


Institut für Software & Systems Engineering
Universitätsstraße 6a D-86135 Augsburg

Towards a Data-Driven Enterprise Resource Architecture

Constantin Gerstberger

Master's thesis in the Elite Graduate Program: Software Engineering


Institut für Software & Systems Engineering
Universitätsstraße 6a D-86135 Augsburg

Towards a Data-Driven Enterprise Resource Architecture

Matriculation number:	1279626
Started:	31. April 2014
Finished:	15. January 2015
First assessor:	Prof. Dr. Alexander Knapp
Second assessor:	Prof. Dr. Bernhard Bauer
Supervisors:	Dipl.-Inf. Univ. Ralf S. Engelschall
	Dipl.-Inf. Univ. Achim Mueller


SOFTWARE ENGINEERING
Elite Graduate Program

ERKLÄRUNG

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

I, hereby certify that this thesis has been written by me, that it is the record of work carried out by me and that I have not used anything else but the indicated sources and tools.

Augsburg, den 15. Januar 2015 Constantin Gerstberger

Abstract

CONTEXT In recent years, the REpresentational State Transfer (REST) architectural style has become one of the commonly employed approaches for realizing client-server business applications. Among other things, this is because it promises various benefits, such as standardized communication (e.g. based on the Hypertext Transfer Protocol (*HTTP*)) between and independent evolution of distributed system components.

MOTIVATION Though being labeled REST-compliant, many implementations however seem to ignore various of its core principles; Most prominently: Hypermedia As The Engine Of Application State (*HATEOAS*). As a result, developers often encounter a broad range of problems seemingly not being addressed by REST. Typically this surfaces in questions such as “How do you design URLs in REST?” or “Why should you put links in representations when working with a well known application programmer interface (API)?”.

CHALLENGE Potentially, one of the main problems concerning REST is that its concept of hypermedia based interaction is often explained based on how humans browse the Web. While not being wrong, this approach distracts from one important difference concerning enterprise applications:

Nowadays, their clients are often implemented as so called *rich clients*. In order to provide a unique and elaborate user experience, these clients commonly represent a (more or less) functional application on their own which, however, relies on specific data exchange with a server.

According to the idea of hypermedia, *rich clients* would need to determine respectively employed server interfaces at runtime - just as humans access websites, figure out their content and subsequently decide what to do. While humans normally do not have any trouble in this regard, implementing a machine to be able to do the same is anything but easy. The result are applications being build on style often referred to as *pragmatic REST*. While following some of its principles, they ignore the most central one: hypermedia.

Consequently, the main challenge with regard to enterprise applications using *rich clients* is machine-driven interaction: Enabling a machine client to determine and interact with these features exposed by a server which are not handled by a human user.

APPROACH To address these problems, this thesis first covers some concepts being relevant for the combination of enterprise applications and REST: Good API design, enterprise applications and remote procedure call (*RPC*) interaction style. Next, it discusses the conceptual side of REST while highlighting various conflicts of business-process and REST-driven environments. After that, the thesis goes into detail with regard to implementing hypermedia, covering status quo and problems concerning REST-based, machine-driven interaction. In order to structure the identified problems and to improve the possibilities in this regard, the thesis then presents an ontology and approach combining the advantages of heavily human and machine-readable documentation. After discussing the application of said approach for an enterprise ap-

plication architecture, the thesis concludes with the description of some experiences made during the implementation of a respective server prototype.

Contents

List of Figures

1. Introduction

As an introduction, this chapter presents an application to be used as an ongoing example through out thesis. To give an overview, it first explains the general concepts of the application and discusses its main use cases. After that, the chapter covers structure and relationship of its underlying entities.

1.1. Overview and Use Cases

The overall goal of the application is to allow people (*speakers*) giving a talk, presentation or lecture to get realtime feedback.

For that purpose, the application lets them prepare a relatively simple set of questions along with weighted, predefined answers (a *questionnaire*). During a talk, the audience (*voters*) can repeatedly vote for a specific answer to reflect their current estimation.

To actually provide feedback, the application provides a continuously updated representation of the average vote per question based on submitted votes and weighting of answers. This representation is accessible by *speakers* as well as *voters*

Use Cases

As shown in figure 1.1, the application altogether features six uses which will be covered in the following.

Authenticate

As part of the *Authenticate* use case, the clients obtains some way (e.g. token) which subsequently can be used for authentication. Mainly, this is required for the *Manage*, *Publish* and *Retract* use cases.

Manage

The *Manage* use case involves creating and editing questionnaires including their questions and answers. Authentication allows each *speaker* to manage his or her own set of questionnaires. To prevent votes from being traceable in case of low submission rates, a *speaker* can define an analysis delay which allow votes to only be used for average vote calculation when having been submitted for a specified duration.

Publish

When intended for voting, a *speaker* can publish a previously created, not already published questionnaire making it visible and accessible for *voters*. Before publishing,

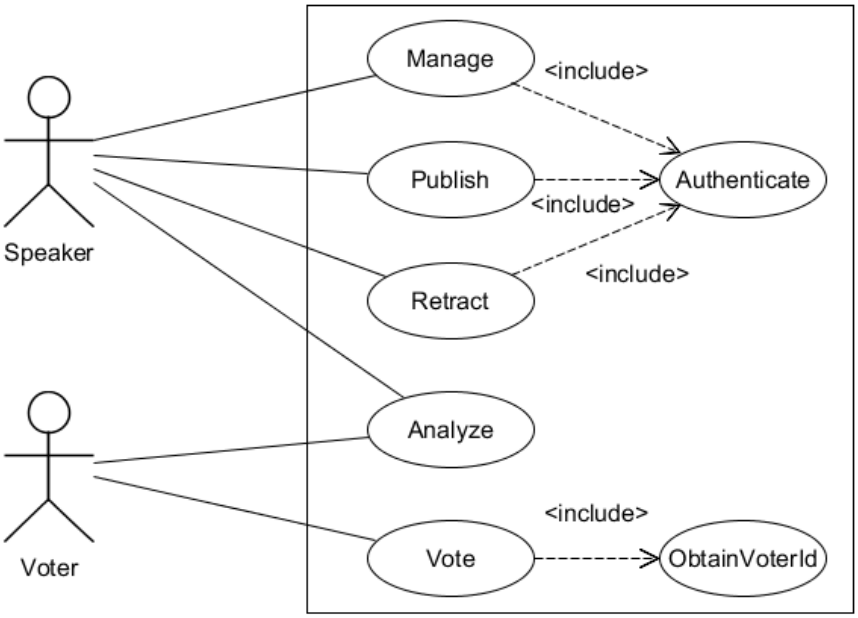

Figure 1.1.: Use case diagram of the example application

a *speaker* must define a unique *identifier* by which the *questionnaire* is thereafter accessed.

Retract

To end the possibility of voting, a *speaker* can retract a previously published questionnaire.

Analyze

The *Analyze* use case comprises obtaining (graphical) average vote representations of a specific time interval from the server. Votes are first averaged per *voter* before being averaged by question. This allows to balance individual *voters* submitting different quantities of votes.

ObtainVoterId

To be able to vote for *answers* of a *questionnaire*, a *voter* beforehand obtains a voter-id, that is an opaque, rather short-lived token which is mainly required for the *Analyze* use case.

Vote

As part of the *Vote* use case, *voters* access a published questionnaire and repeatedly vote for a particular textitanswer of the *questions* listed by the questionnaire.

1.2. Data Model

Figure 1.2 shows an Unified Modeling Language (*UML*) class diagram derived from these use cases describing six entities: *Speaker*, *Questionnaire*, *Question*, *Answer*, *Vote* and *Voter*. Thereby, each class also lists its anticipated, required methods.

Given that the exact authentication procedure does not really matter for the remainder on this thesis, details on the speaker were left out. To allow for averaging of votes per time interval, the *Vote* entity received a respective *submissionTime* attribute.

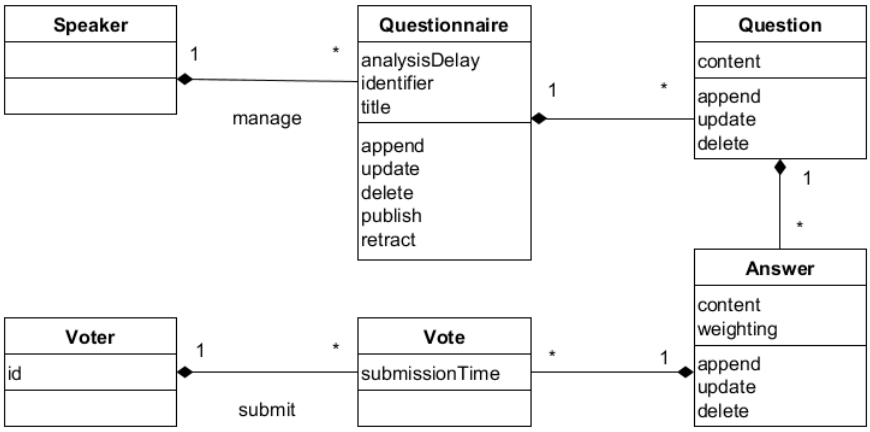

Figure 1.2.: UML class diagram of the example application

2. Fundamental concepts

After the introduction in the last chapter, this one covers three basic concepts to establish additional context for the later parts of this thesis.

First, it discusses typical characteristics of enterprise applications with the goal to anticipate common problems and pitfalls when trying to realize such applications in a REST compliant manner.

After that, it outlines best practices for API design in general. This allows not only to better understand the trade-offs induced by the REST outlined in chapter 3, but also to evaluate the design choices with regard to formats presented in chapter 5.

Last, the chapter discusses the basics of RPC paradigm and shows that, in fact, many REST-labeled APIs rather feature RPC characteristics [Fie08].

2.1. Enterprise Applications

Considering that, especially in context of enterprise applications, the majority of APIs claiming to be REST compliant is actually not [RSK12], the question rises whether or not there are fundamental conflicts between the design goals of enterprise applications and REST. In preparation to answer this question in the next chapter, this section hence describes the general nature of enterprise applications.

Unfortunately, the term *enterprise application* is not very distinct. Alternatively, they are also referred to as *information systems* and *data processing* [Fow02], or categorized as *business applications* [Mic14]. Overall, no commonly agreed definition seems to exist.

Nonetheless, the following characteristics are frequently mentioned one way or another:

- Individual
- Process Driven
- Data Driven
- Secured
- Integrated
- Evolutionary

The remainder of this section discusses them in more detail.

Individual

First and foremost, enterprise applications are *business driven*, meaning that they are to provide value to the organization they are deployed in. This is achieved by implementing so called *business logic* which in turn consists of *business processes* and a typically large amount of *business rules*. The latter thereby describes cross-cutting concerns which may affect many processes [Ulr10].

As a consequence, enterprise applications frequently prove to be fairly complex. This is not only due to the sheer amount of business rules to be considered during application development, but also because said rules often interact in unanticipated ways [Fow02]. In addition, they rarely end up being employable in other organizations given that they have been tailored to specific needs and to present a competitive advantage.

Given that enterprise applications are thus built on and validated against business processes - after all, that is how users will evaluate the application - their design is often influenced as well as explained in the following.

Process driven

Generally, an informational process models a transformation of information input to information output by means of executing a variety of ordered functions [Krc10] (see top of figure 2.1).

In context of *single page applications* (SPAs), this model is often altered so that only the information required per process step (i.e. function) is provided at the time the step is performed (see bottom of figure 2.1). In terms of an application client, this corresponds to presenting the user dedicated views to collect information input or enable a decision.

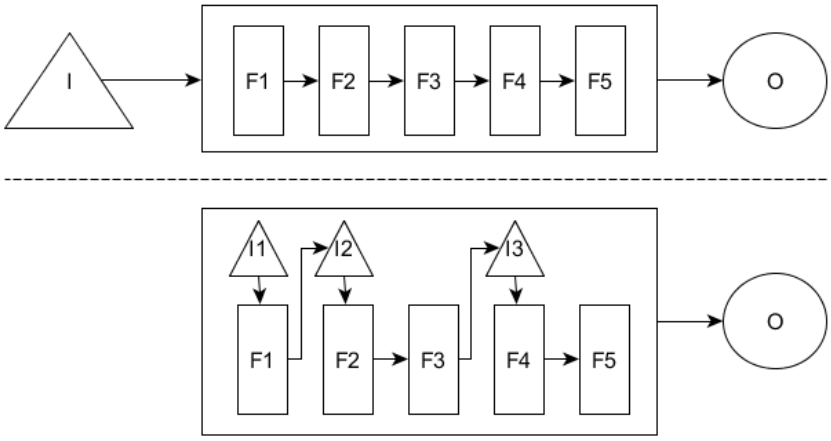

Figure 2.1.: Traditional informational process (top) based on [Krc10] and alternation due to single page applications (bottom)

Interpreting the use cases presented in section 1.1 as business processes, both the *manage* and the *vote* business process require information input based on the *question* and *answer* entities of a *questionnaire*.

With regard to the *manage* use case, a typical *graphical user interface* (GUI) could for instance provide two list-based views: While the first one would serve to manage the *questions* belonging to a particular *questionnaire* as a first (process) step, the second view would allow to equally handle *answers* of a respective *question* in a consecutive (process) step.

For the *vote* use case, this GUI could serve a single view to simultaneously show all *questions* of a *questionnaire* including their *answers* to support the repetitive (process) of voting individual questions.

Following the idea of only providing the required information, these two processes would thus motivate the creation of three interaction endpoints in the client-server based architecture: A first one to provide the list of *questions*, a second one to do the same concerning *answers*, and a third one to support the combined view.

Effectively, the process orientation thus influenced the interface design between client and server. While this is not necessarily good or bad, it is definitely something to be aware of with regard to the next chapter covering the concepts of REST.

Data Driven

Along with business processes usually comes respective data - tons of it [Mic14] - which is persisted for years. Thus, it is not uncommon for data to outlive the environment it has initially been created in. This includes not only hardware components or programming language choices but potentially entire application (stacks) as well [Fow02].

Given that the data is created and used by business processes, you can anticipate users to place convenience and consistency related requirements on application (data) given that they have extensive knowledge of the domain the application acts in.

With regard to example application presented in chapter 1, a *speaker* deleting a *questionnaire* could for example expect the application to clean up dependent entities such as *questions*, *answers* and *votes*.

Security

Enterprise applications typically feature a high emphasis on security [Ora12]. Reasons for that are not only the implementation of business requirements, but also the protection of competitively valuable data as well as the adherence to legal constraints.

Hence, these applications normally implement a sophisticated access restriction system (e.g. represented by roles and permissions) and require users to authenticate early on.

Integration

Even though security is one the primary concerns, enterprise applications are almost never developed and operated in isolation. Instead, they are usually accessed concurrently by many users from a variety of devices and interact with other - internal and external - machines, applications and data all being distributed across the network [Mic14]. Thereby, involved systems have most likely been built at different times using different technologies and collaboration mechanisms [Fow02].

In this regard, a common problem is known as *conceptual dissonance* [Fow02] meaning that involved parties (e.g. organizational units or cooperating companies) have

contradictory definitions of the same term. Due to this conflict potential, integration always needs some degree of isolation. This is to allow applications to remain autonomous and especially to deal with changes locally without disrupting the whole landscape [WFV00].

Maintenance

Business environments are constantly influenced by different aspects such as technology, organizational restructuring, competitors or market changes [Cum02]. For the most part, changes at this level impose the same on respective enterprise applications, i.e. maintenance.

However, implementing the required adoptions is mostly anything but easy which leads to a steadily growing maintenance backlog. In many cases, involved problems are associated with inflexible design generated by ill-designed information structures [AK10] or exaggerated coupling of business logic and technical details [Cum02].

2.2. API Design

As described in the last section, integration is a vital part of enterprise applications. In the context of software development, the standard way to do that is to define an API which the accessing component can use.

Due to this prominent position, APIs have the potential to be either among an organization's greatest assets or liabilities [Blo07]. While a well designed API may significantly increase development productivity, an ill-designed one literally undermines the work of developers. Unfortunately, creating an API of the first kind is rather difficult, while creating one of the second kind is very easy [Hen07]. But what distinguishes a good API from a bad one?

Good APIs are often described as something that "did not throw any more rocks in my way" or "simply did what it was supposed to do". Both these descriptions actually outline an important aspect: APIs are always used by someone to achieve a specific goal. Consequently, the quality of an API directly depends on how well it has been fitted to (or fits in - depending on the perspective) the environment it will be used in [Hen07]. This in turn implies that there is no way to describe exactly how a well designed APIs looks like in general.

Respectively, available sources typically list a selection of rather abstract quality attributes to be found in such APIs according to their experience. An example of such a list is shown in figure 2.2. How to actually achieve, prioritize and balance these properties is left to the API designer (after all, design is all about making trade-offs [CA08]).

Covering all available characteristics would exceed the scope of this thesis. Instead, the following focuses on a few attributes which appear to have been especially emphasized.

Minimalism

Surprisingly, one of the best ways to design a good API is *minimalism*. This means to leave out any feature, functionality or other exposed detail which is potentially unstable, not required or not fully clarified. In other words:

- Easy to learn
- Easy to use, even without documentation
- Hard to impossible to misuse
- Easy to read and maintain code that uses it
- Sufficiently powerful to satisfy requirements
- Easy to evolve
- Appropriate to audience

Figure 2.2.: Characteristics of a Good API by [Blo07]

> When in doubt leave it out [Blo07]

Though the approach may seem counter-intuitive at first, it helps to avoid a variety of problems. This is because you cannot readily change or remove anything that has been *published* by an API in a non-backwards-compatible manner.

Doing it nonetheless will either break any other system that uses the API or prevent it from updating to newer versions and thus from receiving any further, compatible additions and improvements. This may not be that much of a problem if the API is only used by few consumers. But considering the previously described integration of enterprise applications or environments like the Web, there is a high chance that efforts escalate quickly.

What is worse about ill-designed *published* details of an API is that they often cause the creation of workarounds and wrappers. The more widely the API is used, the more of them will appear. Even though its creators typically invest much time and effort to mitigate the drawbacks of the original API, these wrappers often end up being fairly complex and introduce design flaws of their own. A very good example of such a situation is described by [Hen07] based on the *.NET* socket `Select()` function (prior to version 2.0).

One approach which aims to avoid or at least diminish these consequences is using a deprecation strategy. This however forces users to deal with both the deprecated and the replacing feature which increasingly worsens the tangibility of the API. In the worst case, users may even disregard the replacement due to not being able to find and/ or comprehend the reasons for the underlying change.

On the contrary, one could point out that exaggerated minimalism may cause an API to lack essential features. However, this should rarely be a problem given that, in such a case, you know about the potential problem, thus are able test for it and add it later after its necessity has been assured.

Sufficiently powerful

A characteristic which is somewhat opposing to *minimalism* is the requirement of making an API *sufficiently powerful* [Blo07]. One thing that appears to be often neglected in this regard is that sufficient power applies to both success and error cases. A well

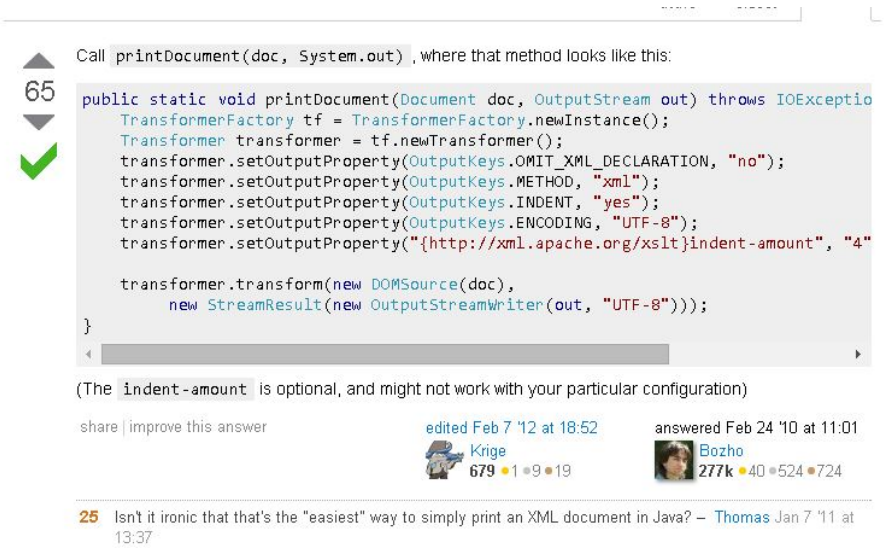

Figure 2.3.: How to print a DOM document in Java: An example of an insufficiently powerful API [15]

designed API hence should provide appropriately detailed and programmatically processible error information [Hen07].

The term *sufficient* thereby highlights the fact that the power of an API can be mismatched in two ways: While insufficiently powerful APIs typically require users to manually perform steps the API could do for them, overly powerful ones usually provide a great number of parameters for each step which are mostly not changed by or relevant to the user. Both cases thus produce lots of boiler code for tasks being perceived as simple by the user. A rather infamous example of such a situation is shown in figure 2.3.

For the most part, such verbosity indicates that the API has been designed without adequately contemplating its usage scenarios. One approach to avoid this problem is to let non-involved developers or known future users specify how they would expect the API to work [CA08].

While being the solution to one problem, this approach can just as well be problematic: Overly focusing on the requirements of some users, may prevent others from effectively using the API. In this regard, doing three different implementations against an API is hence considered to be a good way to ensure its re-usability [Tra95].

However, there is still the problem of perceived complexity: Just because a task is considered simple, is does not mean it actually is. If that is not the case, it remains a design decision whether or not or how complexity should be made visible to the user (e.g. performance heavy operations like recursive tree access).

Intentional Consistency

A good way to reduce the general complexity of an API is to design it *intentionally consistent* due to the fact that it allows users to transfer knowledge (e.g. relationship terms or data structures) they gained about one part of an API to other ones [Hen07].

This reduces the memory load required to learn and interact with the API and thus lowers its perceived complexity - a quality that is associated with well designed human computer interfaces (HCIs) since quite some time [MN90].

Respectively, the methods *append*, *update* and *delete* of all entities shown in figure 1.2 are named identically. Naming one of them *add*, *replace* or *remove* would only force users to learn two terms to do the same thing. It might even confuse them as they could suspect one of the methods to do something different.

The flip side of *intentional consistency* is that APIs should never employ the same names or structures for things having deviant purposes as such design mostly causes errors which are very hard to trace. This is because users typically disregard the actual origin of the error in consequence of assumed consistency.

A good example for such a case are the Unix methods *strncopy* and *bcopy* which have inverted source and destination parameters [Blo07]. Deducing the behavior of one function based on the other would thus result in copying a random piece of memory while expecting specific content. As the result is most likely garbage, the error is not easy to understand as long as the involved function is not suspected.

On a larger scale, consistency thus should also exist between multiple APIs because people generally do not want to repeatedly unlearn one API just to learn another [Blo07].

Documentation

As evident from the part about consistency, misunderstandings when dealing with an API cannot be avoided solely by good design. Given that especially enterprise applications are highly connected and integrated 2.1, there are also possibilities for conceptual dissonances, i.e. incompatible definitions of concepts or terms [Fow02].

An example for such a case is the *answer* entity displayed in 1.2. For the purpose of the application discussed in chapter 1, the entity represents a predefined answer. In another context, the term *answer* could as well refer to the response to a question given by a participant. Without documentation, the “weighting” attribute thus would likely seem pointless and confusing.

Beyond that, it can be hard to anticipate how exactly users will interact with an API:

> “People using your API will be doing things so complicated, you cannot possibly let them do everything without needing to check the documentation.
>
> One way to think about it, is that lines of code can be really powerful because one line of code can save hours and hours of work if you are doing it manually.
>
> Documentation has the same power. One line of great documentation in the right place can save hours and hours for just one consumer of your API. It might tell them that something is possible. That they do not need to build it themselves. That all they need to do is use this method they’ve never heard of” [Lac13].

Due to all these reasons, a good API must not only be well designed but also equally well documented.

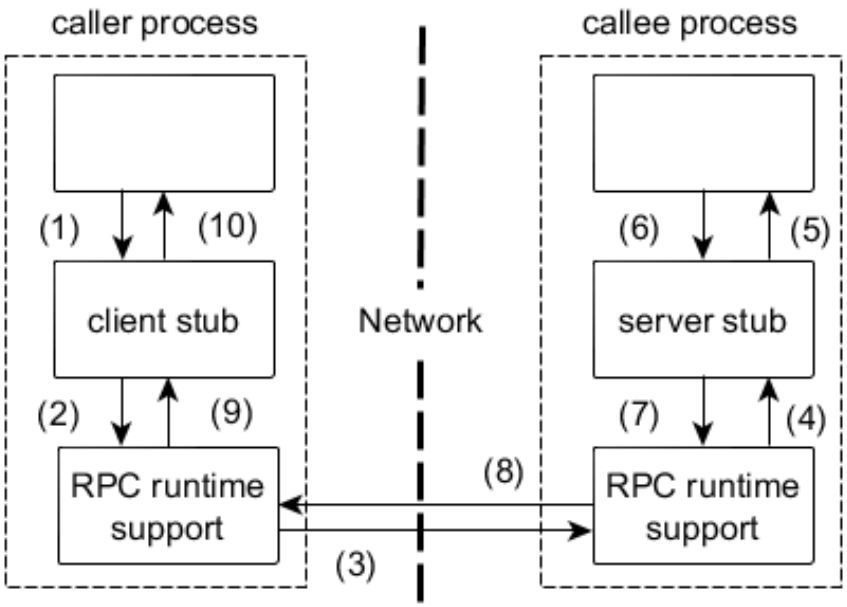

Figure 2.4.: Basic RPC structure and interaction based on [Soa92]

2.3. RPC and enterprise application development

As mentioned in section 2.1, most (enterprise) application claiming to be REST compliant are actually not. Instead, they are usually accused of rather following the *RPC* paradigm [Fie08]. This is why this section covers the basics of *RPC* and describes how its ideas may end up being applied in what was said to be a REST-compliant API.

RPC

Basically, RPC allows a component (*client*) of a distributed system to access functionality of another, remote component (*server*) as if it was local [Soa92]. In object-oriented languages, this pattern is typically extended by way of allowing the execution of methods of an object instead of independent procedures (e.g. *Remote Method Invocation* (RMI) [Ora14]).

In both cases, the client gains access to remote functionality by interacting with a local stub (client stub). Commonly, execution of the calling thread is thereby blocked until the result of the remote execution is returned by the stub.

When invoked, the client stub transforms the call into a message. By means of the RPC support of client and server, the message is subsequently delivered to a stub at the server (server stub). Based on the received message, the server stub executes the respective procedure (or method) and transforms the result into a message.

Subsequently, this message is sent back to the client stub which rebuilds the result and returns it to the calling thread (the complete interaction sequence is shown in

```
1  public interface IResource extends Remote {
2
3      class Questionnaire {
4          /**
5           * Unique identifier
6           */
7          private long id;
8          private String analysisDelay;
9          private String identifier;
10         private String title;
11
12     }
13
14     public Questionnaire getQuestionnaireForId(long id)
15         throws RemoteException;
16 }
```

Figure 2.5.: Example RMI server interface

figure 2.4).

Server interface

To allow for a successful interaction, client and server must agree on a common contract. In case of *RMI*, they thus have to implement the same interface.

In addition, the caller needs to know the address of the callee as well as the name of the procedure (or class and method name respectively) to be executed. For the most part, this information is not provided dynamically to the client, meaning that it is either hard-coded or read from a configuration source during start-up. Correspondingly, the full set-up for remote access is handled during the initialization phase of the client. This allows developers to solely focus on procedure or method contracts which are known at compile time.

As a result, clients are programmed based on accurate knowledge and expectations regarding possible interactions, data that is sent and returned per interaction, and how retrieved data can be used subsequently. Considering the interface displayed in figure 2.5, a developer would thus expect at least two things:

- There is a class called *Questionnaire* which always has the specified attributes
- The id attribute of the questionnaire is unique
- There is always a method called *getQuestionnaireForId* which takes an id in form of a `Long` and returns one object exactly matching the specified *Questionnaire* `class`

The bigger part of that is probably not surprising. However, being explicitly aware of *RPC* based interactions should help to better understand not only parallels to REST-labeled, yet *RPC* accused application, but also differences to truly REST-compliant ones.

```
{
  "id": 1,
  "analysisDelay": PT5S,
  "identifier": "SE101SS14",
  "title": "Lecture evaluation: Software Engineering 101 (SS14)"
}
```

Figure 2.6.: An example for RPC based response representation

Web based enterprise applications

When developing a typical client-server, Web-based enterprise application, client developers usually know where the server component will be deployed both during development and after release. This may be because the application will receive its very own domain or because the same machine will host both the client and the server component (which makes the address available via context). The remaining, relative paths of web based endpoints are typically hard-coded in the client - either completely or in the form of templates.

Given that enterprise applications are business driven, data is normally received or submitted to support a specific process. Hence, you can assume that clients implement assumptions regarding sent and received data; For instance based on the process step underlying each view.

Altogether, the resulting interaction model is indeed quite similar to *RPC*: Available operations, their contracts and transmitted data sets are well known at compile time. Respective servers thus commonly send data representations such as displayed in figure 2.6 because their clients know and expect everything else as specified at compile time. As a result, the mentioned representation does not contain any information such as an indication of it actually constituting a *questionnaire*.

3. REST from a conceptual perspective

Now that the fundamental concepts have been covered, this chapter addresses the conceptual side of REST. Its goal is to explain everything you need to know to understand REST except for implementing *hypermedia* and *self-descriptiveness* which is covered in the next chapter.

For this purpose, the chapter first explains REST in general: What it is, where it comes from and specifically, what it is not. After that, the chapter discusses the core concepts of REST which is followed by covering its well known constraints. Last, the chapter discusses a few common misconceptions concerning REST including the so called *pragmatic* REST style.

3.1. Defining REST

Originally, REST has been developed "to create an architectural model for how the *Web* should work, such that it could serve as a guiding framework for the *Web* protocol standards" [Fie00]. Nonetheless, REST seems to be most known for its constraints which form the REST architectural style.

To understand REST more in depth, three things are useful to know up-front: The concept of an architectural style, the architectural properties considered to be essential according to REST and especially what REST is not.

3.1.1. Architectural Style

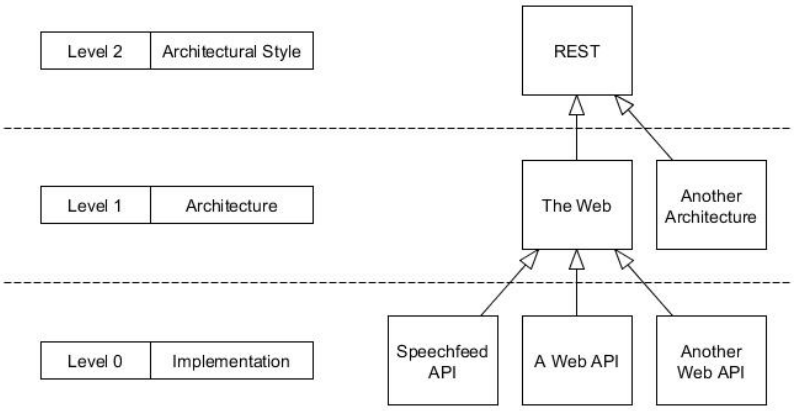

Figure 3.1.: Abstraction levels in context of REST

Basically, an architectural style is an abstraction of conforming architectures which helps architects to communicate architecture in general or details concerning a specific one [Fie07]. Consequently, it represents the second abstraction level above the implementation one [FT02] (see figure 3.1) which is why respective discussions and definitions may not always be very concise.

Furthermore, an architectural style consists of a coordinated set of constraints constituting design choices at the level of architecture [FT02]. Thereby, each constraint "induces certain (good) architectural properties at the expense of certain (bad) tradeoffs" [Fie07] (figure 3.1 shows the design decisions of the REST architectural style).

The fact that the constraints of an architectural style are *coordinated* has one important implication: Designing an architecture while ignoring one or more of them will change the solution significantly.

> REST as a design principle claims to create designs that have positive properties, but it is unlikely that these properties can be expected with designs that ignore certain key constraints [WP11]

In addition, architectural styles commonly do not cover all design decisions. Correspondingly, "REST only elaborates those portions of the architecture that are considered essential for Internet-scale distributed hypermedia interaction" [FT02]. Consequently, it is important to understand "how REST sees the *Web*".

Constraint	**Promotes**	**At the expense of**
Client-Server	UI portability Simplified server Multiple organizational domains	
Stateless	Simplified server Scalability Reliability	Efficiency
Caching	Reduced latency Efficiency Scalability	Reliability
Uniform interface	Visibility Independent evolution Decoupled implementation	Efficiency
Layered system	Shared caching Legacy encapsulation Simplified clients Scalability Load balancing	Higher latency
Code on demand	Simplified clients Extensibility	Visibility

Table 3.1.: Effect of REST's constraints based on [Moo10]

3.1.2. REST's view of the Web

As described by [RAR13], REST identifies four *key architectural properties* which are illustrated in figure 3.2. These are *low entry barrier*, *extensibility*, *internet scale* and *distributed hypermedia*.

In general, these properties are considered to be the main reason for the success of the Web which is why they are prioritized over others. Interestingly enough, you can argue that, in case of Web, the first three ones mostly pose requirements while the fourth property describes a solution for them - which however is based on a set of requirements as well.

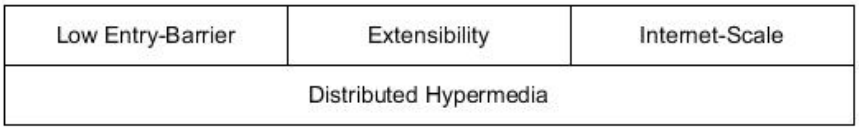

Figure 3.2.: The key architectural properties of the Web

Low entry-barrier

Low entry-barrier is straight forward: Nobody likes to use things whose interaction is overly complicated. This is especially true for the Web because it is based on voluntary participation [Fie00]. Its low entry-barrier is one the main reasons the Web has become that successful. All you need is a browser, no case-specific tools or knowledge of available commands. You can figure out everything else on the way because websites are human readable.

Extensibility

Extensibility refers to a system's ability to change along with its requirements: No matter how well a system may have been designed to match its initial requirements, at some point, the latter will have changed to such an extend that the initial design is no longer sustainable. In the long run, a system which is not capable of adapting to changing requirements is doomed because its former users will quickly leave and search for a replacement [RAR13]. As explained in the last chapter, this is also a major concern with regard to enterprise applications.

> A system intending to be as long-lived as the Web [which has been around for more than 20 years] must be prepared for change [Fie00].

This is why REST promotes the idea of discovering things like resource identifiers, representation or interaction possibilities at runtime: Anything a client is able to determine dynamically at runtime can be changed without breaking effects as long as it semantically stays the same or if the client is capable of understanding the new variant [1].

[1] In some way this is an adoption of the *minimalism* principle given that you try to reduce the amount of information published at design time

Internet-scale

Internet-scale describes that sort of scalability which has anarchic traits [RAR13]. The sheer number of clients and servers makes it impossible to keep track of each other beyond the scope of a communication (that is a session). As a consequence, components of the system will evolve independently as they see fit - coordinated upgrades would not work anyway.

A system which is to remain viable in such an environment thus has to provide the means to maintain old implementations along new ones without preventing new implementations from making use of their extended capabilities [FT02]. Otherwise, clients of old implementation would break upon every change or the system would lack extensibility, both of which, at some point, would mean the end of the system.

Due to the scale of such systems, it is usually no option to deploy changes in a big bang fashion. As a result, they also have to support "the gradual and fragmented deployment of changes within an already deployed architecture" [Fie00].

Respectively, every public *Web* API faces this problem: Generally, there is no way to tell how exactly and when clients will interact with it. To some extend, this also applies to larger enterprise APIs. If they are used by enough systems, you cannot simply change them because it would break dependent systems. And for the most part, it is also not an option to modify all these systems because there is no budget to do so.

The result is very similar to the *Web*: "Un-upgraded clients might stick around for a long time" [RAR13].

Distributed Hypermedia

In typical client-server systems, the server manages information that is organized as data (pieces of information [Fie00]). Furthermore, the server sets and exposes some sort of static, specialized, design-time available interface explaining how clients can obtain and manipulate data. A good example for such an interaction model is RPC (see section 2.3).

In context of the *Web*, this model is not feasible because it denies the server *internet-scalability* and *extensibility*: *Scalability* of the server is limited as it has to deal with the entire application load of all clients. *Extensibility* is restricted because you cannot change said interface without breaking clients that are programmed against it.

To resolve this issue, the principle of distributed hypermedia (also known as the hypermedia strategy) treats the instructions about what you can do with the data just as the data itself [RAR13]. It intends server to send them to clients along with the data which may even allow for their modification (e.g. to control their availability, add or remove them). Just as (standard) information is organized as *data*, control information is thereby organized as *controls* (a control could e.g. be the description of what would otherwise be a specific method of the interface).

However, this change of interaction style has two important consequences:

> Hypermedia is defined by the presence of application control information embedded with, or as a layer above, the presentation of information [Fie00]

First, the interface between client and server no longer serves as a dedicated mediator. There are no more specific methods or other means of interaction by which client

and server can ensure “to talk about the same thing”. This is why REST describes four constraints in terms of an *uniform interface* to compensate for this effect.

Second, hypermedia clients cannot be programmed against specialized, design-time available interfaces simply because there are none by definition. Instead, clients receive hypermedia documents at runtime containing all required data and controls. Based on that, they subsequently have to figure out how to proceed. The “crux” in this regard is that this approach has to work for human *and* machine-driven clients:

> When I say hypertext, I mean the simultaneous presentation of information and controls such that the information becomes the affordance through which the user (or automaton) obtains choices and selects actions. Hypermedia is just an expansion on what text means to include temporal anchors within a media stream; most researchers have dropped the distinction [Fie08]. [2]

As a consequence, hypermedia lowers the entry-barrier in a mostly human-driven client environment. Figuring things out is something that humans are quite good at. Otherwise, the *Web* as we know it today would not work at all. If a human visits a new website, he or she figures out what it is about as well as if there is something interesting to do next. Should some content prove to be rather complex, a website author can simply add some text to explain things.

But as soon as machine clients are involved, quite the opposite is true. Given that machines are terrible at understanding instructions based on natural language, hypermedia actually raises the entry barrier because you have to come up with ways to make machines understand not only the structure but also the meaning of a representation. As suggested by [RAR13], this is called the semantic challenge.

Altogether, “hypermedia [thus] isn’t a single technology described by a standards document. Hypermedia is a strategy, implemented in different ways by dozens of technologies” [RAR13]. The main challenge with regard to implementing a REST API is finding a suitable solution to address the semantic challenge. Details on how to do that are covered in the next chapter after having established more context regarding the conceptual side of REST.

3.1.3. What REST is not

To further specify what REST is, it is important to understand what it is not. The latter includes things like a development framework, a (file) format or a protocol [RAR13].

In fact, a REST API should not be dependent on any single of these aspects though a successful mapping to a specific choice may be dependent on the availability of features such as metadata or (interaction) methods [Fie08]. For instance, the core HTTP standard does not contain a satisfying possibility to partially update a resource representation which is why the PATCH method was suggested [DLS10].

So, despite the fact that HTTP is the de-facto standard for realizing *Web* APIs, REST is especially not limited to or by the HTTP protocol [RAR13]. It works just as well with other protocols such as the Constrained Application Protocol (CoAP) [SHB14]. Interestingly enough, HTTP is not even fully REST compliant itself; For instance because its messages “fail to be *self-descriptive* when it comes to describing which response

[2] Due to that, I will solely employ the term hypermedia in the remainder of this thesis

belongs with which request" [Fie00] (a constraint which is covered later in this chapter).

Likewise, REST does not depend on a single format, such as HTML or *JSON*. From the REST point of view, the latter does not even qualify as a proper format because it violates the (sub-) constraint of *resource identification*. Instead, format employed by an API should be chosen dependent on the type of data provided as well as usage scenarios. In some cases, a more specific but also more restrictive format such as ATOM [NS05] may be suitable while, in other cases, a more generic one such as the *hypertext application language* (HAL) [Kel13] may be more sensible.

Apart from all that, it is essential to understand that REST is not a golden hammer - despite the success of the *Web*. There are definitely cases in which designing an architecture based on the principles of REST makes more or less sense.

Generally, the closer the target environment resembles the *Web*, that is the architectural properties covered in the last section, the higher the chance that a REST based architectures proves to be beneficial. This is also why you can consider REST being overhyped compared to its importance within Fielding's dissertation:

> My dissertation is about principled design, not the one true architecture [Fie07].

3.2. Key concepts of REST

As shown in figure 3.3, REST essentially works based on three definitions: *Identifiers*, *resources* and *representations*. Resources may have more than one resource identifier as well as representations whereas the latter always refer to one resource.

Apart from that, REST features two types of states, namely *application* and *resource state*, which are the reason for the concept of an application being significantly different in context of REST.

Given their central role, these five concepts are discussed in depth in the following.

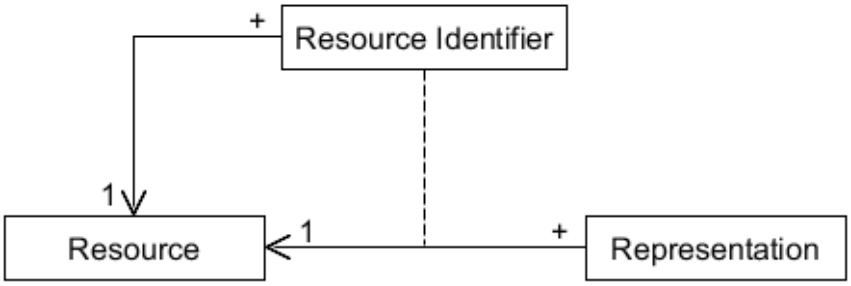

Figure 3.3.: Conceptual UML class diagram showing the relationship(s) between resources, representations and resource identifiers

3.2.1. Resource Identifiers

In one sentence, a resource identifier is a “dumb” key which identifies exactly one underlying resource **and** can be used to access and interact with the latter. To gain access, a resource identifier can be translated into network information by a specific component called “resolver” [Fie00].

Within the *Web*, the common case is to retrieve a representation of the underlying resource. Beyond that, resource identifiers correspond to *URLs* which can be translated by means of the domain name system (DNS).

Technically, *URIs* also qualify as resource identifiers because the original definition by itself only requires unique identification. However, a resource that is only identified but cannot be accessed fails to fulfill the (sub-) constraint of self descriptive messages because there is no way to interact in the first place [RAR13]. As a result, it makes sense to mention this requirement in context of resource identifiers as well.

The denotation of being “dumb” thereby relates to fact that a resource identifier *by itself* does not have any meaning:

> At no time whatsoever do the server or client software need to know or understand the meaning of a URI - they merely act as a conduit [Fie00].

As a result, “resource identifiers do not have to look nice. They do no even have to make sense to human eyes” [RAR13]. You could generate all resource identifiers by using a random generator. In terms of an API, a client even does not need to know any resource identifier at all except for the so called well-known entry *URL* (which does not need to be the root *URL* of the server [WPR10]). From that point on, a client should be able to figure things out as by the idea of hypermedia (just as you interact with most websites). The reason for that requires a bit more explanation.

Stable identifiers are important

Given that centralized link servers are not an option at the scale of the *Web*, it uses embedded resource identifiers and relies on authors to choose an identifier fitting the identified resource [Fie00]. Consequently, changing resource identifiers is extremely counter-productive because it breaks all references to the previous identifier. In theory, there are no reasons at all to change or retract them except for insolvency of the domain owner - “cool *URIs* don’t change” [W3C]. In practice however, things are different.

As just mentioned, people prefer intuitive resource identifiers. But finding resource identifiers that are both intuitive and valid for years to come is anything but easy [W3C]. Often, their quality is proportional to the amount of time and money spent to retain their validity [Fie00].

Extensibility is key

REST however targets very long lived systems such as the *Web* which has been around for (more than) 20 years. So at some point, the requirements of initial system likely have changed to such an extend that the initial design is no longer sustainable. For that case, “servers must have the freedom to control their own namespace” [Fie08] so that they implement the required changes. Otherwise they will be trapped forever by (the shortcomings of) the initial design.

Requiring clients not to know resource identifiers (or other specifics) of an API beforehand thereby allows for the possibility of clients adapting to changes similar to humans when encountering a new design of a website (though the former may prove to be much more difficult than the latter due to the semantic gap).

Achieving balance

Despite all that, it does not mean that REST promotes servers to habitually change resource identifiers or introduce breaking changes. In fact, quite the opposite is true.

As mentioned before, one of its essential goals is to support old implementation alongside new ones whenever possible - but without loosing the ability to move on. If a server has to come up with new identifiers because existing ones where not accurate enough, it should always try to retain old resource identifiers while directing clients to the replacements [RAR13].

This is even more important because, as of today, the majority of machine clients is most likely not capable of adapting to these changes (also known as hypermedia aware). Especially business application clients are usually build as scripts and hence break upon most changes (possibly even intentionally because there are supposed to fulfill exactly one purpose).

Furthermore, REST does not oppose to intuitive resource identifiers. The whole point in this regard is that you must not rely on them being meaningful or clients to guess correctly (after all, you could always guess wrong). So, because humans normally prefer nice-looking *URLs*, a server should also not serve meaningless ones [RAR13].

3.2.2. Resources

To understand the rationale behind resources, it is very helpful to know how *URIs* (and thus resource identifiers) worked in the early *Web*. At that time, their definition suggested that the author would identify the content being transferred (which would correspond to theoretical resource). This in turn implied, that a resource identifier would change whenever the content did, even if it effectively stayed the same [Fie00]. As a consequence, resource identifiers were anything but stable which, as mentioned before, is the opposite of what you want in the *Web*.

To resolve that, resources were conceptualized as an abstraction consisting of all these aspects of the transferred content which would not change over time: Its semantics. Within the *Web* as of today, this idea is quite intuitive:

For instance, when you tell someone about a link pointing to a news article, you normally do that because you deem its topic or (general) content interesting. For the most part, the very exact words or layout of the article is subordinate as long as its conveyed information stays the same. Likewise, you would probably refer someone else to the application presented in chapter 1 due to what it allows to do, instead of how it is presented to you.

Resources are stable semantics

Respectively, a resource is defined to be the semantics the author intends to identify rather than the value (i.e. the transferred content) corresponding to those semantics

at the time the resource is created (or any other particular point in time) [Fie00].

For the *Web*, this definition was additionally important because it allowed for the identification of temporal services such as “the last recently created questionnaire”. The prior definition of resource identifiers made that impossible, because there was no possibility to come up with a stable identifier for the purpose (it would have changed every time the identified questionnaire changed).

This is why the definition of a resource in general is not very concise. Whatever a specific resource constitutes, depends entirely on what its author wants it to fulfill (which is often related to its anticipated usage by clients). “A resource is anything we expose to the *Web*, from a document or video clip to a business process or device” [WPR10]. The only definite restriction is that it must have a *resource identifier* to allow for interaction:

> Do you remember that thing, the thing you had a while a ago, but then... do you know what I’m talking about? Of course you don’t. I wasn’t specific enough. I could have been talking about anything. It’s the same on the *Web*. Clients and servers can only talk about something if they can agree on a name for it. On the *Web*, we use a URL to give each resource a globally unique address. Giving something a URL turns it into a resource [RAR13].

Apart from that, there is only a single, yet important premise:

> Identifiers should change as infrequently as possible [Fie00]

After all, this has been the reason for conceptualizing them as an abstraction in the first place. Hence, storage objects do not qualify as resources [Fie00] because they are technology dependent which makes them susceptible for changes. In addition, they rarely capture the entire semantics (e.g. possibly exposed behavior).

3.2.3. Representations

Given that resources were defined as abstractions, another concept was needed to comprise and transfer the previously mentioned content - in other words to explain the state of the resource: *Representations*.

Following the theme of the definition of resources, REST defines “the things that are manipulated to be representations of the identified resource, rather than the resource itself” [Fie00].

This separation of resources and representations has two main benefits. First, it enables a server to provide multiple, equivalent representations (over time or in parallel) via the same identifier instead of requiring multiple, unstable ones. Second, it allows the server to manage its internal instance of the resource, that is its implementation, independently from what clients get to see of it (namely representations).

Due to both, a server can change its implementation as well as adapt to emerging formats as needed without affecting its client in any way [Fie00]. The popularity shift from *XML* to *JSON* is a good example in this case because APIs committed to XML had many problems (or were unable) to change without breaking old clients that stuck to *XML*.

But what is a representation more specifically? As evident from the explanation so far, a representation belongs to exactly one (underlying) resource, but the latter can have more than one representation [WPR10].

Representations can have many forms

Generally, a representation is a - to some extend - machine-readable document containing any information about its underlying resource [RAR13]. Representations of the same resource can vary in many aspects, for example format (*XML*, *SVG* or even *SQL*), scope (e.g. preview vs. detailed) or language [RAR13]. In this regard, the format is often also referred to as *media type*.

> The *Web* doesn't prescribe any particular structure or format for representations" [WPR10].

Consequently, representations may differ quite drastically despite belonging to the same resource. For instance, the preview representation of a questionnaire may contain much less state information than a detailed representation.

Representations convey state

Furthermore, representations serve two distinct purposes which highlights the way components of a REST based system communicate: Instead of calling operations of specialized interfaces, they exchange messages, that is, representations combined with an intent implying the current or intended state of the underlying resource. As these intents are standardized, the details of how a server may accomplish bringing a resource in the desired state is purposely abstracted by what is referred to as the *uniform interface* [Fie00].

As a result of this model, a client only deals with representations, it never sees or interacts with a resource directly [RAR13]. Correspondingly, you can also define a resource as the abstract interface constraining the representation based communication with the server:

> There are no resources on the server; just mechanisms that supply answers across an abstract interface defined by resources. It may seem odd, but this is the essence of what makes the *Web* work across so many different implementations [Fie00].

Nevertheless, representations are required to communicate the semantics of the underlying resources "indirectly". Otherwise, a client would deal with the state of something it could not understand. In addition, it would be unable to differentiate one resource from another which would violate the requirement of resource identifiers uniquely identifying an underlying resource (within the *Web* this is referred to as an URI collision):

> If the representation communicates the state of the resource inaccurately, this inaccuracy or ambiguity may lead to confusion among users about what the resource is. If different users reach different conclusions about what the resource is, they may interpret this as a URI collision [W3C04]

So, how is a client supposed to understand representations and deduce the underlying resource? The answer to this question is *self-descriptive messages* and *hypermedia*. As mentioned in section 3.1.2, I will defer its detailed coverage until the next chapter.

3.2.4. Resource & Application State

The last concept discussed in this part is the distinction of *resource* and *application state* which is actually an outcome of the *stateless* constraint.

Understanding both is important to see how REST based applications actually work. For that purpose, figure 3.4 shows an overview of their roles and interactions which is discussed in the following.

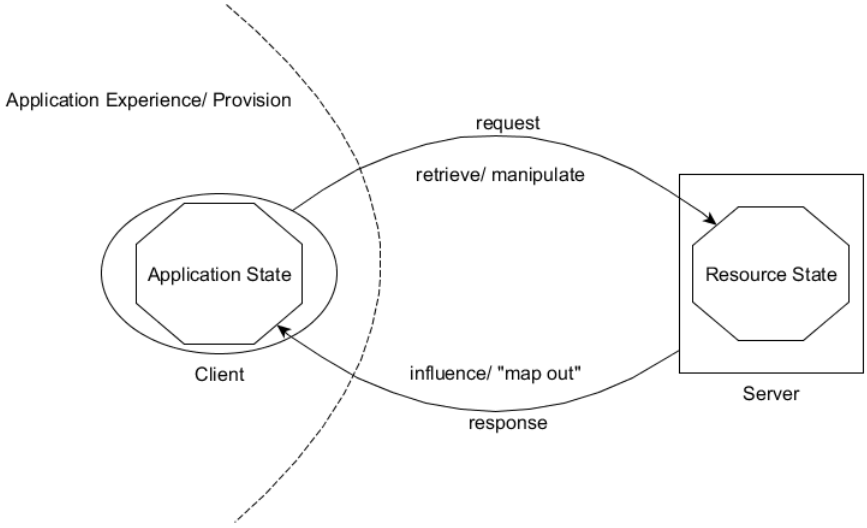

Figure 3.4.: The role and interaction of application and resource state in context of REST

Resource State

In large parts, resource state has already been covered in the last section: Primarily, it is stored and managed by the server; But a client can interact with it, that is view or manipulate it, by obtaining or sending representations [RAR13].

With regard to wording, resource state usually addresses the state of all resources of a server. Single resources appear often to be addressed as “the state of X”.

Applications according to REST

To understand the concept of *application state*, it is crucial to first understand the concept of an application with regard to REST.

This is because in REST-based system, servers (are meant to) solely offer various, logically connected resources as well as interaction possibilities. They do not particularly “care” how these resources might be used. Within the *Web*, this “attitude” is actually an advantage because it leaves a high chance that interaction is not restricted with regard to a specific purpose:

> It is the unexpected re-use of information which is the value added by the *Web* [Ber06]

By contrast, having a "particular usage pattern of resources" [Sin14] - possibly provided by multiple servers [Fie00] - is exactly what makes up an application. So in REST-based systems, the responsibility of providing an application (experience) pertains much more to the client than to the server [Sin14]. In this regard, servers are often not much more than data providers which a client might replace over time assuming that there is no relevant, persisted data at a specific server.

Application state

Respectively, *application state* is all the information maintained at the client side while trying to accomplish a goal. "This information is built up piece by piece based on the *resource state* that is transferred between client and server" [Sin14].

As a result, the server can somewhat influence the application state [RAR13] given that the client - to some extend - is limited by the possibilities of its provided representations. But apart from that, everything else is controlled by the client. How long and how many representations it keeps active, context information build up based on the received representations (browser history, authentication or other, domain specific information), or where the client is in its process towards said goal, all that falls to the control and responsibility of the client.

In this regard, browsers usually replace the bigger part of their application state whenever the user accesses a new website (i.e. resource). By contrast, a client implemented to support the *voting* use case explained in chapter 1 would successively load (and keep) the combined information about the respective *questionnaire* including its *questions* and *answers* even if it is distributed across multiple resources. As a third example, a client implementing the *manage* use case might "only" load all *questionnaires* the *speaker* created but additionally keep previously obtained authentication information to do so.

Beyond that, application state is often associated and thus limited to a single session of the application the client realizes. But as illustrated by the browser history example, the client might as well persist parts of its state.

Application vs. resource state

If parts of the application state are important enough, they can alternatively be implemented as *resource state* on the server. This is usually the case if a single user would repeatedly access the same information (possibly from multiple devices and thus client instances) or if multiple users are to access shared information.

A good example in this case is the shopping cart scenario. You could implement the current state of the shopping cart - that is which items it currently contains - as application state. As a result, the client would need to keep track of the shopping cart and, at some point, would interact with some sort of purchasing resource to actually buy the items. By contrast, shopping carts are usually implemented as resource state (and thus as a resource) because it allows for all sorts of convenience features (e.g. a user can shop over the course of multiple session or using multiple devices).

This is why in general - and corresponding to the rationale of defining resources - anything important enough for the server to care should be realized as resource state

instead of application state [RAR13]. How to make that decision in particular case depends mostly on the domain background of the application as well as whether or not you can influence resources provided by the server at all.

3.3. Constraints

Now that the foundation is covered, it is time to look at constraints of REST. Altogether there are six architectural constraints. However, the most famous one, the uniform interface constraint, in turn consists of four sub-constraints [RAR13].

Most of the other constraints are supposed to be well known and understood. Nevertheless, there still seem to be various occasions in which they appear to be misunderstood or ignored.

3.3.1. Client-Server

The client-server constraint promotes separation of concerns by way of separating user interface concerns from data storage concerns [Fie00]. Ultimately, the goal of this constraint is to allow for independent evolution of system components [Fie00].

This in turn enables the system as a whole to cope with the anarchy characterized by the internet-scale property (which it is to support). Consequently, the client-server constraint does not imply at all that a client must know about the server interface before runtime. In addition, the server should not provide resources which are tailored towards a single client.

This is why I introduced the concept and example of process-driven applications and views in the last chapter. Enterprise applications appear to have a tendency of defining resources in way that suits the required information per process step. Respectively, you can argue that this approach violates the client-server constraint because the needs of the client (where the information is displayed) actually dictates how resources are build.

Provided that you could assume many clients to interact with the API in the same fashion, it would not be a problem because resources should actually be defined in a way that supports clients interacting with the API.

However, enterprise applications - especially the ones not being publicly accessible - often seem to be designed with a single client in mind. Altogether, the resource design thus is driven by visualization concerns of a single client which has a high chance of causing the definition of unstable resources (and thus identifiers): Every time something in the user interface changes, the underlying resources are affected as well. Consequently, the server effectively does not have control over its data.

3.3.2. Stateless

With regard to the previously defined key architectural properties of the Web, the stateless constraint can be considered as a consequence of the internet-scale property: As a server could never keep track of what its clients are doing, the clients have to provide all required context as part of each request-response interaction.

Even though the constraint is normally only referred to as “stateless”, it primarily restricts the communication between client and server which in turn affects the state of said components:

> Communication must be stateless in nature [...]. Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is entirely kept on the client [Fie00]

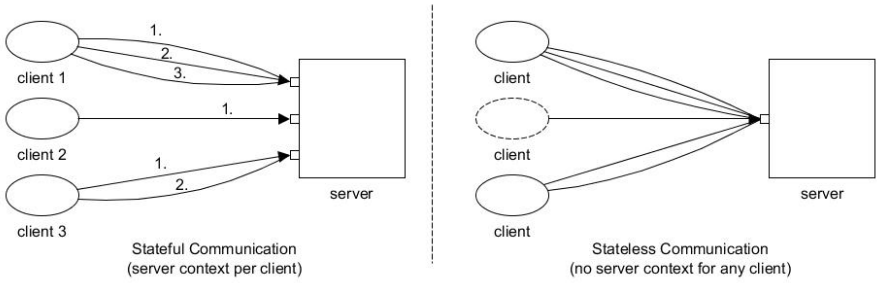

Figure 3.5.: Schematic illustration of stateful and stateless communication between client and server

Based on this quote, the reader might wonder why section 3.2.4 only mentioned resource and application state, but disregarded session state. This is because the definition of application state is much wider than the one of session state: While session state usually gets destroyed after the end of a session, a client may persist parts of its application state for a subsequent session. As a result, the distinction of session state and application state does not yield any real value which is why it usually seems to be dropped. The session state is simply a part of the application state.

As mentioned in section 3.2.4, one important outcome of this constraint is the distinction of resource and application state. Apart from the scalability, the stateless constraint also eases recovery on the server-side because there is no volatile state involved. It thus benefits the simplicity and reliability of the server [Fie00]. Beyond that, the constraint is usually relevant due to being violated. One of the most widespread examples is HTTP *cookies*.

HTTP Cookies

In HTTP, cookies are often used to let the server track actions (which is a concern in its own) or store non-server-persisted context information such as configuration choices of each client [Fie00]. This is achieved by using the respective Set-Cookie response header to provide an opaque string token (the cookie) to the client which is to (re-) transmitted on every request.

On the one hand, cookies thus violates REST’s idea of the client being solely responsible for managing the application state [Fie00]. As a result, they often lead to confusion if the client changes the application state in a way that does not include the server, e.g. by using a *back*-button or history feature. In such a case, the state transmitted by the cookie no longer matches the actual application state of the client. Consequently,

the server would either have trouble dealing with the request or send an unexpected response to the client, both of which tends to impede a successful interaction.

On the other hand, cookies are usually just an opaque string. This does not only contradict the hypermedia strategy - and especially the self-descriptive messages constraint - because a client is unable to figure out its semantics (it could only guess). It also makes cookies a concern with regard to security and privacy [Fie00].

3.3.3. Caching

The goal of the cache constraint is to "partially or completely eliminate some interactions" [Fie00]. This is to balance the efficiency loss in consequence of the stateless and uniform interface constraint. Correspondingly, caching is no early optimization in REST based systems, rather it is an essential part of their design.

Specifically, caching helps to reduce bandwidth consumption, latency as well as load on the server and can even hide network or origin server failures [WPR10]. Most of these benefits are realized by placing components acting as shared caches between clients and servers.

Whenever a cache is capable of answering the request, the latter never reaches the origin server. This is also why polling actually makes sense in the *Web*: The more clients constantly ask for the same resource, the faster in-between caches are filled with the respective responses which counters the bandwidth limitation you would otherwise run into [WPR10].

In addition, clients can further maintain their own, non-shared cache which, in an ideal case, helps to totally avoid a request because the clients knows based on a prior request that there are no changes (this is referred to as freshness [FNR14a]).

> The most efficient network request is one that doesn't use the network [FT02])

For that to work, representations sent by a server must be "implicitly or explicitly labeled as cacheable or non-cacheable" [Fie00] by either including cache information or stating that the content is non-cacheable. In addition, requests sent by a client are to be made conditional by including the last-received cache information of the resource in question.

Conditional retrieval (read) requests thereby allow a server or cache to skip retransmitting the same representation (the process of figuring this is out is called validation [FNR14a]). Conditional other (write) requests enable a client to ensure that the involved resource state did not change since the client last obtained a representation.

Caching in practice

Though caching may initially sound relatively simple in theory, it is not that easy in practice. With regard to HTTP, it is actually one the most complex parts [RAR13].

One the one hand, this is because protocols such as HTTP provide a variety of features to control caching behavior in a fine-granular manner (see table 3.2 for a brief and thus incomplete overview).

One the other hand, implementing the correct handling of these headers for things other than files requires some effort: While today's *Web* servers handle caching of files automatically, handling of script-based generated content must be provided for by the

Header	Purpose
	Server
`Last-Modified`	Timestamp indicating date and time the representation was last modified (one-second resolution)
`ETag`	Opaque validator distinguishing multiple representations (can be weak or strong)
Expires	Timestamp indicating date and time until a representation can be used without contacting the server (one-second resolution)
`Cache-Control`	Directives for caches along the request/ response line
	Client
`If-Unmodified-Since`	The request is only fulfilled if the server timestamp is earlier or equal to the one provided in the request
`If-Modified-Since`	`GET` or `HEAD` request is fulfilled without sending a representation if the server timestamp is not more recent than the one provided in the request
`If-Match`	The request is only fulfilled if the server has a representation having the `ETag` provided in the request
`If-None-Match`	The request is only fulfilled if the server has no representation having the `ETag` provided in the request

Table 3.2.: Brief overview of HTTP's cache headers based on RFC7232 & RFC7234 [FNR14a]

implementer. Supposedly, this is why it is rarely done especially in non-public APIs. It may simply be perceived as extra effort that does not pay off for quite some time.

Cacheable vs Non-Cacheable

> By default, the response to a retrieval (read) request is cacheable while the responses to other (write) requests are non-cacheable [Fie00].

Respectively, RFC7231 [FNR14b] defines that responses to HTTP `HEAD` and `GET` requests are cacheable while responses to other requests - for the most part - are non-cacheable. Responses to `DELETE` requests are non-cacheable because they communicate the intent to remove all current representation (hence there is nothing to be cached anymore).

An exception is the response to a HTTP `POST` requests may be cached if they contain freshness information though many cache implementations in practice just ignore these requests entirely. In addition, while the response to a HTTP `PUT` request itself must not be cached, the server can still send validation information as part of the response to that request in case the representation sent by the client is not transformed by the server in any way (which effectively means that the client can consider the representation it just send as cached).

Beyond that general distinction, there are a few scenarios in which even caching of retrieved responses may not be beneficial. Here are a few examples based on [WPR10]:

- Resources being restricted to one user
- High-frequency “fire and forget” interaction
- The server needs to be aware of every of the clients requests

This first case is fairly simple: There is no real benefit in terms of scalability from caching representation of a resource that are only accessed by a single client. In second case, you can assume that clients are not interested in conditional write requests (e.g. a logging resource). Even if you would realize it, clients would probably never be able to have a non-stale state of the resource due to the constant interaction.

By contrast the third case is more interesting. Even if a server has to be aware of every interaction with a specific resource, the resource itself should not be made non-cacheable. Instead, the resource should reference another, tiny resource that clients are to resolve when accessing the former one. By making this second resource non-cacheable, a server can still keep track of the interaction without suffering scalability drawbacks [Not13].

Things to avoid in terms of caching

Apart from that, there are a few things which may make it hard to cache content provided by an API which is why they should be avoided from a mere caching perspective.

Encryption First of all, encrypting messages effectively prohibits any cache between client and server because the transferred representations cannot be inspected by intermediaries [WPR10]. Clients themselves as well as nodes within the trusted network of the origin server can still make use of it though being much less effective compared to publicly available shared caches. So from the point of view of REST, you should provide unencrypted content where reasonably possible and avoid combining content of both kinds in the same representation (e.g. static images) [Not13].

Authentication In addition, authentication protected content must be explicitly labeled as public within API because by default it must not be cached (this is one of the applications of the Cache-Control directives [FNR14a]). With regard to enterprise applications, this is a common pitfall because they feature early authentication as mentioned in the last chapter. Just as with encryption, resources thus should not combine anonymously accessible content and such that requires authentication (or authorization). In many cases, it also makes more sense to split both types of information by way of resources.

Resource Design Furthermore, the benefit of caching is influenced by the actual resource and representation design. Typically, the *Web* features large-grain representation such as entire *Web* sites for which REST has been optimized [Fie00]. Consequently, designing resources in a way that causes many resources to be changed at the same time is likely to make caching ineffective because it significantly lowers the chance that nothing changed upon a subsequent request. So, it should be avoided if possible.

Representation Design Likewise, having too many, fine-granular representations of the same resource is counter-productive due to the limited capacity of caches and the potentially lower chance that clients actually request the same representation. However, this is something enterprise applications seemingly often tend to do in consequence of the process driven approach. With regard to application presented in chapter 1, a *speaker* might for instance receive all attributes of a *questionnaire*, while a *voter* might not receive the *analysisDelay* attribute because it is not rendered in the respective view. If there are good reasons to retain this information from voters (e.g. privacy related), it is better to separate the information into different resources. Otherwise, such fine-granular distinctions should be avoided.

Embedding Embedding describes the idea of exposing a resource that combines the information of multiple, related resources (for instance a questionnaire including its questions). What is interesting about embedding is that it is often only used for obtaining representations from the server. When it comes to manipulating things, clients usually interact with the resource underlying the part which is to be manipulated. Hence you could argue that this approach is somewhat redundant.

From the perspective of REST, embedding is probably the most inefficient design choice with regard to caching: It would be more efficient to make separate requests only for those representations not already cached [Fie00]. This is because a resource featuring embedding always changes whenever a part of it does which is why clients have to re-retrieve a representation.

Especially when only using the resource featuring embedding for read purposes, you frequently reload content of which the bigger part did not change which could be avoided. In addition, you could also argue that not embedding makes an API easier to understand because it leads to a better separation of concerns in terms of resources. This is because messages would only contain one representation of a single resource instead of multiple ones.

Due to performance reasons as well as probably the process orientation discussed in section 2.1, embedding is however quite common within the *Web* and especially business applications. Often, it is discussed in context of the so called “chattiness” of REST APIs. But actually, most of the claimed performance issues are a result of de-facto standard protocol HTTP instead of REST itself.

HTTP 1.1 vs. HTTP 2.0 The design HTTP 1.1 has some flaws with regard to how the *Web* is used nowadays (see [Not] for details).

First, there is the head of line blocking problem: In HTTP 1.1, requests must be fully processed in the order they arrive at the server - even with pipelining . As TCP connections start with small transmission rates, an initial, big request will thus delay (block) all following ones.

Second, it is often tried to compensate this problem by using multiple TCP connections. Yet, this does not only make it very difficult for clients to use them effectively (which request via which connection), it also leads to network congestion events because all connection together exceed the available bandwidth of the client.

As a result, the transmission window of all underlying TCP connection is shrinked which considerably increases the overall transmission time. Third, headers in HTTP 1.1 are very verbose. This is especially unpleasant in case of requests because they often do not convey any payload; You mostly transmit headers around. Due to all that,

a common goal in context of HTTP 1.1 is to reduce requests by various tricks such as *spriting*, *sharding* or *concatenating* which effectively is embedding.

HTTP 2.0 promises to address these problems: “Making 20 HTTP requests, and getting 20 responses, will be almost as fast as making one big request and getting one big response” [RAR13].

As a consequence, HTTP 2.0 will allow for a much cleaner design which is why respective decisions discussed later in this thesis will assume its availability. In addition, making more fine-granular requests should no longer be that inefficient.

3.3.4. Uniform Interface

The idea of a uniform interface is to employ a single, common interface for all component interaction instead of defining and using a dedicated one for each context or application. In short, it favors interface generalization over interface specialization. This is what distinguishes REST the most from other interaction styles such as RPC.

In section 3.1.2, I stated that the forth key architectural property, distributed hypermedia, resolves the requirements introduced by the first three properties while introducing some requirements of its own (basically, to implement the hypermedia strategy). Correspondingly, REST’s uniform interface describes how to design the component interaction of a system that is to achieve these properties by using the hypermedia strategy. It constitutes a design trade-off that prioritizes extensibility and internet-scale at the price of efficiency and favors large-grain data transfer (as brought up in context of cashing).

> The trade-off, though, is that a uniform interface degrades efficiency, since information is transmitted in standardized form rather than one which is specific to application’s needs of one being tailored to the individual application. The REST interface is designed to be efficient for large-grain hypermedia data transfer, optimizing for the common case of the *Web*, but resulting in an interface that is not optimal for other forms of interaction [Fie00].

Specifically, REST’s uniform interface is characterized by four constraints which will be covered in the following.

Resource Identification

For the most part, this constraint has been covered in context of resource identifiers, resources and representations:

> The definition of resource in REST is based on simple premise: Resource identifiers should change as infrequently as possible [Fie00].

REST uses identifiers to identify resources, that is stable semantics, instead of representations which is why this constraint is also referred to as *addressability*. The distinction allows resources and their identifiers to remain stable over time while the representations corresponding to a resource may change as needed. If a server nonetheless needs to change resource identifiers, it should not simply delete the old ones but instead forward clients to the new identifiers [RAR13].

Manipulation Through Representations

Section 3.2.3 stated that clients never see resources directly; They are only an indirection. Instead, client and server exchange representations corresponding to the state of the underlying the resource. Depending on the intent sent along with the representation, it constitutes either the current or intended one.

This constraint is the reason for this interaction model. It characterizes how components communicate through the uniform interface.

> The protocols are specific about the intent of an application action, but the mechanism behind the interface must decide how that intention affects the underlying implementation of the resource mapping to representations [Fie00].

In this regard, it is important to note that REST does not restrict the general vocabulary used to specify the intent (e.g. HTTP methods or status codes). Just as a REST API is generally not limited to the HTTP protocol, it is also not limited to *create*, *read*, *update* and *delete* (CRUD) operations realized as `POST`, `GET`, `PUT` and `DELETE` requests [WP11]. `POST` is not even a good match for *create* because the former has no specific semantics - it can do anything [RAR13].

Contrariwise, the uniform interface constraint only requires that there is a "general and functionally sufficient set of methods" [WP11] as well as other required types of intents. How many and which there are is subject to standardization. [3]

Consequently, a REST API implemented with HTTP may just as well employ other methods such as PATCH [DLS10] or `LINK` and `UNLINK` [Sne14] even though, in practice, there are often technical hindrances: "If you need to talk to a variety of caches and proxies, you should stay away from `PATCH` and other methods not defined in RFC 7231" [RAR13][4]. A few years back, firewalls apparently even blocked any HTTP method other than GET or POST [Hyu+09].

Self-descriptive Messages

The *self-descriptive messages* constraint requires that any component of a system can understand messages it receives based only on the content of the individual message. In other words, "there's no free-floating documentation nearby that client are also expected to [know of or] understand" [RAR13].

> The uniform interface must provide a way in which information exchanges can "label" representations, so that no out-of-band information or prior agreement is necessary to "understand" a representation that is received [WP11].

This chapter already covered various examples of the constraint:

- Resource identifiers by themselves have no special meaning. However a representation may explain what to expect of the underlying resource (section 3.2.1)

[3] This is why there is no partial GET as of now - there is now according standard [RAR13].

[4] The quote originally referenced RFC2616 which has been obsoleted by RFC7230 till RFC7235. The HTTP methods are now covered in RFC7231

- Representations have to contain a resource identifier linking to the underlying resource (section 3.2.3)
- As the client solely manages its application state, each request to a server has to contain all context information required to properly understand the request (section 3.2.4
- Responses have to be explicitly labeled as cacheable or non-cacheable (section 3.3.3)
- Components of a REST based system have to employ standard vocabulary to state the intend (and thus the semantics) of exchanged representations (section 3.3.4)

Apart from these examples, probably the most important concept with regard to self-descriptiveness are media types which I briefly mentioned in section 3.2.3 and will cover in depth in section 4.3.

Basically, media types identify the format of a representation. Knowing the format allows a component of a REST based system to parse a respective representation and to understand parts of its semantics (i.e. what it means) [RAR13].

From the perspective of an implementer, the *self-descriptive messages* constraint is the most challenging and important one because, in case of machine-driven interaction, it requires bridging the semantic gap.

Provided that all exchanged messages are truly self-descriptive, a client has to be able to understand the full meaning of a message as well as comprehend the available possibilities of what to do next. As mentioned before, the next chapter will cover this topic in depth.

Hypermedia as the Engine of Application State

This constraint is likely the most notorious one of REST. Given that it does not have the catchiest name, the constraint has received other names over time. Often, it is just referred to as the *hypermedia constraint*, or simply *connectedness* [RAR13].

The constraint specifies how components of a REST based system are to interact: Clients build up and adapt their application state based on representations they receive from one or multiple servers over time.

Due to self-descriptiveness (i.e. the hypermedia strategy), each obtained representation offers the client choices on what to do next. It can manipulate resource state on a server or adapt its application state by obtaining a representation of another, related resource. Effectively, hypermedia thus drives the application state of the client.

Interestingly enough, the hypermedia constraint is nothing you have to adhere to directly. Rather, it is a reward for adhering the other REST constraints.

> The hypermedia constraint is not a chore you must perform to be "RESTFUL". It's the payoff for obeying the other constraints. It gives you extensibility. The hypermedia constraint allows a smart client to automatically adapt to changes on the server side. It allows a server to change its underlying implementation without breaking all of its clients [RAR13].

3.3.5. Layered System

Generally, the layered system constraint limits accessing components to only "see" the top layer of the accessed component [Fie00]. This often makes interaction less efficient because the accessing component cannot make use of implementation details. In return, it however enables the accessed component to retain extensibility because it is able to change lower layers as needed without externally visible effects.

In addition, the combination of uniform interface and layered system constraint has one important effect: It allows for deploying intermediary components between client and server which act and thus affect the communication in a *transparent* way [RAR13]. This is possible because all components of a REST based system hide their implementation details and share the same, uniform interface. In practice, there are mostly two types of intermediary components: Caches and proxies (see figure 3.6).

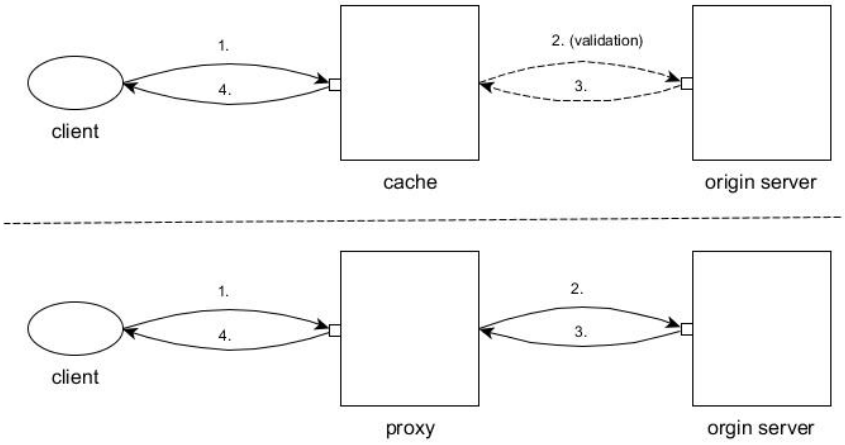

Figure 3.6.: Schematic illustration of communication via a cache or proxy component

Caches

As mentioned in section 3.3.3, a cache stores representations sent through a network. If a request gets routed through the cache, it may be able to respond to the request instead of the origin server provided that it actually has, or can ensure to have, a current response (representation). The latter is done by sending a respective validation request to the origin server whose response is usually much smaller than the one to the request of the client [FNR14a].

Overall, caches thus greatly help in reducing network traffic by moving content the closer to the requesting clients the more it is requested [WPR10]. However, this is also why implementing caches is not exactly trivial.

Proxies

Instead of handling a request received from a component (client or other proxy), a proxy re-routes the request to another component (server or other proxy) returns the

received response to the requesting component. Depending on its purpose, it may also manipulate the request or response in some other way [RAR13].

Proxies are employed for various purposes: To re-route requests to internal servers (including load-balancing), strip or add encryption, compress or decompress the message or to even translate and forward requests using another protocol [RAR13].

3.3.6. Code On Demand

The code-on-demand constraint is the only optional one of the REST constraints. It allows clients to obtain code from the server which represents know-how on how to process specific resources [Fie00].

The typical example for this case is combination of HTML pages and *Javascript* files. Apart from that, the code-on-demand constraint does not have that much relevance because security concerns usually oppose the idea of automatically downloading and executing some code from a server (which is why *Javascript* runs in a sandbox). Hence, (pure) *Web* APIs do not really use it [RAR13].

Nevertheless, there is one important misconception about the code-on-demand in context *rich-clients*: The fact that scripts implementing the client were obtained by code-on-demand does not mean that the communication between the rich-client and its backing server(s) does not need to employ hypermedia.

Rather, the rich-client *application* should be considered as a machine-client of the backing-servers - one among many. If this client is hard-coded towards the servers, others inevitably are as well. As a result, the server looses the ability to change without breaking everything not covered by code-on-demand scripts: You loose extensibility while still being required to lock-upgrade the client scripts along with the server.

3.4. Common misconceptions

Before ending this chapter, this section covers two topics which seem to repeatedly show up in context of REST: Version management and pragmatic REST.

3.4.1. Version Management

Every API needs to be changed provided that it has been running long enough. But if the API and clients are not capable of hypermedia there is often a problem. You cannot change the API without breaking its clients.

> This is why public APIs change so rarely. You can’t change an API based on API calls without causing huge pain among your users, any more than you can change the API of a local code library without causing pain [RAR13]

Often, it is tried to work around this issue by versioning the API by way of URLs patterns or prefixes. Thereby, clients are required to know the version they should use. With regard the application discussed in chapter 1, versioned URLs of a specific questionnaire could look like this:

- `/api/v1/questionnaires/1`

- `/api/v2/questionnaires/2`

For implementers, such an approach is only helpful if they know the implications of *v1* or *v2* as well as which one is to be used preferably. Usually, this information is provided outside the scope of the API - in other words, *off-the-wire* - which violates the *self-descriptiveness* constraint of REST. If a client were to understand that both URLs identify different versions of the same resource, the server would be required to provide a respective representations explaining just that. Otherwise, the resources underlying both identifiers would be considered to be completely unrelated.

Apart from that, you can also argue that this approach violates the resource identification constraint [RSK12]. This is because in many cases, the relevant semantics of both versions and thus the resource stay the same. What actually changed were its available representations. If that is not the case, there is probably an issue regarding the definition of the resource which should be addressed as explained in section 3.2.1.

Despite that, you would probably have difficulties to comprehensively explain the differences between multiple version. Often, the requirements simply changed without any special reasoning that would be relevant and meaningful to the client.

In addition, clients are probably rarely interested in the fact if and how a resource would be versioned. The only thing that is interesting to them is to figure out which variant best suits their need. Given that REST already provides the concept of representations allowing for clients to obtain different state descriptions of a resource, it makes sense to employ it for the versioning use case as well.

Provided that you would get a client as far as being able to figure out the preferred choice out of a fully-detailed or preview representation, the same should work for multiple “versions” of that resource. Though in this case, the fact that representations are versioned becomes an implementation detail and thus no longer matters to clients. As an additional benefit, this approach also saves you from versioning the entire API even if its bigger part did not change.

3.4.2. Pragmatic REST

As stated in section 3.1.2, the hypermedia strategy actually raises the entry-barrier in case of machine-to-machine communication. Apart from the fact that hypermedia generally is the least understood aspect of REST [RAR13], this is probably one of the main reasons why enterprise application try to work around hypermedia.

The resulting style is often coined by the term *pragmatic REST* and usually involves one of the following two approaches [5].

Drop Internet-Scale

If you are able to force-upgrade all clients at once, you can skip the internet-scale architectural property required by REST. As a result, you no longer need hypermedia while being able to provide a low entry-barrier as well as extensibility:

Implementers can read the documentation and apply changes when and however they see fit. This approach is usually viable for internal, isolated applications that are fully controlled by an organizational unit.

[5] Both are taken from [RAR13]

Drop Extensibility & Hypermedia

As a second approach, you can skip extensibility and hypermedia to allow for a low-entry barrier and internet-scale.

As a consequence, changes to the respective API will almost always affect clients in some way. As non-internal APIs tend to require internet-scalability, this represents the most common approach for public *Web* APIs.

3.4.3. Why not to call it REST

Calling one of these approaches *pragmatic REST* causes one crucial problem: Its suggests that it is just some style of REST while in reality it is something utterly different because it ignores the very key concept:

> REST APIs must be hypermedia driven [Fie08].

In this regard, the problem not necessarily is using a name for something that does not match it. Rather, the problem is that many issues perceived in context of self-proclaimed REST APIs actually seem to be a result of "cherry-picking" these constraints of REST that seem convenient. As a consequence, you end up with questions such as:

- How do you design URLs with REST?
- Which HTTP methods are part of REST?
- Why are REST APIs so chatty?

As to my experiences during the preparation and work on this thesis, you can only really understand the implications of *pragmatic REST* when having understood REST itself. Using the term *pragmatic REST* just seems to cause more confusion and unnecessary problems on its own.

In lack of a better alternative, it might even be better to name the employed style based on the respective protocol (e.g. HTTP API) as that would help people being unfamiliar with REST finding respective documentation.

In the end and as mentioned in section 3.1.3, the question of whether or not REST poses a suitable, sufficiently powerful solution for a specific context depends on how closely the target environment matches the one of the *Web*. REST is not and never was about the one true architecture [Fie08]

4. Implementing the hypermedia strategy

Given that last chapter excluded implementing hypermedia and *self-descriptive messages*, this one covers these two topics in detail.

First, the chapter describes the general structure and semantic types found in hypermedia representations. After that, it explains *media types* and how these influence structure and semantics of a representation. Given that media types do not cover both aspects completely, the last part of the chapter discusses ways to provide human and machine-readable documentation for that purpose.

4.1. Elements of a hypermedia representation

As explained in section 3.1.2, hypermedia is about simultaneously presenting information (structured as data) and controls. Consequently, you can split a hypermedia representation into *data* and *control* elements which are normally referred to as hypermedia controls.

To a great extend, this distinction is introduced because available documentation often seems to mostly focus on hypermedia controls. Presumably, that is because they are what distinguishes hypermedia from other interaction styles. As evident in the following, control elements usually also require more descriptive effort compared to data elements.

Nonetheless, discerning both element types is very useful because it allows for a much more structured approach concerning how to realize self-descriptive messages which is covered later in this chapter.

4.1.1. Data Elements

Data elements convey that kind of information which you would also send along the network in other interaction styles such as RPC. Clients usually retrieve them for some sort of processing (e.g. to be rendered for users) or subsequently use them to describe a change in resource state submitted as part of a request. Often, data elements are hierarchically organized and represented as key-value pairs.

As an example, figure 4.1 shows a hypermedia representation which is a slightly modified variant of the representation displayed in figure 2.6 and is based on the media type *Siren* [Swi14]. Altogether, it contains four data elements: *id*, *analysisDelay*, *identifier* and *title*.

As defined by the format, values of the *class* attribute describe the nature of the underlying resource (which *Siren* calls entity) based on the representation they are found in. Thus, it constitutes what you could call *meta data* and does not qualify as a data element.

```
1  {
2    "class": ["questionnaire"],
3    "properties": {
4      "id": 1,
5      "analysisDelay": "PT5S",
6      "identifier": "SE101SS14",
7      "title": "Lecture evaluation: Software Engineering 101 (SS14)"
8    }
9    // other representation elements
10 }
```

Figure 4.1.: Data elements based on the *Siren* hypermedia format

Depending on employed format and resource design, data elements may also form more complex hierarchies to better express the semantics of the underlying resource. For example, the *analysisDelay* could also be expressed as a combination of a time-unit and a duration.

However, embedding overly complex data elements within others is usually considered to be a resource design smell. This is because it is often better to encapsulate the embedded, complex data elements in a separate resource as discussed in section 3.3.3).

4.1.2. Control elements

As stated in section 3.1.2, the idea of hypermedia is to send both, data and interaction instructions, to the client. Respectively, control elements constitute and organize these instructions within a hypermedia representation. Due to the fact that REST components exchange resource representations to interact, a control element always contains at least one resource identifier which is usually identified by way of a *href* attribute or property.

Based on the states concept described in section 3.2.4, you can characterize a control element as a description of a state transition, that is a mutually non-exclusive change of application or resource state [RAR13]. Depending on the expressiveness of the employed media type, a control element can either describe a single or a family of transitions, for instance by using an URI-template.

Some transitions, for example in case of HTML's `<img>` and `<script>` tags, are supposed to be triggered automatically while others fall to the choice of the client. As an example of the latter, figure 4.2 shows the remaining part of the representation displayed in figure 4.1 which contains three control elements.

The first control element, the *append-question* action, describes how a client may append, that is to create and add, a new question to the *questionnaire* resource underlying the representation. It thus affects the resource and possibly also the application state. The latter thereby mostly depends on the response of the server as well as whether or not or how the clients incorporates it.

By contrast, the other two control elements (line 17 and 18) both solely describe a possible change of application state. As defined by the *Siren* media type, a client may retrieve a representation of the respectively identified resource by making a `GET`

```
{
  // other representation elements
  "actions": [
    {
      "name": "append-question",
      "title": "Appends a question to this questionnaire",
      "method": "POST",
      "href": "http://example.com/questionnaires/1",
      "type": "application/x-www-form-urlencoded",
      "fields": [
        { "name": "content", "type": "text" }
      ]
    }
    // other actions
  ],
  "links": [
    { "rel": [ "self" ], "href": "http://example.com/questionnaires/1" }
      ,
    { "rel": [ "item" ], "href": "http://example.com/questions/25" }
    // other links
  ]
}
```

Figure 4.2.: Control elements based on the *Siren* [Swi14] hypermedia format

request.

Thereby, the first control element identifies the resource underlying the representation (self). The second control element identifies a resource that is characterized as an item belonging to the first mentioned resource. Due to its URL, a human user might correctly anticipate that it in fact identifies a resource representing a *question* belonging to the respective *questionnaire*.

Control elements may be manipulated

What is important to understand about control elements is that a client may interact with them the same way as with data elements.

In the example shown in figure 4.2, a client would probably make a request based on the *append-question* action to append a question to the underlying questionnaire resource. Subsequently, the representation would probably contain an additional control element to access the newly created resource.

In another scenario, a client might just as well create the question by other means and interact more directly with the control elements of the representation. For example, a HTTP server might allow the client to use the `LINK` method [Sne14] to "manually" add the question to the underlying questionnaire. Naturally, this in turn would require a control element describing the corresponding request.

In many situations, a client would probably not be able to change control elements such as the *action* one shown in figure 4.2. This is because changing things that in-

volve behavior on the server is usually much more implementation dependent and thus cannot be handled that arbitrarily. Nonetheless, it is generally possible.

Categorizing control elements

Beyond that, controls elements are often further categorized in some way to emphasize certain characteristics or to simplify their description. So far, two variants have already been covered: Based on their effect on application or resource state as well as based on groups defined by the employed media type.

In case of *Siren*, control elements which are purely navigational are grouped as links: These describe transitions which only affect the application state and do not involve any special behavior at the server. Anything not matching these criteria is grouped as an action control.

In this regard, other media types such as *Collection+JSON* even define three categories: *links*, *queries* & *(append-) templates* [RAR13]). By contrast, others such as the *Hypertext Application Language (HAL)* [Swi14] define none at all.

Whether or not to group control elements in most cases seems to be a trade-off of enhanced readability (for humans) at the cost of backward-incompatibility in case one of the groupings should prove to be inconsistent at some point in time.

As a third possibility, transitions - and thus the respective control elements - may also be categorized based on their general execution semantics. The therefore employed properties *safe*, *idempotent* and *unsafe* are probably best known from the context of the HTTP methods (see [FNR14b]).

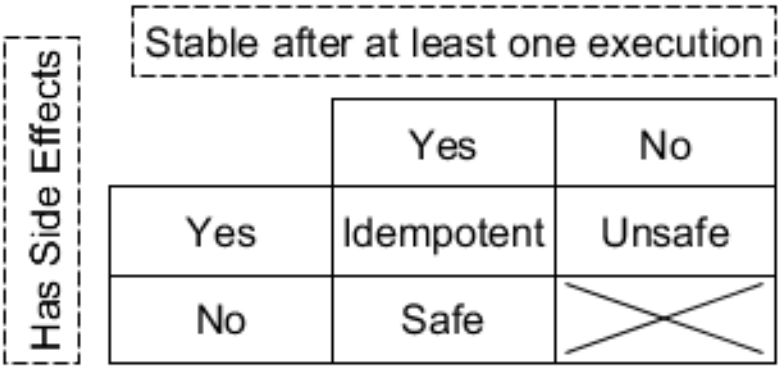

Figure 4.3.: Categorization of state transition types

In this regard, a state transition is considered *safe* if it only affects application state (e.g. `GET` or `HEAD`). It does not have any side effects concerning resource state except for things that are fully internally handled such as access counts or logging. Consequently, the *link* control elements shown in figure 4.2 describes safe state transitions.

> Triggering a safe state transition should have the same effect on resource state as doing nothing at all [RAR13].

Beyond that, a state transition is classified as *idempotent* if the change of resource state (i.e. its side effects) stays the same regardless of the transition being triggered exactly once or more than once (e.g. `PUT` or `DELETE`). Correspondingly, *safe* state transitions are always *idempotent*.

Due to this correlation, the term *idempotent* is often used to only identify *idempotent* state transition that have side effects (e.g. in context of [ARF14]). A respective categorization of state transitions is shown in figure 4.3.

Last, *unsafe* state transitions are such that have side effects **and** feature a different outcome every time they are executed. In other words, they are neither *safe* nor *idempotent* (e.g. `POST`). A good example for such a case is the *append-question* action displayed in figure 4.2.

4.2. Semantic types of a hypermedia representation

Besides element types, a hypermedia representation can be discerned in terms of semantics, namely *protocol* and *application semantics* - a distinction that has been suggested by [RAR13].

While not being standardized at this time, the semantic types still offer a second, very helpful view point to understand how hypermedia representations are structured.

4.2.1. Protocol Semantics

Protocol semantics comprise what you can describe, understand and thus do - based on the employed protocol.

Thereby, it is essential for components of a REST based systems to agree on these semantics ahead of time: Otherwise, they would be incapable of communicating and interacting with each other [RAR13]. Without prior agreement on protocol semantics, a client would not even know how to access a server via its well-known entry URL.

Beyond that, the concept protocol semantics is most useful in two contexts: Control elements and media types. As protocols revolve around rules of communication, protocol semantics are usually irrelevant in context of data elements.

Control elements

In context of the control elements, protocol semantics explain how to trigger a single or a family of state transitions (protocol-wise). Thereby, the latter is commonly realized by describing URL templates.

With regard to the control elements displayed in figure 4.2, all control elements grouped by the *links* property share the same protocols semantics: A client can make a `GET` request to the resource identified by the value of the *href* property to obtain a representation.

In case of the *append-question* action, its protocol semantics state that a client can make a `POST` request to the identified resource as well as that it should use the media type named by the value of the *type* property. As HTTP is incapable of describing a payload beyond its intent, the payload specified by the content of the *fields* property is not part of the protocol semantics of the control element.

In contrast to these examples, control elements can be very verbose concerning protocol semantics. In case of HTTP, a control element could also describe other headers to be used (aside from the media type) or which status code to expect as part of the response. Nevertheless, it seems that - by convention - most control elements only state the minimally required features and leave remaining choices to the client (e.g. the usage of *cache* headers) [RAR13].

Media types and representations

In context of media types, the concept of protocol semantics can be used to describe the subset of protocol features which is supported by the media type in question.

For instance, the protocol semantics of HTML only allow for HTTP `GET` and `POST` requests. The format does not provide a way for control elements to describe `PUT` or `DELETE` requests [RAR13]. By contrast, the media type *Siren* allows its control elements grouped by the *actions* property to not only specify these four methods, but also the `PATCH` [DLS10] method.

As another example, the protocol semantics of the media type *HAL* are limited in another way. While it generally supports any HTTP method, the format itself is incapable of specifying the method to be used (see figure 4.4). *HAL* expects this information to be provided by another source of documentation [Swi14]. This could for instance be a *profile* which will be covered in section 4.4.2.

```
1 <resource href="http://example.com/questionnaires/1">
2
3   <link name="append" title="Append a question to this questionnaire"
        href="http://example.com/questionnaires"/>
4
5   <link rel="self" href="http://example.com/questionnaires/1"/>
6   <link rel="item" href="http://example.com/questions/25"/>
7
8 </resource>
```

Figure 4.4.: Example representation using the *XML* variant of the *HAL* [Swi14] hypermedia type

Expressiveness of protocol semantics

As evident from these examples, the expressiveness (and conciseness) of protocol semantics depends on the involved protocol or its feature. For example, the protocol semantics of a HTTP `DELETE` request are relatively straight-forward: It deletes the server-side mapping of the involved identifier to the respective resource meaning that subsequent `GET` requests to the identifier will not yield a representation [FNR14b]. By contrast, the protocol semantics of a HTTP `POST` request are completely arbitrary: It basically means “whatever” [RAR13].

So to fully understand the contents of a hypermedia representation (e.g. in case two control elements are indistinguishable based on their protocol semantics), a client or human user also needs to understand the second type of semantics.

4.2.2. Application Semantics

The purpose of application semantics is similar to the one of protocol semantics: Description, understanding and subsequently, action. But in contrast to protocol semantics, application semantics, more precisely their vocabulary, is based on real world terms instead of a protocol.

In addition, application semantics do not necessarily need to be and often are not known ahead of time. A machine client or human user might be able to figure them out at runtime or - e.g. in case of the former being a Web browser - might not be required to deal with them.

Figures 4.1 and 4.2 already displayed multiple examples of application semantics: The value of the *class* attribute, the *title* explaining the purpose of the *append-question* action, the *fields* property explaining the structure and content of the payload to be sent along to the server, or the value of *rel* property as part of the *links* control elements; All that describes application semantics.

What further distinguishes application from protocol semantics is that they can usually be found in context of data as well as control elements. By contrast, protocol semantics are - for the most part - only relevant with regard to control elements. In this regard, application semantics further help to distinguish control elements being equal in terms of protocol semantics as demonstrated by the *links* control elements shown in figure 4.2.

> A representation's application semantics explain the underlying resource in terms of real-world concepts. Two HTML documents may use exactly the same tags but have completely different application semantics - one of them describes a person, and the other describes a medical procedure [RAR13].

Abstraction Level

While the concept of application state primarily comprises something that only concerns the client, the term *application* has a different meaning with regard to application semantics.

In this case, it rather refers to the abstraction level of these semantics. Similar as to how the OSI layer abstraction model describe an application layer above the presentation layer [Mic15], application semantics are to be considered as an abstraction above protocol semantics.

Respectively, application semantics of a control element could theoretically be inconsistent with is protocol semantics. As showcased in figure 4.5, you could for instance specify the *append-question* action from figure 4.2 to be triggered by a `GET` request.

Generally, there is no common way to avoid such specifications. Instead, it is left to the party having control over a resource to ensure that its published representations are consistent. Within the Web, this is anyway essential for API providers, as an inconsistent API would most likely not be used by any client.

4.3. Media types

Now that the basics of a hypermedia representation have been covered, this section addresses probably the most important aspect in terms of hypermedia and self descrip-

```
{
  // other representation elements
  "actions": [
    {
      "name": "append-question",
      "title": "Append a question to this questionnaire",
      "method": "GET",
      "href": "http://example.com/questionnaires/1",
      "type": "application/x-www-form-urlencoded",
      "fields": [
        { "name": "content", "type": "text" }
      ]
    }
    // other actions
  ]
  // links
}
```

Figure 4.5.: Control element with inconsistent semantics

tiveness: media types. They are what brings together the four aspects discussed so far in this chapter. In addition, they are **the** starting point in terms of self-descriptiveness.

> To some extent, people get REST wrong because I failed to include enough detail on media type design within my dissertation. That's because I ran out of time, not because I thought it was any less important than the other aspects of REST [Fie08].

In short, a media type defines and thus allows to understand the syntax (i.e. the format) as well as (parts of) the semantics of a representation [RAR13]. Due to their purpose, media types must be known and understood ahead of time just like protocol semantics. Otherwise, a client would be unable to discover the features of an API due to being unable to parse obtained representations.

4.3.1. Aliases and term history

Originally, the concept of a media type was defined in a set of documents called multi-purpose internet mail extensions, in short *MIME*, whose latest version are RFCs 2045 [FB96a], 2046 [FB96b], 2047 [FB96a].

These documents defined an internet mail format header named `Content-Type` which was to describe the nature and format of the data being transferred. The value of this header was initially called *MIME* type but later renamed to *media type* [Pos94]. Along with other concepts defined by the *MIME* documents, the `Content-Type` header along with its *MIME* type value was later re-used by the HTTP protocol.

Given that HTTP is still the most prominently used protocol with regard to REST, this is presumably why the terms *format*, *media type*, *MIME type* and *content type* are often used interchangeably.

4.3.2. Media types and hypermedia types

What distinguishes a hypermedia type from a “normal” media type is that the former comprises dedicate definitions for data **and** control elements [RAR13]. This is why formats such as *HAL*, *HTML* or *Siren* are considered to be hypermedia types. They all come with elements like *links*, *actions* or *forms* to define control elements.

Respectively, media types such as *plain JSON* or *XML*, more precisely `application/json` and `application/xml`, do not qualify as hypermedia types: Their definitions do not allow to distinguish an incidental resource identifier from an intended one [RAR13]. In case of plain *JSON*, this problem is demonstrated in figure 4.6.

```
{
  "links": {
    "self": "http://example.com/questionnaires/1",
    "item": "http://example.com/questions/25"
  }
}
```

Figure 4.6.: Example of an attempt to specify control elements based on `application/json`

Even if the *links* element looks identical to the one defined by *Siren* shown in figure 4.2, it does not convey the same information. This is because *Siren* explicitly defines the content of that element to be control elements. As far as *plain JSON* is concerned, it is just an arbitrary element. There is no way to tell whether or not these links actually identify resources or how they are supposed to be used for hypermedia based interaction.

Despite all that, there are still ways to use (non-hypermedia) media types like *JSON* or *XML* to some extend. As demonstrated by the *Siren* hypermedia type, you can define a new hypermedia type as an augmentation of an existing media type. Among other things, this is indicated by the name of the media type.

4.3.3. Media type naming and structure

In the beginning, media type names were defined based on three concepts: A *(top-level) type*, a *subtype* and optionally, a set of parameters. In consequence of two conventions that arose over time, *subtypes* are now further distinguished in terms of two concepts: *Subtype trees* and *structured syntax name suffixes* [FKH13].

Figure 4.7 shows these different parts of a media type name based on an example. In addition, the following briefly covers each part.

Top-level type

The top-level type is more of a general categorization of the content being transferred. This is why it must be selected from a list of standardized terms which is expected to be extended quite rarely [FB96a].

For example, “the `text` top-level type is intended for sending material that is principally textual in form” [FKH13]. In context of APIs, probably the most relevant top-level

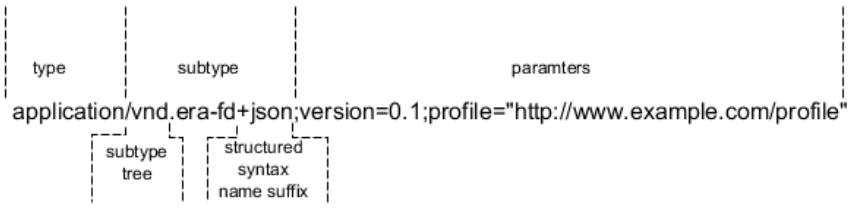

Figure 4.7.: Structure of a media type name

type is `application`: It is reserved for data to be processed by an application program or such that does fit in any category such as `image`, `audio` or `multipart` [FB96b].

Subtype

The *subtype* specifies a particular format of the category described by the *top-level type*. For example:

- `text/plain` - Text that does not have any formatting commands or directives
- `text/html` - Text based on the HTML format
- `application/json` - *JSON* based application content

In contrast to *top-level types*, they are the primary extension point for defining new media types [FB96a]. As mentioned before, the original *subtype* is now conceptually divided into a *subtype tree*, a *subtype name* and a *structured syntax name suffix*.

Subtype trees

Subtype trees further characterize a media type, for instance in terms of purpose or usage context. Generally, they are represented as a prefix of the *subtype* name which is terminated by a dot. An exception to this rule are media types that underwent an internet standardization process such as the ones just shown. These are grouped in the so called *standards tree* which must not have a prefix [FKH13].

As of today, probably the most commonly used *subtype tree* is vendor (`vnd.`). It characterizes media types being used in context of publicly available products [FKH13]. A respective example is the full name of the media type *Siren* which was used in figures 4.1 and 4.2:

- `application/vnd.siren+json`

Alternatively, there are two additional trees: Personal (`prs.`) and unregistered (`x.`). The first one serves for experimental or non-commercially distributed media types. The second one “may be used for types intended exclusively for use in private, local environments” [FKH13].

Structured syntax name suffixes

The idea of a *structured syntax name suffix* is to define a new media type as an augmentation of an existing one. According to [FKH13], the first media type of that kind was defined based on *XML* by RFC3023.

This RFC established the convention of adding the name of the augmented media type as a suffix to the name of the newly created media type using the + character as a delimiter (`+xml` in this case). Over time, this convention has become a de-facto standard for media types in general [FKH13].

As mentioned before, the *Siren* media type is an example of a *JSON* augmentation. On the other hand, the media type *HAL* is based on *XML*:

- `application/vnd.siren+json`
- `application/hal+xml`

4.3.4. Semantic coverage

As mentioned in the beginning of this section, a media type may also allow a client to understand parts of the semantics of a representation. For that purpose, a media type may define application semantics, restrict their combination with protocol semantics or limit protocol features to be used in representations. Thereby, the extent to which each is the case depends on the definition of the individual media type.

Collection+JSON

In order to demonstrate these possibilities, figure 4.8 displays a representation based on the media type *Collection+JSON* [Amu11] which is somewhat equivalent to the combination of the *Siren* representation shown in figures 4.1 and 4.2 as explained in the following.

In terms of application semantics, *Collection+Json* defines every resource to be a collection, that is one which “lists other resources [(its items)] by linking to them” [RAR13]. This why the resource underlying the representation shown in figure 4.8, is actually a collection of questionnaires instead of a single one: Representations based on Collection+JSON cannot specify properties equivalently to, for instance, *Siren*.

In addition, *Collection+JSON* defines several operations which can be performed upon collections and whose availability is indicated by special elements such as the *template* one. Its presence indicates that a client can use the *data* object defined by the *template* element to append a new item to the collection. This is done by sending a `POST` request containing a representation based on said object to the collection resources whose identifier is specified by the top-level *href* attribute. Furthermore, it also allows for updating an existing item by sending a `PUT` request containing a respective representation to resource identified by the *href* attribute of the item in question.

Thereby, the update operation is a good example of restricted protocol features and their combination with application semantics: Theoretically, you could also update an item by sending a `PATCH` request to the collection resource. However, *Collection+JSON* does not support `PATCH` requests in general which is why this combination is not allowed.

```
{
  "collection": {
    "version" : "1.0"
    "href": "http://example.com/questionnaires",

    "items": [
      {
        "href": "http://example.com/questionnaires/1",
        "data": [
          { "name": "analysisDelay", "value": "PT5S" },
          { "name": "identifier", "value": "SE101SS14" },
          { "name": "title", "value": "Lecture evaluation: Software
             Engineering 101 (SS14)" }
        ]
      }
    ],
    "template:" {
      "data": [
        { "name": "analysisDelay", "value": "" },
        { "name": "identifier", "value": "" },
        { "name": "title", "value": "" }
      ]
    }
  }
}
```

Figure 4.8.: Minimal example representation of a questionnaire based on *Collection+JSON*

Domain specific & generic hypermedia types

Depending on their semantic coverage, hypermedia types are often classified into *domain specific* and *generic* ones (e.g. in [RAR13]). The advantage of this classification is that it gives a quick impression concerning the usage characteristics of a media type.

Hypermedia types which do not define application semantics on their own or notably restrict features and usage of the underlying protocol are categorized as generic ones. Respectively, formats such *HAL*, *HTML* or *Siren* fall into this group. Without additional documentation, neither human nor machine can fully understand the semantics of, for instance, the *Siren properties* data elements as they are not covered by the specification. By contrast, domain specific media types describe all semantics of said domain as part of their definition. [1]

As an example, figure 4.9 shows a representation based on a hypermedia type named *Maze+XML* [Amu10]. Essentially, this media type allows to define a virtual maze con-

[1] Due to these definition, *Collection+JSON* does not qualify as a generic media type, but is definitely more of a generic instead of a domain specific media type because it does not explain things like the actual nature of its collections and items

```
<maze version="1.0">
  <cell href="/cells/C">
    <title>The End of the Tunnel</title>
    <link rel="west" href="/cells/B"/>
    <link rel="exit" href="/success.txt"/>
  </cell>
</maze>
```

Figure 4.9.: Example of representation based on the domain-specific media type *Maze+XML* (taken from [RAR13])

sisting of a network of cells [RAR13]. All cells are realized as separate resources and connected by control elements describing cardinal directions (e.g. `rel="east"`) or special cells such as *exit* of the maze (`rel="exit"`).

Given that all allowed elements such as *maze*, *cell*, *title* and *link* as well as the values for cardinal directions are specified by the hypermedia type, a client is able to understand the semantics of all *Maze+XML* based representations. So in this case, a client would understand that the representation describes a *cell* titled "The End of the Tunnel" which has one control element describing a connection to a cell in the west as well as another one to exit the maze.

As a result, *domain specific* hypermedia types have one crucial advantage with regard to the semantic challenge: Due to the fact that their specification covers all semantics, humans can read it to understand respective representations instead of being required to figure things out on the go. In addition, a client can be programmed against this specification so that it can autonomously deal with respective representations. By contrast, generic hypermedia types have to somehow provide additional documentation at runtime to fill in the blanks.

In return, one of the main problems of *domain specific* hypermedia types is that they only work that way within their domain. As soon as an application context even partially ranges outside said domain, you essentially face the same problems as when using a generic hypermedia type which makes them more or less useless. Beyond, *domain specific* hypermedia types seem to be quite rare [RAR13].

Usage and creation

So when designing an API, you should make use of existing, *domain specific* hypermedia types whenever possible. In case no suitable one exists, [RAR13] recommends to rather work with a generic hypermedia type and to provide the additionally required documentation by other means. This is because creating a media type which proves itself over time is not that simple. In addition, media types require client and server frameworks to be effectively usable in practice (e.g. to parse, validate and convert representations).

> Different hypermedia formats suit different services. The choice depends on a trade-off between reach and utility — between the ability to leverage existing widely deployed software agents and the degree to which a format matches our domain's needs [WPR10].

Consequently, an API should also avoid using too many different media types though it may be a valid option if the API provides significantly divergent content as indicated by the *top-level* type of a media type. Given that textual media types such as the ones used in this chapter were not designed to convey images, videos or the like, it would definitely makes sense to use an additional, domain specific media type like *SVG* for that purpose.

> If you ever find yourself defining 5 or 10 media types for a single API, that's a bad sign [RAR13].

Naturally, this lead to one question: How can you close this semantic gap left by generic media types? [2]

4.4. Closing the semantic gap

As mentioned in section 3.1.2, hypermedia has a different entry-barrier level depending on either human or machine clients interacting with a server. Respectively, there two approaches of working towards closing the semantic gap left by media types: Embedded documentation and profiles.

> If you're designing an API, and you know that all the decisions about state transitions will be made by human endusers, you don't need a profile at all. Websites don't have profiles. If you know that all the decisions will be made by automated clients, you don't need embedded documentation at all [RAR13].

Which approach to use thereby depends on the types of clients to be expected. Given that resources may be accessed by human and machine clients, both approaches may also be applied within the same representation.

Furthermore, both may fill in missing application as well as protocol semantics. Covering either would for instance be required when using a media type such as *JSON* (identify controls and cover their entire protocol semantics) or *HAL* (cover protocol semantics such as the HTTP method to be used).

Given that hypermedia types such as HTML or *Siren* are capable of dealing with the bulk of protocol semantics within representations themselves, [RAR13] recommends to prefer these if possible. That way, additional documentation only needs to handle application semantics. This also guarantees that a machine client can at least operate on behalf of a human user even if it is not capable of fully comprehending the content being transferred or the "higher-level" interaction being carried out.

4.4.1. Embedded documentation

As mentioned before, humans are fairly good at figuring things out. This is why in case of mostly human-driven clients, the simplest solution is to embed documentation within the elements of a hypermedia representation.

As demonstrated in figure 4.10, HTML allows to add embedded documentation by specifying a *title* or *value* attribute, or else as plain text within the element itself. Most

[2] This term has been coined by [RAR13] along "the semantic challenge".

media types seem to follow this example and support specifying *title* attributes as part of the elements they define.

```
1 <!-- actions -->
2 <form method="POST" action="/questionnaires/1">
3   Content: <input type="text" title="The content of the question"/>
4   <input type="submit" title="Creates and appends a question to this
        questionnaire" value="Append"/>
5 </form>
6
7 <!-- links -->
8 <a rel="self" title="Links to this questionnaire" href="/questionnaires
    /1">This questionnaire</a>
9 <a rel="item" title="Links to a question belonging to this questionnaire
    " href="/questions/1">Question 1</a>
```

Figure 4.10.: Example of embedded, human readable application semantics in HTML control elements

What is interesting about embedded documentation is that it often seems to be mainly used and discussed in context of control elements while being skipped in context of data elements (as simulated in figure 4.10). Presumably, this is because data elements tend to have dedicated identifiers which are meaningful enough to human users (one exception being the *analyisDelay* attribute). By contrast, control elements may be much more complex and thus require a more detailed description.

This in turn was one of the main motivations for distinguishing data and control elements as part of this thesis: Fundamentally, the approaches to make both element types human or machine-readable work mostly analogous in each case. But if you ignore data elements in context of human readable descriptions, it may misleadingly seem that their machine-readable description is somewhat special if you have not yet worked with similar approaches.

So for the sake of completeness figure 4.11 shows an example of data elements with human readable documentation using the HTML *title* as well (in an actual representation, the control elements shown in figure 4.10 would have been included within the top `<div>` tag). [3]

4.4.2. Concerning machine-driven interaction

As explained in section 3.1.2, the main problem with regard to machine-driven interaction is the semantic challenge: Working around machines being terrible at understanding instructions written in natural language. Naturally this is the biggest problem concerning REST APIs:

> Why bother to design APIs that serve self-descriptive message if those message won't be understood by their software consumers? [RAR13]

[3] Section 4.4.3 will explain why the keys from figure 4.1 are realized using the class attribute.

```
1  <div class="questionnaire" title="Represents a questionnaire">
2    <div class="analysisDelay" title="Duration based on ISO8601 format.
       Specifies how long a vote within this questionnaire must have been
        submitted to be used for analysis purposes">
3      PT5S
4    </div>
5    <div class="identifier" title="A short, unique name">
6      SE101SS14
7    </div>
8    <div class="title" title="A short description of the subject of the
       questionnaire">
9      Lecture evaluation: Software Engineering 101 (SS14)
10   </div>
11 </div>
```

Figure 4.11.: Example of embedded, human readable application semantics in HTML data elements

Generally, little progress has been made in this area and the semantic challenge will likely never be solved completely [RAR13]. Respectively, there are much less approaches than compared to, for instance, media types.

Despite the fact that those which are available may seem to be quite divergent, they can be broken down to the following three elements: Semantic aliases, profiles and profiles identifiers (see figure 4.12).

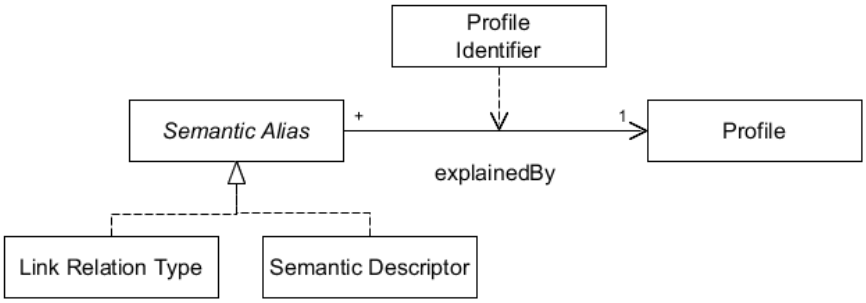

Figure 4.12.: Conceptual UML class representation of semantic aliases, profile and profile identifiers

Essentially, semantic aliases are short names which are embedded into representations. As indicated by their name, they serve as an alias for specific semantics (to be assigned to an element) which is why they are intentionally meaningless on their own. Instead, the semantics of one or more semantic aliases are explained in separate documents, called *profiles*. Due to this separation, representations additionally include profile identifiers to connect the included semantic aliases with a respective definition.

The whole purpose of this structure is to allow for clients to be programmed against the explanations contained in profiles *ahead of time*. This is why these explanations can be comprised in natural language. Due to the combination of semantic aliases and profile identifiers, clients can determine at runtime whether or not they are capable of understanding the semantics found within a representation. Assuming that representations provided by a server only employ semantic definitions clients have been programmed to handle, they can interact autonomously.

Given that there are quite a few challenges with regard to this approach in practice, the following discusses them in more detail including arising problems.

4.4.3. Semantic aliases

Equivalent to the elements a hypermedia representation, you can discern semantic aliases into two groups: Link relation types which are found in context of control elements and semantics descriptors which found in context of data elements. Though both serve the same purpose and essentially work the same, there are a few differences.

Link relation types

As defined by RFC5988 [Not10], link relation types identify the semantics of a link. Thereby, a Link is defined to be a typed connection between a contextual and a target resource, and to be comprised of four parts: [4]

- A context URI
- A link relation type
- A target URI
- Optional target attributes (which is why they will be ignored in the following)

With regard to the element types discussed in section 4.1, the target URI corresponds to the resource identifier found within a control element. The context URI identifies the resource of the representation within which the control element is found (often also referred to as the link's context). Respectively, a link relation type identifies the semantics of the state transition described by the control element. As demonstrated by various representations shown so far, media types mostly provide an attribute or property named *rel* to specify link relation types with control elements. For instance in case of *HAL*:

```
<link rel="self" href="http://example.com/questionnaires/1"/>
```

Furthermore, RFC5988 defines two kinds of link relation types: registered and extensions. As the name indicates, the first kind has been registered somewhere [RAR13], mostly the IANA registry [IAN14] from which all link relation type used so far in this chapter have been taken. As with media types, it is advisable to make use of such repositories whenever possible.

[4] RFC4287 actually states IRI instead of URI, but also says that the former will usually also be the latter. See section 3 of the RFC for details

Due to their registration, these link relation types may be short strings such as *item*, *current* or *self* because no naming collisions are expected. For exactly that reason, all other link relation types - the extensions - must be named by URI which may (but does not need to) identify a resource providing its definition (i.e. a profile).

For example, the IANA registry does not contain a link relation type equivalent to the semantics of the *append* control element in figure 4.2. A *HAL* control element using a accordingly defined extension link relation type could look like this:

```
<link rel="http://example.com/relations/append"
href="http://example.com/questionnaires"/>
```

Based on this definition, extension link relation types can serve as both semantic alias and profile identifier while registered ones only qualify semantic aliases. However, there are two more important conventions to be aware of as described by [RAR13]:

First, the IANA link relation type registry, more precisely, the definitions to be found there, are normally considered to be available implicitly. Consequently, registered link relations allow for self-descriptiveness even though there are no explicit profile identifiers for them.

Second, extension link relation types may be treated as registered ones (i.e. referred to by a short string) within a representation if the latter contains an explicit profile identifier. Assuming that the remainder of a representation does contain a profile identifier, the *append* link could thus look like this:

```
<link rel="append" href="http://example.com/questionnaires"/>
```

Respectively, the extension link relation type in this case looks and works just as a semantic alias.

Link relation types & HTML and Siren

Even though link relation types are generally supported by HTML, there is an inconvenient limitation: Both formats only allow for them to be used in context of navigational control elements, that is Siren's links object and HTML's `<a>` tag. Hence, you cannot specify link relation types with HTML forms or *Siren actions*.

In case of HTML, a common workaround is to use the *class* attribute as a substitute (see figure 4.13). Given that *Siren* defines equivalently named property which seems to be somewhat inspired by HTML, you could do the same.

```
1 <!-- other representation parts -->
2 <form method="POST" action="/questionnaires/1">
3   <input type="text" class="content"/>
4   <input type="submit" class="append" value="Append"/>
5 </form>
```

Figure 4.13.: Example of using the HTML's class attribute to specify semantic aliases

Nevertheless, this is more of a hack: "HTML's *class* attribute was designed to convey information about visual display (via CSS), not to convey bits of application semantics"

[RAR13]. Likewise, the specification of *Siren* does not state that its class attribute is supposed to be used that way.

So, if you use one of these media types and want machine-readability, you have to work around this problem. You have to provide extra documentation which however will be only human-readable. Probably one of the best places to do that is the specification of a profile format which will be covered in section 4.4.5.

Semantic descriptors

In contrast to link relation types, *semantic descriptors* are not standardized. Just as the concepts of application and protocol semantics, they have been suggested by [RAR13]. Consequently, there are no rules concerning registered or unregistered semantic descriptors.

Furthermore, semantic descriptors are often not specified by way of a dedicate representation attribute or property. As explained in section 4.4.1, data elements usually feature meaningful names or identifiers. Respectively, keys of *JSON* based representations as well as *tag* names of *XML* based ones (see figure 4.14) often simultaneously serve as semantic identifiers even though this rarely seems to be stated that explicitly. Nevertheless, there are two exceptions:

```
1 <resource href="http://example.com/questionnaires/1">
2   <analysisDelay>PT5S</analysisDelay>
3   <identifier>SE101SS14</identifier>
4   <title>Lecture evaluation: Software Engineering 101 (SS14)</title>
5 </resource>
```

Figure 4.14.: Example of semantic descriptors based on HAL data elements

One the one hand, representations which already misuse the HTML *class* attribute for link relation types, usually seem to do the same for semantic descriptors. This is why in figure 4.11, the attributes names of a questionnaire were listed as class attributes. [5]

On the other hand, there is an open standard called *Microdata* which defines five new properties for HTML5: *itemprop*, *itemscope*, *itemtype*, *itemid* and *itemref* [RAR13]. As shown in figure 4.15, *itemprop* can be used to specify semantic descriptors while *itemtype* serves as a *profile identifier* (*itemscope* is a required flag to indicate the presence of Microdata). While solving the problem of "misusing" the *class* attribute, the standard still leaves two problems.

First, the standard is incapable of defining and specifying link relation types which is why you would need an additional approach to deal with them. Second, it is only applicable to HTML5 which, compared to *XML* or *JSON* based formats, is far more rarely used for Web APIs [RAR13].

Other than their placement within the representation, semantic descriptors thus work relatively similar to (extension) link relation types and the concept explained in the beginning of section 4.4.2. They are short strings and require a profile identifier to be stated within the same representation in order to machine-readably communicate the semantics of data elements.

[5] Interestingly enough, Siren employs both possibilities

```
1 <div itemscope itemtype="http://example.com/descriptors/questionnaire">
2   <div itemprop="analysisDelay">PT5S</div>
3   <div itemprop="identifier">SE101SS14</div>
4   <div itemprop="title">Lecture evaluation: Software Engineering 101 (
        SS14)</div>
5 </div>
```

Figure 4.15.: Semantic descriptors using Microdata

4.4.4. Profile Identifiers

Sections 4.4.2 and 4.4.3 mentioned several times that semantic aliases require a profile identifier to be present within the same representation to effectively communicate the semantics of an element.

Interestingly enough, many APIs actually feature profiles. The only problem is that clients are expected to know where to find them and how to apply them ahead of time. As explained by [RAR13], there are generally three ways to identify profiles.

Special purpose control elements

As demonstrated in figure 4.15, a standard or format may define special control elements such as the itemtype property for that purpose. In contrast to HTML5, HTML4 allows to do the same thing by specifying an attribute named *profile* within the <HEAD> tag: [RAR13]

```
 <HEAD profile="http://example.com/profiles/questionnaire">
...
</HEAD>
```

Link relation type

Alternatively, you can use the *profile* link relation type defined in RFC6906 [Wil13]: Adding it to a control element indicates that the link's context conforms to the profile identified by the respective target URI. Using *HAL*, a respective element could look like this:

```
<link rel="profile" href="http://example.com/profiles/questionnaire"/>
```

In case the employed media type does not allow to specify an according control element, the HTTP Link header can be employed the same way:

```
Link:  <http://example.com/profiles/questionnaire>; rel="profile"
```

Media type parameter

Nevertheless, both these options have one problem: They can only be used to characterize what is being sent rather than what is being requested. The latter is for

instance useful if a client was to request one out of multiple representations each being described by a profile. Provided that the definition of the respective media type explicitly allows to do so, you can use a media type parameter for both scenarios using the HTTP `Content-Type` and `Accept` headers:

```
Content-Type: application/vnd.collection+json; profile="http://example.com
/profiles/question"

Accept: application/vnd.collection+json; profile="http://example.com
/profiles/questionnaire"
```

As of now, this is possible when using *Collection+Json*, *JSON-LD* [14], *HAL* or *XHTML* [RAR13].

4.4.5. Profiles

Section 4.4.2 defined profiles as documents explaining the semantics of semantic aliases. What is both advantage and disadvantage of this concept, is that its definitions are generally not much more concise than that. Aside from not conflicting with the semantics of a media type, their only requirement is being identified by URI [Wil13].

> The concept of a profile has no strict definition on the Internet or on the web. For the purpose of this specification, a profile can be described as additional semantics that can be used to process a resource representation, such as constraints, conventions, extensions, or any other aspects that do not alter the basic media type semantics (RFC6906) [Wil13].

Despite the fact that RFC6906 does not require profiles to be realized as resources, it is essential to make the concepts of *semantic aliases*, *profile identifiers* and *profiles* work: At design time, developers need to access and learn about semantic definitions anyway to correctly implement a respective component. At runtime, being able to access profiles representations is further beneficial in case a client is able to deal with previously unknown definition; Even if it just renders the definition for a human user.

In this regard, a common misused concerning API specific profiles is to use them for code generation [RAR13]. Even though profiles may be available of design time (in terms of the API), they should always be considered being only available at runtime. Otherwise, you effectively implement against a static interface described by these profiles which is equivalent to using RPC in the first place. By contrast, creating implementation bits to deal with reusable pieces used within these profiles does not contradict the idea. [6]

Writing profiles

Given that humans program against definitions provided by profiles, the latter can be based on natural language which seems to quite often be the case. However, working

[6] As mentioned before, RPC is not be deemed a better or worse interaction style. But it is important to understand in which situation you effectively change the interaction style while trying to be REST compliant

with, editing or reusing such profiles tends to be quite tedious. In contrast, machine-readable profiles can easily be searched and accessed to obtain a representation as well as reused and remixed in context of other APIs [RAR13].

Due to REST being based on exchanging representation via a *uniform interface*, reuseability of semantic definitions is essential for integration. The only way to avoid n-to-n integration scenarios in such an environment is to have a common ground concerning the parts used to compose representations.

As a consequence, profiles should at least to this extend be machine-readable. The easiest way to achieve that is to use a respective media type which, in this context, is more often referred to as an interface description language (IDL). Beyond that, whether to focus on machine or human readability, is a trade-off.

Naturally, working with heavily machine-driven approaches normally requires effort by developers given that they have to deal with parsing, validation and the like. In return, a respective format may however allow for autonomous adaption of implementations in the event of some changes, or even for handling entirely new profiles.

On the other hand, formats relying heavily on machine-readability run the risk of quickly becoming "too complicated for most developers to even consider using" (*RDF Schema* in this case) [RAR13]. As exemplary explained by [RAR13], the goal of *RDF-Schema* and *OWL* (Web Ontology Language) is to create profiles using a rather limited set of core definitions and rules based on which new definition can be derived. The result are predominantly machine-readable descriptions which may conceptually be constructed like this:

> Date of birth: The date of the event in which a person change state from nonexistence to existence.

Following the idea that, for most applications, it is easier for client developers to just write some code based on a human readable description [RAR13], many other IDLs such *XMDP* [GMP15], *Swagger* [Wor14] or *ALPS* [ARF14] focus much more on the human-readable side of things with varying extend of machine-readability.

The down-side of these approaches is that according implementations will break upon most changes because there is no way to anticipate or to adapt to them. If you were to define a title as a string consistent of 50 characters and later change it to 20, old implementations have a high chance of failing to work. Given the popularity of Swagger, most developers however seem to prefer these approaches to the heavily machine-driven ones.

ALPS

As an example, figure 4.16 shows an according profile using the media type *ALPS* (`application/alps+xml`) [ARF14]. Essentially, this format uses `<descriptor>` tags to define human readable semantics (`<doc>...</doc>`) for a specific semantic alias (`id="analysisDelay"`). It thereby expects the description to be informative enough for a developer to figure out how to implement it.

In addition, ALPS allows for specifying a *type* attribute to characterize data (`type="semantic"`) and control elements based on their state transition type (`type="unsafe"`). So in contrast to the *Microdata* standard, *ALPS* may also define link relation types.

```
<alps version="1.0">
  <doc format="text">A exemplary questionnaire profile</doc>

  <-- base elements -->
  <descriptor id="analysisDelay" type="semantic">
    <doc>Duration based on ISO8601 format (...) </doc>
  </descriptor>
  <descriptor id="identifier" type="semantic">
    <doc>A short, unique name</doc>
  </descriptor>
  <descriptor id="title" type="semantic">
    <doc>A short description of the subject of the questionnaire</doc>
  </descriptor>

  <-- state transitions -->
  <descriptor id="append" type="unsafe">
    <doc>Creates a new question and appends it to this questionnaire</
       doc>
  </descriptor>

  <!-- container -->
  <descriptor id="questionnaire" type="semantic">
    <doc>A questionnaire featuring a set of questions</doc>
    <descriptor href="#analysisDelay"/>
    <descriptor href="#identifier"/>
    <descriptor href="#title"/>
  </descriptor>

</alps>
```

Figure 4.16.: A machine readable profile of a questionnaire based on *ALPS*

Beyond that, you can hierarchically group multiple descriptors elements to provide more expressive description of complex (data) elements. By referring to other descriptors using the *href* attribute, the referenced definition remain independent, may be contained in a separate profile and reuse by others.

4.4.6. Reusing machine-readable descriptions

At this point, you might argue that a profile format like *ALPS* does not satisfyingly address the semantic gap - in other words, is not sufficiently powerful - given that it lacks too many details as the following examples:

- Is an identifier a number or string? (types)
- Is a questionnaire required to have an identifier or could it have none or multiple? (multiplicities)

- Does a client need authentication to append a question to a questionnaire? (authentication)

This is because *ALPS* has been specifically created to solve a problem which almost all other available IDLs have [RAR13]: Due to the fact that they allow for describing these and other details, definitions created by these formats rarely end up being reusable. In other words, they can only describe “a single instance of a single server for a single service” [Amu13].

Apart from an exceeding level of detail, this might also be due to the following reasons:

- Dependency on a specific protocol
- Dependency on a specific media type
- Specification of actual URLs
- Fixed workflow or representation structure

As demonstrated in figure 4.16, ALPS solves these problems by omitting all respective details. While this makes its definitions well reusable, it introduces another problem: Clients now lack essential, semantic details concerning the API.

Consequently, the approach discussed so far is only capable of working towards either self-descriptiveness or reuse by sacrificing the respective other. Given that you actually need both to make the efforts towards machine readability worthwhile, this is a problem.

Specification lock-in

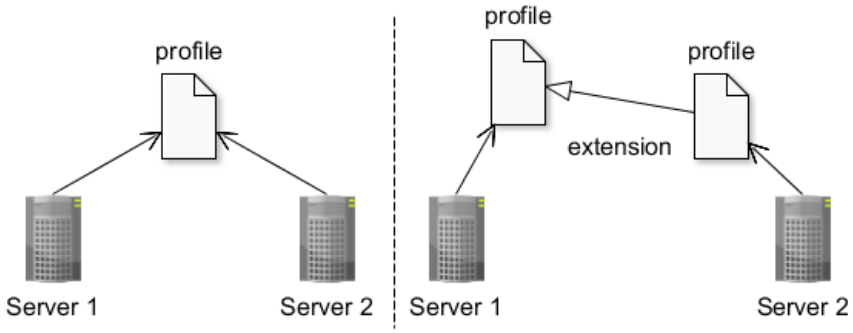

Figure 4.17.: Reuse scenarios concerning profiles definitions

Beyond that, there is another issue with regard to reusing profiles in general. Ignoring the limits of client adaption for a moment, a profile can be changed by a server as needed provided that it is exclusively used by said server. But in case a profile (document) is shared between multiple or extended by other servers (see figure 4.17), updates must be coordinated.

Otherwise, the server changing the profile would break the remaining ones given that these would still provide the same, “old” content while including the updated, now incompatible profile or an inconsistent extension.

Consequently, you would need to change all servers at the same time. Though, in large distributed systems such as the Web or enterprise application landscapes, you usually do not have the authority or budget to do so. Effectively, you thus loose extensibility due to what you could call a specification lock-in: Once published, profiles being reused cannot be changed. [7]

> The other objection to the concept of a uniform interface is that it merely shifts all coupling issues and other problems to the data exchanged between client and server [Vin08].

Both these problems are addressed in the next chapter.

[7] The effect is actually quite similar to problems discussed in context of API minimalism

5. Towards a Data-Driven Enterprise Resource Architecture

To address the identified problems concerning expressiveness, reuse and extensibility of profile definitions, this chapter first introduces a domain ontology which represents an approach of combining human and machine-readable documentation for that purpose. Subsequently, it discusses an implementation of said domain ontology based on three formats. Last, the chapter reflects employing this approach in an enterprise application environment based on its potential influence on respective architectures.

5.1. A new domain ontology

From a conceptual perspective, the profile based approach concerning machine-readability can be partitioned into three domains: A *protocol domain* dealing with the protocols and their features (e.g. HTTP vs COAP), a *format domain* whose subject are the various media types available for a particular protocol, and a *semantic domain* which comprises approaches of how to close the semantic gap left by media types.

With regard to this classification, the problems discussed at the end of chapter 4 mostly occur within the semantics domain. As shown in figure 5.1, the domain ontology presented in the following addresses these problems by splitting said domain into two replacements: A *problem* and a *solution* domain. [1]

Goals

First and foremost, the ontology model aims to resolve the profile related conflicts between expressiveness, reuse and extensibility discussed in the end of the last chapter. For that purpose, it is to exploit the benefits of both machine and human-readable definitions without overburdening developers with complexity.

> One line of great documentation in the right place can save hours and hours [Lac13].

Second, it aims provide multiple views on descriptive efforts in context of REST based on its domains. This in turn is to help avoid misunderstandings when talking about the different aspects of realizing machine-readable representations which happened frequently during the preparation of this thesis.

Overview

If a client is to request a representation from a server, it uses an instance of the protocol domain, that is a protocol, by which request and said representation are carried.

[1] The term “problem domain” is inspired by the ALPS documentation [ARF14]

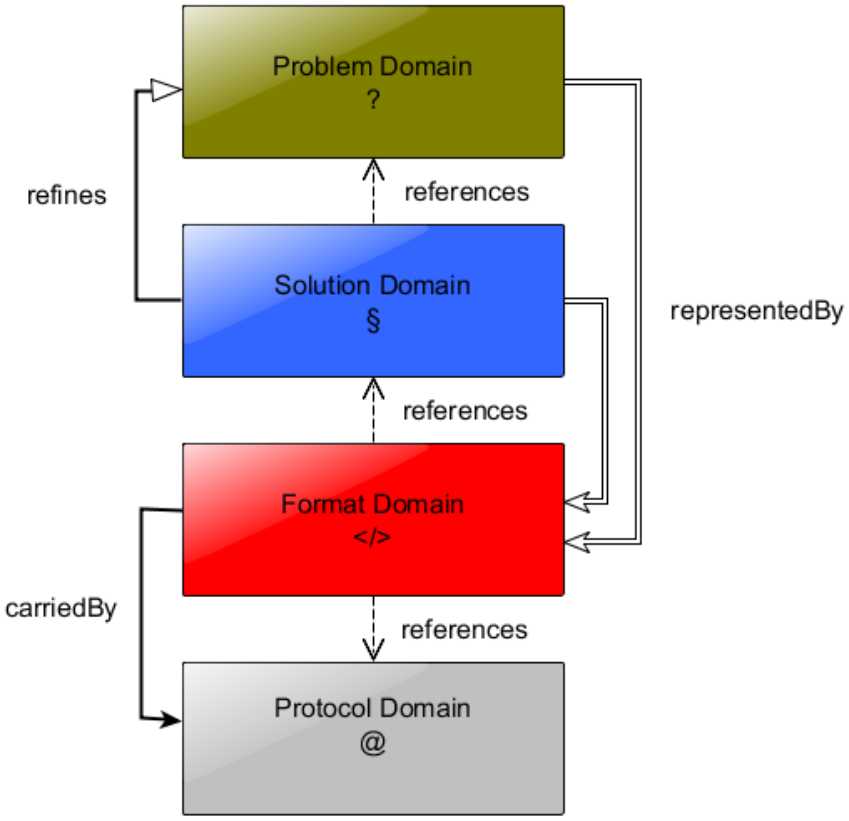

Figure 5.1.: Domain Ontology for Connected REST APIs

In order to process the received representation, a client has to understand its media type which corresponds to an instance of the format domain. As explained in the last chapter, elements of a hypermedia representation can generally convey protocol as well as application semantics. This is why the format domain conceptually refers to both, the protocol and solution domain.

Provided that the representation contains semantics not covered by the media type, the client can use included profile identifiers which refer to solution domain profiles explaining these semantics. To allow for machine-readability, these profiles are based on, or in other words represented by, a predominantly machine-readable format.

However, solution domain profiles do not define all semantics in place. Instead they reference and refine reusable definitions captured as problem domain profiles which, by contrast to the solution domain, are represented by a predominantly human-readable format.

Impact on development

From the perspective of a server, the problem domain “just” provides reusable specifications being used to build fully fledged (solution domain) profiles which are sent

along with representation. Due to the abstraction characteristic of resources, the implementation of the server is not necessarily affected. Likely, only the design approach is affected by the two semantic abstraction levels. By contrast, things on the client side would be more evolutionary.

Due to initially limited framework support, a client would probably be programmed to expect specific, fully detailed solution domain profiles. Over time and with increasing framework support, client implementations should be able to determine, handle and adapt to more and more details defined as part of solution domain profiles at runtime.

Ultimately, it should ideally be possible to implement a client based solely on profile domain definitions the implementer expects the client to be dealing with.

The more clients can handle as solution domain refinements, the higher the reuse potential of problem domain definitions without facing the problem of limited expressiveness or extensibility.

5.1.1. Problem Domain

As just explained, the problem domain is about defining semantics in a fundamental way allowing for their reuse and refinement as needed per application context. For that to work, its definitions must be independent of a particular format or protocol. To facilitate their reuse, they should ideally be collected in widely accessible repositories or registries.

Based on definitions of the problem domain, developers should understand the context they are dealing with. That is, which concepts, terms or relationships to expect without knowing all the specific details of an actual implementation. Respectively, definitions of the problem domain should answer questions such as:

- What is a title?
- Which other elements can be reasonably expected in its context?
- What is the general meaning of terms such as *append* or *publish*?

If one is familiar with definitions of the IANA link relation registry, one may notice that many of its definitions suit the problem domain. However, the goal of the problem domain is to provide definition of a comparable abstraction level for both types of semantic aliases, not only link relation types.

Formats

As stated in the last chapter, people generally seem to prefer human-readable formats for defining profiles. With regard to the example of deriving a birth date from basic concepts, you can further argue that a great deal of the complexity of machine-readable profiles stems from establishing rather basic definitions. Also considering the responsibilities of problem and solution, it thus makes sense to employ a predominantly human-readable format for representing definitions of the problem domain.

Nevertheless, a respective format must also be machine-readable to some extend to allow for definitions to be reused and extended within the solution domain.

Abstraction level

In general, one of the main challenges with regard to the problem domain is finding a generally suitable abstraction level as guidance for its definitions. Given the fundamental difference between data and control elements it makes sense to keep up this distinction. To some extend, the solution domain can also help in this regard: Every detail being easily expressible in a machine readable way should likely be excluded from the problem domain.

```
1  <!-- before -->
2  <descriptor id="title" type="semantic">
3    <doc>A short, description of the subject of the questionnaire</doc>
4  </descriptor>
5  <descriptor id="append" type="unsafe">
6    <doc>Creates a new question and appends it to this questionnaire</doc>
7  </descriptor>
8
9  <!-- after -->
10 <descriptor id="title" type="semantic">
11   <doc>A short, description of the element's context</doc>
12 </descriptor>
13 <descriptor id="append" type="unsafe">
14   <doc>Creates a new resource and appends it to the link's context</doc>
15 </descriptor>
```

Figure 5.2.: Abstracting definitions to the level the problem domain

Based on this principle, you can reduce a control element to the semantics of its state transition. Everything else, transition input and output, authentication or authorization restrictions can be added within the solution domain. As a consequence, the semantics of a (former) link relation type describing *append-question* action discussed in the last chapter, would be reduced to a generic *append* not specifying what is being appended. With regard to data elements, you can strip all information outside its very nature such as types, constraints as well as structural aspects like multiplicities or affiliation to bigger context. The difference of respective definitions is exemplary shown in figure 5.2.

Thereby, standards such ISO8601 used for the *analyisDelay* attribute can be a controversial subject. The problem is that they often come with a dedicated syntax which is one of the things you would generally not want to specify within the problem domain. Given that they also frequently come with several syntax variants of which only one is effectively used, you could however argue that it is enough of an abstraction. Altogether, this is one of the aspects that go beyond the scope of this thesis given that they have to be evaluated over time.

Basic elements

Respectively, the main focus of the problem domain are general, stable and atomic definitions of semantics for semantic descriptors and link relation types. Inspired by

ALPS and to avoid using the term “profile” in two contexts, I will refer to single (definition) element of the problem domain as a descriptor. [2]

Thereby, “stable” means that the definitions of descriptors should ideally never change. Given that these are intended to be reused and refined in many APIs, there is a high chance of breaking dependent definitions - especially in consequence of using a primarily human-readable format. Though flooding the problem domain with similar definitions generally contradicts the objectives of the problem domain, it may be a better idea to create a new descriptor in such a case.

In addition, the “term” atomic is to emphasize that it not should be possible to partition its definitions into multiple, independently sound ones. This is to improve reuseability and to reduce disruption due to changes which is best explained based on the example.

The IANA registry [IAN14] contains a link relation type named *edit* which essentially composes the behavior of read, replace and delete. The problem of such a definition is that you can expect its components to be employed independently which makes the composite definitions redundant (as explained in section 4.4.3, a control element can feature multiple link relation types).

```
1 <!-- before -->
2 <link rel="edit" href="http://example.com/questionnaires/1"/>
3
4 <!-- after -->
5 <link rel="http://example.com/relations/read" href="http://example.com/
     questionnaires/1">
6 <link rel="http://example.com/relations/replace" href="http://example.
     com/questionnaires/1">
7 <link rel="http://example.com/relations/delete" href="http://example.com
     /questionnaires/1">
```

Figure 5.3.: Splitting non-atomic definition in the problem domain

Furthermore, changing any part of the definition will likely affect all clients using the link relation type. As shown in figure 5.3, this effect can be diminished by rather using respective, atomic link relation types each combined with a control element. That way, only clients using the respectively changed atomic definition are affected. [3]

Conflicting descriptors

Due to the fact that the problem domain is heavily driven by human-readable documentation, the expressiveness of descriptors always depends on the decisions of their author. Just as the problem of updating existing definitions, it requires the problem to support multiple, conflicting yet distinguishable definitions of the same semantic alias.

> But, like the Web, we would have to accept some things [...]: That we would never be able to get everybody to agree on everything. That we would never

[2] Furthermore, ALPS will be used as problem domain format later in this chapter

[3] For the most part, you would skip defining these link relation types explicitly as their semantics are well covered by for instance HTTP.

> be able to guarantee that any two sets of knowledge mixed together on the web would be guaranteed to be consistent [Jim].

As explained in the last chapter, this can easily be solved by the combination of semantic alias and profile identifier. Nevertheless, it is important to understand that the general problem of conflicting definitions potentially undermines the objectives of the problem domain: You cannot reap the benefits of partitioning the semantic domain if authors cannot agree on **any** common basis.

Having said that, you can also consider this problem to be an advantage. This is because it allows you to incrementally converge on a suitable solution or to intentionally keep things separated in case the unified, big bang solution does not yield effective benefits.

Composite descriptors

Motivated by the definition of a *questionnaire* using *ALPS* in figure 4.16 (see figure 5.4 for the according excerpt), the problem domain also has to allow for defining *composite descriptors*: Descriptors which are characterized by their *component descriptors* but also feature semantics on their own which is why they cannot be partitioned losslessly.

```
1 <!-- container -->
2 <descriptor id="questionnaire" type="semantic">
3   <doc>A questionnaire featuring a set of questions</doc>
4   <descriptor href="#analysisDelay"/>
5   <descriptor href="#identifier"/>
6   <descriptor href="#title"/>
7 </descriptor>
```

Figure 5.4.: Excerpt of figure 4.16 showing the definition of the composite descriptor based on ALPS

In this specific example, the composite descriptor does not contain a component one describing link relation types. Furthermore, none of its component descriptors is in turn a composite one. That is, the top level composite descriptor has a recursive depth of one.

Generally, it is left to an author to decide the recursive depth as well as whether or not a component descriptor of any type is relevant enough to be included. Given that these structures eventually serve as blueprints for representation, designing overly nested descriptors however often implies designing embedded resources which should be avoided if possible. Likewise, including a component descriptor representing a link relation type should rather rarely be the case because their definition within the problem domain is fairly generic. Including a descriptor such as the previously explained *append* would not convey much information because it does not state what could be appended (a question or category or tag or ...).

Beyond that, the presence or absence of a component descriptor within a composite one must not indicate its assured presence or absence in a refining (solution domain) profile. Rather, the presence of a component descriptor within a composite one indicates a reasonable probability of its presence in a refining (solution domain) profile.

In this regard, the *questionnaire* descriptor shown in figure 5.4 is a good example. While the *identifier* and *title* descriptors are probably found in most implementations, the *analysisDelay* `descriptor` is definitely an exception which is why it should not appear in a problem domain definition.

The rationale of this approach is to reduce the general problem of conflicting definitions while, at the same time giving developers a better idea of what to expect in context of a specific composite descriptor. That way, client implementations may have a higher chance of being able to deal with variants or changes of (solution domain) profiles that are based on the same composite descriptor. Admittedly, this approach is the less exploitable the more these refinements generally deviate from the structure indicated by the composite descriptor.

Overly extensive definitions (component-wise) cause huge and mostly redundant efforts for implementing clients when being used for rather simple APIs. Imagine a questionnaire abstraction having 50 or more component descriptors with less than 10 being relevant to the respective implementation.

Keeping definition of composite descriptors too limited on the other hand defeats their purpose because clients are “regularly” confronted with unexpected component descriptors. Overall, finding a suitable and effective composite descriptor definition hence will likely take time and practice.

5.1.2. Solution Domain

While the problem domain is about defining rather abstract, fundamental and reusable semantics by way of descriptors, the solution domain aims to refine these definitions by supplementing all those details the problem domain left out on purpose. Examples of such details are:

- Multiplicities of data and link items
- Allowed values of a data item
- Parts of a representation that can be modified by a client (mutability)
- Relations to other resources
- Capabilities required to trigger a state transition
- Representations sent and received as part of a state transition

Semantic-wise, a (solution domain) profile thus should provide an exact blueprint of a representation in terms of data and control elements. However, and in accordance with the problem domain, the definition of such a profile should as well be independent of a particular format or protocol.

Formats

Given that the problem domain heavily relies on human-readable descriptions to define semantics, its complexity is rather “unbound”: Implementing descriptors becomes as difficult and extensive as definitions that humans can come up with.

As a result, the most important property and objective of the solution domain is constant complexity. This is achieved by referencing definitions of the problem domain

and by using a static set of properties, values and rules; That is, machine-readable formats.

On the one hand, this makes it easier for humans to understand corresponding definitions because they can rely on a repeating pattern. In addition, there is no need to bother with entire problem domain definitions on every occasion as you can ensure (by reference) that everybody refers to the same descriptor. On the other hand, this approach is simply required to enable machines to auto-process and adapt to changes of solution domain definitions assuming that they have been taught the involved problem domain definitions.

Both would not work in case you allow for defining arbitrary semantics within the solution domain. In addition, it would open up the possibility of essential definitions leaking from the problem domain. As a consequence, you could certainly challenge the separation of problem and solution domain in the first place and expect seemingly unending discussions regarding which bits of semantics should be defined in which domain.

Elements

Despite the fact that some IDLs such as Swagger allow to describe multiple resources up to an entire API within one document, a (solution domain) profile should only cover the semantics of a single representation. Apart from reduced complexity, there are three more reasons for that:

> Don’t keep all the hypermedia in one place [RAR13].

First, it tempts server developers to generate the description document based on their implementation as well as client developers to generate code on what they assume to be a stable document [RAR13]. Especially the latter contradicts the idea of the solution domain because the generated code will most likely break upon changes instead of adapting as intended.

Second, it often diminishes human readability by bothering the reader with situationally irrelevant information. Depending on the technical support for viewing and navigating the respective document, developers would always be required to skim through other definitions in search of a specific one.

Third, every change needlessly affects the entire API description which worsens both previously described issues. By partitioning the descriptions, you ideally only need to review and adapt the code concerning the changed part (in case autonomous adaption failed).

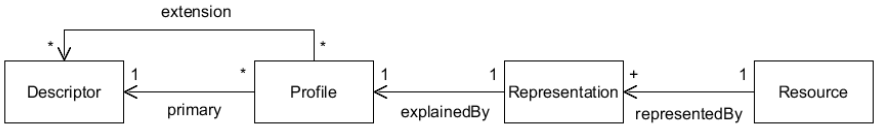

Figure 5.5.: Structure and relationships between descriptors, profiles, representation and resources based on UML class diagram

Just like the problem domain, the solution domain hence should feature a modular structure consisting of profiles each describing a single representation. The resulting

model is displayed in figure 5.5. [4]

Conceptual mechanisms

From a conceptual perspective, the solution domain mainly employs two mechanisms: Selection and extension.

The idea of **selection** is to derive a profile from suitable (composite) descriptors by explicitly stating which ones are to be found in the resulting profile. As a benefit, this also enables the problem domain to incorporate backward-compatible changes without running the risk of unintentionally leaking them into solution domain profiles.

Given that profiles are likely often based on a single, primary composite descriptor, the selection mechanism further allows to employ said descriptor as a type classifier for the resulting profile (and thus representation).

In contrast to selection, the **extension** mechanism is to extend descriptor definitions by adding others. Based on the rationale of composite descriptors, the *analysisDelay* `descriptor` would be an example of such a case.

Scope & reuse

Even though the solution domain is about adding details, it must not contain implementation related ones. This includes aspects like URL or data values as well as security details like the employed authentication protocol or the meaning of authorization terms (i.e. roles, permissions and the like).

Initially, this is required for allowing to independently choose media type and protocol as needed. Beyond that, keeping implementation details out of profiles facilitates their reuse not only for identical instances of the same profile in different locations (replication) but also when moving profiles between locations [MSW09]. In the long run, it might even allow to use profiles to simulate and assert an API similar to how mocks are used for graphical user interfaces (GUIs).

Furthermore, considering that especially enterprise application APIs often feature many resources of the same kind (e.g. *questionnaires*, *questions*, *answers*, ...), there is also an efficiency-related motivation for reusing profiles. Nonetheless, they must not be reused by reference.

This is because profiles, in contrast to descriptors, do not avoid the specification lock-in problem. While descriptors are required to change as rarely as possible - ideally never - profiles are virtually expected to do so over time.

Consequently, profiles must only be reused *by copy* (see figure 5.6). If a server is to reuse an existing profile, it must stay independent; For instance by way of a separate resource providing a copy of the respective document. The only exception is when being able to guarantee that all involved resources are (required) to change at the same time: For instance due to shared implementation or in replication scenarios. [5] Overall and in contrast to descriptors, profiles hence should be located nearby the representations referring to them.

[4] Up to this point I most used the term *(solution domain) profile* to emphasize the intended type of profile. For the remainder of this thesis, term profile will always refer to the solution domain.

[5] This is why figure 5.5 shows a one-to-one relationship between profiles and representation

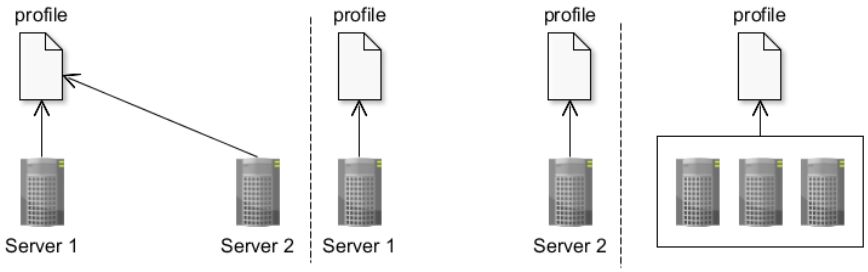

Figure 5.6.: Reuse by references vs. reuse by copy vs. replication

5.1.3. Format & protocol domain

Given that the domain ontology makes extensive use of profiles, a chosen media type should definitely have some type of support for connecting its representation elements to the semantic definitions found within a linked profile. In fact, the format specified for the solution domain in section 5.2.2 will even require a respective media type parameter to exploit all its features. Due to the extensive coverage of media types in section 4.3, there is not much more to add at this point except for the interaction between format and protocol domains.

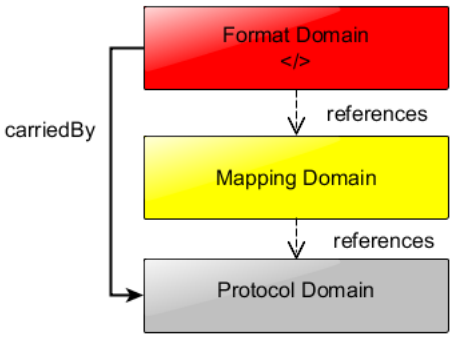

Figure 5.7.: Earlier draft of the domain ontology including the mapping domain

Section 3.1.3 stated that a REST API should not be dependent on a single protocol; Yet, most media types are. In view of media types requiring frameworks to be effectively used as well as other reasons such as supported features, resolving this dependency by switching media types as required per protocol may often not be feasible.

In consequence of this issue, earlier drafts of the domain ontology contained a additional *mapping domain* between format and protocol domain (see figure 5.7). Its function was to provide a uniform interface as well as mapping between media types and protocols.

However, the meaningfulness of such an approach mainly depends on the meaningfulness of using or switching between multiple protocols for the same API. Given that this question goes beyond the scope of this thesis, the mapping domain was removed from the final version of the domain ontology.

5.2. Implementing the domain ontology

Now that the domain ontology has been covered in terms of goals, structure and concepts, this section presents a respective implementation. While the problem domain is realized by using ALPS, the following introduces a new format for each the solution and the format domain along with a motivation for their creation.

5.2.1. Problem Domain

Given that section 4.4.5 already discussed the central aspects of the format, this section mainly focuses on its usage according to the objectives of the problem domain.

The main reason for using *ALPS* is that its motivation and goals are actually quite similar to the ones of the problem domain: Providing the means to create a reusable vocabulary and transition set for a common domain [ARF14]. In fact, the format was one of the main inspirations for formalizing the problem domain and creating the domain ontology.

Nevertheless, the two approaches differ in terms of intended usage. While the *ALPS* specification envisages its definitions to be directly applied to representation, the approach presented in this chapter employs it for profile definitions. As demonstrated by the examples in section 5.1.1, ALPS can nonetheless be employed according to the objectives of the problem domain provided that authors keep definitions compliant to the discussed abstraction level.

Apart from human-readable definitions themselves, there are two more things found within the ALPS specification which should be avoided for that purpose.

```
1  <descriptor id="questionnaire" type="semantic">
2    <doc>A questionnaire featuring a set of questions</doc>
3
4    <descriptor id="identifier" type="semantic">
5      <doc>A short, unique name</doc>
6    </descriptor>
7
8    <descriptor id="title" type="semantic">
9      <doc>A short description of the element's context</doc>
10   </descriptor>
11 </descriptor>
```

Figure 5.8.: Nested descriptor definitions in ALPS

First, *ALPS* generally allows for nested definitions of descriptors to indicate that these should only appear within its composite descriptor [ARF14]. As demonstrated in figure 5.8, you could define the *identifier* and *title* descriptors within the *questionnaire* descriptor itself.

Given that the problem domain heavily emphasizes reuse of definitions, such component descriptors should instead be included by reference using the *href* attribute as for instance shown in figure 5.4.

Second, *ALPS* allows descriptors representing a link relation type to specify a *rt* attribute indicating the resource type returned when executing the transition. As shown

```
1 <descriptor id="item" type="safe" rt="question">
2   <doc>Links to a question</doc>
3 </descriptor>
```

Figure 5.9.: Descriptor specifying a return type in ALPS

in figure 5.9, you could for instance define a specialized *item* link relation type suggesting that it refers to a *question*.

Considering that its value is defined to be an opaque string [ARF14], its specification would not only contradict the abstraction level of the problem domain. It would also conflict with the objective of the solution domain to provide this information in a machine-readable, that is non-opaque, way.

5.2.2. Solution domain

By contrast to the *problem domain*, the implementation of the *solution domain* features a specifically created format. This is simply because, due to the novelty of this approach, no sufficient format exists.

In reference to the title of this chapter as well as the *solution domain*, the respective media type is named *era-sd*. Given that *YML* allows for the most concise syntax and can be converted to *XML* and *JSON*, the following examples are based on said format.

Overview

As shown in figure 5.10, the *era-sd* profile model is build on the hypermedia representation model build in the last chapter as well as the idea of refining *problem domain* definitions: [6]

Essentially, each profile comprises a sequence of *data* and *control* elements each being based on a dedicated descriptor. In addition, it references another descriptor as the primary, composite one by way of the *descriptor* child node within the *profile* node.

Beyond that, the *era-sd* format features three distinct design decisions which will be covered in the following.

Opaque security handling

First, the *era-sd* format almost only employs opaque strings for the indicating security related details. This is to allow for their specification without requiring the inclusion of implementation details such as definitions of capabilities or a chosen authentication protocol.

As demonstrated by the *append* control element in figure 5.10, the format allows to specify an *authentication* keyword by way of *addTypes* node. This indicates that some form of authentication is required to execute the transition of the respective control element.

[6] The full format specification can be found in appendix A.

```
curies:
  - { name: descriptors,  href: http://example.com/descriptors/ }
  - { name: iana,  href: http://alps.io/iana/relations.xml# }
  - { name: profiles, href: http://example.com/profiles/ }
profile:
  id: profiles:questionnaire
  descriptor: descriptors:questionnaire
  variant: detailed
  data:
    -
      name: analysisDelay
      syntax: string(16)
      descriptor: descriptors:analysisDelay
    -
      name: identifier
      syntax: string(8)
      descriptor: /identifier
    -
      name: title
      syntax: string(50)
      descriptor: /title
  controls:
    -
      name: append
      addTypes: authentication
      multiplicity: 1
      descriptor: descriptors:append
      argProfile: /profiles/question
      returnProfile: [profiles:question, profiles:questionLocation]
      authorization: speaker
    -
      name: question
      addTypes: transclude
      multiplicity: "*"
      descriptor: iana:item iana:collection
      returnProfile: profiles:question
```

Figure 5.10.: era-sd profile of a questionnaire

Furthermore, capabilities required to do so are indicated as values of the *authorization* node. Equivalent to *semantic aliases*, these values are meaningless on their own. The *era-sd* format expects a chosen media type to provide some way for clients to access their definitions at runtime; For instance, by means of a dedicated control element.

Embedding

Second, the format generally does not allow for embedding representations of other resources. Instead, a profile may include *transclude* as a value of the *addTypes* node to indicate that clients are recommended to access the identified resource(s) and to include their representations (see the *question* control element in figure 5.10). [7]

Thereby, the decision has two motivations: First, it avoids the caching-related problems discussed in section 3.3.3. Second, it leads to a better single responsibility in terms of representation management. This is because it encourages resource authors to provide specific profiles to be used for embedding instead of creating a separate resource.

For instance, one could have created a separate resource to represent all questions belonging to a questionnaire (i.e. server side embedding), in place of instructing clients to do so. That way, both the separately created resource as well as the *question* ones would have provided (possibly different) ways to represent a question. Hence, the according responsibility of deciding its general representation would have been distributed over the implementations underlying the two types of resources.

Focus on success case & limitations

Third, the *era-sd* format solely focuses on the success case of interactions. Consequently, there are no special elements or dedicated profiles to represent error messages. Rather, respective profiles should be created as needed. Furthermore, the format does not allow for specifying profiles returned in case executing a state transition failed.

The main reason for this decision is that, for a single success case scenario, there commonly are many more error case scenarios. Explicitly stating these within a profile would thus drastically diminish readability. In addition, a client would ideally never require this type of information provided that the success case scenario is sufficiently well described.

One down side of this decision is that *era-sd* is incapable of describing more complex validation scenarios. For example, there is no way to indicate that the *identifier* must be set in order to be allowed to *publish* a *questionnaire* explained in section 1.1.

Though you could solve this scenario by only sending the respective control element if the current resource state fulfills that requirement, there is no way to communicate this context to the client. The same is true for validation scenarios in more complex forms such as “field X must only be filled out in case box y is checked”).

5.2.3. Format domain

In order to fully exploit the possibilities provided by the *era-sd* format, a corresponding representation media type requires the following features:

- A *profile* media type parameter
- A way to assign semantics to the opaque security related information
- Ideally, generic application semantics to avoid inconsistencies with *era-sd* definitions

[7] The term *transclude* is inspired by the *UBER* format specification [Amu14].

Thereby, the *profile* media type parameter is necessary given that an *era-sd* profile may list multiple profiles from which a client can choose one being returned in case of a successful state transition by way of the *returnProfiles* node (see for instance the *append* control element shown in figure 5.10). To perform an according request using HTTP, a client would need specify an *Accept* header like this:

```
Accept:  application/vnd.era-fd+yml;profile="http://example.com/
         profiles/questionLocation"
```

As stated in section 4.4.4, only *Collection+JSON*, *HAL* and *XHTML* define this media type paramter. Given that, the first one defines all resources to be profiles, the second one has no protocol semantics and the third one only allows for GET and POST requests, this section introduces a new format to fulfill the before-mentioned requirements.

```
1  id: http://example.com/questionnaires/1
2  security: http://example.com/security
3  data:
4    analysisDelay: PT5S
5    identifier: SE101SS14
6    title: Lecture evaluation: Software Engineering 101 (SS14)
7  controls:
8    append:
9      method: POST
10     href: /questionnaires/1
11   question:
12     method: GET
13     href: [/questions/1]
```

Figure 5.11.: era-fd representation of a questionnaire

Format design

Figure 5.11 shows a representation of the *questionnaire* example based on said format. In reference to the title of this chapter as well as the *format domain*, it is called *era-fd*. [8] Essentially, the format sticks to the structure laid out by *era-sd* profiles by specifying a *data* and a *controls* node.

To handle semantics of the opaque security tokens, an *era-fd* representation may define a dedicated *security* node which is expected to link to a representation explaining said tokens. Given that controls element may identify resources in different authentication realms, the top-level *security* node only provides a default value which can be overwritten by each control element by specifying an equally named node.

In addition, all names of the *data* and *controls* node are required to match these defined by the profile linked in the media type parameter. As a result, this parameter is always required.

[8] The full format specification can be found in appendix TODO

5.3. Usage in enterprise application environments

As explained in section 3.1.3, REST generally is the more suitable as an architectural for environment the closer it resembles the Web along with its four key properties. Consequently, the decision of whether or not to employ (an implementation) of the domain ontology within enterprise application environments as well mostly depends on that correlation. Thereby, a fully committed architecture could look like as shown in figure 5.12.

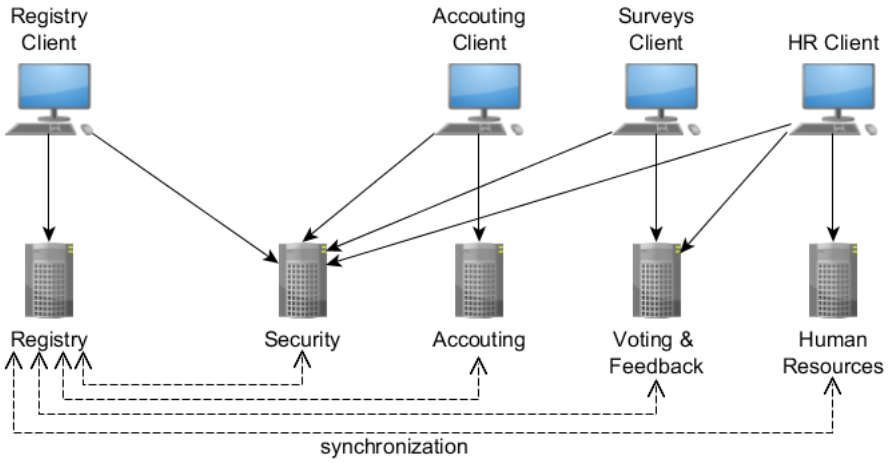

Figure 5.12.: An exemplary data-driven enterprise resource architecture

The ideal solution

Instead of providing a dedicated server per application, such an data-driven enterprise resource architecture would feature multiple servers each providing a dedicated set of resources (e.g. security, accounting or feedback & voting). On top of that, there could be additional servers providing cross-resource services (in turn realized resources) as for instance a registry for indexing and searching available resources.

When developing a new application, resources and servers would be updated and added as needed. Given that the root URL of each server would be well known within the enterprise environment, clients could access all resource servers as needed for the application the to be provided.

Altogether, such an architecture would allow for much more flexibility concerning the development of resources as the overall environment changes over time. Due to clients discovering locations and features of resource on-the-go, the latter could be changed (to some extend) and migrated between servers as needed.

Current limits

Due to budget, effort required to establish *descriptor* and *profile* definitions and other limitations, the advantages such an architecture can only take effect over time. Much

more importantly however, its power and effectiveness depends on the capabilities of clients (as explained in section 5.1). The more these can figure out autonomously, the more of the just-described benefits may take effect. Though for that to work, one must be committed towards building and using hypermedia clients for that purpose.

So, if most applications are realized as rather isolated, “silo”-solutions - which would avert most of the benefits by REST -, or if you can effectively pass on hypermedia given that not all the architectural properties discussed in section 3.1.2 are required, such an approach will most likely not pay off; In other words: Another architectural style may be more beneficial. As also stated in section 3.1.2:

> REST is not the one true architecture.

Intermediate solutions

Even if the fully committed approach may prove not to be feasible, there are two alternated ones which may yield a better trade-off.

The first one would be to implement the domain ontology only at the server side while hard-coding clients. That way, you would still be able to exploit the documentation-related benefits. Given that seemingly, many so-called REST APIs are still documented by custom made formats, the domain based approach would provide a framework for that purpose as well as help not to forget relevant design aspects.

More importantly, it further leaves the possibility to gradually develop hypermedia support at the client side as outlined in section 5.1 should the respective decision be revoked at some point in time.

As a second solution, you could add an additional tier between the *client* and *server* one. Considering that especially rich-client implementations often seem to have problems adapting to representation structures not being known before runtime, this tier would act as a *mediator* between the two worlds (see figure 5.13).

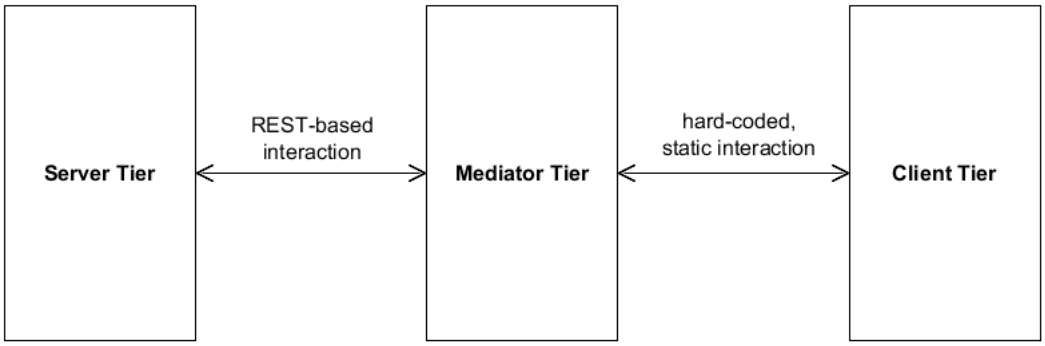

Figure 5.13.: A three tier architecture featuring a mediator tier

While communicating with a server in a generic, REST based fashion, it would communicate with the client in a rather hard-coded, specific way while internally mapping sent and received messages as required. Hence, this extended tier architecture would most likely require at least as much effort as the fully committed approach. Whether or not the former would offer more benefits than the latter however goes beyond the scope of this thesis.

6. ERA-Prototype

This chapter documents some of the experience made while implementing a proof of concept API for the example application presented in chapter 1 using the formats covered in the last chapter and Scala. To make things more realistic concerning enterprise application environments, the implementation was mostly based on rather traditional technologies such as relational database instead of document-based ones.

Given that resources and representation are central aspects of REST and the domain ontology presented in chapter 5 makes heavy use of profile, the chapter focuses on these areas.

6.1. Resources & representations

Implementing resources with staticly typed programming languages such as `Java` or `Scala` as well as a relational database poses a variety of challenges. Of these, the following covers three different problem sources: Parsing of representations, interaction with the database and (persistent) storage of resources.

6.2. Parsing representations

First of all, servers usually receive representations containing less content than the ones they originally sent. This is because clients are rarely allowed to modify control elements as these are dependent on implementation details.

```
1 id: http://example.com/questionnaires/1
2 security: http://example.com/security
3 data:
4   analysisDelay: PT5S
5   identifier: SE101SS14
6   title: Lecture evaluation: Software Engineering 101 (SS14)(updated)
7 controls:
```

Figure 6.1.: Exemplary era-fd representation of a questionnaire which could be sent along with a `PUT` request

For example, *actions* specified by a *Siren* representation only allow for sending representations consisting of data elements as described by the *fields* property. By contrast, *era-fd* uses a dedicate node named *mutable* to indicate which elements - including control ones - may be submitted to the server. For the purpose of demonstration, figure 6.1 shows an *era-fd* representation which could be sent along with a `PUT` request after having received the representation displayed in figure 5.11.

Handling representation variants

Due to these varying representations, you normally cannot use a single class at the server (or client) side to handle a specific resource; At least not without any extra effort.

One option to work around this problem would be to create specific parsing routines to include and exclude respective representation elements as needed per resource and interaction scenario. Given that most parsing frameworks rather appear to be intended for simpler, static scenarios, this is arguably not a preferable approach.

Alternatively, one could specify all elements not sent to the server to be optional. Unfortunately, this option only shifts the required efforts. This is because you have to validate each received representation after parsing to make sure it matches the scenario. Otherwise, a client might send elements to be updated which, in the worst case, would simply be ignored by the server despite sending a success response to the client. Respectively, this approach would probably not only make the code more confusing but also more error-prone.

As a result, the third option, which is to employ multiple classes to represent different scenarios, ended up to be the preferable choice. Figure 6.2 shows the respective framework as well as an example of representation classes implemented for that purpose as part of the prototype.

Implementation

In this regard, the *ControlElement* trait and object were implemented to reflect the different types of control elements found within *era-fd* representations. [1] Given that the remaining structure of *era-fd* representations is quite regular, it was possible to specify a single class to be used for all representation objects. To allow for incorporating individual `classes` which implemented the *data* and *controls* node per resource, the *Representation* `class` shown in figure 6.2 specifies two according type parameters. [2]

Based on that, each resource could be implemented by a set of classes representing the different variants of sent and received *data* and *controls* nodes. Given that the example application did not feature immutable data elements, a single `class` per resource (type) such as *Questionnaire* sufficed to represent its data elements.

Thereby, requiring each of these type `classes` to be a `case class` - as indicated by the derivation from `Product` - made parsing these object quite easy because every parsing definition ended up to be a simply, one-line statement.

6.3. Database interaction

Given that hypermedia representations usually contain control elements identifying related resources, requests reaching the database require transaction-based access to multiple tables. For instance, assembling the representation of a *questionnaire* from

[1] Admittedly, this implementation completely relies on not using another media type or multiple security realms due to exploiting the default values of the format

[2] Given that the implemented API would always use the same *era-fd security* link which would only be required for sending representations, its addition was implemented as part of parsing instead of the *Representation* `class` itself which is why it does not appear in the following code snippets.

```
sealed trait ControlElement

object ControlElement {
  case class One(method: HttpMethod, href: Uri)
  case class Many(method: HttpMethod, href: Seq[Uri]){
    require(href.nonEmpty)
  }
}

case class Representation[D <: Product, C <: Product](id: Uri, data:
    D, controls: C)

object Questionnaire {

  // data elements sent by server or client
  case class Data(identifier: String, title: String, analysisDelay:
      String)
  // control element sent by the server
  case class ServerControls(append: ControlElement.One, publish:
      ControlElement.One, question: ControlElement.Many)
  // control elements sent by the client
  case class ClientControls()
}
```

Figure 6.2.: `Class` based implementation of representations in Scala

a respective table would further necessitate accessing a table containing *question* representations to be able to correctly build up respective control elements. [3]

Provided that the database also contains cache values, *update* operations likely require re-calculating respective values of the involved representation. In case of a *append* or *delete* operation, it may even be necessary to do the same for other representations. For example, appending a *question* to a *questionnaire* would probably alter the cache values of said *questionnaire* given that it now features an additional resource identifier pointing to its *questions* (the opposite is the case when deleting a question). Overall, database interactions thus become much more complex compared to approaches which treat entities stored in respective tables rather isolated.

With regard to read requests, one could have diminished the described problems by always grouping related resources in rather artificial, intermediate collections. For example, one could have defined resources grouping all *questions* of a *questionnaire* having identifiers such as: `/questionnaires/1/questions`.

While not solving the update complexity of write-requests, such an approach would have always required one additional, rather redundant request to actually obtain, for instance, the questions of questionnaire which is why it is was discarded.

[3] Assuming that you try to stick to the relational model

Implementation

To facilitate the previously mentioned interactions, the prototype featured a set of factory methods. Of these, the one used for creating *append* methods is shown in figure 6.3. Essentially, it performs four steps:

```
@throws[ResourceNotFoundException]
@throws[CacheConditionFailedException]
protected def createAppendAndRebuildCacheBy[D <: Product, C <: Product]
(
  dao: BuildCacheAndInsert[D, L],
  rebuildCache: Uri => ServerCache
)(
  resourceId: Uri,
  clientCache: ClientWriteCache,
  buildResource: Long => ServerResource[D, L]
)(implicit session: Session): CachedAppendResult = {

  // validate conditional request
  val serverCache = findCacheBy(resourceId)
  validateWriteCache(resourceId, serverCache, clientCache)
  // insert (i.e. create the resource to be appended)
  val cachedInsertResult = dao buildCacheAndInsertBy buildResource
  // update cache values of the representation to which the created
     resource is appended
  val updatedCache = rebuildCache(resourceId)

  CachedAppendResult(cachedInsertResult.newResourceId)
}
```

Figure 6.3.: Factory for creating resource specific append methods

First, it validates the cache values provided along with the conditional *append* request. Should these be out-of-date or the resource not be found, an according exception is thrown.

Second, it uses a provided data access object (*dao*) as well as a *buildResource* method to create and insert the resource to be appended. For example, an append request executed by a questionnaire data access object would accordingly employ a question data access object to properly insert the question to be appended.

In this regard, the *buildResource* method was necessary given that the database generated numeric ids for new resources. Consequently, the information provided by the client was insufficient to be passed as a *representation* object as defined in figure 6.2.

Third, the method uses the provided *rebuildCache* method to update the cache values of the resource representation to which the newly created resource is to be appended.

Last, it returns the resource identifier of the created resource so that it can be sent back to the client; For instance using the *HTTP* `Location` header.

6.4. Persisting representations

With the just covered, increased complexity concerning database access in mind, the question of how to persist resource representations becomes rather interesting. Given that the classes used to specify representations sent to and received from clients contains things one "traditionally" would not store in the database, there are two options.

The first one would be to only store "raw" values, that is data element values as well as ids used to generate resource identifiers. However, this approach would always require additional mapping efforts and potentially classes to mediate between the resulting (conceptual) persistence and representation objects.

Given the mentioned database interaction complexity as well as due to limited time, I decided to, instead, persist all representation values in an intact fashion (which would be the second option). The advantage of this approach is that, except for the case of inserting or appending resources, it makes it relatively easy to read, update and delete representation elements as these can essentially be treated as plain-old data values.

```
case class Questionnaire(
  // id + cache values
  id: Uri, lastModified: DateTimeDuration, eTag: EntityTag,
  // data values
  analysisDelay: String, identifier: String, title: String,
  // control values
  appendMethod: HttpMethod, appendHref: Uri,
  publishMethod: HttpMethod, publishHref: Uri,
  questionMethod: HttpMethod
)
```

Figure 6.4.: `Class` based representation of a database table containing all values of a *questionnaire era-sd* representation

By contrast, storing links of what would be an *action* in *Siren* as well as *HTTP* methods of all control elements *per representation* can definitely be considered redundant given that their indicated behavior is usually hard-coded and thus does not change (see figure 6.4 for a respective class based representation of such a table).

In retrospect, the first option hence would have probably lead to an overall cleaner and clearer implementation - especially in case one would be required to support more complex variants of control elements.

6.5. Profiles

Given that the *era-fd* format heavily relies on the *profile* media type parameter, a server employing said format must extract or set these when receiving or sending representations. Probably due to the fact that only few media types make use of this feature, the framework employed for handling *HTTP* requests did not provide immediate support for extracting it.

Due to the actual code turning out not to fit on single page, the following rather briefly discusses some of the implementation related problems.

Specification means

As defined by RFC7231 [FNR14b], media types specified in the `Accept` header are first prioritized by way of a quality parameter (e.g. `q=0.7`) ranging from 1 (most preferred) to 0.001 (least preferred). This parameter can also be set to 0, to actively exclude specific media types.

After that more specific definitions precede more generic ones as demonstrated by this example (in descending order):

1. `text/plain;format=flowed`
2. `text/plain`
3. `text/*`
4. `*/*`

As shown by this example media types furthermore can be grouped into *media ranges* by *top-level type* (`*/*`) or *sub-type* (`text/*`). Respectively, one could also specify a quality value of 0 to declare an entire media range to be unacceptable.

Determining the most preferred profile

So to determine the preferred profile, you first have to check whether or not profiles supported by the server are flagged as non-acceptable (either specifically or by *media range*). If that is case, a server can use a respective *status code* to communicate the problem to the client: `406: NotAcceptable`.

Otherwise, you have to work from higher quality values to lower ones and subsequently from more specific to less specific choices to see if one of the choices matches what is supported by the server. Thereby, the process can have three outcomes:

In first case, the choices of the client are, again, too narrow so that the server would need to respond with `406: NotAcceptable`.

On the other hand, they may as well be too unspecific as for instance `application/*`. In such a case, a server may either choose a profile in place of the client or respond with a `300: MultipleChoices` status code explaining the remaining choices.

In the third case, the client submitted a choice specific enough to determine a most preferred profile which is also supported by server. As a result, the server can start processing the actual request.

7. Conclusion

This chapter concludes the thesis. First, it gives a brief summary of the content of the thesis, then it discusses potential future work.

7.1. Summary

Initially, this thesis discussed three essential topics concerning enterprise applications based on REST APIs, namely enterprise applications, API design as well as the remote procedure call interaction style. It showed that neither enterprise applications nor well designed APIs have a generally agreed definition. In addition, the thesis made suggestions why enterprise applications often end up being based on the RPC paradigm.

After that, it covered the conceptual side of REST starting with more general concepts such as the idea of an architectural style as well as the insight that REST is not a golden hammer. In addition, the thesis explained the core concepts of REST, its well-known constraints and two misconceptions. During the whole respective part, various motivations for non REST-compliant designs were identified and explained. Furthermore, the thesis argued for not using the term *pragmatic REST* due to its misleading properties.

Subsequently, the thesis discussed structure and semantics of hypermedia representations. In this regard, it also explained the central role and importance of media types. Furthermore, the respective part showed possibilities of how to work towards self-descriptive representations concerning human and machine-driven clients. Given that said part revealed several fundamental problems with concerning machine-readable documentation (limited expressiveness, reuse or extensbility), the next one presented domain ontology and approach to address these issues.

One of the main ideas of this approach was to combine the advantages of human and machine-readable documentation. Based on the ontology, the respective part also introduced a combination of one existing and to newly created formats to work towards the goals of said ontology. Furthermore, it discussed the potential concerning enterprise application architectures facilitated by employing said approach.

Last, the thesis presented some of the experiences of implementing a prototype API using the previously introduced formats. In this regard, it focused on the implementation of resource representations and profile based interaction.

7.2. Future work

Future work is imaginable in multiple aspects as explained in the following:

Standards for defining problem domain descriptors

First of all, the employment of standards for defining problem domain descriptors (see section 5.1.1) should be further evaluated. Given that standards are usually defined including a specific syntax it would be interesting to see whether or not this circumstance conflicts with the objectives of problem and solution domain. As of now, it seems that this would only be the case when having multiple, conflicting standards concerning a single descriptor.

Mapping domain

In addition, it might be interesting to investigate whether or not there are plausible scenarios requiring a media type to be defined independently on an underlying protocol. In such a case, it would be worth discussing the inclusion of the *mapping domain* in the domain ontology as reflected in section 5.1.3.

Evaluation of format stack

With regard to the three formats to implement the domain ontology, further evaluation of their employment and the approach as a whole in practice is definitely a possibility for further work. This also applies to the intermediate solutions discussed in section 5.3.

Era-sd extensions

Last but not least, the *era-sd* format could be extended in various ways. In fact, its earlier drafts contained three additional features which were removed due to lack of clarity.

The first one was a node to indicate **deprecation** of a respective element and point to a replacement within the same profile. Given that this feature somewhat promotes changing profiles more regularly and especially staticly type clients would have problems due to the profile containing an additional element, it was removed in later drafts.

The second features was to allow profile to define generic profile **type parameters** for the *argProfile* and *returnProfile* nodes similar to their purpose in programming languages. For example:

```
returnProfile:  profiles:result@profiles:question
```

That way, authors could have specified a single profiles which could have been used in multiple scenarios being identical except for specific “profile types”.

The third features involved a specification that would have allowed clients to **change semantics** of elements. For example, API administrators might be interested in being able to change the authorization values of a state transition by sending respective requests to the server. However, there is no way in the final draft of *era-sd* to specify interactions such as this.

Due to limited time, both these last features could not be fully evaluated which is why they were removed from the final draft. In the future these and other feature might very well be re-added to the format.

A. ERA-SD

Overview

Name:	*era-sd*
Potential media type name:	`application/vnd.era-sd+yml` [1]
Current Version:	0.9

Media type parameters

Name	Description	Required	Template	Example
version	Indicates the *era-sd* format version used in this document	True	String	*version=0.9*

Remarks

- This format specification makes use of the compact URI format defined by [BM10]
- In the following a document using this format is referred to as *profile*
- The following employs terms *node*, *scalar*, *sequence* and *mapping* as defined by [BEN15]
- The following makes references to Unified Modeling Language (UML)

[1] *XML* and *JSON* variants analogous

A.1. Maximum skeleton

```
curies:
  -
    name:
    href:
profile:
  id:
  descriptor:
  variant:
  data:
    -
      name:
      syntax:
      descriptor:
      multiplicity:
      mutable:
  controls:
    -
      name:
      addTypes:
      descriptor:
      multiplicity:
      mutable:
      argProfile:
      returnProfile:
      authorization:
```

Figure A.1.: Maximum skeleton of an *era-sd* profile showing all specification possibilities

A.2. Nodes

curies

Description:	A *sequence* of compact *URI* definitions available in this profile. Each item is a mapping consisting of a *name* and a *href* node describing *prefix* and *reference* of the respective *CURIE*.
Rationale:	Increases readability.
Required:	False
Default:	-
Template:	-
Example:	see figure A.2

```
1 curies:
2   -
3     name: descriptors
4     href: http://example.com/descriptors/
5   -
6     name: iana
7     href: http://alps.io/iana/relations.xml#
```

Figure A.2.: Example of a curies sequence within an era-sd profile

profile

Description:	Mapping containing the actual profile. May contain the following nodes: *id*, *semantics*, *variant*, *data*, *controls*
Rationale:	-
Required:	True
Default:	-
Template:	-
Example:	-

id

Description:	*URL* of this profile.
Rationale:	Required for unique identification and reference of profiles.
Required:	True
Default:	-
Template:	*URI* or *CURIE*
Example:	`http://example.com/profiles/questionnaire`, `profiles:questionnaire`

descriptor

Description:	URL of the descriptor refined by this profile. When appearing as a child of the *profile* node, it identifies the primary, composite *descriptor* of this profile. When appearing as a child of an item of the *data* node, it identifies a component or extension *descriptor* refined by the data element as explained in section 5.1.2. When appearing as a child of an item of the *controls* node, it identifies ideally a single, but possibly multiple link relation type *descriptor* refined by the respective control element.
Rationale:	Reuse by reference of *descriptor* definitions
Required:	True
Default:	-
Template:	*URI*, *CURIE* or relative *URI* (to identify a component descriptor)
Example:	`http://example.com/descriptors/questionnaire`, `descriptors:questionnaire`, `/analysisDelay`

variant

Description:	URL of a descriptor characterizing this profile in context of multiple variants of the same resource.
Rationale:	Used to help clients with selecting one of multiple profiles (e.g. preview vs. detailed, small vs. large, mobile vs. desktop).
Required:	False
Default:	-
Template:	*URI* or *CURIE*
Example:	`http://example.com/relations/preview`, `relations:detailed`

data

Description:	A sequence of data items defined by this profile. Each item may contain the following child nodes: *name*, *syntax*, *descriptor*, *multiplicity*, *mutable*.
Rationale:	Separated from control elements due to different child nodes and default values.
Required:	True
Default:	-
Template:	-
Example:	see figure A.3

```
data:
  -
    name: title
    syntax: string(50)
    descriptor: /title
    multiplicity: 1
    mutable: true
```

Figure A.3.: Example of a *data* node within an *era-sd* profile

controls

Description:	Contains the data items of this profile. Each item may contain the following child nodes: *name*, *type*, *rel*, *argProfile*, *returnProfile*, *returnType*, *authorization*.
Rationale:	Separated from data elements due to different child nodes and default values.
Required:	True
Default:	-
Template:	-
Example:	see figure A.4

```
controls:
  -
    name: append
    addTypes: auth
    multiplicity: 1
    descriptor: basic:append
    argProfile: /profiles/question
    returnProfile: profiles:result@profiles:question
    authorization: speaker
    mutable: false
```

Figure A.4.: Example of a *controls* node within an *era-sd* profile

name

Description:	Identifies the parent node within a sequence of identical items such as *curies*, *data* or *controls*).
Rationale:	Used to refer to curies within a profile as well as to data and control items from a representation.
Required:	True
Default:	-
Template:	*camel case* string
Example:	analysisDelay

syntax

Description:	Defines the basic syntax of a data element value.
Rationale:	Provides low level parsing information which can be processed without understand the semantics of the data element.
Required:	True
Default:	-
Template:	Must be on of these alternatives:

boolean	true or false
number	basically of double
values(<val1>, <val2>, ...)	enumeration of allowed values
string(N)	string of length N
regex(<pattern>)	a regular expression

Example:	string(10), values(small, medium, large)

multiplicity

Description:	Describes the multiplicity of a *data* or *controls* item values.
Rationale:	Representations often refer to many resources providing an identical profiles or may contain multiple data values of the same type. For instance, a questionnaire may link to many questions.
Required:	True
Default:	1
Template:	*UML* multiplicity values
Example:	*, +, 1, 0..1

mutable

Description:	Describes whether or a client is allowed to change the value of the respective *data* or *controls* item.
Rationale:	Control element values are not necessarily always immutable just as data element values do not need to mutable.
Required:	false
Default:	*true* when appearing as a child of a *data* item. *false* when appearing as a child of a *controls* item.
Template:	Boolean
Example:	true

addTypes

Description:	Lists one more additional type characterizing the state transition described by the *controls* item it appears in.
Rationale:	Enriches the state transition type definition from the *problem domain*. Possible future extension point.
Required:	false
Default:	-
Template:	*Scalar* value or *sequence* of *scalar* values. May consist of the following values:

authentication	Triggering the state transition requires some sort of authentication. A respective representation should provide information of how to get it
transclude	Clients are recommended to automatically retrieve a representation using the respective control element and to include in the current representation [2]

Example: `[authentication, transclude]`

argProfile

Description:	*URL* of a profile describing the representation to be sent to trigger the respective state transition. Elements not being mutable must be excluded.
Rationale:	A more generic approach compared to providing specialized elements such as the *fields* property of a *Siren* representation.
Required:	false (some transition do not need an input representation)
Default:	-
Template:	*URI* or *CURIE*
Example:	`http://example.com/profiles/question`, `profiles:question`

[2] The term *transclude* was inspired by the *UBER* format specification [Amu14]

returnProfile

Description:	*URL(s)* of one or more profiles describing possible response representations from which the client is to choose
Rationale:	A more generic approach compared to providing specialized elements such as the *fields* property of a *Siren* representation.
Required:	false (in case the respective media type does not support profiles)
Default:	-
Template:	Either a single *scalar* represented by an *URI* or *CURIE*, or a *sequence* of such *scalar* nodes
Example:	`http://example.com/profiles/question`, `profiles:question`

authorization

Description:	One or more opaque tokens representing capabilities such as roles or permissions required to trigger the state transition described by the *controls* item it appears in
Rationale:	Allows for specification without being dependent on implementation details
Required:	false
Default:	-
Template:	Either a single *scalar* represented by an string token, or a *sequence* of such *scalar* nodes
Example:	author, admin

B. ERA-FD

Overview

Name:	*era-fd*
Potential media type name:	`application/vnd.era-fd+yml` [1]
Current Version:	0.9

Media type parameters

Name	Description	Required	Template	Example
version	Indicates the *era-fd* format version used in this document	True	String	version=0.9
profile	References a profile describing the contents of a request or response representation	False [2]	*URI*	`profile="<URI>"`

Remarks

- This format specification makes use of the compact URI format defined by [BM10]
- The following employs terms *node, scalar, sequence* and *mapping* as defined by [BEN15]

B.1. Maximum skeleton

B.2. Nodes

id

Description:	*URL* of the resource underlying this representation.
Rationale:	Required due to resource identification constraint.
Required:	True
Default:	-
Template:	*URI*
Example:	`http://example.com/questionnaires/1`

[1] *XML* and *JSON variants analogous*

[2] Must not be required to allow for more general requests such as `Accept: application/vnd.era-fd+yml` which a client would likely make when starting to interact with an API

```
1  id:
2  security:
3  data:
4    <name>: <value>
5  controls:
6    <name>:
7      method:
8      href:
9      security:
10     argType:
11     returnType:
```

Figure B.1.: Maximum skeleton of an *era-fd* profile showing all specification possibilities

security

Description:	*URL* of a resource providing security information concerning the entire representation or control element. The node found at the top level serves as default value and can be overwritten by additional specifications as part of control elements.
Rationale:	Assign semantics to opaque security values for instance provided by an *era-sd* profile
Required:	False (an API might not have any security related features)
Default:	-
Template:	*URI*
Example:	`http://example.com/security`

data

Description:	A mapping of name-value pairs. Each name-node should match the name of data element described in a profile referenced by the *profile* media type parameter. Each value node must match the semantics identified in said profile.
Rationale:	Groups all data elements.
Required:	True
Default:	-
Template:	-
Example:	see figure B.2

```
1  data:
2    analysisDelay: PT5S
```

Figure B.2.: Example of the *data* node of an *era-fd* representation

controls

Description:	A mapping of name-value pairs. Each name-node should match the name of control element described in a profile referenced by the *profile* media type parameter. Each value must in turn be a mapping with may have the following child nodes: *method*, *href*, *security*, *argType*, *returnType*.
Rationale:	Groups all control elements
Required:	True
Default:	-
Template:	-
Example:	see figure B.3

```
1 controls:
2   append:
3     method: POST
4     href: /questionnaires/1
5     argType: application/vnd.era-fd+yml
6     returnType: application/vnd.era-fd+yml
```

Figure B.3.: Example of the *data* node of an *era-fd* representation

method

Description:	Specifies the *HTTP* method to be used for the respective control element.
Rationale:	Specification of protocol semantics
Required:	True
Default:	-
Template:	One of the following *HTTP* methods: `GET, DELETE, PATCH, POST, PUT`
Example:	`GET`

href

Description:	Specifies the identifier of the resource providing the state transition described by the control element.
Rationale:	Required due to resource identification constraint.
Required:	True
Default:	-
Template:	*URI*, relative URI based on the *domain name* of the value of the *id* node or a *URI template*. In the last case, values of elements specified by a profile such as *era-sd* which match the variable names of the template must be inserted into the *URI* instead of being send in the payload
Example:	`http://example.com/questionnaires/1`, `/questionnaires/1`, `/questionnaires/identifier`

argMediaType

Description:	One of more full media type names which may used to submit the request
Rationale:	Integration with other media types
Required:	False
Default:	If not specified, clients are expected to use `application/vnd.era-fd+yml` including the respectively set *profile* parameter as for instance stated by an *era-sd* profile
Template:	Media type name definition as explained in RFC7231 [FNR14b].
Example:	-

returnMediaType

Description:	One of more full media type names which may are to expected and/ or requested to be used by the server for the response message.
Rationale:	Integration with other media types.
Required:	False
Default:	If not specified, clients can expect to receive a representation based on `application/vnd.era-fd+yml` including the respectively set *profile* parameter as for instance stated by an *era-sd* profile.
Template:	Media type name definition as explained in RFC7231 [FNR14b].
Example:	-

C. Bibliography

[14] *JSON-LD*. 2014.
URL: http://json-ld.org/ (visited on 09/02/2014).

[15] *Printing a DOM document in Java*. 2015.
URL: http://stackoverflow.com/questions/2325388/java-shortest-way-to-pretty-print-to-stdout-a-org-w3c-dom-document (visited on 01/14/2015).

[AK10] A. Al Kalbani and Kinh Nguyen. „Designing flexible business information system for modern-day business requirement changes“. In: *Software Technology and Engineering (ICSTE), 2010 2nd International Conference on*. Vol. 2. 2010, DOI: 10.1109/ICSTE.2010.5608774.

[Amu10] M. Amundsen. *Maze+XML*. 2010.
URL: http://amundsen.com/media-types/maze/ (visited on 01/07/2015).

[Amu11] M. Amundsen. *Collection+JSON*. 2011.
URL: http://amundsen.com/media-types/collection/format/.

[Amu13] M. Amundsen. *Describing the Possible with ALPS: REST FEST*. Greenville SC USA, 2013.

[Amu14] M. Amundsen. *Uniform Basis for Exchanging Representations (UBER)*. 2014.
URL: https://rawgit.com/mamund/media-types/master/uber-hypermedia.html (visited on 12/10/2014).

[ARF14] M. Amundsen, L. Richardson, and M. Foster. *Application-Level Profile Semantics (ALPS): draft-amundsen-richardson-foster-alps-00*. 2014.
URL: http://tools.ietf.org/html/draft-amundsen-richardson-foster-alps-00 (visited on 12/09/2014).

[BEN15] O. Ben-Kiki, C. Evans, and I. döt Net. *YAML Ain't Markup Language (YAML™) Version 1.2*. 2015.
URL: http://www.yaml.org/spec/1.2/spec.html (visited on 01/13/2015).

[Ber06] T. Berners-Lee. *Linked Data - Design Issues*. 2006.
URL: http://www.w3.org/DesignIssues/LinkedData.html (visited on 08/07/2014).

[Blo07] Joshua Bloch. *How to Design a Good API and Why it Matters*. Ed. by Google Tech Talks. 2007.
URL: https://www.youtube.com/watch?v=heh4OeB9A-c (visited on 06/17/2014).

[BM10] M. Birbeck and S. McCarron. *CURIE Syntax 1.0: A syntax for expressing Compact URIs*. 2010.
URL: http://www.w3.org/TR/curie/ (visited on 01/11/2015).

[CA08] Krzysztof Cwalina and Brad Abrams. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*. 2nd. Addison-Wesley Professional, 2008. ISBN: 0321545613.

[Cum02] Fred A. Cummins. *Enterprise Integration: An Architecture for Enterprise Application and Systems Integration*. New York, NY, and USA: John Wiley & Sons, Inc, 2002. ISBN: 0471400106.

[DLS10] L. Dusseault, L. Lab, and J. Snell. *The Patch Method for HTTP: RFC5789*. 2010.
URL: http://tools.ietf.org/html/rfc5789.

[FB96a] N. Freed and N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies: RFC2045*. 1996.
URL: http://tools.ietf.org/html/rfc2045 (visited on 12/15/2014).

[FB96b] N. Freed and N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types: RFC2046*. 1996.
URL: http://tools.ietf.org/html/rfc2046 (visited on 12/11/2014).

[Fie00] Roy Thomas Fielding. „Architectural Styles and the Design of Network-based Software Architectures“. PhD thesis. 2000.

[Fie07] Roy T. Fielding. *The Rest of REST*. 2007.
URL: http://roy.gbiv.com/talks/200709%5C_fielding%5C_rest.pdf (visited on 08/06/2014).

[Fie08] Roy T. Fielding. *REST APIs must be hypertext-driven*. 2008.
URL: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven (visited on 09/04/2014).

[FKH13] N. Freed, J. Klensin, and T. Hansen. *Media Type Specifications and Registration Procedures: RFC6838*. 2013.
URL: http://tools.ietf.org/html/rfc6838 (visited on 12/11/2014).

[FNR14a] Roy T. Fielding, M. Nottingham, and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Caching: RFC7234*. 2014.
URL: https://tools.ietf.org/html/rfc7234 (visited on 11/14/2014).

[FNR14b] Roy T. Fielding, M. Nottingham, and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content: RFC7231*. 2014.
URL: https://tools.ietf.org/html/rfc7231 (visited on 11/26/2014).

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Boston, MA, and USA: Addison-Wesley Longman Publishing Co., Inc, 2002. ISBN: 0321127420.

[FT02] Roy T. Fielding and Richard N. Taylor. „Principled Design of the Modern Web Architecture“. In: *ACM Trans. Internet Technol.* 2.2 (2002), pp. 115–150. ISSN: 1533-5399. DOI: 10.1145/514183.514185.
URL: http://doi.acm.org/10.1145/514183.514185.

[GMP15] GMPG. *Xhtml Meta Data Profiles*. 2015.
URL: http://gmpg.org/xmdp/ (visited on 01/14/2015).

[Hen07] Michi Henning. „API Design Matters“. In: *Queue* 5.4 (2007), pp. 24–36. ISSN: 1542-7730. DOI: 10.1145/1255421.1255422. URL: http://doi.acm.org/10.1145/1255421.1255422.

[Hyu+09] Hyuck Han et al. „A RESTful Approach to the Management of Cloud Infrastructure“. In: *Cloud Computing, 2009. CLOUD '09. IEEE International Conference on*. 2009, pp. 139–142. DOI: 10.1109/CLOUD.2009.68.

[IAN14] IANA. *Link Relations*. 2014. URL: http://www.iana.org/assignments/link-relations/link-relations.xhtml (visited on 09/29/2014).

[Jim] Jim Hendler. „ESWC 2011 Keynote Speech: "Why the Semantic Web will never work"“. In: URL: http://videolectures.net/eswc2011%5C_hendler%5C_work/ (visited on 06/17/2014).

[Kel13] Mike Kelly. *HAL - Hypertext Application Language*. 2013. URL: http://stateless.co/hal%5C_specification.html (visited on 10/09/2014).

[Krc10] Helmut Krcmar. *Informationsmanagement*. 5th ed, completey revised and extended. Berlin and Heidelberg: Springer-Verlag, 2010. ISBN: 978-3-642-04285-0.

[Lac13] Kevin Lacker. *How to design a great APIs*. Parse Developer Days, 2013. URL: https://www.youtube.com/watch.

[Mic14] Microsoft Developer Network. *What is an Enterprise Application?* Ed. by Microsoft Developer Network. 2014. URL: http://msdn.microsoft.com/en-us/library/aa267045(v=vs.60).aspx (visited on 08/21/2014).

[Mic15] Microsoft. *The OSI Model's Seven Layers Defined and Functions Explained*. 2015. URL: https://support.microsoft.com/kb/103884 (visited on 01/14/2015).

[MN90] Rolf Molich and Jakob Nielsen. „Improving a Human-computer Dialogue“. In: *Commun. ACM* 33.3 (1990), pp. 338–348. ISSN: 0001-0782. DOI: 10.1145/77481.77486. URL: http://doi.acm.org.eaccess.ub.tum.de/10.1145/77481.77486.

[Moo10] Jon Moore. *Hypermedia APIs: The Rest of REST*. 2010. URL: http://vimeo.com/20781278 (visited on 11/05/2014).

[MSW09] J. Mangler, E. Schikuta, and C. Witzany. „Quo vadis interface definition languages? Towards a interface definition language for RESTful services“. In: *Service-Oriented Computing and Applications (SOCA), 2009 IEEE International Conference on*. 2009, pp. 1–4. DOI: 10.1109/SOCA.2009.5410459.

[Not] M. Nottingham. *HTTP/2.0: Challenges and Opportunities*. URL: https://www.mnot.net/talks/http2-expectations/%5C#/.

[Not10] M. Nottingham. *Web Linking: RFC5988*. 2010. URL: http://tools.ietf.org/html/rfc5988 (visited on 09/29/2014).

[Not13] M. Nottingham. *Caching Tutorial for Web Authors and Webmasters*. 2013. URL: https://www.mnot.net/cache%5C_docs (visited on 12/15/2014).

[NS05] M. Nottingham and R. Sayre. *The ATOM Syndication Format*. 2005. URL: https://tools.ietf.org/html/rfc4287 (visited on 01/14/2015).

[Ora12] Oracle. *Overview of Enterprise Applications: JEE6 Documentation*. Ed. by Oracle. 2012. URL: http://docs.oracle.com/javaee/6/firstcup/doc/gcrky.html (visited on 08/21/2014).

[Ora14] Oracle. *An Overview of RMI Applications*. 2014. URL: http://docs.oracle.com/javase/tutorial/rmi/overview.html (visited on 09/05/2014).

[Pos94] J. Postel. *Media Type Registration Procedure: RFC1590*. 1994. URL: http://tools.ietf.org/html/rfc1590 (visited on 12/15/2014).

[RAR13] L. Richardson, M. Amundsen, and S. Ruby. *Restful Web Apis*. Oreilly & Associates Incorporated, 2013. ISBN: 9781449358068.

[RSK12] Dominik Renzel, Patrick Schlebusch, and Ralf Klamma. „Today's Top "RESTful" Services and Why They Are Not Restful". In: *Proceedings of the 13th International Conference on Web Information Systems Engineering*. WISE'12. Berlin and Heidelberg: Springer-Verlag, 2012, pp. 354–367. ISBN: 978-3-642-35062-7. DOI: 10.1007/978-3-642-35063-4_26. URL: http://dx.doi.org/10.1007/978-3-642-35063-4%5C_26.

[SHB14] Z. Shelby, K. Hartke, and C. Bormann. *The Constrained Application Protocol (CoAP): RFC 7252*. 2014. URL: https://datatracker.ietf.org/doc/rfc7252/ (visited on 10/09/2014).

[Sin14] R. Sinnema. *The State of REST*. 2014. URL: http://securesoftwaredev.com/2014/11/10/the-state-of-rest/ (visited on 11/23/2014).

[Sne14] J. Snell. *HTTP Link and Unlink Methods: draft-snell-link-method-11*. 2014. URL: https://datatracker.ietf.org/doc/draft-snell-link-method/ (visited on 01/14/2015).

[Soa92] Patr\'ıcia Gomes Soares. „On Remote Procedure Call". In: *Proceedings of the 1992 Conference of the Centre for Advanced Studies on Collaborative Research - Volume 2*. CASCON '92. IBM Press, 1992, pp. 215–267. URL: http://dl.acm.org.eaccess.ub.tum.de/citation.cfm?id=962260.962276.

[Swi14] K. Swiber. *Siren: A hypermedia specification for representing entities*. 2014. URL: https://github.com/kevinswiber/siren (visited on 12/20/2014).

[Tra95] Will Tracz. *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*. Boston, MA, and USA: Addison-Wesley Longman Publishing Co., Inc, 1995. ISBN: 0-201-63369-8.

[Ulr10] William Ulrich. *How Business Rules Relate to Business Processes from a Business Persons Point of View*. Minneapolis, 2010.

[Vin08] S. Vinoski. „Demystifying RESTful Data Coupling“. In: *Internet Computing, IEEE* 12.2 (2008), pp. 87–90. ISSN: 1089-7801. DOI: 10.1109/MIC.2008.33.

[W3C] W3C. *Cool URIs don't change*.
URL: http://www.w3.org/Provider/Style/URI.html (visited on 03/11/2014).

[W3C04] W3C Technical Architecture Group. *Architecture of the World Wide Web, Volume One*. 2004.
URL: http://www.w3.org/TR/2004/REC-webarch-20041215/ (visited on 10/31/2014).

[WFV00] I. Wijegunaratne, G. Fernandez, and J. Valtoudis. „A federated architecture for enterprise data integration“. In: *Software Engineering Conference, 2000. Proceedings. 2000 Australian*. 2000, pp. 159–167. DOI: 10.1109/ASWEC.2000.844573.

[Wil13] E. Wilde. *The 'profile' Link Relation Type: RFC6906*. 2013.
URL: http://tools.ietf.org/html/rfc6906 (visited on 12/31/2014).

[Wor14] Wordnik. *Swagger 2.0*. 2014.
URL: http://swagger.io/ (visited on 01/14/2015).

[WP11] E. Wilde and C. Pautasso. *REST: From Research to Practice*. Springer, 2011. ISBN: 9781441983039.

[WPR10] J. Webber, S. Parastatidis, and I. Robinson. *REST in Practice: Hypermedia and Systems Architecture*. Theory in practice series. O'Reilly Media, 2010. ISBN: 9781449396923.