is a reference to an existing *serviceSpecification* object, tipically the parent of the message broker itself. Referencing a *serviceSpecification* means that all channels defined in such specification will be exposed by this broker.

The `brokerLocation` is a string that allows clients to connect to the broker. (e.g., the URI where the broker is available *amqp://mybroker-location.com*).

`bindingsObject` follows the same syantax described in the *OneWayChannel* above, without the need to specify again the protocol, as for message brokers it is already defined.

Message brokers can include a simple security policy that indicates to clients how to authenticate. The `securityPolicyName` is a string representing the name of the chosen policy, while `securityPolicy` can be `BASIC_AUTHENITICATION` or `API_KEY`. If the security policy uses `API_KEY` as authentication mechanism, also the `securityPolicyExpression` must be specified. It represents where the message broker will find the API key in the message for authenticating clients. It follows the JSON Pointer specification as defined in section 4.3.3.

Message Endpoints

Message Endpoints are the last type of children of a *serviceSpecification*. A Message Endpoint represents a client connected to a broker, and it follows the syntax:

```
message endpoint <messageEndpointName>
[of type <msgEndpointType> [, <msgEndpointType>]]
[serves as <primaryRole> [, <otherRole>]]
[description <msgEndpointDescription>]
uses
  [channels:
    <channelContract>
      [where consumed if <leftExp> <operator> <rightExp>]
      [via protocol <channelProtocol> ]
  ]
  [
    from <messageBroker>:
      <channelContract>
        [where consumed if <leftExp> <operator> <rightExp>]
  ]
```

Listing 55: Message Ednpoint syntax

A Message Endpoints can be of many types — each representing a different pattern in [23] — and can assume a combination of the following attributes:

- `SELECTIVE_CONSUMER`
- `DURABLE_SUBSCRIBER`

- `POLLING_CONSUMER`
- `EVENT_DRIVEN_CONSUMER`
- `IDEMPOTENT_RECEIVER`
- `TRANSACTIONAL_CLIENT`
- `MESSAGING_GATEWAY`
- `MESSAGING_MAPPER`
- `COMPETEING_CONSUMER`
- `MESSAGE_DISPATCHER`
- `SERRVICE_ACTIVATOR`

`primaryRole` and `otherRole`, which represent Responsibility patterns[4], can assume one of the following values:

- `PROCESSING_RESOURCE`
- `INFORMATION_HOLDER_RESOURCE`
- `OPERATIONAL_DATA_HOLDER`
- `MASTER_DATA_HOLDER`
- `REFERENCE_DATA_HOLDER`
- `DATA_TRANSFER_RESOURCE`
- `LINK_LOOKUP_RESOURCE`
- or a string value.

A Message Endpoint can use Message Channels exposed by a Message Broker, or can directly reference channels defined in a *serviceSpecification* without the need to specify an actual Message Broker instance. Channels which are not consumed from a Message Broker are declared using the `channels` construct, while the ones referenced from a Message Broker are specified under the `from` construct.

A Selective Consumer is defined exploiting the `where` construct: `leftExp` represent a JSON Pointer that will be evaluated at runtime, the `operator` is a binary operator (`==`,`<`,`>`,`>=` or `<=`) and the `rightExp` can be another JSON Pointer (that will be evaluated at runtime too), an integer value or a string.

[4]https://microservice-api-patterns.org/patterns/responsibility/

Appendix B

Loan Broker Conversion

The Loan Broker example, modeled in chapter 5.1, once converted to AsyncAPI by our generator, will appear as follows:

```yaml
asyncapi: '2.0.0'
info:
  title: LoanBrokerExample
  version: "1.0.0
  description: |
    This example models parts of an EIP scenario.
    See [here](https://www.enterpriseintegrationpatterns.com/
    patterns/messaging/SystemManagementExample.html)
    for more info
servers:
  LoanBrokerAmqpProvider:
    url: amqp.loanbroker.com
    protocol: AMQP
    description: No description specified
channels:
  loan-broker/request:
    subscribe:
      description: |
        This channel is used by a customer to make a request.

        Request channel. Reply channel is
        [LoanReply](#operation-publish-loan-broker/reply)
      operationId: loanRequest
      message:
        $ref: '#/components/messages/LoanRequest'
  loan-broker/reply:
    publish:
      description: |
        The loan broker will reply to the customer after
```

```
        having contacted all the banks and found the best quote.

        Reply channel. Request channel is
        [LoanRequest](#operation-subscribe-loan-broker/request)

        Where:
            - CORRELATION_ID is "$message.payload#/requestId"
            - MESSAGE_EXPIRES in 60m
      operationId: loanReply
      message:
        $ref: '#/components/messages/LoanReply'
  credit-bureau/request:
    subscribe:
      description: |
        Request the credit score and customer history.

        Request channel. Reply channel is
        [CreditScoreReply](#operation-publish-credit-bureau/reply)
      operationId: creditScoreRequest
      message:
        $ref: '#/components/messages/CreditScoreRequest'
  credit-bureau/reply:
    publish:
      description: |
        Return the credit score and customer history

        Reply channel. Request channel is
        [CreditScoreRequest](#operation-subscribe-credit
                                          -bureau/request)

        Where:
          - CORRELATION_ID is "$message.payload#/requestId"
      operationId: creditScoreReply
      message:
        $ref: '#/components/messages/CreditScoreReply'
  banks/${bankId}/loans/request:
    parameters:
      bankId:
        description: The identifier of the bank to contact
        schema:
          type: number
    subscribe:
      description: |
        Request a loan proposal

```

```
75          Request channel. Reply channel is
76          [LoanProposalReply](#operation-publish-banks
77                                              /${bankId}/loans/reply)
78        operationId: loanProposalRequest
79        message:
80          $ref: '#/components/messages/LoanProposalRequest'
81    banks/${bankId}/loans/reply:
82      parameters:
83        bankId:
84          description: |
85              The identifier of the bank that replied
86          schema:
87            type: number
88      publish:
89        description: |
90          The loan proposal for the given customer
91
92          Reply channel. Request channel is
93          [LoanProposalRequest](#operation-subscribe-banks
94                                              /${bankId}/loans/request)
95
96        operationId: loanProposalReply
97        message:
98          $ref: '#/components/messages/LoanProposalReply'
99    banks/${bankId}/loans:
100     parameters:
101       bankId:
102         description: |
103             The bank from which the loan has been requested.
104         schema:
105           type: number
106     publish:
107       description: |
108         Subscribe to be notified when a new loan request happens.
109
110         One way channel (does not expect reply).
111       operationId: newLoanRequested
112       message:
113         $ref: '#/components/messages/NewLoanRequested'
114 components:
115   messages:
116     NewLoanRequested:
117       name: NewLoanRequested
118       title: New Loan Requested
119       description: |
```

```
120         No description specified
121       payload:
122         type: object
123         required:
124           -  timestamp
125         properties:
126           timestamp:
127             type: string
128     LoanRequest:
129       name: LoanRequest
130       title: Loan Request
131       description: |
132         This channel is used by a customer to make a request.
133
134         Request message. Reply message is *LoanReply*.
135       payload:
136           $ref: '#/components/schemas/LoanRequestDto'
137     LoanReply:
138       name: LoanReply
139       title: Loan Reply
140       description: |
141         The loan broker will reply to the customer
142         after having contacted all the
143         banks and found the best quote.
144
145         Reply message. Request message is *LoanRequest*.
146       correlationId:
147         location: '$message.payload#/requestId'
148       payload:
149           $ref: '#/components/schemas/LoanReplyDto'
150       headers:
151          unnamedParameter4:
152            $ref: '#/components/schemas/CommonHeaders'
153     CreditScoreRequest:
154       name: CreditScoreRequest
155       title: Credit Score Request
156       description: |
157         Request the credit score and customer history.
158
159         Request message. Reply message is *CreditScoreReply*.
160       payload:
161           $ref: '#/components/schemas/CreditBureauRequestDto'
162       headers:
163          unnamedParameter5:
164            $ref: '#/components/schemas/CommonHeaders'
```

```
    CreditScoreReply:
      name: CreditScoreReply
      title: Credit Score Reply
      description: |
        Return the credit score and customer history

        Reply message. Request message is *CreditScoreRequest*.
      correlationId:
        location: '$message.payload#/requestId'
      payload:
          $ref: '#/components/schemas/CreditBureauReplyDto'
    LoanProposalRequest:
      name: LoanProposalRequest
      title: Loan Proposal Request
      description: |
        Request a loan proposal

        Request message. Reply message is *LoanProposalReply*.

      payload:
          $ref: '#/components/schemas/BankLoanRequest'
    LoanProposalReply:
      name: LoanProposalReply
      title: Loan Proposal Reply
      description: |
        The loan proposal for the given customer

        Reply message. Request message is *LoanProposalRequest*.
      payload:
          $ref: '#/components/schemas/BankLoanReply'
  schemas:
    LoanRequestDto:
      type: object
      required:
        -  socialSecurityNumber
        -  amount
        -  termInMonths
        -  requestId
      properties:
        socialSecurityNumber:
          type: number
        amount:
          type: number
        termInMonths:
          type: number
```

```
      requestId:
        type: number
  LoanReplyDto:
    type: object
    required:
      - quoteId
      - interestRate
      - requestId
    properties:
      quoteId:
        type: number
      interestRate:
        type: number
      requestId:
        type: number
  CreditBureauRequestDto:
    type: object
    required:
      - socialSecurityNumber
      - requestId
    properties:
      socialSecurityNumber:
        type: number
      requestId:
        type: number
  CreditBureauReplyDto:
    type: object
    required:
      - socialSecurityNumber
      - creditScore
      - requestId
    properties:
      socialSecurityNumber:
        type: number
      creditScore:
        type: number
      creditHistory:
        type: array
        items:
          type: object
          properties:
            additionalProperties:
              type: string
      requestId:
        type: number
```

```
    BankLoanRequest:
      type: object
      required:
        - creditScore
        - creditHistoryLength
        - requestId
      properties:
        creditScore:
          type: number
        creditHistoryLength:
          type: number
        requestId:
          type: number
    BankLoanReply:
      type: object
      required:
        - quoteId
        - interestRate
        - requestId
      properties:
        quoteId:
          type: number
        interestRate:
          type: number
        requestId:
          type: number
    CommonHeaders:
      type: object
      required:
        - brokerId
      properties:
        brokerId:
          type: number
```

Listing 56: Loan Broker example converted to AsyncAPI

If we run the *async-api-generator*[1] specifying as input the content shown in Listing 56 and using the *html-template*[2], we obtain a static website for documentation purposes, as illustrated in Figure 7.

[1]https://github.com/asyncapi/generator
[2]https://github.com/asyncapi/html-template

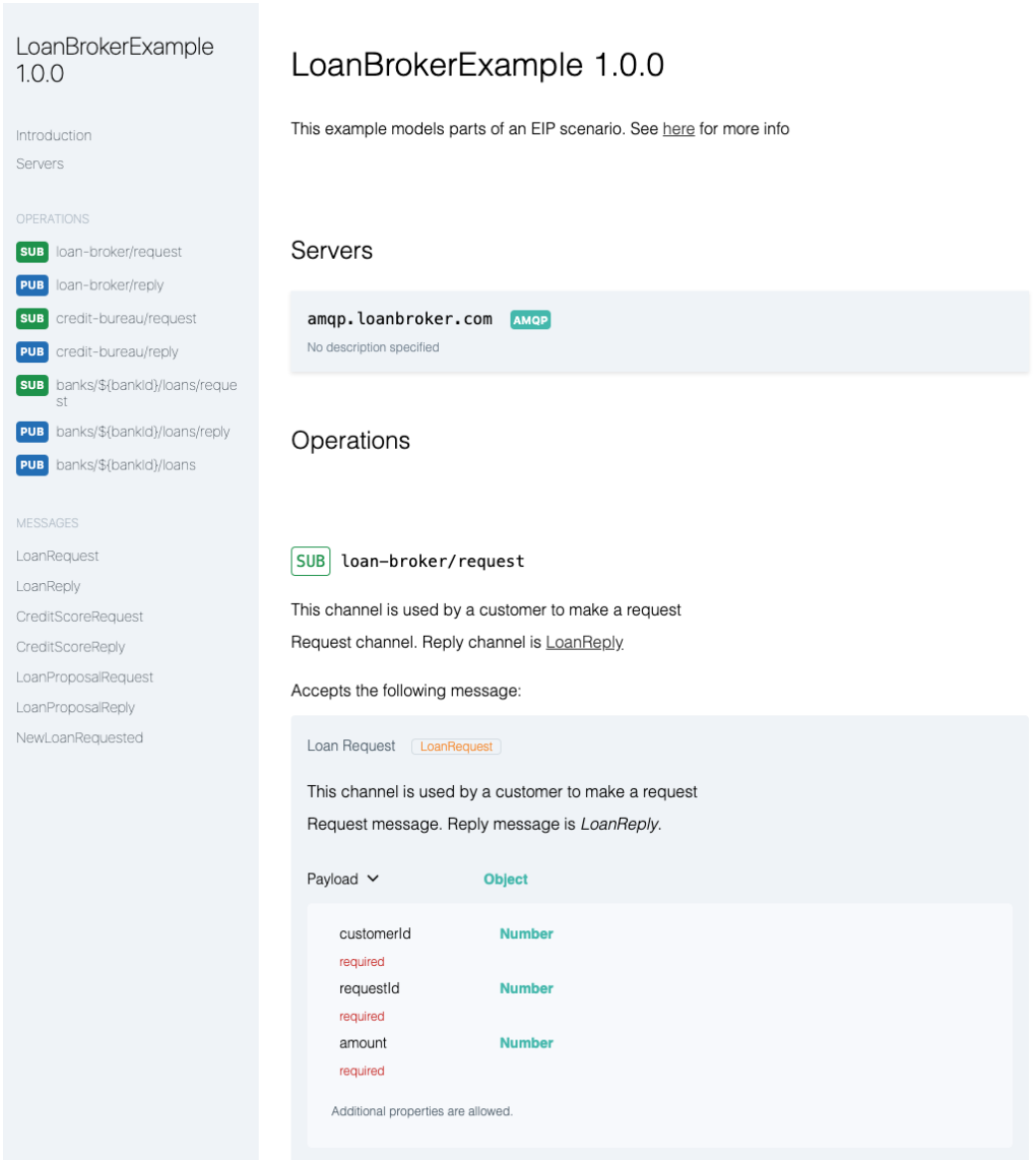

Figure 7: AsyncAPI documentation generation for the Loan Broker example