UNIVERSITÀ DEGLI STUDI
DI GENOVA

Semantic Driven Modeling

Master's degree thesis in Electronic Engineering
Tesi di laurea in Ingegneria Elettronica

Author: Raffaele Rialdi
Email: rrialdi@gmail.com

Publication date: December, 2021
Registration number: 1208188

Supervisors:
Professor Pierpaolo Baglietto (UniGe)
Professor Massimo Maresca (UniGe)

This page intentionally left blank

Foreword

Since I was a child in the primary school, I always had a great passion for hacking electronics. My mother taught me how to use the basic tools like the screwdriver and a giant soldering gun that I used to do my own audio cables. But beyond the basics, I didn't have any opportunity to understand how electronic works. Even at the amatorial level, it was hard to discover the secrets of the electronics alone. At that time, there was no Internet and the books I could find were too advanced for a child to understand the basics without a teacher.

Later on, my father gave me a Sinclair ZX81 which appeared mind-blowing to me. Given the total absence of commercial software, I self-taught the BASIC programming language just trying all the available instructions. I discovered my software programming passion which never kicked away the passion for electronics. I started interfacing the ZX81 first and later on the Sinclair Spectrum using the serial port with a printer and my passion grew exponentially.

I had many other similar experiences but I was always aware of the fact that self-teaching was not the right way. In the primary school I already knew inside of me that I really wanted to be an Engineer. I wrote about this profession in a short essay and my parents still remember it.

Right after my graduation, I was already pretty good and confident in software programming but still very weak in electronics. I was aware I could have a slightly easier life at the University by choosing Computer Engineering instead of Electronics Engineering, but my goal was different. I really wanted to deeply learn electronics and understanding all the theory that was behind the things I hacked since I was a child.

In the eighties, the Engineering University in Genoa was definitely hard but also very valuable. Hundreds of students in very large rooms with zero audio amplification, long queues to enter the university at 6 am in the morning and anybody after the 10th row had small chances to hear the professor voice. In these years I was leaving alone in my parents' house because they went abroad for work. But I did it, I passed the majority of the most difficult exams and entered the last three years.

In the meanwhile, my software skills were growing over time. While studying, I fully embraced the "8088" PC learning its assembly code, programming in C and also the "experimental language" C++, fully embracing the object orientation. I wrote a software for a very popular digitizer (which is now in a computer museum in the Verona University), migrated an Amiga video-game to the PC 80286 and wrote "Terminate and Stay Resident" interrupt-based programs for the PC. I did amazing discoveries and made a lot of experiences which determined a solid personal growth which still have a great value in my current work.

One exam was extremely long but special. Other two students and I decided to write a book for the "**Elettronica Applicata**" course with Professor Alessandro Chiabrera. Writing in LaTex with lots of formulas and drawing all the graphs was dramatic and resulted in 670 pages of small-sized font book. I remember Professor Chiabrera lessons in my heart. He had an incredible passion for the "bio" future of electronics and I was deeply saddened when I later knew he was passed away.

My life was split between writing software and studying. I totally disrupted my days and nights. I used to spend night and days either writing software at home or in the DIBE laboratories to learn electronics and work on my exams. I used to spend far more than anyone else in the DIBE building discussing with Professors Parodi, Donzellini, Bisio and many others on an incredibly large number of awesome topics.

I was not too far from finishing my university dream when an incredible working opportunity knocked at my door. I talked with some of the professors, and they talked to me not as professors but as they were my parents. They were very sad for me leaving the university and warned me that coming back later was a "mission-impossible". They were absolutely right. I got a marvelous wife and two children, we worked hard for our house and meanwhile the time passes fighting all the daily battles that adult life presents and challenges.

Over the years, I always told to myself I was blessed to have had the opportunity to work in my life with the things I dreamed of, since I was a child. As a consultant, I had the opportunity to work in many different industries. I became an instructor in training courses first, and later a speaker in international conferences. Over this time, since 2003 Microsoft every year evaluates and recognizes me the special MVP (Most Valuable Professional) Award in security and developer technologies. I spent an incredibly amount of astonishing and rewarding years fulfilling a lot of my dreams, but my very first was still missing.

Over the years I did find some time to pass one exam, then another one but the required amount of time was not sufficient to finish my studies. I was busy with my work, travels, and of course studying which is mandatory for anyone, but absolutely necessary for anyone working in the software industry.

Even if I had an incredible professional software career, I still spend some of my spare time in my personal electronics lab where I have professional tools like air soldering, 3D microscope for SMD-sized components, 3D printer, oscilloscopes and a spectrum analyzer which I also used to prepare the "Comunicazioni Elettriche" exam few months ago. As I always did, I need to study, observe and understand to make the topic mine.

The recent pandemic had a tragic balance but gave me the opportunity to use the spare-time to focus on my last seven exams. Initially I was scared not to be able to ride again my math as everybody is able to do with bicycles. Despite that, I was able to pass the most scaring exams with the highest scores, which is a solid proof that my studying abilities has grown over the years. Studying is for me what the adrenaline is for a professional sportsman.

I am a professional software guy but with a profound passion for electronics. Choosing the topic for my thesis was not an easy task. I really wanted to choose something innovative and providing a concrete help. Since my whole experience is in software, I decided for an ambitious topic.

My thesis aims to provide a concrete answer to the versioning issues in distributed services. This is something that a lot of my customers faced and could not resolve easily or completely. I already presented the Semantic Driven Modeling idea to 5 international conferences, received a lot of feedback from attendees and colleagues, discussed in a conference panel with illustrious architects working for well-known leading cloud-providers.

Semantic Driven Modeling is not just a theory. I wrote about 15'000 lines of .NET code implementing all the required algorithms and tools to provide a concrete solution. The code makes use of the most advanced software techniques such as the creation of a new, simple Domain Specific Language and dynamically generated code. Furthermore, it fully respects REST principles and the overall micro-benchmarks demonstrate it is extremely efficient.

Even if this is just the short version of my university travel, it may be enough to explain why I want to dedicate my master degree in Electronic Engineering to myself and my university.

This is not a finish line, I will continue hacking in software and hardware as I always did.

Raffaele

Abstract

This master's degree thesis deals with a proposal to solve a very common problem in modern software architecture: versioning the public models of the services that makes up a distributed application.

Every distributed system is, by definition, composed by different components living in a separate process. The goal of the application's architecture is to provide the best possible orchestration of those components so that the application requirements can be fulfilled. Decoupling is the first step to fulfill a number of advantages such as scalability, reliability, availability, component reuse and easier maintenance.

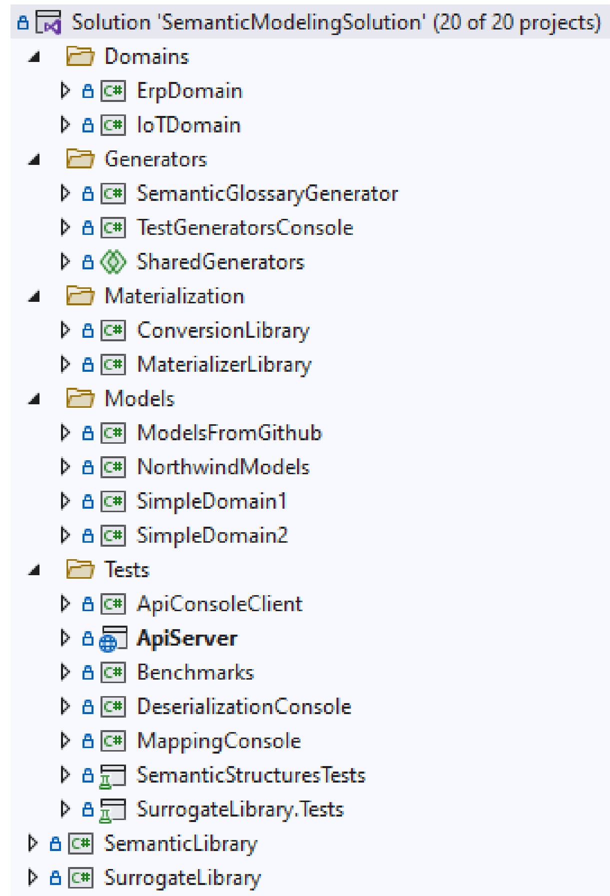
Anyway, regardless the strategy chosen to decouple those components, the knowledge of the data structures exposed by a component inherently causes a dependency in the consumer components. The friction caused by those dependencies resides in the public model changes that may occur over time.

Our goal is to provide the design strategy and tools that allows to freely evolve the public model used in a closed environment and to easily define the policies, rules and validators that ensure the model semantic correctness.

This thesis is structured in two sections. The first analyzes how the public model is affected by versioning in the distributed systems. The second exposes the proposed solution in both architectural and technical perspectives, concluding with an analysis on the practical outcomes that may derive from this work.

Proof of Concept code

The Semantic Driven Modeling was implemented in a Proof of Concept which is made of 20 Projects and 15'000 lines of code.



Presentations in international conferences

Worldwide Software Architecture Summit (August 4, 2021)

Title: *Easing the rough edges of distributed computing with code generation techniques*

CodeCamp Romania (September 23, 2021)

Title: *Introducing Semantic Driven Modeling (SDM), the solution to versioning issues in distributed apps*

DotNext Moscow (October 22, 2021)

Title: *Overcome model versioning nightmare using Semantic Driven Modeling (SDM) in distributed systems*

Windows Professional Conference (WPC) Milano (October 26, 2021)

Title: *Risolvere l'incubo del versioning in applicazioni distribuite usando il Semantic Driven Modeling (SDM)*

CodeMotion Europe (December 1, 2021)

Title: *Easing the rough edges of distributed computing with code generation techniques*

Table of Contents

- Foreword (page 1)
- Abstract (page 5)
- Proof of Concept Code (page 6)
- Presentations in international conferences (page 7)
- Table of Contents (page 8)
- Introduction (page 11)
- Part I
  - Chapter 1. Versioning and distributed systems (page 21)
    - Model and contracts in the distributed systems context (page 22)
    - Versioning in the modern architectural styles (page 25)
      - Command Query Responsibility Segregation Pattern (page 26)
      - Domain Driven Design (DDD) (page 26)
    - Service communication patterns (page 27)
    - Versioning over time and the data storage (page 30)
  - Chapter 2. Model analysis (page 33)
    - One business, different models (page 33)
      - Where the transformation should happen (page 36)
      - Dealing with the model differences (page 37)
      - The behavior of standard serializers (page 38)
    - The nature of the model changes (page 39)
- Part II
  - Chapter 3. Abstracting the model (page 42)
    - Terms (page 43)
      - Domain glossary (page 44)
    - Concepts and Weights (page 44)
    - Concept specifiers (page 45)
    - The domain (page 45)

        - Domain Policies (page 47)
        - Domain Rules (page 47)
        - Semantic validations (page 48)
    - Creating the domain (page 48)
    - The semantic layer (page 49)
- Chapter 4. The Domain Specific Language (page 51)
    - Example 1 (page 52)
    - Example 2 (page 54)
    - Example 3 (page 54)
    - Creating the text file (page 54)
    - The generated code (page 55)
    - The domain language (page 56)
- Chapter 5. Semantic metadata (page 57)
    - The type system abstraction layer (page 57)
        - Navigating the model graph (page 59)
    - The semantic layer (page 61)
    - Mapping two models (page 63)
    - Automatic mapping strategies (page 65)
        - Custom Rules (page 67)
        - Context evaluation (page 67)
        - Assigning the score (page 68)
        - Saving the results (page 68)
    - Other ways to build the mappings (page 69)
- Chapter 6. Custom Serialization (page 70)
    - The serializer algorithm (page 71)
        - Serializing objects (page 72)
        - Serializing collections (page 72)
        - Serializing other data types (page 73)
        - Accessing the object instances in the graph (page 73)
        - Null management (page 74)
        - Serializing the object graph (page 74)
    - The deserializer algorithm (page 75)
        - Creating object instances (page 76)

        - Removing objects from the cache (page 77)
        - Convert and assign the value (page 78)
    - Conversion Engine (page 79)
    - Performance considerations (page 81)
        - Profiling performance (page 82)
            - Serialization measures (page 83)
            - Deserialization measures (page 83)
- Chapter 7. Content negotiation in practical scenarios (page 85)
    - Standard HTTP negotiation (page 85)
        - Versioning: media type or custom headers (page 86)
    - Scenario 1. Server publishes multiple versions using only the latest model (page 87)
    - Scenario 2. Server-side conversion of different client versions (page 88)
    - Scenario 3. Dynamically exposing multiple reading models (page 90)
    - Scenario 4. Server-side versions converted on the client-side (page 91)
- Chapter 8. Conclusions (page 93)
- Bibliography (page 98)

Introduction

The work presented in this thesis deals with the public model versioning issues that every distributed system may hit during their lifecycle and present the full-featured implementation code representing one possible solution to resolve the problem.

There is a general consensus among all the modern software architectures in decomposing an application into small, independent services that provide specific functionalities. For example, an e-commerce application can be structured into a series of independent services such as "User management", "Orders", "Price lists" and "Payments". Each service acts as a server by exposing an API to its "clients" but it can also be a client of other services.

The relationship between the client, which consumes the API, and the producing server must adhere to a shared contract so that data can be exchanged and processed. The two parties are required to share the knowledge of the protocol, the endpoint and the schema of the data being exchanged, which is also called the public model. For the sake of this work, we focused on the public model because it represents the data structure and it is often subject to changes over time. Anyway, in the last chapter of this document we will briefly talk about the possible improvements, including the ability to use the semantic approach to map the endpoint as well.

The nature of changes in a model may be of three different types:

- Adding, removing or renaming the name of a field. For example, renaming the field "Weight" to "NetWeight" to improve clarity or to avoid ambiguities with the introduction of "GrossWeight".
- Modifying the data type for the field. For example, a lot identifier can be changed from integer to string to support alphanumeric coding. Another example is to change the temperature readings from 16-bit floating point to the decimal type to improve its precision.
- Re-structuring the model graph. If in a first release the company address was expressed in three string fields like "Street", "ZipCode" and "City", they could be moved into a separate type named "Address" to make it possible reusing it in other APIs.

Those changes often are made in preparation to a new feature being implemented, but sometimes are just done to improve readability or to optimize the size of its serialization.

In any case, as soon as any of those changes are performed by the model owner, the contract is broken and its clients need to be modified in order to understand those changes and restore the functionality. This translates in a domino effect because every time we need to modify a service, we also need to modify and re-deploy all its clients at the same time.

When a service needs to be updated to fix a bug or add a new functionality, its availability can be still guaranteed by mean of a redundant infrastructure like, for example, Kubernetes. In fact, the old code can run until the new one is ready to be started. The infrastructure can then start the new version of the service, divert all the calls to the new instance and finally shut down the old one. This allows a smooth transition without any hiccups in the service availability.

On the contrary, when the service contract is broken, this strategy cannot be applied because the two parties, client and server, must be rebooted at the same time, causing a troublesome service interruption. This is normally addressed by introducing a new endpoint supporting the new version of the contract while maintaining active the old one. The downside of this solution is the need to duplicate the infrastructure needed for the redundancy. This infrastructure can be easily configured when hosting the application in the Cloud, but it comes with an additional cost, not to count the possibility of a human error in the configuration.

Being able to update a service functionality without interruption is often critical like the emblematic case of the e-commerce where the interruption of the payment service would prevent its users to complete a purchase.

Another interesting scenario is where the data is stored in its serialized form over time. The most popular case is in architectures based on Event Sourcing[1] which has the peculiarity of storing the sequence of commands needed to change some data, rather than updating the data. You can think about Event Sourcing as a traditional "transaction log" of a relational database that will be committed at a later time to the database tables. The Event Sourcing strategy maintain this historical sequence of changes for the time on and uses a different database, asynchronously updated, to provide the current state of the data. While the "transaction log" is just an internal implementation detail, the sequence of commands in Event Sourcing are available to the application for any kind of query. This means that we need must be able to understand all the previous versions of the model persisted in the Event Sourcing storage.

Another similar issue happens when the data transits over a stored queue. This data is accepted from the queuing system, serialized and stored. At a later time, some other service will retrieve the data and process it. In those scenarios the data processing does not happen in real-time making very difficult to pick the right time to update the data consumer code.

Other time-dependent issues may arise when not all the clients can be updated at the same time. Some of them may be managed in different time-zones, some others may be temporarily turned off in a virtual machine and unreachable.

These are only few examples debunking the illusions that a service contract can be everlasting or that can be easily updated.

Our goal is to move the contract to a higher level so that the public models become an implementation detail that we are totally free to modify. This goal can be reached by:

- Abstracting the type system in order not to depend on a specific programming language.
- Introducing a new semantic abstraction layer and its parameters.
- Enforcing the rules which ensure the new contract integrity.
- Adopting the libraries required to transform a model into another.

For this thesis, we coined the name "Semantic Driven Modeling" to define a design approach that shapes the public models starting from their semantic abstraction, following a small set of rules.

The idea is to formalize what each data field means, and how types and fields are related to each other. For example, it is intuitive understanding what the Temperature field in the Sampling class means. It is intuitive for a human being but there is not enough information for processing it on a computer. Anyway, we don't want to introduce any requirements involving the human adhering to a schema. There is of course a classification step, but this can be completely automated and we believe that it must be frictionless for the developer and architect by reducing the effort in declaring the meaning of each term to the minimum.

If a service publishes a Temperature field but the client model has instead a Heat field, they can be easily mapped if they live in a domain where an agreement exists on linking those equivalent terms to the same concept. The big assumption here consists in both the client and the server sharing the knowledge of the same concepts which directly reflect

what they are able to process in their internal business-logic code.

In our experience, we have observed that the code needed to map a client to the new version of the server is almost always straightforward. This is true until the semantic of the data expressed by the server model stays the same. In the e-commerce example, we all expect that a payment service continues to deal with prices, credit cards, discount and any other concept is behind the model structure used to represent the data.

While straightforward, the required manual changes to the client code can be very error-prone. Remembering the three types of differences that a model change may present, many possible issues may arise. When a field is renamed, the developer may just forget to copy it resulting in a loss of data. If the data type was changed, a wrong conversion may result in a runtime exception. Structural graph changes may be tricky to implement, in particular when the data is part of a one-to-many relationship in a collection.

The experience also tells us that, even with the effort of providing a decent amount of code coverage in unit-testing, there are changes that may be very difficult to test, not to count the trade-offs due to time constraints.

Over the years, having matured a deep knowledge of code generation, we always wondered how to deal with these kinds of problems and we tried to imagine what is the knowledge needed by a developer to do the manual work.

The main idea behind Semantic Driven Modeling is to provide all the required information to an algorithm that can guess the best mappings and eventually make a proposal to the designer for approval. This information allows the client to re-generate the required code dynamically, at run time, without having to even restart the service.

In the literature there are two different approaches that have been proposed to address the model transformation problem: ontology based and model based.

The work proposed in "Ontology-driven Semantic Mapping"[17] suggests to extract the information required for two systems belonging to different sources. The authors' proposal is called "Logical Data Model ontology" and consists in abstracting the ontology by mean of annotations. Anyway, we could not find any detailed information about the steps needed to reach their goals or a reference implementation. The example in the article shows a relational-database like approach using OWL relationships. In Semantic Driven Modeling we rely on a model graph which preserves the conceptual meaning of the entire path going from the root entity to the leaves of the graph.

A different model-based approach is instead suggested by the Object Management Group, called "Model Driven Architecture"[15]. Their goal is the «representation and exchange of models in a variety of modeling languages, the transformation of models, the production of stakeholder documentation, and the execution of models.». They substantially says to abstract the model in a modeling language such as, for example, UML or OWL[3] which uses "subject", "predicate" and "object" triplets to obtain logical relationships.

These two approaches are compared in a paper: "Semantic mapping: ontology-based vs. model-based approach Alternative or complementary approaches?"[16]. While the general idea of extracting information from the semantic expressed by the models is the idea shared between them and SDM, there are still significant differences in the approach that SDM propose in terms of the effort necessary to reach the goal. For example, we don't aim to transform the model straight into another model because its representation could have a great complexity and therefore would require a lot of metadata that necessarily involves more human-driven assistance. Instead, SDM focuses on the wire-format of the communication channel of the distributed application which represents a lightweight format to express the data shape. This allows to dramatically simplify the effort needed by a human supervision. The semantic information obtained by SDM is also simpler and relies on the hierarchy rather than triplets or pre-defined schemas.

On the ontology side, the standards cited by the article such as RDF (Resource Description Framework)[2] standardized by W3C whose schema (RDFS) and OWL[3], are schema based and require a great effort in terms of classification needed to be compliant to those standards. The human-driven approach is not new. For example, a standard like the WHATWG HTML[4] specification allows search engines to extract metadata from a web page in order to provide better indexing. Schema based approaches are valuable in terms of high-quality classification but, many companies are not **willing** in procedures requiring the human-intervention in providing structured information.

In the Semantic Driven Modeling we start from the opposite direction which consists in modeling the public model with a semantic-first approach which ensures the terms and naming conventions used in the model definition are conformant to the language used in that specific business domain. The quality and quantity of the "required information" needed by Semantic Driven Modeling were chosen with a practical trade-off in mind.

The Semantic Driven Modeling does not require any schema or operative changes to the way the model is designed, but there is of course a price to pay for the total absence of additional information that can be summarized in the following rules:

- Ensuring a discipline is applied to in the choice of terms used to name the types and fields of the public model. They must be meaningful and possibly avoid any ambiguity.
- The second is restricting the adoption of this technique to just those services that have been modeled using the same principles. In a distributed system, this requirement is easy to achieve since every service is functional to the same application.
- For each domain of application, the developer is required to associate the terms used in the class and field names to a concept.

The first rule has a very low impact on development because it is very popular for developers adopting naming conventions and terms that clearly identify classes and fields. Therefore, in a Person class it is quite clear what the FirstName and LastName fields mean.

Interestingly, this requirement is totally frictionless for applications developed using the Domain Driven Design[5]. In fact, one of the main points in DDD is defining the so-called Ubiquitous Language[6] which consists in carefully choosing the appropriate term to define each class and field name as they should be representative of the domain being abstracted.

Since we shifted the contract at an upper level, the second rule should sound obvious. The new abstraction level is conceptual where one or more terms are synonyms for a well-known concept that is associated to a class or field. Those concepts and glossary of terms are needed in order to provide the candidate mappings between the models to be mapped. Let's be very clear, the absence of those conceptual metadata does not prevent a client and a server to communicate, but some error-prone manual work would be required.

The third and last rule defines the minimum requirements for the Semantic Driven Modeling to work. This effort is required only once for all the parties that deals with the same "domain". The domain is a sort of scope that includes all the services sharing the same concepts. In the e-commerce example, we may define a single domain that includes all the terms used in the e-commerce business. But it may also happen to define two or more domains for a single application. For example, the administrative services to manage users or the internal application infrastructure is unrelated with the e-commerce business and therefore it makes sense defining a separate administrative domain.

Defining the parameters becomes a central point of the whole system. The development team must declare the terms, concepts and the other parameters and link them together so that the basic vocabulary of the semantic abstraction can be defined. At a first sight this might look a long, hard and error-prone work.

In order to simplify this task and make it straightforward, we defined a new simple language that just requires to list the concepts along with the terms and the other parameters in a text file. The language syntax is very simple and thanks to a basic compiler developed ad-hoc for this task, all the information contained in the text file are converted in the Microsoft .NET code. This domain-specific language (DSL) dramatically simplifies the construction of the basic parameters and, since it generates .NET code that is then compiled along the rest of the code, it also validates the parameters.

The development team just has to write a simple text file by simply grouping the terms by their meaning which is the concept they represent, while the domain-specific language uses that file to dynamically generate all the required code.

The initial list of terms may eventually be captured by writing a tool that uses introspection to read all the class and property names from the existing code (the Proof of Concept uses .NET reflection library). This means that the development team task can be further reduced to just grouping those terms by concept.

The preparation work is functional to enrich the metadata of the models with the lowest possible effort. At this point, the application can benefit from the information associated with the model definitions in two different stages of its lifecycle.

At development time the developer can write and enforce rules based on the strongly typed classes generated from the DSL. For example, we can create a rule stating that any field associated to the concept of "weight" should be always expressed in any model using a "decimal" data type rather than a floating point. Other rules may include that any class declaring a field identified as Money should also declare tax information. The power of the DSL allows declaring not only rules, but validations to make sure there are no unknown concepts that potentially compromise the runtime functionalities.

In a closed environment, which is what happens in a distributed system, when the server model changes, all the clients are preemptively tested for a successful mapping, avoiding any potential issue at runtime.

The metadata is structured in a format that can be used across programming framework and languages. It can be exposed in a separate endpoint as it happens for OpenAPI[7] in REST[8] or WSDL[9] in SOAP[10]. When a client makes a request to the server, it verifies the metadata version and, when different, it downloads it and computes or loads the new mappings based on the new conceptual level.

The client can therefore dynamically generate and load in memory the new code to map the data being pushed or pulled from the server. This code runs at the boundary during either the serialization or deserialization phase, in order to minimize the memory pressure and execution timings.

The code provided in the proof of concept generates this code using the .NET expression trees. This technique was used to demonstrate the mappings do not affect the system performance. Even better the serialization code is, in most of the cases, more efficient than a traditional serialization, because the generated code contains specific optimizations hinted by the metadata.

The same mappings can also be done using traditional code for programming languages that do not have code-generation capabilities.

The libraries and tools developed for the Semantic Driven Modeling (SDM) proof of concept include:

- A Domain Specific Language that aims to help the developers and architects in tagging the semantics and concepts expressed in a model. The outcomes are auto-generated classes for the specified domain that can be used to enforce rules or validate the model structure.
- A semantic analyzer tool that automatically classifies by introspection the existent public model of the publisher and subscriber models, using the definitions created by the DSL.
- A matching tool used to find the best mappings between the publisher and subscriber models, typically described with a hierarchy of classes. The tool can produce automatic mapping by selecting the best match using a scoring algorithm, but may also be run supervised to guide the choices.
- A JSON[11] serializer transforming the publisher model into a JSON formatted according the consumer model.
- A JSON deserializer which transforms the JSON wire format of the publisher straight into the subscriber model.

- The infrastructure to manage the HTTP content negotiation in the context of REST based services. This infrastructure also provides the optional production of the OpenAPI schema for the projected entities.

The first section of this thesis is an analysis of the current situation in distributed systems, with a particular focus to contract and versioning. In **chapter 1** we start from an historical perspective with SOAP and its four tenets which guided for a long time the architects in the SOA world. We then analyze the REST perspective whose contract is weak and accommodating, but where a different public model structure change can still totally compromise the communication among the parties. The analysis then moves into the current and most popular architectural styles "Command Query Responsibility Separation [12] (CQRS) and DDD which provide hints on the versioning friction point that we aim to resolve. At the end of the chapter, we also take a look at the Even Sourcing strategy and how versioning over time introduces an additional difficulty, as already mentioned before in this introduction.

The topics discussed in **chapter 2** consists in a closer look at the technical analysis of the type of changes that may occur in a public model. This includes the most appropriate layer, during the serialization and deserialization, where the generated code should perform the mapping between the two different models.

In the second section we introduce the elements of the Semantic Driven Modeling proposal. **Chapter 3** deals with the new abstraction layer analyzing the parameters that constitute the metadata. Specifically, we talk about the extraction of terms, their mapping to the concepts, an optional weight parameter for special cases. Finally, we discuss the concept specifiers which permits to disambiguate certain corner cases. This chapter goes in detail of the "domain" which is the SDM object linking all those parameters together. The domain is generated automatically by the Domain Specific Language.

Chapter 4 discusses the details of the DSL or Domain Specific Language. This chapter discusses the textual format of the SDM custom language, the code generator which compiles the text file into strongly-typed .NET classes and discusses the produced code. The classes generated by the DSL are also the starting point for developers to define policies, rules and validators which allow to enforce domain-wide constraints in unit tests.

In **chapter 5** we discuss two pillars of the Semantic Driven Modeling. The first is the abstraction of the type system which describes the data types, their structure in a graph and the relationships in the given model. This abstraction is platform neutral and allows the SDM to be implemented and used from other languages out of .NET. The second abstraction is the semantic layer which provides the metadata to be attached to every type and property described in the type system. These two layers allow to work with complex graphs of types, flattening it while preserving the knowledge of their navigation paths and attaching the semantical parameters described by the DSL. Finally, we describe the mapping process that consists in building a structure that associates every leaf of the source model graph to the one of the destinations.

In **chapter 6** we enter in the details of how can a model be serialized to a JSON document structured as a different model. Or, vice versa, how can a JSON document be deserialized to instances of an object graph that belongs to a different model. This process is able to transform models regardless of the change of property names, data types by mean of an automatic conversion process, and graph shape. The chapter concludes the discussion with the very important aspect of the performance. Being performant is vital for the introduction of any technology that hooks the communications of the distributed system. For this reason, we measured the performance using a well-known micro-benchmark library which provides very important results. By employing the dynamic generation technique, the gain factor is three orders of magnitude better than standard coding technique. The results will show that serialization is even faster than the standard serialization provided by the basic libraries. The deserialization is also very fast, but since it is a more complex process, it pays a very light overhead of a rough two times factor.

Chapter 7 has a very pragmatic approach analyzing the scenarios where SDM can be used in the context of distributed applications using REST as the communication infrastructure. We first make a short recap about the standard HTTP content negotiation and then we dig into the most interesting use-cases where SDM can be applied, showing the flow of the request-response dialogue.

In the final **chapter 8** we summarize the current status of this proposal and the accompanying implementation. The code of the implementation is not just a proof-of-concept aiming to demonstrate the robustness of the idea, but can be considered as a reference implementation for other programming languages as well as a solid base for future evolutions. This chapter analyzes how the choices made in SDM fits very well with the modern architectures and technologies that are currently available.

Part I

The first part of this thesis deals with model versioning issues in the context of the distributed systems, from the early days to the most recent architectural styles. This analysis is meant to identify how the public model versioning affects the communication among services and what are the requirements to identify the mappings between two public models when one of owning services makes some changes.

Chapter 1. Versioning and distributed systems

Versioning is probably one of the most challenging problems to solve in software engineering, but when it comes to distributed systems, the problem becomes even harder.

A monolithic application has an implicit dependency on the process it is running on, as all its components run in the same address space. As a result, every time the application is updated and restarted it won't affect any other external processes.

On the other hand, a distributed application is composed by different components, running in different processes and possibly even on different remote systems. Its distributed nature introduces additional difficulties whereas any of those processes will ever need to be updated. Will the new version of a component coexist with the existing ones? What happens if multiple components need to be updated at the same time?

These difficulties are not a good reason to lose interest in distributed systems. Decomposing an application has very valuable advantages that are worth to embrace.

One of the most appealing reasons is scalability: once a component is decoupled from the rest of the application, there is the opportunity to either parallelize multiple requests on different threads in the same process or run multiple components in different processes. Of course, providing scalability is far more complex as it depends from the code and the whole chain of dependencies in terms of libraries and dependent services.

Another attractive motivation is the ability to provide the redundancy and availability of critical tasks. Small and well-decoupled functionalities are far easier to redound avoiding therefore discontinuity every time the hosting operating system will need to reboot.

When it comes to maintenance, it is also clear that separating the various components over multiple teams is a big advantage. The points of contact among the teams coincide with the public model shared on the wire. But these contacts may also produce a friction because versioning the shared model may cause breaking changes to the other team.

Model and contracts in the distributed systems context

The first approaches in separating application functionalities into different processes were mostly RPC (Remote Procedure Call) oriented. Among them it's worth remembering the IDL specifications by the DCE (Distributed Computing Environment) which led to COM (Component Object Model) in the Windows and Corba (Common Object Request Broker Architecture) in the Unix/Linux operating systems.

The leading specification which instead formalized the idea of remoting the behavior of a component and introduced the Service Oriented Architecture (SOA) is the SOAP protocol (Simple Object Access Protocol).

The SOA specification was embraced by multiple vendors on all the major platforms. They mark a fundamental step in the architectural design thanks to the definition of four tenets introduced by Don Box, written while designing the Microsoft implementation:

1. Boundaries are explicit
2. Services are autonomous
3. Services share schema and contract, not class
4. Service compatibility is based on policy

The first tenet states that every interaction with the service must be done through a channel that has an inherent cost (paid in terms of performance), serviceability and that nothing is known a-priori about the service behavior. In the context of this thesis, it is important underlying that the message is the only implementation detail tying the publisher to its consumers.

The second tenet states that every service is independent in terms of development and versioning. This independence is also very relevant to our discussion because one of the goals in SOA is to avoid re-deploying all the pieces of the distributed system at the same time.

The third tenet clearly affirms the necessity not to share anything more than is strictly needed. Many programming languages expose the contract concept as interfaces. Others provide the same notion under the form of abstract class, but SOA recommends in sharing only what is strictly needed to deserialize the wire format. This tenet is very important because it means the publisher may freely evolve its implementation, as long as the contract is respected, without impacting the subscribers. The SOAP contract is very strict and quite verbose, but in terms of data is limited to the model, consisting in a graph of classes with its fields and datatypes.

Finally, the fourth tenet recognizes the difficulties in expressing all the possible requirements in the service manifesto (which is WSDL in SOAP) exposed to its consumers. Security constraints are a common example of such requirements. Those constraints may be expressed as policies and must be known by both parties to be enforced. In our context, this fourth point is very interesting because it highlights the fact that each system may have its own context that is shared among all the pieces of the distributed system. Adhering a common policy and sharing a context also allows to make assumptions that are valid for every service sharing the same policy. Later on, we will call this context with the word "domain".

The SOAP protocol has been widely accepted and embraced, but even limiting the publisher and subscriber point of contact to just the contract does not resolve the versioning problem. Before or later the contract will need to be updated and real cases testify that changes to the model are the primary reasons for a breaking change. Furthermore, the prolixity and the strictness of the contract definition in the SOAP manifesto made it very difficult to implement on smaller devices and caused a lot of compatibility troubles in services and clients implemented with different languages and frameworks.

The need for a contract and its version is the primary reason for the advent of Representational State Transfer (REST[8]). Roy Fielding's idea was to make APIs discoverable by fully leveraging the Hypermedia capabilities defined by HTTP.

When a user browses the internet, she does not need to know the address for each page to navigate to. She just relies on a single entry-point (the search engine) to make an initial request for the desired web site and then get a list of possible results as response. This process does not necessarily return the same set of results set every time. She read the response from the web engine and click on an URL representing the resource having the

best chances to be the desired one. The process then continues on the destination website by making new requests and getting new responses.

According to REST[8], when dealing with APIs instead of web pages, resources should be modeled to provide hypermedia style references that can be actioned through HTTP verbs. In other words, there should be no a-priori knowledge about the endpoints allowing to update or modify a resource. Instead, every time a resource is returned to a client, it also contains the URLs identifying the available actions. The REST[8] strategy frees the clients from the complex and strict WSDL definitions of SOAP. The downside of this strategy is that it leaves a wide degree of freedom which requires additional (external) knowledge about the publisher APIs and its models.

With regards to the models, REST[8] does not mandate anything, not even a recommendation beyond the generic concept of resources and their representations. The most common approach in public services is to provide an additional endpoint publishing the JSON metadata describing the available service endpoints and models. The OpenAPI specification has become the standard de-facto for this metadata and is loved by most of the developers because it can easily be used to generate the Data Transfer Objects (DTOs) and even the client code to deserialize the data coming from the publisher.

Thanks to REST[8], the friction between the publisher and subscriber is even thinner and now even a cheap and small IoT device powered with an ESP32 processor may publish or consume data through HTTP. Anyway, the versioning problem is still not resolved as a single change in a field name or model graph will cause the deserialization process to break on the consumer side.

Another emerging communication technology is gRPC[13] which prefers the performance to the service coupling. It uses the Protocol Buffers, a binary serialization method which requires sharing the precise version of a model in order to work correctly. While JSON is accommodating towards certain model changes, Protocol Buffers are very strict. This serialization method works well in scenario where there is a tight coupling among the producer and its consumers but it is certainly harder to manage when the model changes. Anyway, it finds its best in Single Web Applications (SPA) because every time the server-side is re-deployed, it also contains all the web pages and scripts that are used from the browser to access the service.

Beyond the service contracts and the models, we should also consider the different architectural approaches to understand how and if they have a role in the versioning problem.

Versioning in the modern architectural styles

Software architecture is about creating harmony and orchestrating the entire structure that compose a system. Historically, the first step into distributed enterprise systems was based on the three-tier based pattern. The following milestone was the Service Oriented Architecture (SOA) era who saw an increasing decomposition in SOAP based services.

All the most recent architectural styles have in common the increasing number of services composing the system. This should not surprise because in both cloud and on-premises installations, there are new opportunities for the software architects. One of the most intriguing is a new sandbox called "container". It consists in a sort of small and lightweight virtual machine that is offered from the modern operating systems to guarantee the environment requirements and isolation in a deployed application.

For a developer, containers are a unique opportunity to ensure that every aspect of the deployment is controlled, and making the "works on my machine" dream a reality. It is been used as a unity of deployment to quickly update the services in a system and accelerate the development iteration. This is the principle at the basis of the microservices architecture.

The extreme version of a microservice is found in the cloud environment where the reduced size of a single service may culminate in a "serverless" design strategy. Serverless is used to describe a specific hosting environment currently available on all major cloud providers. Its main benefit is its reduced price as it is meant for stateless and disposable pieces of code, which may be as small as a single function.

The direct consequence of the services fragmentation is the increasing number of different models that gets mixed in the same system. This happens differently, depending on the architectural style and patterns adopted in the architecture.

Command Query Responsibility Segregation Pattern

A clear example is CQRS (Command Query Responsibility Segregation), one of the patterns that gained many favors in the recent years. Its main principle is enforcing the separation between the model used to write into the system from the reading one. Interestingly, the pattern suggests, even if not mandatory, the use of multiple reading models which provides different advantages. The first one is avoiding a single large and complex monolithic model which can be difficult to manage and keep updated. The second is versioning. Not necessarily updating one of the reading models should impact the others. This allows a smoother transition over newer versions of the model because it is possible to maintain active both the old and the new one. If we consider these reading models as views over the data, in the context of this thesis, each of them may also be seen as a **set of concepts**. Later we will see how the Semantic Driven Modeling may help in validating the model's consistency.

Domain Driven Design (DDD)

The model becomes even more centric in the architectures using the Domain Driven Design (DDD) approach. DDD is not just a pattern but a whole set of practices and, recommendations that are used to smooth the friction between the developer and the end-user.

The typical habit of a developer is trying to reduce the business representation to something familiar, but unfortunately often incomprehensible to the end-user. The main recommendation in DDD is to talk with the customer, fully understand the business-specific glossary and adopt it. This is called by DDD the "ubiquitous language", a sort of *lingua franca* whose **terms** are familiar to the business users and used to communicate all the business intents representing the **concepts**.

It is interesting observing how DDD recommend to strictly observe this principle. For example, as soon as different areas of the same business adopt different terms, DDD suggests that their services should also implement a different model.

For example, in the IoT sector, the hardware team designing the physical devices uses a electronic-oriented terminology. They typically model each sensor in terms of physical measures, boundaries, tolerance and sample frequency. As soon as the sensor data is pushed on the application bus, this data get consumed by several other services which provide different businesses. For example, some environmental data being read from the

bus may be used to control a painting process using different parameters. The two teams owning the hardware devices and the painting process are different, talks different languages and therefore use a different language and terminologies.

Interestingly, the adoption of a calibrated terminology in the model is the starting point in Semantic Driven Modeling which also provides enforcing rules, policies and validations at semantic level.

In the DDD world, the interface between two different models is managed by the so-called Anti Corruption Layer (ACL) which acts as a translator between the two businesses. This is an important aspect because, once again, versioning is a crucial point as any change to the model used by the devices team will also affect the ACL. Moreover, ACL is a layer that depends on both models.

Outside the DDD world, we find a concept similar to the Anti Corruption Layer that is just called Proxy. Basically, each consumer creates a proxy of the publisher service who cares about making requests and translating the responses back to its own world.

Those translation layers may be reused as well, because there may be multiple services sharing the same model. This concept is formalized in DDD as the Bounded Context that is used to identify the set of services sharing a common model. The shared model may contain concepts that are not used from some of the services that are part of a Bounded Context. This is a useful concept because it allows to avoid rigidity or an excessive fragmentation of the services that otherwise would need many ACLs.

Later on, we will see that the extents of the **domain** in Semantic Drivel Modeling are similar to the DDD Bounded Context.

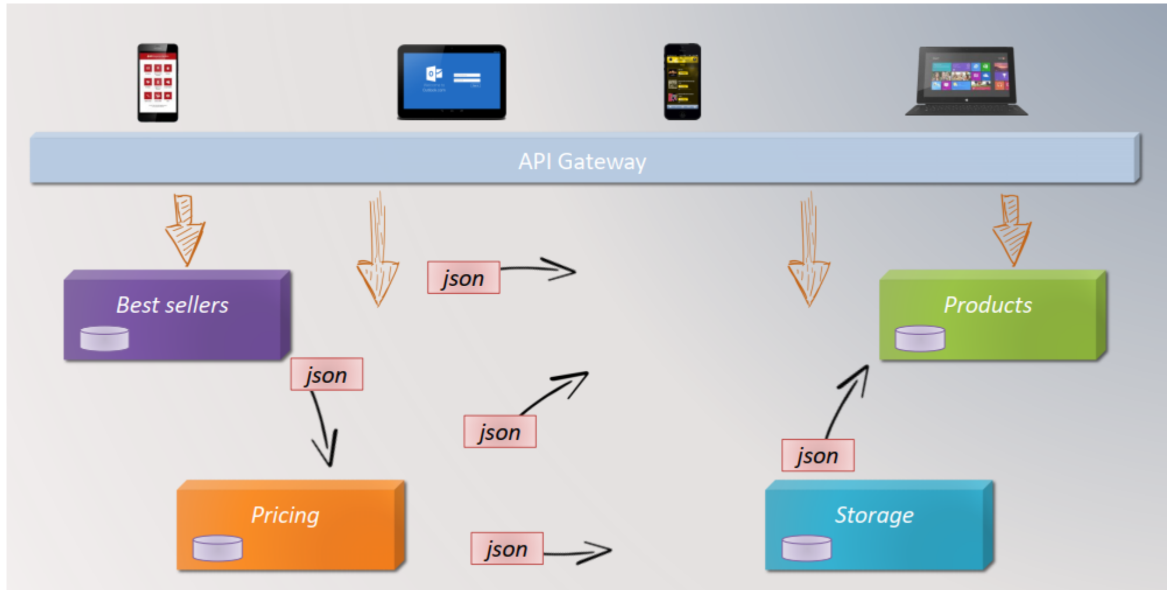
It is worth noting how, at their heart, the SOA tenets are still noticeable in these major patterns and architectural styles. In particular, the "Bounds are explicit" tenet is relevant because the boundary is the place where we will take some action to resolve the versioning issues.

Service communication patterns

We already talked about REST[8] in the context of contracts and models. Its resource-oriented approach follows the classic CRUD (Create, Read, Update Delete) model. Another model, which is becoming more and more popular in the recent years, is pushing data into a queuing channel such as a bus. Any consumer in the system can then decide to

subscribe a specific type of message, grab it under the form of event, and process it.

In the classic, direct access using REST calls, an e-commerce system could look like the following schema:



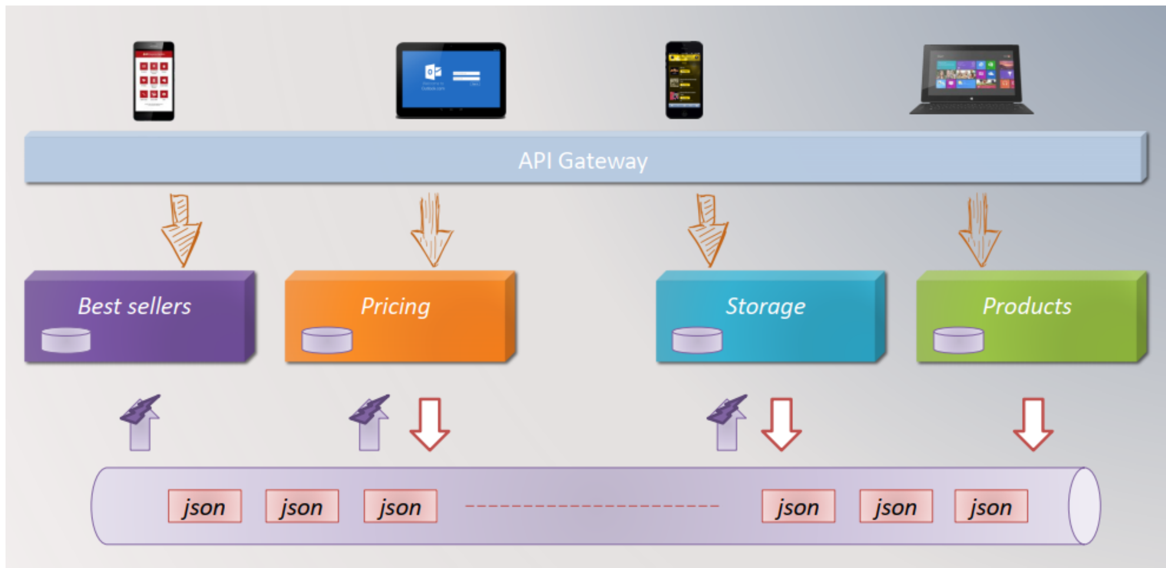
The API Gateway is an important layer which resembles as a proxy. It allows to hide the knowledge of the individual services knowledge to any external consumer. This solution is very popular and allows to change and evolve the back-end architecture without impacting on the external clients which, of course, are difficult to update in a reasonable amount of time. The importance of the API Gateway is so high that the major cloud providers advertise this service as part of their infrastructure. In other words, there is not even the need of manually building it, but just configure the one provided in the hosting environment. The API Gateway configuration may just contain the routings towards the internal services but it can be much more sophisticated.

Once a service is invoked, it may need to use other services to fulfill the request. In other words, the internal services need some strategy to avoid the coupling to the other services. You may think to a component that is functionally equivalent to the API Gateway, but for the internal services. Unfortunately, even if using a proxy for this purpose may work, the number of the required chained calls could be much higher, causing a significant performance hit.

Another issue is that the number of calls depicted with arrows raise geometrically as more services are added.

Despite the direct REST[8] cross-calls mess, we immediately see that, every time the publisher and subscriber do not share the same model, the arrow also represent the need of a **model translation**.

The alternative solution is using a queuing system which has become very popular in many recent architectural styles.



By introducing a bus, every service can publish its own changes in the queue and subscribe all the data it is interested into. In certain cases, it may also process data that may *potentially* be useful and store in a local database in order to avoid calls to other services that may need for future requests.

In terms of scalability, the adoption of a queue allows to fully exploit the computing power avoiding any locks needed to arbitrate the access to a resource, such as a database. If we consider a CQRS based architecture, the requests to write some data are always asynchronous and happens as a consequence of a request called Command. Each concurrent command can be processed at the same time without needing any lock since it is a write channel and no response, out of an acknowledge, is necessary.

Generally, queues are provided by message queuing systems that can be used using standard popular protocols like AMQP (Advanced Message Queuing Protocol) and MQTT (MQ Telemetry Transport) which is mostly used in the IoT (Internet of Things) context. They typically provide, as optional, many characteristics such as persistent queues, delivery guarantee, broadcasting and clustering. Those guarantees are very important because it is not necessary to wait for the message to be completely processed

or stored. Instead, the sole delivery guarantee is sufficient to consider the data as committed.

There are a number of interesting advantages in adopting this solution. For example, message-based systems are perfect to geographically distribute the data. In fact, the latencies can be very different depending on the region, therefore accumulating data in a local database result in a more responsive system.

While the arrow mess is gone, from a versioning perspective this last picture is cheating. In fact, every JSON message in the bus still needs to be translated into the local model from all its subscribers. Also, understanding the real dependency graph of the services becomes more difficult.

At this point we know that, regardless the communication pattern, the mismatch always happens when the message has to be transformed into the model or vice-versa, in other words during the **serialization/deserialization process** that we will tackle in the next section.

Anyway, another important aspect of the problem is the message lifetime.

Versioning over time and the data storage

We have seen how the introduction of a queuing system provides a cleaner solution but with a price. Things get even more complicated with another pattern called **Event Sourcing** which consists in storing the sequence of events causing a change of state in the system.

In a classical relational database, every change overwrites the current state losing therefore the previous version of the data. In order to ensure the consistence of the data, they provide the ACID (atomicity, consistency, isolation, durability) properties which ensure you will always observe the most recent version of the data. It is anyway amusing observing that the transaction logs in a relational database works pretty much like event sourcing. In fact, the transaction log stores the sequentially ordered commands that are needed to change the data stored in the database tables. Anyway, after the commands in the transaction logs are persisted, all the previous versions of the data are gone.

With Event Sourcing, all the sequences of events changing the system are persisted in an "append-only" modality so that they represent the actions over the data and not just the data itself. Those events are not just changes but even deletions, which are recorded by adding a new data structure representing a certain piece of data being deleted. This allows to rewind or fast forward the entire history of the system.

As a result, we now have to deal with **versioning over time**: every time the model evolves, we start writing a different data structure into the event store. As soon as all the services and components in the system as updated, they understand the new format, but not able to read the previous versions anymore.

There may be the temptation to convert the stored events in the new format but this is usually not a good solution. The first reason is because certain regulations pretend the historical data to be stored on a WORM (Write Once Read Many) support. The second is because the conversion is delicate and may corrupt the data. The third and most obvious is the time required to update the store which may translate in keeping the system offline.

Since we are storing events, but users usually need the most current state to work on, the separation between the writing and reading channel (such as in CQRS) is a good idea. Practically, after storing the events sequence on the Event Store, the user gets an acknowledge that its operation is guaranteed. Then, data is replicated *asynchronously* on a relational database that will be queried by a reading channel. In large architectures this is a big advantage because there may be multiple, geographically distributed, readings channels, allowing better querying performance.

There is a big deal in the word "asynchronous" because, now that we have multiple databases to keep in sync, if we want to maintain the classic ACID properties in the system, it is necessary to make distributed transactions.

Pretending ACID properties in a distributed system is a burdensome lesson learned in the nineties when architects were still struggling in finding a solution. In his famous paper "Life beyond Distributed Transactions: an Apostate's Opinion"[14], Pat Helland explains how impractical distributed transactions are (in his own words, he «has been implementing transaction systems, databases, application platforms, distributed systems, fault-tolerant systems, and messaging systems" in Microsoft, Amazon and Salesforce).

The alternative solution is to let the system give up on the ACID properties and enter the world of **Eventual Consistency** which translates in the inability for the system to immediately reflect the updates to the users that are reading data because of the replication delay. It's worth noting that, in my personal experience, on systems that are not geographically distributed, the delay is usually not critical but greatly improve the overall performance and scalability.

> The Event Store recording these events can be either a classical relational database or a NoSQL database. But usually, the best choice for an Event Store is a document-oriented or NoSQL database. The main reason is because, while the service model version may change over time, there is just one schema in a relational database at a given time. The second is the number of insert statements needed in the system to store all the sequence of events: adding large amounts of documents is cheaper on NoSQL databases than adding large number of records into a relational database.

This complex scenario is very challenging in terms of versioning because employs a write model and multiple reading models where all of them may change over time. Anyway, the concepts and semantics expressed by those models is inherently invariant. Even in the case the system evolves adding more concepts to its models, the newest code is able to make decisions over their conversions as we will see in the next section.

Chapter 2. Model analysis

In the previous chapter, we observed the versioning issues from the service contract to the storage passing through modern architectural styles and communication channels. The common point is always the relationship between one subscriber and its consumers which always transition through a wire format.

In this thesis we are not interested in many aspects of design modelling. Instead, we focus on the model used on a service boundary. In many cases this model is the same used from the service in its domain model, but this is not necessarily true. We also don't care about the model behavior, if any, as it would not be transferred over the wire.

It's also interesting observing that certain properties found in the model strictly depends on the architectural style or pattern used in the system architecture. For example, in an Event Sourcing based system, the model is very likely to contain some versioning information. More generally, it is not uncommon for models to contain more than the strict characteristics of the object they represent. The most popular example is the unique identifier (typically the Id property) which is used to properly query, store and work with data.

We can now drill into the model to see what can possibly change over time and understand what is needed to let the system adapt to the changes.

One business, different models

Depending on the programming language employed in the service, the model may be implemented differently. Typically, it is described as a hierarchy of classes, each one having public fields or properties (which are an abstraction over the fields).

> The proof of concept provided in this thesis makes use of Microsoft .NET 6 and the C# language, whose syntax embraces the usage of properties to encapsulate the class fields. At compile-time the compiler will generate a private field and two accessor methods: one for the getting the field and another one for the setting it.

The C# language code for a hypothetical model exposed by a service is the following:

```
public class Order
```

```
{
    public Guid Id { get; set; }
    public string Reference { get; set; }
    public List<OrderItem> OrderItems { get; set; }
}

public class OrderItem
{
    public Guid Id { get; set; }
    public Article Article { get; set; }
    public Company Customer { get; set; }
    public decimal Quantity { get; set; }
    public decimal Price { get; set; }
    public decimal Discount { get; set; }
}

public class Article
{
    public Guid Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public DateTime ExpirationDate { get; set; }
}

public class Company
{
    public Guid Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public Address Address { get; set; }
}

public class Address
{
    public Guid Id { get; set; }
    public string Street { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Country { get; set; }
}
```

The starting point is the `Order` class which is shaped with some properties and an `Id` property of type Guid, acting as a unique identifier for any instance of its type.

There is no semantic to describe the runtime value uniqueness of a property but this is clearly the intent of the Id property. The class also contains a `string` named `Reference` and the property `OrderItems` which is a list of objects of type `OrderItem`. Once again there is no semantic to clearly identify the fact that `OrderItems` is a collection but this can be understood looking at the interfaces implemented by the generic type `List<T>`. For the sake of this discussion, collections are important because provide a `1-*` (one-to-many) relationship among object in the graph.

The connections among those classes can be clearly identified. For example, there is a `1-1` relationship in the property `Customer` class `OrderItem` whose type is `Company`.

Another service working in the same business, may have used a different and simpler model:

```
public class OnlineOrder
{
    public Guid Id { get; set; }
    public string Description   { get; set; }
    public List<OrderLine> OrderLines { get; set; }
}

public class OrderLine
{
    public Guid Id { get; set; }
    public string ProductName { get; set; }
    public string ProductCode { get; set; }
    public DateTimeOffset Expiry { get; set; }


    public Guid CustomerId { get; set; }
    public string CustomerName { get; set; }
    public string Street { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Country { get; set; }

    public double Net { get; set; }
    public double Payment { get; set; }
```

```
}
```

The hierarchical structure is definitely simpler. It can fit very well on a service whose task is to import/export Microsoft Excel spreadsheets or create reports. It's worth noting that some of the data types are also different:

- The `decimal` data type is a high-fidelity representation of a decimal numeric, while `double` is an `IEEE-754` conformant floating-point type which is subject to power of two approximations.
- The `DateTime` data type represent a date and time, while the `DateTimeOffset` also have the notion of time zones.

Before understanding how to deal with the transformation between those two models, we need to identify which layer is the most appropriate to transform the model.

Where the transformation should happen

Given the various architectural topology and communication channels, and considering that reads and writes make the data flow in different directions, we need to answer these questions:

1. Who is the owner the model and which party should take care of the transformation?
2. Is the data being read or written (what is the direction of the data)?

The **owner of the model** should never transform the data for others. There is nothing preventing doing differently, but it does not make much sense for a service to behave as a polyglot towards all its possible clients. Furthermore, when a service publishes some data inside a queue, it may adhere to just a single wire format. For this reason, it makes sense for **consumers to transform the data** for themselves, even if the transformation may happen multiple times in different services and processes.

Every time a **client reads** some data with a direct REST call or receives an event from a queue, the wire format of the data will conform to the model owner and therefore it will be a client responsibility to transform the data during the **deserialization process**.

When instead the **client posts** some data, either via REST or inside a queue, it will still have the transformation responsibility but it will happen during the **serialization process**.

We now need to know what the differences between the two models could be.

Dealing with the model differences

The transformation from a model to another one may fall in one or more of the following categories:

1. The hierarchical structure of the two model is different
2. The data types for the same property is different. This may just require a conversion or some more complex algorithm
3. The name of the property representing the same concept is different. This happens when the model owner just used a different term for the same data.

The most popular solution to this problem consists in creating the required Data Transformation Objects (DTOs) on the consumer side for all the types of the publisher model. Then, right after the deserialization process into the DTOs, the data gets copied into the model owned by the consumer. This transformation can be trivial or very complex, depending on the differences outlined before. For the most trivial cases, there are third-parties generators allowing to generate the needed code.

The first issue with this solution is that it requires the consumer side to be updated as well. This is non-trivial when dealing with a microservices architecture because there may be multiple consumers to update. If instead the two models are very different (typically because of different data types and graph shape) the transformation must be coded by hand.

Another issue is about performance. In fact, the data on the wire format, JSON for example, is copied first into the DTOs and then into the final model. This double copy requires more CPU power and memory, stressing the garbage collector as well.

It is worth remembering that we are just dealing with data coming from outside the boundary. This means that any transformation, being manually or automatically generated, will never pose the use of complex structures like class hierarchies, dynamic bindings or native pointers that would deeply affect the complexity of the transformation. Specifically, whereas the serialization process is done with a standard JSON serializer, the structure has well-known limitations that simplify the transforming code.

The behavior of standard serializers

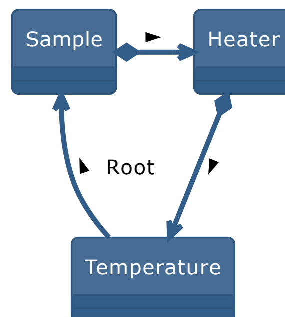
Since the serialization process is performed by the service layer with the intent to talk to an external service, the wire format should be agnostic towards the framework and language used in the two contexts.

We have already talked about the communication channel which, in the microservices context, is typically REST or gRPC. In this thesis we will only cover REST using the JSON wire format even if the client may request other formats like XML. The gRPC serialization is conceptually equivalent from our perspective, but its implementation requires an implementation from scratch of the serialization process.

The advantage in using REST and JSON is not limited to be the most popular but also benefits from the JSON default serializers characteristics. In fact, the serialization process creates a directed acyclic graph where each node has only a single parent. Avoiding cycles greatly simplify the mapping process, because it is easier identifying the object instance where the deserialized value should be written into.

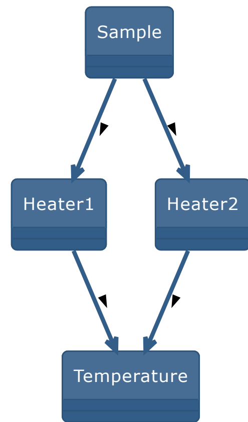
As a side note, should the original model present a circular reference (such as object A connected to B and B connected back to A), the JSON serializer, using the standard settings, will throw an exception. Anyway, it is quite unusual for public models being structured with reference to the parent items.

Here is an example of a class structure exposing the circular reference problem.

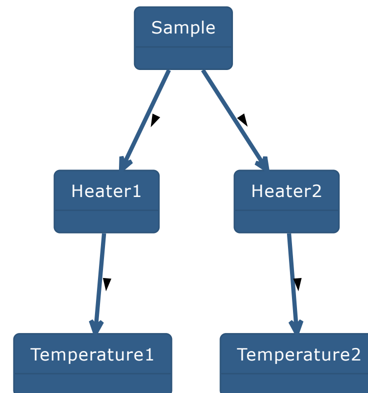


Another advantage in relying on the deserialization process is that JSON objects always have a single parent. As outlined before, JSON serializers do not provide by default the concept of "reference" therefore only a single parent is allowed. If model graph is characterized by two objects A and B pointing to the same child C, the serializer will duplicate the child C, resulting in two different graphs A-C and B-C.

Class structure



Object structure after serialization



Please note that the behavior described for standard JSON serializers may be changed. For example, some serializer allows to serialize references by adding an extra field representing the instance ID. Anyway, it is very unusual to see such custom options in REST services as they publish information on the boundary and therefore the serialization process should be standard.

Given our goal to remap a model into another, we have finally identified the need of intervene during the serialization and deserialization process depending on who is the owner of the model and the operation that is being done. We have understood that the differences between the models fall into three different categories: different graph structure, different data types, different property names.

There is still another important aspect to consider: why is the model changing? In IoT a model may change because the precision is higher and requires a floating point, or maybe there are multiple temperatures available and the property names must be changed and moved into a child object.

The nature of the model changes

Every change, addition or removal in the model "schema" (class or property names) may have different motivations going from a simple cosmetic refactoring to optimizing the schema for a better clarity.

Renaming is very common while refactoring but this can't be done easily. The most common habit is to add a new property having the same data type and value without removing the old one. The result is a bit messy because the wire format contains a duplicate value and it may also cause confusion on the consumer side. In any case it is necessary to take actions on both the publisher and the subscriber sides.

Another simple reason is removing some data. The versioning rule of thumb is to never remove fields, but just obsoleting and let them die over time so that, in future releases, it will be safe removing the field. The JSON serialization nature simplify this task because it is greedy, meaning that any data that is not physically available in the wire format will not cause any deserialization error on the consumer side. The consumer still the need to take some precaution and validate every field because, for example, a missing integer for the "Price" property will result in assigning a zero value, which may cause a bug.

Adding a new field or property another and more interesting change to the model. In most of the cases it happens when a new feature has been implemented on the publisher side. But again, thanks to the JSON serializers standard behavior, the consumers can safely ignore the new field also because they still don't know how to use it. As there is no code to consume, it does not make any sense struggling to map the new field.

A more subtle change comes from changing the data type. Since the JSON type system is very limited, whenever the data serialization remains the same, such as changing from 16-bit to 32-bit integer, the consumer does not need any change on its side. If instead the change also affects the wire format, the consumer is potentially broken and need to be **updated at the same** time of the publisher.

A more disruptive change comes from changing the structure of the graph. For example, moving the address related properties in a child class is a breaking-change and will also affect the subscriber.

Most of the designers try to avoid the breaking-changes and publish the service on a different endpoint using the new model so that the transition can be done smoothly over time, but this is often not straightforward and may require some extra configuration in the infrastructure.

Beyond the technical changes, the initial question is not completely answered. We can rephrase it this way: is the nature of the model changed?

Of course, if the changes are exclusively driven by a cosmetic or efficiency refactoring, the nature of the model remain unchanged. But, as soon as the model owner adds a new feature, the model may change or just augmented in order to provide a **different meaning**.

Later we will dig into the formalization of the concepts being expressed by model. From a versioning standpoint, every time a new concept is added to the model, this requires the consumer to be able to manage it. For example, if an order management service starts receiving new fields about the delivery of the packages from the shipping service, it will not be able in processing it until the business logic gets updated.

While this is absolutely obvious, it is important underlying that our goal is to **let the model evolve without causing breaking changes**. The order management service can continue working by ignoring the new fields because their meaning goes beyond its initial requirements. On the other side the shipment service can definitely starts populating those fields because, at a later time, the order management service will be updated to process the deliveries.

Part II

In this second part we introduce the elements of the Semantic Driven Modeling proposal. They consist in tools and libraries that let the developers free to evolve a model while not breaking the publisher/subscriber compatibility and without requiring to restart the consumer process.

We will first look at the building blocks, followed by all the tools and libraries that are required to make it work.

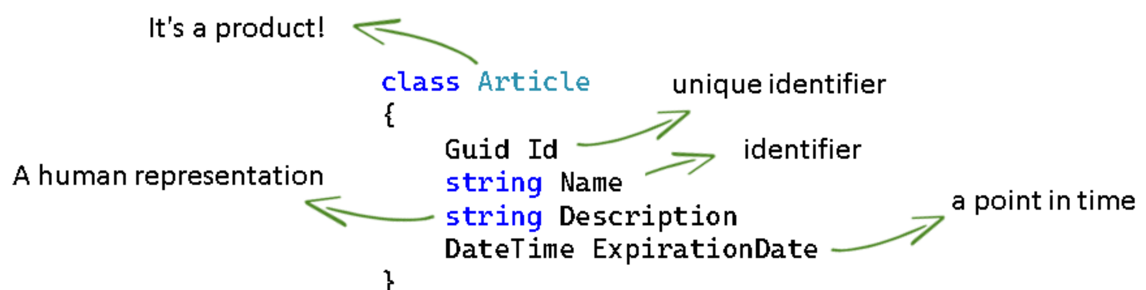
Chapter 3. Abstracting the model

Considered the different kind of changes that may impact a model, it may absolutely result non-trivial for a developer manually identifying the mappings from the new, updated model to other models.

In a previous example, we supposed to rename and move some temperature fields from the root object to a child object in the model graph. Clearly, this doesn't change the nature of the data being represented, but it may result quite difficult for a consumer to figure out what property has been renamed and understanding the changes in the graph. In addition to that, since we want to leave a very high degree of freedom on the typology of changes, any mistake while identifying the mappings may result disruptive.

As often happens in software, the solution may come by adding a level of abstraction.

Let's consider some pseudo-code defining a class named "Article" with few public properties or fields. For a human, it would be straightforward understanding the **concepts** they represent. She would use this knowledge to map the "Article" class into a "Product" class belonging to another model and having *similar* properties.



Identifying a concept from a bunch of words is an ontology problem. Even if there is no grammar or any schema, and this does not fall in the natural language analysis, we still can extract important information from the words composing the class and properties. Those **terms** can be easily identified because developers and software architects normally employ rigid naming conventions which have the intent to make the code understandable from a human being.

Our goal is instead to classify these names in order to make them comprehensible from a machine.

> In order to extract the names of the classes, properties, data types and their relationships, .NET provides the **reflection** technique (introspection) which retrieves the metadata that is available with the code in the binary file. Metadata is normally available and used by the .NET to provide important runtime services such as *generics*. Later we will see that this information is collected at an initial stage and that reflection will not be needed at runtime anymore.

The first step is to identify all the terms used in an existent model. We can just consider the class name and its properties or fields. As methods just define an object's behavior and can't be serialized, they can be just ignored.

Terms

Extracting the words from a class definition can be as simple as taking the name of the class or property ("Article" and "Id") and split composed words based on the casing. For example, "ExpirationDate" becomes "Expiration" and "Date".

Extracting the words involves the following requirements:

- We assume to use the English language. Even if not mandatory, this is widely adopted as a standard for code and comments in most of the major software houses. When using other languages certain rules for extracting the words may change from the ones used for the English language.
- The same property may use multiple words, not separated by any space. They are all useful to provide more specific information.
- It's quite common using the plural form for lists, collections, arrays or dictionaries.
- Some words may be repeated. Even if it is not a best practice, it is not uncommon in an Article class defining a property named "ArticleName". The repetition does not provide

any additional information and is removed by the current implementation.

- Some other words like "ZipCode" are often used as they were a single term instead of two separated words "Zip" and "Code". This is allowed and evading ambiguities.

The algorithm extracting all those words walks all the classes in a given graph, avoiding circular references and considering both the camel case (twoWords) and pascal case (TwoWords) conventions. It also employs some filters to revert plural words into the singular form.

Since the goal is to understand the meaning of the term, these words must be matched against a **predefined glossary of terms**.

Domain glossary

The whole set of terms typically used in a specific domain, makes up the glossary.

It is needed to identify the terms extracted from the model graph and to verify there are no unknown terms. These pre-defined terms may also be composed words such as "ZipCode" so that it does not get split during the initial extraction phase.

Writing the initial list of terms that may be used in a model can be difficult or boring, therefore the proof of concept is accompanied by a tool that blindly extract all the terms providing a draft glossary that is then used by the designer to finalize it. Of course, the words like "ZipCode" must be reassembled manually.

A very interesting similarity can be noticed between the Semantic Driven Modeling Glossary and the Domain Driven Modeling Ubiquitous Language. Even more interesting, we will see in a moment how SDM can be used to validate the model in terms of semantic, and ensure that a DDD model is conformant to the business it is built for.

The abstraction process can be finally done by associating the terms to the concept they refer to.

Concepts and Weights

Concepts are the most important elements because they represent the abstract idea behind any of the terms used in the model.

There is a one-to-many relationship between the term and the concept, as one concept may be identified by different terms. In addition to that, as a concept name may also be used as a term, every time we define a concept, we also declare a term having the same name.

Depending on the industry we may find different terms for the same concept. For example, the terms "Material", "Article" and "Item" may all be referred to be a "Product". Anyway, in some industries there are other possible terms for that, like "Ingredient" and "Excipient". This anticipates the idea that is safer to consider all the defined terms and concepts to be business specific.

Some of those terms are effectively synonyms for "Product" while others like "Item" are debatable. For this reason, every association of a term to a product may define a **percental weight**. This additional parameter provides the opportunity to lower the probability that "Item" is identified as a "Product".

Some concepts express very generic, high-level ideas, therefore there is not enough information to clearly understand them.

Concept specifiers

If we think to the terms "Range", Min" and "Max", they may be associated with the "Bounds" concept but their distinction is fundamental. For example, "Range" may be a string specifying two values separated by the "÷" character. In a different model the two values may be expressed in separated fields called "Min" and "Max". Preserving the "Minimum" and "Maximum" information is crucial.

To handle those cases, we may want to add the "**Concept Specifier**" optional parameter which can be seen as a second level concept. In the previous example the term "Min" is associated to the concept "Bounds" and "Minimum" as concept specifier.

The domain

By just using terms, weights, concepts and specifiers, there is the risk that some term may still be too vague to correctly identify a concept. As we already said, this may happen because context matters.

To minimize the resulting entropy, we have two possibilities:

- Add some metadata that can further specify the correct meaning of a given term.

or

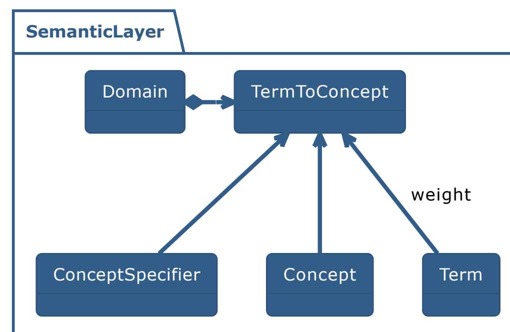
- Restrict the field of application and define the context where the association of a term with a concept makes sense.

The first solution is definitely feasible by using AOP (Aspect Oriented Programming). In .NET additional metadata can be specified by using custom attributes on the class and properties definitions. The con of this solution is that many developers do not like to pollute model definitions with attributes. The aesthetic problem can be solved by providing a fluent API like the one used, for example, in the .NET Entity Framework to model the database mappings. In any case, both solutions require additional manual work for the developer. While this remains a possible opportunity in future releases, the current implementation does not use those techniques to stay as simple as possible.

The second solution fits very well for a distributed system architecture where the same company owns all the services belonging to the same system where each service is managed by different teams. We can certainly define a limited number of "domains" in a microservice-oriented architecture.

When instead the services belong to different applications, it can still work but both the publisher and subscriber parties should embrace SDM to make the process effective.

A domain can be defined by putting all the previously defined parameters together. The domain links **terms** to **concepts** and their **concept specifiers** using the provided **weight**.



The domain may look like a limitation but it is instead a great opportunity because it allows to make assumptions that otherwise would be hard to formalize in code. These assumptions are a great opportunity to be formalized as **policies** so that they can be used to define domain-wide rules that may help in mapping the models.

Domain Policies

Domain policies are the whole set of standard behaviors that guides the mapping process from one model to another. They are not used to enforce some specific behavior. Instead, they provide the guidelines to set a preference over ambiguity or the parameters and formulas that should be used to convert certain parameters.

For example, we can assume that a certain domain will only use the SI or Imperial units of measure. This is a drastic simplification because it eliminates the burden to associate a given term or concept to specific unit of measures.

Another example would be defining the conversion formula from Celsius to Fahrenheit using the recommended data type among *float*, *double* or *decimal*.

The domain policies may also include a correspondence between a concept and a list of allowed data types. For example, we may want to associate the "UniqueIdentifier" concept to the Guid data type. It would definitely make sense not to mix data types for this concept because it may be impossible to convert. Another example may be the association of the "Money" concept to the "decimal" data type because the floating-point data types follow the IEEE-754 standard which is not acceptable in the financial domain.

While policies can be synthesized as the general guidelines to be used in the domain context, there may also be more strict enforcement which are called **Domain Rules**.

Domain Rules

Domain Rules have the goal to take a specific decision when looking for the best mapping from the source model/property to another one in the destination model.

There are two types of rules.

The first type is a list of "**discard**" rules. Each of them receives a mapping being during the initial evaluation and can decide to discard it immediately. For example, we can use the allowed data types defined in the policies in order to enforce their use (by discarding the other ones).

The second type is a list "**equally scored**" rules. These rules come to play when multiple mapping receive the same score out of the default weighted algorithm. Before making an arbitrary decision (for example, the first in the list), the "equally scored" list of rules are invoked. If one of those rules accept the comparison and answer with a mapping, this wins

over the others.

But we can go further to just policies and rules. There is in fact the opportunity to use the semantic metadata to validate the model.

Semantic validations

We all know what validation means in the context of a traditional model. For example, we can use a regular expression to validate an email address, the zip code or a pre-defined list to validate the country. Such a validation is run at runtime, right before the serialization or immediately after the deserialization, to ensure that the data is conform to certain minimal requirements.

Instead, when talking about semantics, all the reasonings happen at design time, possibly every time we are making any design addition or change to the model.

Is the model conforming to the glossary of terms? Is the model expressing any unknown concept? Does the model conform to the mapping between the data types and the concepts (defined in the policies)?

These are all questions that avoid semantic errors and, if the architectural style is based on DDD, also conformant to the "ubiquitous language".

Creating the domain

It would be difficult working with policies, rules and validators against using terms, concepts and concept specifiers strings without providing their strong-type representation. For example, instead of defining the "Product" term just as a string, it is more convenient creating a Term class encapsulating it:

```
public class KnownTerms
{
    public static Term Product = new Term("Product");
    // ...
}
```

This approach enables any rule and validator to deal with the product term with the following simple code:

```
if(term == KnownTerms.Product) { ... }
```

In addition to that, we certainly don't want to manually write the code linking the concepts to the terms using weights and concept specifiers.

We need a better way to define those parameters and create the domain. Let's first see what are the operative steps to create the domain:

1. Collect all the terms that are used, or that may potentially be needed in our domain. We can start by running the term collector tool on an existing model in order to get an initial glossary of terms.
2. Group all the terms having the same meaning in our domain. The meaning expresses a concept; therefore, we pick one word that will be used to represent all of those terms. The concept is itself a term, therefore there is no need to duplicate the string.
3. Weight every term to indicate whether they uniquely identify that concept or if they may also point to a different one. In the first case, we can skip this step and its default will be 100. In the latter case the number will specify the probability that a given term represent that concept
4. If a term specifies the given concept only in a specific context, annotate it, otherwise it can be skipped.

This procedure could be boring but it can be dramatically simplified by specifying everything in a single text file and letting a tool to create all the rest for us.

The text file becomes our new Domain Specific Language which is compiled with a tool following a simple but effective grammar. The outcome from the DSL compiler is C# source code that we can use to create the policies, rules, validators in our newly defined domain.

The semantic layer

The semantic layer consists in the abstraction which describe a specific business' domain which is made of all the parameters described previously in conjunction with policies, rules and validations.

Every specificity belonging to one or more components of the distributed application should be described in this layer which provides an abstraction.

In the proof of concept, we are still experimenting how to make the most of this abstraction layer and it appears clear that only applying the SDM in real use-cases will provide a final answer.

Chapter 4. The Domain Specific Language

The Domain Specific Language (DSL) for Semantic Driven Modeling is an optional but convenient way to remove any friction when defining the domain policies, rules and validations. A single text file using a simple grammar is used to declare and put together all the terms, concepts, weights and concept specifiers.

The language statements are used to automatically generate .NET code so that they become strong-typed declarations that can be used to create the custom logic governing the domain.

The basic rules of the DSL grammar are the following:

- Variables to be declared in the domain starts with the `"*"` character
  - The declaration `*Name=IoT` set the pre-defined `Name` variable to the string `IoT`.
- The `"#"` token is used to provide comments that are stripped away and hence ignored by the DSL.
- A line starting with a word defines a concept.
- It may exist a line preceding the concept and starting with "//". The text specified after the token will be used as the comment for the formal definition of the concept in the generated code.
- When the lines following a concept declaration are non-empty and start with, at least, one space character, they specify:
  - An optional concept name used as a context for the given term. This limits the term usage to the specified context.
  - A mandatory term name. We can think at it as a synonym for the given concept.
  - An optional concept specifier (between square brackets) providing an additional information to identify the concept. We can think at it as a second-level concept.
  - An optional weight that tells the probability the term is related with the given concept. When omitted this defaults to 100.
  - An optional description for the term that can be used in a user interface when using a supervised strategy to map the models.

Example 1

The following example shows the definition of just two concepts, product and lot.

```
# This is example 1
*Name=Erp

// This is the product concept
Product
    :Material
    :Article: 100
    :Ingredient : 80

Lot
    :Batch
```

The first outcome of the compilation of this file is the `KnownConcepts` class:

```
public class KnownConcepts : KnownBaseConcepts
{
    /// <summary>
    /// This is the product concept
    /// </summary>
    public static Concept Product = new Concept("Product", "",
        KnownConceptSpecifiers.None);

    /// <summary>
    /// </summary>
    public static Concept Lot = new Concept("Lot", "",
        KnownConceptSpecifiers.None);
}
```

The second is the `KnownTerms` class

```
public class KnownTerms : KnownBaseTerms
{
    /// <summary>
    /// </summary>
    public static Term Product = new Term("Product");
```

```
        /// <summary>
        /// </summary>
        public static Term Material = new Term("Material", "");

        /// <summary>
        /// </summary>
        public static Term Article = new Term("Article", "");

        /// <summary>
        /// </summary>
        public static Term Ingredient = new Term("Ingredient", "");

        /// <summary>
        /// </summary>
        public static Term Lot = new Term("Lot");

        /// <summary>
        /// </summary>
        public static Term Batch = new Term("Batch", "");
```

If there were concept specifiers in the file, there would be a third similar class. But in this case, there are just those lines added to the `Domain` class (the class definition and other details are omitted for clarity):

```
Links.Add(new TermToConcept(KnownConcepts.Product,
KnownConcepts.Any,
  KnownConceptSpecifiers.None, KnownTerms.Material, 100));
Links.Add(new TermToConcept(KnownConcepts.Product,
KnownConcepts.Any,
  KnownConceptSpecifiers.None, KnownTerms.Article, 100));
Links.Add(new TermToConcept(KnownConcepts.Product,
KnownConcepts.Any,
  KnownConceptSpecifiers.None, KnownTerms.Ingredient, 80));
Links.Add(new TermToConcept(KnownConcepts.Product,
KnownConcepts.Any,
  KnownConceptSpecifiers.None, KnownTerms.Product, 100));

Links.Add(new TermToConcept(KnownConcepts.Lot, KnownConcepts.Any,
  KnownConceptSpecifiers.None, KnownTerms.Batch, 100));
Links.Add(new TermToConcept(KnownConcepts.Lot, KnownConcepts.Any,
  KnownConceptSpecifiers.None, KnownTerms.Lot, 100));
```

Example 2

In the following example the term "Recipe" is considered a description only if it is found in the context of a Product (which is a concept).

```
HumanDescription
    :Description: 100
    Product:Recipe
```

Example 3

In this other example both the "Net" and "Gross" terms are associated with the mass unit of measure but they have different concept specifiers because it is important to be able to distinguish between the two. The "Gross" term also specifies the "100" percent weight (which is the default and can be omitted) and a convenient description to provide some valuable information in a user interface.

```
UnitMass
    :Weight
    :Net[Net]
    :Gross[Gross]: 100: Gross weight
```

While the DSL dramatically simplify the organization and classification of those parameters, we have to considerate of course the additional work that is necessary when adopting the Semantic Domain Modeling approach.

The first aspect to consider is the operational steps necessary to write and organize these parameters.

Creating the text file

Since SDM proposes an abstraction over the public model, there are two possible starting points. The first is to create the file as previously illustrated. Most of the difficulty is identifying the list of terms that makes up the glossary in the domain.

The second is a code-first approach, which is also the best way to start when SDM is applied on an already existent system. In this case we can use a tool to extract all the words that are used in the class and property names.

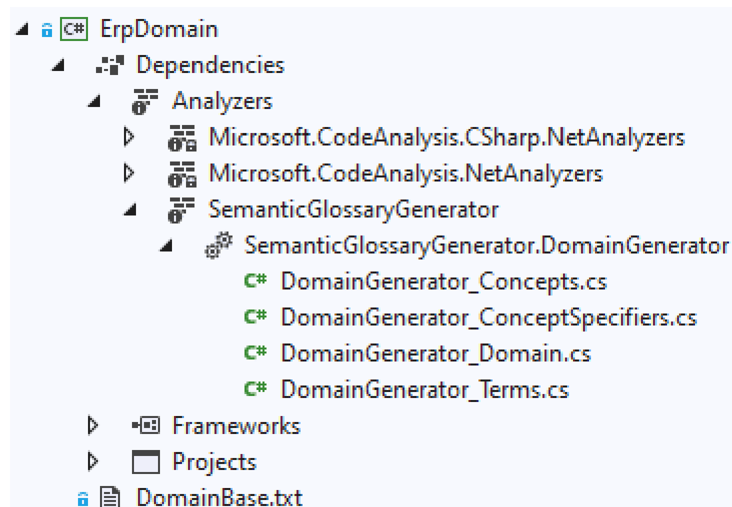
Once the list of terms is created, it may be refined with the following steps:

1. Make sure that special terms that we want consider as a single word are joined back. We previously mentioned that "ZipCode" makes more sense as a single word rather than the two terms "Zip" and "Code". Since we use the tool to create the glossary, it can't know what are those special words.
2. Correct all the terms so that they are singularized
3. Group together all the terms expressing the same concept and move at the top of the groups the term we want to use as concept.
4. Format the text file according the DSL rules, eventually adding the weights and concept specifiers.

The DSL compiler in the proof of concept is integrated in the .NET IDE environment. Once the text file is added to a project referencing the code generator of the DSL, it automatically generates the strong-typed classes corresponding to the declaration in the text file.

The generated code

From the developer perspective the definition of the domain which link terms, concepts, weights and concept specifiers all together, is straightforward.



Every time the `DomainBase.txt` is modified, the `SemanticGlossaryGenerator` automatically compiles it and generates four files whose content was discussed before.

This special .NET generator, known also as "C# code generator" is fully integrated in the build process so that it can also run in a headless context without the IDE.

The advantages to obtain the strong-typed definitions of all those parameters are multiple:

- The syntax of the text file is concise, faster to write in comparison to manually write the strong-typed definitions.
- The compiler makes a number of validations, such as duplicate declarations or the use of incorrect characters.
- The user can write rules and validations using the strong-typed declarations which avoid error-prone mistakes such as misspelling terms or string manipulations.

Every time the developer wants to add policies, rules and validations, she can simply add them to the collections defined in `DomainBase` which is the base class for the generated `Domain` class.

The domain language

We have already mentioned the similarity of the DDD "ubiquitous language" with the terms we defined in the Semantic Driven Modeling domain. It is interesting observing that everything we mentioned in this chapter can be reused just for DDD purposes, even without the automatic mapping, which is instead the primary goal of the SDM.

The domain language is currently very simple and just defines the terms and their links with the other parameters. But there is nothing preventing to extend it to simple rules that don't need to be defined with algorithms.

For example, the domain language can be extended with the following features:

- The type system definition. This would define the base types that are allowed when modeling the model.
- The constraints between a concept and one or more data types. For example, we could restrict unique identifiers to Guid or money to decimal.

The advantage of this approach is to apply all the constraints, rules and validations based on the policies to the design-time and never at runtime. These validations can be run as unit tests as well, and thus fully integrating with the development cycle.

Chapter 5. Semantic metadata

In the previous chapter we have seen how to create, validate and build the strongly-typed parameters defining the domain which totally decoupled from the type system used to define the models. In this chapter we are going to examine how mix up the structural information of types and properties with the semantic parameters obtained from the class and property names.

The proof of concept accompanying this thesis is entirely developed using .NET which has some interesting peculiarities.

The C# language provides the concept of "properties" which are a language abstraction over a field. When defining a property, the compiler will automatically produce the getter and/or setter methods as specified in the property declaration. The underlying field can be explicitly defined in the language or automatically generated by the compiler. Other languages, such as C++ and Java, do not provide the "property" abstraction, but everything mentioned in this paper for properties is equally valid for simple fields.

Another important point is that .NET, as well as other programming platform like Java, provides the ability to introspect the code. Introspection consists in the ability to retrieve the structural information about types, fields and properties at runtime. In .NET the introspection is called "Reflection" and is used in the proof of concept to retrieve the class names and properties. Other languages such as C++ can obtain the same information by parsing the source code. The most popular library in C++ is "libclang" which is part of the Clang front-end compiler.

Even if the provided code is based in .NET, the Semantic Driven Modeling is totally platform agnostic. In order to enforce this decoupling, we defined an abstraction layer over the type system so that any other language can read and write the type definition abstractions and provide a platform specific implementation.

The type system abstraction layer

The types that represent the abstractions on the type system limited to just the definition of types and properties (or fields) and do not provide a full definitions of a classic type system.

The requirements we defined for the "TypeSystem" abstraction are:

- The definition of all the built-in basic types: boolean, Guid, string, signed and unsigned version for integer at 8, 16, 32 and 64 bits, floating point at 32 and 64 bits, numeric decimal (BCD representation), DateTime (with or without the time zone).
- Support for generic types (normally used for lists or dictionaries) with a maximum of two generic parameters.
- Ability to incrementally add new types with all the dependent types recursively.
- Ability to serialize and deserialize the entire types system definitions in JSON format.
- Ability to compare two type systems for equality using standard operators (operator ==).
- Ability to attach an external type to annotate the type system. In our discussion this external type is used to attach the semantic metadata used in the Semantic Driven Modeling.

The type system acts as a container for all the types and properties used in one of the two models. A second instance of the type system will be used for the other model to convert to or from.

Inside the type system there is an indexed collection of types and an indexed collection of properties belonging to those types. It is important to work with indexed objects for a few reasons. The first is avoiding to duplicate objects upon serializations. The second is because types and properties form a complex graph which often causes circular references. Finally, the last reason is that indexes allow to greatly reduce the serialization of the model.

The other two important abstractions are the "SurrogateType" and "SurrogateProperty" which respectively abstract the type and property definitions. They share the same requirements defined before. Since everything is based on indexes, the serialization never includes its dependencies, but just the indexes.

These two classes, in addition to abstract the underlying Type and PropertyInfo of the .NET platform, also provides properties to easily identify one-to-one or one-to-many relationships that are not explicitly identifiable in a type system but are well-defined in the JSON serialization. Their detection is based on the presence of specific .NET interfaces in the type hierarchy. Additionally, a specific search key-value pair interface is done, to identify one-to-many relationships shaped as map/dictionary.

The utility class called SurrogateVisitor allows to visit a graph of type definitions by walking the references from a type to another using properties. The visitor pattern provides a simple way to execute some action every time a type or property is met during the walk. This is used by the upper layers.

The serialization format is optimized to avoid duplicates. Upon serialization every type is only defined once and identified by a 64-bit incremental integer. The index for basic types and complex data types uses two different numeric ranges to ease the human ability to recognize them.

As soon as the type system is deserialized, all the dependencies for each SurrogateType and SurrogateProperty objects just contains the indexes. A special method UpdateCache called over the type system updates all the definitions so that the graph can be navigated using references.

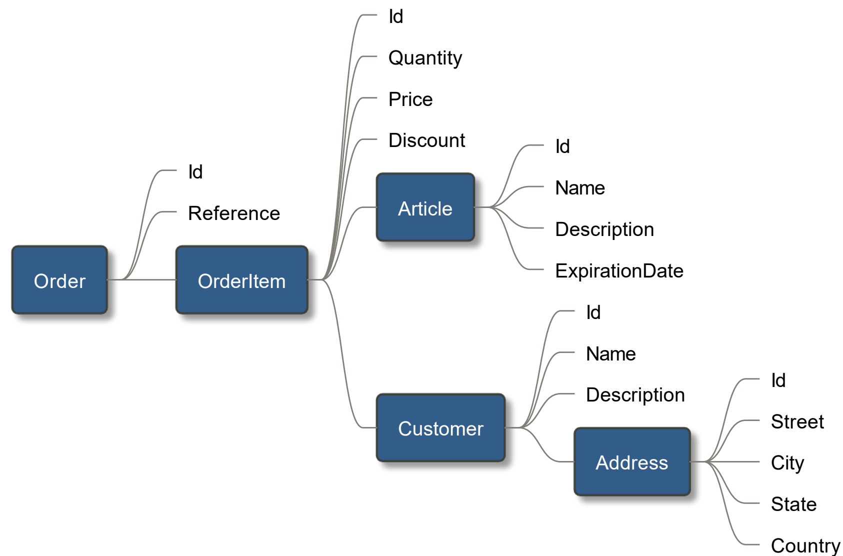
The TypeSystem also defines methods allowing to retrieve the original .NET Type and PropertyInfo objects. They are part of the introspection capabilities in .NET and are very important because are used to dynamically generate the code that implements the cross-model serialization and deserialization. A similar implementation can be provided by other programming platforms.

This abstraction layer is implemented in a separate library called "SurrogateLibrary" which does not have any dependency out of the .NET base libraries and is able to attach external objects that can be used as annotations over the type system and its types and properties. A set of unit tests provide the validation for the initial requirements such as equality and serialization.

Navigating the model graph

An additional use of the previously defined abstraction over the type system is the ability to easily navigate back and forth the graph of the two models. This additional feature will be used during the serialization and deserialization process in order to map a property of the source model on a different property of the target model.

In the following object graph, the fields are the leaves of the Graph. Some of them like "Country" can be reached by navigation Order, OrderItem, Company and finally Address. Others, like Discount, can be found by just walking from Order to OrderItem.



A class called NavigationSegment represents a segment of the path and consists in one link to the previous segment and another one to the following. This structure is very convenient because preserves the complete path by passing a single node as argument. Also, by just getting the last segment in the "Next" direction, we get the leaf of the path which is the node that contains the value to be either serialized or deserialized.

The NavigationSegment class also adhere to the previous requirements in terms of serialization and equality so that we can store in a json file a specific navigation starting from the root of a model to the leaf of the graph.

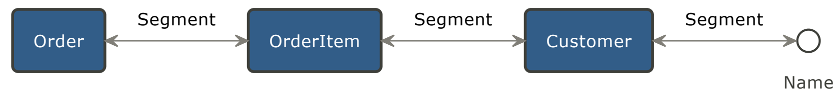
In addition to the aforementioned characteristics, this class provides a number of important capabilities that are used in the proof of concept:

- The string representation from the start of the graph to the current node.
- The ability to find a node by the string path.
- The ability to use a functional-oriented style to walk the path backward and forward.
- Exposing some important information stored in the underlying type or property, such as being a collection or a collected type, which is important in the serialization and deserialization process.

The basic idea behind the possible paths in a model graph is that, regardless the complexity of the graph, at the very end, the relevant data is stored in a basic type which is always in the graph leaf.

Since our goal is the ability to map two models which may have totally different graph structures, we can flatten the graph as it had a single depth level, but still preserving the information of multiplicity (collections and arrays) as well as the semantic information that we will discuss later in this chapter.

The flattening process is done by walking the entire graph and collecting all the segments for the paths that start from the provided root type to all the leaves at the end of the graph.



In the proof of concept, the utility class called GraphFlattener takes the SurrogateType type for the root entity of a model and returns a list of all the possible paths leading to the leaves available in the graph, carefully avoiding any possible circular reference.

All those abstractions and utility classes are the building block for the next level, the semantic layer which attaches the semantic information to the type system.

The semantic layer

The semantic layer sits above the type system layer and provides everything is required at both design-time and run-time to enrich a model with semantics, create the automatic mappings to a different model and finally, at run-time, providing the tools to perform the custom serialization or deserialization from one model to the other.

This layer is implemented in the proof of concept in the SemanticLibrary library and provides the following characteristics:

- The definitions for all the semantic parameters, namely the types Term, Concept, ConceptSpecifier and the aggregating type TermToConcept.
- The base classes containing the definitions of terms and concepts that are common to any domain.
- The analysis and creation of the metadata that is attached to the model types and properties. This includes a lexical helper that extracts the class and property strings and normalize them.
- The algorithms that looks for the best automatic mapping between the source and target models. The result is then reported in types that provides the pairs of the paths to the leaves from one model to the other.

The classes storing the semantic parameters are the ones used from the DSL to translate the simple text format in strongly typed classes. Each of those classes defined by the DSL always derives from a base class that provides the elements that are common to any domain. For example, the KnownBaseTerms contains terms like "Id" or "Name" which are popular and are not ambiguous in any domain. The KnownBaseConcept and KnownConceptSpecifiers types contain similar common definitions. Finally, the DomainBase class contains the aggregation of all the above parameters thanks to a collection of TermsToConcept instances defined in Links, as already discussed in the DSL chapter. When the DSL produces new definitions, the generated code derives from these classes so that the newly defined domain is composed by the union of the base definitions and the ones provided by the user.

The semantic metadata is obtained by walking the model graph by mean of the SurrogateVisitor. Every time a type or property is met, a lexical analysis on its name is performed. This analysis considers the following elements:

- The naming convention (Pascal case or camel case).
- Keeping composed words together.
- Singularizing the plural terms.
- Avoiding any duplication of names.
- The hierarchical structure to correctly assign the Concept based on the context specified in the DSL.

After the analysis, the metadata obtained through those steps is attached to the SurrogateType and SurrogateProperty types across the whole type system, but of course excluding the basic types. By serializing the type system, the information about the model types and the semantic metadata are saved and can be reused at a later time or on a different platform. As this information never changes after the development time, the best moment to perform the analysis is at build time, during the process of Continuous Integration.

Ideally the outcome of this serialization is exposed in a conventional endpoint of the owner of the model, pictured as "Producer" in the following diagram.

Every time a consumer needs to map its own model with the one exposed from the producer, it can simply download the json and **understand the meaning** of each node of the json graph that is emitted by the producer in the normal Web API endpoints.

This information can now be used to perform either a manual, fully automatic or supervised automatic mapping between the producer and consumer public models.

Mapping two models

We are going to describe the process where the mapping is performed on the consumer side. This is the most reasonable scenario since a producer may publish some data in a bus that potentially has multiple consumers using different models. Ideally, we prefer to avoid the publisher to convert and publish the data for each of them. It is certainly more convenient for the consumers to provide the conversions to their own model.

In any case, there may be times where the conversion must be done from the publisher. For example, there may not be an available implementation of the Semantic Driven Modeling on the consumer side. Another reason may be that the consumer side is a low power device, such as an IoT device where the SDM library would be costly in terms of memory occupation.

The current implementation of the SDM in .NET provides a working solution for both serialization and deserialization. This allows the conversion to be in the producer side and of course this is also useful when the communication direction is inverted (HTTP GET verb vs HTTP POST verb).

Assuming the conversion is done by the consumer, it is also required to perform the same initial analysis that we have discussed in the previous paragraph, creating a realization of the type system in conjunction with the semantic metadata.

Once the consumer has both the producer and the consumer metadata, it may proceed to the mapping's creation. Once again, this is not something that should happen at run-time even if it is definitely possible. The considerations on the content-type negotiation will be discussed in a later chapter.

The SemanticLibrary provides an automatic mapper which provides one of the many possible ways to search and assign the best possible matching between the source and the target models.

Those mappings have the following characteristics:

- A property of the source model may be mapped to multiple properties on the target. This may cause some value to be duplicated on the target side. For example, if the source type Product has a Name property but not a Description, it would make sense to map Name to both the Name and Description properties on the target side.
- A property of the target model can be mapped to one and only property on the source side. If this was not the case, values would be overwritten during the mapping, which is useless.
- It is acceptable for any property, either on the source or target sides, not to be mapped at all. There may be cases, such as for properties marked with the "UniqueIdentifier" concept, where a mapping is mandatory. Anyway, there is no guarantee that runtime values could provide a meaningful value such as the case of an empty Guid for an "Id" property. This example suggests that a validation following the copy is the only way to ensure the mapping is correct.

With regards to the graph shape, we want to be free to map a property to any arbitrary graph level in the target model. For example, the source model may have declared the properties "Address", "City" and "ZipCode" in the "Company" class. On the target model the "Company" class may use a one-to-one relationship with an Address class whose properties are "Street", "City" and "Zip".

This requirement gets a bit more complicated when it comes to collections. In fact, there are three different cases:

1. Source is a collection while target is not.
2. Source and target are collections.
3. Source is not a collection while target is a collection.

The first case is clearly an unacceptable loss of data as there is no space in the destination model to set multiple values deserialized from the source collection. This can be detected and avoided during the mapping phase and it would not make sense supporting it.

The second case is the one falling in the vast majority of cases. If I have a one-to-many relationship on the publisher side, we have to match a similar relationship on the subscriber side.

The third is an edge case but supported. It may make a sense mapping a single value inside one or even in all the items of a collection. In the latter case this would result in a duplication of data, as the data was subject to a de-normalization process. It definitely makes sense supporting this case because it happens when a one-to-one relationship is transformed in a more recent model version to a collection.

Automatic mapping strategies

Now that we have identified the mapping requirements, it is time to choose a strategy to identify the best match between two properties.

We initially discarded the possibility to use an approach based on Machine Learning for two reasons. The first is because it would be very difficult to find a significative number of models to use as the base for the training phase. The second is because every domain may significantly differ and would probably require a different training set in order to provide a meaningful result.

A second hypothesis was to evaluate the choice using a best-path algorithm with costs. Anyway, it is not really relevant considering how "long" is the path to reach a given value in the model graph. The validity of the value is more related to its semantic and context rather than topological.

The final choice, that is the one implemented in the proof of concept, is a score-based evaluation. In a model graph, every entity type expresses a context while the actual data value to be copied to a different model is always represented by the leaves of the graph. In other words, we always have to walk to the end of a branch and reach a leaf to get a value.

Since every branch leading to a leaf provides a context, it is important to preserve the related metadata so that we can provide a meaningful comparison.

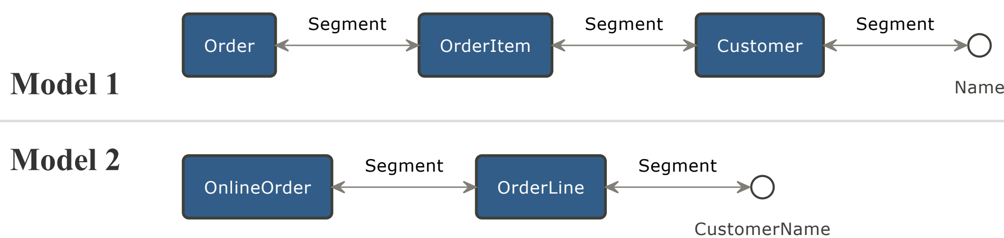
The idea is therefore to flatten the model graph while still preserving all the information related to the path leading to the value in the leaf. This information includes also the multiplicity (whether a branch represents a one-to-one or a one-to-many relationship).

After flattening the source and destination models, the comparison is done by just calculating a score which provides how likely a given property on the source model expresses the same concepts on the target model. The likelihood does not take into consider the data type which we consider just a convenient way to represent the value.

The scoring algorithm takes a pair of source/target properties and calculates the result by considering:

- Custom rules that have been specified in the Domain class by the developer.
- The chain of contexts which is obtained by looking at all the nodes on the path needed to walk from the model root to the current property.
- A score that evaluates the conceptual parameters assigned to the source and target properties.

Every time a pair of properties is evaluated, it gets added to a candidate mapping only if its score is above a minimum threshold. This avoids taking into consideration mappings that are unrelated (zero score) or having a score so low to take into consideration very weak matching.



After all the properties has been evaluated, each target property may have one or more candidates. They are sorted by descending score order so that only the ones having the top score are retained. Even among the top score, there may still be more than a single mapping.

In an **unsupervised** approach, the algorithm does not have any other criteria to discern one. In this case we just take the first available.

In case of **supervised** approach, the algorithm can be hooked with a callback that is invoked every time we meet an equally scored mapping.

Let's now drill down into the criteria that are currently implemented in the proof of concept. The assignment of the best weights to each of them is not necessarily the best and can be probably further enhanced. This is one of the reasons why, in each domain, custom rules can be implemented.

Custom Rules

Custom rules permit to influence the decision of the scoring algorithm depending on the conceptual metadata that is part of the domain.

A built-in rule that is valid through all the domains is the one saying that properties with the "UniqueIdentifier" concept can only be mapped to properties also qualified with "UniqueIdentifier".

Another example could be that every address-related property should always be found in a path with the address context, which means declared inside an Address class graph.

Context evaluation

Context evaluation is important because a property called "Name" may identify a person name or an order name which are totally different.

We have seen that the flattening algorithm preserves the information related to position of a property in the graph. Every time we take into consideration a property, we can walk back to its parent type (the class declaring the property) and then back to the property referencing that class. The jump chain continues until we reach the root of the model.

Every time we walk the graph back, the concept associated with that node represents the context for the previous node.

For example, let's consider an Order class declaring a Name property and a Company class which also has a Name property. The two-Name properties cannot be distinguished as they are both associated with a very generic concept called "Identifier" (which is not unique). As soon as we walk back the graph, we find the Customer property whose return type is Company. The metadata associated with this property tells us that:

- The **Concept** for Customer is Company
- The **Concept Specifier** is Customer

If instead we walk back and find the Order class, the Name property would clearly be an order name.

The graph can be more complex as there may be intermediate entities. For this reason, the algorithm collects all the concepts along all the path and consider them in the evaluation.

Assigning the score

The score is assigned considering all the parameters associated to the properties. A relative increment parameter is first computed by considering whether there is a match between the concepts and concept specifiers. A formula calculates the final score by also considering the number of terms associated to each property and multiplying the result by the increment.

At the end of this procedure, we finally obtain the mappings that will be used at runtime to transform a model into another. These mappings are based on the metadata collected from the two models that we previously described, and can be used during the serialization or deserialization phase to transform a an instance of the source model into the JSON of the target model, or to deserialize a source model into an instance of the target model.

Saving the results

Once all the mappings have been calculated, the results are saved in a class that stores the following fields:

- A string identifier of the current map. This is a convenient name to easily distinguish the maps.
- Two string identifiers matching the type systems used to create the source and target models. They are used to associate the type systems to the maps and therefore also identify the **model version** being used.
- The type for the root of the two models.
- A list of elements where each of them represents the path from the root to the leaf of the graph. These elements form the double linked list discussed previously. These elements also contain the scoring value for each pair.
- The total score

These results are serializable and can be persisted without requiring their calculation at run-time. In order to save space, they are persisted with just the indexes to the types declared in the type system abstraction. Anyway, an "UpdateCache" method is provided to be able to navigate the types through direct references instead of manually decoding the indexes with a lookup.

Other ways to build the mappings

Since we are dealing with distributed systems, it is clear that every single service may be potentially developed with a different platform where SDM is still not available.

For example, if the producer service is written in Java and the consumer with .NET, we may not have the metadata required to create the mappings which are indeed required to make the serialization and deserialization code work correctly.

The first possibility is to just rely on the JSON graph. When analyzing a JSON document, the structure is very poor: for example, the collected items in an array element (between square brackets) are unnamed. Anyway, the path strings in the NavigationSegment type described before may use the "$" marker as a replacement. This allows to code in the proof of concept to work on paths that were built without this knowledge.

Another possibility is to use the OpenAPI metdata to create the type system abstraction and surrogate types so that we can proceed to the analysis as we did for .NET. The information provided by the OpenAPI metadata endpoint are rich and can provide a solid base to build the type system and surrogate types.

A slightly more complex way is to use one of the well-known tools to generate the .NET models starting from a JSON document and proceed with the analysis we described before. If the automatically generated models are close enough to the original ones, the process can take place as it was a .NET service.

The fourth chance is to directly build the mappings. This would require creating an editor that allows a user to manually create the matches between the source and target models.

None of those alternative possibilities are available in the proof of concept, but the abstraction over the type system described in this chapter provide a solid building block to build any of these additional strategies.

Chapter 6. Custom Serialization

In chapter 1, we have already identified the serialization and deserialization process as the most appropriate place where the transformation should occur. In chapter 2 we also anticipated that the proof of concept uses the JSON wire format as it is the most popular format used in the context of the REST services. We also described how the standard serializers work.

We will now dig into the implementation of a custom converter for the standard serializer that is available in the in the `System.Text.Json` namespace of the standard .NET library.

A custom converter has the goal to take care of the serialization and deserialization of a specific data type which is identified as a generic `T` type in the notation. When the custom converter is invoked, it can process just its own type or the entire graph, if any, that descends from it. For example, when serializing an Order having an OrderItems collection, the custom converter may decide to just serialize or deserialize the Order type and let the default serializer to process the OrderItem class. It is also possible to define multiple custom converters so that OrderItem is also processed using another converter.

Since we want to be able to serialize and deserialize models into a potentially different graph structure, we have to process the entire graph in a single step using just a single custom converter. For example, the previous Order and OrderItems may be mapped to a structure having more levels than two or even just one.

A custom converter has to derive a base class and override the `Read` and `Write` methods which are respectively invoked during the deserialization and serialization phase.

We will now analyze separately the custom serializer and deserializer algorithms by only describing the peculiarities that are needed to transform a model into a different one, converting the data types and the optimizations obtained through the dynamic generation of code.

The serializer algorithm

The `Write` method of the custom converter has the following signature:

```
public override void Write(Utf8JsonWriter writer, T value,
JsonSerializerOptions options) { /* ... */}
```

The serialization process is generally straightforward because it works by walking the properties and recursively the children objects. When visiting the types and properties, the corresponding methods on the `writer` object are called.

- WriteStartObject, WriteEndObject are used to delimit the objects and produce the braces in JSON
- WriteStartArray, WriteEndArray are needed every time there is a one-to-many relationship and produce the square brackets in JSON
- WriteString, WriteNumber, WriteBoolean and WriteNull are instead used when a property has a basic type

The main difficulty is the graph transformation, in fact we receive in input the source model but we have to visit the schema of the target model, as it is this last model that is expected to be serialized. It is fundamental to drive the conversion using the target model because we must write the JSON tokens following the order of the target hierarchy. In fact, we cannot jump back and forward the JSON stream being created as it would be extremely inefficient.

In order to visit the target model, we use the visitor class mentioned in the previous chapter. The visitor pattern is used to navigate the model graph and metadata structures. It recursively walks all the graph nodes calling a virtual method for every interesting occurrence:

- A new type has been found
- The current type has finished (no more properties are expected)
- A property has been met
- A collection has started
- A collection has ended

These methods have been designed with the JSON structure in mind and retrieve information such as the multiplicity, the current path relative to the root and others.

In the converter constructor, we build two dictionaries that makes use of the path as the key to efficiently find the corresponding target elements. There are two dictionaries because the direction conversion of the serialization is the opposite of the deserialization.

Serializing objects

The serialization process consists in writing a JSON document for the target model even if the source model is totally different.

We make use of a lookup table that uses the target paths as key because the JSON document must be written following the target model structure. For this purpose, we use the visitor pattern to visit the target model graph described in the metadata. Every time a type is met, we create the corresponding JSON tokens. When instead a property is found, we look in the lookup to find the corresponding property in the source model to copy the value from.

As an additional difficulty to make the whole process flawless and fast, this code is executed only once at runtime in order to dynamically create efficient code for those specific mappings. This allows to reach very performant result, as we will see at the end of this chapter.

Serializing collections

Collections are a special case because the schema that we are visiting just specifies a property with a collection data type. Instead, the object being serialized can contain any number of instances. In other words, there is no one to one relationship between the schema and the content of the object.

Everything that is inside the collection must be repeated multiple times, one for each instance of the collection. This means generating the code for a loop statement that repeats all the assignment inside the collection for each object found inside the StartArray and EndArray tags.

In order to do that, we have to accumulate all the statements needed for the entire sub-tree of the collected items in a separate context. Since the visitor may find other sub-collections, these structures are stored inside a stack.

Once the end of the collection is met, the context of the collection is removed from the stack and its statements are put inside a generated for-each statement which loops over all the items of the collection.

Finally, the parent context will contain the following statements:

- The WriteStartArray statement
- A ForEach repeating the statements described before. They will include the start and the end of the objects specified in the collection.
- The WriteEndArray statement

This strategy allows to keep the complexity controlled and still handle very complex graphs.

Serializing other data types

The other basic data types are simpler to manage because they are leaves of the graph tree. The basic types, when different, are the only ones requiring a conversion.

The problem in this case is managing three different data types:

- The data type originally specified by the source property.
- The data type used in the JSON stream and that we need to use to correctly extract the data from it. The JSON type system is smaller but different than .NET or other languages other than javascript.
- The data type specified in the target property.

Since a JSON string data type can hold strings, Guid, DateTime and other data types, we have to carefully consider all of them, eventually shortening the expression conversions whereas the data type change is needed.

Later in this chapter we will briefly see the conversion engine which provides the convenient methods that are used in the generated code.

Accessing the object instances in the graph

We previously said that every method of the visitor provides a `path` that is used to retrieve the mapping between one source property and the one being evaluated.

In order to access the correct value to be copied into the target property, we have to access the correct objects in the source graph. This is fully described from the metadata obtained through the `path`, in fact we have to just walk back in the graph until either the root or a collection is found.

When the model root is found, it means we can build an access path starting from the object passed to the json converter. For example, a Temperature property can be obtained in the path DataSample.External.Temperature where DataSample is the root object.

When instead there is a collection, the object used as root is the iteration variable in the generated for-each statement, used to loop all the items in the collection.

The two cases can be distinguished through the metadata and the context information being held inside the context stack.

Null management

The conversion engine accepts the null and transform them to the default value of the destination property. For example, .NET reference types such as strings are mapped to the null value; all the other basic types such as double, decimal or integer have zero as the default value. By hooking an appropriate conversion, it is also possible to set different values for each data type.

Anyway, a special consideration is made when a property pointing to an object (one-to-one) or a collection (one-to-many) is null because a "null-dereference" may occur and cause an exception. For example, is External is null, de-referencing it to access Temperature would cause an exception. The countermeasure is adding a null-check in the generated code, for every segment of the path described before.

Serializing the object graph

The entire work described before consists in a long number of statements that are equivalent to the work that a developer could do, if she knew how to serialize the types in the target model.

The final step in the code generation is putting all those statements inside a function so that it can be dynamically compiled and executed.

In the .NET proof of concept, the entire code generation is done using the Linq Expressions where the function is created with a lambda expression which expose a Compile method. The compilation is very lightweight and produce a delegate which is the equivalent of a function pointer.

The generated lambda takes in input the `writer` object and an object of type `T`. Its delegate can be cached so that the code generation process is not needed anymore for the same type `T`.

Since the traditional JSON serializers do not dynamically generate code, the generated code has several advantages from a performance perspective that we will investigate later.

The deserializer algorithm

The `Read` method of the custom converter has the following signature:

```
public override T Read(ref Utf8JsonReader reader, Type
typeToConvert, JsonSerializerOptions options) { /* ... */}
```

The `reader` object provides access to sequence of JSON tokens, conveniently decoded. The `typeToConvert` specifies the type of the object that the converter is expected to create and returned as a generic type `T`. The `options` contains some settings that are not relevant in this discussion.

The deserialization algorithm is different from the one described before because the `reader` object allows to extract the JSON tokens from a sequential stream. While we previously used the target model structure to drive the algorithm, now we have to do it from the source model structure.

For example, if we have the source object X and the destination model graph has A.B.C, we must be ready to support the case where we map a property of X to a property of C even if A, B and C do not exist yet.

For this purpose, during the converter creation, we create an additional lookup that quickly permits to find the paths to the objects that are required to be in memory in order to deserialize the value.

While it would be possible, the deserialization code is not entirely code generated to lower the code complexity. There are anyway several statements that are generated dynamically but they are limited to all those parts that would suffer from performance if they were implemented with just the introspection technique.

The core part of the algorithm is a loop that continues to extract JSON tokens from the stream. At the very beginning we will receive a `StartObject` token which we can use to create the target object. Since the structure may contain nested objects or arrays, we use a stack to preserve the knowledge of the level being served. As soon as we reach the `EndObject` token and the stack is empty, we know the deserialization process is concluded.

When a `PropertyName` is found, we temporarily store the property name and wait for its value. This information must be saved again in the stack because the value for the property can be of any type among arrays, objects or basic data type values. If it is a basic data type, the context is removed from the stack as soon as the value is retrieved. If instead is an object or an array, the context is removed from the stack when an `EndArray` or an `EndObject` are found.

Creating object instances

As we already mentioned, it may happen to find a mapping specifying a target property of an object that does not exist yet. Every time we have to assign a value, a method deals with the problem to retrieve the correct instance, eventually creating it if it has not been found.

This strategy requires to maintain a dictionary associating the string with the path of the object in the target model to the corresponding object.

The method tries to retrieve the object from the dictionary first. When it is not found, it must walk back the parent objects and repeat the same operation because even the parent objects may not exist yet.

As soon as the first object is found or created, the property used to walk back the tree is assigned with the child object. In the A.B.C example, if we are trying to create C, the algorithm performs the following steps:

- We walk back from C to A, because B and C still do not exist. A is found because it is the root and has been created at the very beginning.

- B is created, the property of A pointing to B is assigned and B is stored in the dictionary
- C is created, the property of B pointing to C is assigned and C is stored in the dictionary
- C is returned to the caller

All those steps are currently implemented using standard code since they are generic and there would be no performance improvement in generating them dynamically.

The value obtained from this method can be finally used to assign the value that has been deserialized from the JSON stream.

We now have the object and the property to assign, as specified from the map for the JSON tag being deserialized. The assignment can be finally done, but it may require a data type conversion which definitely benefits from the code generation. In fact, the .NET type system makes a distinction between "value types" (allocated on the stack) and "reference types" (allocated on the heap). If we treat as a generic "object" a value type, it would be subject to a "boxing" process that wraps the value type inside an artifact allocated on the heap. Conversely, the opposite procedure consists in "unboxing" the value type. In addition to those operations, the procedure would also require to introspect the types in order to call correct conversion method for the types discovered at run-time.

The code just described has a great performance cost but can be totally avoided by dynamically generate the code with the hints provided by the metadata. Later in this chapter we will see the micro-benchmarks providing some relevant number of the performance gain.

Removing objects from the cache

Every time there is a collection/array in the JSON document, the objects that were successfully deserialized and filled with the values must be removed from the cache because otherwise the new element in the collection would overwrite the values of those objects.

This is where we have to use the additional lookup mentioned before. In fact, it allows to quickly find the objects that must be removed from the cache. Those keys points to the entire graph of objects in the sub-tree belonging to the collected item.

Convert and assign the value

As we saw in the serializer, there are three different type systems into play: the source property, the JSON and the target property data types. The pseudo-code that we need to generate in order to convert is a function very similar to the following:

```
void Func(ref Utf8JsonReader reader, Object inputObject)
{
    ((Article)inputObject).ExpirationDate = reader.TokenType ==
JsonTokenType.Null
        ? default(DateTime)
        :
((ToDateTimeConversion)_conversion).From(reader.GetDateTimeOffset())
;
}
```

The generated lambda takes in input a reference of the `reader` object and the object to assign.

The `inputObject` is first casted to the target data type that we obtained from the target model metadata. The casted object is then accessed in the `ExpirationDate` property which is assigned with a value.

The value is evaluated with an inline conditional expression. In fact, if the token in the JSON stream is null, we have to assign the property to its default data type, in this case DateTime.

If instead the JSON token is not null, the converter is called into the `From` method. The converter instance is captured from the JSON converter and casted to the correct type. The value from the JSON stream is obtained by calling the method matching the source data type, in this case DateTimeInfo.

The JSON value is therefore subject to the following conversions:

- From string to DateTimeInfo using the reader object.
- From DateTimeInfo to DateTime using the converter (From method).

In the serializer we had just a single function for the whole serialization process. In this case we have several smaller functions, one for each target property. Similarly to what we did in the serializer, these functions can also be cached and reused at a later time to obtain the performance advantages.

Conversion Engine

One of the requirements we wanted to fulfill is the ability to map two properties having different data types. For example, a Product Lot can be expressed as a number from one model and using a string in a different model.

These changes are purely a representation issue and do not modify the nature of the value being converted. For example, a weight can be converted from double to the decimal data type but, by default, the conversion engine does not modify the number.

It is important remembering that we restricted the validity of the semantic assumption to just the services belonging to the same domain. In fact, this defines the boundary of the assumptions we have made, such as the unit of measures. In other concept, every time we assign a concept, we can safely make an assumption regarding its unit of measure.

Anyway, there are times where we may want to take control over the conversion. This is possible because all the conversion methods provided by the engine can be hooked in two different ways:

- By deriving one of the conversion classes and overriding the virtual method for a specific conversion.
- By providing an instance of the conversion context which contains the delegates (function pointers) that returns the formats to use in the conversions.

Even if .NET supports the generic programming, it is not convenient to define a generic conversion method because every conversion is very different and may even not be required at all. From the performance perspective, conversions can be very expensive.

The conversion engine has the following goals:

- Converting any basic data type to another one with the best code in terms of performance.

- Allowing nullable types (wrappers over a value type adding null support) and null conversions.
- Supporting hooks to take control over specific conversions.

The basic data types supported in the proof of concept are 8-, 16-, 32- and 64-bit integer in both signed and unsigned flavors, 32- and 64-bit floating points, decimal, DateTime, DateTimeOffset, TimeSpan, boolean and Guid.

In order to provide the best possible conversion, there is one converter for each data type. Each converter provide a From method with as many overloads as all the basic data types. This basically provide a long list of permutations that were manually coded to provide the best possible conversion.

When converting from a string, the conversion is done using the Parse method offered by .NET with a long list of possible formats for the dates, times and numbers. For more flexibility, it is possible to hook the conversion engine and take control over the parsing operation.

It is quite obvious that not every permutation among those data types can lead to a convertible value. The outcome of a conversion can be one of the following:

- Allowed. For example, assigning a 16-bit integer to a 32-bit integer is just an assignment
- Lossy or potentially dangerous. Assigning a 32-bit unsigned integer to a 32-bit signed integer is potentially lossy operation. Another lossy operation is assigning a double to a decimal.
- Not supported. For example, converting from Guid to integer cannot take place.

There may be cases where the developer may want to support a Guid to integer conversion by creating a lookup table. This is another case for hooking the conversion engine.

Separating the conversions from the JSON converter is very convenient because it can be easily replaced. It also removes any complexity from the code generation algorithm which can just take care of dealing with the source and target data types.

Performance considerations

The performance topic is particularly relevant in a distributed application because the time spent from the infrastructure to transport a piece of information from one process to another is perceived from the user as unproductive latency.

Depending on the application complexity and structure, a single call may translate in a cascade of calls to several sub-services and where the total latency would result in the sum of all those latencies.

Also, every time the producer and consumer use a different model, typically the consumer will first deserialize the data into a model resembling the structure of the publisher and immediately after will copy this data into its own model. This second step must also take into account the time spent from the Garbage Collector (for programming frameworks like JVM and .NET) to get rid of the objects used for deserialization purposes.

The total time spent in the calls will be therefore the sum of these parameters:

- (Number of calls in the chain) * (serialization time + deserialization time)
- Time spent in copying the deserialized model to the one owned by the consumer and used by the Garbage Collector.

For this reason, the performance considerations in the serialization and deserialization process are a key factor to consider when we plan to put our hands on the data in transit.

The algorithms described earlier in this chapter to serialize and deserialize a model into a different one, can be implemented by just relying on the dynamic interpretation of the metadata collected at development time. For example, in .NET this can be done by relying on the Reflection (introspection) capabilities of the language which is what every standard JSON serializer generally use. This is an important point, because we can consider this technique as a fallback strategy for every language, like C++, which does not provide introspection capabilities.

In this hypothesis, the algorithms presented here would cause an additional work, but they would also avoid the need for copying the data after the standard deserialization.

Instead, in the proof of concept we decided to use the code generation technique to improve the performance profile which has anyway a little price to pay. The first time the serialization or deserialization occurs, there is some additional time spent in generating the required code. Once obtained the delegates, they can be cached and reused at a later time.

Therefore, from the second time on, the serialization or deserialization code is compiled and can run much faster.

The real performance gain provided by the code generation consists in these main areas:

- The time required to retrieve the method descriptors (introspection).
- The time needed to invoke methods and assign values (introspection).
- The boxing and unboxing operations required by the runtime to deal with untyped values.

The first point only occurs the first time when a serialization or deserialization occurs. They can be cached similarly to what we do in code generation and reused at a later time. This is an expensive operation because this information is extracted from the metadata that is held in the binary assembly file containing the code.

The second point is a time spent for every execution and cannot be avoided. The overhead derives from the fact that the compiler could not enforce any check, therefore the runtime must guarantee that the operation will not result in an invalid call.

The third and last point is about the separation between value and reference types in .NET that we mentioned before. Among all the data types, the string is the only reference type and all the others are value types. This means that avoiding the boxing/unboxing operations for all the value types results in a great performance gain.

Profiling performance

We have measured the performance characteristics of the serialization and deserialization process using a well-known .NET library called "Benchmark.NET". This library provides a valuable help to avoid common mistakes in measuring the timings, but repeating the tests multiple times. The main advantages in using this library are the following.

- A preparation phase avoids the time spent from the .NET JIT (Just In Time) compiler to generate the native code for the test.
- The same preparation phase also allows to prepare the generated code and store the corresponding compiled delegates in a cache. The measured times takes only into account the effective code resulted from the dynamic compilation and not the compilation time itself. This is acceptable because, in a real use-case, it would happen just once, during the very first request.

- Tests are run several times to avoid outliers deriving from outstanding processes running at the same time on the operating system.
- Mean, error and standard deviation numbers are calculated to properly analyze the results.

The performance separately measures the serialization and deserialization procedures. For each of them, four tests are executed:

- Processing model 1 using the standard library serialization.
- Processing model 2 using the standard library serialization.
- Converting model 1 to model 2 using the mapping obtained through SDM.
- Converting model 2 to model 1 using the mapping obtained through SDM.

Serialization measures

The serialization process was implemented only using the dynamic code generation. The results of the micro-benchmarks are the following:

Method	Mean	Error	StdDev
PlainOrders	9.702 us	0.1862 us	0.1992 us
PlainOnlineOrders	6.832 us	0.1318 us	0.1518 us
SemanticOrderToOnlineOrder	6.108 us	0.1186 us	0.1318 us
SemanticOnlineOrderToOrder	6.064 us	0.0987 us	0.0923 us

The first two are the standard serialization which does not use code generation techniques. Interestingly the other two, thanks to the code generation, are faster even if they have the additional work to deal with a different graph structure and the conversion of data types.

Deserialization measures

The deserialization process is more complex as already described in this chapter. In order to better highlight the advantages of the code generation, this process has been also implemented using the introspection technique.

It is important underlying that a portion of code (the one reading the value from the JSON document and converting to the final data type) is still generated because .NET reflection does not allow to access "ref struct" data types used by the JSON serializer. In order to avoid any performance gain, this portion of code (which uses reflection to build the code) is simply not cached at all. The result is a simulated reflection code that we believe it is close enough of the full use of reflection. Other generated code is instead fully translated to use reflection.

The performance measurements using .NET reflection are the following:

Method	**Mean**	**Error**	**StdDev**
PlainOrders	18.83 us	0.376 us	0.386 us
PlainOnlineOrders	11.42 us	0.202 us	0.189 us
SemanticOrderToOnlineOrder	18,175.24 us	235.091 us	196.312 us
SemanticOnlineOrderToOrder	21,196.31 us	304.601 us	284.924 us

There are three orders of magnitude that severely impact the deserialization process when using reflection.

When instead, we switch to the dynamically generated code that is cached after the first pass, the results are the following:

Method	**Mean**	**Error**	**StdDev**
PlainOrders	18.16 us	0.363 us	0.496 us
PlainOnlineOrders	10.23 us	0.172 us	0.205 us
SemanticOrderToOnlineOrder	32.91 us	0.638 us	0.655 us
SemanticOnlineOrderToOrder	28.59 us	0.519 us	0.433 us

Thanks to the code generation, the performance is comparable even if slightly slower of the standard deserialization. This is due to the additional work needed to create and remove instances from the graph in a different order than the JSON document being deserialized.

Chapter 7. Content negotiation in practical scenarios

In the previous chapters we have seen the various steps that allows to automatically map two models and how they can be transformed in a different wire format during serialization or deserialization.

In a REST based service, an agent (client) may request or send a resource to a service. From the HTTP perspective, the resource consists in the abstract content being exchanged, while the model graph and data types are just a representation of the given resource.

From the Semantic Driven Modeling perspective, we are stating that the semantic level fully describes the resource while its representation depends on the wire format generated by the serialization process.

This chapter focuses on how a client and a server can use the standard HTTP conversation to request a resource and negotiate its representation. After a brief explanation of the standard content negotiation handshake in HTTP, we will see the most interesting use-cases to negotiate a representation, depending on the HTTP verbs and the capabilities of the producer and the consumer.

Standard HTTP negotiation

In the context of REST services, content negotiation is described by the RFC 7231 (https://datatracker.ietf.org/doc/html/rfc7231) and can be done in different ways:

- By mean of HTTP Accept and Content-Type headers.
- Specifying the representation via URL or query string.
- Using a custom header.

We will only focus on the first one which is the most popular and part of the RFC7231. The negotiation is either agent or server driven.

The "Proactive Negotiation" is server driven and consists in the following sequence:

- The client sends a request using the Accept header to specify the preferred formats (one or more).
- The server may fail, typically with the following status codes:

    - Status code 406 (Not Acceptable) is used from the server to say that no representation is available for the request.
    - Status code 415 (Unsupported Media Type) states the requested representation is not supported, eventually specifying the Accept-Post or Accept-Patch in the response to provide alternatives.
- Otherwise, the server succeeds with 200, accompanied by the formatted resource and the Content-Type header specifying the representation format used in the response.

Alternatively, the "Reactive Negotiation" is client driven:

- The client sends an initial request.
- The server responds with a list of alternative representations.
    - The server may decide to provide the resource using the default representation or not.
    - The server provides the list of alternative representations using status code 300 (Multiple Choices).
    - The server may entirely reject the request by using status code 406 (Not Acceptable).
- The client can now make a request in the correct format.

There is no strict regulation on how to proceed and therefore this depends on the agent and server capabilities as well as the data flow direction. When using GET, it is more convenient to use the Accept header to avoid roundtrips. When using POST, the client may try to send a resource using a given format and eventually re-issuing the request if the server rejects it.

Versioning: media type or custom headers

The typical media type used by a REST service is "application/json" which specifies the resource will be serialized in the json wire format. Applications may be more specific about the schema of the json document being exchanged. This is possible by using the syntax "application/[subtype]+json" media type where "[subtype]" is the format while "json" is called suffix. For example, "application/dns+json" is a registered subtype for dns messages using the json format as described in https://www.iana.org/assignments/media-types/application/dns+json.

Types, sub types and suffixes are all subject to official registration procedures as described by the RFC6838 (https://datatracker.ietf.org/doc/html/rfc6838). Anyway, there is a special class of subtypes called "unregistered" where the subtype starts with "x." and that are not subject to any registration. They are equivalent to the HTTP headers starting with "X-". This class of subtypes is described at the following URL: https://datatracker.ietf.org/doc/html/rfc4288#section-3.4.

The proof of concept uses the media types for the content negotiation. Anyway, there is nothing preventing doing the same negotiation using custom HTTP headers starting with "X-".

A hypothetical media-type could be the sequence of the following strings:

$$\underbrace{\text{application}}_{\text{type}} / \underbrace{\text{sdm.}\{\text{DomainName}\}.\{\text{TsId}\}}_{\text{subtype}} + \underbrace{\text{json}}_{\text{suffix}} \quad (1)$$

- "application/" is the standard type used for web APIs
- "sdm" can be the hypothetical prefix that could be registered in the IANA registry.
    - Without an official registration the string should be "x.sdm"
- "DomainName" is the name given to the "Name" variable in the DSL. It is used to identify the concepts used in the SDM mappings.
- "TsId" is the version identifier for the type system that contains the description of the projected (target) type.

The resource (and its source type) is instead obtained as part of the endpoint of the REST request.

Scenario 1. Server publishes multiple versions using only the latest model

This scenario is very interesting because the component of the SDM implementation are only installed on the producer server, which is the one owning the public model, while its consumers are able to always receive the wire format matching the one they are aware of.

In this scenario the server wants to avoid breaking changes to its clients, while still being free to evolve its own public model:

- The server initially publishes version 1 of its model, and several consumers actively use it.
- The server now updates its model to version 2 without preserving the classes used to represent version 1. Thanks to the SDM serialization process, it can still produce JSON documents that represents the old version 1, avoiding breaking changes with its clients.

As soon as version 2 is published, the server is able to provide both version 1 and version 2 by hooking the web service infrastructure that provides the content-negotiation. For example, using ASP.NET Core 5 it is possible to add one pair of "formatters" associated to each media-type. Therefore, when a client requests a resource, the server picks a different formatter depending on the media-type requested by the client.

This is a case where the negotiation is client-driven, in fact the client requests a specific representation (version) of the resource requested by the agent. As soon as the client gets updated to support version 2, it will be able to start requesting the new format by specifying an updated media-type. The standard negotiation also allows the client to discover all the supported representations for the given resource, whereas more than one should exist.

The same can be done during a POST, by specifying the media-type in the header together with the resource. On the server side, the formatter detects the correct handler and executes the deserialization of the resource according to the format version specified in the client headers.

It is important noting that, in this scenario, the client is completely unaware of SDM and does not have to use any SDM library.

This scenario is very similar to the **event sourcing use-case** where the service stores slices of a model that is evolved over time. The historical serializations stored in the database may go out of sync with respect to newer versions of the model. Similarly to what we have just described, the SDM may provide the correct deserialization process from the older JSON documents to the most recent version of the model.

Scenario 2. Server-side conversion of different client versions

In this new scenario, the client is going to update its public model. For example, it can be an IoT device pushing data to an IoT gateway or in a bus. Since it is a small device, it is not convenient implementing the infrastructure to transform the data from the updated client model to the old one or even the model controlled by the service on the gateway.

The IoT scenario is emblematic because it is very likely that a large number of devices are part of the distributed system. This means that is difficult updating all of them at the same time and that supporting multiple versions at the same time is more desirable by far.

As just said, the communication channel can either be a bus where the devices can post messages containing, for example, the data sampled on the field. Otherwise, it can also be a direct call made from the device to the IoT gateway.

In both cases the service must be prepared before any device being updated so that no data gets lost. This can be done in two different ways:

- By updating the service and re-deploying it with the additional map of the SDM infrastructure. This can be convenient when the service needs additional business logic to process the new data contained in the new model.
- By dynamically adding the new map (for example a file) to the running service that is already using the SDM infrastructure. Since SDM works on its own abstractions, it can be easily reconfigured to dynamically recognize a new version, load the new map and be ready for the new version of the data as soon as one or more devices are updated.

Without SDM, the update process is usually done by exposing a different endpoint on the service side that is able to process the new data but this solution comes with a lot of infrastructural problems like reverse proxies and firewalls. Furthermore, the server side must continue to load and support old code with may be subject to security issues as well and quickly become very hard to maintain.

Once the service has been updated with the new map, the communication flow becomes very easy.

- An old device can continue to push data to the service using, for example, an HTTP POST that specifies the Content-Type header to "application/sdm.IoT.Sample.IoTv1+json"
- The updated devices can start pushing data to the service using the header value "application/sdm.IoT.Sample.IoTv2+json"
- On the server side, the formatter infrastructure recognizes the Content-Type header and set-up a different map for the data being serialized.

Since the formatter intervene at each request, the process is completely transparent to the rest of the service code which does not need to be modified at all. The differences between processing version 1 from version 2 are entirely encapsulated in the deserialization process which depends on the map that has been prepared and tested offline.

Scenario 3. Dynamically exposing multiple reading models

In modern architectures such as CQRS, it is quite common exposing more than a single reading model, each one exposing a different schema over the same data. For example, the main Order-based complex graph is exposed to the distributed system clients to fulfill the UI needs. Anyway, the printing service need the same data but flattened to just a master-detail relationship to produce the reports. Other reading models may be created for searching, syncing with external systems and more.

The main model is the one providing the more extensive description of all the available data and is desirable to query the data with an ORM (Object Relational Mapping) library, the additional reading models are just for projection purposes and are not used for any business logic. From the REST perspective all the other models are just a mere representation of the same data.

A common issue in this scenario is that the additional models must be manually generated in code along with the transformations needed to populate their objects. In other words, there is a lot of manual work and the service must be re-deployed.

The abstractions of the SDM allows to provide additional reading model using just a configuration approach:

- A hypothetical user interface (not developed in the proof of concept) can create the mappings from the main model to the desired reading model. There is no need for the new reading model to exist because the data will be serialized straight from the main model objects to the JSON format represented by the mappings.
- The mapping file is deployed on the service exposing the reading models along with the media type desired naming.
- The service may auto-discover the presence of the mapping file and automatically load and update the configuration.
- Finally, the client may start requesting the new media type and receive the JSON for the new reading model.

Given there are no performance hits caused by the transformation, the reading model design process can be demanded to the client who is free to create a new projection.

Scenario 4. Server-side versions converted on the client-side

In a microservice-based architecture, a service updating its own model directly impacts on all its clients. Updating all of them at once is very complex and may lead to down times.

It is not unusual for a service to require a model update. The most typical situation is when it is preparing a new feature but the model is not ready to represent the new data. As soon as the new feature will be deployed, the clients will also need to be updated to leverage the new functionalities. Anyway, before this happens, the clients may continue working with the portions of the data they are able to consume from the new version of the service model.

This scenario is quite complex and may lead to two different strategies.

The **first strategy** is optimistic and consists in letting the clients automatically calculating the SDM mappings and using the new model coming from the service. We call this "optimistic" because we rely on the dynamic calculation of the mappings instead of deploying the mapping on the clients. Anyway, the risks to employ wrong mappings can be drastically minimized by running tests validating the correctness of the automatic mappings in a test environment. Once the mappings are satisfactory, the new model on the server side can be published. The SDM proof-of-concept does not provide any tool to instruct/hint the choices of the automatic mapper, but in future releases this could be done by enriching the metadata.

The **second strategy** is applied when the mappings are consolidated in the test environment and manually deployed on the client side. The manual deploy of the mappings does not necessarily mean re-deploying the client code. It can be done dynamically by just making the client load the new mapping definitions.

The content negotiation workflow in those two strategies is the following.

- The service and the client are up and running with the version 1 of the model.
- The media type on the service side specifies the version of its current model, version 1.
- In the **strategy B**, if the client manually gets the new mappings, this is the point in time where this happens.

- The service is updated with version 2 of the model.
- As soon as the client make the first request to the service, it specifies the "accept" header using version 1 in the media type.
- The service answer with HTTP status code 415 or 406 refusing the response for that request.
- In the **strategy A**, when the client wants to dynamically generate the mappings, it proceeds by downloading the type system describing the metadata for the new model from a special endpoint of the service, possibly provided in the last answer of the service.
    - The client can proceed to calculate the new mappings and storing them locally.
- The client uses the new mappings to make a new request, indicating the version 2 in the media-type.
- The service responds with the data and a successful HTTP Status code 200.

A similar handshake is performed for the HTTP POST verb.

Chapter 8. Conclusions

In the first chapter we have seen the implications of SOAP and REST on the versioning problem, the former based on a strict schema and the latter on the hypermedia loosely-coupled strategy. By observing the current distributed panorama, it is quite clear that neither of the two provide a concrete solution to the versioning issues and evolution over time. Furthermore, with the advent of containers and cloud computing, the fragmentation of the services is increasing making versioning management harder than ever.

Regardless the attention we can pose to anticipate any future evolution of a system, reality is that, before or later, we will need to make some change to the service model. If we make peace with that and we acknowledge that a transformation is healthy for the system, the problem shifts to deciding what other information is needed to make the whole process automatic. Some programming languages provide a way to tag classes, fields and properties with attributes that are then stored in the compiled binary and accessible through some APIs. In SDM we decided not to embrace this strategy but instead add the metadata as part of the type system information, so that the type system and the metadata describing the meaning of each property can be consumed from any other language. The ultimate outcome is the need for metadata to make the mapping algorithm mimic the decision that a human would do. We explained the reasons why we believe that machine learning or the best-path algorithms are not appropriate to solve this problem and we decided instead to require a sort of discipline during the modeling process, which is also the reason for the name Semantic Driven Modeling.

The metadata is therefore the key to let the system create automatically and dynamically a mapping between the consumer and the producer. The two parties are able to freely evolve their own model without the risk to compromise the contract with the others. This freedom is healthy because it does not require to maintain legacy code or being bound from a specific model. The implications on the maintenance should be evident to anyone who lived an experience on real systems.

Assuming this is the real solution to the versioning problem, it is not enough to be adopted. In fact, it must also take into account all the side-effects that the additional transformation abstraction may cause. In other words, in order to be adopted, any working solution should also present other important requirements:

- Being performant

- Provide reliable results
- Easy to adopt
- Usable in hybrid environments (different programming languages)
- Playing well with the existent technologies

Most of the implementation that accompanies this thesis was designed to provide a solid proof to these points.

The performance goal was fully achieved by using the code generation techniques available in .NET. On other platform where the dynamic code generation may not be available, the code can be generated "offline" and dynamically loaded to avoid any service disruption. Code generation is becoming more and more popular in many high-end applications. The Microsoft .NET team is also making use of these techniques as the .NET runtime and libraries have already reached very high optimization levels.

With regards to reliability, this is extremely important for a proposal that allows to change the shape of the data at runtime, without the need to restart the service. In order to avoid any service disruption, the SDM was designed to produce and test the mappings before deploying them in production. Every time we are planning a change in a model, the system can produce a proposal on the mappings and validate them through design-time tests (for example, component tests). Once the validation is complete, the new mappings can be deployed in a running environment which will safely update and start supporting the new transformation. There should not even be the need to provide a roll-back strategy as the transformation is always guided from the other party. In fact, we have seen how the previously discusses Content-Negotiation always provides the choice over the model wire-format.

Adoption is one of the most important obstacles to make SDM successful. The advantages for the developer should be very clear in terms of time spent to embrace a new technology. For this reason, we decided to add the Domain Specific Language that compiles few lines of text into 4 different classes which would be otherwise very long to write by hand. There is a great potential for future improvements here. The simplicity, compactness and expressivity of the DSL language is directly proportional to its adoption. There is also a great space to write a tool that initially produces the DSL text language by parsing the code of an existent model. This last-mentioned tool is not currently implemented but it could be a great idea to improve the adoption.

Many enterprise-wide solutions are written using different languages. Sometimes this happens for historical reasons, other times because certain domains have consolidated libraries to implement the business logic. Since this is a proposal, it was clearly clueless implementing the SDM projects in multiple languages. Instead, we concentrated our efforts in abstracting everything could be specific to .NET, which we used to implement the proof-of-concept. The SurrogateLibrary is a key project in this sense because it abstracts the type system and any reasoning on the data transformation is done using this library. In addition to that, we needed to make this abstraction serializable so that any model represented using this library could be eventually used by other languages. This approach makes the portability to other languages far easier. In fact, we can think to port just the SurrogateLibrary to Java so that we can serialize a Java model which can be later loaded from the current .NET implementation which takes care of computing the mappings. In other words, there is no need to port the entire implementation into other languages, but just few of them.

The last point was to play well with existent technologies. This is because no one wants to sacrifice the application design even if can resolve the versioning problem. A first example is to be respectful of the REST and OpenAPI specifications. This goal was totally fulfilled as described in the content negotiation chapter. By no mean, any external service outside the application should be aware of the SDM structure in order to consume the service. Any external consumer will never see the difference between a traditional or SDM based service. On the other side, an SDM consumer sharing the same domain, will be easily integrated and benefit from the SDM remapping facilities.

Beyond those points, Semantic Driven Modeling is not just a name but also represent the one and only important requirement to make the whole process work smoothly. In fact, service models that does not use meaningful and unambiguous names for class and properties will probably require a significant refactoring. On the contrary, models designed using the DDD guidelines should not require any additional effort and the adoption of SDM could be straightforward.

During the development we have made several tests over the models of many open-source projects. In order to have a significative number of models available, we decided to get the ones ERP (Enterprise Resource Planning) space. The glossary of terms we presented in this thesis was augmented with the terms we found in those models that were not refactored to test the SDM code in the worst possible conditions. All the unambiguous terms were successfully mapped, even when a dramatic change of the graph structure was

needed. On the other hand, all the terms that were not enough meaningful were poorly mapped.

In every case, the naming issue is not a blocker. There is still the possibility to supervise the mapping process and manually force the mappings. This process can be tough, especially when the graph structure is very different. Even when the mapping is done manually, the system can still benefit from the code generation of the transformation as we described in the previous chapters.

The mappings present another aspect that may be evolved in future versions of the SDM implementation. They are currently not reversible. In other words, the mapping from one model structure to another are not necessarily symmetric. We did not go further to this point because, in real use-cases, it happens quite often that a transformation is needed only in a single direction. For example, if we need to export data towards a system, it is unlikely that the opposite transformation will ever be needed. Anyway, this may be valuable in other scenarios and the scoring system could take into favorable account the mappings that can be transformed bi-directionally without a significative loss of information.

We also want to say a last word on a very recent technology called GraphQL (by Facebook) which consists in a specification that allows to query and specify the graph depth of the data coming from a REST service. On the client side, the consumer uses a query language to specify which portions of the graph should be returned from the service. On the server side, the producer of the data interprets the query and builds the response shaped as requested. The resulting JSON document has the same shape of the original service model, but some of the branches and leaves of the serialized graph are cut to save bandwidth.

SDM does not overlap on GraphQL and they can co-exist, but we may think for the future to adopt the same query language to provide, on the fly, the desired shape of the graph including, thanks to SDM, the change of the graph structure and data types. This should not be too hard to implement because the GraphQL query could just contain the concepts to query instead of the real fields to extract.

The pivotal point of the Semantic Driven Modeling proposal is about obtaining information about the concepts and the meaning they have for humans. We strongly believe that programming languages should support this category of metadata which could truly take off Domain Specific Languages and dynamic code generation which are still a

niche strategy.

Bibliography

[1] M. Fowler, "Event Sourcing", 2005. Available at: https://martinfowler.com/eaaDev/EventSourcing.html

[2] RDF Working Group, "Resource Description Framework". Available at: https://www.w3.org/RDF/

[3] OWL Working Group, "Web Ontology Language". Available at: https://www.w3.org/OWL/

[4] Web Hypertext Application Technology Working Group, "MetaExtensions". Available at: https://wiki.whatwg.org/wiki/MetaExtensions

[5] E. Evans, "Domain-Driven Design: Tackling Complexity in the Heart of Software", 2003.

[6] E. Evans, "Ubiquitous Language", 2003. Available at: https://martinfowler.com/bliki/UbiquitousLanguage.html

[7] Linux Foundation, "OpenAPI Initiative". Available at: https://www.openapis.org/

[8] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures", 2000. Available at: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[9] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, "Web Services Description Language (WSDL) 1.1", 2001. Available at: https://www.w3.org/TR/wsdl.html

[10] M. Gudgin, M. Hadley, N. Mendelsohn, J.J. Moreau, H. Frystyk Nielsen, A. Karmarkar, Y. Lafon, "SOAP Version 1.2", 2007

[11] D. Crockford, "ECMA-404. The JSON Data Interchange Standard", 2000. https://www.json.org/json-en.html

[12] G. Young, "CQRS". Available at: https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf

[13] Linux Foundation, "gRPC specifications", 2021. Available at: https://grpc.io/

[14] P. Helland, "Life beyond Distributed Transactions: an Apostate's Opinion", 2016. Available at: https://queue.acm.org/detail.cfm?id=3025012

[15] Object Management Group, "Model Driven Architecture", 2014. Available at: https://www.omg.org/cgi-bin/doc?ormsc/14-06-01.pdf

[16] Arnarsdóttir, K. & Berre, Arne-Jørgen & Hahn, Axel & Missikoff, Michele & Taglino, Francesco, "Semantic mapping: ontology-based vs. model-based approach Alternative or complementary approaches?", CEUR Workshop Proceedings, Vol 200, 2006. Available at: https://www.uio.no/studier/emner/matnat/ifi/INF5120/v07/undervisningsmateriale/EMOI06Ppaper.pdf

[17] D. Beneventano, N. Dahlem, S. E. Haoum, A. Hahn, D. Montanari, M. Reinelt, "Ontology-driven Semantic Mapping", 2008. Available at: http://www.dit.unitn.it/~p2p/RelatedWork/Matching/Ontology-driven_Semantic_Mapping.pdf

This page intentionally left blank