Institute of Architecture of Application Systems

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Master's Thesis Nr. 3703

REST compliant clients for REST APIs

Mustafa Jaber

Studiengang:	INFOTECH
Prüfer:	Prof. Dr. Frank Leymann
Betreuer:	Dipl.-Inf. Florian Haupt
begonnen am:	18.06.2014
beendet am:	18.12.204
CR-Klassifikation:	D.2.2

Declaration

I hereby declare that the work presented in this thesis is entirely my own.
I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations.
Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before.
The electronic copy is consistent with all submitted copies.

Mustafa A. Jaber
Stuttgart, December 18, 2014

Abstract

In today's distributed systems, REST services play a centric role in defining applications' architecture. Current technologies and literature focus on building server-side REST applications. But they fail to build generic and REST compliant client solutions. Therefore, most offered services and especially client applications rarely comply to the constraints that constitute the REST architecture. In this thesis, the architecture of a new generic framework for building REST compliant client applications is introduced. In addition, a new description language that conforms to REST's constraints and helps reduce development time is presented. We describe in this work the building-blocks of the proposed solutions and show a software implementation of a library that leverages the solutions' architectures. Using the proposed framework and description language, client applications that conform to the full set of REST's constraints can be built in an easy and optimized way. In addition, REST service providers can rely on the proposed description language to eliminate the complexity of repetitively building customized solutions for different technologies or platforms.

Acknowledgment

I would like to express my gratitude and appreciation to Prof. Frank Leymann and especially my supervisor Dip.-Inf. Florian Haupt from the Institute of Architecture of Applications Systems (IAAS) at the University of Stuttgart who believed in my vision towards achieving the results of this work. Your encouragement, guidance and advice have been priceless while accomplishing the goals of my tasks.

I would like also to express my deep appreciation to my family. Words are not sufficient to express how thankful I am for the prayer and sacrifices of my mother, my father, my brother, and my sisters. I would like also to thank my uncle Ahmad, without your support I would not have had the chance to see this moment.

I would like also to thank all the people who have taught me and helped me achieve what I have achieved. Special thanks to my friends and especially Monir, your encouragement has been always a true motivation for me.

Contents

List of Figures

Chapter 1

Introduction

Development of distributed services based on the REST architectural style has been widely adopted by service providers. The success of the REST architectural style could be related to the constraints that restrict the way service components should be developed. Those constraints (see section 2.3), when applied, could ensure improvements in the overall distributed system. For instance, network performance, scalability of server components, and the overall architecture's simplicity could significantly benefit from applying REST's constraints.

REST compliance could be achieved when applying the architectural constraints on the different components of the REST distributed system. Unfortunately, many REST services providers claim to support the complete architectural constraints of REST services but they, however, do not apply the entire set. As a result, those services lack many of the advantages that the architectural constraints offer. For instance, few number of service providers and consumers apply the Hypermedia As The Engine Of Application State (HATEOAS) constraint. Therefore, the REST's advantage of building loosely-coupled client-server applications is omitted.

Most of today's technologies that help develop REST applications focus on the server side of the architecture [1]. For instance, server side technologies tend to build REST services using simple and modern software techniques such as annotation-based resources and routing of client's requests to the appropriate business logic. However, limited number of REST services, and therefore technologies, adopted the concept of HATEOAS in which response messages include information about the next possible interactions. On the other hand, client-based technologies which support developing full REST compliance applications are still not well investigated.

Developing client applications that consume REST services is claimed to be an easy and straightforward activity. In fact, it is a repetitive, complex and sometimes frustrating activity due to the amount of information that have to be dealt with before start focusing on the actual client's business goals. Todays REST client developers rely heavily on the documentation that should be provided along with the service. Understandability of the service heavily depends on the way the documentation is written, and how clear, updated and detailed it is. To cope with the documentation problem, service providers tried to naively solve the problem via providing a set of client libraries for each technology (i.e. Programming Lan-

guage) that could use the service. These client libraries are often developed independently for each technology. As a result, development time and cost are often much higher. Moreover, all the simplicity restrictions and debate of RPC-style services that REST tries to solve have been introduced again by using RPC-like, documentation-dependent, technology-tightly-coupled and service-provider-developed client libraries.

1.1 Requirements

Based on the issues that face REST services' application developers mentioned earlier, a set of requirements have been developed to serve as the initial point of solving these issues. These requirements are:

1- Eliminate the need for referring to human-based documentation to understand how to interact with REST service's resources.

2- Eliminate the need for developing technology-dependent REST service's client libraries.

3- Design a generic solution for building REST's compliant client applications which leverage the complete set of REST's constraints.

Many solutions could be developed to cope with the issues that face the developers of REST client applications. In this work, a discussion about a generic framework for developing compliant REST client applications is to be established. In addition, an easy and usable approach for describing REST service's resources is to be developed. This approach could help reduce the complexity of developing repetitive tasks that could be faced when interacting with naively documented REST services. In addition, it helps service providers reduce time-to-market costs via avoid building technology-dependent client libraries.

To be able to design solutions for the mentioned requirements, the current solutions, their comparison and deficiencies have to be investigated. In addition, to design a generic solution of client applications, the types of clients have to be investigated including their functional and behavioral forms.

The description of REST services has been already established by many technology vendors and researchers. However, many of these technologies and description languages do not conform to REST's constraints. In this work, a discussion about a compliant REST description language should be established. This description language should apply the constraints of REST, enable better discoverability and understandability than current documentation-centric solutions, and help generate REST services client executables.

To eliminate the process of developing technology-dependent REST service's client libraries, reduce the time-to-market costs, and ensure applying REST's constraints, a code-generation engine could be developed. This engine should be able to discover, define resources' and transport system's data and metadata, and gen-

erate dedicated resource's code as a set of technology-dependent executables. For example, in Object Orientation, these executables could be represented as a set of classes that the developer can use to interact with the server. The code-generation engine could be based on the service's resources description language that is to be investigated.

To reduce the complexity of building REST client applications that conform to the complete set of REST's constraints, a compliant and generic REST's client framework should be investigated and designed. This framework should be able to consume compliant REST services, support different message representations, and offer extensibility features for supporting additional message representations. In addition, this framework could be able to work with the generated code that is to be developed using the above mentioned code generation engine.

In addition to all the functional requirements that are mentioned above, the technologies that are to be designed should consider the Development Experience as well; this means that in addition to the compliant and more useful functionalities that are to be developed compared to current solutions, developers of REST applications should be able to build their solutions in an easy, intuitive and functional manner.

1.2 Related Work

Distributed applications have been rapidly evolving to meet the requirements and advantages that a distributed system offers to its users. As a result, many distributed applications have adopted the architecture of REST services since its introduction. This was due to REST's simplicity and scalability advantages. The rapid development and increasing usage of REST services have led to advancements in the tools and frameworks that are used to build such systems. In this section, some of the technologies and established literature are discussed.

1.2.1 Existing Description Languages

Different technologies have been developed to describe REST services and resources to reduce development time. In this section, some of the description languages that are mostly known in the REST industry are discussed.

WSDL 2.0: following WSDL 1.1 [2] as the first formalization of web services, WSDL 2.0 introduced major modifications to become W3C's web service recommendation. Among other changes, WSDL 2.0 supports the full set of REST-related HTTP's operations (POST, GET, PUT, DELETE) instead of only POST as in WSDL 1.1. The introduction of the new operations opened new possibilities not only to describe SOAP services, but it also enables the description of HTTP-based REST services. However, when describing REST services in WSDL 2.0, a core concept of the REST architecture which is being resource-oriented is ignored. WSDL describes services in terms of their functionalities, that is, the exposed interfaces represent procedures that are offered on the network. On the other hand, REST services expose a finite set

of resources that are manipulated through standardized CREATE, READ, UPDATE, and DELETE operations.

WADL: having WSDL in mind as a description language for SOAP-based web services, Web Application Description Language (WADL) [3] was developed to offer a resource-oriented description language for REST services. WADL enables the representation of REST service's resources and their data. However, WADL defines tightly coupled resource identification via its URI. This restricts WADL from supporting the REST essential HATEOAS constraint. In addition, WADL supports only HTTP as the Transport System of REST services.

SWAGGER and RAML: both description languages offer modern technologies to help build REST services. SWAGGER [4] and RAML [5] are centered on resource-orientation. That is, they focus on the service's resources and their data representation. However, they are similar to WADL in describing tightly coupled URIs to identify the service's resources. SWAGGER and RAML describe REST services in YAML [6] as their main metadata language.

1.2.2 Existing REST Client Frameworks

As it was mentioned before, limited number of technologies and literature focus on the client side of REST applications. In this section, some of the related contributions are highlighted which include server and client technologies and literature.

H. Cho and S. Ryu [7] introduced a new approach for developing WEB APIs based on JavaScript. Their work defines an approach for transforming REST services into JavaScript WEB APIs to make it more accessible to client applications developers. However, this approach is only available using JavaScript.

JAX-RS [8] is a standard specifications for building REST applications using Java as a programming language. However, it does not provide any special facilities to help build fully-compliant REST applications especially on the client side. What it offers instead is a set of helper functionalities to integrate URIs with resources' representations [9]. This basically does not conform to what R. Fielding stresses on in his blog post [10] about the way REST services should be designed; REST services should be based on hypermedia, and any web-enabled service that does not apply hypermedia should not be called REST.

M. Amundsen explains how to build hypermedia APIs in [11]. He discusses how hypermedia Media Types could be designed using different markup languages. These media types can be used by REST client applications to understand the possible transitions and functionalities that could be performed next.

Hydra [12] provides a lightweight vocabulary to help create generic client applications based on the semantics the Hydra project supports. Using Hydra's vocabulary, REST services respond to the client's request with the possible transitions

that the client application can request.

1.3 Outline

Chapter 2 - Background This chapter serves as the infrastructure for most of the concepts introduced in this work. The definition of distributed systems, their applications and architecture are explained. Moreover, this chapter introduces the layer that abstracts the complexities of building distributed applications which is called the middleware. In addition, examples of middleware systems for distributed systems are introduced. These examples include WSDL/SOAP services as well as REST services which is the focus of this work. The concepts, architectural constraints, and some of the related work of REST services are also discussed in this chapter.

Chapter 3 - REST Clients As one of the goals of this work which is to build REST compliant client applications, a study of the current behavior and types of client applications has to be done. In this chapter, a set of REST client applications type are introduced. The behavior of these applications and their functional requirements are introduced as well. These types serve as the basis for developing a solution to build generic REST client applications.

Chapter 4 - REST Description Language Based on the requirements introduced in section 1.1, this chapter introduces a description language for REST services. The REST description language (RESTDL) describes REST services' resources and their interconnections in an easy, human- and machine-understandable manner. RESTDL helps build server and client applications that apply the REST architectural constraints introduced in Chapter 2. The architecture, building blocks and advantages of RESTDL are discussed in this chapter.

Chapter 5 - HypREST This chapter introduces a new generic solution for developing REST compliant client applications. In this chapter, a discussion about the architecture of a generic framework is introduced. In addition, the components, their functionality, and interconnection with other components are introduced. Using the introduced framework helps build hypermedia client applications in an easy and usable way.

Chapter 6 - Implementation This chapter introduces prototype implementations of libraries that leverage the concepts introduced in this work, mainly RESTDL and HypREST. An integration between RESTDL and HypREST is also introduced. In addition, examples of how REST client applications could be build are also introduced.

Chapter 7 - Conclusion This chapter discusses what has been completed in this work. In addition, it maps the introduced solutions to the requirements of this work.

Chapter 2

Background

This chapter provides an overview of some of the current challenges and solutions that are faced when building modern software applications. In addition, it introduces the concepts, background information and challenges of REST services. This chapter also introduces the concepts and definitions that are needed for the discussion of this work.

2.1 Distributed Systems

A distributed system can be described as a computer system which consists of at least two different computers that are connected over a network. The set of connected computers should be perceived and function as a single computer unit. A computer in the context of a distributed system should include at least a processing unit and a shared memory unit. Figure 2.1 depicts the architecture of a possible distributed system which consists of multiple electronic devices that are connected over different network channels to consume offered applications on the Internet.

Distributed systems help cope with modern computing challenges that require high performance and scalable processing requirements. These requirements can be covered via the architecture that distributed systems offer. This distributed architecture avoids building applications which are built of centralized computing resources. Instead, the architecture is based on the connection of computer resources over a network. As a result, this architecture may help scale and extend distributed applications by adding additional computational and storage resources dynamically. In addition, distributed systems could offer highly-available application's resources (data and functionalities). This is due to its capability to replicate application's resources on different distributed machines. Therefore, the architecture of distributed systems can cope with network and component shortage because when a machine fails at serving a request, another machine could perform the operation. On the other hand, availability might be affected when the underlying system is concerned more about the consistency of the data. In fact, the more computer resources the system has, the more challenging it is to ensure data consistency.

To reduce the complexity of working with distributed systems, and build computer resources that act as a single computer unit, the details of the underlying

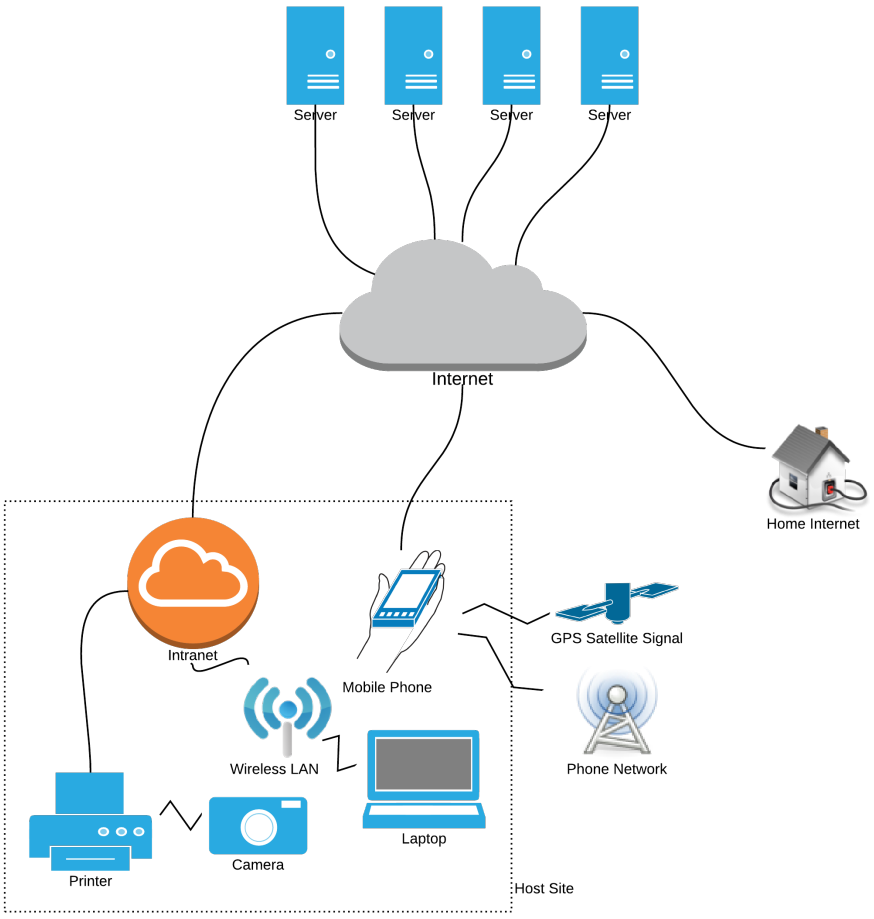

Figure 2.1: Architecture of Distributed Systems [13].

implementation should be hidden via applying the principles of *Distribution Transparency* [13]. The location of application resources could be hidden by applying the *Location Transparency*. In fact, resources might be stored in different locations and storage technologies (e.g. database, file system). However, resources should be accessed without knowing any details about them. *Replication Transparency* could improve the performance of distributed applications and help cope with node or network failures. This can be done by hiding the fact that resources might be replicated. In fact, Location Transparency together with the Replication Transparency help hide the details of replication. When a fault occurs either in the software or hardware of a distributed system, a predefined logic could be initiated and executed to cope with different failures. As a result, the application could be served seamlessly. This can be done by applying *Error Transparency*. Moreover, *Concurrency Transparency* allows accessing application's resources in the same way local resources are accessed in a centralized system.

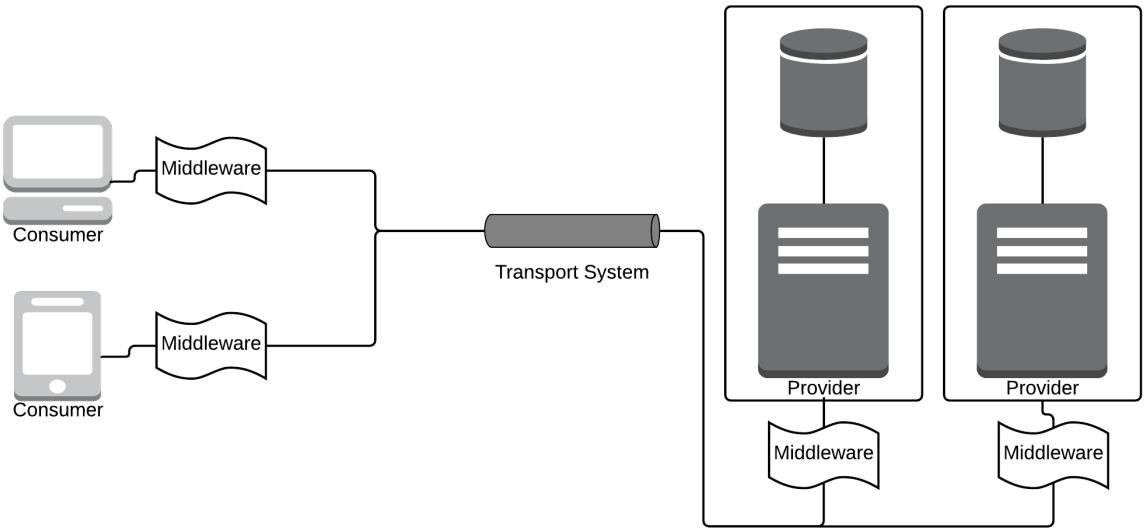

Figure 2.2: Distributed System Building Blocks.

Distributed Systems could be realized as a set of building blocks which might be referred to as *Component Types*. These types could be represented by an arbitrary number of components that interact with each other to form the logic of the application. The *Consumer Type* requests application's resources that are offered by the distributed application. Accordingly, the *Provider Type* receives requests from the consumer and responds with the required resources. In order to realize the request and response interaction, a *Transport System Type* is used which tries to deliver requests and responses. To reduce the complexity between the application layer in both Providers and Consumers and the Transport System layer, *Middleware System Types* could be used which separate the layers and help interact in heterogeneous environments. Figure 2.2 depicts the building blocks of Distributed Systems and how they interact with each other.

2.2 Service-Oriented Architecture

Following section 2.1 which introduced distributed systems, this section introduces an architectural style for building distributed applications. This architectural style is called Service-Oriented Architecture.

A *service* [14] could be defined as a business activity that is made always available to consumers. Service consumers may not need to implement and maintain its functionality. Instead, consumers use the service as a "black box" which ensures

separation of concern. A service in computing could be defined as a business functionality that is offered over a network. It might be offered via different transport systems, representations, and qualities which help consumers decide what is best for their business goals.

The paradigm of developing services to offer different functionalities is called *Service-Oriented Computing (SOC)*. SOC represents a set of principles that define how to build computing systems in a *Service-Oriented Architecture (SOA)* [15]. SOA can be describied as an architectural style that identifies how applications could be constructed based on services which represent the components of distributed applications.

Building distributed applications based on SOA offers many advantages that help achieve business goals. For instance, SOA could offer loose-coupling and heterogeneous interoperability. A function can be invoked by a consumer without the need for previous knowledge about the platform, framework of the service or where it is located. This can be achieved using a distributed-middleware system that supports the service-oriented paradigm. Such middleware systems hide all the complexities needed to complete an interaction successfully. Section 2.2.1 explains Web-Service Technology as an example of a SOA technology which enables building distributed applications.

2.2.1 Web-Service Technology

World Wide Web Consortium (W3C) [16] defines a web service as *"a software system designed to support interoperable machine-to-machine interaction over a network"* [17]. Web-Service technologies enable the development of distributed systems in large scales. This could be achieved using the WS-* stack which is a set of specifications that could be used to implement SOA applications. These applications include provider's offered services and their consumers' applications. The stack also provides many specifications that might be used to extend the functionality of web services. For instance, *WS-Notification* [18] is a standard that could be used to develop and consume *Event-Driven Applications*.

W3C defines Web Service development as describing the interfaces of a Web-service's functionalities in a description language [17]. An example of such description languages is the Web Service Description Language (WSDL) [2] (see section 2.2.3). WSDL is a machine-processable format that is mainly used to describe service interfaces and how a consumer could be bound to the service provider to start consuming the service's functionalities. To interact with a Web service based on its pre-defined interfaces, requests and responses are wrapped in SOAP messages (see section 2.2.2).

2.2.2 SOAP

Components of distributed applications might use SOAP [19] to exchange data and information over a network. SOAP can be described as an architecture to exchange messages in distributed environments. It is mainly used by Web-Service technologies to interact between service providers and consumers. This interaction can be

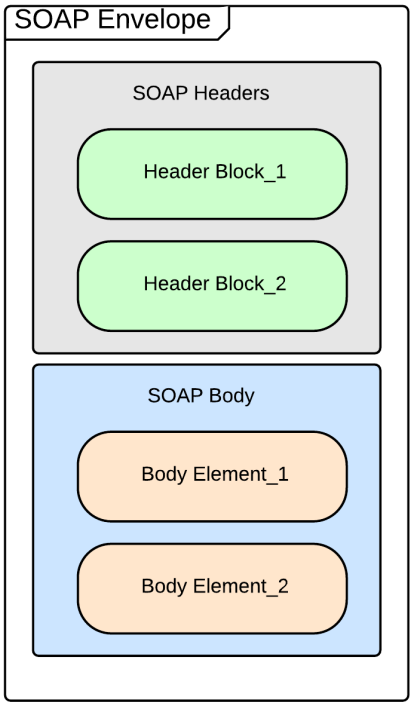

Figure 2.3: SOAP Message Building Blocks.

achieved by wrapping the information in a defined structure and send it over a Transport System as request or response messages. SOAP messages are structured using XML format [19] which consists of five main building blocks that can be seen in Figure 2.3.

The SOAP Envelope is a wrapper for the message's building blocks. It can be used as a definition for starting and ending a message. The *SOAP Header* is an optional construct [19] which could be used to define information that might be needed to process the message. Processing of the message could be performed at the ultimate receiver or at an intermediary between the producer and target receiver of the message. The information of who could process the message and how to do processing should be defined in the *Header Block*. Finally, the real information that is being exchanged between a provider and a consumer is represented as *Body Elements* in the *SOAP Body*.

SOAP does not define a standard Transport Protocol to carry the messages between providers and consumers. Instead, it can be used on top of many protocols, but *Hypertext Transfer Protocol (HTTP)* [20] (see section 2.3.1) and *Simple Mail Transfer Protocol (SMTP)* [21] , however, are mainly used to carry SOAP messages.

Listing 2.1 shows an example of a SOAP message that is communicated over HTTP Protocol. In the example, a request "*AddCustomer*" is sent to the server to add a new customer. The request message has the parameters needed for adding a customer which are: *FirstName*, *LastName* and *Address*. The *Customer* functionality is identified via the namespace: *http://www.example.org/customer*. The endpoint of the Web-Service that handles the requests to add new customers is handled by the HTTP protocol. In the example, the endpoint of the customer service is: *www.example.org/myservice/customer*.

```
1  POST /customer HTTP/1.1
2  Host: www.example.org/myservice
3  Content-Type: application/soap+xml; charset=utf-8
4  Content-Length: nnn
5
6  <?xml version="1.0"?>
7  <soap:Envelope
8  xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
9  soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
10   <soap:Body xmlns:m="http://www.example.org/customer">
11     <m:AddCustomer>
12       <m:FirstName>Mustafa</m:FirstName>
13       <m:LastName>Jaber</m:LastName>
14       <m:Address>Stuttgart</m:Address>
15     </m:AddCustomer>
16   </soap:Body>
17 </soap:Envelope>
```

Listing 2.1: SOAP message communicated over HTTP Protocol

2.2.3 WSDL

WSDL is a Web-Service technology to describe the interfaces of a service in a structured way. The documents of WSDL are formatted using XML, which enables the definition of detailed information needed by the service provider or consumer. WSDL can be also used to help implement services on the provider's side. This could be achieved by defining the interfaces of the service first, then the implementation of the interfaces functionality will be followed. To do that, two main approaches could be used; the *Top-Down* approach is to describe the web service as a WSDL representation first, and the concrete implementation will be generated later based on the described interfaces in the WSDL document. The second approach is the *Bottom-Up* approach; this means that the web services is implemented or the set of functionality interfaces are implemented first, then the WSDL document could be generated based on the defined interfaces the service. In addition, WSDL enables service consumers to interact with the service provider via providing the necessary information to start the interaction. The information includes the endpoints of the service, service interfaces, parameters of the interfaces and their schema.

The provider and the consumer generate service stubs using WSDL documents which in turn helps develop custom applications based on the service in-

terfaces. When requesting a service functionality by calling its interface or receiving a response from the service provider, SOAP messages are marshalled and un-marshalled to exchange the necessary information and complete the interaction.

2.3 REST

In this section, another architectural style is explained that defines a way to build distributed applications. This style is called *Representational State Transfer (REST)*.

REST was first introduced by Roy Fielding in his doctoral dissertation [22] as a way of designing distributed applications. REST is based on a set of features that are collectively called an architectural style. The main differentiator between REST and SOAP-based services is that a SOAP-based service is Functionality-Centric while REST is Resource-Centric . A Resource in a REST application could be thought of as a reflection of a real-world object or a representation of abstract objects or entities. A Functionality, however, is an application specific procedure that is exposed over the network.

The approach that Fielding used to identify the features of REST is *Constrained-Oriented*. This means that the architecture of REST is based on a set of factors that define the process of finding a solution to build a specific system, in fielding's case, a distributed application. The constraints that identified for designing REST applications are [22]: *Client-Server, Layered System, Statelessness, Cacheability, and Uniform Interfaces*.

Client-Server

All the interactions between the different components of a distributed system are identified to be between a client and a server. This constraint helps ensure separation of concerns by enabling different kinds of clients to work with the same server. As a result, the complexity of client applications can be reduced because they should be capable of addressing only the server. In addition, the server should not have any previous information about the communicating clients. Instead, the client application is responsible for getting the information of how to communicate with the server.

Figure 2.4 depicts the interaction of a client process with a server by requesting a specific functionality. This example represents a client that communicates with the server synchronously; when a request is issued to the server, then the client will be blocked (i.e. waiting) until a response is received from the server.

The Client Server Constraint helps build software applications that could run in heterogeneous environments; this is defined as software *portability* [23]. Portability could improve the scalability of distributed applications because they could be divided into generic interfaces that could be deployed onto different platforms. As a result, server components could be simplified by porting repeatable logic onto different components which helps process

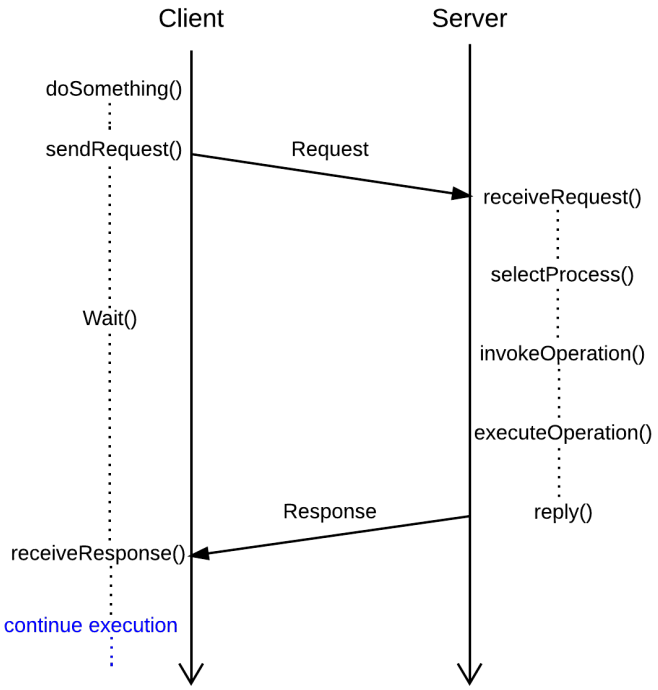

Figure 2.4: Client-Server Interaction.

requests of more clients. For example, The *User Interface Layer* could be implemented and ported onto the client environment. This design reduces the overhead on the server by running the user interface logic on Client's machines. Additionally, the separation between the client and server logic improves the evolvability of their components. This is due to the fact that communicating components should agree only on the shared interfaces to complete successful interactions.

Layered System

This constraint manages the complexity of a distributed system by defining a hierarchy for the application components. For instance, client applications might interact with a server component directly. Alternatively, clients might be connected to an intermediary that serves their requests if the data is stored in its cache. Or else, it forwards the request to the server which receives the request and then sends a response back to the intermediary. When the intermediary receives a response, it forwards the response to the requesting client and might save a copy in its storage system. Either ways, components on different system layers follow the client-server constraint which makes any interacting components behave as client-server components.

In Figure 2.5, an example of a layered system is shown. The *Client Layer* in this figure is capable of communicating with the *Intermediary Layer* Only. Similarly, The *Intermediary Layer* mediates the communication between the *Client Layer* and the *Server Layer*. If a server should respond to a request that a client issued, then this response has to go through the intermediary before it reaches the client.

The Layered System constraint could reduce the complexity of a distributed application by reducing the number of components that each client has to interact with. In fact, each component could communicate only with the layer that it is connected to. Additionally, layering a distributed application might scale the overall application and enhance its performance. This could be seen, for instance, when introducing a cache as it will be discussed next.

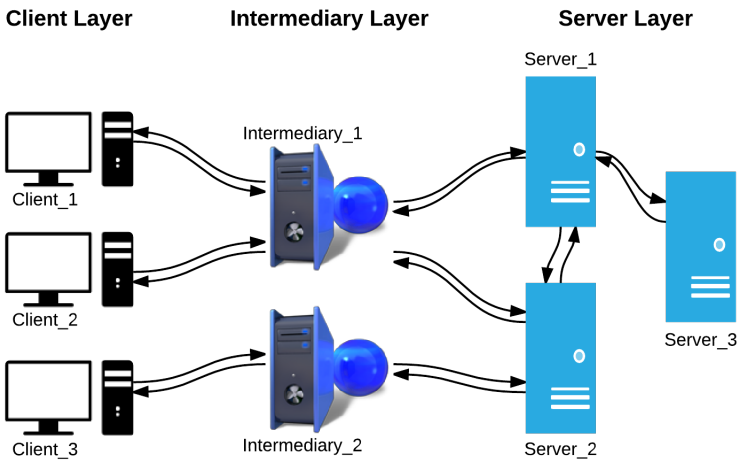

Figure 2.5: Example of a Layered System.

Cacheability

The REST architecture defines that server responses have to be clearly identified of whether they should be cached on a different layer or not. This constraint enables three levels of caching. The first is realized at the client side. When the server sends a response message, the client stores a copy of the response on its local storage "e.g. web browser" and processes it depending on the context of the application. The second kind of cache is realized on the server side. This type of cache ensures storing responses of frequent requests in the main memory, allowing fast access to their computed responses. Finally, responses could be cached on any intermediary that resides between the client and server, e.g. proxy server.

The Cacheability constraint helps reduce the latency of client requests. This

could be achieved by either avoid sending requests to the server, or by reducing the travel distance that the client message has to go through. For instance, when an intermediary has a cached response, then it is taken from the intermediary's cache and sent back to the requesting client without the need to forward it to the server. Consequently, the network bandwidth and the cost required to serve client requests are significantly reduced.

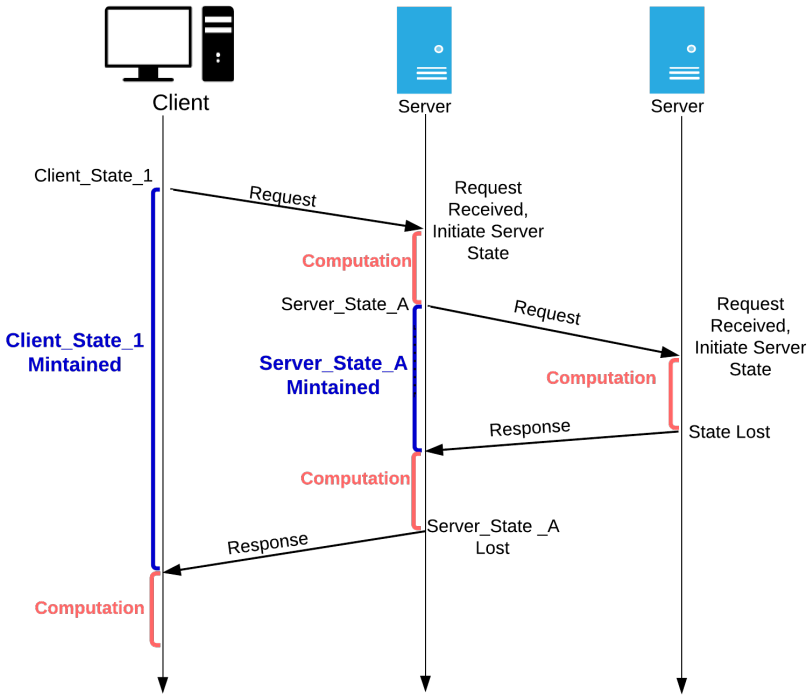

Figure 2.6: State between Clients and Servers.

Statelessness

The communication between clients and servers in the REST architecture is expected to be stateless; this means that all the information needed by the server to process a request should be included in the request message. The state of a REST application which includes the context of both the server and client should be entirely managed by the client in what is called the *session state*. The session state should be communicated with the server when issuing each request. This helps build a stateless server which does not need to store any information about the communicating client and the current state of its application.

The Statelessness constraint helps build reliable distributed applications because the state is stored always on the client. For instance, in case of any partial failure in the server or underlying network, the client can recover

from the failure by sending the request again. In addition, this constraint helps build scalable distributed applications that can add or remove system resources (e.g. intermediaries and servers) without affecting client applications. This is mainly achieved by managing the state only on the client side which in turn reduces the complexity state management.

REST client applications should be able to keep information about the current resource it is interacting with, and the resource's next possible transitions using its links (see section 2.3.4). The server should be able to only respond to client's requests. Once the server sends the response message, it should forget everything about the messages of the request and its response [24]. Figure 2.6 depicts how clients and servers manage state. In the shown example, a client sends a request when it reaches *State_1* and then waits for a response while maintaining its state. Once the server receives the client's request message, a server state is established to process the request. To continue processing, the server needs to communicate with another server for a needed information. Before sending the request message to the server that holds these information, the server stores the state as *State_A* and then issues the request. Once the second server receives the request, it starts a server state, then it processes the request and replies with a response. When sending a response, the server should delete any state associated with the request after ensuring that the requester received the message. The first server receives the response, it then continues processing the client's request, and once the response is ready, it sends the result to the client as a response message. When the client receives the response message, it continues its operations and the server deletes the server state and waits for other requests.

Uniform Interface

REST architecture defines that distributed components should apply the software engineering principle of generality which could be described as *"being not limited to one particular case"* [25]. As a result, the implementation details of the communicating client and server applications are hidden and decoupled. To be able to do that, previously agreed-upon interfaces that decouple and abstract the implementation could be used. Using such interfaces in REST services introduces a standardized mechanism to perform standard operations on targeted resources. Consequently, a generic and standardized way for communicating between system layers is introduced.

Applying the Uniform Interface constraint improves understandability between clients and servers in the way they exchange information. Additionally, building components in a heterogeneous environment becomes easier because these components have to understand each other on a higher level of abstraction. This could be achieved by constraining client-server components to communicate via generic interfaces which describe application's resources independent from any underlying technology. As a result, system design becomes less complex.

The Uniform Interface constraint defines a set of sub-constraints which identify how to constrain the design and communication of an application's com-

ponents using uniform interfaces:

Identification of Resources REST defines the information that is offered to be interacted with by a server as Resources. A resource has to be identified uniquely in order to complete an interaction. When using HTTP as a Transport system - will be discussed in the following section - REST uses the *Uniform Resource Identifier (URI)* to uniquely identify application's resources.

Manipulation through Representations To be able to work with REST service's resources, information should be serialized to be transported over a network. Representing Resources in distributed applications can be achieved using representations. These representations are structured formats to define objects' data. In REST, resources can be represented in different formats, and it is the decision of the client to choose what representation is best for its application. This process is called ***Content Negotiation***.

Self-descriptive Messages descriptiveness of a message is identified by including meta-data information to a request or a response. These information allow identifying the way this message has to be processed and interpreted. The relation between Self-descriptiveness and statelessness is that statelessness deals with the information needed for the communication while descriptiveness deals with the interpretation of the messages.

Hypermedia As The Engine Of Application State (HATEOAS) REST defines that the client state should be driven by the possible transitions that the server sends in response messages. For instance, resource's representation tells the client the next state that could be possibly followed. The representation of the server's responses should be the only information needed to drive client application's state. To enable state transitioning, the REST service should be *discoverable*. *Discoverability* could be described as a mechanism of identifying all the information needed to successfully drive the next interaction between the client and server. These information should be included in the server's response messages so that the client understands what possible states could be transitioned to. Applying this constraint could ensure loose-coupling between client and server components.

Applying the REST architectural style results in a set of benefits for distributed applications. ***Scalability***, for instance, can be improved by applying Client-Server constraint. This results from the fact that it makes it easier to add more server instances to serve clients depending on the server's load. The Layered System constraint could also improve scalability because different layers can grow independently from each other and might be improved beyond the proprietary infrastructure. Statelessness can scale applications as well; this results from the fact that adding server instances becomes easier because the client is responsible for the session state. In addition, Cacheability improves the efficiency of applications by

reducing the latency and bandwidth needed to server client's requests. As a result, the system can scale as it is able to serve more clients. ***Evolvability*** is another benefit that could be obtained by applying the Client-Server constraint, this could be achieved due to the separation between clients and servers. Consequently, client applications could evolve independently from servers and their offered services. In addition, The Uniform Interface constraint helps improve evolvability by the ability to replace system components with improved resources without affecting the functionality they offer. ***Visibility*** is another benefit that refers to the ability to manage system components by monitoring their behavior [22]. Visibility could be achieved by applying the Statelessness constraint; monitoring components should be capable of analyzing the system by looking only on the communicated messages which, fundamentally, carry all the necessary information. The Uniform Interface is another constraint that helps improve visibility by the fact that messages have consistent semantics over the components of the system. ***Reliability*** is a crucial benefit that could be obtained by applying the statelessness constraint. This is due to the fact that failures in servers and underlying network could be recovered from by the client via resending messages again.

2.3.1 HTTP

In the Distributed Systems section (see 2.1), the Transport System was introduced as a building block of distributed applications. This section introduces a widely used and fundamental transport system that can be seen in most of todays distributed applications, the Hypertext Transfer Protocol (HTTP) [20].

HTTP has been used mainly to drive the internet since the initial development of the World-Wide Web (WWW) in 1990. It is defined by the *Internet Engineering Task Force (IETF)* [26] in its RFC7231 specifications as *"An application-level protocol for distributed, collaborative, hypermedia information system"* [20] . HTTP is used to access remote resources across the Web. This is achieved using the Client-Server model of communication, that is, a client starts the interaction with the server by requesting a specific resource. This resource is uniquely identified via its *Uniform Resource Identifier (URI)* [27]. The request is realized as an HTTP message which is delivered to the server. Once the server receives the request message, it processes the request and sends an HTTP message to the client as a response.

HTTP could be described as a simple and scalable transport system. These advantages helped HTTP dominate the internet's transport systems. Distributed applications that leverage HTTP could be scalable due to the properties that HTTP offers. For Instance, *Statelessness* helps scale distributed applications via separation of concern. Another property that improves the performance of distributed applications when using HTTP is *Cacheability*. As a result, the load on the server might be reduced by reducing the number of requests it needs to process. Caching is made possible via HTTP due to the fact that resources are uniquely identified. Consequently, caches of clients, intermediaries and servers can uniquely identify the requested resources and serve their responses quickly. In addition, HTTP offers many possibilities for different caching policies which can be realized via metadata information of request and response messages. This depends mainly on the fact that HTTP messages contain the necessary information needed to describe a spe-

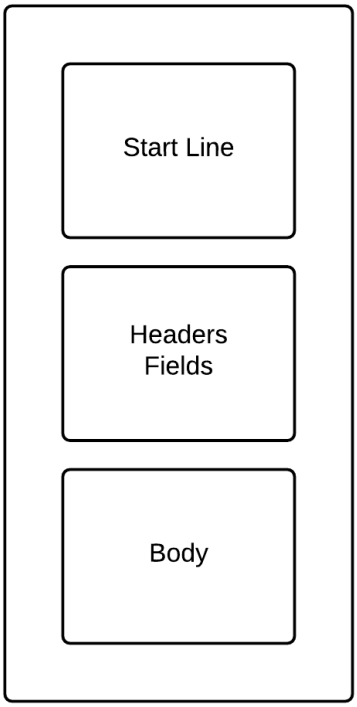

Figure 2.7: Main building blocks of HTTP.

cific resource.

HTTP messages can be distinguished into a request message and a response message. In general, a message consists of three main building blocks as shown in Figure 2.7; a *Start Line*, *Header Fields*, and a *Message Body*. The *Start-Line* and *Header Fields* represent the meta-information of HTTP message while the *Message Body* include the actual message information to be consumed. The Start-Line for a request message identifies the target resource and operation. For a response message, a Start-Line identifies the status of the request message. Header Fields could vary based on the type of message. In fact, some header fields are only applicable to request messages, and others are only for response messages. Some other header fields are, however, applicable for both request and response messages. Header fields define, in general, meta-information about the message itself or its content. The body of the HTTP messages carries the actual information that the server or client application is mainly interested in.

2.3.2 REST Maturity Model

Leonard Richardson classified web technologies and applications in a model called *Richardson Maturity Model* [28]. This model distinguishes different levels based on a set of requirements that have to be fulfilled by the technology or application. Level 3 in this model is considered the highest and most mature level. In addition, each level of the model must fulfill all the requirements of the other levels below. The levels of this model represent the incremental improvements that could be implemented to develop a compliant REST application by applying the REST constraints.

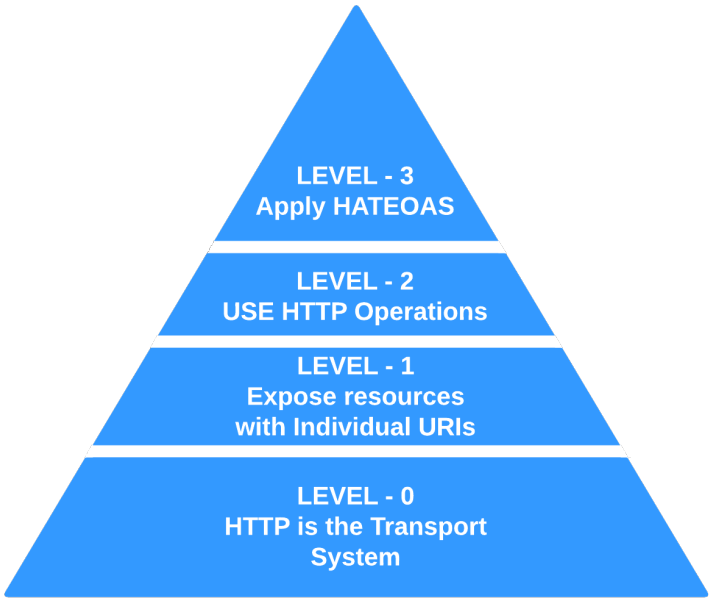

Figure 2.8: Richardson Maturity Model [28].

The technologies in Level-0 use HTTP as a Transport System to carry exchanging messages between distributed nodes. However, different message architectures could be used to communicate information and knowledge. For instance, this could be seen in *Remote Procedure Calls* technologies such as SOAP.

Technologies and applications of Level-1 expose individual resources. This ensures having a unique identification scheme for each resource and prevent having a single endpoint for all of them. Using HTTP, this could be realized by using a URI for each resource.

HTTP offers a set of operations that REST is natively based on. These operations are GET, POST, DELETE and PUT. Relying on these operations for

representing the semantics of resource's operations of an application ensures fulfilling Level-2 requirements.

Loose coupling between services providers and consumers can be ensured in Level-3. This could be done by applying the HATEOAS constraint. As a result, client applications navigate through the possible states that the server offers in the responses.

2.3.3 Resource Representation

REST architecture defines a consistent way of exchanging information between distributed components. To achieve this consistency, REST applies the '*Manipulation through Representation*' constraint as it was discussed earlier. This constraint allows a machine-processable representation of objects. It also allows distributed components to understand each other by sending and receiving information that they can interpret and consume.

Objects can be represented in many formats. Therefore, the concept of ***Media Types*** [29] (formally MIME types) is used to ensure a consistent way of representing objects' data. A media type is a specific representation of object's properties and data. On the other hand, different media types represent the same object in different formats and markup languages.

Because REST messages are stateless, the exchanged messages between the server and client must include all the required information to complete a successful interaction. These information could be represented in the media type used between the components. These information include the data and metadata information that describe the resource that could be manipulated and what action should be performed.

2.3.4 Hypermedia Links

Resources on the global Web can be thought of as a *Directed Graph*. The Vertices of the Web graph are the resources that are exposed to the Web users (e.g. documents, photos, videos etc.). The Edges between these vertices represent the link between each resource that points to another. That is:

$G = (V, E)$ where,
V : a non-empty set of resources.
$E : E \subseteq V \times V$ a set of directed edges representing links between resources.

Transitioning from one resource to another is performed via visiting n resources along the path. That is, $n \geq 0, u, v \in V$ a path of length n from resource u to v in G is a visiting sequence of resources and links $r_0, l_0, r_1, l_1, ..., r_n, l_n$ such that $r_0 = u$ and $r_n = v$ and $l_i = (r_i, r_{i+1}) \in E$ for all $i \in \{0, 1, ..., n-1\}$.

Figure 2.9 shows an example of a *Resource Graph* that is offered by a company which uses URIs to link resources. These resources might be exposed on the network to help build distributed applications. When navigating the company's re-

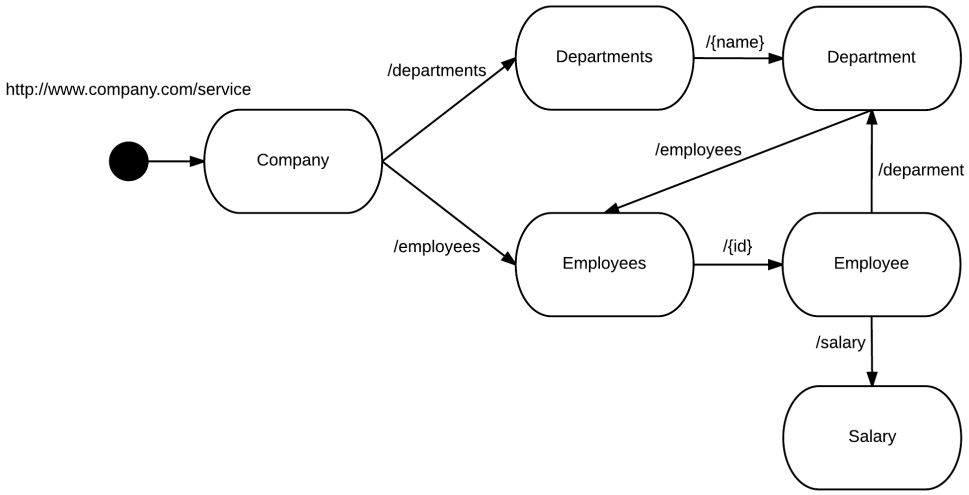

Figure 2.9: Resource Graph of a Company.

sources, the starting point should be the *Company's* resource. That is, by visiting the *http://www.company.com/service* the Company resource will be the resulted response. To be able to check all the departments that the company has, the link *http://www.company.com/service/departments* is used to transition from *Company* resource to *Departments* resource. Similarly, to be able to check the salary of an employee whose id is "1234" and works in the *finance* department, the following link could be used:
http://www.company.com/service/departments/finance/employees/1234/salary

To be able to navigate through Web resources, *Hypermedia* should be used as a mechanism to provide links between related resources. Furthermore, when using HTTP as a Transport System for REST services, the constraint of *Identification of Resrouce* could be realized by identifying resources uniquely via URI links. This can be shown in the previous example.

Hypermedia Links could be defined as a mechanism to uniquely identify resources and provide semantics to their relationship with each other. This mechanism helps control the transitioning between application's resources and could be used to realize REST's HATEOAS constraint.

IETF in its RFC5988 [30] specifications defined that *Web Links* should be comprised of a *context URI*, *a target URI*, *a link relation type* and *optional target attributes.* Following this definition, a hypermedia link could be built of:

href: This attribute of the link represents the URI of the target resource.
rel: This attribute represents the relationship between the current resource and

the target resource. *rel* is mainly used to identify the semantics of the link.

The *context URI* should be known to the client application. In fact, the context URI represents the URI of the last resource that was interacted with.

Listing 2.2 shows an example of an HTTP response message which carries the resource representation with Hypermedia Links when requesting the *Company* resource in figure 2.9.

```
1  HTTP/1.1 200 OK
2  Date: Tue, 23 Nov 2014 22:38:34 GMT
3  Content-Type: text/xml; charset=utf-8
4  Content-Length: nnn
5
6  <?xml version="1.0"?>
7  <company>
8    <name>Uni Stuttgart</name>
9    <links>
10     <link href="http://www.company.com/service/departments" rel="departments">
11     <link href="http://www.company.com/service/employees" rel="employees">
12   </links>
13 </company>
```

Listing 2.2: HTTP response when requesting Company Resource

2.4 Why HATEOAS

This section introduces the advantages that REST client applications can get when applying the HATEOAS constraint.

Today's services can be differentiated in terms of evolveability intervals to long-term and short-term evolving services. If a client consumes functionalities of a long-term evolveable services, then this client might be able to support service evolution when applying the HATEOAS model. On the other hand, clients that consume services that evolve frequently could benefit the most when applying the HATEOAS model. Such clients might be able to cope with rapid changes of services which results in reducing development cost and time.

One of the most important challenges that face software designers and engineers in the process of designing their distributed applications is the ability to cope with changes. These changes might differ from one application to another. This could be due to load changes, fixing errors, better performance design, or due to strategic changes such as the cost of the underlying infrastructure or its reliability. As a result, distributed applications should be designed and implemented of components which represent application's subsystems. These components are then interconnected and integrated to form the larger view of the distributed application. This design of component-based applications along with the concept that REST applications should be divided into resources that represent the business domain yield a hierarchy in the URI of the resources. Using HATEOAS helps separate the concern of changing the location, name, and hierarchy of application components on different servers. This is because of the fact that when applying HATEOAS, clients

will be supplied with the location of the resources' endpoints on runtime.

The current process of developing custom client applications that consume REST services involves going to a special web page that is exposed by the service provider, then reading the service's documentation, and trying understand the data model, resource transitions, and interaction responses. This way of consuming REST services is error prone. The ability to develop against such services depends on how well-documented and understandable the service is, how consumers perceive the documentation and implement interaction requirements, and how to implement correct transitions between resources. Applying HATEOAS helps reduce issuing invalid requests and wrong state transition calls. This can be achieved because the client will be driven by what state transitions the server allows. The server that applies the HATEOAS constraint responds only with the links to the next possible states. This implies that the server is completely responsible for the structure and hierarchy of URIs that point to the next possible interaction, which in turn helps design loosely coupled clients. Similarly, invalid requests could be reduced based on the concept of discoverability; when designing discoverable resources via describing the possible interactions in the response messages, the developer might not need to go through the service's documentation to understand how the interaction should be done, which could be error prone. Instead, the client application should be able to automatically construct the request messages based on the information included in the response messages.

When developing REST client applications, it is important to consider the logic that changes the client state. This logic is what drives the application and it should depend on the interactions that the server allows. When using HATEOAS, these transitions between resources are the responsibility of the service provider. When developing clients that understand how to drive the application based on the HATEOAS model, then the logic of such applications could be easier to implement. The fact that transitions are the responsibility of the service provider helps client developers focus on solving the business and technical problems of their applications instead of worrying about how to interact with the service.

Chapter 3

REST Clients

To drive the solution of building a generic framework for building compliant REST client applications, a study of the current types of service consumers had to be conducted. This chapter introduces the types of REST's client applications and their behavior to consume REST services.

3.1 Client Types

REST clients could be described according to their functional or behavioral forms. In the context of REST clients, a ***Functional Form*** describes the purpose and actions a service client pursues to achieve a set of defined goals. The ***Behavioral Form***, on the other hand, can be thought of as a way to describe how the functional forms could be accomplished i.e. what drives the client to achieve the set of goals that the functional form is trying to achieve. In addition, combinations of the two forms could be realized.

The behavioral form of REST clients can be classified into two main categories. These categories identify the forms in which REST applications could be used:

Human-Oriented Clients
: The target of the client application is mainly a human-being. Clients of this form are based on the interaction between a human and a physical object. These interactions cannot be completed without a human-being initiating the interaction. To build such applications, physical and software interfaces are provided so that the interaction could be done. For instance, a computer with a web browser is a form of an environment that could be used to build a client application that interacts with a REST service. Another example could be an internet-enabled fitness gadget that sends activity data to a REST service when the user presses a button after finishing an exercise.

Machine-to-Machine Clients
: These kind of clients are machines that interact with a REST service independently and without any human intervention. An example of such clients is a modern car GPS system. It interacts with a server to get traffic information, then it calculates the shortest route to a destination point based on

this data. This workflow is kept running and updated until the destination point is reached.

The functional forms of REST clients are mainly about how a client application might work:

Mediator Client
These REST clients are mainly based on human's ability to understand information. Such applications serve as interfaces to the human in which they help communicate information to the user. For instance, a web page shows information in a visual way about an activity in a workflow. To complete the activities of this workflow, a user has to fill in text fields to continue to the next activity which might be represented as a web page. Although the same visualization elements might be used, they could differ in the meaning they provide. For example, a text field might mean to fill in a name, or it could mean to fill in a credit card number. In fact, such client applications do not have any specialized business logic. Instead, they mediate the operations between the user and the server.

Curious Client
It is a kind of client applications which is interested in everything a REST service offers. It tries to visit every single data or resource on the server and decides later what to do with such data. An example of a client which has this kind of behavior is a web crawler; it fetches all the data that can be found on the server and indexes this data for later usage. Fetching the data can be done in many ways, one of them is following all the hyperlinks found in each response. Another example could be a service visualizer. This visualizer tries to visit all the resources of a REST service in order to visualize its resources' interconnection and transitioning workflow.

Lazy Client
It is called lazy because it is interested in one resource that a REST service offers. These kind of clients send or receive information data by interacting with a resource periodically. As a result, they are built of simple and rarely changing workflows. An example of such clients is a Stock Price viewer. It fetches the stock price periodically and shows it to the user whenever it is needed. Another example in the field of embedded system could be a sensors-network system. Each sensor interacts periodically with a specific resource on the server to send its readings so that further analysis could be done.

Constrained Client
These clients are constrained because they execute a pre-programmed workflow that is hardly to be changed on runtime. This workflow is executed to achieve a specific goal that is often coupled to the current architecture of a REST service's resources. When any modification is made on the server side, this kind of clients will break. Most of todays' REST clients are Constrained Clients. Programmers specify the endpoints of resources and program the workflow in which a specific goal has to be achieved. To cope with the prob-

lem of breakable clients, REST services providers ask developers to upgrade their clients to work with an updated version of their service implementation.

Autonomous Client

This kind of clients is considered intelligent; this is due to their ability to adapt to different changes that might be faced on runtime. These changes are mainly observed on the server side which might affect the client's original workflow. An autonomous client is driven by algorithms that change its application workflow based on what the REST service offers. This could be realized by applying the HATEOAS constraint that was discussed earlier. A drawback of such clients is that when the architecture of the REST service is changed, the user experience of the application is sometimes going to change.

Based on the mentioned client applications, the following chapters introduce generic solutions that help building custom applications for these clients.

Chapter 4

REST Description Language

This chapter studies some of the relatively widely used description languages of REST services. In addition, it introduces a proposed solution for describing hypermedia REST services. This chapter describes the motivation, components and architecture of a description language that could be used to build compliant REST applications.

4.1 RESTDL

Based on the existing description languages and their deficiencies discussed in section 1.2.1, *REST Description Language (RESTDL)* had to be designed and developed. RESTDL offers, unlike other description languages, the possibility to describe REST resources without identifying their locations e.g. URIs. In addition, RESTDL is developed with readability and discoverability in mind. This way, developers of REST client applications can have better understandability when they are introduced to new REST services. In addition, RESTDL defines an architecture for a description language that is independent from the underlying Transport System, technology, or markup language that represents its documents. Moreover, RESTDL applies REST's architectural constraints to enable building LEVEL-3 REST applications based on Richardson's Maturity Model.

RESTDL, unlike other description languages, is mainly based on the concept of hypermedia relations. Therefore, compliant REST client applications could be built by driving their business logic based on the current available resources' relations on the server side. In addition, RESTDL enables the possibility to identify many different interactions on the same resource. This way, a resource is not only restricted to the CRUD semantics, but it can also define its own semantics based on standardized CRUD operations. For example, an *Account* resource could define two relations, *credit_account* and *debit_account*. When using HTTP, both relations could be based on the POST operation. For example, to credit money to an account record on the server's database, *credit_account* relation should be used. Similarly, *debit_account* should be used when processing the account to get some money out of it.

RESTDL was developed to establish a fully-functional and consistent way for communicating and exchanging data models between service providers and their

clients. It is a machine-processable description language that defines all the needed information to establish successful interactions between a client and a server. In addition, RESTDL is a client-first description language. That is, the information provided by the description language was mainly intended to be used by client applications. Therefore, the data definition is optimized to be consumed by different client environments. However, RESTDL can be also used on the server side. In fact, RESTDL can be used as Code-First or Describe-First. In Code-First model, resources on the server side are implemented, then RESTDL is generated. On the other hand, Describe-First model allows describing the server's resources using RESTDL, then the server and client code of the data model will be generated based on these definitions.

In REST architecture, resources could be manipulated using a standard set of operations: Create, Read, Update and Delete; which are known as the *(CRUD)* operations. Each of these operations has its own set of requirements to complete a successful information exchange between a server and a client. These requirements include resource-specific properties, and communication-specific properties. When the information of these requirements are supplied by the server, and assuming no other failures in the system, then Client's requests will mostly be processed successfully by the server. To help supply and understand the data requirements needed to establish successful interactions, RESTDL defines how clients and servers should exchange information. Exchanging information is achieved by describing how a client should request a specific resource with any of the CRUD operations on the server, and what the client should expect from the server as a response. This concept is called ***Interactions Definition***

4.2 RESTDL Assumptions

RESTDL aims at supporting a full-compliant REST interactions. Therefore, RESTDL was designed to utilize REST constraints and specially the HATEOAS constraint. However, to ensure HATEOAS, RESTDL assumes that the responses from the server include links using a *Hypermedia Media Type*. These links identify the next possible interactions that could be performed on the server side. To achieve REST compliance, a set of assumptions were made so that the client can completely understand how to interact with the resources on the server side and what to expect as responses. These assumptions are:

Relations Schema

Relations represent the relationship between the current resource and the next possible resources of a REST service. To be able to work with relations, clients should be able to understand what they mean and how they could be utilized. This can be achieved by providing specific information that defines the structure of each relation and how could it be used. This can be achieved by providing a description of all the data and metadata of the associated resource and underlying transport system. This ensures having the knowledge needed to complete an interaction to follow a specific link in a response message.

Root URI
The client framework that is to be developed assumes that it will be fed with only the root URI of the REST service. The root URI must have all the necessary information needed to continue interacting with the server. This includes a link to the endpoint where the client can fetch the server's relations schema.

Hypermedia Media Type
RESTDL doesn't constrain resources' information to be represented in a specific format. Instead, it assumes that the resources will be communicated with a suitable Media Type that the designer of the REST service chooses. This Media Type should support the HATEOAS model to ensure full REST compliance.

Based on the above assumptions, RESTDL was designed to describe the schema of the relations that are offered by the server. This description ensures a consistent way for identifying the semantics of links, how the client should communicate with the server, and what to expect from the server as a response.

4.3 RESTDL Architecture

RESTDL is mainly based on REST's *Identification of Resources* and *HATOEAS* constraints. This is achieved by identifying the hypermedia links of service's resources. Identification of links includes understanding what the combination of a resource's ***URI***, its ***Type*** and ***Relation*** could mean with respect to the current application state.

Figure 4.1 depicts the architecture of RESTDL's format. Basically, RESTDL consists of five main constructs which describe all the interactions that can be communicated between a server and its clients. When a request is sent by a client to a server, this server replies with a response to the request which includes a list of links. Each link has its relation which describes uniquely what interaction can be performed with the server when following the link's URI. To describe the interactions between clients and servers, RESTDL defines the following constructs:

Server Interactions
A wrapper to all the interactions of a REST service. An application in REST architecture should be built in a resource-oriented approach. That is, the application is constructed of resources that reflect the business domain of an organization's application. For example, a resource could represent a Customer, Purchase Order or any named object that adds a business value. To describe how a client could get benefit from an organization's service resources, exposed resources should be encapsulated by the *Server Interactions*.

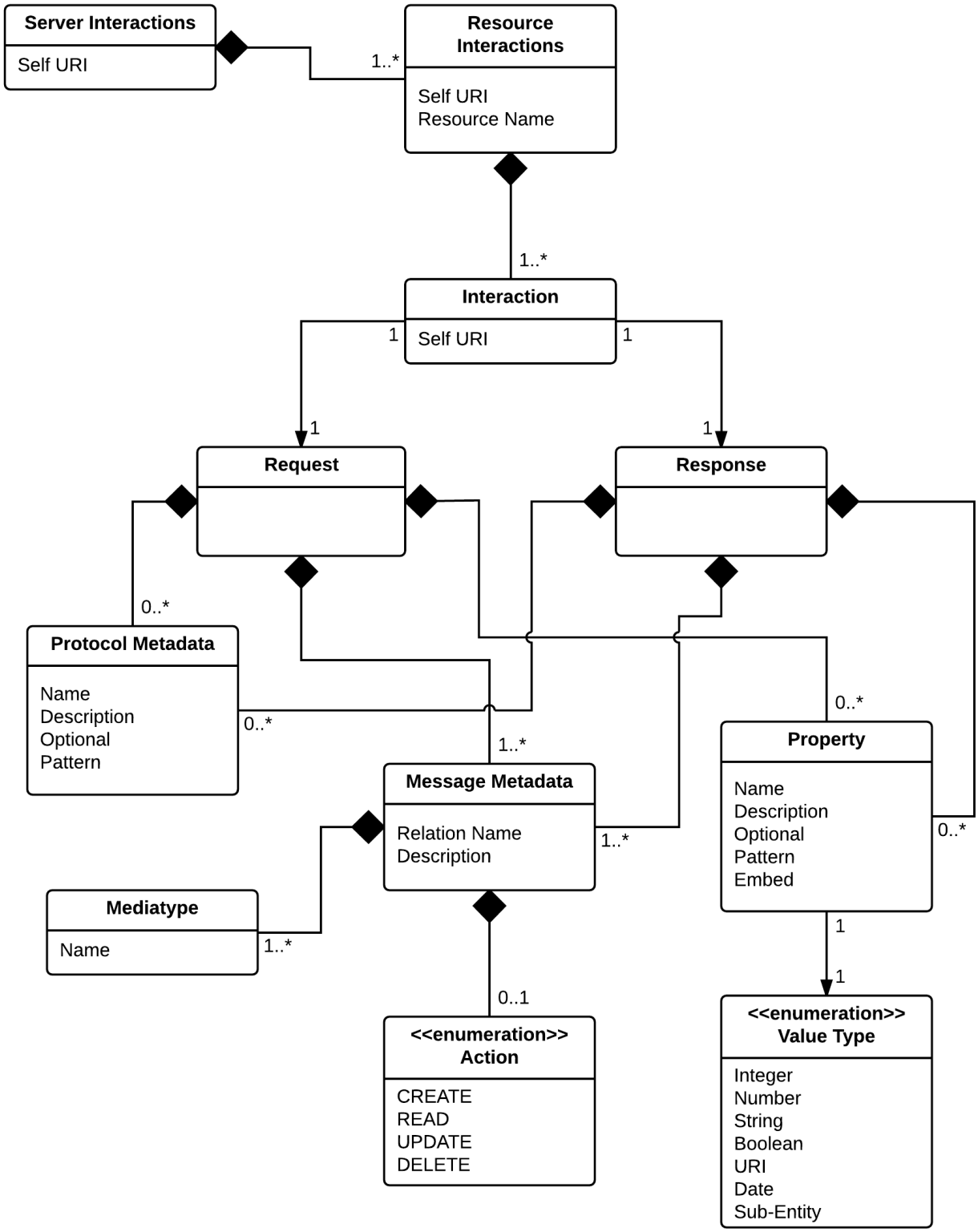

Figure 4.1: The Architecture of RESTDL.

```
<serverInteractions>
  <interactions resource="resource_1" />
  <interactions resource="resource_2" />
  <interactions resource="resource_3" />
  ...
</serverInteractions>
```

Listing 4.1: Server Interactions Wrapper

Listing 4.1 shows how a server wrapper can encapsulate the interactions of exposed resources. In this example, REST service exposes three resource: *resource_1*, *resource_2*, and *resource_3*.

Resource Interactions

Applications as discussed earlier are designed of resources. Each resource could offer a set of operations that manipulate its state. For instance, CRUD operations might change the state of a resource when completing an interaction successfully. To describe the different interactions that a specific resource might have, *Resource Interactions* construct is used to wrap each resource's interactions independently.

```
  <interactions resource="resource_1">
    <interaction />
    <interaction />
    <interaction />
  </interactions>
```

Listing 4.2: Resource Interactions Wrapper

Listing 4.2 is an example of how an interaction should be wrapped by a specific resource' interactions wrapper. *resource_1* in this example has three interactions. It should be noted that the interactions of a specific resource might be not the standard CRUD operations only. Instead, any number of interactions could be uniquely defined that could add a business value to the application.

Interaction

REST defines the *Client-Server* constraint as its *Message Exchange Pattern* [31]. This constraint is realized in RESTDL using the *Interaction* construct which wraps the *Request-Resonse* information. This construct defines what message request has to be sent to the server and what response to expect from the server in return as a result of following a specific link in a response message.

```
<interaction>
  <request>
  </request>
  <response>
  </response>
</interaction>

```

Listing 4.3: Request-Response Interaction Wrapper

Listing 4.3 shows how an interaction should wrap a request-response pair.

Request

REST architecture defines that a request message should have the necessary information to be processed successfully. This is identified by the *Self-descriptive Messages* constraint. To help fully describe how a message should be sent to the server, the *Request* construct is introduced. This construct encapsulates all the information needed by the client to issue a successful request. Request construct defines the information schema of how the message should look like. This schema includes resource-specific and communication-specific information which will be discussed in the following section. It should be noted that the definition of the Request information is independent from any Media Type. Instead, the representation of information should be left to the component which decides on the preferred available Media Type.

Response

Following the same purpose of describing the information for successful message's communication as it was earlier discussed in the *Request* construct, the *Response* construct defines how the response message is going to be like. A service client starts an interaction by sending a request message. When the message is sent to the server, the client always expects a response in return. RESTDL defines what the client should expect from the server by defining the schema of the response message. The schema defines resource-specific and communication-specific information.

Figure 4.2 shows RESTDL constructs as building blocks. In this Figure, a server exposes two resources; Resource_A and Resource_B. Each resource has a set of three interactions; Interaction_1, Interaction_2 and Interaction_3. For each of these interactions, Request and Response constructs are identified.

4.3.1 Request-Response Architecture

In the previous section *Request* and *Response* Constructs were introduced as parts of the RESTDL building blocks. This section introduces the architecture of these constructs to be able to represent resources' interactions in precise schema that the client can understand.

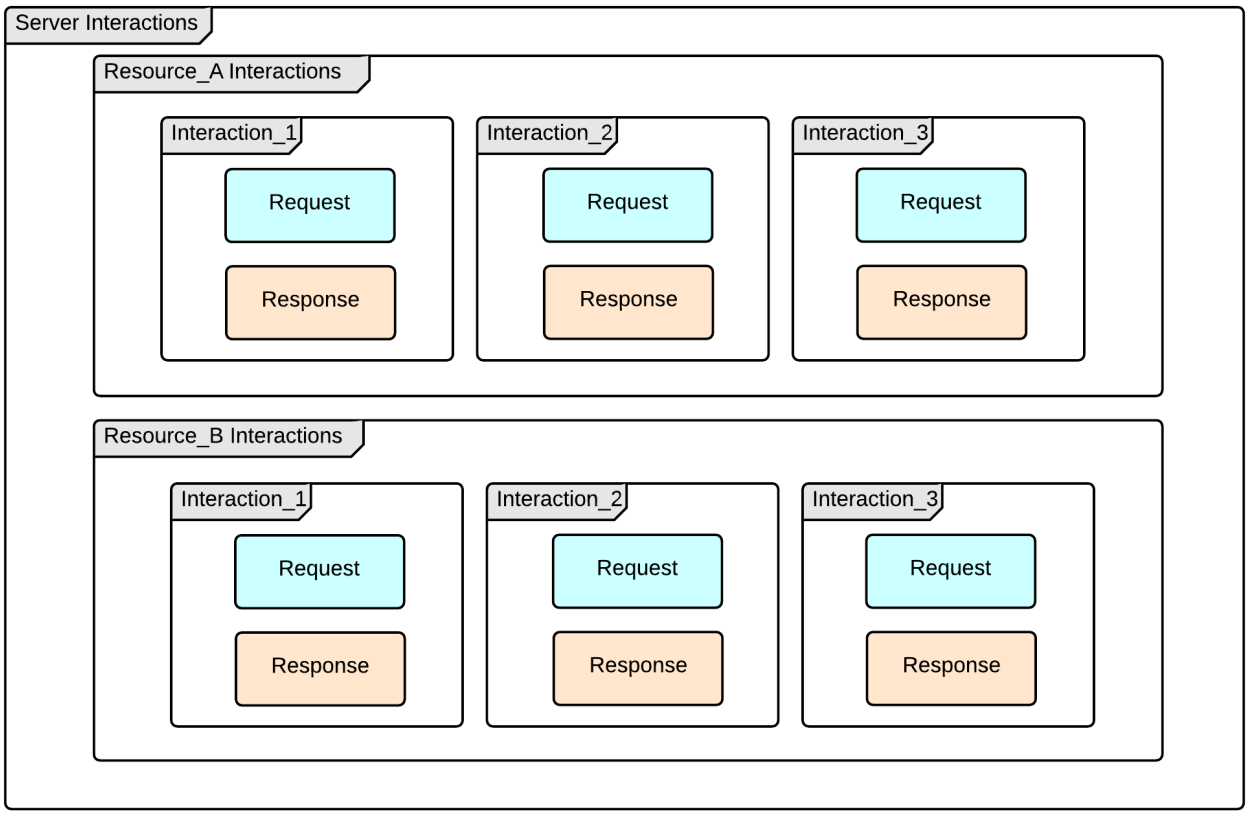

Figure 4.2: The Main Building Blocks of RESTDL.

Figure 4.1 introduced the architecture of RESTDL. It also shows how Request and Response constructs can be built. Figure 4.3 depicts the building blocks of RESTDL's Request and Response constructs. These building blocks identify, as discussed earlier, how the communication and resource information should look like. To send a message over the network, the *Message Metadata* , *Protocol Metadata* and resource's *Properties* should be identified. Collectively, they form the schema of an interaction:

Message Metadata

This part of the schema defines the necessary information needed to identify the message itself. As a result, clients and servers could understand what is in the message and how it should be interpreted. These Metadata help ensure satisfying REST's *Self-descriptive Messages* constraint. *Message Metadata* consists of the *Relation Name*, *Description*, *Media Type* and *Action*:

Relation Name Provides the name of the relation for the communicated message. *Relation Name* is used mainly to identify the current interac-

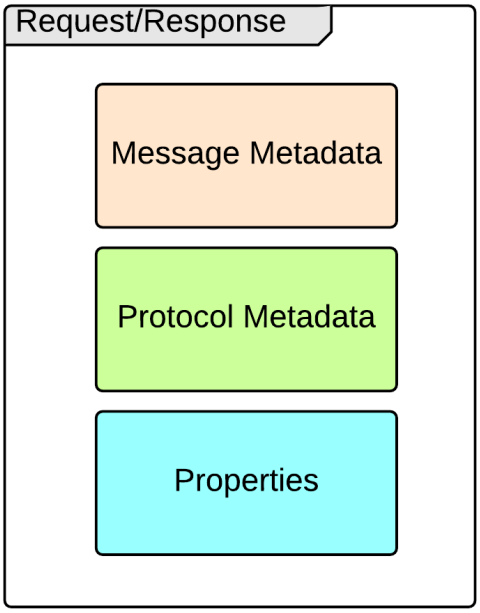

Figure 4.3: RESTDL's Request/Response Building Blocks.

tion between a server and a client. This is mainly used in conjunction with the URI of a resource when being included in a link of a response message. The *Relation Name's* importance for the client could be described as analogous to the importance of the HTTP parameters for routing the message to the corresponding implementation's method on the server side. For instance, the Relation Name identifies what activity should be executed on the client side.

Description A human-readable text to describe the message and what can be offered when sending or receiving this message. The information represented in this part of the schema could be useful in different scenarios. For instance, it could be used for *understandability* purposes in which a programmer can interpret the interaction when reading its description. In addition, it could be used for documentation autogeneration. This documentation could be Web-based or for documenting client's application.

Media Type Communicated messages between clients and servers should be represented in specific formats. The *Media Type* identifies the list of a resource's supported media types. The client decides what media type to choose following the concept of *Content Negotiation*. One assumption that was introduced earlier is that these Media Types should be hypermedia-enabled. This helps build REST Compliant Applications.

Action *This part is only available for Request Messages.* Resources can be manipulated in different interaction schemes. These interactions vary based on the data that needs to be communicated, and the *CRUD* operation needed to start the interaction. This part of the schema is responsible for specifying the CRUD operation that is needed to change the state of a resource.

Protocol Metadata

This part of the schema specifies the underlying communication protocol's specific metadata to establish and complete a successful interaction. *Protocol Metadata* is mainly used to ensure successful delivery of interaction messages. In fact, the metadata specified in this part of the schema identifies what should be supplied through the communication protocol so that the client, server, and intermediares can receive and handle the message successfully. Metadata are specified in the message as a key-value pair, and in order to represent this pair in the schema, the *Name*, *Description*, *Optional*, and *Pattern* parameter should be specified:

Name This represents the *key* of the key-value pair for representing a metadata. It should be noted that when using HTTP as a Transport System, this would represent a *Header* tag. In addition, the specified name should be uniquely identified so that the client, server and intermediaries could create and receive messages successfully.

Description A human-readable text to describe the metadata and what can be offered when sending or receiving a message that has this metadata. The information represented in this part of the schema could be useful in different scenarios. For instance, it could be used for *discoverability* purposes in which a programmer can understand the interaction when reading its description. In addition, it could be used for documentation auto-generation. This documentation could be Web-based or for documenting client's application.

Optional Some resources define metadata that are not necessarily needed to be supplied with the message. For example, a resource could add featured data to a response when specifying the type of environment the client uses in a *Protocol Metadata* . If this is not specified, the featured data are not sent back.

To specify whether the *Protocol Metadata* is mandatory for successful communication or not, the boolean *Optional* parameter is used.

Pattern A computer-expression which defines what the server or the client expect the value of the metadata to look like. This is mainly used for matching strings to ensure consistent and successful communication between a server and a client. For example, *Regular Expression Language* [32] can be used to specify what values to expect.

Properties

This part of the message represents the payload of the exchanged messages. They carry the actual data of a certain resource. When sending a request

message, *Properties* might represent the future intended state of a resource. On the other hand, when receiving a message from the server, *Properties* represent the current state of the resource on the server side. Properties are defined as key-value parameter pairs which could be identified by specifying the *Name*, *Description*, *Optional*, *Pattern*, *Value Type* and *Embed*. These parameters identify properties independent from any Media Type representation. The components that use the schema should decide on the representation of information.

Name A resource mostly represent a real-life object. When describing these objects, a set of properties could be identified. Each property could be represented in the schema by specifying its unique *Name*. For example, the *Family-Name* of a *Customer* resource could be identified as a property.

Description A human-readable text to describe the resource's property and what it represents. The information represented in this part of the schema could be useful in different scenarios. For instance, it could be used for *discoverability* purposes in which a programmer can understand the interaction when reading its description. In addition, it could be used for documentation auto-generation. This documentation could be Web-based or for documenting client's application.

Optional Some resources define properties that are not necessarily needed to be with the message. For example, when a resource's schema has been changed by introducing a new version of it, clients could still use the old resource structure if the service provider identified the new properties to be *Optional*. This parameter helps build *Backward and Forward Compatibile Applications*.

Pattern A computer-expression which defines what the server or the client expect the value of the metadata to look like. This is mainly used for matching strings to ensure consistent and successful communication between a server and a client. For example, *Regular Expression Language* [32] can be used to specify what values to expect.

Value Type This parameter identifies the type of the property. It indicates whether the property is a string, number, URL etc.

Embed This is an optional parameter that can be defined to tell the client that the property should be embedded in the response message. This entity might be on a different domain or needs a separate interaction with the server to be communicated to the client.

```
<request>
  <!-- Message Metadata -->
  <relationName />
  <description />
  <mediaTypes />
  <action />

  <!-- Protocol Metadata -->
  <header>
    <name />
    <description />
    <optional />
    <pattern />
  </header>

  <!-- Properties -->
  <property>
    <name />
    <description />
    <embed />
    <optional />
    <pattern />
    <value />
  </property>
</request>

```

Listing 4.4: Request Message Wrapper

Listing 4.4 represents a Request Message Wrapper. It shows all the attributes that are needed by the client to send a valid request.

4.4 Why RESTDL

This section introduces the advantages that REST service providers and consumers benefit from when using RESTDL as a description language for their distributed applications.

RESTDL ensures better *discoverability* by specifying how a resource-specific interactions could be performed. RESTDL also helps start interacting with the service provider without any previous service-specific training or documentation. This is due to RESTDL's model of exposing detailed information about the possible interactions with the server. The service information is exposed in a structured scheme which includes the data, metadata and human-readable documentation of each interaction and its attributes. As a result, the service's learnability and usability are improved. This can be seen as the time and resources needed to start focusing on the problems that the client application is trying to solve have been cut, and due to the efficient and consistent usage of service resources without problems.

RESTDL ensures full-REST compliant service description by applying REST constraints. The description of the service interactions in RESTDL is mainly based

on the Uniform Interfaces constraint. For Instance, RESTDL forces service provider to describe, which helps also implement, the domain model in a resource-oriented model. RESTDL doesn't specify any prescribed URI or identification structure. Instead, resources are identified uniquely on runtime. This helps build loosely-coupled clients that cope with the evolution and integration of service resources.

The description of interactions in RESTDL allows to specify a set of Media Types that could be used to represent service resources. As a result, clients can choose a suitable representation to satisfy their needs via applying Content Negotiation.

RESTDL could be communicated as REST response messages on runtime. It could be thought of as description language or a Media Type that describes everything a client needs to build request messages. These request messages are mostly ensured to be successfully completed by a server. This is due to the exposed information that describes interactions via specifying the data schema, and protocol and message metadata.

RESTDL ensures the HATEOAS constraint by allowing a hypermedia Media type to describe its resources. RESTDL improves the efficiency of applications by reducing the network bandwidth when sending messages in a hypermedia format. R. Fielding says that the hypermedia format should describe the possible interactions with the server [10]. This process, however, adds overhead to the network bandwidth via sending information that the client might not be interested in. Instead, RESTDL ensures having references to the description of interactions using the Link tags in the response messages, and by applying the HATEOAS model, clients can fetch the necessary interaction description from the server. This model of fetching resource representation results in a significant bandwidth reduction and improvement in client and server applications.

RESTDL is based on standard REST operations which are Create, READ, UPDATE, and DELETE. This could be utilized along with standard resource's schema (e.g. schema.org) to describe multiple domain, multiple service interactions. It also defines a possible way to design custom domain-specific interactions. In addition, RESTDL is protocol independent. Specifying the protocol could be done on the resource level by explicitly defining it in the protocol metadata.

RESTDL could be defined as a contract to be communicated for building and integrating Enterprise Applications. In the enterprise world, applications are often integrated with other complex systems that have been developed by other companies. In such environments, communication between different partners could be very challenging due to the number of involved parties. Therefore, a contract that is considered as a way of communication between partners would be the best solution to meet business and development goals.

RESTDL can be used to generate server and client stubs. It could be also used with Code-First or Description-First models. Auto-code generation is optional. However, using code generation could reduce the wasted time on repetitive and boring tasks for REST applications development. It also could improve the quality

because applications would have consistent quality restrictions across the whole application.

Chapter 5

HypREST

This chapter discusses the main goal of this work; the design of a REST-compliant client framework. It introduces the components, requirements and design details that result in building a framework for developing REST client applications. Based on the benefits of HATEOAS that was introduced in section 2.4, this framework introduces an architecture for a generic framework of the development of compliant REST clients.

5.1 REST Client Applications Components

Developing REST client applications involves implementing different components that should be responsible for handling *Exchanged Data*, *Communication*, and *Client-Custom Logic*.

Exchanged Data refers to the data models and their representations that could be beneficial for the client application. These representations are mainly exchanged with the server to manage the states of server's resource data and client's application data.

Dealing with today's REST services can be considered a challenging task; it involves sending, receiving and understanding different representations and data formats when communicating with different service providers. Therefore, the client framework should be able to support wide range of representations and include extensible functionalities for new representations. This way, generic frameworks could be built to support and build wide range of REST client applications.

In order to be able to exchange information, **Communication** with the service provider should be managed. The Communication component should be able to control sending request and receiving response messages between the client and server. It also includes managing communication sessions and overall client-server interactions.

Client Applications are developed to achieve goals. These goals are often different when developing different applications that consume the same REST service. The variations of goals are implemented using the **Client Logic** component. This component executes the application's specific business logic to achieve its goals. In addition, it is responsible for controlling and communicating with other applica-

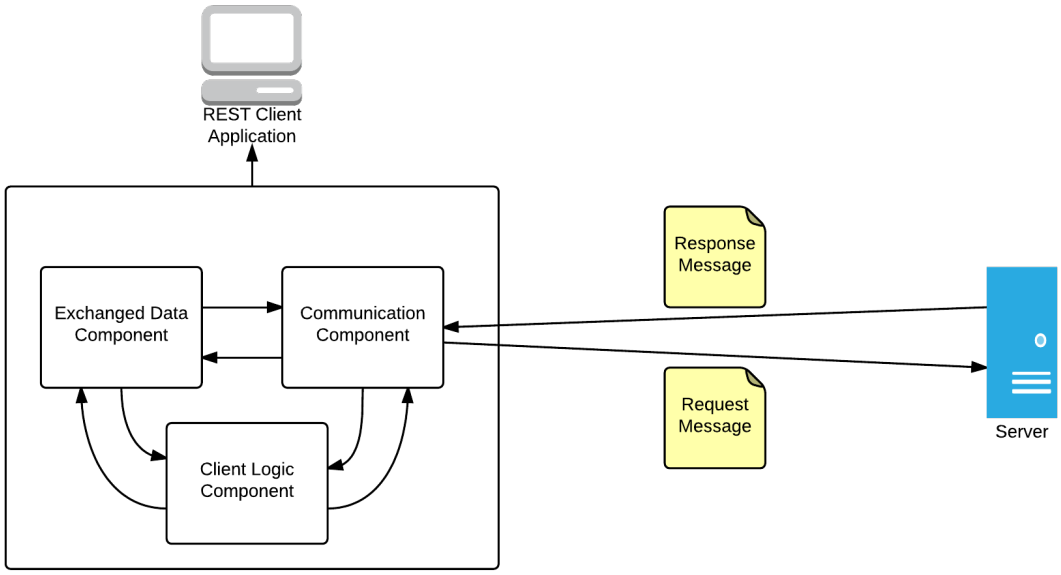

Figure 5.1: REST Client Components.

tion's layers. For instance, it triggers the Communication component to start an interaction with the service provider.

Figure 5.1 depicts the components of REST client applications and their interactions. In the shown figure, the Client Logic Component is responsible for all the business activities that define the client application and its goals. It is mainly responsible for executing the custom-client logic. When the Client Logic Component decides that there is a need for an interaction with the server, the data that represents the interaction is manipulated in the Exchanged Data Component. This way, the data needed for the interaction is made ready to be communicated with the service provider. To do that, this component is responsible for the transformation, and translation of data representations to make it understandable by different interactions parties. Moreover, the Communication Component is responsible for interacting with the server. This includes sending and receiving messages, and interacting with the Exchanged Data and Client Logic components to notify them about the availability of interactions.

5.2 HypREST Layers

To achieve the client application's business goals by managing the development of *Exchanged Data*, *Communication*, and *Client Logic* compnents that are discussed in the previous section, *HypREST* was designed to offer these components as a framework for developing REST-compliant client applications.

HypREST consists of integrated layers that manage the complete development lifecycle of REST client applications. These layers are *Message Interfaces*, *Programmer Interfaces*, and *Client Workflow*. The Developers of REST client applications can use the layers to build mature and compliant REST applications that serve the needs of their technical and business goals. In addition, HypREST helps apply REST's best practices and constraints on the client side. This is achieved by restricting the developers from applying bad practices of client development. For example, client applications are supplied only with the root URI, and assumes that the HATEOAS constraint is applied on the server side.

The layers of the HypREST framework are represented in building blocks as it can be seen in figure 5.2.

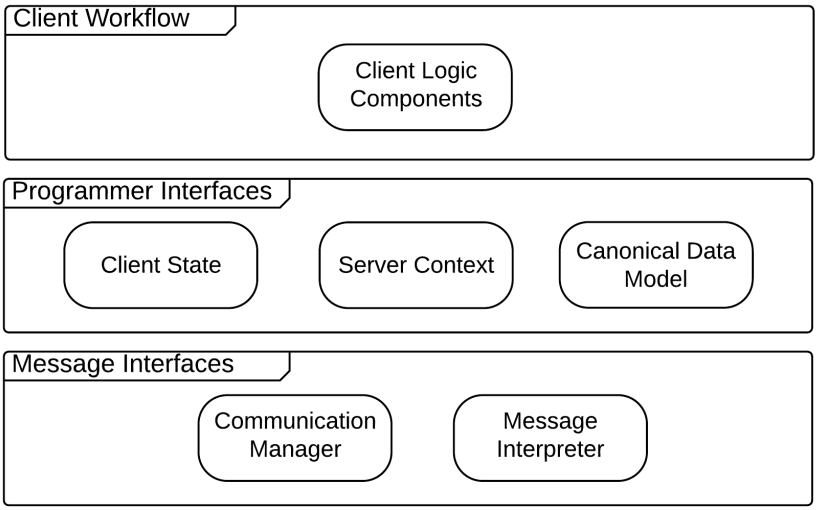

Figure 5.2: Building Blocks of HypREST framework.

The layer of **Message Interfaces** encapsulates two of the REST client's components that are discussed in section 5.1; *Exchanged Data* and *Communication*. In this layer, client-server interactions including the communication, and transformation and interpretation of exchanged information are managed and delivered to other components and layers.

The communication between the client and server should always start by the client. This is done by targeting resources that are offered by the REST service provider on the server side. These resources might be represented in different formats which are called Media Types (see 2.3.3). Consequently, request and response messages could be represented in different Media Types depending on the offered formats for each resource. Hence, the client can apply the concept of Content Negotiation to choose between the available media types for each resource on the server

side. It should be noted that the framework does all the representation's transformations automatically. However, client-specific preferences could be applied.

The layer of *Message Interfaces* manages the transformation of client's data models to different media types. As a result, the server can understand the message semantics when sending a request message for a targeted resource. Similarly, *Message Interfaces* transforms the response message, which could be represented in different media types, to data models that the client application can interpret and manipulate. This process helps clients and servers communicate and share knowledge in a consistent and understandable manner. In addition, it enables supporting multiple resources and services, which results in a generic framework design that supports the development of different client's domains. For example, a mobile-based client application might prefer data representation that does not consume data bandwidth. Similarly, An enterprise application might consume the same resources with more concrete representation.

To enable message transformation and consumption, a *Canonical Data Model* and associated *Media Types Translators* are used to transform resources' representation to a unified representation that could be used by the client application. The Canonical Data Model represents a unified data representation for the client application. Media Types Translators represent the engines that are responsible for transforming dedicated message representations to and from the Canonical Data Model.

Furthermore, *Message Interfaces* layer manages the communication with the service provider. In fact, it hides the complexities of managing communication sessions with the server. This could be achieved by a *Communication Manager* that is responsible for sending and receiving messages between the client and server and manage the underlying protocol details for communication.

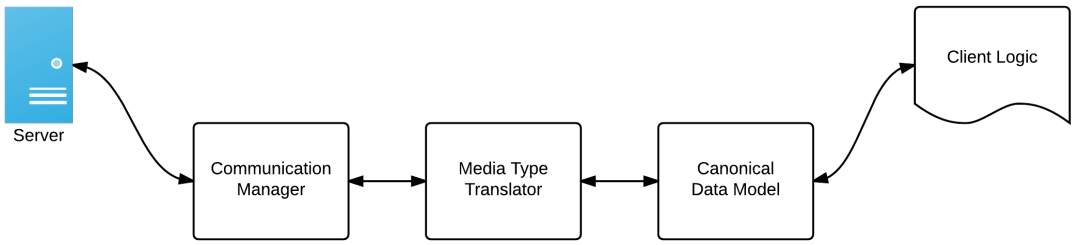

Figure 5.3: Data Flow between Client and Server.

Figure 5.3 depicts the flow of data between the client and server applications. When requesting a specific interaction by the client, the data to be communicated is represented in a Data Canonical Model. This data is then fed into the Media Type Translator to represent the represented data in the appropriate data repre-

sentation that is understandable by the targeted resource. Then the representation is sent to the server via the Communication Manager. When receiving a response from the server, the data flow takes the reverse path. That is, the Communication Manager receives the response message, then feeds it into the Media Type Translator. After transforming the message to the unified Data Canonical Model, the data is fed back to the client logic.

Client Workflow is a layer that represents the custom business logic of client applications. Each application has a set of goals that need to be realized via the execution of software logic. In general, the software logic of the client could be seen as an execution of a workflow sequence. This workflow might be realized in different forms based on the goals of the client (see 3.1). For instance, the logic could be based on Object Orientation, or Sequential Paradigms. To help execute the client's custom workflow while considering the states of resources on the server, Client Workflow layer helps build software logic that could be seen as a set of workflow activities which work in isolation from each other. The workflow activities are executed based on conditional logic that determines if the activity should be executed.

Programmer Interfaces represents a mediator layer between *Message Interfaces* and *Client Workflow* layers. This layer encapsulates all the needed functionalities by the application developer. These functionalities help work with the Communication Manager and access messages data in the *Message Interfaces* layer. Programmer Interfaces layer also offers interfaces which represent the *Server Context*. These interfaces allow navigating through the possible state transitions of server resources. To enable state transitions, they should be represented in the hypermedia links of the different media types of response messages. Programmer Interfaces layer also works with the *Client Workflow* layer to manage the state of the client application. For instance, it offers developers the needed interfaces to represent the client application's states. As a result, the software logic of client workflow could be driven based on the available interactions and resources. In short, this layer interprets the context of server's resources, helps work with a canonical data model, and offers a set of interfaces to manage client application's state.

5.3 Communication Manager

To realize REST's constraint of Client-Server interactions, the *Communication Manager* component should be developed to manage the interactions via sending and receiving messages. This section discusses a proposed design for the Communication Manager component that helps drive REST-compliant client application.

Communication Manager interacts with the server via sending request messages that hold information to manipulate or receive infomration from the server. When using HTTP as a Transport System for REST services, the commuincation manager is responsible for sending the CRUD interactions. Similarly, the communication manager should be able to receive response messages from the server and forward it to other interested components (e.g. Media Type Translator). To manage the interactions between the client and servers, the communication manager consists of an HTTP Client and a Request Message Builder as it can be seen in

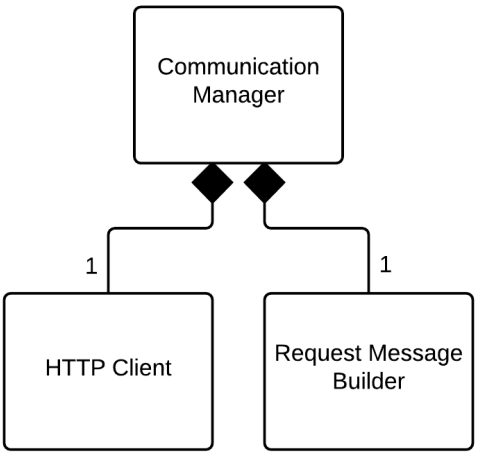

Figure 5.4: Communication Manager Components.

Figure 5.4:

HTTP Client: Because HTTP is heavily used as a major Transport System for REST services, this component is essential to manage all the connections between a client and server. HTTP Client is responsible for hiding all the communication complexities that might face the developer of a REST client application. For instance, this component should be able to send HTTP messages that identify the intended operation to be executed on a specific resource. This could be achieved via resource identification using a unique URI. In addition, HTTP Client should be able to manage all the communication sessions, including communication timeouts, request-response message identification, and control data streams when writing and reading interaction messages.

Request Message Builder: to send a request message to the server, the message has to include all the necessary information that the server needs to successfully process the request. This could be achieved using the *Request Message Builder*. This component allows creating messages by building the structure of the message. The structure of the message includes protocol- and resource-specific information that might be needed to complete an interaction. For instance, CRUD operations have to be identified in the HTTP messages using POST, GET, PUT and DELETE methods, which could be achieved using the Request Message Builder. In addition, the data and metadata of the request messages are integrated in the message with the correct representation. After building the structure of the message, this component feeds the message to the HTTP Client to send it to the server.

Figure 5.5 depicts the data flow in HypREST framework. The data is repre-

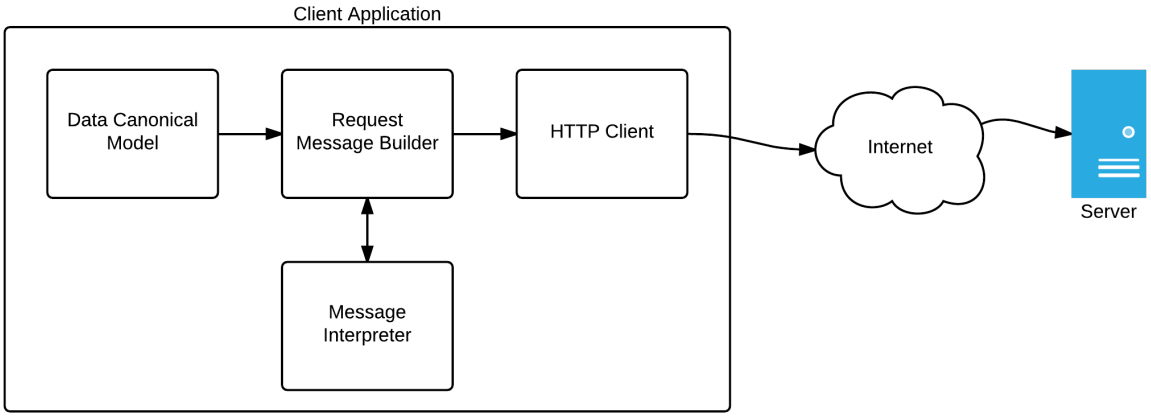

Figure 5.5: Communication Manager Data Flow.

sented in a normalized scheme in the Canonical Data Model. When a request is needed, the data model is passed to the Request Message Builder. The data might in turn be fed into the Message Interpreter (will be discussed in 5.4) to format message representation. Then, HTTP Client receives the message information to send it to the server.

5.4 Message Interpreter

Consistency could be correlated with the quality of work over time. In an architecture where there is a great potential of variability and subsystem changes, the need for a consistent methodology of dealing with changes arises. One of the variability factors in REST applications is the representation of exchanged messages. This section describes an architecture that could be used to build a generic framework that could support different message representations.

In REST architecture, the service designers choose to support a media type, possibly multiple ones, that suits the application and its constraints. On the other hand, it could be painful for client application developers to build repetitive tasks to achieve the same goal. As a result, a mediator that translates different data representations to a consistent data model that can be dealt with in an easy and unified manner is needed.

Request and Response messages that are communicated between a client and server could be represented in the same structure for the client application. However, other constraints could influence their preferred representation. For example, an XML representation could be suitable for a client application because of its simplicity and availability of supporting tools. On the other hand, due to its high bandwidth needs, it would be not considered the best for sending its stream over

the network for mobile applications. To solve this incompatibility issues, a *message interpreter* is needed that work in conjunction with the concept of Content Negotiation.

The *Message Interpreter* is responsible for receiving request and response messages from the framework's components, and transform their representations to a suitable format that is needed to perform a specific interaction. For example, a server could send a resource's state via XML representation. The Message Interpreter receives this message from the Communication Manager component and transforms the message to an understandable Canonical Data Model that the client application understands. Then, the Message Interpreter forwards it to the next component that needs to deal with the message. Similarly, the client application might need to send a request to the the server. The client can represent the information that is needed in the interaction in a normalized structure using the Canonical Data Model. To send a request message that the server understands, the canonical data model is transformed into a resource's suitable representation before sending it through the communication manager. To do this transformation criteria, *Media Type Translator* and *Data Model Normalizer* components are needed as depicted in figure 5.6:

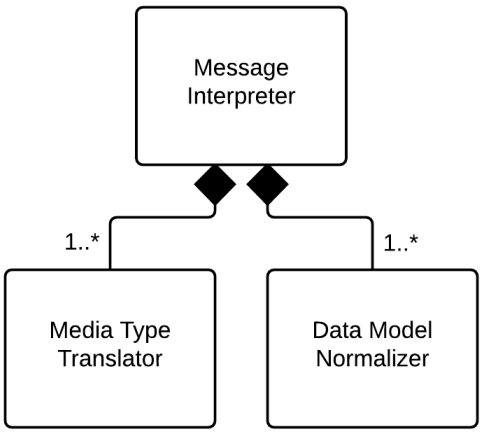

Figure 5.6: Message Interpreter Components.

Media Type Translator: Resources could be represented in different formats. In order to support these formats and understand the data included in the communicated messages, a component that is able to understand the metadata and structure of the data representation is needed. *Media Type Translators* are components that represent the backbone of data translation. Each component of these translators is dedicated to one media type. As a result, this component receives the message payload, computes its data and metadata and passes these data to

the next component which is the *Data Model Normalizer*.

This component has to be extensible; new media types could be designed that need to be supported. As a result, Media Type Translators should be added to client and server applications to support wide range of media types. This results in a REST framework that supports not only the media types that are supported by the framework itself, but it also give the possibility to the developers to enlarge the translators set and support even proprietary media types.

Data Model Normalizer: After translating and understanding the data included in the media type of the communicated message, a consistent and easy to use representation has to be used to represent exchanged information. The *Data Model Normalizer* works together with the *Media Type Translator* to transform the actual information in the exchanged message to and from a Canonical Data Model. This way, the developer of client applications as well as service designers can implement and design their services without worrying about the complexity of data representation understandability.

5.5 Canonical Data Model

A resource of a REST service consists typically of a set of data entities that represent the behavior of a real world object. These data entities consist often of key-value pairs; the key represents the name of an object's attribute, while the value represents the attribute's state. Data entities could represent different data fields, ranging from a simple primitive type to complex hierarchical sub-entities.

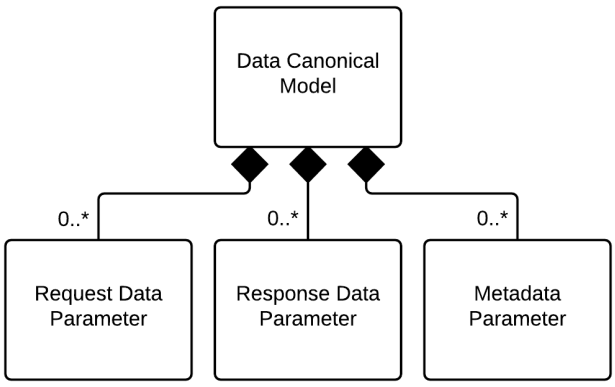

Figure 5.7: Data Canonical Model Components.

The data entities of a REST service's resource could be represented in different media types. As a result, data representations could be more beneficial if they are

transformed into a generic and normalized data model that the client can consume. Therefore, a *Data Canonical Model* has been designed as a way to hold data on the client side. This component should be able to hold the necessary information to uniquely identify a resource and its data entities. The attributes that identify a resource in a Canonical Data Model are:

Media Types: different resources can produce and consume different media types. A Data Canonical Model holds information about the available media types that could be communicated with when sending or receiving messages. In fact, these information are important to apply the concept of Content Negotiation.

Choosing a media type might be based on different factors when building different client applications. For instance, the choice could be based on the functionality of the resource, its usage and its available interaction that the server offers. Moreover, media type selection also depends on customized client preferences. For example, data usage, availability of media type translators, interpretation and tools learnability, efficiency, and features support are a set of important factors that client applications developers as well as service providers take into considerations when choosing an appropriate media type to represent a resource.

Media Types in HypREST are assumed to support HATEOAS; this assumption ensures that server's responses should include links to the possible transitions of the server's workflow. Applying this assumption helps build compliant and mature REST applications.

Relation: REST services might consist of different resources that form the functionality and workflow of a REST service. Moreover, each resource could expose different interactions that manipulate its state in a different way. For example, *Create* and *Read* are different interactions that could be done on the same resource.

When developing client applications that consume the exposed resources of a REST service, interactions of the service's resources have to be identified and interpreted in a way that enables proper workflow execution. This identification could be achieved via a *Relation* name. The name of the relation could be used to identify the current interaction of the server resource. In addition, the relation's name could identify the current client's application state. This could be achieved as the relation might be combined with other state identifiers to form a generic application state. The generic state could be used to identify the next possible state transitions on the client or server sides.

URI: a client can interact with a resource only when it can identify it. Using HTTP as a Transport System to interact with REST services enables identifying REST resources using a unique URI for each resource. When the logic of a client application decides to interact with a specific resource to achieve a goal in its workflow, this interaction could be possible by sending a request message to the resource's endpoint which is identified using its URI.

Properties: When interacting with a REST service's resource, information is exchanged between the client and sever. This information shares the knowledge of

a client application's state or a REST service's resource state between each other. To represent the state so that it could be interpreted, key-value pairs could be used to identify the *Properties* and *Attributes* of a specific object. The client application could use the properties of the resource to identify the next possible states in its workflow. On the other hand, the server could use the properties value to change the state of its resource, or to send the value of a resource's attribute state in a response message.

Sub-Entities: REST resources could be designed in a hierarchical structure. This means that properties of a resource might represent the state of another object that has an association with the encapsulated resource's representation. To represent this association, *Sub-Entities* could be used as a key-value properties that represent the state of a child object. For example, a *building* object could have a property of five floors and might include a sub-entity object which represents a *flat* with a property of three *rooms*.

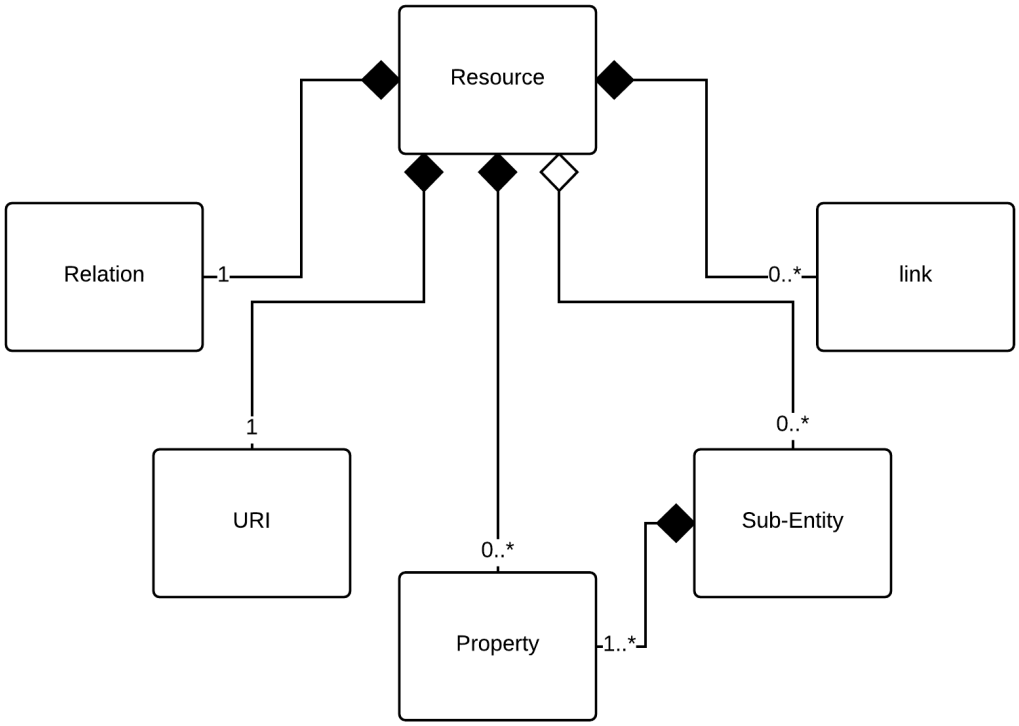

Figure 5.8: The architecture of the Canonical Data Model.

Links: developing compliant REST applications requires that the server drives the state of the client applications using the HATEOAS constraint. To enable this concept, hypermedia media types could be used to include the possible transitions

in the server's response messages. Hypermedia media types could include links to the next possible interactions. These interactions could be used by the client applications to drive its application state. Similarly, these links help guide the client application to change the state of remote resources.

Figure 5.8 shows the architecture of the *Canonical Data Model*. Basically, each resource could be represented using a unique URI and relation. In addition, sets of Property values, Sub-Entities, and Links hold the representation information of a resource object. Listing 5.1 shows an example of how an object class could be represented in a Canonical Data Model using a set of meta-information that will be described in details in section 6.1.3.

```
1  @ResourceModel(relations={"relation_1","relation_2"})
2  class ResourceExample {
3
4    @ResourceRelation
5    String relation;
6
7    @ResourceProperty(rel={"relation_1"})
8    String property1;
9
10   @ResourceProperty(rel={"relation_1","relation_2"})
11   String property2;
12
13   @ResourceProperty
14   String property3;
15
16   @ResourceHref
17   URL href;
18
19   @SubEntity
20   ResourceExample_2 instance;
21
22   @SubEntityList
23   List<ResourceExample_3> subResources;
24
25   @ResourceLinkList
26   List<Link> links;
27 }
```

Listing 5.1: Client's Annotations Example

5.6 Server Context

REST services are built in a resource-oriented model. The resources of a REST service could be seen as a finite state machine in which requesting one resource yields transitioning to another state if the request was performed successfully. In addition, the architecture of REST applications should apply the HATEOAS constraint which offers client applications the ability to be driven via following possible hypermedia transitioning links.

To be able to drive client applications based on the state of the resources on the server side, the information of the possible transitions have to be visible to the client applications. As a result, components that offer the functionality of reading,

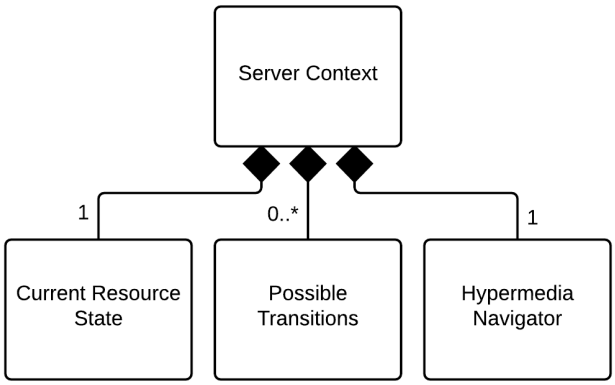

Figure 5.9: Server Context Components.

understanding and storing possible state transitions are needed. To do that, the *Server Context* components have to be designed.

Server Context consists of three main components that represent the behavior and possible interactions that could be communicated with the client. These components as shown in Figure 5.9 are *Current Resource State, Possible Transition* and *Hypermedia Navigator*:

Current Resource State represents the last interaction that happened between the client and server. When a client requests a specific resource, or sends a request to manipulate the state of a resource, the interaction type should be stored in the client's local storage. This stored state could be used to further change the state of the client application in the future.

Changing the state of the client application could depend on many factors. One of these factors could be the state at what it was before. In this case, the new intended state to be transitioned to is said to be correlated with the previous state. That is $f : X \to Y$ where X is the current state, Y is the state to transition to, and f is the condition that enables transitioning.

Possible Transitions represents a component that holds the next possible transitions that are allowed by the server. When interacting with a compliant REST service that supports full maturity by enabling HATEOAS, the possible state transitions should be included in the server responses as hypermedia links. The client should be able to read, understand and decide on what next state it should transition to.

Hypermedia Navigator represents a mediator component between the client

application's developer and the REST application. *Hypermedia Navigator* includes interfaces that could be used by the programmer to examine the available state transition and decides on which one to follow. This could be used when building the logic that defines the condition at which next state the client application is shifting to.

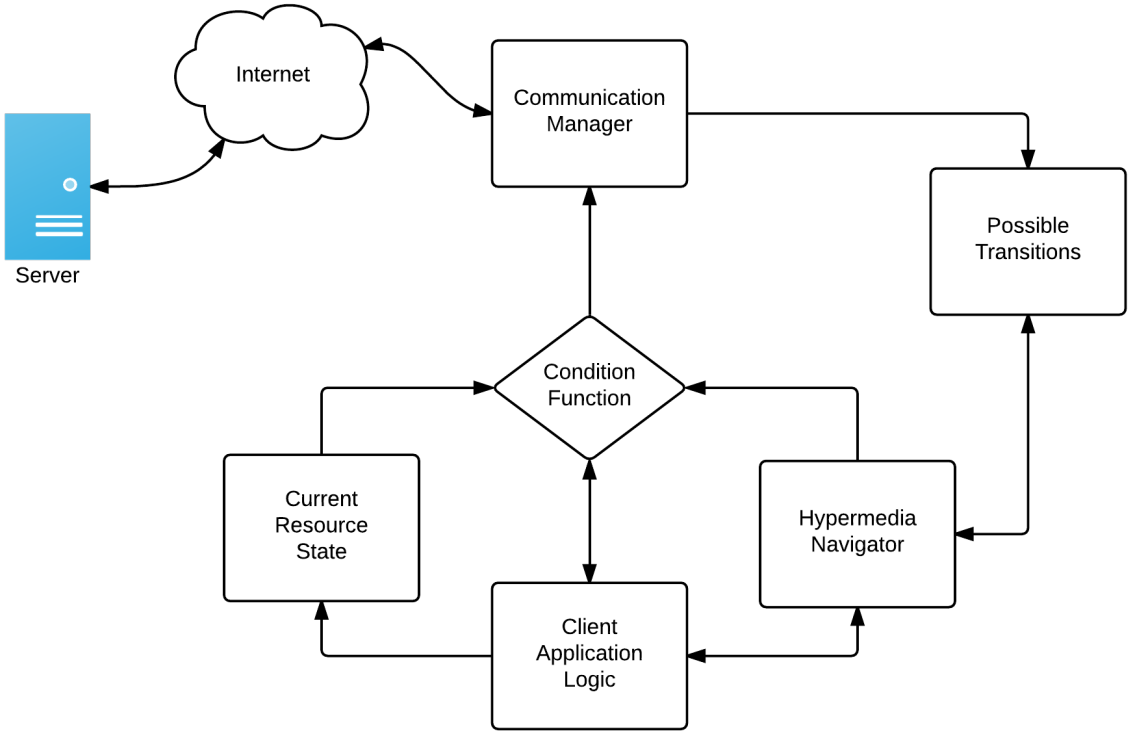

Figure 5.10: Server Context Data Flow.

Figure 5.10 shows how data flows to drive the next state of client application based on what the server offers. When the Communication Manager interacts with the server, the server responses should contain the next possible states, which should be stored in the *Possible Transitions* component. The next possible transitions are communicated to the client through an easy to use interfaces that could be used by the programmer, this should be achieved via the Hypermedia Navigator. To drive the next possible state, the combination of *Current Resource State*, possible transitions in *Hypermedia Navigator* interfaces, and the custom *Client Application Logic* are examined in a *Condition Function*. If the condition function decides to change the state of the client application, the change signal is either sent to the Communication Manager and to the Client Logic to change a resource's state and update current states, or it is sent only to the Client Logic to update a local Client state.

5.7 Client Context

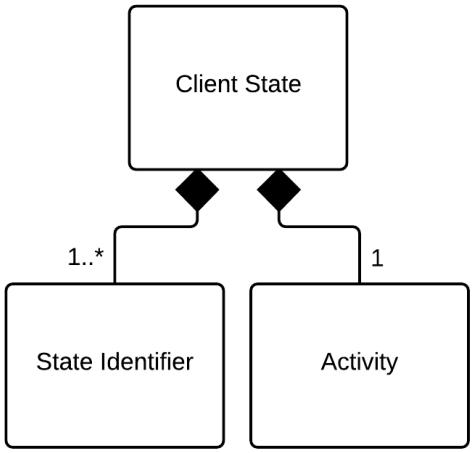

Figure 5.11: Client State Components.

Client Applications are built to achieve different goals. When a REST client application is designed, its logic could be divided into modules that are executed based on different conditions. For example, the availability of a specific interaction on the server side could be a condition to execute a specific logic. Another example could be based on a specific client event or value that controls the execution of different modules.

To be able to build REST compliant client applications that consider the availability of specific interactions while responding to internal events, *Activity-Based Clients* could be built. Such client applications are developed using a set of custom business logic entities that are executed when a dedicated event occurs. Events in the context of client applications could depend on many factors. For instance, availability of server interactions, availability of specific resource's attribute value, internal computational value, external physical event etc. could be considered to start the execution of their dedicated activities.

Using the *Activity-Based Clients* model to develop client applications, the logic of such applications could be thought of as a *Finite-State Machine* in which activities are the states, and events are the state's transition condition.

Building client applications based on the Activity-Based model yields many advantages. For instance, the overall application could be designed of decoupled modules that could be executed independent from each other. This helps build applications that are more tolerant to service changes. If a REST service provider

decides to drop an interaction of a resource, then the client application will not be affected because it will not encounter the relation of the interaction in the response messages. As a result, the framework will decide to execute another activity. In fact, the business logic that is responsible for achieving a specific goal will be executed in isolation from any other module, which results in loosely coupled application's architecture. In addition, using activity model, the separation of concern is applied; the developer of the client application should focus on choosing the events that trigger the execution of the activity while the framework is responsible to iteratively check for the next activity to be executed. As a result, hypermedia client applications could be built in an easy and improved user-experience.

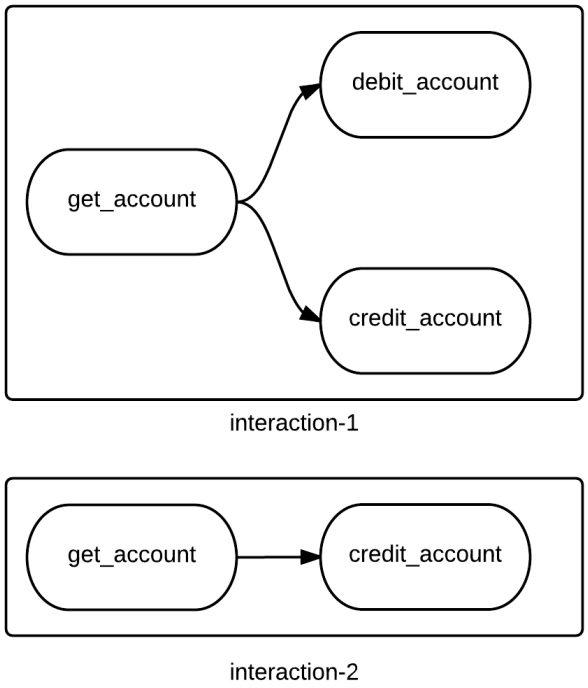

Figure 5.12: An example of Hypermedia-driven interactions.

An example that could illustrate the benefits of using hypermedia with the Activity-based model is shown in Figure 5.12. In this figure, interaction-1 represents a REST interaction to retrieve the information of a bank account. When hypermedia is applied to the response messages, the REST service should return the next possible interactions. In this case, if the bank account has enough money, the *debit_account* interaction's relation is returned in the hypermedia links along with the *credit_account* interaction's relation. If the account doesn't have enough money, then the client application should not be able to issue debit requests. As a result, the *debit_account* interaction's relation is not returned as it is shown in

interaction-2 in the figure. This way, the client application will be able to decide on what activities to execute. For example, when there is not enough money in the account, then the client application should not be able to issue transactions that require money from the account, which could be represented in dedicated activities in the client application. Instead, the activities that could credit money to the account could be executed.

To support the development of client application that could be divided into activities, HypREST defines two components that help drive the execution of the client's logic. These components are *State Identifier*, and *Activity* as shown in Figure 5.11:

State Identifier component represents the condition that triggers the execution of a dedicated logic. Identification of states should be carefully considered in order not to have any conflicts. For example, the framework should not face a situation in which there is two state identifiers that could trigger two different activities at the same time. This results in a non-deterministic state flow.

A specific state could be executed based on multiple state identifiers. For example, a state activity to order a meal at a restaurant could be triggered using a drive-through service, online order, or in-house order. When evaluating these state identifiers, any of them could activate the same state.

Activity represents the logic that should be executed when a specific event occurs. This logic could be implemented as an interface to the user to wait for his/her inputs, calculate a specific value, populate a database etc.

An activity could be as simple as requesting a user input, or it could have more complex algorithm and business logic.

5.8 Client State Dispatcher

When building an application that is subdivided into multiple activities, there should be a mechanism to evaluate events to trigger the execution of the activities. *Client State Dispatcher* helps evaluate events and starts their dedicated activities as is described in the previous section.

When an activity finishes its execution, the *Client State Dispatcher* takes control of the client application, evaluates events to identify the next application state, and then executes its activity. To perform this logic, the following components are needed:

Next State Processor represents the logic that evaluates the different activities' identifiers and selects the next state to be executed.

Activity Executer represents a component that is called when identifying the next possible state. This component fetches the activity that represent the state and executes its logic.

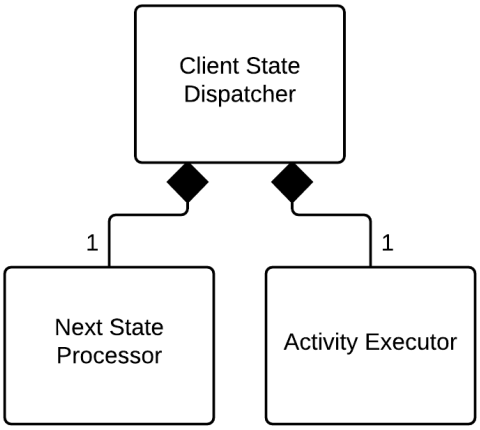

Figure 5.13: Client State Dispatcher.

5.9 Hypermedia and Client-Server Workflows

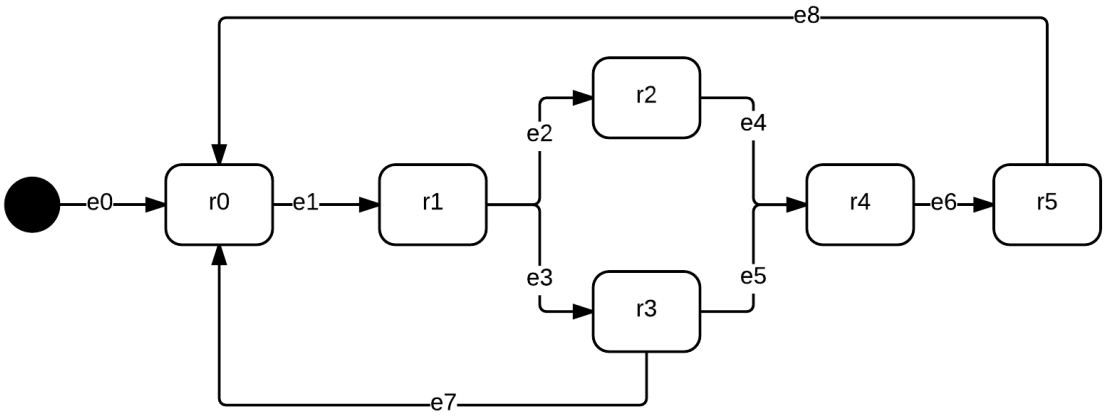

Figure 5.14: Resource Workflow of a REST service.

REST services consist of a finite but not necessarily fixed set of resources. These resources are connected to each other to form a *Resources Graph* (see 2.3.4). In

fact, the resources graph form a workflow to serve clients' requests as shown in Figure 5.14. In this figure, a set of resources {r0, r1, ... , r5} are exposed to the service's client. In addition, a set of trigger events {e0, e1, ... , e7} cause the transitioning between resources. For example, when a client application interacts with resource r0, then transitioning to r1 requires triggering event e1. Similarly, each resource in the resources graph is transitioned to by triggering a dedicated event. The combination of resources with their trigger events enables representing the resources graph as a workflow where the resources serve as unit of work and events serve as conditions for transitioning between these units. To interact with a specific resource, the client application has to send a request to this resource when its interaction is available and visible to the client. This request could serve as the trigger event that causes transitioning between resources. In fact, the trigger event relies on the REST's Uniform Interface constraint. This includes, the data, metadata and resource identifier. Moreover, the HATEOAS constraint states that interactions with a REST service should always start from the root resource which is identified by the root URI when using HTTP as a Transport System. As the client sends a request to the REST service, it targets a unique resource that is identified by its URI. Therefore, changing the resource's state by moving to a new one is a *deterministic transition* that is identified by the data and metadata in the request message, and the exposed graph of resources of the REST service.

In this section, a simplified formal model of the resources graph and their clients' interactions is derived. This formal model serves as a starting point for a proposed, and more formalized future-work. This formal model helped in the derivation of architectural requirements for the proposed HypREST solution. The proposed formal model could be represented using a *Deterministic Finite-State Automaton*:

The 5-tuple $S = (R, \sum, \delta, r_0, F)$ represents a REST service's state machine where:

$\mathbf{R} = \{r_0, r_1, ..., r_n\}$ is a finite set of non-terminal states. These states represent the *Resources* of a REST service.
$\boldsymbol{\sum} = \{e_1, e_2, ..., e_n\}$ is a finite set of state events. These events represent the *Interactions Requests* that could be issued by service's clients.
$\boldsymbol{\delta} : R \times \sum \rightarrow R$ is the *Transition Function* that could move the state to another one. This represent transitioning to another REST's resource. $\boldsymbol{r_0}$ is the root resource of the REST service.
$\boldsymbol{F} \subseteq R$ is a subset of R which represents *Terminal States*. In the context of REST, this could represent the resources that have transitions to only the root resource. For example, in figure 5.14 r5 could represent a terminal state because the workflow has to be started from the beginning again. This could mean reaching a goal, and then the application has to start over again.

Interacting with a REST service should start from the root resource r_0. Targeting a specific resource r_i and interacting with it results from a series of interactions represented by a series of trigger events $e_0, e1, ..., ei$ that starts from resource r_0 to r_i. The series of interactions results from the client's interaction requests which are based on the server's response messages which include the next possible resources to interact with. That is, $\forall e \in \{e_0, e1, .., e_i\} : T = (H, P, L)$ where T is a 3-tuple

that represents a response message and:

$\boldsymbol{H} = \{h_1, h_2, ..., h_i\}$ is a finite set of *Response Message's Metadata.*
$\boldsymbol{P} = \{p_1, p_2, ..., p_n\}$ is a finite set of *Response Message's Payload Data.*
$L = \{l_{r_m}, l_{r_{m+1}}, ..., l_{r_i}\}$ is a finite set of hypermedia links to next possible resource transitions.

The trigger events which represent request messages could be represented as a 3-tuple $\boldsymbol{e_i} = \boldsymbol{(I, M, D)}$ where:

$\boldsymbol{I}$ is the resource's *Identifier* which is included in one of the previously received response message's links l_i.
$\boldsymbol{M} = \{m_1, m_2, ..., m_n\}$ is a finite set of *Request Metadata.*
$\boldsymbol{D} = \{d_1, d_2, ..., d_n\}$ is a finite set of *Request Payload Data.*

To transition from r_{i-1} to r_i, the trigger event e_i which represents a request message should be received by the server, successfully processed, and if the request enables resource transitioning, then the server will move its state to r_i. This is the mechanism that the *transition function* uses to decide on the next state.

Similarly, REST client applications could be built using a finite set of *states* that represent a graph of logic components. These states are triggered using a set of conditions which should be examined before start executing the state's logic. The conditions that trigger the execution could be local or remote events. In fact, the combination of states and their respective triggers form a workflow that executes the goal of REST client applications. Figure 5.15 depicts an example of such workflow. In this figure, a set of logic states {s0, s1, ... , s8} represent the logic components of the client application. Moreover, the set {t0, t1, ... , t11} represent the trigger events that cause the logic states to be executed. To move from state s_n to s_m, the event t_m has to be triggered. These conditional events along with the logic states form a combination that represents a client application's workflow.

The simplified model of REST client applications could be represented using a *Deterministic Finite-State Automaton*:

The 5-tuple $C = (S, \sum, \delta, s_0, F)$ represents a REST client application state machine where:

$\boldsymbol{S} = \{s_0, s_1, ..., s_n\}$ is a finite set of non-terminal *States.* These states represent the client's logic components that execute business logic.
$\boldsymbol{\sum} = \{t_1, t_2, ..., t_n\}$ is a finite set of conditional *Trigger Events.* These events decide whether to start executing states' logic or not.
$\boldsymbol{\delta} : S \times \sum \rightarrow S$ is the *Transition Function* that could change the client's state.
$\boldsymbol{r_0}$ is the initial state of the REST client application.
$\boldsymbol{F} \subseteq S$ is a subset of S which represents a *Client Final States.* In the context of a client application, these states could represent many events. For instance, it could represent re-opening or closing the application.

Transitioning from s_{i-1} to s_i requires evaluating a trigger event t_i that de-

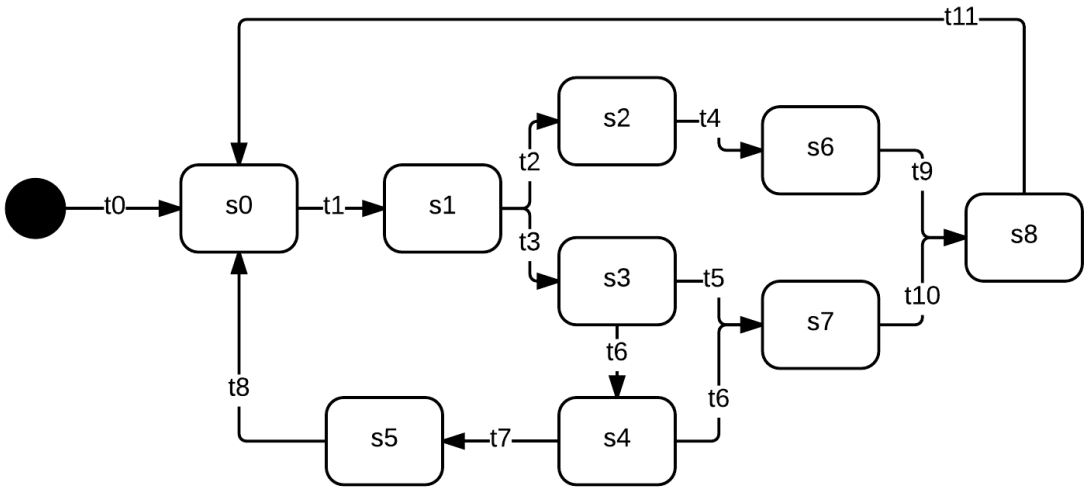

Figure 5.15: Client Application states Workflow.

cides whether the transition is possible or not. t_i could depend on many factors that control state changing. For instance, t_i could be represented as a 4-tuple $\boldsymbol{t_i = (l_p, l_c, s_c, x_T)}$ where:

$\boldsymbol{l_c}$ is the previous REST's hypermedia link that was followed which targets a specific resource.
$\boldsymbol{l_c}$ is the current link that the client's state is following. This link is received from the previous response message and targets the next possible resource on the server side.
$\boldsymbol{s_c}$ is the client's current state.
$\boldsymbol{x_T}$ represents an external, computational or any event that affects triggering states at a specific time instance.

The workflows of both REST client's and servers are tightly correlated. That is, changing the state on the server side is based on a state's logic on the client side that issues a request message by following a server's previous response message's link. Similarly, the client's next state could be based on the response message that's received as a correlation of the request message.

States are represented as Activities in HypREST. Figure 5.16 depicts the data flow that is used to decide on the next activity to be executed. An activity in HypREST changes the *Current State* based on different factors that are decided by the developer of the client application. The activity could also identify *Local Events* that might be utilized for changing to the next possible state. If an activity decides to interact with the server based on the possible hypermedia links that were

sent in the previous response message, the *Current Hypermedia Link* component should be updated. When the server sends a response, its message should include the next possible transitions via hypermedia links. As a result, the *Possible Hypermedia Link* component should be updated with the returned links. In addition, remote events could affect the next possible transition by identifying them in the *Local/Remote Events* component.

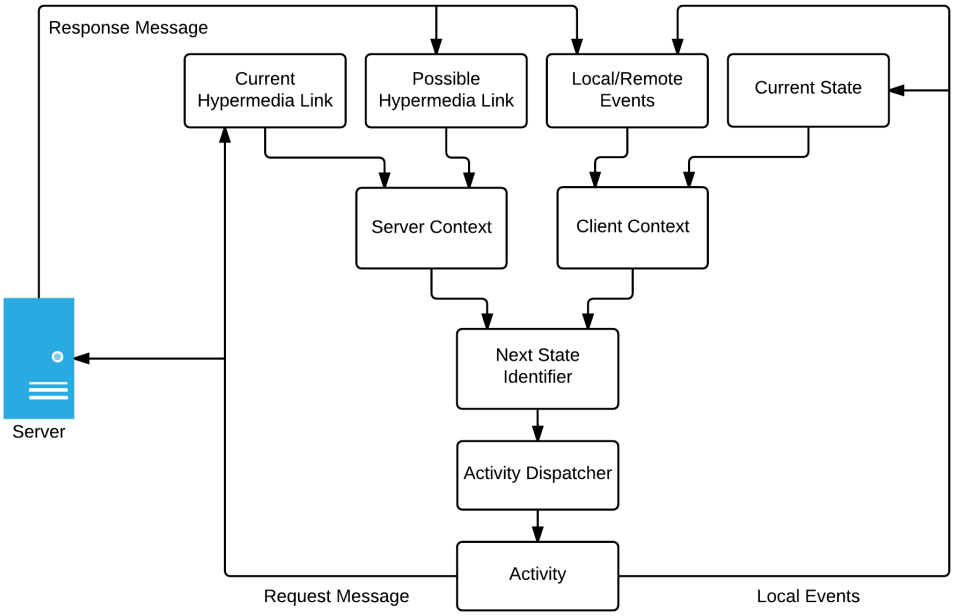

Figure 5.16: Client Control Flow.

Chapter 6

Implementation

This chapter describes an example of how *RESTDL* and *HypREST* could be implemented and used to consume REST services. The implementation was developed using Java as a programming language. Examples of how the components of REST client applications could be built are also introduced.

6.1 RESTDL

RESTDL is a description language that is mainly focused on representing resources in a structured, easy and visible way. The following sections discuss an implementation prototype of a library that supports the description of REST resources based on RESTDL's architecture that is discussed in chapter 4 (for simplicity, the library will be called *RESTDL-Lib* to be distinguished from RESTDL documents). RESTDL-Lib allows defining meta-information to REST service's resources. Figure 6.1 highlights the components of RESTDL-Lib's implementation, how they can be used in REST applications, and how the data flows between the client, server, and their components. Basically, RESTDL-Lib's meta-information could be used to identify the resources interactions' data and metadata (Step 1 in the figure). Based on these meta-information, a RESTDL document that represent the interactions on the server side are generated (Step 2). This document could be then used by the clients of the REST service to identify the service's interactions (Step 3). In addition, RESTDL-Lib is capable of generating code based on the RESTDL document that is fetched (Step 4 & 5). The generated code could be used by the client's business logic to interact with REST service's resources (Step 6 & 7).

To be able to expose the description of the resource's interactions, a set of components on the server side were developed. Figure 6.2 shows the main components of RESTDL-Lib. The *Resource's Annotations meta data* component represents a set of Annotation-based meta-information that could be used to identify the data and metadata of server's resources (details are followed in section 6.1.2). *Annotations parser* component is responsible for scanning server's resources that are annotated with RESTDL-Lib's annotations. This component extracts the necessary information for generating the RESTDL document. These information include the message metadata, protocol metadata, and resource's properties. *XML RESTDL Generator* takes the extracted information from the *Annotations parser* component and generates a RESTDL document in XML.

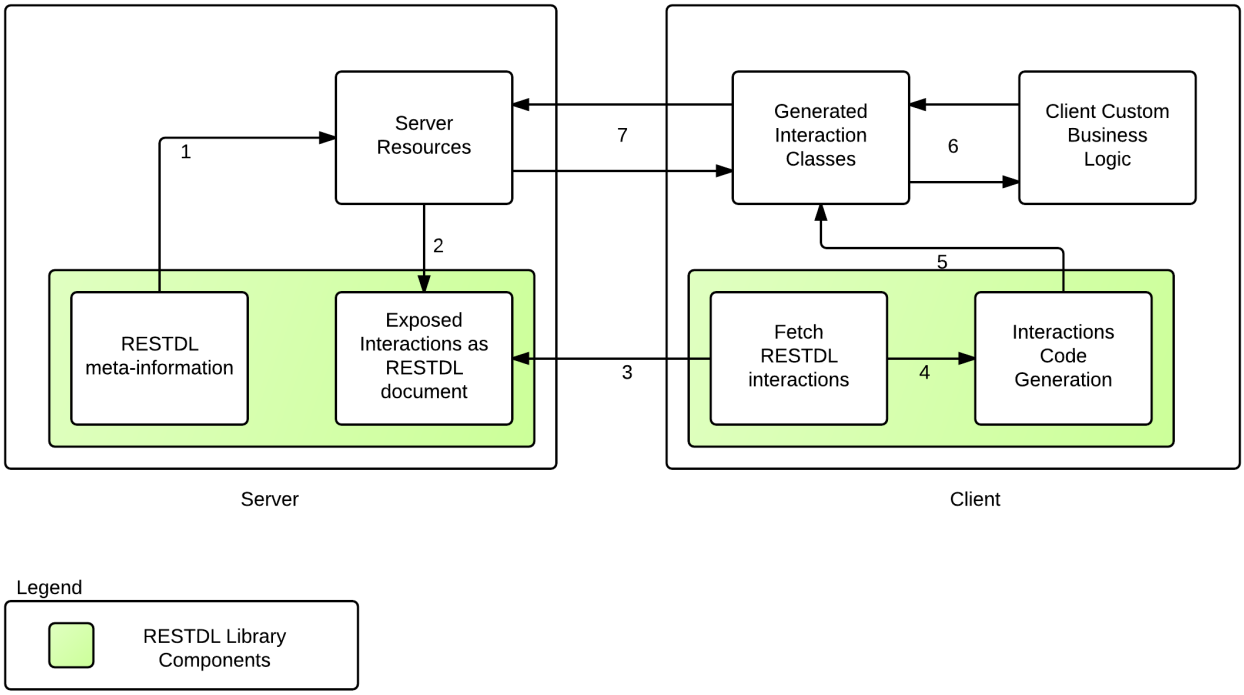

Figure 6.1: Main components and data flow of RESTDL-Lib in clients and servers.

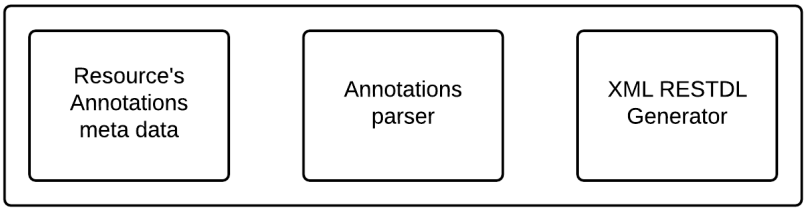

Figure 6.2: Server's main components of the RESTDL-Lib.

RESTDL-Lib was implemented to leverage the server's exposed RESTDL document on the client side as well. Figure 6.3 shows the main components of RESTDL-Lib that can be used on the client side. *RESTDL Document Fetcher* component is

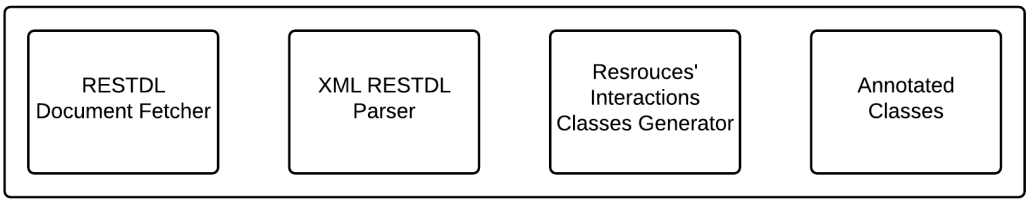

Figure 6.3: Client's main components of RESTDL-Lib.

used to fetch the exposed document from the server. RESTDL-Lib assumes that the interactions are exposed on the server side as it was discussed in section 4.2. After fetching the RESTDL document, the *XML RESTDL Parser* component parses the document and extracts the information of resources' interactions. The *Resources' interactions Classes Generator* component then takes the extracted information and generates executable interactions classes. These classes are annotated with the *Canonical Data Model* meta-information as it will be explained in section 6.1.3.

The following sections will discuss in more detail how the mentioned components can be used by the developers of REST applications.

6.1.1 Resources

The communication between a client and a server of a REST service should rely upon an interaction of an exposed resource on the server side. To be able to communicate the data, metadata and resource's specific requirements to establish successful interactions, RESTDL has been introduced to offer an easy and consistent approach of delivering these information to the service consumers.

RESTDL defines a set of building blocks to represent the required information that establish successful interactions. In this work's prototype of RESTDL-Lib which is based on Java, *Annotations* have been used as a method of defining the resources' meta-information. Annotations could offer many advantages when used instead of traditional text-based metadata representations; among others, annotations offer an easy way to define information that are checked on compilation time. As a result, it eliminates many of the errors that could exist when running the applications. In addition, annotations are visible to the developer when implementing the business logic of the application, which makes it easier for interpretation and change on development time.

To help represent the resources on the server side, which includes their data, metadata and interactions, a set of server side annotations have been defined. In addition, a set of annotations have been defined to represent exposed resources on the client side. The following two sections discuss each type of annotations on both

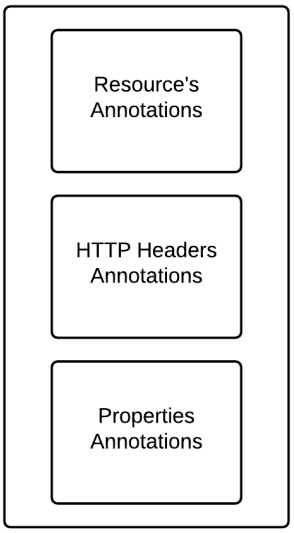

Figure 6.4: Annotations of REST resources in RESTDL's implemented library.

sides of the communications.

6.1.2 Server Annotations

Server Annotations are mainly used to represent the resources and their interactions. As a result, a RESTDL document could be generated based on the annotated resources. The resulted RESTDL document could be used by the clients to understand and establish a consistent way of communicating with the servers. Figure 6.4 shows the main building blocks of the RESTDL-Lib's implementation of server annotations. Basically, the implemented annotations represent the main building blocks of RESTDL's architecture which are introduced in section 4.3.

RESTDL has three main constructs (see 4.3), *Server Interactions*, *Resource Interactions*, and *Interaction*. In the proposed implementation, *Server Interactions* are represented as a sum of all the *Resource Interactions*. Similarly, Resource Interactions construct is represented for each resource as the sum of all the *Interaction* definitions of a resource. To be able to represent the Interaction of a resource, the following special server side annotations *@InteractionType*, *@RelationHeader*, and *@RelationProperty* have been designed and implemented.

Listing 6.1 represents an annotation example of how a resource's interaction could be defined. In this example, the request and response metadata are defined. These metadata include the *Relation Name* of the interaction, human readable *Description*, REST's specific *Action*, and the possible *Media Type* to interact with

the resource.

```
@InteractionType(
  request = @RequestType(
    relationName="register_person_request",
    description="The request to Register a new Person.",
    action=Action.CREATE,
    mediatypes={"application/x-www-form-urlencoded"}
  ),
  response = @ResponseType(
    relationName="register_person_response",
    description="The response of Registering a new Person.",
    mediatypes={"application/hal+json", "application/vnd.siren+json"}
  )
)
```

Listing 6.1: RESTDL's Interaction Definition

Listing 6.2 represents the definition of the metadata that could be identified for Transport Systems. The *rels* parameter represents the name of the relations that this specific metadata should be associated with.

```
@RelationHeader(name="API-VERSION", optional=false,
  description="Specifies the API Version",
  rels = { "register_person_request", "register_person_response"
  "receive_person_information_request", "receive_person_information_response"}
)
```

Listing 6.2: RESTDL's Header Definition

Listing 6.3 represents the definition of a property of a resource. This includes the *name*, associated *relations*, whether it is *optional* or not, and its *value* type.

```
@RelationProperty(description = "Specifies the id of the registered person",
  name = "id", optional = false, value = ValueType.STRING,
  rels = { "receive_person_information_request", "receive_person_information_response",
  "register_person_response"  }
)
```

Listing 6.3: RESTDL's Property Definition

Listing 6.4 shows an example of how RESTDL-Lib can be used to define the interactions of a resource. In this example, a resource that should be exposed by the REST service is called *Persons*. A client application of this service can interact with the *Persons* resource via two interactions. First, to register a new person by following a link which has *register_person_request* as its relation. Second, to retrieve the information of a specific person via issuing a request of a link that has *receive_person_information_request* as its relation. These interactions are defined as annotations to the *Persons* resource. Lines 1 and 14 of Listing 6.4 show the interaction definitions of registering a person, and receiving a person's information respectively.

Each interaction consists of a *Request* and *Response* messages. The *Media Type* of the request and response messages should be identified in the definition of the interactions. In addition, the request message has to identify the *Action* that should

be included in the message when interacting with the resource. Moreover, a description of each message should be included for human readability. Finally, the relation name of each message should be uniquely identified to make it possible for the client application to identify each interaction when sending and receiving messages.

Listing A.1 in Appendix A shows how the library could represent the Persons resource as an XML RESTDL document. This representation is mainly consumed by the clients of the service to understand the interactions or generate client side code to interact with the service.

```
@InteractionType(
  request = @RequestType(
    relationName="register_person_request",
    description="The request to Register a new Person.",
    action=Action.CREATE,
    mediatypes={"application/x-www-form-urlencoded"}
  ),
  response = @ResponseType(
    relationName="register_person_response",
    description="The response of Registering a new Person.",
    mediatypes={"application/hal+json", "application/vnd.siren+json"}
  )
)
@InteractionType(
  request = @RequestType(
    relationName="receive_person_information_request",
    description="The request to Receive a Person's information.",
    action=Action.READ,
    mediatypes={"application/x-www-form-urlencoded"}
  ),
  response = @ResponseType(
    relationName="receive_person_information_response",
    description="The response of Receiving a Person's information",
    mediatypes={"application/hal+json", "application/vnd.siren+json"}
  )
)
public class Persons {

  @RelationHeader(name="API-VERSION", optional=false,
  description="Specifies the API Version",
  rels = { "register_person_request", "register_person_response",
  "receive_person_information_request", "receive_person_information_response"})
  private String api_version;

  @RelationProperty(description = "Specifies the id of the registered person",
  name = "id", optional = false, value = ValueType.STRING,
  rels = { "receive_person_information_request", "receive_person_information_response",
  "register_person_response"  })
  private String id;

  @RelationProperty(description = "The name of the person",
  name = "name", optional = false, value = ValueType.STRING,
  rels = { "register_person_request", "receive_person_information_response" })
  private String name;

  @RelationProperty(description = "The email of the person",
  name = "email", optional = false, value = ValueType.STRING,
  rels = { "register_person_request", "receive_person_information_response" })
  private String email;

  @RelationProperty(description = "The age of the person",
  name = "age", optional = true, value = ValueType.INTEGER,
  rels = { "register_person_request", "receive_person_information_response" })
  private int age;
}
```

Listing 6.4: A server Resource annotated with RESTDL

6.1.3 Client Annotations

The client defines Resource's Annotations for the goal of Content Negotiation as well as interpreting the data streams when interacting with the server. Defining the data and metadata in a technology-specific representation results in the need of transforming this representation into the one that the server understands. For instance, in the case of Java, resources along with their data and metadata are represented as primitive or object types. To be able to represent these information in a specific mediatype, annotations are used as a way to identify each construct.

```
1  @ResourceModel(relations={"relation_1","relation_2"})
2  public class ResourceExample {
3
4    public ResourceExample(){}
5
6
7    public ResourceExample(String relation, String property1,
8    String property2, String property3, URL href,
9    ParserResourceTesterTwo instance) {
10     this.relation = relation;
11     this.property1 = property1;
12     this.property2 = property2;
13     this.property3 = property3;
14     this.href = href;
15   }
16
17   @ResourceRelation
18   private String relation;
19
20   @ResourceProperty(rel={"relation_1"})
21   private String property1;
22
23   @ResourceProperty(rel={"relation_1","relation_2"})
24   private String property2;
25
26   @ResourceProperty
27   private String property3;
28
29   @ResourceHref
30   private URL href;
31
32   @SubEntity
33   private ResourceExample_2 instance;
34
35   @SubEntityList
36   private List<ResourceExample_3> subResources = new LinkedList<>();
37
38   @ResourceLinkList
39   private List<Link> links = new LinkedList<>();
40 }
```

Listing 6.5: Client's Annotations Example

Listing 6.5 represents an example of a class that is defined on a client application. In this example, the resource is identified by the *@ResourceModel* annotation. This annotation identifies two server *relations* , relation_1 and relation_2. *@ResourceRelation* annotation specifies the object instance's current relation type. *@ResourceProperty* annotation identifies properties of the resource, the *rel* at-

tribute identifies which relation is associated with this property. *@ResourceHref* represents an annotation that identifies the location of the resource on the server side. Using HTTP as a Transport System, this could be achieved using the URI. Resource could be structured in a hierarchal manner, to identify a sub-entity or a collection of sub-entities, *@SubEntity* and *SubEntityList* annotations are used, respectively. The hypermedia links that the server sends back in its response are held in the resource's links using the annotation of *@ResourceLinkList*.

6.1.4 Code Generation

One of the most frequent assigned tasks to a REST client application developer is to build a representation of the REST service's resources in the programming language that is mainly used. This task could be considered repetitive, boring, and error-prone due to the approach that is mainly used in representing service's resources. In today's solutions, service's resources are mainly represented in a human-readable documentation. A developer should find the documentation, mainly on the service provider's website, read the documentation and understand it, build a mental model that covers the resource's data, metadata, interactions, and their graph model, then this mental model has to be translated onto a representation for the technology that is mainly used on the client side. In addition, the programmer has to be able to cope with the transport system details and its connections.

To avoid this repetitive approach, eliminate sending invalid requests, and add consistency to the way client applications handle the interactions with the service provider, a code generator that takes the description of the resources as an input, in this case RESTDL documents, and produces the necessary code to handle the data, metadata, interactions and the connections with the server in a seamless and usable manner is needed. In this work, a code generator that fulfills the mentioned goals has been developed.

Listing A.2 in Appendix A shows an example of a Java generated code that represents the *Persons* resource of listing 6.4 and its interactions. Using RESTDL-Lib, the interactions are represented as individual inner classes which are encapsulated by the Resource itself, in our example the Persons resource. This way, the code is kept clean, independent, and ensures having all the interactions of a specific resource in one place.

6.2 HypREST

As one of the main requirement of this work, HypREST's implementation prototype has been developed to help build REST compliant client applications. HypREST framework was implemented as a library that can be included in client projects. The developer leverages the components of HypREST to implement the business logic of the client application. In fact, HypREST does all the data transformation and communication with the server automatically when a request or response message is present.

Building a client application based on the HATOEAS constraint could be con-

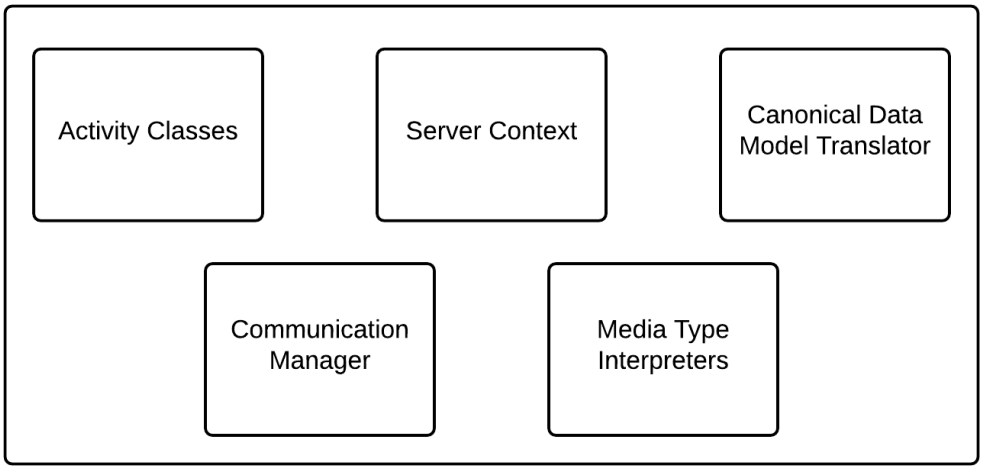

Figure 6.5: HypREST prototype's main components.

sidered a challenging task. Therefore, HypREST was developed to help build REST compliant clients in a simple way. Figure 6.5 shows the components of the implemented HypREST solution. HypREST divides the responsibilities of the developer into two main tasks, *Media Type Interpreters* definition, and *Activity Classes* definition which includes their activation context. In addition, HypREST defines other components that work automatically to handle the interactions with the server. These components represent the *Communication Manager* to handle the communication with the server, *Server Context* to hold information about server's possible interactions, and *Canonical Data Model Translator* to help transform the Activity Classes to different Media Type representations and vice versa.

HypREST has been built having the varieties of media types in mind. As a result, different media types in the market have been integrated into the solution. For instance, *Application/hal+json*, and *Application/x-www-form-urlencoded* are already supported in the framework. In addition, the framework allows integrating other media types.

The proposed implementation of HypREST in Java is mainly based on *Java API for RESTful Services (JAX-RS)* specifications [8]. As a result, media types could be defined to read and write response and request messages, respectively. The definition of new media types and registering them in the framework to be used on runtime could be considered an easy process. Based on the client annotations that are described in section 6.1.3, request and response messages could be transformed to and from a message stream of information using the *Canonical Data Model Translator* component. For instance, a media type (e.g. *Applica-*

tion/hal+json) could be identified in the framework to be used to read and write response and request messages. Listing 6.6 shows how to register this media type within the framework, specifically at line 5.

```
1
2  Hypo hypo = Hypo.createInstance(new URI(
3    "http://example.com/api/"));
4
5  hypo.registerMediaType(MediaTypeHAL.class);
6
7  hypo.register(
8    new CreatePerson(hypo, "home", "root", "register_person_request"));
```

Listing 6.6: HypREST's Activity registration with its triggers

An activity within the proposed implementation of HypREST framework is defined as an individual class. This class is executed by the framework itself when it examines its state identifiers and finds that it needs to be executed. To show how an activity could be defined, an example in listing 6.7 has been built. This activity takes the name and email of a person as a terminal input from the user, sends a registration request to the server, receives the response, and then prints the response's person ID to the terminal.

Defining the triggers that help the framework decide on the execution of the activity is simply done when registering the activity with the framework. Line 7 of listing 6.6 shows how an activity could be registered with the framework. The creation of an *CreatePerson* identifies the triggers of this activity. In this example, the client state has to be *home*, the server state has to be *root* and the possible relations on the server should have *register_person_request*. In these conditions have met, then the framework will execute the *CreatePerson* activity.

It can be seen from Listing 6.6 (line 2) that the client application is only supplied with the root URI of the REST service. This is because HypREST was designed for REST compliant applications. As a result, HypREST applies the HATEOAS constraint and relies on the server's response messages which should hold the information about the resources' locations. Using this model, the HypREST implementation forces REST services' providers and consumers to conform to the set of all REST's constraints as it does not define a way to define resource's statically typed URIs.

```java
package org.hyprest.demo;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

import org.hyprest.hypo.Hypo;
import org.hyprest.hypo.dispatcher.Activity;
import org.hyprest.restdl.generated_classes.Person.Register_person_request;
import org.hyprest.restdl.generated_classes.Person.Register_person_response;

public class CreatePerson extends Activity {

  // the client state identifier when the activity execution is done
  private String currentState = "created_person";

  public CreatePerson(Hypo context, String clientState,
      String serverCurrentState, String serverNextState) {
    super(context, clientState, serverCurrentState, serverNextState);
  }

  @Override
  public void doAction() {
    //Input buffer to get User's input data from the console
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

    try {
      //read Person's name
      System.out.println("Enter Person's Name: ");
      String name = br.readLine();

      //read Person's email
      System.out.println("Enter Person's Email: ");
      String email = br.readLine();

      //Instantiate registering a new person request
      Register_person_request request = new Register_person_request(name, email);

      //Send the request to the server, and get the response
      Register_person_response response = context.followLink(request,
          Register_person_response.class, request.getRelation());

      //Print the server's response of Person's ID
      System.out.println(name+"'s ID is: "+ response.getId());

    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  @Override
  public String getResultClientState() {
    return this.currentState;
  }

}
```

Listing 6.7: Person's Activity Definition

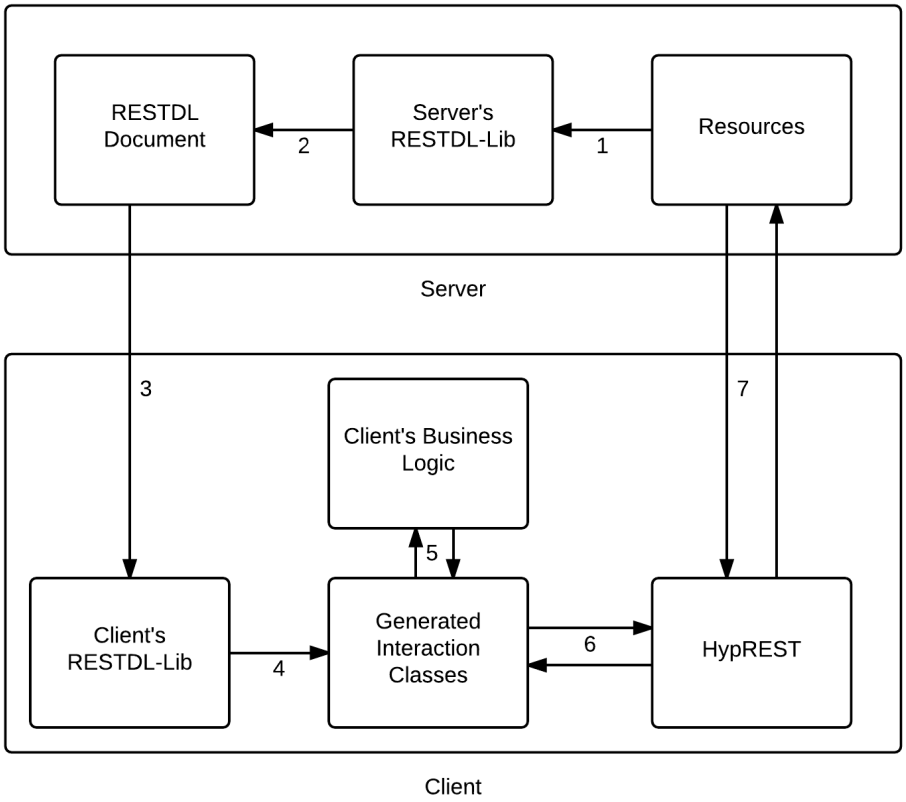

Figure 6.6: RESTDL-Lib, RESTDL, and HypREST integration.

6.3 RESTDL-Lib, RESTDL, and HypREST integration

Figure 6.6 depicts the flow of information to help the client start communicating with the server based on the proposed solutions in this work. First, the REST service developer defines the set of server's resources. These resources are annotated RESTDL-Lib's server's annotations. RESTDL-Lib then extracts the information from the resources and exposes the interactions as a RESTDL document. On development time, the client fetches the RESTDL document and generates the interactions' classes. The developer of the client application uses the generated classes to build up the business logic of the application. When there is a need for an interaction, the class instance's data is transformed into a media type representation that the server understands using HypREST's media type and canonical data model translator components. Finally, the media type is sent to the server which targets a unique resource. When sending a response from the server, the server sends it as a media type representation which is then transformed into a class instance. This instance is then used by the application logic to continue its execution.

6.4 Evaluation of the solutions

While developing the introduced solutions of this work. Different advantages and disadvantages have been realized. In this section, the pros and cons of using the presented solutions are introduced.

Pros

Better Understandability and Learnability: using RESTDL for describing REST services helps learn and understand the service's resources easily. This is due to the consistent way resources are described. In fact, RESTDL structures the information needed to issue request messages into different data and metadata constructs. In addition, RESTDL defines human-readable properties for better understandability.

Separation of Concern: using a code-generation engine that translates RESTDL documents into executable code helps separate the concern of building custom libraries for each REST service. This task could be delegated to the engine to generate the required code to start communicating with the server and help focus more on the business requirements rather than technology requirements.

Conformance to the specifications: based on HypREST, compliant client applications can be built. This enables client applications benefit from the advantages that the architectural constraints of REST introduce. For instance, compliant client applications become more tolerant to changes on the server side.

Cons

New Client Applications Model: HypREST introduces the Activity-based model as an approach to structure the business logic of the client applications. One drawback of this approach could be related to the learnability that is needed to start developing client applications. In fact, this model is not the usual way of developing REST client applications as it delegates examining the events to the framework's components.

The Need to Cover Different Scenarios of Trigger Events: Client applications' developers should be aware that using HATEOAS and the Activity-based model, trigger events could vary based on different factors. These factors could be related to the business requirements of the application. For instance, the response messages' relations could be different when issuing the same request because different hypermedia links could be faced. As a result, different activities might be needed to cope with the changes of transitioning events that might be introduced by the server applications. In addition, it might be useful to have a default activity that the client application can transition to whenever it is not possible to transition to any other activity.

Chapter 7

Conclusion

This chapter wraps everything that has been accomplished in this work. In addition, it highlights some improvement possibilities that could be achieved in future work.

7.1 Summary

REST's architecture offers many advantages for building distributed applications. Most of today's REST services apply only a sub-set of the REST's architectural constraints. Therefore, many advantages that can be beneficial for the applications when applying the full set of constraints are missing. For instance, client developers have to understand all the architectural details of the service before start developing their business solutions. Alternatively, developers rely on tightly-coupled technology-based libraries that are offered by service providers to start communicating with the service. Using these solutions introduces inconsistencies in the way REST applications are developed. This is due to the differences in the way client libraries are developed which mostly differs when interacting with other REST services. In addition, these client libraries do not conform to the full architectural constraints of REST services especially in applying the HATEOAS constraint. Therefore, most of today's client applications are intolerant to REST service's changes. To cope with these problems, a set of solutions have been introduced in this work.

To help solve the introduced requirement in section 1.1 of eliminating the need of human-based documentation for understanding how to interact with REST service's resources, RESTDL has been introduced. RESTDL helps describe REST service's resources in a machine-understandable manner. Therefore, client applications understand how to interact with the REST service's resources based on the discoverability concept. This way, client applications' developers do not need to go through detailed documentations to start communicating with the service. Instead, the developer is only needed to understand what the service offers and which exact resource does it. Moreover, RESTDL's architecture restricts how to describe the service's resources in a way that helps the service conform to REST constraints. For example, RESTDL does not specify an identifier for the location of REST service's resources, which results in forcing REST services developers apply the constraint of HATEOAS when using RESTDL.

RESTDL documents can be used to generate client-side, REST-compliant executable code. This could be achieved using a code-generation engine that discovers RESTDL documents and convert them to executable code.

The requirement of designing a generic solution for building REST's compliant client applications has been met via introducing the design of the HypREST framework. HypREST is a generic framework that helps build client applications which conform to the full set of REST's constraints. Using this framework helps structure client applications into components that are executed when their respective trigger events are met. Such events could be local or remote events. This is based on the introduced Activity-based model. Using this model, client applications can cope with the changes that could be introduced on the server side. For instance, when changing the location of the resources. In fact, HypREST is only supplied with location of the REST service's root resource. Therefore, the HATEOAS constraint can be leveraged on the client side. In addition, HypREST introduced the concept of a Canonical Data Model, allowing the transformation of interactions' information between different representations on the network while offering a consistent way of interacting with the REST service's resources to the developer of the client application.

The solutions introduced in this work solve many problems for service providers and consumers. While designing these solutions, the types of client applications introduced in chapter 3 were taken into considerations. This helped build generic solutions that support all kinds of client application's business goals. In fact, RESTDL's interaction description and HypREST's Activity-based model allow building structured business logic that could represent any type of client applications.

Based on RESTDL's architecture, RESTDL-Lib (see section 6.1) was implemented as a Java library that helps describe REST service's resources and generate RESTDL documents. In addition, RESTDL-Lib implements a code-generation engine which generates client-side Java executables that could be used to communicate with the REST service's resources. This library leverages Java annotations to identify the meta-information of service's resources. In addition, an implementation of the HypREST architecture was developed to help build compliant client applications (see section 6.2). Using the HypREST implementation, client application's business logic can be structured to individual components. These modules are executed via the framework components when their trigger events are met. The integration of RESTDL-Lib, RESTDL, and HypREST has been also introduced (see section 6.3) to show how the proposed solutions can be used together so that REST service's providers and consumers benefit from the full-compliance solutions.

7.2 Future Work

Many solutions have been introduced in this work that help build compliant REST applications. However, further improvements could be worked on in the future. This section introduces some of the ideas that could be implemented.

In this work, RESTDL has been introduced as an architecture for describing

REST services. In addition, a code-generation engine library has been implemented for Java applications. In fact, further work is needed to implement an engine that supports different technologies. This way, client applications that run using different technologies and platforms could benefit from the architecture of RESTDL.

RESTDL could also be improved by adding semantics to its properties. This will help client applications understand the meanings of each property within the communicated request and response messages. In addition, it will help build much more intelligent and generic client applications that react based on the data that is being communicated between clients and servers. This could be achieved using globally available data semantics repositories such as *http://schema.org*.

HypREST can be also improved by providing a mechanism to dynamically bind client's business logic to the properties' changes that could be introduced by the server. This, however, is a challenging task as the client application needs to cope with the missing information when a resource evolves its communicated information.

Finally, the preliminary and simple formal model introduced in section 5.9 could be further improved to help understand and implement client technologies that conform to the architecture of REST.

Bibliography

[1] Florian Haupt et al. “A model driven approach for REST compliant services”. English. In: *Proceedings of the IEEE International Conference on Web Services, ICWS 2014*. IEEE, 2014. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2014-23&engl=1.

[2] Erik Christensen et al. *Web services description language (WSDL) 1.1*. 2001.

[3] Marc J Hadley. “Web application description language (WADL)”. In: (2006).

[4] Reverb. *SWAGGER, The World’s Most Popular Framework for APIs*. URL: http://swagger.io/ (visited on 12/04/2014).

[5] RAML Workgroup. *RAML, RESTful API Modeling Language*. URL: http://raml.org/ (visited on 12/04/2014).

[6] Oren Ben-Kiki, Clark Evans, and Ingy. *YAML Ain’t Markup Language (YAML™) Version 1.2*. URL: http://www.yaml.org/spec/1.2/spec.html (visited on 12/04/2014).

[7] Hyunghun Cho and Sukyoung Ryu. “REST to JavaScript for better client-side development”. In: *Proceedings of the companion publication of the 23rd international conference on World wide web companion*. International World Wide Web Conferences Steering Committee. 2014, pp. 937–942.

[8] Oracle Corporation. *Java API for RESTful Services (JAX-RS)*. URL: https://jax-rs-spec.java.net/ (visited on 11/30/2014).

[9] Bill Burke. *RESTful Java with JAX-RS 2.0*. " O’Reilly Media, Inc.", 2013.

[10] Roy T Fielding. “REST APIs must be hypertext-driven”. In: *Untangled musings of Roy T. Fielding* (2008).

[11] Mike Amundsen. *Building Hypermedia APIs with HTML5 and Node*. " O’Reilly Media, Inc.", 2011.

[12] Markus Lanthaler and Christian Gütl. “Hydra: A Vocabulary for Hypermedia-Driven Web APIs.” In: *LDOW*. Citeseer. 2013.

[13] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.

[14] The Open Group. *Service Oriented Architecture: What is SOA?* URL: http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm (visited on 10/23/2014).

[15] John B Oladosu, Justice O Emuoyinbofarhe, and Christopher O Oyeleye. “Modelling a Service Oriented Architecture for an" e-Doctor" Mobile Health Services”. In: (2010).

[16] Wide Web Consortium. *Wide Web Consortium*. URL: http://www.w3.org/ (visited on 10/23/2014).

[17] Hugo Haas and Allen Brown. "Web services glossary". In: *W3C Working Group Note (11 February 2004)* (2004).

[18] OASIS. *Web Services Notification (WSN)*. URL: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn (visited on 10/23/2014).

[19] Martin Gudgin et al. "SOAP Version 1.2". In: *W3C recommendation* 24 (2003).

[20] Roy Fielding and Julian Reschke. "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing". In: (2014).

[21] Jon Postel. "Simple mail transfer protocol". In: *Information Sciences* (1982).

[22] Roy Thomas Fielding. "Architectural styles and the design of network-based software architectures". PhD thesis. University of California, Irvine, 2000.

[23] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. Prentice Hall PTR, 2002.

[24] Ruben Verborgh. "Serendipitous web applications through semantic hypermedia". In: *Sort* 100 (2014), p. 250.

[25] Pavol Návrat and Roman Filkorn. "A Note on the Role of Abstraction and Generality in Software Development". In: *Journal of Computer Science* 1.1 (2005), p. 98.

[26] The Open Group. *The Internet Engineering Task Force (IETF)*. URL: https://www.ietf.org/ (visited on 10/23/2014).

[27] Tim Berners-Lee, Roy Fielding, and Larry Masinter. "RFC 3986: Uniform resource identifier (uri): Generic syntax". In: *The Internet Society* (2005).

[28] Martin Fowler. "Richardson Maturity Model: steps toward the glory of REST". In: *Online at http://martinfowler.com/articles/richardsonMaturityModel.html* (2010).

[29] Ned Freed and Nathaniel Borenstein. *Multipurpose internet mail extensions (MIME) part two: Media types*. Tech. rep. rfc 2046, November, 1996.

[30] Mark Nottingham. "Web linking". In: (2010).

[31] Erl Thomas. "Service-oriented architecture: concepts, technology, and design". In: *Prentice Hall* 31 (2005), W3C.

[32] Microsoft. *Regular Expression Language - Quick Reference*. URL: http://msdn.microsoft.com/en-us/library/az24scfc(v=vs.110).aspx (visited on 10/24/2014).

Appendix A

Code Examples

A.1 RESTDL Document

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<serverInteractions>
  <interactions resource="Persons">
    <interaction>
      <request>
        <relationName>receive_person_information_request</relationName>
        <description>The request to Receive a Person's information.</description>
        <mediaTypes>application/x-www-form-urlencoded</mediaTypes>
        <action>READ</action>
        <properties>
          <name>id</name>
          <description>Specifies the id of the registered person</description>
          <embed>false</embed>
          <optional>false</optional>
          <value>STRING</value>
          <pattern></pattern>
        </properties>
        <headers>
          <name>API-VERSION</name>
          <description>Specifies the API Version</description>
          <optional>false</optional>
        </headers>
      </request>
      <response>
        <relationName>receive_person_information_response</relationName>
        <description>The response of Receiving a Person's information</description>
        <mediaTypes>application/hal+json</mediaTypes>
        <mediaTypes>application/vnd.siren+json</mediaTypes>
        <properties>
          <name>age</name>
          <description>The age of the person</description>
          <embed>false</embed>
          <optional>true</optional>
          <value>INTEGER</value>
        </properties>
        <properties>
          <name>email</name>
          <description>The email of the person</description>
          <embed>false</embed>
          <optional>false</optional>
          <value>STRING</value>
```

```
42            </properties>
43            <properties>
44               <name>name</name>
45               <description>The name of the person</description>
46               <embed>false</embed>
47               <optional>false</optional>
48               <value>STRING</value>
49            </properties>
50            <properties>
51               <name>id</name>
52               <description>Specifies the id of the registered person</description>
53               <embed>false</embed>
54               <optional>false</optional>
55               <value>STRING</value>
56            </properties>
57            <headers>
58               <name>API-VERSION</name>
59               <description>Specifies the API Version</description>
60               <optional>false</optional>
61            </headers>
62         </response>
63      </interaction>
64      <interaction>
65         <request>
66            <relationName>register_person_request</relationName>
67            <description>The request to Register a new Person.</description>
68            <mediaTypes>application/x-www-form-urlencoded</mediaTypes>
69            <action>CREATE</action>
70            <properties>
71               <name>age</name>
72               <description>The age of the person</description>
73               <embed>false</embed>
74               <optional>true</optional>
75               <value>INTEGER</value>
76            </properties>
77            <properties>
78               <name>email</name>
79               <description>The email of the person</description>
80               <embed>false</embed>
81               <optional>false</optional>
82               <value>STRING</value>
83            </properties>
84            <properties>
85               <name>name</name>
86               <description>The name of the person</description>
87               <embed>false</embed>
88               <optional>false</optional>
89               <value>STRING</value>
90            </properties>
91            <headers>
92               <name>API-VERSION</name>
93               <description>Specifies the API Version</description>
94               <optional>false</optional>
95            </headers>
96         </request>
97         <response>
98            <relationName>register_person_response</relationName>
99            <description>The response of Registering a new Person.</description>
100           <mediaTypes>application/hal+json</mediaTypes>
101           <mediaTypes>application/vnd.siren+json</mediaTypes>
102           <properties>
```

```
103                 <name>id</name>
104                 <description>Specifies the id of the registered person</description>
105                 <embed>false</embed>
106                 <optional>false</optional>
107                 <value>STRING</value>
108             </properties>
109             <headers>
110                 <name>API-VERSION</name>
111                 <description>Specifies the API Version</description>
112                 <optional>false</optional>
113             </headers>
114         </response>
115     </interaction>
116 </interactions>
117 </serverInteractions>
```

Listing A.1: A server Resource represented in XML

A.2 Code Generation

```
1  package org.hyprest.restdl.generated_classes;
2
3  import java.net.MalformedURLException;
4  import java.net.URL;
5  import java.util.ArrayList;
6  import java.util.List;
7  import org.hyprest.hypo.communicationManager.Link;
8  import org.hyprest.hypo.mediatype.annotations.ResourceHref;
9  import org.hyprest.hypo.mediatype.annotations.ResourceLinkList;
10 import org.hyprest.hypo.mediatype.annotations.ResourceModel;
11 import org.hyprest.hypo.mediatype.annotations.ResourceProperty;
12 import org.hyprest.hypo.mediatype.annotations.ResourceRelation;
13 import org.hyprest.restdl.classGenerator.IRequest;
14 import org.hyprest.restdl.classGenerator.IResponse;
15
16 public class Persons {
17
18   /**
19   * The request to Receive a Person's information.
20   *
21   */
22   @ResourceModel(relations = {
23     "receive_person_information_request"
24   })
25   public static class Receive_person_information_request
26   implements IRequest
27   {
28
29     @ResourceHref
30     private URL href;
31     @ResourceRelation
32     public static String relation = "receive_person_information_request";
33     @ResourceProperty
34     private String id;
35
36     public Receive_person_information_request() {
37     }
38
39     public Receive_person_information_request(String id) {
```

```
40      this.id = id;
41      }
42
43      public Persons.Receive_person_information_request setHref(String url)
44      throws MalformedURLException
45      {
46        href = new URL(url);
47        return this;
48      }
49
50      public URL getHref() {
51        return href;
52      }
53
54      public String getRelation() {
55        return relation;
56      }
57
58      public String getAction() {
59        return "READ";
60      }
61
62      /**
63      * Specifies the id of the registered person
64      *
65      */
66      public String getId() {
67        return this.id;
68      }
69
70      public Persons.Receive_person_information_request setId(String id) {
71        this.id = id;
72        return this;
73      }
74
75      public static enum HEADERS {
76
77        API_VERSION("API-VERSION");
78        private String value;
79
80        private HEADERS(String val) {
81          this.value = val;
82        }
83
84        public String getValue() {
85          return value;
86        }
87
88      }
89
90      public static enum MEDIATYPES {
91
92        APPLICATION_X_WWW_FORM_URLENCODED("application/x-www-form-
        urlencoded");
93        private String value;
94
95        private MEDIATYPES(String val) {
96          this.value = val;
97        }
98
99        public String getValue() {
```

```
100             return value;
101         }
102
103     }
104
105 }
106
107
108 /**
109 * The response of Receiving a Person's information
110 *
111 */
112 @ResourceModel(relations = {
113   "receive_person_information_response"
114 })
115 public static class Receive_person_information_response
116 implements IResponse
117 {
118
119     @ResourceHref
120     private URL href;
121     @ResourceRelation
122     public static String relation = "receive_person_information_response";
123     @ResourceLinkList
124     private List<Link> links = new ArrayList<Link>();
125     @ResourceProperty
126     private String id;
127     @ResourceProperty
128     private String name;
129     @ResourceProperty
130     private String email;
131     @ResourceProperty
132     private Integer age;
133
134     public Receive_person_information_response() {
135     }
136
137     public Persons.Receive_person_information_response setHref(String url)
138     throws MalformedURLException
139     {
140         href = new URL(url);
141         return this;
142     }
143
144     public URL getHref() {
145         return href;
146     }
147
148     public String getRelation() {
149         return relation;
150     }
151
152     public List<Link> getLinks() {
153         return this.links;
154     }
155
156     /**
157     * Specifies the id of the registered person
158     *
159     */
160     public String getId() {
```

```
161         return this.id;
162       }
163
164       public Persons.Receive_person_information_response setId(String id) {
165         this.id = id;
166         return this;
167       }
168
169       /**
170       * The name of the person
171       *
172       */
173       public String getName() {
174         return this.name;
175       }
176
177       public Persons.Receive_person_information_response setName(String name) {
178         this.name = name;
179         return this;
180       }
181
182       /**
183       * The email of the person
184       *
185       */
186       public String getEmail() {
187         return this.email;
188       }
189
190       public Persons.Receive_person_information_response setEmail(String email) {
191         this.email = email;
192         return this;
193       }
194
195       /**
196       * The age of the person
197       *
198       */
199       public Integer getAge() {
200         return this.age;
201       }
202
203       public Persons.Receive_person_information_response setAge(Integer age) {
204         this.age = age;
205         return this;
206       }
207
208       public static enum HEADERS {
209
210         API_VERSION("API-VERSION");
211         private String value;
212
213         private HEADERS(String val) {
214           this.value = val;
215         }
216
217         public String getValue() {
218           return value;
219         }
220
221       }
```

```
222
223     public static enum MEDIATYPES {
224
225         APPLICATION_HAL_JSON("application/hal+json"),
226         APPLICATION_VND_SIREN_JSON("application/vnd.siren+json");
227         private String value;
228
229         private MEDIATYPES(String val) {
230             this.value = val;
231         }
232
233         public String getValue() {
234             return value;
235         }
236
237     }
238
239 }
240
241
242 /**
243 * The request to Register a new Person.
244 *
245 */
246 @ResourceModel(relations = {
247     "register_person_request"
248 })
249 public static class Register_person_request
250 implements IRequest
251 {
252
253     @ResourceHref
254     private URL href;
255     @ResourceRelation
256     public static String relation = "register_person_request";
257     @ResourceProperty
258     private String name;
259     @ResourceProperty
260     private String email;
261     @ResourceProperty
262     private Integer age;
263
264     public Register_person_request() {
265     }
266
267     public Register_person_request(String name, String email) {
268         this.name = name;
269         this.email = email;
270     }
271
272     public Persons.Register_person_request setHref(String url)
273     throws MalformedURLException
274     {
275         href = new URL(url);
276         return this;
277     }
278
279     public URL getHref() {
280         return href;
281     }
282
```

```
283     public String getRelation() {
284       return relation;
285     }
286
287     public String getAction() {
288       return "CREATE";
289     }
290
291     /**
292     * The name of the person
293     *
294     */
295     public String getName() {
296       return this.name;
297     }
298
299     public Persons.Register_person_request setName(String name) {
300       this.name = name;
301       return this;
302     }
303
304     /**
305     * The email of the person
306     *
307     */
308     public String getEmail() {
309       return this.email;
310     }
311
312     public Persons.Register_person_request setEmail(String email) {
313       this.email = email;
314       return this;
315     }
316
317     /**
318     * The age of the person
319     *
320     */
321     public Integer getAge() {
322       return this.age;
323     }
324
325     public Persons.Register_person_request setAge(Integer age) {
326       this.age = age;
327       return this;
328     }
329
330     public static enum HEADERS {
331
332       API_VERSION("API-VERSION");
333       private String value;
334
335       private HEADERS(String val) {
336         this.value = val;
337       }
338
339       public String getValue() {
340         return value;
341       }
342
343     }
```

```
344
345     public static enum MEDIATYPES {
346
347         APPLICATION_X_WWW_FORM_URLENCODED("application/x-www-form-
        urlencoded");
348         private String value;
349
350         private MEDIATYPES(String val) {
351             this.value = val;
352         }
353
354         public String getValue() {
355             return value;
356         }
357
358     }
359
360 }
361
362
363 /**
364 * The response of Registering a new Person.
365 *
366 */
367 @ResourceModel(relations = {
368     "register_person_response"
369 })
370 public static class Register_person_response
371 implements IResponse
372 {
373
374     @ResourceHref
375     private URL href;
376     @ResourceRelation
377     public static String relation = "register_person_response";
378     @ResourceLinkList
379     private List<Link> links = new ArrayList<Link>();
380     @ResourceProperty
381     private String id;
382
383     public Register_person_response() {
384     }
385
386     public Persons.Register_person_response setHref(String url)
387     throws MalformedURLException
388     {
389         href = new URL(url);
390         return this;
391     }
392
393     public URL getHref() {
394         return href;
395     }
396
397     public String getRelation() {
398         return relation;
399     }
400
401     public List<Link> getLinks() {
402         return this.links;
403     }
```

```
404
405     /**
406     * Specifies the id of the registered person
407     *
408     */
409     public String getId() {
410       return this.id;
411     }
412
413     public Persons.Register_person_response setId(String id) {
414       this.id = id;
415       return this;
416     }
417
418     public static enum HEADERS {
419
420       API_VERSION("API-VERSION");
421       private String value;
422
423       private HEADERS(String val) {
424         this.value = val;
425       }
426
427       public String getValue() {
428         return value;
429       }
430
431     }
432
433     public static enum MEDIATYPES {
434
435       APPLICATION_HAL_JSON("application/hal+json"),
436       APPLICATION_VND_SIREN_JSON("application/vnd.siren+json");
437       private String value;
438
439       private MEDIATYPES(String val) {
440         this.value = val;
441       }
442
443       public String getValue() {
444         return value;
445       }
446
447     }
448
449   }
450 }
```

Listing A.2: Auto-generated class code of the Person resource