UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Code Generation for RESTful APIs in HEADREST

Telmo da Silva Santos

Mestrado em Engenharia Informática
Especialização em Engenharia de Software

Dissertação orientada por:
Prof. Maria Antónia Bacelar da Costa Lopes
Prof. Vasco Manuel Thudichum de Serpa Vasconcelos

2018

Resumo

Os serviços web com APIs que aderem ao estilo arquitetural REST, conhecidos por serviços *web RESTful*, são atualmente muito populares. Estes serviços seguem um estilo cliente-servidor, com interações sem estado baseadas nos verbos disponibilizados pela norma HTTP.

Como meio de especificar formalmente a interação entre os clientes e fornecedores de serviços REST, várias linguagens de definição de interfaces (IDL) têm sido propostas. No entanto, na sua maioria, limitam-se ao nível sintático das interfaces que especificam e à descrição das estruturas de dados e dos pontos de interação. A linguagem HEADREST foi desenvolvida como uma IDL que permite ultrapassar estas limitações, suportando a descrição das *APIs RESTful* também ao nível semântico. Através de tipos e asserções é possível em HEADREST não só definir a estrutura dos dados trocados mas também correlacionar o *output* com *input* e o estado do servidor.

Uma das principais vantagens de ter descrições formais de *APIs RESTful* é a capacidade de gerar código *boilerplate* tanto para clientes como fornecedores. Este trabalho endereça o problema de geração de código para as *APIs RESTful* descritas com HEADREST e investiga de que forma as técnicas de geração de código existentes para os aspectos sintáticos das *APIs RESTful* podem ser estendidas para levar em conta também as propriedades comportamentais que podem ser descritas em HEADREST. Tendo em conta que a linguagem HEADREST adota muitos conceitos da *Open API Specification (OAS)*, o trabalho desenvolvido capitaliza nas técnicas de geração de código desenvolvidas para a OAS e envolveu o desenvolvimento de protótipos de geração de código cliente e servidor a partir de especificações HEADREST.

Palavras-chave: *REST*, *RESTful*, HTTP, Geração de código, *API*, Serviços *Web*

Abstract

Web services with APIs that adhere to the REST architectural style, known as RESTful web services, have become popular. These services follow a client-server style, with stateless interactions based on standard HTTP verbs.

In an effort to formally specify the interaction between clients and providers of RESTful services, various interface definition languages (IDL) have been proposed. However, for the most part, they limit themselves to the syntactic level of the interfaces and the description of the data structures and the interaction points. The HEADREST language was developed as an IDL that addresses these limitations, supporting the description of the RESTful APIs also at the semantical level. Through the use of types and assertions we not only define the structure of the data transmitted but also relate output with input and the state of the server.

One of the main advantages of having formal descriptions of RESTful APIs is the ability to generate a lot of boilerplate code for both clients and servers. This work addresses the problem of code generation for RESTful APIs described in HEADREST and aims to investigate how the existing code generation techniques for the syntactical aspects of RESTful APIs can be extended to take into account also the behavioural properties that can be described in HEADREST. Given that HEADREST adopts many concepts from the Open API Specification (OAS), this work capitalised on the code generation tools available for OAS and encompassed the development of a prototypical implementation of a code generator for clients and servers from HEADREST specifications.

Keywords: REST, RESTful, HTTP, Code Generation, API, Web Services

Contents

1 Introduction **3**

1.1 Motivation . . . 3
1.2 Context . . . 5
1.3 Objectives . . . 5
1.4 Document structure . . . 6

2 Background and Related Work **7**

2.1 RESTful APIs . . . 7
2.2 RESTful Interface Definition Languages . . . 8
2.2.1 WADL (Web Application Description Language) . . . 9
2.2.2 Hydra . . . 10
2.2.3 RAML (RESTful API Modeling Language) . . . 12
2.2.4 Open API Specification . . . 14
2.2.5 HeadREST . . . 17
2.2.6 RDLs side by side . . . 20
2.3 RESTful API frameworks . . . 22
2.3.1 Client frameworks . . . 22
2.3.2 Server frameworks . . . 23
2.4 Program generation for RDLs . . . 24
2.4.1 wadl2java . . . 25
2.4.2 RAML for JAX-RS . . . 25
2.4.3 Swagger Codegen for OAS . . . 25

3 HEADREST Codegen Overview **31**

3.1 HEADREST Codegen in a nutshell . . . 31
3.2 Encoding of HEADREST specifications into OAS . . . 32
3.2.1 Type encoding . . . 32
3.2.2 Assertion encoding . . . 33
3.3 Generation . . . 34
3.3.1 Type generation . . . 35
3.3.2 Operation generation . . . 36

3.4 Resources and representations . . . 38
3.5 Resource logic . . . 39
3.6 HEADREST Codegen tool . . . 41

4 Encoding HEADREST into OAS 43
4.1 HEADREST analyser . . . 43
4.2 Core HEADREST . . . 44
4.2.1 Derived syntax . . . 46
4.3 HEADREST extension . . . 46
4.4 Encoder . . . 48
4.4.1 Behaviour view . . . 49
4.4.2 Module View . . . 51
4.5 OAS generation . . . 53
4.5.1 Type encoding to OAS . . . 53
4.5.2 Assertions . . . 56
4.6 Limitations . . . 58

5 Generating code from HEADREST specifications 63
5.1 The code produced by the generator . . . 63
5.1.1 Models . . . 63
5.1.2 Assertions . . . 65
5.1.3 Module views . . . 69
5.1.4 Components/Deployment view . . . 71
5.2 The code generation process . . . 71
5.2.1 Behaviour view . . . 72
5.3 Limitations . . . 77
5.4 Good practices . . . 78

6 Evaluation 79
6.1 Case Studies . . . 79
6.2 Swagger Codegen vs HEADREST Codegen . . . 81
6.3 HEADREST vs HEADREST with Resource Logic . . . 83
6.4 Experimental Study . . . 84
6.4.1 Context . . . 85
6.4.2 Experimental plan . . . 85
6.4.3 Experiment execution . . . 87
6.4.4 Quantitative Results . . . 88
6.4.5 Qualitative Results . . . 91
6.4.6 Conclusions . . . 92
6.4.7 Threats to validity . . . 93

7 Conclusions **95**
7.1 Summary . . . 95
7.2 Future work . . . 96

A Encoding rules **99**

B Predicates class **103**

C Experiment files **107**
C.1 Experimental Plan . . . 107
C.2 Work paper . . . 109
C.3 Presentation . . . 112

D Experiment Specification files **121**
D.1 OAS Spec . . . 121
D.2 HRSpec . . . 125

E Experiment classes **129**
E.1 Controller . . . 129
E.2 Controller Test . . . 131
E.3 Contact Representation . . . 135
E.4 Client API . . . 139
E.5 Boolean Validation Class . . . 143
E.6 Justified Invalid Class . . . 145

F HEADREST Codegen Form **147**

Bibliography **154**

Acknowledgements

I would like to thank my supervisors, Prof. Antónia Lopes and Prof. Vasco Vasconcelos for supporting me, pushing me to my limits and granting me the oportunity to develop such a project. It was a privilege to work together with them. Throughout the year I consolidated my knowledge and learned even more, which will definitely be beneficial to me as a professional. Thank you both.

Words will never be able to describe the gratitude I have towards my family. All the laughter, all the tears ... they made me into who I am today and for that I thank my father, my mother and my brother. They sacrificed so much in order to provide me with means of becoming better and I used the tools they gave me to their maximum to return the favour and the trust they put in me.

I also thank my friends and coleagues, for helping me through FCUL and helping me reach my objectives.

Thank you all.

À minha família.

Chapter 1

Introduction

This chapter describes our main motivation for the development of this work, its context, objectives and ends with the overall document structure.

1.1 Motivation

Web services with APIs that adhere to the REST architectural style, known as RESTful web services, are currently very popular. These services follow a client-server style, with stateless interactions based on standard HTTP verbs.

RESTful APIs were first introduced by Fielding on *Hypermedia as the Engine of Application State* (HATEOAS)[8] and the main idea was that the client code should not be written against a static service interface description but rather it should only use well known entry-points and explore the service through interaction of various requests and responses. However REST was popularised by services that do not actually follow this vision and, instead, have static interface definitions towards which clients are programmed.

The diversity in the design of web services lead to the proposal of the *Richardson Maturity Model* [18], which defines four levels based on how much web services are REST compliant. At level 0 essentially what we have are web services that use HTTP to communicate, usually based on Remote Procedure Invocation[14]. A remote method is invoked with the necessary information via the body of the request message and receive a result, resulting in URIs usually referring to some sort of service (e.g. `/FoodService`) and only one HTTP verb used. At level 1 we start to have the notion of resource, the URIs now reflect resources (e.g., `/restaurants/chinese/orders` and `/restaurants/indian/orders`) but the services still only use one HTTP method. At level 2 services make full use of all HTTP verbs combined with the resource URIs and this allows for more meaningful interaction with POST typically meaning addition of a resource, PUT updating, GET obtaining and so on. At level 3 we have services that make use of hypermedia controls and follow HATEOAS. This implies that URIs are exchanged back and forth and the user is provided with the requested resource as well as how to proceed from

there on, to explore the system.

In this thesis, the focus is on REST services at level 2. These services have static interface definitions towards which clients that consume their API are programmed. In order to facilitate this task, some sort of standardization of the API's interface was required, through means of a description document. This lead to the proposal of various *Interface Definition Languages* for documenting RESTful APIs (which we refer simply by RDLs). However, their expressiveness is limited mainly to the syntactic aspects of the API.

Most RDLs are able to describe the representation of the resources, for example, that there is a pet resource which is known to have a *name* of type *String*. They are also able to document the operations, for example, that if a *PUT* is called to create a *Pet* resource with a request body which does not represent a pet then a 400 code (Bad Request) will be sent back. This is limited since it mainly relates the structure of the data transmitted in the response with that sent in the request and does not allow us to describe more sophisticated things such as, if a pet is sent with a cute name then the response will have a hidden easter egg.[1]

Open API Specification (OAS) is currently *de facto* standard for documenting RESTful APIs and is supported by various industry heavyweights including Google, Microsoft, IBM, and Adobe. Its popularity results from its expressive power and versatility in what concerns the description of the request and response models and also to the large number of available tools, such as an online editor, a user friendly web UI, and several code generation tools, to name a few. However, like other RDLs, the expressive power of the language is rather limited when it comes to describing the way responses correlate with input and with server state.

In order to address the syntactic limitations of the current RDLs, the language HEADREST was proposed [6]. The language is inspired in OAS and relies on the use of refinement types to add extra restrictions over the structure of the data and on the use of logic assertions to support the description of the behaviour of the operations in the API.

One of the main advantages of having formal descriptions of RESTful APIs is the ability to generate a lot of boilerplate code for both clients and servers. Although there are various frameworks that ease the creation of RESTful applications and clients, a reasonable part of the code is tedious to write and amenable to be generated from abstract descriptions of the APIs. Code generation tools can also take advantage of development frameworks in order to generate less and simpler code, that is programmed at a higher level of abstraction.

This thesis addresses the problem of code generation for RESTful APIs described in HEADREST. We capitalise on the code generation tools developed for OAS and developed prototypes of code generation for clients and servers from HEADREST. We start by

[1] `https://en.wikipedia.org/wiki/Easter_egg_(media)` (Seen: 2017-10-20)

encoding the HEADREST specification into Open API Specification, taking advantage of the extension mechanisms provided by OAS, and then generate using an extension of the Swagger Codegen. We decided that we would be using Java and generating for the RestEasy framework.

1.2 Context

This work was developed in the context of the project CONFIDENT (Communication Contracts for Distributed Systems Development), a project financed by FCT, that started in May 2016 and is a collaboration between researchers at LASIGE and IT.

The project aims at making effective development and agile evolution of complex systems effectively viable, predictable and productive by developing tools for describing, inferring, and statically verifying component communication contracts for effective construction and evolution of complex distributed systems, notably RESTful applications.

The development of the language HEADREST [6] was the first step toward achieving these goals. The writing of HEADREST specifications is currently supported by an editor and validator made available through an Eclipse plugin. Techniques for generating tests from HEADREST specifications were also investigated in [7] and implemented in a prototypical tool. These tools are made available at the CONFIDENT project page at `http://rss.di.fc.ul.pt/tools/confident/`.

1.3 Objectives

The overall goal of this work is to contribute for the development of techniques and methods that facilitate the development of RESTful systems. Specifically, the aim of the work is to extend existing code generation techniques for REST API specifications to take advantage of behavioural properties that can be expressed with HEADREST.

To achieve this we:

- defined a mapping from HEADREST to OAS specifications, including HEADREST properties that go beyond what can be expressed by OAS
- extended the Swagger Codegen in order to take the extra properties into account, extending what it already generated with additional code
- extended the HEADREST language and analyser to allow us to generate even more code, that abstract resources and complete their representations
- generated client SDKs and server stubs and completed them to have a working client and server model, and assert the benefits of the extra generated code

- conducted an experiment where users completed client SDKs generated by HEADREST Codegen and Swagger Codegen, to compare both tools and further assess HEADREST Codegen benefits

1.4 Document structure

This document is structured as follows. In Chapter 2, we provide some background of the work and discuss RDLs, methods and tools more important and relevant for the work aswell as interesting information relative to the tools used. In Chapter 3, we do an overview of the encoding and generation process, and the extension we made to HEADREST in order to generate more code. In Chapter 4 and 5 we cover the encoding and generation process implementation. In Chapter 6, we show the resulting from the evaluation of our tool, which includes an experiment. In Chapter 7 we conclude the document with the work that can be done to extend the generation tool in the future.

Chapter 2

Background and Related Work

In this chapter we present a brief introduction to REST and RESTful APIs, we survey several RDLs and also some RESTful API frameworks that are currently used to program RESTful applications and their clients.

2.1 RESTful APIs

Representational state transfer (REST) is an architectural style developed to abstractly model the arquitecture of the web, based on the concept of resource [8]. According to Fielding and Taylor [9], a resource is a function $M_R(t)$ that maps each instance of time *t* on a set of values, which might be identifiers or resource representations. The identifiers serve to identify the resource of an interaction. To execute actions on resources, REST components need to use representations that capture the actual state or the intended state of the resource. An illustration of this instance can be seen in Fig.2.1 where a pet resource has an identifier and two representations associated to it.

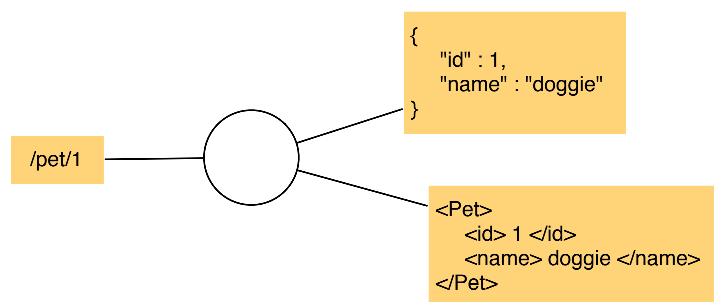


Figure 2.1: A resource at a given time instance – *Identifiers* on the left and *Representations* on the right.

In this project we focus on RESTful APIs that use HTTP for interaction and URIs[1] as

[1] Unique Resource Identifier

identifiers for resources. Thus, actions can be called via the HTTP verbs – GET, POST, PUT and DELETE – and the meta-data and data are sent in the *header* and the *body* of the request, respectively.

2.2 RESTful Interface Definition Languages

Interface definition languages (IDLs) are specification languages used to describe APIs in a language-agnostic way. They promote both modularity, as the various components only need to know the definition of the API, and interoperability of different systems, as they will communicate defined data via a defined communication protocol (in the case of web services, mostly HTTP).

For illustration purposes, we use as example a popular RESTful API of a pet store service, that includes methods to manage (add, update and get) pets, users and orders. The pet store service is available at `http://petstore.swagger.io` and has been often used to illustrate documentation of RESTful APIs.

In the next subsections, we provide an overview of some of the more relevant Interface Definition Languages for RESTful APIs (RDLs, for short): WADL, Hydra, RAML and Swagger. In brief, as illustrated in Fig.2.2.

- WADL is a RDL originated from its popular predecessor WSDL (Web Services Description Language), which is a popular IDL to describe SOAP Web Services.

- Hydra Core Language has a very different approach when compared to the other RDLs because it uses ontologies to add semantics to data.

- RAML and Swagger are two RDLs that preceded and strongly influenced the development of Open API Specification, which is the basis of HEADREST.

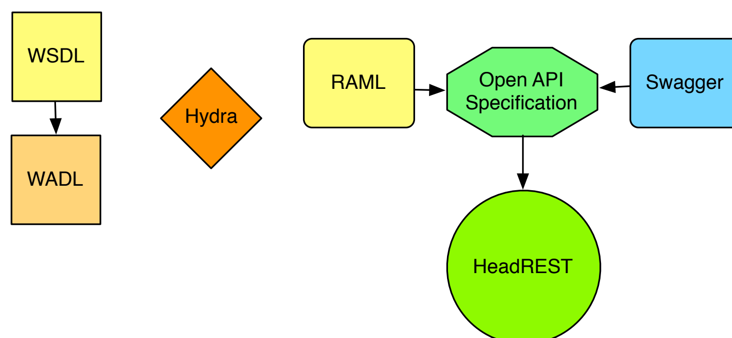


Figure 2.2: RDLs addressed in this section and how they relate

In what follows we discuss about the most relevant aspects of these RDLs. We also introduce HEADREST, the RDL addressed in this work and that was developed to be able to specify syntactic and semantic properties of RESTful APIs.

2.2.1 WADL (Web Application Description Language)

WADL is an XML-based RDL similar to WSDL but REST-oriented. It uses XML tags and structure, hence it is very verbose, but still simple to use. It models the resources and relationships between them through the XML hierarchy and specific tags. WADL's types are defined through XSD.[2]

Listing 2.1: WADL example

```
1  <application
2    xsi:schemaLocation="http://wadl.dev.java.net/2009/02 wadl.xsd"
3    xmlns:mns="petstore.swagger.io:v2"> <!-- mns = mynamespace -->
4
5    <resources base="petstore.swagger.io/v2">
6      <resource path="pet">
7        <method name="POST" id="addPet">
8          <request>
9            <representation mediaType="application/json" element="mns:Pet"/>
10         </request>
11         <response status="200">
12           <representation mediaType="application/json" element="mns:Pet"/>
13         </response>
14         <response status="400">
15           <representation mediaType="application/json" element="
                mns:ApiResponse"/>
16         </response>
17       </method>
18       <resource path="{petId}">
19         <method name="GET" id="getPetById">
20         <request>
21           <param name="petId" style="template" type="xsd:int">
22         </request>
23         <response status="200">
24           <representation mediaType="application/json" element="mns:Pet"/>
25         </response>
26         <response status="400">
27           <representation mediaType="application/json" element="
                mns:ApiResponse"/>
28         </response>
29         <response status="404">
30           <representation mediaType="application/json" element="
                mns:ApiResponse"/>
31         </response>
32         </method>
33       </resource>
34     </resource>
35   </resources>
36 </application>
```

[2]XML Schema Definition.

In Listing 2.1 we can see an example of a specification in WADL. It starts by defining the schema used for the specification file (line 2), followed by the name space that is used (line 3), which contains the data structures.

The example documents the "/pet" (lines 6-17), "/pet/{petId}" (lines 18-31). It says that if we call a POST HTTP verb, on the first path, with a body of type Pet, two outcomes are possible: code 200 returns the created resource (lines 11-13) and code 400 returns an ApiResponse type (lines 14-16) detailing the error. If we call a GET HTTP verb, on the second path, we need to fill the template with an integer (lines 20-22) and we are to expect a code 200 with the requested pet (lines 23-25) otherwise we get code 400 (invalid ID supplied – lines 26-28) or 404 (pet not found – lines 29-31).

In the listing below we can see an excerpt example of a grammar file,[3] which is used to encapsulate information, that can be then imported by various specifications.

```
<xs:schema targetNamespace="urn:yahoo:yn" ... xmlns="urn:yahoo:yn">
  <xs:complexType name="ImageType">
    <xs:sequence>
      <xs:element name="Url" type="xs:string"/>
      <xs:element name="Height" type="xs:integer" minOccurs="0"/>
      <xs:element name="Width" type="xs:integer" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

WADL's installation requires Maven and Git. It has both documentation generation from code and reversed through tools such as wadl2java.

2.2.2 Hydra

Hydra [15] is a JSON-based RDL, more concretely JSON-LD,[4] that adds semantics to data via ontologies such as Hydra Core Vocabulary, an ontology for RESTful APIs.

JSON-LD is designed around the concept of *context*, that provides a mapping from JSON to a RDF model, adding semantics to the data. The `@context` tag can be used to define the context and additional tags such as `@id` and `@type` enabling the addition of an URI and a type, respectively. The description itself can be demanding due to various new @-tags and the way nesting them works, which can involve some optimisations, but mainly because of the ontology behind the types' semantics.

The Hydra Core Vocabulary, as mentioned before, is a RESTful API ontology which adds the semantics of what an Operation, a Status and other REST specific concepts are.

Listing 2.2: JSON-LD example

```
1 {
```

[3] https://github.com/felps/Proxy/blob/master/soapui-4.0.1/Tutorials/WSDL-WADL/NewsSearchResponse.xsd (Seen: 2017-11-06).

[4] JSON Linked Data.

```
2    "@context": "http://petstore.swagger.io/v2/contexts/pet.jsonld",
3    "@id": "http://petstore.swagger.io/v2/pet/0",
4    "id": 0,
5    "category": {
6      "id": 0,
7      "name": "string"
8    },
9    "name": "doggie",
10   "photoUrls": ["string"],
11   "tags": [
12     {
13       "id": 0,
14       "name": "string"
15     }
16   ],
17   "status": "available"
18 }
```

In Listing 2.2 we can see an instance of a JSON-LD object. This object not only has an instance of `id` (line 4), `category` (lines 5-8), `name` (line 7), `photoUrls` (line 10), `tags` (lines 11-16) and `status` (line 17) but also adds a unique id to this object (line 3) and adds semantics to all its tags via the `@context` tag (line 2).

```
1  {
2    "@context": "http://www.w3.org/ns/hydra/context.jsonld",
3    "@id": "http://petstore.swagger.io/v2/doc/pet/",
4    "@type": "Link",
5    "title": "Petstore",
6    "description": "A link to pets with operations to create/update a pet.",
7    "supportedOperation": [
8      {
9        "@type": "Operation",
10       "title": "Add a new pet to the store",
11       "method": "POST",
12       "expects": "http://petstore.swagger.io/v2/doc/#Pet",
13       "returns": "http://petstore.swagger.io/v2/doc/#Pet",
14       "possibleStatus": [
15         {
16           "@context": "http://www.w3.org/ns/hydra/context.jsonld",
17           "@type": "Status",
18           "statusCode": 200,
19           "title": "Success",
20           "description": "Pet created successfully.",
21           ...
22         },
23         {
24           "@context": "http://www.w3.org/ns/hydra/context.jsonld",
25           "@type": "Status",
26           "statusCode": 405,
27           "title": "Invalid input",
28           "description": "The provided input was incorrect.",
29           "returns": "http://petstore.swagger.io/v2/doc/#ApiResponse",
30           ...
31         }
32       ]
33     }
34   ]
35 }
```

In the previous listing we can see a schema documentation of a POST operation on the pet resource (denoted by line 3). It starts with a context (line 2), to add semantics to our tags, followed by the id and the type of the resource we are *POST*ing to. The supported POST operation expects a Pet type request and will return the same type as response. The possible status are then documented in an array of possible status (lines 14-32).

We are not aware of any code generation tools for Hydra. In terms of installation, Hydra offers the Hydra Bundle which is a package for Symphony2, a PHP framework.

2.2.3 RAML (RESTful API Modeling Language)

RAML[5] is a YAML-based language. Supported types are defined in a type hierarchy shown in Figure 2.3, which include the types defined with JSON or XML Schemas.

A RAML document that specifies a RESTful API starts with the definition of a title, a *baseUri* and a *version*. Libraries may include data types, traits, resource types, schemas, examples among other things. Library imports are made via the *uses* tag. Annotations and traits can be also added to the main document. Entry-points to the API are defined via the nesting of URL fragments and documented with HTTP method tags and, possibly, other elements such as schemas to follow or examples to show.

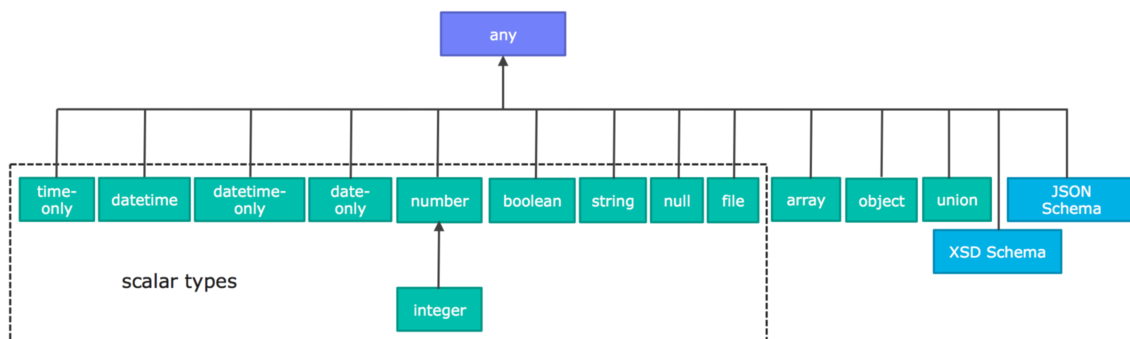


Figure 2.3: RAML type hierarchy

In Listing 2.3 we can see an example of a excerpt of specification of the Petstore API. The "hierarchy of tabs" allows for a very intuitive and human-readable specification.

Listing 2.3: RAML example

```
1 title: Petstore API
2 baseUri: http://petstore.swagger.io/{version}
3 version: v2
4
5 uses:
6   Pets: libraries/pets.raml
7
```

[5] https://raml.org/. (Seen: 2017-10-23).

```
8  /pet:
9    post:
10     body:
11       application/json:
12         type: Pets.Pet
13     responses:
14       200:
15         body:
16           application/json:
17             type: Pets.Pet
18       405:
19         description: Invalid input
20         schema: !include schemas/ApiResponse.xml
21         example: !include examples/ApiResponse.xml
22   put:
23     body:
24       application/json:
25         type: Pets.Pet
26     responses:
27       200:
28         body:
29           application/json:
30             type: Pets.Pet
31             schema: !include schemas/Pet.json
32             example: !include examples/Pet.json
33       400:
34         description: Invalid ID supplied
35         schema: !include schemas/ApiResponse.json
36         example: !include examples/ApiResponse.json
37       404:
38         description: Pet not found
39         schema: !include schemas/ApiResponse.json
40         example: !include examples/ApiResponse.json
41       405:
42         description: Validation exception
43         schema: !include schemas/ApiResponse.json
44         example: !include examples/ApiResponse.json
45   /{petId}:
46     type: integer
47     get:
48       responses:
49         200:
50           body:
51             application/json:
52               type: Pets.Pet
53           application.xml:
54             schema: !include schemas/Pet.json
55             example: !include examples/Pet.json
```

In the following listing we can see the library used in the previous example.

```
#% RAML 1.0 Pets Library
types:
  Pet:
    properties:
      id: integer
      category: Category
      name: string
      photoUrls: [string]
```

```
      tags: [Tag]
      status:
        enum: [available, pending, sold]
  Category:
    properties:
      id: integer
      name: string
  Tag:
    properties:
      id: integer
      name: string
```

There are several IDEs for RAML, for instance, API Workbench[6](a fully featured IDE for API Design), API Designer[7] (an intuitive web-based designer with a built-in API console), and many IDE plugins. RAML also has many code generation tools such as RAML for JAX-RS (a two-way code generator from RAML to JAX-RS), raml-python (which uses NodeJS to generate a framework in Python), Nobel (to create a REST API for Arduino board), and many more.[8]

2.2.4 Open API Specification

Open API Specification, formerly known as Swagger Specification, is the most commonly adopted RDL today. It is based on JSON/YAML and its types are inherited from the JSON Schema. OAS allows us to define what data is exposed by the operations on resources and how the client calls the operations. Currently, it is used to document many APIs, as can be seen in the APIs Guru website. [9]

Listing 2.4: OAS Operation Spec example

```
1  "/pet": {
2    "post": {
3    "tags": ["pet"],
4    "summary": "Add a new pet to the store",
5    "description": "",
6    "operationId": "addPet",
7    "consumes": [
8      "application/json",
9      "application/xml"
10   ],
11   "produces": [
12     "application/xml",
13     "application/json"
14   ],
15   "parameters": [
16     {
17       "in": "body",
18       "name": "body",
```

[6]http://apiworkbench.com (Seen: 2018-05-26).

[7]https://www.mulesoft.com/platform/api/anypoint-designer (Seen:2018-05-26).

[8]https://raml.org/developers/build-your-api. (Seen: 2017-10-23).

[9]https://apis.guru/browse-apis/. (Seen: 2017-10-06).

```
19          "description": "Pet object that needs to be added to the store",
20          "required": true,
21          "schema": {
22            "$ref": "#/definitions/Pet"
23          }
24        }
25      ],
26      "responses": {
27        "405": {
28          "description": "Invalid input"
29        }
30      },
31      "security": [
32        {
33          "petstore_auth": [
34            "write:pets",
35            "read:pets"
36          ]
37        }
38      ]
39    },
40    ...
```

In Listing 2.4 we can see the specification of a POST operation on the "/pet" URI identified resource (lines 1 and 2, respectively). Consumed (lines 7-10) and produced (lines 11-14) format types are indicated as well as a list of parameters (lines 15-25). In this specific case it is defined a request body of type Pet (denoted by line 17 and 22). In Figure 2.4 we can see a more visually appealing version of the specification, using the SwaggerUI online tool.

In abstract, an OAS file starts with the definition of the specification version, standard information (a description, API version, title, terms of service, contact and license), host (the API host) and a base path. An array *tags* is added to categorize the information with a name and brief description. A *schemes* array is added to provide the supported transfer protocol of the API. In the *paths* we detail the various paths of the API with its possible operations, which can be further detailed and that we went over previously. A *definitions* section is added to specify data types. Security definitions and external documents can also be added.

POST /pet Add a new pet to the store

Parameters

Name	Description
body * required (body)	Pet object that needs to be added to the store

Example Value | **Model**

```
Pet {
   id            integer($int64)
   category      Category {
                    id          integer($int64)
                    name        string
                 }
   name*         string
                 example: doggie
   photoUrls*    [
                 xml: OrderedMap { "name": "photoUrl", "wrapped": true }string]
   tags          [
                 xml: OrderedMap { "name": "tag", "wrapped": true }Tag {
                    id          integer($int64)
                    name        string
                  }]
   status        string
                 pet status in the store
                 Enum:
                  [ available, pending, sold ]
}
```

Responses

Code	Description
405	*Invalid input*

Figure 2.4: A print of SwaggerUI for the Petstore POST /pet specification

Listing 2.5: OAS Data Definition example

```
1  "Category": {
2    "type": "object",
3    "properties": {
4      "id": {
5        "type": "integer",
6        "format": "int64"
7      },
8      "name": {
9        "type": "string"
10     }
11   },
12   "xml": {
13     "name": "Category"
14   }
15 }
```

In Listing 2.5 we can see the basic Open API Specification object definition. In this case a Category is an object with an `id` field of type integer and a `name` of type string (line 4-7 and 8-10). An xml tag is then indicated at the end, so that encoding from JSON to XML is possible. Figure 2.5 shows how this description is rendered in SwaggerUI.

```
Category ∨ {
    id          integer($int64)
    name        string
}
```

Figure 2.5: A print of SwaggerUI for the Petstore Category specification

OAS is backed by a large ecosystem of tools that helps in designing, build, document, and consume RESTful APIs. Some tools are available online and others can be downloaded, ranging from code editors to code generators.

2.2.5 HeadREST

As mentioned before, HEADREST is a RESTful API specification language that adopts several concepts from OAS. The language has been developed in order to support the description of behavioural properties of RESTful APIs and its key ideas lie in:

- Detailing the structure of the data transferred on the different interactions, namely resources and representations with types
- Utilising pairs of pre and postconditions to express for each interaction (a) the relation between the data sent in the requests and received in the responses and (b) the state changes that occur afterwards

These ideas are materialized in HEADREST with three fundamental concepts:

- Refinement types, `x:T where e`, that consist of `x` values of type `T` that satisfy a property `e`.
- A type predicate, `e in T`, which returns *true* or *false* depending if the value `e` is or not of type `T`.
- Logical assertions, which are structures reminiscent of a Hoare triple, where the behaviour of an interaction is detailed, namely through quantification over resources and their representations.

Representation types

To describe the structure of the data transferred in the different interactions we use *representation types*. These are inspired in *models* used in the Open API Specification, enabling us not only to structure the data but also to express properties about its values. The supported representation types are: objects, arrays, refinement types, scalar types (including integer, boolean, string and URITemplate) and `any` (the top type).

Listing 2.6: HEADREST representation type example

```
1  type PetRep = {
2    ?id: integer,
3    ?category: Category,
4    name: string,
5    photoUrls: URI[],
6    tags: Tag[],
7    ?status: (x: string where x=="available" || x=="pending" || x=="sold")
8  }
```

In Listing 2.6 we present the representation of a pet resource, called `PetRep`. This representation dictates that a pet has an id, a category, a name, a list of photo URLs, a list of tags and a status that matches specific strings. As we can see, the object is described using basic types, reference to other defined types and refinement types. Additionally, optional fields are denoted with “?”.

Request & Response

REST data is transferred in request and response, following a standardized HTTP structure. HEADREST abstracts that structure with two predefined variables, *request* and *response* which are of the following types:

```
type Request = {
  location: URI,
  ?template: {},
  ?header: {}
}
```

```
type Response = {
  code: integer
  ?header: {}
}
```

A request may have information sent from an URI template, such as a path or query argument. For instance "pet/{id}" would generate a specific type of request which would have the following subtype:

```
{
  location: URI,
  template: {id: any},
  ?header: {}
}
```

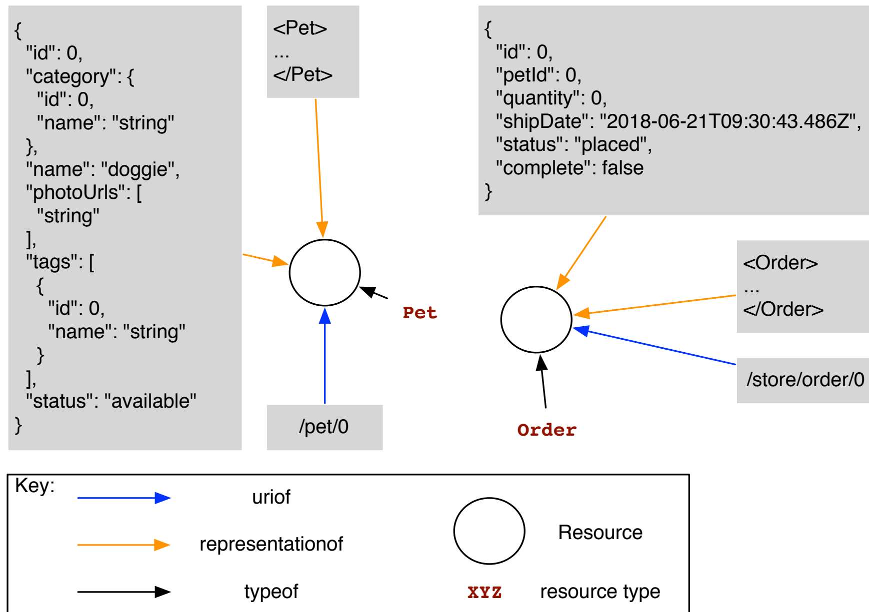


Figure 2.6: A petstore system state

States

RESTful APIs enable interactions that let us observe and modify resources in the system, or part of them. In terms of RESTful API specification, it is considered that a set of values associated to a resource of the system at a given instance of time is what defines the system at that instant.

Each resource is assumed to be of a certain type, for instance, in the Petstore service we need to manage three resources: pets, users and stores. These resources can be declared in HEADREST as follows:

```
resource Pet, User, Store
```

In Figure 2.6 we can see an example of a petstore system state. The states represent what values associated with a resource are, through the primary binary operations `uriof`, `representationof` and `in` (typeof in the image).

Assertions

HEADREST supports the formal description of observations and state changes resulting from interactions exposed on a RESTful API through a set of assertions with a Hoare-triple structure. Concretely, assertions in the shape of

$$\{\phi\}\ a\ \ t\ \{\psi\}$$

where a is an action (GET, POST, PUT, DELETE), t is an URI template and ϕ, ψ are expressions of boolean type. The ϕ formula, called precondition, addresses the state

in which the action is executed and the data transmitted in the request while ψ, called postcondition, addresses the state resulting of the execution of the action and the values transmitted in the response. The assertion says that if a request for an execution of an action a over an expansion of t sends data that complies to ϕ and the action is realised in a state that fulfils ϕ, then the data transferred in the response fulfils ψ, as well as the state resulting of the execution of the action.

Listing 2.7: Assertion example

```
1  // addPet 200, If pet doesn't exist
2  {
3    request in {body: PetRep} &&
4    (isdefined(request.body.id) ==>
5      (forall pet:Pet .
6        (forall petRep:PetRep .
7          petRep representationof pet && petRep.id != request.body.id
8        )
9      )
10   )
11 }
12 POST /pet
13 {
14   response.code == SUCCESS &&
15   response in {body: PetRep} &&
16   (isdefined(request.body.id) => response.body == request.body) &&
17     (exists pet:(p: Pet where response.body representationof p) .
18       expand(/pet/{petid} , {petid: response.body.id}) uriof pet
19   )
20 }
```

Listing 2.7 presents an example of an assertion. The precondition states that if a request has a body of type `PetRep` (line 3) and value for `id` is provided (line 4 – it is an optional field of `PetRep`), then, if we are in a state in which no pet has the same id as the one we are about to add (lines 5-9), this POST interaction will result in a success response code (line 14), returning in the response body the representation of the pet (line 15) that was just added and acknowledges that the state will have that pet in the system with said representation (lines 16-19).

2.2.6 RDLs side by side

We conclude the RDL section with Table 2.1, a consolidated view of the analysed RDLs, for easy comparison. The Syntax property will cover the writing of the specification file and the Types property describes what we can specify in terms of types. I&O relation, means the possibility to establish some sort of connection between the request and response of an execution. Installation refers to how easy it is to setup the specification tools. Codegen refers to possible code generation tools available for the RDL.

RDL	Properties	Description
WADL	Syntax	Simple XML tags and structure
	Types	XML Schema Definition
	I&O relation	Indication of mandatory fields and return values, but no relation
	Installation	Maven + Git
	Codegen	wadl2java
Hydra	Syntax	JSON-LD, which is JSON with new tags
	Types	JSON standard types with the addition of semantics through the @type tag
	I&O relation	Relates an expected information with a returned information, in case of success
	Installation	Installer bundle
	Codegen	-
RAML	Syntax	YAML based (easy to write and read)
	Types	Complex type hierarchy which extends XSD and JSON Schema capabilities
	I&O relation	Relation between expected content and responses, in case of success and others
	Installation	IDE Installer or online via internet browser
	Codegen	Yes
OAS	Syntax	JSON & YAML support. Online tools for live-feedback on code writing.
	Types	JSON Schema
	I&O relation	Relation between expected content and responses, in case of success
	Installation	Easy installer, online via internet browser or plugins
	Codegen	Swagger Codegen
HEADREST	Syntax	Data definition is simple with JSON like format. Hoare triples can get complex
	Types	Objects, arrays, refinement types, scalar types and `any`
	I&O relation	Full input & output relation discrimination
	Installation	Eclipse IDE plugin
	Codegen	-

Table 2.1: Overview of the RDLs analysed and their properties.

2.3 RESTful API frameworks

RESTful API Frameworks support the development of RESTful applications and their clients at a higher level of abstraction, making code simpler and less prone to errors. Therefore it comes with no surprise that code generation tools take advantage of such frameworks. For example, the Swagger Codegen has various RESTful API Frameworks for which it generates client code as well as server code.

In what follows, we provide an overview of some of the most popular Java-based frameworks. More concretely, for client-side we consider: RestTemplate, Vertx, OkHTTP, Retrofit 1.x/2.x, Feign, Jersey 1.x/2.x and RestEasy; and for server-side we consider: JAX-RS, MSF4J, Spring, Undertow, Inflector, PlayFramework and RestEasy.

2.3.1 Client frameworks

RestTemplate is the central Spring class for client-side HTTP access. Because it is synchronous, it poses problems in terms of scalability, which in general is very important for web services. Nevertheless, the invocation of methods on a RestTemplate instance makes it very easy to consume web services if synchronicity is not an issue.

Vert.x is a toolkit for building reactive applications on the JVM. It is asynchronous oriented, with the use of lambda expressions, streams and concurrency related classes.

OkHTTP is a simple HTTP client for both Android and Java Applications. It supports both synchronous blocking calls and asynchronous calls with callbacks. It is very low level and requires that programmers construct HTTP requests themselves, with the use of fluent API helpers. After that, they just need to invoke the OkHttpClient instance and the response is returned.

Retrofit is a type-safe HTTP client for both Android and Java built on-top of OkHTTP in a annotation-based matter. It uses annotations to map the method invocation into the appropriate request. It uses annotations similar to that of JAX-RS and new ones, such as @Header which can be used to map an argument to a Header field on the HTTP request.

Feign is a Java to HTTP client binder inspired by Retrofit, JAXRS-2.0, and WebSocket. It only supports text-based HTTP APIs and, as a result, it is very easy to use and test. Its usage typically starts with the creation of a JAX-RS like annotated interface to represent the service and its operations, and Plain Old Java Objects (POJO) to represent the data structures. The standard API invocation consists of the creation of a Retrofit java object instance, via a fluent Builder. Creating an instance of the API interface via the Retrofit instance and preparing a Call followed by its execution

Jersey, by Oracle, and **RestEasy**, by JBoss, are both powerful JAX-RS toolkit implementations. Jersey has a fluent API client creation and operation invocation. RestEasy has the peculiarity of being able to use the server API interface (the interface annotated with JAX-RS to expose the service) to automatically create the underlying HTTP connec-

tor (called a proxy because it mimics the interaction with a server in a RMI like fashion). This allows for enhanced modularity and server implementation abstraction.

2.3.2 Server frameworks

JAX-RS, which stands for Java API for RESTful Web Services, is an API specification that provides support in creating RESTful web services. JAX-RS defines annotations that allows simplifying the development and deployment of web service clients and endpoints.

MSF4J, which stands for Microservices Framework for Java, is a lightweight high performance framework for developing and running microservices that supports JAX-RS and Swagger annotations. The generated server is composed of an <API>Service class which has typical JAX-RS annotations to expose the API and of an Application class in which the service class is deployed onto a MicroservicesRunner class (which will work as a container).

Spring is a container and cloud-based framework with various foundational support such as JDBC and JPA, and also supporting dependency injection and Aspect-oriented Programming. Spring provides annotations like "@RestController" and "@RequestMapping("/uri")" that allow programmers to tell the container how to map the resources

Undertow is a flexible performant web server written in Java, providing both blocking and non-blocking APIs. It is sponsored by JBoss and is the default web server in the Wildfly Application Server. The server is configured by chaining handlers together. Undertow has its unique flavour precisely because our code will comprise of the creation of the Undertow instance through a factory, configuring the server, chaining the handlers and finally starting the server, all in a fluent API fashion

Inflector is a JAX-RS and javax servlet-based server, which routes requests to appropriate controllers. Swagger Inflector utilizes the Swagger Specification as a DSL (Domain Specific Language), removing the need for annotations and other wiring necessary to produce Swagger descriptions. It utilizes *converters* and *processors*, and others, to deal with the serialisation, processing and validation of the data. The swagger classes are used throughout these phases.

PlayFramework is a framework built on Akka, a toolkit for building highly concurrent, distributed, and resilient message-driven applications for Java and Scala. It presents itself as a developer friendly, scale predictable and modern web & mobile framework. It uses basic concepts such as Action, Controller and Results. Action is a typical Java method that takes care of a request. Controllers are the classes that group Actions together and Results are the classes that represent an HTTP Result, having helpers to create standard code responses with ease. The linking of requests to Actions is done in a configuration file.

RestEasy is the JBoss implementation of the JAX-RS with WildFly Application Server integration. It utilizes the typical JAX-RS annotations to map HTTP requests to its corre-

sponding methods. Features range from caching, gzip content-encoding, OAuth2 support and many more. Like the other JAX-RS implementations, the use of annotations to delegate the mapping of the request to a class/method makes it exceptionally easy and modular to program web services, not only that but we also have access to helpers to build responses.

In this work, we decided that our generation prototype would make use of the RestEasy framework since it can generate both client and server code, and the fact that it is easy to install and use with Eclipse.

2.4 Program generation for RDLs

The development of software has two major parts, the creative part and the boring part. The creative part is where software engineers apply their expertise to create reliable, scalable solutions. The boring part refers to the fact that a large part of the code that we sometimes need to write is boilerplate code, i.e., code that has to be written in many places with little or no alteration. Program synthesis eases the creation of solutions through the customisable generation of big chunks of code [16].

In general, there are three main approaches to program synthesis [12]:

- Deductive, where there is an almost direct translation from specification to actual code, in a very compiler like fashion.
- Inductive, where we give input-output to the generator and let it "cook" until we get a program that responds to our requirements, using genetic approaches for example.
- Hybrid, where we have a meta-synthesis framework which allows the user to define the program space, through grammar or skeleton programs, and lets us define some guidelines for the generation algorithm encoding the program in SAT/SMT restrictions or inverse semantics of the program's operators.

One of these approaches is then adopted by what is known as a *Code Generator* (i.e., a compiler, a computer software that writes software for us). It is code writing code. Code generators can be divided in two different categories [16]:

- Passive code generators, that run once producing code to be completed by the developer.
- Active code generators, that produce developer-free code (code that the developer is not supposed to touch), that is continually regenerated throughout the project's lifecycle.

Both approaches can save time and effort with the subtle difference on how the generated files are maintained over the course of the project. As explained in [16], "If the code generator is made part of the build process and generates fresh source each time a build is instigated, then we are using active generation. If the output from the same generator is taken by the software engineer, modified, and placed under source control, then we have passive code generation".

In the next subsections we present three code generators that are used to generate from different used RDLs.

2.4.1 wadl2java

wadl2java[10] is a code generation tool for, as the name says, creating Java code from a WADL specification file. In particular, the wadl2java tool generates a Java client from the provided WADL specification file. It uses the deductive approach and, given its compiler like nature, uses an Abstract syntax tree (AST) to generate code from. It starts by obtaining the AST that represents the WADL file and then transforms it into an intermediate representation (IR) AST that is very close to Java code which is finally compiled directly into Java files.

2.4.2 RAML for JAX-RS

RAML for JAX-RS[11] is a project that enables code generation from RAML specifications. In particular, RAML for JAX-RS generates Java server code, also in a deductive way. As in the previous generator, it starts by obtaining the AST representing a specification file, converts it to an IR and finally compiles the code to Java files. We start noticing a consistent pattern with deductive compilers, which will change in the following last generator.

2.4.3 Swagger Codegen for OAS

The Swagger-Codegen[12] contains a template-driven engine to generate API clients and server stubs by parsing Open API specifications. The templates are written with Mustache.[13]

The Mustache template file has the structure as shown in Listing 2.8. The `{{name}}` tag (line 1) in a basic template will try to find the name key in the current context and, if there is no name key, the parent contexts will be checked recursively. If the top context is reached and the name key is still not found, nothing will be rendered. A section begins

[10] `https://github.com/javaee/wadl/tree/master/wadl` (Seen: 2017-10-25).
[11] `https://github.com/mulesoft-labs/raml-for-jax-rs` (Seen 2017-10-25).
[12] `https://github.com/swagger-api/swagger-codegen` (Seen: 2017-10-11).
[13] `https://mustache.github.io` (Seen: 2017-10-11).

with a hash and ends with a slash (lines 3-5). That is, `{{#in_ca}}` begins a "in_ca" section while `{{/in_ca}}` ends it. The behaviour of the section is determined by the value of the key. False or empty list values will result in the section not being rendered while non-false value and non-empty list will be rendered or will have its elements rendered, respectivelly. Mustache has more complex instructions, but these will be our basic blocks for this project.

In summary, the compilation process of a Mustache template starts with the template file shown below:

Listing 2.8: Mustache template example

```
Hello {{name}}
You have just won {{value}} dollars!
{{#in_ca}}
Well, {{taxed_value}} dollars, after taxes.
{{/in_ca}}
```

that provided with the model below:

Listing 2.9: Mustache model example

```
{
  "name": "Chris",
  "value": 10000,
  "taxed_value": 10000 - (10000 * 0.4),
  "in_ca": true
}
```

compiles into the file below, via the Mustache compiler:

Listing 2.10: Mustache compiled file example

```
Hello Chris
You have just won 10000 dollars!
Well, 6000.0 dollars, after taxes.
```

The Swagger Codegen starts by reading its specification file into an AST and even though it contains all the information of the specification file, given that Mustache is a logic-less template engine, this AST is converted/processed into a template ready IR, which is a very close representation to the AST but with added information so that it can be used as a model for the template. It ends compiling the various template files, that represent in whole a project, provided with the model. The listing below provides a simplified Swagger Codegen template example, used for the operation method generation of a Java JAX-RS project.

Listing 2.11: Simplified Mustache template for ApiImpl classes

```
{{#operation}}
  @Override
  public Response {{nickname}}({{#allParams}}...{{>serviceBodyParams}}...,
      {{/allParams}}) throws NotFoundException {
    // do some magic!
    return Response.ok().entity("magic!").build();
  }
{{/operation}}
```

Swagger Codegen sample from PetStore

Since the HEADREST Codegen capitalises on the Swagger Codegen, in what follows, we consider the generated code for the PetStore API Specification,[14] specifically for a RESTEasy client[15] and server.[16]

In Figure 2.7 depicts a runtime view of the generated system, showing how the various components interact with each other.

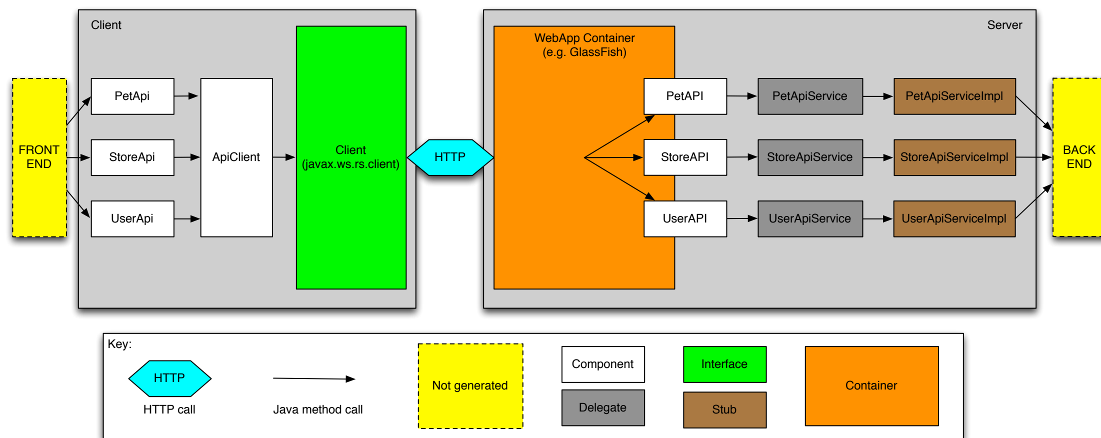


Figure 2.7: Pet store runtime view

We will be refering to various classes, packages and methods that are illustrated in the following figures: Figure 2.8 presents a module view of the generated code for the client side, Figure 2.9 illustrates a module view of the generated code for the server side. Both client and server generated code include a /model package with a set of classes that map resources as POJOs with Jackson, for JSON serialization/deserialization, and Swagger annotations, to have embedded specification.

On the client side, there is an ApiClient class used to send HTTP requests, which uses the JAX-RS Client. This is used by the various <service>Api classes (e.g. PetApi) which is a Remote Method Invocation (RMI) like class, with methods such as addPet(Pet body) that take a model type data and call the ApiClient accordingly, setting the correct method, the correct headers, passing the body, etc.

On the server side, in the /api package we have <service>Api class, which has typical JAX-RS annotations to map HTTP requests into methods, and then delegates it to a corresponding <service>ApiService abstract class which has a corresponding stub implementation in the /impl package.

[14] `http://petstore.swagger.io` (Seen: 2017-10-25)
[15] `https://github.com/swagger-api/swagger-codegen/tree/master/samples/client/petstore/java/resteasy` (Seen: 2017-10-25).
[16] `https://github.com/swagger-api/swagger-codegen/tree/master/samples/server/petstore/jaxrs-resteasy/default` (Seen: 2017-10-25).

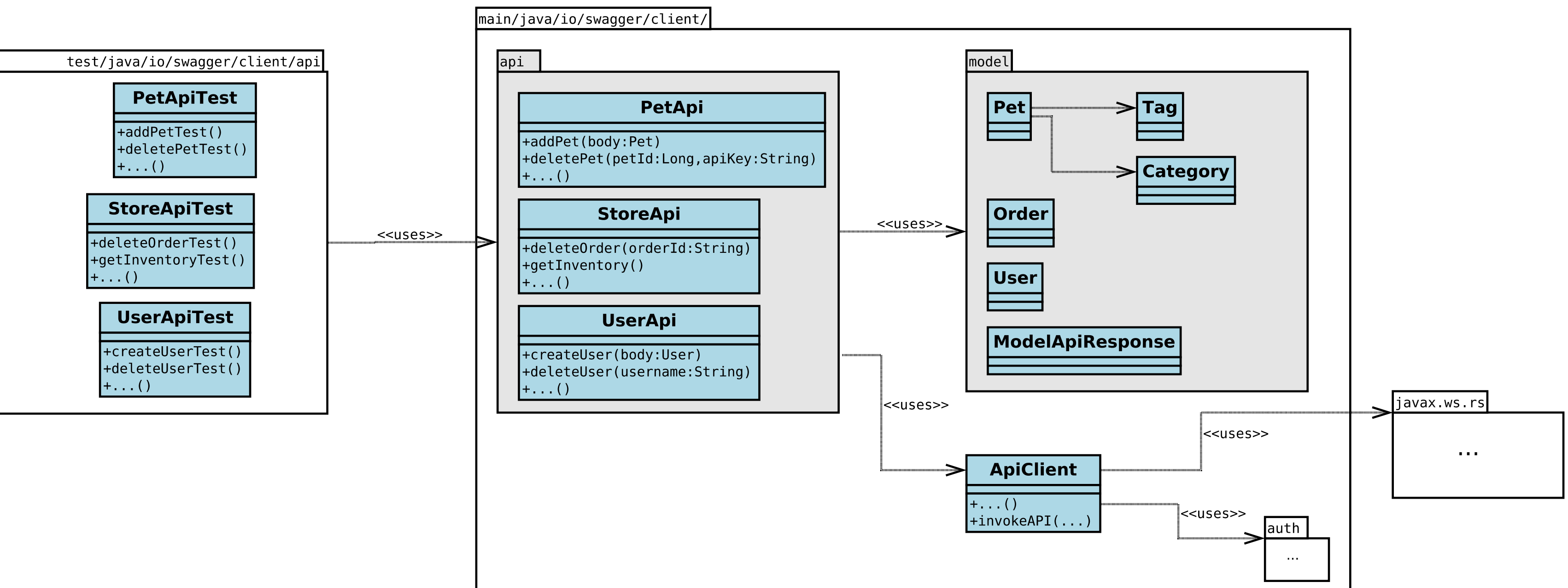


Figure 2.8: Pet store client uses view

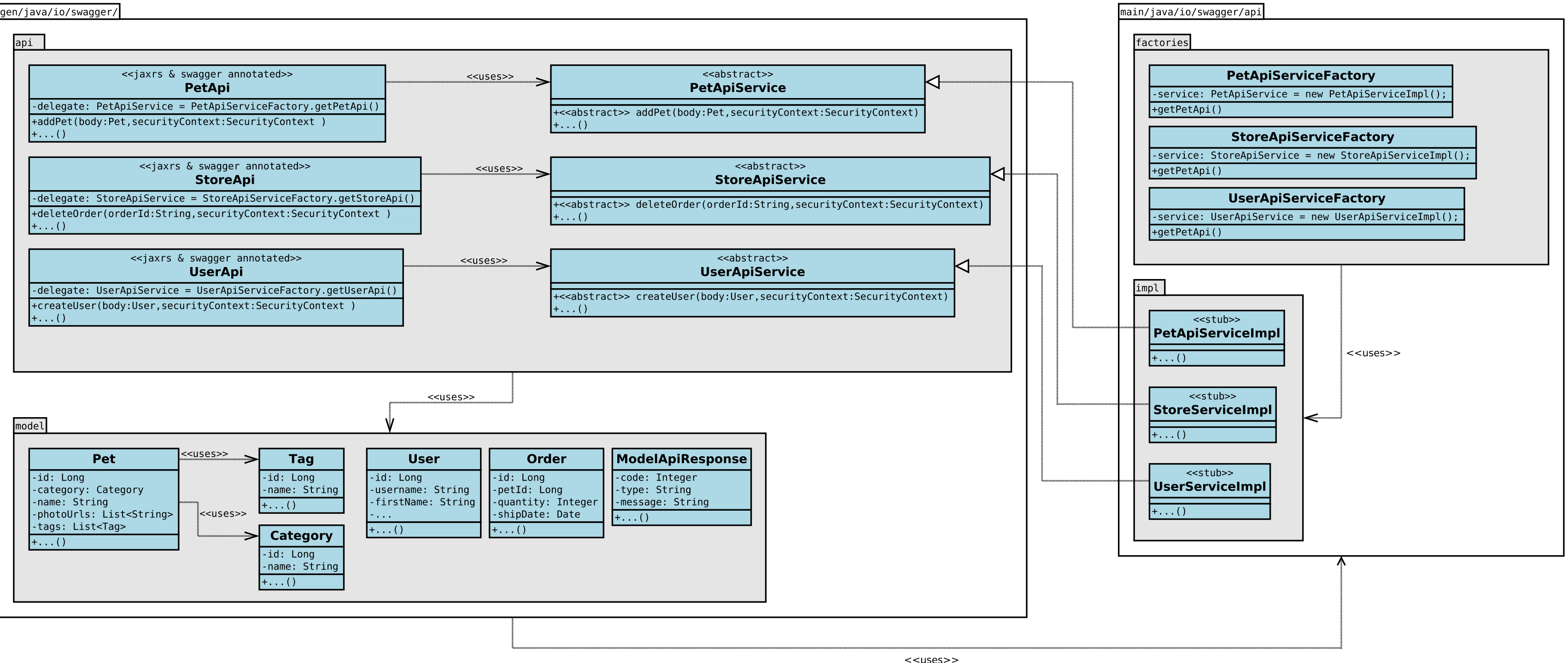


Figure 2.9: Pet store server uses view

Chapter 3

HEADREST Codegen Overview

This chapter presents an overview of the main ideas underlying HEADREST Codegen, the tool that encodes HEADREST specifications into OAS and extends Swagger Codegen, OAS generation tool, in order to generate client SDKs and server stubs for RESTful services with APIs described with HEADREST.

3.1 HEADREST Codegen in a nutshell

As mentioned before, HEADREST adopts many concepts from Open API Specification namely in what concerns the use of models to describe the requests and responses and the underlying data schema. HEADREST then relies on the use of refinement types to add extra restrictions over data and on the use of logic assertions to support the description of the behaviour of the operations in the API.

Since the generation of code from OAS specifications was already addressed by a family of code generation tools available for a panoply of different programming languages and REST frameworks, the decision was to capitalise on the underlying generation techniques and extend them to accommodate the extra expressiveness of HEADREST. This was achieved through the definition of a mapping from HEADREST to OAS specifications. The main idea of this mapping is to identify the properties expressed in a HEADREST specification that can also be expressed in OAS and take care of their encoding, attempting to use the native OAS as much as possible, to better capitalise its generation tool. By making use of the mechanisms provided by OAS to extend its specifications at different points with additional data, the generated OAS specifications also include HEADREST properties that go beyond what can be expressed in OAS. The extension of existing code generation techniques for OAS mainly targets the generation of code that takes advantage of these properties. This work was developed around Swagger Codegen for Java and JAX-RS framework.

The extension of Swagger Codegen in order to take into account refinement types is achieved by equipping the classes that represent the models with validation methods,

providing a way of both clients and servers to validate the extra conditions imposed on the data models. The extension for assertions is achieved in two ways: (1) for server code, we create a verification structure to validate the pre and postconditions (2) for client code, we extract the information on the assertions that is not state related and use it to create a structure that is available for programmers so they can verify if they meet the required conditions for a successful interaction with the server.

In order to be able to generate more code for the server, related with resources and their representations, the HEADREST language was extended with new elements. This extension, which we designate by *Resource Logic*, allows us not only to generate more code but also produce HEADREST specifications that are easier to write and read. This extension of the language supports the declaration of predicates on resources and the binding of representations to resources, adding an implementation to those predicates. Code generated from specifications that use the resource logic has richer interfaces, with methods that represent the resource predicates, and the implementation of the resource predicates on representations that represent resources.

In the rest of this chapter, we discuss the key aspects of the encoding process of HEADREST into OAS and discuss the main ideas explored in the generation process of the server code and also the client code, ending with an overview of the resource logic. More details about encoding and generation, covering also the implementation aspects, are discussed in Chapters 4 and 5, respectively.

3.2 Encoding of HEADREST specifications into OAS

The encoding phase can be broken down in two parts: types and assertions. OAS already supports various types, but only covers part of HEADREST types. To cope with this we attempted to encode as much of HEADREST as possible into native OAS taking advantage of OAS extension mechanisms when necessary.

3.2.1 Type encoding

Encoding HEADREST basic types (`integer`, `boolean`, `string`) into OAS is pretty straightforward, as we use the OAS schema supported types. Object and Array types however demand some recursion and may need some processing if internal refined types occur. In the following listing we illustrate a basic HEADREST object that represents a person.

```
Person = {
  name: string,
  age: integer
}
```

The previous type is then encoded into an OAS type, shown below.

```
Person:
  type: "object"
  properties:
    name:
      type: "string"
    age:
      type: "integer"
```

The biggest challenge in type encoding arises with refinement types, one of HEADREST novelties when describing data. From refinement types, structured as `x:T where e` the refining expression cannot be encoded into native OAS. We decided to approach this by encoding the type `T` directly and appending the refinement expression `e` (the extra information that needs to be carried on) in a OAS extension tag. For instance, the HEADREST refinement type

```
type Digit = (d:integer where d >= 0 && d < 10)
```

is encoded in OAS as:

```
Digit:
  type: "integer"
  x-refinement: "d >= 0 && d < 10"
```

While the previous types (`Person` and `Digit`) can be directly mapped to OAS, HEADREST contains other types that do not have a direct/obvious counterpart in OAS, which make things a bit tricky. Some examples, shown below, include: intersection types, singletons, union types and optional fields in objects.

```
type IntersectType = Digit & OddInteger
type Singleton = ["a"]
type UnionType = ["a"] | ["b"]
type OptionalFieldObject = {?b: integer}
```

These types however are not part of the core fragment of HEADREST and hence, for analysis purpose, get transformed in elements of the core. For instance, `Singleton` is transformed into `(x:any where x == "a")`. In order to use this type, processing is done in order to revert this translation, as the type gets much easier to encode into OAS that way.

While most of the HEADREST types are covered by the encoder, there are some types for which no appropriate encoding in OAS was found. This is the case for instance of `any` and `[null]`, which are currently unsupported by our tool.

3.2.2 Assertion encoding

While we may have various assertions over the same operation in HEADREST, operation specification in OAS only supports the description of a single request model (defining the data schema required for success) and the response models for different response codes. This results from the fact that OAS operation specification is, to some extent, happy path oriented.

In OAS, the specification of an operation is done within a path verb member (see lines 1 and 2 of the following listing) and allows to specify (1) the parameters that are required (body, path, query) and their types, and (2) the response types for various response codes.

In Listing 3.1 we present an example of an OAS specification resulting from the encoding of HEADREST specification of PetStore example discussed in Chapter 2 (see excerpt in Listing 2.7). The OAS specification is presented in JSON format for a clearer view of the map-like structure of the OAS specification. As mentioned before, there is only one request type specified per operation (denoted by the single "parameters" field) and, hence, the encoding has to find the request type associated with the success scenario. This is easy if we have a single HEADREST assertion for that OAS operation with sucess response code, but HEADREST specifications may have various assertions over the same operation and some of them may even have the same response code. To cope with this it was necessary to merge the various request types in assertions (for success codes) into one and the various response types (for each response code) into one as well. This merging restricts HEADREST specifications for which code generation is supported (see details in Chapter 4). Due to the fact that there are various assertions over the same operation, we propagate them making use of extension mechanisms at the operation level, namely via the "x-axioms" tag.

Listing 3.1: Example of OAS operation

```
1  "paths": {
2    "/pet": {
3      "post": {
4        "parameters": [{
5          "in": "body",
6          "name": "body",
7          "description": "Pet object that needs to be added to the store
               ",
8          "required": true,
9          "schema": {
10           "$ref": "#/definitions/Pet"
11         }
12       }],
13       "responses": {
14         "200":{
15           "description": "Success",
16           "schema": "#/definitions/Pet"
17         },
18         "405": {
19           "description": "Invalid input"
20         }
21       },
22       "x-axioms": <assertion list>
23     }
24   }
25 }
```

3.3 Generation

With all the additional information that was used to create and extend the encoded OAS specification, what remains is the generation of the RESTful API. This required the extension of the Swagger Codegen to take the new extension tags into account. As in the

previous section, we discuss the types first and then assertions.

3.3.1 Type generation

In the case of refinement types, from the data under extension tags included in the generated OAS specification we generate validation methods that validate instances of that type. This required the extension of Swagger Codegen to create the validation method and be able to translate HEADREST expressions used in refinement types into Java expressions. As shown in the following listings, a static method is generated that validates a given instance. Refinements can be applied to primitive types and Java primitive type classes (e.g., Integer, String). The fact that these classes are final prevents the use of instance methods (types like Integer cannot be extended to include the validation method, and we wanted to keep everything coherent all across the model classes).

The refinement type:

```
1  type PetRep = {
2    ?id: integer,
3    ?category: Category,
4    name: string,
5    photoUrls: URI[],
6    tags: Tag[],
7    ?status: (x: string where x=="available"||x=="pending"||x=="sold")
8  }
```

is encoded to an OAS type:

```
1   PetRep:
2     type: "object"
3     required:
4     - "name"
5     - "photoUrls"
6     - "tags"
7     properties:
8       id:
9         type: "integer"
10      name:
11        type: "string"
12      tags:
13        type: "array"
14        items:
15          $ref: "#/definitions/Tag"
16      category:
17        $ref: "#/definitions/Category"
18      photoUrls:
19        type: "array"
20        items:
21          $ref: "#/definitions/Uri"
22      status:
23        type: "string"
24        enum:
25        - "available"
26        - "pending"
27        - "sold"
28
29  Uri:
30    type: "string"
31    x-refinement: ...
```

where `Uri` is present for illustration purposes. The generated validation methods, included in the Java classes for `PetRep` and `Uri`, are as follows.

Listing 3.2: Examples of generated validation methods

```
1  public static boolean validate(PetRep instance) { // PetRep
2    return instance.getPhotoUrls().stream().allMatch(e->Uri.validate(e));
3  }
4
5  public static boolean validate(String instance) { // Uri
6    return Predicates.matches(instance, URI_REGEX);
7  }
```

3.3.2 Operation generation

For servers we generate code that verifies the pre/postconditions. It starts by capturing the preconditions, followed by a stub response creation, then the postconditions and asserting that if a precondition was met, then a postcondition was fulfilled.

Extension tags that carry the various pre/postconditions for each operation are added to the generated API's operation method, generated natively by Swagger Codegen. Pre/-postcondition variables are declared and assigned a Java expression extracted from the respective HEADREST expression (similar to the refinement expression translation to Java), and in the end various pre/postcondition verifications are created, as illustrated below. This structure captures the concept of a Hoare logic triple where: when the precondition is met, executing the operation establishes the postcondition.

The assertion specification:

```
1  // addPet 200, if pet does not exist
2  {
3    request in {body: PetRep} &&
4    (isdefined(request.body.id) =>
5      (forall pet:Pet .
6        !pet.hasid(request.body.id)
7      )
8    )
9  }
10   POST /pet
11 {
12   response.code == SUCCESS &&
13   response in {body: PetRep} &&
14   (isdefined(request.body.id) => response.body == request.body) &&
15     (exists pet:Pet .
16       response.body representationof pet &&
17       expand(/pet/{petid} , {petid: response.body.id}) uriof pet
18   )
19 }
20
21 // addPet 200, if pet exists and id is over 9000
22 {
23   request in {body: PetRep} &&
24   (isdefined(request.body.id) =>
25     (exists pet:Pet .
26       pet.hasid(request.body.id) && request.body.id > 9000
27     )
```

```
28     )
29   }
30     POST /pet
31   { ... }
32
33   // addPet 405, Invalid input
34   {
35     !(request in {body: PetRep})
36   }
37     POST /pet
38   {
39     response.code == INVALID_INPUT &&
40     ...
41   }
```

generates the following operation method:

```
1  public Response petPost(final PetRep body, ...) throws NotFoundException {
2    boolean pre1 = PetRep.validate(body) && ...;
3    boolean pre2 = PetRep.validate(body) && body.getId() > 9000 && ...;
4    boolean pre3 = !PetRep.validate(body);
5
6    //TODO create response
7    final Response response = Response.ok().entity(...).build();
8
9    boolean pos1 = Objects.equals(response.getStatus(), 200) && ...;
10   boolean pos2 = Objects.equals(response.getStatus(), 200) && ...;
11   boolean pos3 = Objects.equals(response.getStatus(), 405) && ...;
12
13   if(pre1) assert pos1 : "{pre1} petPost {pos1} failed.";
14   if(pre2) assert pos2 : "{pre2} petPost {pos2} failed.";
15   if(pre3) assert pos3 : "{pre3} petPost {pos3} failed.";
16
17   return response;
18 }
```

For generating client code from assertions, we take advantage of specific parts of the preconditions and postconditions. As seen in the previous listing, assertions often have a request/response refining part (see lines 3 and 13) and a state related part (see lines 4-8 and 14-18), with postconditions typically starting with the response code. Clients cannot verify the state part of assertions (this can only be verified at the server-side). So we decided to extract every part of the precondition of assertions relative to success scenarios that can be processed at the client-side and generate methods that the client can use to verify whether he is guided towards success or not – note the use of *guided*, as without the state verification we cannot really assume success will happen.

This makes it possible to identify potential failure scenarios even before communicating with a server. An example of this client-side generated code is shown below, where we can see the declaration and definition of variables that capture the client useful part of the preconditions and disjoin them, as complying with one of the partial preconditions may lead to success.

Listing 3.3: Generated method at client-side for validation of operation precondition

```
public boolean petPostRequiredSuccessPrecondition(PetRep body) {
```

```
    boolean pre1 = PetRep.validate(body);
    boolean pre2 = PetRep.validate(body) && body.getId() > 9000;

    return pre1 || pre2;
  }
```

3.4 Resources and representations

HEADREST specifications declare resources and assertions express properties about the state of these resources and their representations, often with quantificatiers over them. Given the fact that only representations trigger code generation (the models), we needed to generate code relative to the resources and, more importantly, code that would allow to perform iteration over resources and representations.

A typical assertion expression is shown in Listing 3.6, where the part concerning the state has quantifiers over resources and representations, and binds them through the `representationof` predicate. In order to be able to convert this expression into a valid Java expression, code generation encompasses the creation of (1) a `Resources` interface that provides methods to iterate over the various types of resources and (2) an interface for each resource type. To iterate over representations, since they are bound to resources, the resource interfaces are equipped with methods that allow to convert them to representations. A `RMCFactory` class, internal to the `Resources` interface, is exposed in order to be able to obtain the `Resources` implementation and be able to iterate over resources.

Examples for the aforementioned class/interfaces can be seen in Listings 3.4 and 3.5.

Listing 3.4: Example of a Resources Interface

```
1  public interface Resources {
2    /** Gets a stream of Pet resources */
3    Stream<Pet> getPetStream();
4    ...
5    public class RMCFactory{
6      /** The instance */
7      private static RMCFactory factory;
8
9      private RMCFactory() { }
10
11     /** @return the current instance */
12     public static RMCFactory getInstance() { ... }
13     /** @return the Resources managing class */
14     public Resources getResources() { ... }
15   }
16 }
```

Listing 3.5: Example of a Resource Interface

```
1  public interface Pet {
2    /** Gets Pet as its representation PetRep */
3    public PetRep getPetAsPetRep();
```

```
4     ...
5   }
```

Listing 3.6: Example of an expression used in a precondition

```
1   request in {body: PetRep} &&
2   (isdefined(request.body.id) =>
3     (forall pet:Pet .
4       (forall petRep:PetRep .
5         petRep representationof pet && petRep.id != request.body.id
6       )
7     )
8   )
```

The use of these classes is illustrated below with an example that shows the code generated from the expression presented in Listing 3.6, with a `forall` chain.

```
1   RMCFactory.getInstance().getResources().getPetStream().
2      allMatch(pet -> pet.getPetAsPetRep().getId() != body.getId())
```

In the previous code we iterate over all pet resources and assert if all of them comply with a given function, in this case, we convert the pet resource to its representation and compare the id from the request with the representation.

3.5 Resource logic

In order to be able to generate more code for the server side, namely code related to resources and their representations, the HEADREST language was extended with new elements. The elements added to HEADREST, which we designate *Resource Logic*, allow us not only to generate more code but also to simplify the writing of HEADREST specifications.

With the new elements included in the language: (1) we can declare predicates relative to resources, (2) we can bind resources to their representations with the `represents` expression and define resource predicates, (3) we can use the predicates on assertions. These new elements are materialised in the generated code as: (a) interfaces that represent resources are added predicate calls (b) representations have predicate call implementations (c) ability to call resource predicates as methods because of how they were added to the interface/class by the previous two points.

For instance, in the PetStore example, we can use the Resource Logic to define a predicate *hasId* over a Pet resource, that captures if a pet resource has a given id. We could then iterate over all pets and ask if all of them have a given id by using this predicate.

In Listing 3.7 we present an example that illustrates the elements that were added to the language. As we can see, the assertion was simplified (one less `forall` loop when

compared to the previous version, presented in Listing 3.6), which results in a more modular and, as a consequence, organised specification. This is also reflected in the generated code.

Listing 3.7: Example with Resource Logic in PetStore Example

```
resource Pet { // resource with logic
  pred hasId(integer)
}

type PetRep = { // representation
  id: integer
  ...
}

pr:PetRep represents Pet { // resource and representation bind
  hasId(arg0) => pr.id == arg0
}

{
  request in {body: PetRep} &&
  (isdefined(request.body.id) =>
    (forall pet:Pet .
      pet.hasId(request.body.id) // resource logic predicate usage
    )
  )
}
  POST /pet
{
  ...
}
```

The resulting code is then extended with this additional information:

- Resource interface is added the predicate declaration (see Listing 3.8)
- Representation class is added the predicate implementation (see Listing 3.9)

Listing 3.8: Resource interface with logic example

```
public interface Pet {
  ...
  /** Default hasid implementation */
  default boolean hasid(Integer id){
    return getPetAsPetRep().hasid(id);
  }
}
```

Listing 3.9: Representation class appended method example

```
public class PetRep {
  ...
  /** A defined predicate */
  public boolean hasid(Integer id){
    return this.id == id;
```

```
6     }
7   }
```

Making use of the elements expressed with the resource logic, in the generated code, instead of `(...).allMatch(pet -> pet.getPetAsPetRep().getId()!= body.getId())` we could simply have `(...).allMatch(pet -> !pet.hasid(body.getId())))`. Hence, we obtain code that is more abstract, modular and in the long run simpler, if various representations are used for the same resource.

3.6 HEADREST Codegen tool

As a proof of concept, the ideas presented before were implemented in a prototype tool. As shown in Figure 3.1, HEADREST Codegen has an encoder component that reads a HEADREST specification and encodes it into OAS, and a generator component that receives the generated OAS specification and generates client SDKs and server stubs, extending the Swagger Codegen for Java and generating for the RestEasy framework. The encoder and generator components will be described in detail in the following Chapters (4 and 5).

The figure illustrates the use of HEADREST Codegen in a very simple example. The figure shows:

- an excerpt of a HEADREST specification that is provided to the tool, which declares `ContactR` as a representation type for `Contact` resources and that specifies that the creation of resources of type `Contact` is possible through the execution of a POST operation over `/contacts` with a body request of type `ContactPutData`,

- a partial view of the structure of the client SDK and server stubs generated by the tool,

- an excerpt of the class that represents the representation type `ContactR`, equipped with a validation method that can be used to check whether an object of type `ContactR` satisfies the required properties

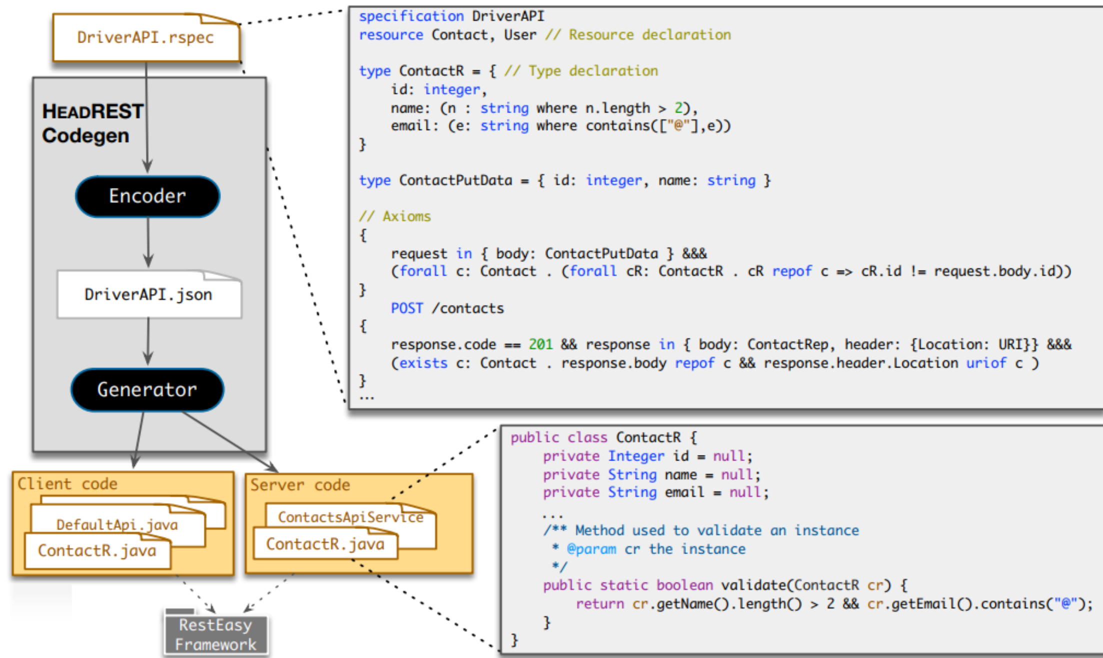


Figure 3.1: HEADREST Codegen in a nutshell

Chapter 4

Encoding HEADREST into OAS

In this chapter we discuss the encoding of HEADREST in OAS, a process that is accomplished by the HEADREST Codegen Encoder component. We cover the HEADREST analyser, used to read and analyse the specification, and the extension we made to the HEADREST language and its analyser. We then describe the structure of the encoder (from multiple views), the OAS generation, and end with the limitations of the encoding process.

4.1 HEADREST analyser

The HEADREST language analyser was developed in a previous work [7] making use of XText [2], a framework for developing programming languages and domain-specific languages. XText was used to define the syntactic and semantic analysis of HEADREST specifications. It provides us with many additional benefits, the major one being that it generates an Eclipse IDE for the language.

XText uses Eclipse Modelling Framework (EMF) models as the in-memory representation of any parsed file. In what follows we use *XText EMF AST* to designate the Abstract Syntax Trees (AST) of these models. This AST contains the essence of our specifications, abstracting from concrete syntax. It is used in later processing steps, including validation, compilation, and interpretation. In EMF a model is made up of instances of interconnected EObjects and contains various informations about the model. XText automatically validates the syntactic part of a HEADREST specification by generating a lexer and parser for a given language grammar.

In order to semantically validate HEADREST specifications, custom validators were added to the XText project that rely on Z3 SMT Theorem prover [3] More concretely, the process of semantic validation is accomplished by converting the HEADREST into a core intermediate representation AST, which we refer as *Core* HEADREST *AST*.

4.2 Core HEADREST

In Figures 4.1 and 4.2 we present in detail the grammar of the core part of HEADREST. This relies on a few base sets: that of *variables* denoted by x, y, z, that of *resource types* denoted by α, β, and that of *labels* denoted by l,m. Integer literals are denoted by n, string literals by s, and regular expressions by r. Regular expression literals conform to ECMA-262. URI templates conform to RCF-6570 [11]. In an object of the form $\{l_1 \colon e_1, \ldots l_n \colon e_n\}$, labels $l_1, \ldots, l_n$ must be pairwise distinct.

Multi-field record types are broken down into an empty record type $\{\}$ and a singleton record type $\{l \colon T\}$, cf. [5, 13]. URITemplate is a primitive type. In order to ease language implementation, it was made primitive, and not, say, $(x \colon \mathsf{String}\ \mathsf{where}\ \mathsf{matches}(r_{\text{urit}}, x))$ for an appropriate regular expression r_{urit}.

HEADREST specifications, rule S, are constituted by any number of variable declarations, type declarations, resource declarations and assertions.

Variable declarations start with the keyword var binding a variable to a type and allow us to have free variables to carry information from preconditions to postconditions.

Type declarations start with the keyword type and allow us to create types that are identified by a label name. They are the data structures that are exchanged back and forth in the various interactions. Rules B, G, and T describe the supported types, ranging from primitive types such as Integer to more complex types such as an array of positive integers, $(x \colon integer\ \mathsf{where}\ x > 0)\,[]$.

Resource declarations start with the keyword resource and allow us to declare *resource types*, which are used in assertions to describe the current state of resources.

Finally, assertions are structures that allow us to specify preconditions and postconditions over operations. The pre/postconditions are written as expressions, described by the first rule e. A typical precondition starts by describing the request structure (the *body*, *path* or *query* argument types) followed by state related expressions, such as resource and representation quantification (last two expression of rule e). A typical postcondition starts by describing the response code, followed by the response structure (the *body*) and ending with state related expressions, as described in the precondition. An assertion example can be seen in Chapter 2 (see Listing 2.7)

Signatures for some primitive functions are presented in Figure 4.2.

In the following, we detail the behaviour of a few operators central to HEADREST. Resource related operators representationof and uriof assert whether a given element is a representation or identifier of a resource, respectively. The first is typically used in assertions, when iterating over resources and representations, to associate a representation with a resource and then evaluate an expression over the representation (see example in Chapter 3, Listing 3.6).

Operator expand converts an URITemplate and an object into a String. The function works according to RFC 6570 [11] when all template variables show up in the object,

$$
\begin{array}{llcl}
\text{Expression} & e & ::= & x \mid c \mid \oplus(e_1, \ldots, e_n) \mid e_1 \mathbin{?} e_2 : e_3 \mid e \,\mathsf{in}\, T \\
 & & & \mid \{l_1 \colon e_1, \ldots l_n \colon e_n\} \mid e.l \mid [e_1, \ldots, e_n] \mid e[e] \\
 & & & \mid \forall x \colon T.e \mid \exists x \colon T.e \\
\text{Scalar constant} & c & ::= & n \mid s \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{null} \mid F \mid r \\
\text{URI template} & F & ::= & \epsilon \mid sF \mid \{l\}F \mid \{?l\}F \\
\text{Regex} & r & ::= & \ldots \\
\text{Type} & T & ::= & \mathsf{Any} \mid G \mid \{l \colon T\} \mid (x \colon T \,\mathsf{where}\, e) \mid \alpha \mid \{\} \mid T\,[] \\
\text{Basic type} & G & ::= & B \mid \mathsf{Regexp} \mid \mathsf{URITemplate} \\
\text{Primitive type} & B & ::= & \mathsf{Integer} \mid \mathsf{String} \mid \mathsf{Boolean} \\
\text{Verb} & a & ::= & \mathsf{get} \mid \mathsf{put} \mid \mathsf{post} \mid \mathsf{delete} \\
\text{Specification} & S & ::= & \epsilon \mid \mathsf{var}\, x \colon T; S \mid \mathsf{type}\, t = T; S \mid \mathsf{resource}\, \alpha; S \mid \{e\}ae\{e\}; S \\
\text{Context} & \Gamma & ::= & \epsilon \mid \Gamma, x \colon T \\
\text{Resource} & \Delta & ::= & \epsilon \mid \Delta, \alpha
\end{array}
$$

Figure 4.1: HEADREST syntax

otherwise it leaves the variables as are. This justifies the fact the range of the function is a String and not an URI.

The inductive definition of the expand function is as follows.

$$
\begin{array}{rll}
\mathsf{expand}(\epsilon, v) & \mapsto \mathsf{root} & \\
\mathsf{expand}(sF, v) & \mapsto s +\!\!+ \mathsf{expand}(F, v) & \\
\mathsf{expand}(\{l\}F, v) & \mapsto \mathsf{tostring}(u) +\!\!+ \mathsf{expand}(F, v) & \text{if } v.\textit{template}.l = u \\
\mathsf{expand}(\{l\}F, v) & \mapsto \texttt{"}\{l\}\texttt{"} +\!\!+ \mathsf{expand}(F, v) & \text{otherwise} \\
\mathsf{expand}(\{?l\}F, v) & \mapsto \texttt{"}?l = \texttt{"} +\!\!+ \mathsf{tostring}(u) +\!\!+ \mathsf{expand}(F, v) & \text{if } v.\textit{template}.l = u \\
\mathsf{expand}(\{?l\}F, v) & \mapsto \texttt{"}\{?\}\texttt{"} +\!\!+ \mathsf{expand}(F, v) & \text{otherwise}
\end{array}
$$

It should be clear that expand yields a string that may not represent an URI. For example $\mathsf{expand}(\textit{item}\{id\}, \{\}) \mapsto$ `"item{id}"`, which is not a proper URI. This allows expand to remain a total operator, hence may be used in types.

Predicate isdefined queries whether a given field is present in an object. The predicate is defined by induction on its parameter as follows.

$$
\begin{array}{rll}
\mathsf{isdefined}(e.l) & \mapsto \mathsf{isdefined}(e) \mathbin{?} e \,\mathsf{in}\, \{l \colon \mathsf{Any}\} : \mathsf{false} & \\
\mathsf{isdefined}(e) & \mapsto \mathsf{true} & \text{otherwise}
\end{array}
$$

The matches predicate returns true if a given regular expression is matched by a given String, false otherwise.

$$\begin{aligned}
\text{representationof} &: \text{Any}, \alpha \to \text{Boolean} \\
\text{uriof} &: \text{String}, \alpha \to \text{Boolean} \\
\text{expand} &: \text{URITemplate}, \{\textit{template}\colon \{\}\} \to \text{String} \\
\text{isdefined} &: \text{Any} \to \text{Boolean} \\
\text{matches} &: \text{Regexp}, \text{String} \to \text{Boolean} \\
\text{tostring} &: \text{Any} \to \text{String} \\
\text{length} &: \text{Any}\,[] \to \text{Natural} \\
\text{size} &: \text{String} \to \text{Natural}
\end{aligned}$$

Figure 4.2: Signatures of some primitive operators

4.2.1 Derived syntax

The concrete syntax counts with a few extensions, all obtained by translation into the core language. We distinguish derived expressions from derived types.

Figure 4.3 and 4.4 present some *derived types* and *derived expressions*, respectively. Most of these abbreviations are from [3]. In the definition of multi-field object types, $\star$ denotes ? or empty. Type URI abbreviates strings generated by a regular expression e_{uri} defined according to RFC 3986 [1]. The symbol "!" symbolizes negation.

The first two types, $[e\colon T]$ and $[e]$ are singletons and they allow us to describe value instanced types. As an example: $[\text{“}a\text{”}]$ describes a $String$ type with value “a”.

The $|$ operator allows the creation of a sum type – sometimes called union or choice type, to name a few –. For instance: Integer | Boolean is a type that is either an Integer or a Boolean. A more interesting example would be: $[\text{“}a\text{”}] \mid [\text{“}b\text{”}] \mid [\text{“}c\text{”}]$ which is a String that is either the string literal “a”, “b” or “c”, an enumeration.

The $\&$ operator, intersection type, is analogous to the $|$ operator but intersects types instead. For instance: $(x\colon integer \text{ where } x > 0)\ \&\ (y\colon integer \text{ where } y < 10)$ will result in $(z\colon \text{Any where } (z \text{ in } (x\colon integer \text{ where } x > 0)\ \&\&\ z \text{ in } (y\colon integer \text{ where } y < 10)))$ which could be written simply as $(z\colon integer \text{ where } z > 0\ \&\&\ z < 10)$.

The $\&\&\&$ expression is typically used in assertions so that the information in e can be correctly assumed in the following expression f.

Finally, the $\implies$ is the logical implication operator. A typical usage scenario is to assert something relative to an optional field, if the field is present, (see example in Chapter 3, Listing 3.6).

4.3 HEADREST extension

The extension to HEADREST was designed to be backward compatible so that through abstraction and translation we could still obtain a valid and meaningful specification in

$$
\begin{array}{rcll}
[e\colon T] & \triangleq & (x\colon T \text{ where } x{==}e) & x \notin \text{fv}(e) \\
[e] & \triangleq & [e\colon \mathsf{Any}] & \\
T \mid U & \triangleq & (x\colon \mathsf{Any} \text{ where } (x \text{ in } T \mid\mid x \text{ in } U)) & x \notin \text{fv}(T, U) \\
T \;\&\; U & \triangleq & (x\colon \mathsf{Any} \text{ where } (x \text{ in } T \;\&\&\; x \text{ in } U)) & x \notin \text{fv}(T, U) \\
!T & \triangleq & (x\colon \mathsf{Any} \text{ where } !(x \text{ in } T)) & x \notin \text{fv}(T) \\
T \text{ where } e & \triangleq & (x\colon T \text{ where } e) & x \notin \text{fv}(e) \\
\{?l : T\} & \triangleq & (x\colon \mathsf{Any} \text{ where } x \text{ in } \{l\colon \mathsf{Any}\} \Rightarrow x \text{ in } \{l\colon T\}) & x \notin \text{fv}(T) \\
\{\star l_1\colon T_1, \ldots, \star l_n\colon T_n\} & \triangleq & \{\star l_1\colon T_1\}\& \ldots \&\{\star l_n\colon T_n\} & n \geq 1 \\
\text{if } e \text{ then } T \text{ else } U & \triangleq & (x\colon T \text{ where } e) \mid (x\colon U \text{ where } !e) & x \notin \text{fv}(T, U) \\
\mathsf{URI} & \triangleq & (x\colon \mathsf{String} \text{ where } \mathsf{matches}(r_{\text{uri}}, x)) & \\
\mathsf{Natural} & \triangleq & (x\colon \mathsf{Integer} \text{ where } x \geq 0) & \\
\mathsf{Empty} & \triangleq & (x\colon \mathsf{Any} \text{ where false}) &
\end{array}
$$

Figure 4.3: Some derived types

$$
\begin{array}{rcl}
e \;\&\&\&\; f & \triangleq & (e \,?\, f : \mathsf{false}) \\
e \implies f & \triangleq & (e \,?\, f : \mathsf{true})
\end{array}
$$

Figure 4.4: Some derived expressions

the original language. This also made possible to recycle the existing semantic validation.

As mentioned before we extended the HEADREST language with resource logic. Concretely, this was accomplished by adding new elements to the language: declaration of resource predicates, binding of representation with resource including a definition of the resource predicates and use of these predicates (see Listing 3.7 for an example). The extension are described in Figure 4.5.

$$
\begin{array}{llcl}
\text{Specification} & S & ::= & R; S \\
\text{Represents} & R & ::= & \epsilon \mid R, x : T \text{ represents } \alpha\{D_0, ..., D_n\} \\
\text{Predicate definition} & D & ::= & m(x_0, ..., x_n) => e \\
\text{Expression} & e & ::= & \alpha.m(e_0, ..., e_n) \\
\text{Resource} & \Delta & ::= & \Delta, \alpha\{P_0, ..., P_n\} \\
\text{Predicate declaration} & P & ::= & \text{pred } m(T_0, ..., T_n)
\end{array}
$$

(extends Figure 4.1)

Figure 4.5: HEADREST syntax extension

In order to take advantage of the core validation, predicate call expressions are derived, hence converted in core HEADREST. This conversion is accomplished by following the rule, where $e[e'/x]$ is the result of replacing free occurrences of variable x by expression e' in expression e. Given the following section in a HEADREST specification:

resource α {pred m ($\vec{T_m}$, ...)}

$x_0 : T_0$ represents $\alpha\{m(\vec{y_0}) => e_0, ...\}$
...
$x_n : T_n$ represents $\alpha\{m(\vec{y_n}) => e_n, ...\}$

then:

$$
\begin{aligned}
\alpha.m(\vec{e}) \triangleq\ & \forall x_0 : T_0 \text{ representationof } \alpha => e_0[\vec{e}/\vec{y_0}] \\
& \&\&...\&\& \\
& \forall x_n : T_n \text{ representationof } \alpha => e_n[\vec{e}/\vec{y_n}]
\end{aligned}
$$

An example of this rule in action can be seen in Chapter 3 (see Section 3.5 where we use resource logic to simplify Listing 3.6 to Listing 3.7).

In the end of the encoding phase we append the HEADREST specification to the generated OAS, via its extension mechanisms. In this way, we ensure that all information in the original specification (even the parts not used in the encoding) is available.

4.4 Encoder

In order to create the HEADREST Codegen, our code generation prototype, we had to pick an AST to work with for the encoding process, the XText EMF AST or the Core HEADREST AST.

A way of encoding HEADREST into OAS would be to use the XText EMF AST obtained directly from reading a HEADREST specification. This is much closer to the OAS format, but we would be forced to process it for semantic validity, which is already done by the analyser, and also we would need to cover many specification cases.

The other way, which we decided to use, is using the Core HEADREST AST since it is canonical and, in this way, our encoder has to deal with much less possible specification cases, making it simpler and easier to extend. Additionally, we can request the analyser to run semantic validations directly. In the end we convert it back to XText EMF AST to get the closest possible to the OAS structure, apply some simplifications and we can start processing it into a OAS specification.

4.4.1 Behaviour view

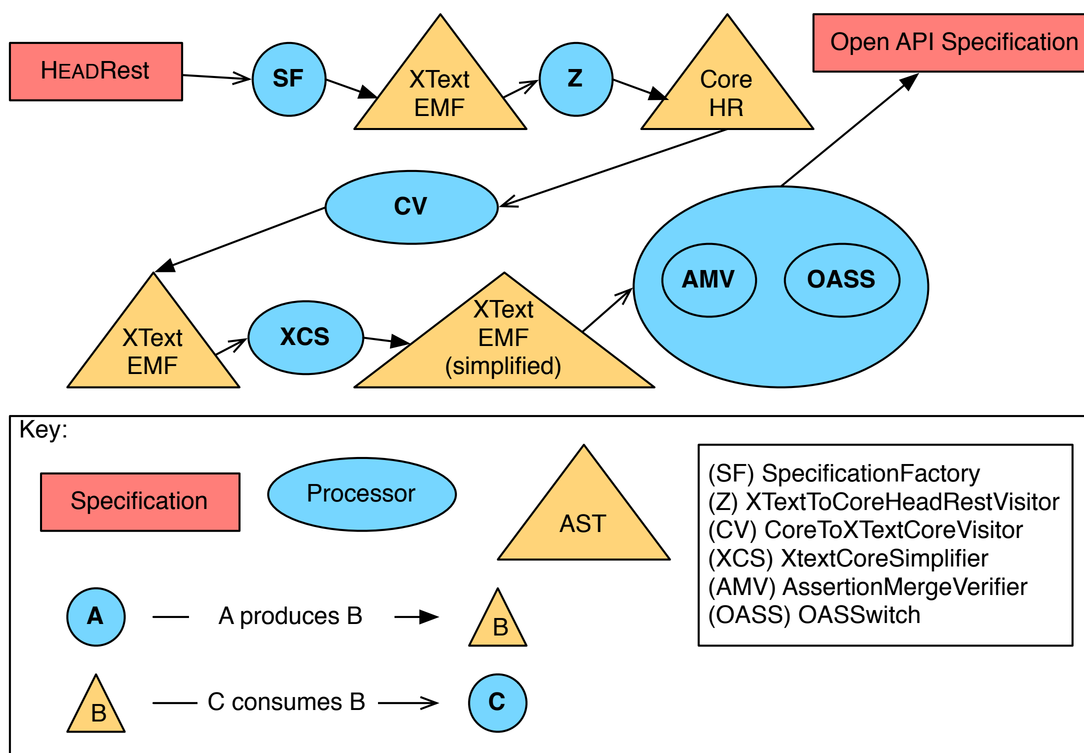


Figure 4.6: Encoder behaviour view

In Figure 4.6 we present a runtime view of the structure of the encoder component while in Figure 4.7 we provide an illustration of the transition between the various ASTs. With this example we intend to provide an better idea about the various ASTs used in the process and a better insight on the decisions we made, namely the AST we chose to encode HEADREST from and the processing/simplification done through them.

In our specification the user can use either the intersection type or use the core element version. When translated to Core HeadREST both versions will be converted to the core element (we go from XText EMF to an internal AST format, at this point). After that we convert it back to XText applying the inverse rules of the ones presented in Figure 4.3. Since this was an intersection type of refinement types we simplify it. Finally, we generate the OAS specification in a straightforward way.

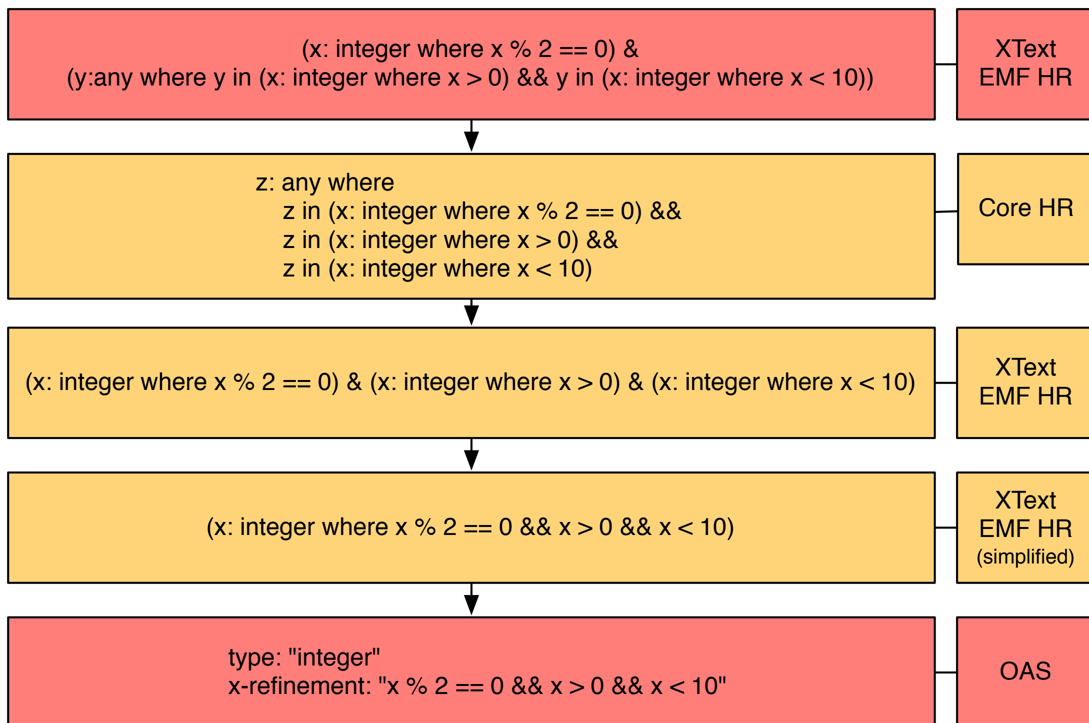


Figure 4.7: Textual AST transition view

SpecificationFactory (generated by XText) is responsible for reading a specification, creating the XText EMF AST and converting it to the Core HeadREST AST. It also runs the semantics analysis if needed.

CoreToXTextCoreVisitor converts the Core HeadREST AST back into the XText EMF AST.

XTextCoreSimplifier is responsible for adapting the XText EMF AST for encoding into Open API Specification. This includes:

- adding the URI type declaration (which is a basic type in HeadREST, but not supported by OAS),

- replacing response codes constants by their actual values,

- applying type intersection rules to resolve intersection types to a single type (in order to be able to encode into OAS),

- extracting singletons and singleton unions.

  Details about these simplifications are presented in Section 4.5.1.

AssertionMergeVerifier, like the name implies, verifies that assertions can be merged later in the encoding (recall that OAS has limitations relative to the operations – e.g., only one request type for success). It consists in evaluating the merging of request and response types, and that all the required arguments are defined across assertions relative to the same operation.

OASSwitch is responsible for parsing the prepared XText EMF AST and generate an object of type `io.swagger.models.Swagger`, that abstracts the OAS specification.

The assertions require a special treatment because there can only be a definition per response code per path, while in HEADREST we can have multiple assertions for the same operation. This requires the organisation of information and merging of assertions.

To extract the single request/response type to add to our specification, when we have multiple assertions relative to the same operation, we gather all the possible request/response types and create a sum type.

OASSwitch also applies certain simplifications, for instance, when we find a singleton (e.g., `["someString"]`), we create a string type and use the *enum* field provided natively by OAS, that way we force the type to only be that specific value. When we have type disjunctions of singletons, the same rule applies, after converting the type we add all the possible values to the *enum* field.

4.4.2 Module View

In Figure 4.8 we present a top level view of the encoder that shows the package structure of the encoder and how the different classes use each other. The encoder is mainly comprised of various visitors that manipulate, verify and generate from ASTs. Some utility classes were developed, as well as helpers and printers, whose role is as described in what follows:

Catalog

This subsection is dedicated to classes that were not described previously.

- Encoder class is the entry point to the encoding process.

- StringUtils class has, as the name suggests, methods to help with string manipulation,

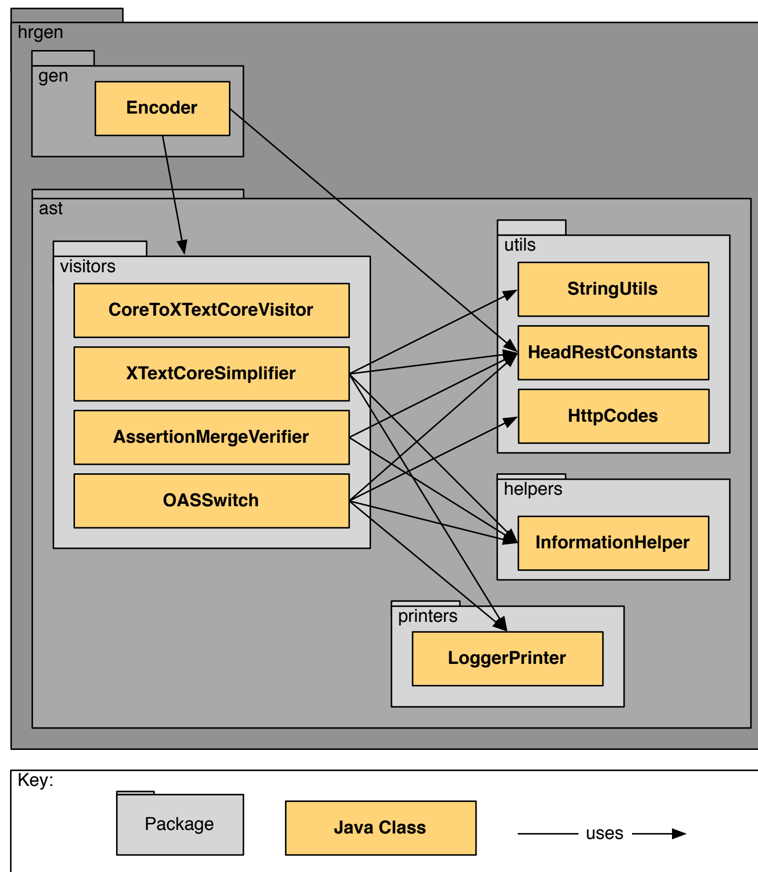


Figure 4.8: Encoder top level uses view

- HeadRestConstants class isolates all the constants that are supposed to be easy to change and that are required in the classes and template files used in the encoding and generation process classes and template files,

- HttpCodes class loads information relative to HTTP codes from a CSV file obtained from IANA.[1] This information is then used to complete the encoded file,

- InformationHelper class provides various methods to help with extracting useful information from the HeadREST XText AST, these include request/response type extraction, conjunction/disjunction flattening methods, and many more,

- LoggerPrinter class prints EObjects to a logger, for error logging messages.

[1] `https://www.iana.org`

4.5 OAS generation

In this section we go over the process of encoding HEADREST (a simplified XText EMF AST) into OAS. We cover the encoding of basic types, refinement types, derived types and assertions. We present the rules that the encoder follows to generate the OAS specifications from HEADREST specifications (a compiled version of the rules presented in this section can be seen in Appendix A).

4.5.1 Type encoding to OAS

Encoding HEADREST's types is challenging, mainly because we want to make use of what OAS can express natively as much as possible. This allows to take advantage of the existing Swagger Codegen. In order to illustrate the encoding rules, we took advantage of the HEADREST language definition.

Basic types

The types currently supported by HEADREST Codegen are `integer`, `boolean`, `string`, `URI`, $\{l_0 : t_0, ..., l_n : t_n\}$, and $T[\,]$. We consider `URI` and multi-property objects as basic types, even though they are derived types, because we can obtain and use them almost directly. These are fairly easy to convert into OAS core language because it supports basic types. An example that illustrates this is presented below. The basic type:

```
type Person = {id: integer, name: string}
```

is encoded in OAS as:

```
Person:
  type: "object"
  required:
  -"id"
  -"string"
  properties:
    id:
      type: "integer"
    name
      type: "string"
```

HEADREST `URI` type is not supported by OAS. To address this, as mentioned before we create a refinement type of `string` and add it to the HEADREST specification. Usages of type `URI` simply become references to this new type. Below we present an example of a HEADREST type definition that uses an `URI` type and the resulting encoding in OAS. Note that label `$ref` is used in OAS to reference a named type.
The HEADREST type:

```
type UsingUri = {link: Uri}
```

is encoded in OAS as:

```
1 UsingUri:
2   type: "object"
3   required:
```

```
4     - "link"
5     properties:
6       link:
7         $ref: "#/definitions/Uri"
```

where `"#/definitions/Uri"` (line 7) is defined in the definitions section of the OAS specification as follows:

```
Uri:
  type: "string"
  x-refinement: "matches(UriRegex, uri)"
```

In what follows we present the rules that define the type encoding of HEADREST types into OAS. Function [[.]] receives a type T (as in Figures 4.1 and 4.3) and returns an OAS specification.

$$
\begin{aligned}
[\![\ B\ ]\!] &= \text{type: ``}B\text{''} \\
[\![\ URI\ ]\!] &= \text{\$ref: ``\#/definitions/Uri''} \\
[\![\ \alpha\ ]\!] &= \text{\$ref: ``\#/definitions/}\alpha\text{''} \\
[\![\ \{l_0 : T_0\ , ...,\ l_k : T_k\ , ...,\ ?l_{k+1} : T_{k+1}\ , ...,\ ?l_n : Tn\}\ ]\!] &= \text{type: ``object''} \\
&\quad \text{required:} \\
&\quad \text{- ``}l_0\text{''} \\
&\quad \ldots \\
&\quad \text{- ``}l_k\text{''} \\
&\quad \text{properties:} \\
&\quad \text{- } l_0 : [\![\ T_0\ ]\!] \\
&\quad \ldots \\
&\quad \text{- } l_n : [\![\ T_n\ ]\!] \\
[\![\ T\ [\,]\ ]\!] &= \text{type: ``array''} \\
&\quad \text{items: } [\![\ T\ ]\!]
\end{aligned}
$$

Refinement types

Refinements of the basic types have the refinement expression carried on the type via OAS extension mechanisms. These mechanisms are provided in the form of extra tags that can be appended to any location of the document, to carry extra information. In OAS, any tag that begins with `x` is considered an extension and is ignored by OAS-specific tools. Some examples of refinement types are presented below. In these examples, we first present the HEADREST definition and then the resulting encoding. First, we refine a digit and then an object that contains a `string` field to match a regular expression.

The encoding of the digit:

```
type Digit = (d:integer where d >= 0 && d < 10)
```

results in the OAS type:

```
Digit:
  type: "integer"
  x-refinement: "x >= 0 && x < 10"
```

The encoding of the object:

```
type MyObject = (y: {b: string} where matches(/aaa|bbb|ccc/ ,y.b))
```

results in the OAS type:

```
MyObject:
  type: "object"
  required:
  - "b"
  properties:
    b:
      type: "string"
      x-refinement: "matches(/aaa|bbb|ccc/ ,y.b)"
```

The rule that allows us to encode refinement types into OAS is the following:

$$[\![\, (x\colon T \text{ where } e) \,]\!] = [\![\, T \,]\!]$$
$$\text{x-refinement: ``}e\text{''}$$

Derived types

Type intersection is the most complex type we support. We have defined some rules that attempt to encode as many intersections as possible. In order to explain the complexity of intersection types, let us start by considering an intersection of two object types with different non-optional fields.
The intersection:

```
type Intersection = {a: integer} & {b: string}
```

is encoded into:

```
1  Intersection:
2    type: "object"
3    required:
4    - "a"
5    - "b"
6    properties:
7      a:
8        type: "integer"
9      b:
10       type: "string"
```

From the encoding of the previous example, notice that the required parameters created a `required` entry (lines 4 and 5). Also, notice that object intersection results in field conjunction.

HEADREST allows optional object fields as shown below. Having a mandatory field with the same name of an optional one results in the mandatory field creating a `required` entry. However, the types must be intersectable regardless of being optional or not.

```
type Intersection = {a: integer} & {?a:integer, ?b: string}
```

The resulting encoding for this example is the same as for the previous, except the `b` field does not create a `required` entry.

In what follows we show the rules that enable us to intersect types ($\star$ denotes ? or empty).

$$
\begin{aligned}
[[\, G \,\&\, ... \,\&\, G \,]] &= [[\, G \,]] \\
[[\, \alpha \,\&\, ... \,\&\, \alpha \,]] &= [[\, \alpha \,]] \\
[[\, (x\colon T \text{ where } e_1) \,\&\, T \,]] &= [[\, (x\colon T \text{ where } e_1) \,]] \\
[[\, (x\colon T_0 \text{ where } e_0) \,\&\, (x\colon T_1 \text{ where } e_1) \,]] &= [[\, (x\colon T_0 \,\&\, T_1 \text{ where } e_0 \,\&\&\, e_1) \,]] \\
[[\, \{l_0 : T_0, ..., l_n : T_n\} \,\&\, \{\star l_x, T_x\} \,]] &= [[\, \{l_0 : T_0, ..., l_n : T_n, \star l_x : T_x\} \,]] \\
&\quad \textit{where } \; l_x \notin \{l_0, ..., l_n\} \\
[[\, \{l_0 : T_0, ..., l_k : T_k, ..., l_n : T_n\} \,\&\, \{\star l_k : U\} \,]] &= [[\, \{l_0 : T_0, ..., l_k : (T_k \,\&\, U), ..., l_n : T_n\} \,]]
\end{aligned}
$$

Other derived types that need some specific processing are: singleton and union of singletons. These are encoded quite easily into OAS with the following rules:

$$
\begin{aligned}
[[\, [e : G] \,]] &= [[\, G \,]] \\
&\quad \text{enum:} \\
&\quad \text{- } e
\end{aligned}
$$

$$
\begin{aligned}
[[\, [e_o : G] \mid ... \mid [e_x : G] \,]] &= [[\, G \,]] \\
&\quad \text{enum:} \\
&\quad \text{- } e_0 \\
&\quad \dots \\
&\quad \text{- } e_x
\end{aligned}
$$

Union types are not supported beyond singletons due to the fact that their translation requires the use of OAS extension mechanisms to preserve type semantics. Union types will be discussed in the following subsection, as we make use of it to merge request and response types for operations.

4.5.2 Assertions

Encoding HEADREST assertions is even more challenging than encoding types. This is because OAS requires a single request per operation, unlike HEADREST that supports various assertions for the same operation.

The solution we found merges the various request/response types for each successful operation code. With this, we force the request and response types of assertions relative

to the same operation to be mergeable, where request types are extracted from success code assertions only (we explained previously that OAS only enables the specification of one request type, the success case). More concretely, the types need to be mergeable via union (denoted by the operator " | "), for which we have certain rules that we present later in this section.

Assertion base encoding rule

$[[\ \{e_0\}\ a\ /p\ \{e'_0\}\ \ ...\ \ \{e_n\}\ a\ /p\ \{e'_n\}\ ]] =$

```
/p:
  a:
    parameters:
      -in: "body"
       name: "body"
       required: true
       schema: |0≤i≤k[[Ti]]
    responses:
      c0:
        description: <code detail extracted from IANA csv file>
        schema: |0≤j≤n[[Tj]] where response.code == code0
      ...
      cn:
        description: <code detail extracted from IANA csv file>
        schema: |0≤j≤n[[Tj]] where response.code == coden
    x-axioms:
    - "{e0} a /p {e'0}"
    - ...
    - "{en} a /p {e'n}"
```

where:

- T_i and T_j are extracted from the respective e_i.request.*body* in T_i (from preconditions) and e'_j.response.*body* in T_j (from postconditions) expressions
- c_0,...,c_n are extracted from the postconditions `response.code == c_i` expressions,
- we assume that assertions from 0 to k refer to successful cases, i.e., response codes in the interval [200,300[, while the remainder assertions have different codes,
- IANA stands for Internet Assigned Numbers Authority, which provides a list of HTTP codes and related information (e.g., code 404 has description "Not Found").

The previous example was the encoding of simple assertions that only restricted the body parameter of an operation. Things get complicated when we have assertions that specify path and query parameters.

In HEADREST a path (URI Template) can take the form `/path{a,b,?c,d}`. This means `a` and `b` are path parameters, and `c` and `d` are query parameters. Because OAS requires that all parameters have an associated type, our encode inherits this limitation. If a path or query parameter is present, then the request must be refined with an object that gives that parameter a type. Additionally, this must be verified for all preconditions of the assertions relative to the same operation and a successful situation.

In this way, we cannot encode the following HEADREST specification, as it indicates a query parameter `status` that does not have any associated type.

```
{
  true
}
  GET /pet/findByStatus{?status}
{...}
```

However, we can encode the following specification, as it defines the type of the query parameter.

```
1 {
2   request in {template: {status: integer}}
3 }
4   GET /pet/findByStatus{?status}
5 {...}
```

We can observe in the previous listing (line 2) that some processing is required in order to extract the query path type (located in `request.template.status`).

As a result of validating the specification (via the AssertionMergeVerifier class) we extract the request type directly. The validation takes care of 1) identifying path and query parameters, 2) checking that they are defined in all assertions relative to the same operation, and 3) checking that their types are mergeable.

The addition of path and query parameters is analogous to that of the body, modifying the `-in` tag to the corresponding parameter type.

Type merging

The merging of types, used to merge operation request and response types, is done via a type union, following these rules:

$$
\begin{aligned}
[[\, G \mid ... \mid G \,]] &= [[\, G \,]] \\
[[\, \alpha \mid ... \mid \alpha \,]] &= [[\, \alpha \,]] \\
[[\, (\; x_0 : G \text{ where } e_0) \mid ... \mid (x_n : G \text{ where } e_n) \,]] &= [[\, (x : G \text{ where } e_0 \mid\mid ... \mid\mid e_n) \,]] \\
[[\, \mid_i \{\vec{a_i} : \vec{T_i}, ?\vec{b_i} : \vec{T_i}\} \,]] &= [[\, (x : \{?\vec{a_i} : \vec{T_i}, ..., ?\vec{a_n} : \vec{T_n}, ?\vec{b_i} : \vec{T_i}\} \\
&\qquad where \;\; \mid\mid_i isdefined(\vec{a_i})) \,]]
\end{aligned}
$$

4.6 Limitations

In this section we present some limitations of the encoder. It is advised to take them into consideration for best results with the HEADREST Codegen tool.

Union of objects with optional parameters Type disjunction has a restriction relative to optional parameters. If an optional parameter appears on an object, then it must appear

on the other objects as well. For a certain `Verb Path` the request body and the response body must both declare the same optional parameters. Let us explain why with a simple example.

Suppose that for a given `Verb Path` there are two success request types: (1) $\{a : T_1, ?b : T_1\}$ and (2) $\{c : T_1\}$. Valid instances of the resulting sum type $\{?a : T_1, ?b : T_1, ?c : T_1\}$ *where* $isdefined(a) \parallel isdefined(c)$ are: $\{a : T_1\}$, $\{a : T_1, b : T_1\}$, $\{c : T_1\}$, and $\{c : T_1, b : T_2\}$.

Almost every case is good, except for the last case. When we add the request type to the swagger object we are indicating that there is an optional field b of type $T1$, so the last case would be excluded as it would not match the generated method signature. Because of this, we decided that optional types are required to be present in all the objects in the disjunction, coherently.

So $\{a : T_1, ?b : T_1\}|\{c : T_1\}$ would turn into $\{a : T_1, ?b : T_1\}|\{c : T_1, ?b : T_1\}$, excluding this last problematic instance while not really being very restrictive on the user.

Type any The type **any** in HEADREST corresponds to Object in Java. Because OAS does not support **any** it requires that a type is defined, as such **any** is not currently supported.

null value As of OAS version 2.0 (the version we are using) **null** is not supported and therefore we decided not to support it.

Sum types As mentioned before when adding assertions to the Swagger object, if there are many assertions relative to the same `Verb Path`, the request and response body types are converted to a unique sum type. This is the only moment where type union is supported, because we use Swagger's extension mechanisms to attempt to conserve the semantics of the created type.

At type definition level only one sum type is supported, between singletons. Singletons are converted by the encoder into an enumerated type.

If we were to support sum types at type definition level, we would have to limit that type to only be used on a singular assertion (an assertion that is the only assertion for a given `Verb Path`), so that merging of two sum types would not happen. Merging two arbitrary is a complex operation, due to our approach to object type union.

Refinement types on parameters The use of refinement types in body/path/query parameters requires the type to be defined externally. For instance:

```
1 {
2   request in {body: (x:integer where x > 0)}
3 }
4   POST /something
5 {
```

```
6    response.code == 200
7  }
```

The previous axiom specification generates a method signature that receives an integer body as argument, but its refinement cannot be lost. If we extract the type however we can refine it correctly. It would look something like:

```
1  type bodyT = (x:integer where x > 0)
2
3  {
4    request in {body: bodyT}
5  }
6    POST /something
7  {
8    response.code == 200
9  }
```

The latter would generate a `BodyT.validate(body)` method call, which verifies the refinement.

Type negation Type negation is not supported as there are an infinite number of types and OAS does not support negation.

Type negation with string singletons The type negation:

```
1  type ReservedWords = ["word1"] | ["word2"]
2  {
3    !(request.template.word in ReservedWords)
4  } ... { true }
```

can however be worked around as an external refined type:

```
1  type NotReservedWords = (s: string where !matches(/word1|word2/ , s))
2  {
3    request.template.word in NotReservedWords
4  } ... { true }
```

Success via type negation HEADREST allows stating that if a call with a body of any type different from integer will trigger a response with code 200. For generation however we don't support this.

Unlimited Variables We consider unlimited variables, those that do not have a value associated on the moment of a variable usage.

HEADREST allows declaring variables via the **var** construct. These variables can be used in all sorts of ways, but most commonly to associate a variable in a precondition to be the same as in the postcondition. Code generation can only use variables that have been associated via the **uriof** or **representationof** predicates.

- x **uriof** e associates a resource variable x with an **URI** e
- x **representationof** R associates a representation variable x to a resource R

For cases where a variable has not been associated, the current assertion encoding will be replaced by the string literal `true /*list of unbound variables followed by commented out expression*/`.

Chapter 5

Generating code from HEADREST specifications

Once the encoding of HEADREST in OAS is ready, we can proceed with code generation. This is achieved by a component that extends the Swagger Codegen, taking advantage of the extra information that we encoded into the generated OAS specification.

Swagger Codegen is available for many programming languages and REST frameworks. For HEADREST Codegen we picked the version for the JAX-RS framework, as it is popular for Java, is easy to install and deploy, and integrates well with our tools. More concretely, we use the RestEasy [4] implementation of the JAX-RS.

In this chapter we start by discussing the most important decisions underlying HEADREST Codegen in what concerns the code to be generated. Then, the generation component of HEADREST Codegen is presented, with its most important elements, the generation process, ending with its limitations.

5.1 The code produced by the generator

In what follows we justify our decisions on what concerns the code generated from HEADREST specifications, capitalising on the code generated by Swagger Codegen. As before, we use the function $[[.]]$, that receives an expression e (as in Figures 4.1 and 4.3) and returns the corresponding Java expression, and $\star l$ to denote $?l$ or l in object types.

5.1.1 Models

Models that were encoded from refinement types have the refining expression appended via an x-tag. To take advantage of this information, it was decided to extend the generated model classes with a validation method. There are many different cases that need to be handled in a specific way, namely because of how basic, object, and array types generate different sorts of Java classes.

In what follows, we present the different cases that were considered. We use $\mathcal{C}$ to denote the set of classes generated by HEADREST Codegen.

- Case type declarations of the form `type RB = (x:B where e)`. $\mathcal{C}$ includes a class `public class RB`, generated by Swagger Codegen, that includes a single method:

  ```
  public static boolean validate(B instance){return [[ e ]];}
  ```

- Case `type RO = (x:T where e)`, a refinement of an object type T. $\mathcal{C}$ includes a class `public class RO`, generated by Swagger Codegen, that includes a single method:

  ```
  public static boolean validate(RO instance){return [[ e ]];}
  ```

- Case `type RA = (x:T[] where e)`, a refinement of an array type $T[\,]$. $\mathcal{C}$ includes a class `public class RA extends ArrayList<T>`, generated by Swagger Codegen, that includes with a single method:

  ```
  public static boolean validate(RA x){return [[ e ]];}
  ```

- Case `type A = (x:B where e)[ ]`. $\mathcal{C}$ includes a class `public class A extends ArrayList <B>`, generated by Swagger Codegen, that includes with a single method:

  ```
  public static boolean validate(A instance){
    return instance.stream().allMatch(elem -> [[ e ]]);
  }
  ```

- Case `type A =` $R[\,]$, where R is the name of a non-basic refinement type. $\mathcal{C}$ includes a class `public class A extends ArrayList<R>`, generated by Swagger Codegen, that includes a single method:

  ```
  public static boolean validate(A instance){
    return instance.stream().allMatch(elem -> R.validate(elem));
  }
  ```

- Case `type O =` $\{\star l_{rb_0} : B'_0, ..., \star l_{rb_i} : B'_i, ..., \star l_{rx_0} : X_0, ..., \star l_{rx_j} : X_j, ..., \star l_{r_0} : R_0, ..., \star l_{r_k} : R_k, ..., \star l_{t_0} : T_0, ..., \star l_{t_n} : T_n\}$, where $B'_0...B'_i$ are refinements of basic types (in the form `(x:B where` e_b`)` where `B` is a basic type and e_b is a refinement expression), X are either object or array types that comprise refinement types, R are refinements of non-basic types, T are non-refined types (where objects and arrays do not comprise refinement types). $\mathcal{C}$ includes a class `public class O`, generated by Swagger Codegen, that includes a single method:

  ```
  public static boolean validate(O instance){
    return [[ B'_0.e_b ]] && ... && [[ B'_i.e_b ]] &&
        X_0 .validate( l_rx_0) && ... && X_j.validate( l_rx_j) &&
        R_0 .validate( l_r_0) && ... && R_k .validate( l_r_k );
  }
  ```

Recall that examples of generated validation methods were already presented, for instance, in Listing 3.2. Another example of a generated validation method can be found in Appendix E.3, generated for the `ContactRep` type specified in Appendix D.2.

5.1.2 Assertions

Operations that were encoded from assertions have the assertion expressions appended via an x-tag. To take advantage of this information, it was decided to extend the generated API classes (e.g., `DefaultApi` client class).

Clients

The client-side extension is translated in methods in the `DefaultApi` generated class (generated by Swagger Codegen). These methods allow the user to verify whether the interaction satisfies the conditions for a successful call to the operation. An example of a client side operation validation method is presented in Listing 3.3.

- Let $\{e_i\}\ a\ /p\ \{e'_i\}$, for $1 \leq i \leq n \leq m$, be a set of assertions relative to a given operation $a\ /p$. Suppose also that, assertions that refer to successful cases are those for $1 \leq i \leq n$.

  Considering that $[\![\ e\ ]\!]$ for clients only translates part of the precondition expression e, the part that is not state-related (recall that state properties are not verifiable client-side). $\mathcal{C}$ includes a class **public class** `DefaultApi`, generated by Swagger Codegen, that includes a method for each operation (the method name prefix `op` denotes the name associated by Swagger Codegen to the generated operation — e.g., the **POST** operation over the `/pet` path results in an operation named `petPost`):

```
public boolean opRequiredSuccessPrecondition(){
  boolean pre1 = [[ e1 ]];
  ...
  boolean pren = [[ en ]];

  return pre1 || ... || pren;
}
```

Servers

The server extension is translated in code that validates the preconditions and postconditions in the various `ApiImpl` class methods generated by Swagger Codegen. Example of a server side operation validation method can be seen in Chapter 3 (see Listing 3.3.2).

- Let $\{e_i\}\ a\ /p\ \{e'_i\}$, for $1 \leq i \leq n \leq m$, be a set of assertions relative to the operation $a\ /p$. Suppose also that, assertions that refer to successful cases are those for $1 \leq i \leq n$.

$\mathcal{C}$ includes a class `public class MApiServiceImpl` (the prefix `M` denotes the name associated by Swagger Codegen to a group of operations), generated by Swagger Codegen, that has its operation methods logic (returning a response stub) completed with precondition and postcondition evaluations (`N` denotes the name associated by Swagger Codegen to the generated operation):

```
public Response N(...){
  boolean pre1 = [[ e_1 ]];
  ...
  boolean prem = [[ e_m ]];

  final Response response = Response.ok().entity(...).build();

  boolean pos1 = [[ e'_1 ]];
  ...
  boolean posm = [[ e'_m ]];

  if (pre1) assert pos1;
  ...
  if (prem) assert posm;

  return response;
}
```

Assertions have state related expressions (e.g., resource quantification) that need to be converted into Java code. Extra classes and interfaces are added to accomplish this, namely a predicates handling class, a Resources interface and interfaces for each resource.

Predicates

Class Predicates covers the various predicate that HEADREST provides (e.g., `matches`, `contains`, etc - See Figure 4.2 for more examples). This class is generated equally for all projects projects, except that package names change. The current Predicates class implementation can be seen in Appendix B.

Resources

Introduced in Chapter 3 (see Listing 3.4), the Resources interface allows generating expressions from resource quantifiers. We decided to take advantage of the Java 8 Stream API[17] to iterate over the resources, using *anyMatch* and *allMatch* methods are what we need for our quantification expressions.

- Let α be a resource and assume that `uriof` expressions are used in some assertion to associate an URI to these resources, of the form: `resource` $\alpha_0, \ldots, \alpha_n$

  $\mathcal{C}$ includes an interface `public interface Resources`, generated by HEADREST Codegen, that has resource stream getter methods appended for each resource declared in the HEADREST specification, has resource getters by URI extracted from

each `uriof`, and an internal class `public class RMCFactory` that provides a way of obtaining the `Resources` implementation:

```
Stream<α0> getα0Stream();
...
Stream<αn> getαnStream();

α0 getResourceα0(String uri);
...
αn getResourceαn(String uri);

public class RMCFactory{
/** The instance */
private static RMCFactory factory;

private RMCFactory() { }

/** @return the current instance */
public static RMCFactory getInstance() {
if(factory == null)
factory = new RMCFactory();
return factory;
}

/** @return the Resources managing class */
public Resources getResources() {
  throw new UnsupportedOperationException("RMCFactory.getResources() not
      implemented. Please replace stub to return an object of type
      Resources."); //TODO
}
```

Resource interfaces

In order to talk about resources and iterate over them, we generate interfaces for every resource. These interfaces abstract the various resources and provide us with a way of refering to them. An example of resource interfaces can be seen in Chapter 3 (see Listing 3.5). If resource logic is used, the generated resource interfaces and models, from types that represent resources, can be augmented with more code. An example of a resource interface with logic can be seen in Chapter 3 (see Listing 3.8).

- Suppose that the resource logic of a HEADREST specification includes the following definitions concerning the resource α, where R is an alias for T:

```
resource α {
  pred m_0(T_0, ..., T_i),
  ...,
  pred m_m(T_0, ..., T_j)
}

x_0 : R_0 represents α {
  m_0(arg_0, ..., arg_i) => e_{0_0},
  ...,
  m_m(arg_0, ..., arg_j) => e_{0_m}
}
...
x_n : R_n represents α {
  m_0(arg_0, ..., arg_i) => e_{n_0},
  ...,
  m_m(arg_0, ..., arg_j) => e_{n_m}
}
```

$\mathcal{C}$ includes an interface `public interface` α, generated by HEADREST Codegen, that has resource to representation conversion methods, extracted from each `representationof` used in the specification, has default methods appended for every resource logic predicate declared, and is appended a `hasUri` method to verify if a given URI identifies the resource:

```
public R_0 getαAsR_0();
...
public R_n getαAsR_n();

default boolean m_0(T_0 arg_0, ..., T_i arg_i){
  return getαAsR_0().m_0(arg_0, ...,arg_i) &&...&&
      getαAsR_n().m_m(arg_0, ...,arg_i);
}
...
default boolean m_n(T_0 arg_0, ..., T_j arg_j){
  return getαAsR_0().m_n(arg_0, ...,arg_j) &&...&&
      getαAsR_n().m_m(arg_0, ...,arg_j);
}

public boolean hasUri(String uri);
```

Moreover, for each $1 \leq i \leq n$, $\mathcal{C}$ includes a class `public class` R_i, generated by Swagger Codegen, that is extended with resource logic methods, where $\triangleright e$ denotes the translation of e into a Java expression:

```
public boolean m_0(T_0 arg_0, ...,T_i arg_i){
  return [[ e_{i_0} ]];
}
...
public boolean m_m(T_0 arg_0, ...,T_j arg_j){
  return [[ e_{i_m} ]];
}
```

The representations (i.e., types) that represent resources are rewritten, appending the new predicate methods and their implementations. An example of a representation that represents a resource can be seen in Chapter 3 (see Listing 3.9).

Translation of HEADREST expressions to Boolean Java expressions

The translation of expressions is accomplished with a set of mapping rules from HEADREST to Java expressions (because we are generating Java projects). Below, we present some examples of the more interesting translation rules. We use the function $[\![.]\!]$, that receives an expression e (an in Figures 4.1 and 4.3) and returns the corresponding Java expression.

- [[`forall x:integer .` e]] =
  `IntStream.rangeClosed(MIN, MAX).allMatch(`x `->` [[e]]`)`
- [[`exists x:integer .` e]] =
  `IntStream.rangeClosed(MIN, MAX).anyMatch(`x `->` [[e]]`)`
- [[`forall x:(y:integer where` r`) .` e]] =
  `IntStream.rangeClosed(`$\vee$`(MIN,`r_1`),` $\wedge$`(MAX,`r_2`)).filter(`r_3`).allMatch(`x`->`[[e]]`)`
  where
  – `MAX` and `MIN` stand for `Integer.MAX_VALUE` and `Integer.MIN_VALUE`, respectively
  – $\vee$ and $\wedge$ are functions that return the maximum and minimum of the arguments, respectively
  – r_1 is the integer extracted from processing r lower bound constraints
  – r_2 is the integer extracted from processing r upper bound constraints
  – r_3 are the remaining expressions.
- [[`forall r:`R `.` e]] =
  `RMCFactory.getInstance().getResources().get`R`Stream().allMatch(`r `->` [[e]]`)`
  where
  – R is a resource type

Example of translation for the third rule:
[[`forall x:(y:integer where y > 0 && y < 10 && y % 2 == 0). e`]] =
`IntStream.rangeClosed(0, 10).filter( y -> y % 2 ).allMatch(`x `->` [[e]]`)`

5.1.3 Module views

In what follows we present various views of the code that is generated by our tool, client SDKs and server stubs.

Client module view

In Figure 5.1 we present a top level uses view of the generated client. The client is provided with a DefaultApi class, various Model classes, the ApiClient class and a Predicates class. The DefaultApi class represents the entrypoint to consume the API and it expects

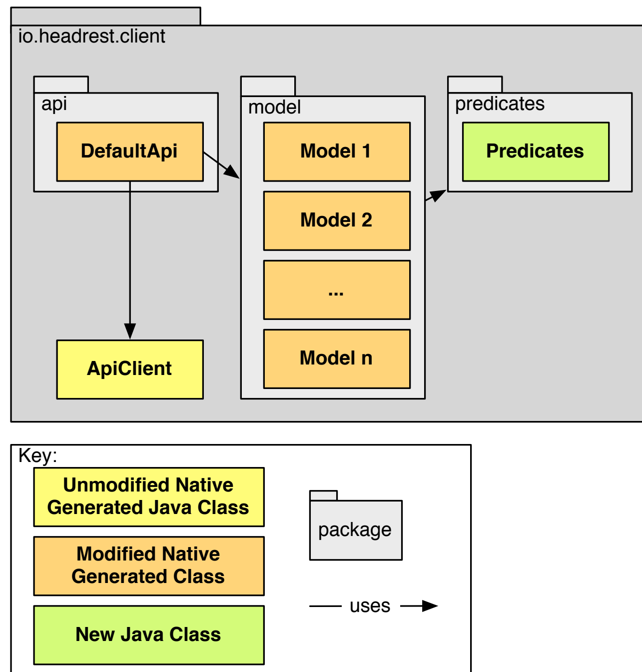


Figure 5.1: Generated client top level uses view (minor classes ommited for simplicity)

Models as its arguments, making use of the ApiClient to generate the HTTP requests. The Predicates class is used to provide predicates used when models are refinement types.

The DefaultApi class comes from the Swagger Codegen but is equiped with new methods that enable the client to verify if any success precondition are (partially) met, so to enable clients to send back error messages before communicating with the server. The models come native as well, but have validation methods added when they are specified as refined types.

DefaultApi contains various methods that represent the operations and allow the consumption of the API (e.g., `public PetRep petPost(PetRep body)`).

ApiClient is an adapter that converts high level calls from the DefaultAPI into low level HTTP requests, making use of the JAX-RS Client implementation.

Model classes are generated with various fields, with their setters and getters, and, if refined, are appended with a validation method.

Server module view

In Figure 5.2 we can see a top level uses view of the generated server. The server is provided with an Api class which utilizes the various JAX-RS annotations to capture HTTP requests and then delegates them to implementations of the ApiService, which has stubs created under the `io.headrest.api.impl` package. The `ApiServiceImpl` stub was extended to have precondition and postcondition verifications and the user should make

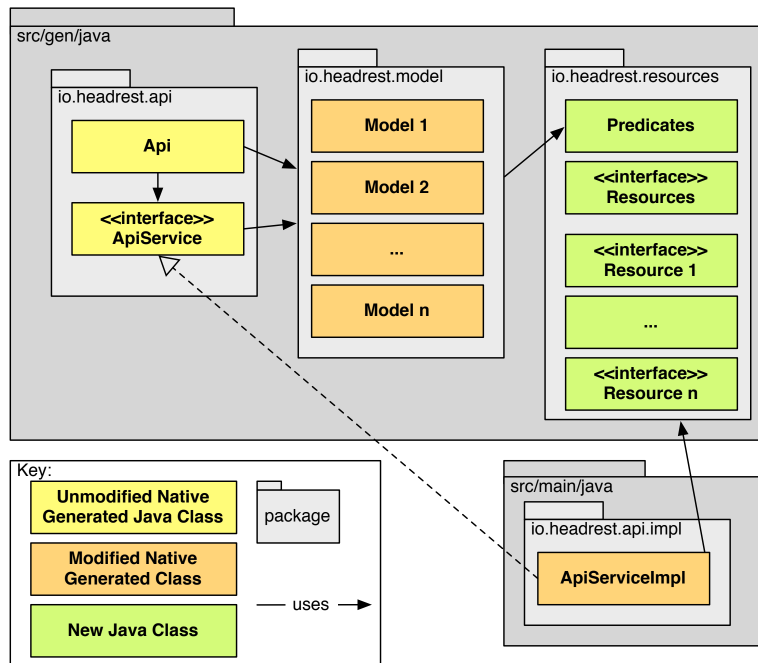


Figure 5.2: Generated server top level uses view (minor classes ommited for simplicity)

use of the extra supporting files present in *io.headrest.resources*, provided by the extended generator, to manipulate the various resources. Similar to the client, the same models are provided to serve as representations used in the requests and responses.

5.1.4 Components/Deployment view

In Figure 5.3 we can see a very simplified hybrid component and deployment view of the client SDK and server stubs that we generate with HEADREST Codegen. Clients may interact with the server with our client SDK or, for a developer to debug, a standalone REST client. The server was tested to work on both GlassFish [1] and WildFly [2] containers.

5.2 The code generation process

HEADREST Codegen encompasses a wide variety of classes that further manipulate the OAS specification, generate files and support the whole generation process. In what follows we go over the structure of the generator, providing illustrations and describing the different components involved.

[1] `https://javaee.github.io/glassfish` (Seen: 2018-05-19).
[2] `http://www.wildfly.org` (Seen: 2018-05-19).

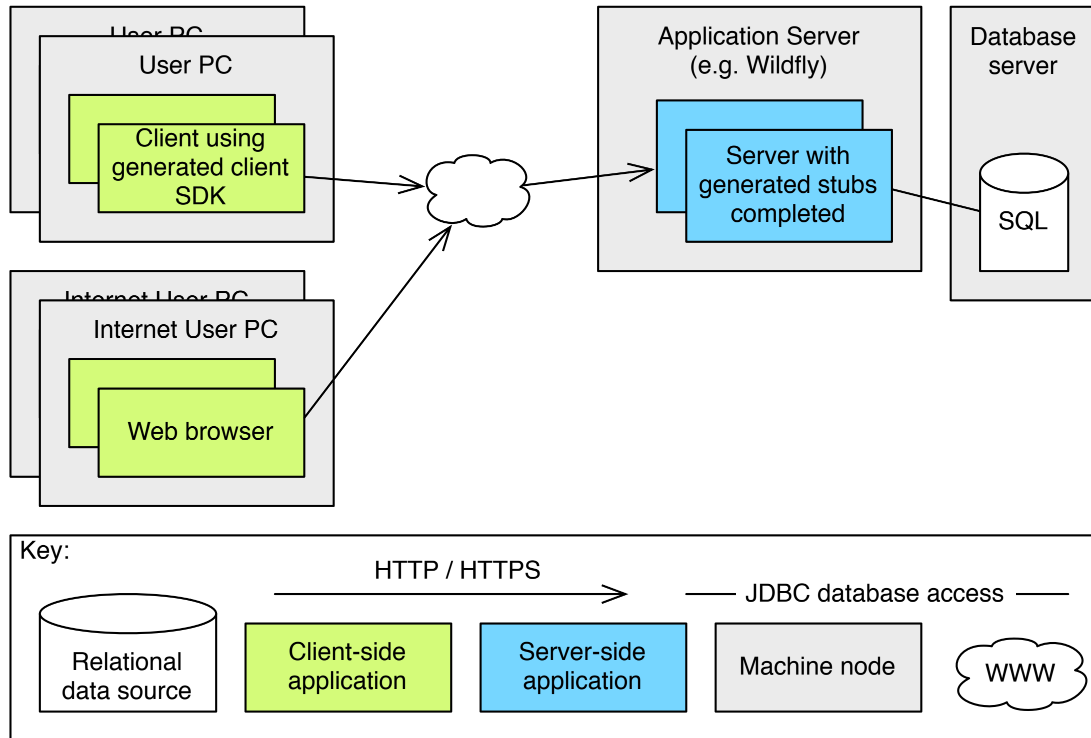


Figure 5.3: Hybrid components and deployment view

5.2.1 Behaviour view

In Figure 5.4 we present a behaviour view of the generator, showing the runtime interactions of various components that compose the generator. As mentioned before, at the end of the encoding phase we insert the HEADREST specification inside the OAS so that we can later retrieve and use it. This allows us to generate the extra resource logic information.

In order to compile the Mustache templates we start by converting the XText EMF expressions into valid Java expressions, while also retrieving some useful extra information to be used by our generator. Secondly, we run the specification through a processing class that appends more extension information to the OAS specification, to help with the templates. In the end we provide the specification and all additional information to the generator and have it compile the templates.

In Figure 5.5 we present a top level view of the module structure of the generator to better understand the package structure and how the different classes use each other.

Generator is the entrypoint class to the generation process, receiving directly the Swagger object created from the Encoder. In the end it copy pastes the *Predicates* class with some minor manipulations.

SwaggerXRefinementConverterToJava is the core element of the generator, as it

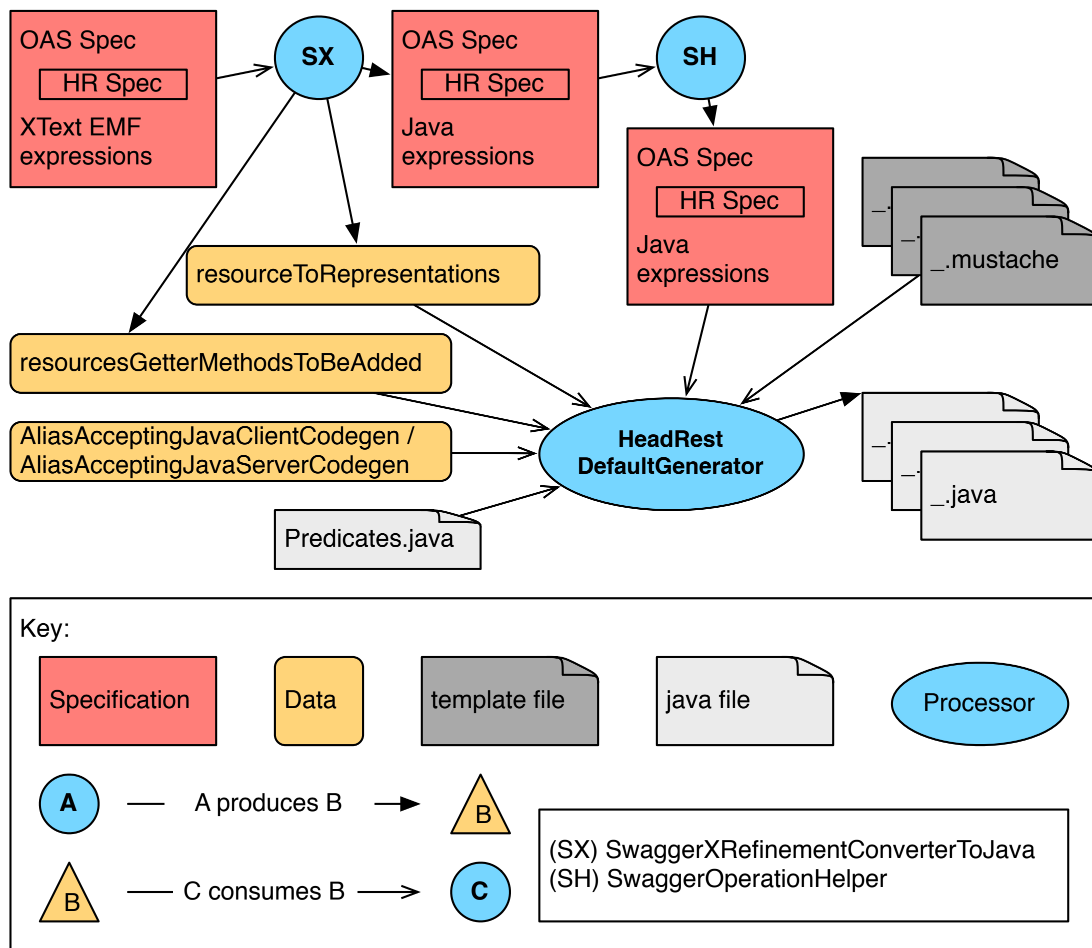


Figure 5.4: Generator behaviour view

processes the encoded file x-tags expressions into valid Java expressions, as well as extract extra information.

SwaggerOperationHelper runs through the encoded file and helps prepare operations for the addition of validation calls.

HeadRestExpressionToJavaVisitor is a visitor class that enables the aforementioned conversion extending the RestSpecificationLanguageSwitch provided by the HEADREST analyser.

AliasAccepter, AliasAcceptingJavaClientCodegen and AliasAcceptingJavaResteasyServerCodegen classes provide a way of accepting aliases of Java native types.

Predicates class is copy pasted into the generated client and server, with minor package name changes. It implements the various HEADREST predicates used in expressions.

Configurator class provides extra configuration for the generator, mainly pom[3] manipulating methods.

[3]Project Object Model from Maven.

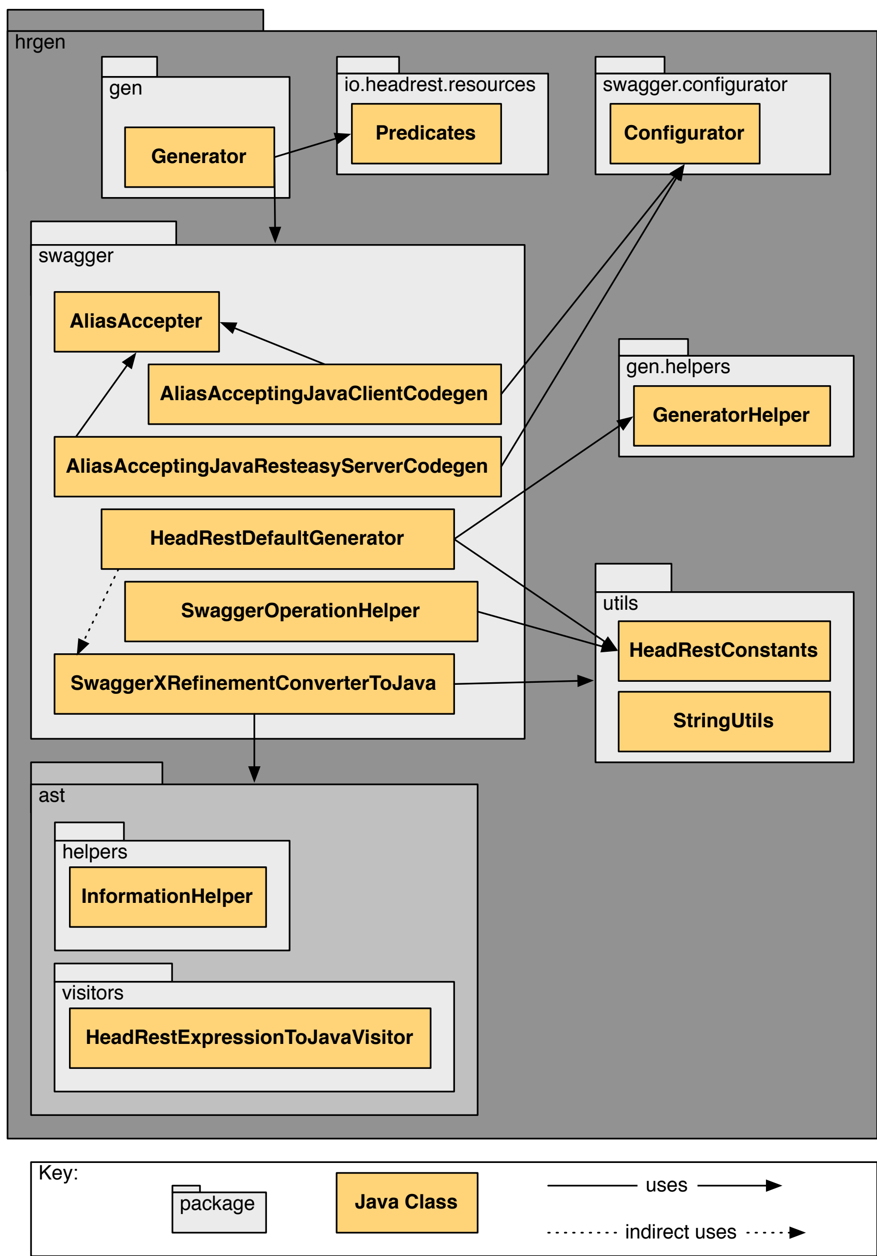


Figure 5.5: Generator top level uses view

HeadRestDefaultGenerator is our extension of the Swagger Codegen generation class (*io.swagger.codegen.DefaultGenerator*). We override the *generate* method and delegate the extra file generation.

GeneratorHelper is a delegate class for helping with the generation of extra supporting files. It is provided, upon construction, with the extracted HEADREST specification file, by the HeadRestDefaultGenerator class. Remember that this file was appended in a OAS extension tag at the end of the encoding phase.

In what follows, we go over the most relevant classes and methods, for the generation

process.

SwaggerXRefinementConverterToJava

This class converts all the expressions (assertion pre/postconditions and refinement types expressions) into Java expressions (because of Mustache). This conversion is accomplished by the HeadRestExpressionToJavaVisitor which, not only converts the expression, but also provides us with a bundle of information afterwards:

- resource to representation conversion methods to be added to each resource interface, extracted from every `representationof` expression
- resource getter methods to be added, to get resources by URI, extracted from every `uriof` expression

The first is relative to resources being represented by representations, requiring them to have a corresponding conversion method, so that when we talk about a representation we can obtain it from the resource (a method such as *getResourceAsRepresentation* is generated), since representations only make sense when bound to a resource. The last is relative to obtaining specific resources by URI, which needs to be added to the Resources interface (class that provides access to all resources).

SwaggerOperationHelper

This class prepares the Swagger object for generation, mainly by creating a list of all the refined arguments of an operation and adding it to the operation via the extension mechanisms, this is then used to facilitate the template processing.

Generator.*generateRestEasyClient/Server* methods

This method creates a codegen configuration object which has been extended to:

- Change packages, maven pom file data
- Change the *toBooleanGetter* method
- Post process models for basic type alias acceptance

The packages and maven pom file data had to be changed in order to be coherent with the project and the *toBooleanGetter* method had to be changed to make it coherent with other getters (avoids bugs). Finally, the post processing of models to accept basic type alias is necessary because of basic type refinement, that will be explained now.

Say we have `type MyInteger = (x: integer where x > 10)`, this is a refinement of an integer, which is translated natively by the Swagger Codegen as a class that extends Integer (which is impossible) and as such every mention to this class is actually exchanged by

the actual Integer java class. To accomplish this HEADREST Codegen extends Swagger Codegen to enable aliases. This will enable us to have the class MyInteger created, rather than discarded because of being an alias - remember that the refinement must be carried forward.

In the end this method calls the *generate* method, explained in what follows.

HeadRestDefaultGenerator.*generate* method

This method is the heart of the generation extension and it runs an extension of the *DefaultGenerator.generate* method (a Swagger native class and method). The method was overwritten in order to: (1) add our HeadRestInlineModelResolver which helps propagate the x-tags (which Swagger Codegen by default does not) and (2) generate extra files.

To try and maintain the original code as much as possible we use Java Reflection [10]. Extensions made to the *generate* method include:

- Usage of the HeadRestInlineModelResolver
- Creation of a GeneratorHelper
- Generating extra supporting files
- Generating new definition models

HeadRestInlineModelResolver

The InlineModelResolver is a class that processes OAS to generate model classes. It attempts to convert properties to models (properties are basically anonymous models, declared on the fly in the specification, instead of being defined in the *definitions* section). HeadRestInlineModelResolver is an InlineModelResolver extension that propagates vendor extensions [4] when converting property to model, which is not natively done by the original. This avoids vendor extensions loss, which would result in validation methods being incorrect (e.g., empty arguments).

GeneratorHelper

GeneratorHelper is a delegate class for generating of extra supporting files. It generates resource files and new definition models (if Resource Logic is used). In more concrete terms, it generates the Resources interface , resources interfaces and modifies representation classes (models).

[4]The programatic name for the OAS extension mechanism.

Generating extra supporting files

Done via the GeneratorHelper.*generateExtraSupportingFiles* method call, it generates the resources interfaces and Resources interface , for accessing the various resources. For every resource an interface is created in which resource logic methods are added. Additionally a Resources interface is created that provides access to iterators over the various resources. For iterators we use streams, to take advantage of the *allMatch* and *anyMatch* methods, which map directly with our `forall` and `exists` quantifier expressions.

Generating new definition models

Done via the GeneratorHelper.*generateNewDefinitionModels* method call, it deals with models that represent a resource, adding the resource logic methods and their implementation. This is done by reading the generated model file and appending the new lines of code.

5.3 Limitations

In this section we present a collection of limitations relative to the generator. It is advised to take them into consideration for best results with the tool.

!isdefined(request.body) The generated method signature for each operation of an API is defined via the success case. With this a problem rises, which is the use of `!isdefined(request.body)` to define an error case without having defined a success case. This means that the generated precondition code will attempt to verify the predicate but there will not be any *body* argument available causing a compilation error. This is analogous for other parameters and to responses, for instance `request.template.somePathParameter` or `response.body`.

Usage of underscores throughout specification The fact that Swagger Codegen has its own internal automatic name handling makes it impossible for the extension generated code to be compatible with the native Swagger generated code if such automatic naming is used. Using good Java code practices, mainly in naming conventions, is advised for better results.

Usage of reserved keywords throughout specification Usage of reserved keywords might generate code that does not compile. Please refrain from doing so. Reserved words include OAS, HEADREST and Java reserved keywords.

Extra generated models In Subsection 5.1.1 we presented the rules that translate objects and arrays. Whenever Swagger Codegen finds the schema of an object or array associated to a property, it generates a corresponding model class. Despite the fact that the class carries the extension tags we add, we cannot generate validation code for the "extracted model" properties, due to the fact that the class name is internally generated by Swagger Codegen. Efforts can be made to obtain the name generation algorithm and append the validation method calls.

5.4 Good practices

To obtain the best generation results users should invest in good practices.

The limitations

As mentioned throughout this document, both the encoder and generator carry certain limitations, attempting to not take them into account might cause both phases to stop with an exception or worse, generate incorrect code.

The semantics

We recommend running the semantic validation to make sure the specification is as correct as possible. A correct specification is more likely to be able to generate from.

The syntactics

HEADREST is capable of very complex types, expressions and assertions. Try to keep it simple, as it will make the specification file much more readable by users, while also helping avoid encoding/generation bugs.

Chapter 6

Evaluation

This chapter addresses the evaluation of the proposed techniques for code generation as well as the code that they generate. We start by briefly presenting the various APIs that were used for evaluating the extensions of the language and the generation techniques. Then, a comparison between the code generated from Swagger codegen and HEADREST Codegen is presented considering mainly the usefulness of the generated code for developers of clients and providers of RESTful web services. We also compare the code generated by HEADREST Codegen from HEADREST specifications that use/do not use the resource logic. In order to complement this comparison, uniquely rooted in intellectual arguments, we also present a small empirical study that was conducted with developers and that is focused on the generated client SDK.

6.1 Case Studies

Different case studies of RESTful APIs were considered for evaluating the developed extensions of the language and the generation techniques. This encompassed the development of HEADREST specifications and the subsequent application of the HEADREST Codegen to these specifications. In what follows, we briefly describe these case studies, highlighting their most important characteristics.

- **Petstore** is a RESTful API often used to illustrate documentation of RESTful APIs, namely OAS.[1] It is a simple RESTful API of a service that supports the management of pets, stores and users. It relies mainly on basic types and has a simple logic, so being able to generate code for this case study means that we are be able to support simple APIs. We specified part of the Petstore API with HEADREST and subject it to HEADREST Codegen and later we used this example also to exercise the different elements of the resource logic added to HEADREST and validated the process of code generation for these new elements.

[1] `http://petstore.swagger.io` (Seen: 2018-05-19)

- **Features** is a more complex RESTful API of a service available at `https://github.com/JavierMF/features-service` that manages features models of software product lines and the configurations of concrete products. This API is built around a larger number of resources — products, their configurations as well as their features, feature configurations, and constraints, and its logic is more intricate as attested by the long assertions of their specification (some with more than 20 lines). For instance, the following assertion relative to a **POST** operation of a *constraint* resource (line 2) specifies that the precondition is met then the operation will be sucessful (line 2 where CREATED is code 201) and that, like the HTTP standard for code 201, it will return the URI of the created resource (line 3). In the following expression (lines 6-15) the creation of the constraint resource and binding of the URI returned by the response `Location` header to the resource is done (lines 6-7), followed by the association with its representation (lines 8-9). In the remaining (lines 10-15) many data assertions are made due to the fact that fields might be optional, hence the usage of the `isdefined` predicate. The final expression (lines 16-26) does a similar assertion, but relative to the `ProductR` resource, which includes the contraint (this inclusion is also denoted by the path URI of the operation in line 2).

```
1  { ... }
2    POST /products/{productName}/constraints/excludes{?sourceFeature,
        excludedFeature}
3  {
4  response.code == CREATED &&
5  response in {header: {Location: URI}} &&& (
6    (exists constraintR: ConstraintR.
7      response.header.Location uriof constraintR &&
8      (exists constraint: Constraint .
9        constraint representationof constraintR &&
10       constraint.type  == "excludes" &&
11       isdefined(constraint.excludedFeatureName) &&& (
12         ((isdefined(request.template.sourceFeature) ==> constraint.
              sourceFeatureName  == request.template.sourceFeature)) &&
13         ((!isdefined(request.template.sourceFeature) ==> constraint.
              sourceFeatureName  == null)) &&
14         ((isdefined(request.template.excludedFeature) ==> constraint.
              excludedFeatureName  == request.template.excludedFeature))
               &&
15         ((!isdefined(request.template.excludedFeature) ==> constraint
              .excludedFeatureName  == null))
16   ))) &&
17   (exists product: Product. product representationof productR &&
18     (exists i: (x: integer where x >= 0 && x < length(product.
          constraints)).
19       product.constraints[i].type  == "excludes" &&
20       ((isdefined(request.template.sourceFeature) ==> product.
            constraints[i].sourceFeatureName  == request.template.
            sourceFeature)) &&
21       ((!isdefined(request.template.sourceFeature) ==> product.
            constraints[i].sourceFeatureName  == null)) &&
```

```
22        ((isdefined(request.template.excludedFeature) ==> product.
             constraints[i].excludedFeatureName  == request.template.
             excludedFeature)) &&
23        ((!isdefined(request.template.excludedFeature) ==> product.
             constraints[i].excludedFeatureName  == null))
24    ))
25  )
26  }
```

While the specification does not use complex types, their long and complex assertion expressions were a challenge for HEADREST Codegen. Although some expressions use variables that are not bound, which results in the expression being commented out, the fact that we can generate a partially working service is very interesting, as the user would only need to manually work on the commented-out lines of code or, on the other hand, modify the specification.

- **GitLab** is a RESTful API of a Git-repository manager. During the development of HEADREST, an impressive part of the GitLab API was specified with HEADREST. We revisited this specification so that we were able to generate code from it. The challenge posed by this specification to HEADREST Codegen was its large object types with multiple optional fields and their equally enormous assertions. Generation for this specification was successful, despite its verbosity.

- **Contacts** is a RESTful API of a very simple contacts management service. This API offers basic interactions to obtain the current list of contacts, and add or update contacts. It was the API used in the experiment study. Its HEADREST specification has sophisticated refined types over objects relating different internal fields.

These case studies are publicly available at `http://rss.di.fc.ul.pt/tools/confident/`.

6.2 Swagger Codegen vs HEADREST Codegen

In this section we present a comparison between the code generated from Swagger codegen and HEADREST Codegen, considering mainly the usefulness of the generated code for developers of clients and providers of RESTful web services.

As mentioned before, HEADREST adopts many concepts of OAS but extends them with refinement types and logical assertions, that provide richer types and brings behavioural specification of the operations. OAS provides certain type refinements using specific fields (e.g., for an integer type we can add the fields *minimum* and *maximum* to refine an integer's minimum and maximum value, respectively) which are then verified via Java Bean Validation (JBV), while HEADREST Codegen generates a validation method with the refining expression. Since we are in the context of a RestEasy server,

JBV will require the programmer to manually create a validator and validate the data in the operation method, while HEADREST Codegen already generates calls to our validation methods. As a novelty, we can further refine the types with predicates and imposes contraints over the values of different object properties (e.g., `type relatingObject = (x: {a:integer, b:integer} where x.a > x.b)`).

In the listings below we show the class that represents an object with an URI type field in OAS and the same in HEADREST. In OAS we could refine a string with the *@Pattern* annotation, but remember that we would still need to create a validator to verify it.

```
public class ClassThatUsesUri {
  @Pattern(regexp = URI_REGEX)
  private String uri = null;
  ...
}
```

HEADREST Codegen appends the URI type in the encoding phase, as a result we create the `Uri` class, that has its own validation method.

```
public class Uri {
  ...
  public static boolean validate(String instance) {
    return Predicates.matches(instance, URI_REGEX);
  }
}
```

The `ClassThatUsesUri` class, being its `uri` field a refinement type, has a validation method added, as shown below.

```
public class ClassThatUsesUri {
  private Uri uri = null;
  ...
  public static boolean validate(ClassThatUsesUri instance) {
    return Uri.validate(instance.getUri());
  }
}
```

The biggest novelty of HEADREST is the fact that we can specify the behaviour of interactions via a hoare triple like structure. Having this extra expressiveness means that we can generate more code. From assertions described in HEADREST we generate a structure that can not only further validate the input but also assert the state of the system.

In the listings below we illustrate the OAS and HEADREST generated code for operations. As we can see, the OAS code is empty, extracting only method input and output types,

```
public Response addPet(Pet body, ...) throws NotFoundException {
  // do some magic!
  return Response.ok().entity(...).build();
}
```

In HEADREST we cover the assertions we specified, building the presented structure.

```
public Response petPost(PetRep body, ...) throws NotFoundException {

  boolean pre1 = ...;
  boolean pre2 = ...;
```

```
    boolean pre3 = ...;

    //TODO create response
    Response response = Response.ok().entity(...).build();

    boolean pos1 = ...;
    boolean pos2 = ...;
    boolean pos3 = ...;

    if(pre1) assert pos1 : "{pre1} petPost {pos1} failed.";
    if(pre2) assert pos2 : "{pre2} petPost {pos2} failed.";
    if(pre3) assert pos3 : "{pre3} petPost {pos3} failed.";

    return response;
  }
```

We can see that HEADREST Codegen provides us with better code, that will make the client SDK and server stubs development much easier and less error prone. Providing better type validation and the introduction of behaviour validation.

6.3 HEADREST vs HEADREST with Resource Logic

In this section we present a comparison between the code generated by HEADREST Codegen from the two types of HEADREST specifications, considering mainly the usefulness of the generated code for developers of the server code.

Specifications that take advantage of Resource Logic, as already mentioned before, are more organized and contain less boilerplate specification code, which results in an overall better user experience to the programmer, as it becomes easier to understand.

Assertion generated code can get quite verbose due to the fact that we do various state assertions, multiple specification resource/representation quantifications which result in various streams and conversions happening. The use of the Resource Logic allows us to avoid these enormous assertions as the generated code for loops is distributed from resources to its representations and the expressions are delegated to the representations.

In the following listing we present an assertion precondition that does not use Resource Logic. In the listing, we can see a nested loop of resource and representation iteration (lines 4-8).

```
1  {
2    request in {body: PetRep} &&
3    (isdefined(request.body.id) ==>
4      (forall pet:Pet .
5        (forall petRep:PetRep .
6          petRep representationof Pet => !pet.id == request.body.id
7        )
8      )
9    )
10 }
```

In the following listing we present a version that uses Resource Logic. Not only do we reduce the loop lines (lines 4-6) from 4 to 2, but we also make the specification abstract,

as we use the `hasid` predicate, abstracting the implementation.

```
1 {
2   request in {body: PetRep} &&
3   (isdefined(request.body.id) ==>
4     (forall pet:Pet .
5       !pet.hasid(request.body.id)
6     )
7   )
8 }
```

The following listing presents the generated code for the first case (no logic). As we can see, the generated Java expression for the precondition is verbose.

```
boolean pre1 = PetRep.validate(body)
  && (!(Predicates.isdefined(body, new String [] { "id" }))
  || RMCFactory.getInstance().getResources().getPetStream().allMatch(pet ->
      !(Predicates.isdefined(body, new String [] { "id" }) && Objects.
     equals(pet.getPetAsPetRep().id, body.getId()))));
```

On the other hand, the code generated for the second case, shown below, is more abstract and while it is still verbose, it reduces size of the generated expression significantly, making it much easier to read. We can deduct that this will be valuable for assertions that are complex.

```
boolean pre1 = PetRep.validate(body)
  && (!(Predicates.isdefined(body, new String [] { "id" }))
  || RMCFactory.getInstance().getResources().getPetStream().allMatch(pet ->
      !pet.hasid(body.getId())));
```

In the following we provide an illustration of the extra methods generated for the Pet interface and PetRep class as a result of using Resource Logic.

```
1 // Pet resource, resource predicate declaration and delegation
2 default boolean hasid(Integer arg0){
3   return getPetAsPetRep().hasid(arg0);
4 }
5
6 // PetRep representation, resource predicate implementation
7 public boolean hasid(Integer id){
8   return Predicates.isdefined(this, new String [] { "id" }) && Objects.
      equals(this.id, id);
9 }
```

With this we can conclude that the generated code gets much more readeable as well as the HEADREST specification. The result being organized this way, avoids repeated generated code (eventhough it could be refactored manually), helping with modularity, abstraction, and modifiability, to name a few. This results in richer and simpler HEADREST generated server stubs and, as collateral, specifications.

6.4 Experimental Study

In order to evaluate how HEADREST Codegen performs compared with Swagger codegen in what concerns the usefulness of the generated client SDKs, we performed an em-

pirical study with developers. The idea was to check whether the benefits stemming from more expressive specifications could be observed in practice.

Research question: *How useful is the client SDK generated by* HEADREST *Codegen tool in comparison with that generated by Swager Codegen?*

6.4.1 Context

The context of the experiment was that the subjects of the experiment, hereinafter called users/participants, were asked by a client to complete the development of a REST API client using both the Swagger Codegen and HEADREST Codegen generated client SDK code.

Figure 6.1 provides an overview of the code of the two clients that the users were asked to complete. This code is structured in terms of a UI that uses a controller that, by making use of the generated SDKs, consumes the RESTful service.

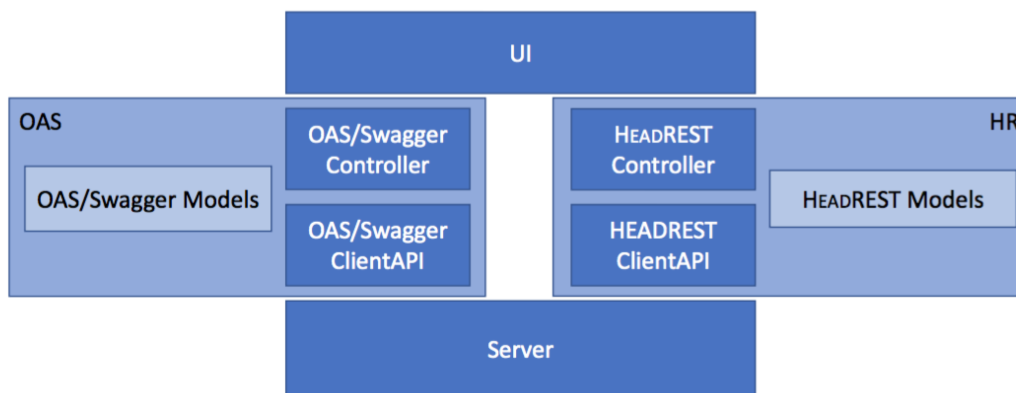


Figure 6.1: System illustration

The UI sends and receives strings to and from the controller, abstracting the data structures that are transmitted to the server. The controller processes the strings and converts them into models using them with the ClientAPI to consume the service provided by the server.

The participants of this experiment were students between the ages of 20-30, that were taking a bachelor's or master's degree in computer science. There was a total of 8 participants, chosen from colleagues to students suggested by the supervisor.

6.4.2 Experimental plan

With this experiment we intend to compare HEADREST Codegen to Swagger Codegen. Concretely we compared the client SDKs generated by both, asking participants to complete a client to consume an API.

The RESTful API used is relative to a contacts management service, with two main resources Contact and ContactList which are represented by ContactRep and ContactListRep, respectively. ContactPostData and ContactPutData are used as data for the available POST and PUT operations.

The final structure of the project should look as follows:

- UI - Developed in Swing, it makes controller calls for the various operations, giving input and expecting a string as return to present to the user.

- Controller - To be completed. Provides methods that represent operations, receiving necessary information via arguments and returning a string. The user is expected to complete the controller by grabbing the input and the correct Client API call if the input is valid.

- Client API - Generated by either the Swagger Codegen or the HEADREST Codegen, it is meant to be used by the controller to consume the API.

- Server - Server files already generated and working, providing the web service.

- JUnit - A set of tests to exercise some use cases on the developed Controller.

Use Cases (to test the controller against):

- User gets (GET) the list of contacts, which is empty
- User adds (POST) a valid contact
- User attempts to add (POST) an invalid contact
- User adds (POST) another valid contact
- User updates (PUT) an existent contact
- User gets (GET) the list of contacts, which is not empty
- User attempts to add (POST) a contact with invalid information
- User attempts to update (PUT) a contact with invalid information

User interface (UI)

In Figure 6.2 we can see an illustration of the UI that was created for the experiment. A simple UI that receives the parameters as string and delegates them to an appropriate controller method call, via the *GET*, *POST* and *PUT* buttons.

User Interface
GET /contacts
GET
(for state check)
POST /contacts
Id: 1
Name: Maria João
Email: mail@mailhost.com
Birth: 31121996
NewsSubscriptionType: ANUAL
NewsSubscriptionWhen: 100
POST
PUT /contacts/{id}
Name: Maria João
Email: mail@mailhost.com
NewsSubscriptionType: ANUAL
NewsSubscriptionWhen: 100
PathId: 1
PUT

Figure 6.2: User interface illustration

Controller stub

For the controller we created a simple Controller class (Appendix E.1), with incomplete methods on par with the ClientAPI class, that receives input via strings and returns a string. The users are meant to complete these controller methods taking advantage of the generated code.

JUnit

A suite of JUnit tests (Appendix E.2) was provided so that the users could verify the development of the controller. These tests reflected the aforementioned use cases.

6.4.3 Experiment execution

Firstly we provided the users with a setting up document, to get the generated server up and running before the experience and also the experimental plan (Appendix C.1) for them to get the context started. On the experiment day, we provided the participants with the necessary files (OAS Spec (Appendix D.1), HR Spec (Appendix D.2), Work paper

(Appendix C.2), both generated client Java-Maven projects) and made a little presentation of the experiment (see presented slides in Appendix C.3). During the experiment we helped the participants with minor difficulties (some did not work with Java for some time). After the experiment users were invited to participate in a survey (Appendix F), which they all did promptly.

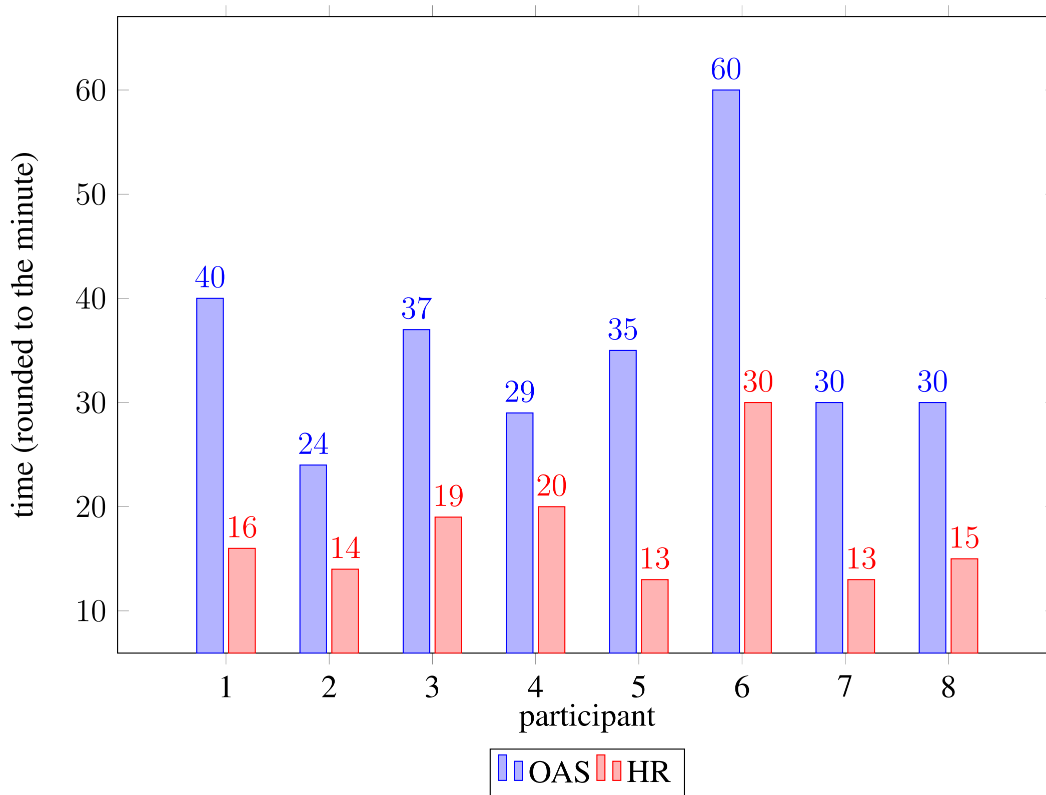
During the execution we took some metrics, namely the time taken and the number of lines of code (LOC) written needed to have a working client. In order to complete these metrics, subjective information was collected via the aformentioned survey that allowed us to compare the perception of usability and usefulness of HEADREST Codegen against Swagger Codegen.

The time taken was measured from the moment the users started to write the first line of code and ended with the last one. The LOC were counted after the previous metric was noted.

6.4.4 Quantitative Results

With this experiment we obtained information on how useful the tool is via the time taken to develop the controller and the number of LOC necessary to have a working controller, resulting in a fully working client. However, some mistakes were done relative to the way the experiment was conducted, that we will discuss further in the conclusion of this section. We concluded that the fact that we provided JUnit tests was a flaw in the experiment, as some users performed a test-driven development instead of using the API specification to guide the development.

Time taken



Participant	OAS time (min.)	HR time (min.)
1	40	16
2	24	14
3	37	19
4	29	20
5	35	13
6	60	30
7	30	13
8	30	15
Total :	285	140
Mean :	36	18
Std Deviation :	10	5

Table 6.1: Condensed time table.

As we can see in the previous plot and Table 6.1, there is a considerable time reduction when writing code in HEADREST, time is reduced by about half. This is mainly because HEADREST extends the OAS with classes that provide refinement type verification, which in the latter specification language normally is written down in natural language, not generating any useful code. Also, participants were requested to start with the OAS, which took more time due to the fact that they were trying to understand the problem at hand and how to work with the provided classes. When going to HEADREST

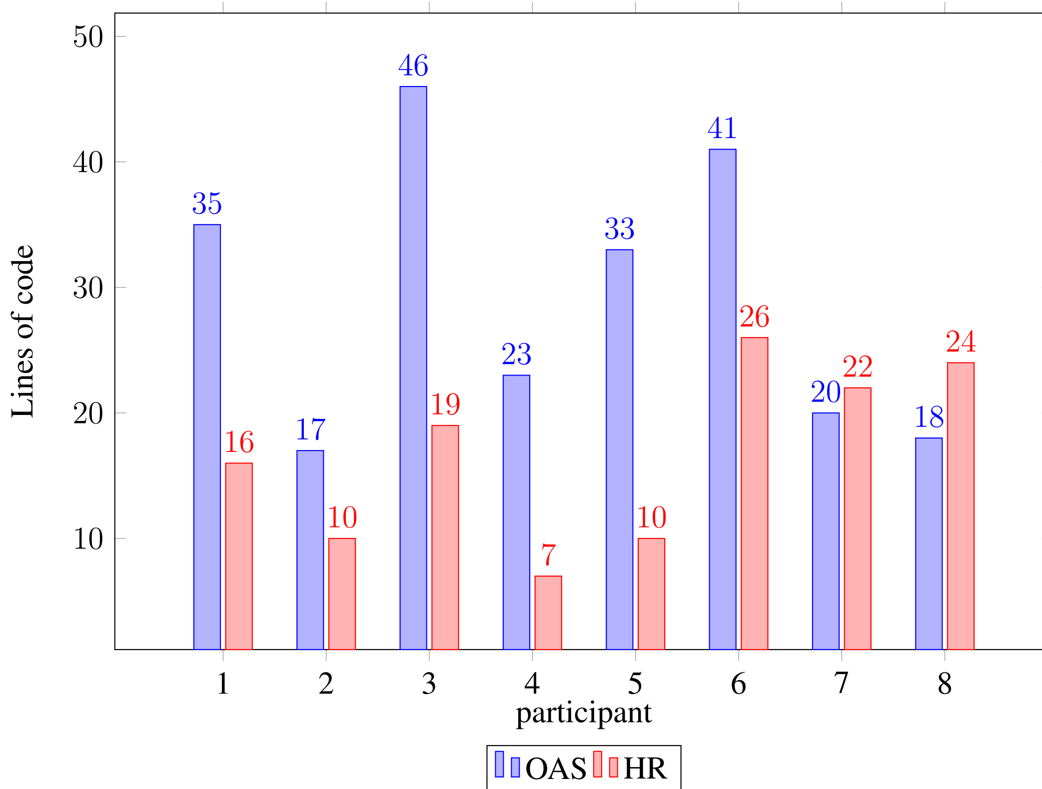
they already had a rough idea of what to do with the core classes, and given the extra functionality and the warmup, this resulted in faster times. Even with those variables, it is safe to assume that there is a time reduction from OAS to HEADREST, as the latter generates most of what the participants wrote by hand in OAS. Additionally, the generated code by HEADREST was better, as in the first case users wrote down the validations directly in the Controller stub methods.

In the following plot and Table 6.2 we can see that, for the vast majority of participants, HEADREST implied a big reduction in the number of written LOC.

Some participants took test-driven development strictly and only made type verifications to cover the tests, if new tests were made with other invalid properties (say we tested an invalid id but not an invalid email) then they would fail. In part HEADREST generated code is better, as it covers specified refinement types completely even in a test-driven development.

While looking at the written code we verified that most of the code in the Open API Specification client to verify the refinement types was written down directly in the controller method, which is an incorrect way of doing so. It makes code have less quality and it is less scalable and more prone to errors.

LOC (manually written)



With this experiment we can conclude that in fact HEADREST is an advantageous tool to use to generate RESTful APIs, reducing time taken to write code to complete the

Participant	OAS LOC	HR LOC
1	35	16
2	17	10
3	46	19
4	23	7
5	33	10
6	41	26
7	20	22
8	18	24
Total :	233	134
Mean :	29	17
Std Deviation :	10	7

Table 6.2: Condensed LOC table.

client as well as the number of lines of code. The overall quality of the code is also better, due to the fact that the code is better organised from the start.

6.4.5 Qualitative Results

We also created a survey with Google Forms[2](see Appendix F) to obtain some qualitative results. The survey was composed mainly of simple "yes or no" and "1 to 6" questions, and the purpose was to obtain some feedback relative to the tools as well as some subjective information relative to the experience with the tools.

Answers to the survey

In what follows, we show the summary of the responses to the different questions of the survey.

Keys used

Table 6.3: Key used for 1 to 6 questions (greener means better)

1	2	3	4	5	6

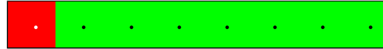
Table 6.4: Key used for yes or no questions

no	yes

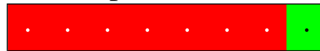
Questions and Answers

[2] `https://www.google.com/forms/about/` (Seen: 2018-05-19)

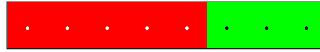
Did you know about RESTful APIs before the experiment ?



Did you know the Open API Specification (OAS/Swagger) language ?

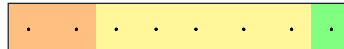


Did you know the HEADREST (HR) language ?

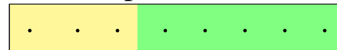


Have you used Swagger Codegen before ?

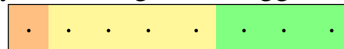
How easy was OAS specification to understand ?



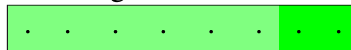
How easy was HR specification to understand ?



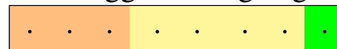
How easy was using the Swagger-Codegen ?



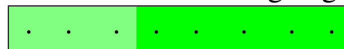
How easy was using the HEADREST-Codegen ?



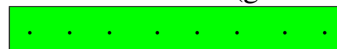
How useful was Swagger-Codegen generated code ?



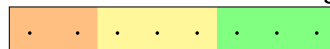
How useful was HEADREST-Codegen generated code ?



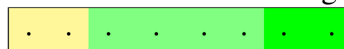
Which tool was the most useful ? (green = HR, red = OAS)



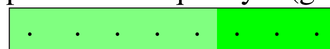
How easy was it to understand the OAS generated code ?



How easy was it to understand the HR generated code ?



Which client ended up with more quality ? (green = HR, red = OAS)



6.4.6 Conclusions

From the results we conclude that HEADREST Codegen is an improvement to the Swagger Codegen, providing more and better generated boilerplate code in less time. This comes as a result of the fact that we extend an existing tool with the extra expressiveness

that HEADREST allows. Models have refinement validating methods appended, which are very easy to use, and clients have methods that allow them to check for successful interactions before communicating with the server, allowing better error handling. All this results in an overall better user experience for the programmer, with improved understandability, usability, usefulness and final product quality.

In what concerns the research question of the study, our conclusion was that the generated code by HEADREST is indeed more useful than the code generated by Swagger Codegen. We conclude from both the quantitative and qualitative results that HEADREST Codegen is a very powerful tool, reducing time taken, lines of code written and overall providing a better final product when developing client SDKs, even with the minor setback that the experiment was not conducted properly.

6.4.7 Threats to validity

Several problems were identified in the design of the experiment. These problems result in additional threats to validity that must be considered when analysing the experimental results.

- We should have conducted the experiment differently, with a special emphasis on the specification languages rather than on the tests. As discussed before, some participants completed the controller only to comply with the tests rather than the provided specification — tests were meant to serve as an auxiliary tool in the development of the controller, not as a substitute to the specification provided.

- We used the same problem example for both tools. Even though the problem solution would resume to the same logic "prepare data, validate data, send request, receive response, prepare response for UI", having different problems with similar complexity would have been beneficial.

- Participants were asked to start with OAS and then HEADREST. We should have split half of the participants to start differently, to avoid bias from one experiment part to the other.

- The small number of participants and the fact that some of them were colleagues of the author constitutes also a thread to validity. The obtained results do not accurately conform to the law of large numbers and it is possible that responses were biased.

- The participant demography was not representative nor heterogeneous of the community of developers of clients of RESTful services, as the participants were all computer science students with very similar work experience.

Chapter 7

Conclusions

7.1 Summary

The work presented in this thesis contributes with a new technique that facilitates the development of RESTful systems. Specifically, it proposes a new technique that extends existing code generation techniques for REST API specification taking advantage of the behavioural properties that can be expressed with HEADREST.

As a proof of concept, it was also developed a tool — HEADREST Codegen— that generates Java code (that uses the RestEasy framework) from specifications of RESTful APIs described in HEADREST. This is achieved by first encoding HEADREST specifications into OAS (taking advantage of OAS extension mechanisms to carry HEADREST's extra expressiveness) and then, generating client SDKs and server stubs (by extending Swagger Codegen to acommodate for the extensions appended to the OAS specification). Additionally, an extension of HEADREST with *Resource Logic* was proposed and implemented that provides a way of generating better code for both resources and their representations and, at the same time, also facilitates the description of RESTful APIs in HEADREST.

The tool has some limitations s.a. not supporting `any`, `null` or union types that were inherited from OAS, Swagger Codegen, and RESTEasy.

If we were to generate directly from HEADREST specifications to Java, as it is our main programming language, the `any` type could, for instance, be translated as `Object` (the top Java type). Other limitations could probably also be avoided in a direct translation. However, a direct solution would require to implement all the translation related with models, which in general terms was already covered by Swagger Codegen.

In order to evaluate the proposed techniques, we conducted an experiment in which users were asked to implement the client side, for given specification and a given server. The experiment should have been approached differently, but the data that was collected suggests that our tool is an improvement with respect to the typical syntactic generation tools.

In the end, the extra semantic information that HEADREST provides, the encoding to OAS and the extension of the Swagger Codegen, all come together providing us with richer client SDKs and server stubs.

7.2 Future work

Even though we successfully implemented HEADREST Codegen, there is still a lot of work that can be done to improve it.

Extract ApiClient

The Swagger Codegen native *ApiClient* currently deals with the error codes. If we extract the *ApiClient* or extend it, we may take more advantage of pre and postconditions on the client side. Indirectly we would be able to have various return types from the server, which might need to have some sort of union type class created and returned.

Update to OAS 3.0

This will enable, for example, the use of null types (OAS new *nullable* keyword) and union types (OAS 3.0 new *oneOf* keyword), but did not exist at the start of this thesis.

Use OAS Categories

Based on the type of resource the assertion is based in we could categorize them. Maybe extracting the category name from the first /uriFragment so that `/person/{id}` would be categorized as "Person" and `/person/{id}/limb{id2}` would be also categorized as "Person" but `/pet` would be categorized as "Pet". Generating more than "DefaultAPI" and making the code that much more organized.

Enumerates value

When HEADREST Codegen encounters a type disjunction it attempts to generate an enumerate from it. When we convert a refinement type that includes enumerates, that uses the equality operator for instance, we may encounter "unlikely argument type" warnings. This is because we use *getEnum* method, which returns the enumerate, where we actually expect its value. Some efforts can be made to try and add a *getValue* method call when we try and obtain an enumerate's value.

Update the HEADREST version used

Even though we updated the HEADREST language with new elements, during the development of this work, further updates were done to the language. Efforts can be made

in order to update HEADREST Codegen to use the new version of the HEADREST language.

Appendix A

Encoding rules

Type encoding rules

$$
\begin{aligned}
[[\ B\ ]] &= \text{type: “}B\text{”} \\
[[\ URI\ ]] &= \text{\$ref: “\#/definitions/Uri”} \\
[[\ \alpha\ ]] &= \text{\$ref: “\#/definitions/}\alpha\text{”} \\
[[\ \{l_0 : T_0\ , ...,\ l_k : T_k\ , ...,\ ?l_{k+1} : T_{k+1}\ , ...,\ ?l_n : Tn\}\ ]] &= \text{type: “object”} \\
&\quad \text{required:} \\
&\quad \text{- “}l_0\text{”} \\
&\quad \dots \\
&\quad \text{- “}l_k\text{”} \\
&\quad \text{properties:} \\
&\quad \text{- } l_0 : [[\ T_0\ ]] \\
&\quad \dots \\
&\quad \text{- } l_n : [[\ T_n\ ]] \\
[[\ T\ [\ ]\ ]] &= \text{type: “array”} \\
&\quad \text{items: } [[\ T\ ]]
\end{aligned}
$$

Refinement type encoding rule

$$
\begin{aligned}
[[\ (x\colon T\ \textsf{where}\ e)\ ]] &= [[\ T\ ]] \\
&\quad \text{x-refinement: “}e\text{”}
\end{aligned}
$$

Type intersection rules

$$
\begin{aligned}
[[\ G \,\&\, ... \,\&\, G\ ]] &= [[\ G\ ]] \\
[[\ \alpha \,\&\, ... \,\&\, \alpha\ ]] &= [[\ \alpha\ ]] \\
[[\ (x\colon T \text{ where } e_1) \,\&\, T\ ]] &= [[\ (x\colon T \text{ where } e_1)\ ]] \\
[[\ (x\colon T_0 \text{ where } e_0) \,\&\, (x\colon T_1 \text{ where } e_1)\ ]] &= [[\ (x\colon T_0 \,\&\, T_1 \text{ where } e_0 \,\&\&\, e_1)\ ]] \\
[[\ \{l_0 : T_0, ..., l_n : T_n\} \,\&\, \{\star l_x, T_x\}\ ]] &= [[\ \{l_0 : T_0, ..., l_n : T_n, \star l_x : T_x\}\ ]] \\
&\quad where\ \ l_x \notin \{l_0, ..., l_n\} \\
[[\ \{l_0 : T_0, ..., l_k : T_k, ..., l_n : T_n\} \,\&\, \{\star l_k : U\}\ ]] &= [[\ \{l_0 : T_0, ..., l_k : (T_k \,\&\, U), ..., l_n : T_n\}\ ]]
\end{aligned}
$$

Type merging rules

$$
\begin{aligned}
[[\ G \mid ... \mid G\ ]] &= [[\ G\ ]] \\
[[\ \alpha \mid ... \mid \alpha\ ]] &= [[\ \alpha\ ]] \\
[[\ (\ x_0 : G \text{ where } e_0) \mid ... \mid (x_n : G \text{ where } e_n)\ ]] &= [[\ (x : G \text{ where } e_0 \mid\mid ... \mid\mid e_n)\ ]] \\
[[\ |_i\{\vec{a_i} : \vec{T_i}, ?\vec{b_i} : \vec{T_i}\}\ ]] &= [[\ (x : \{?\vec{a_i} : \vec{T_i}, ..., ?\vec{a_n} : \vec{T_n}, ?\vec{b_i} : \vec{T_i}\} \\
&\quad where\ \ ||_i\ isdefined(\vec{a_i}))\ ]]
\end{aligned}
$$

Singleton merging rules

$$[[\ [e : G]\ ]] = [[\ G\ ]]$$

enum:

- e

$$[[\ [e_o : G] \mid ... \mid [e_x : G]\ ]] = [[\ G\ ]]$$

enum:

- e_0

...

- e_x

Assertion encoding rule $[[\ \{e_0\}\ a\ /p\ \{e'_0\}\ \ ...\ \ \{e_n\}\ a\ /p\ \{e'_n\}\ ]] =$

```
/p:
a:
parameters:
-in: "body"
name: "body"
required: true
schema: |0≤i≤k[[Ti]]
responses:
c0:
description: <code detail extracted from IANA csv file>
schema: |0≤j≤n[[Tj]] where response.code == code0
...
cn:
description: <code detail extracted from IANA csv file>
schema: |0≤j≤n[[Tj]] where response.code == coden
x-axioms:
- "{e0} a /p {e'0}"
- ...
- "{en} a /p {e'n}"
```

where:

- T_i and T_j are extracted from the respective e_i.request.*body* in T_i (from preconditions) and e'_j.response.*body* in T_j (from postconditions) expressions
- c_0,...,c_n are extracted from the postconditions `response.code == ci` expressions,
- we assume that assertions from 0 to k refer to successful cases, i.e., response codes in the interval [200,300[, while the remainder assertions have different codes,
- IANA stands for Internet Assigned Numbers Authority, which provides a list of HTTP codes and related information (e.g., code 404 has description “Not Found”).

Appendix B

Predicates class

```
1 package io.headrest.resources;
2
3 import java.lang.reflect.Field;
4 import java.util.Arrays;
5 import java.util.List;
6 import java.util.Map;
7 import java.util.Map.Entry;
8 import java.util.regex.Pattern;
9
10 /**
11  * Predicates handling class.
12  *
13  * @author Telmo Santos
14  *
15  */
16 public class Predicates {
17
18     /**
19      * Checks if a string matches a given pattern
20      *
21      * @param pattern
22      *            the pattern
23      * @param string
24      *            the string
25      * @return true if string matches the pattern, false otherwise
26      */
27     public static boolean matches(String pattern, String string) {
28         return Pattern.compile(pattern).matcher(string).matches();
29     }
30
31     /**
32      * @return length of string
33      */
34     public static int length(String string) {
35         return string.length();
36     }
37
38     /**
39      * @return length of array
40      */
41     public static int length(Object[] array) {
42         return array.length;
43     }
44
45     /**
46      * @return length of list
47      */
48     public static int length(List<? extends Object> list) {
49         return list.size();
50     }
51
52     /**
53      * Verifies if a string contains another
54      *
55      * @param s1
56      *            the contained
57      * @param s2
58      *            the container
59      * @return true if s2 contains s1
60      */
61     public static boolean contains(String s1, String s2) {
62         return s2.contains(s1);
63     }
64
65     /**
66      * Verifies if an array contains an object
67      *
68      * @param array
69      *            the array
70      * @param o
71      *            the object
72      * @return true if array contains o
73      */
74     public static boolean contains(Object[] array, Object o) {
75         for (int i = 0; i < array.length; i++)
76             if (array[i].equals(o))
77                 return true;
78         return false;
79     }
80
81     /**
82      * Verifies if a string contains all of the elements in an array
83      *
84      * @param array
85      *            the array
86      * @param s
87      *            the string
88      * @return true if s contains all elements in array
89      */
90     public static boolean contains(Object[] array, String s) {
91         try {
92             for (int i = 0; i < array.length; i++)
93                 if (!s.contains((CharSequence) array[i]))
94                     return false;
95         } catch (ClassCastException cce) {
96             return false;
97         }
98         return true;
99     }
100
101     /**
102      * Verifies if an instance has a member/sub-member defined
103      *
104      * @param instance
```

```
105      *            the instance
106      * @param remaining
107      *            the list of members
108      * @return true if members are defined, false otherwise
109      */
110     public static boolean isdefined(Object instance, String[] remaining) {
111         try {
112             // Get the field in question, if field doesn't exist
113             // NoSuchFieldException is thrown
114             Field field = instance.getClass().getDeclaredField(remaining[0]);
115 
116             // If last field, it exists otherwise NoSuchFieldException would've
117             // been thrown, so return true
118             if (remaining.length == 1)
119                 return true;
120 
121             // If not last field, recursively descend on the field
122             return isdefined(field.get(instance), Arrays.copyOfRange(remaining, 1, remaining.length));
123         } catch (NoSuchFieldException nsfe) {
124             return false;
125         } catch (Exception e) {
126             throw new RuntimeException("isdefined failed for instance:" + instance + " with remaining: " + remaining);
127         }
128     }
129 
130     /**
131      * Expands an uri template with a given dictionary
132      *
133      * @param uriTemplate
134      *            the uri template
135      * @param dictionary
136      *            the dictionary
137      * @return the expanded uri template
138      */
139     public static String expand(String uriTemplate, Map<String, Object> dictionary) {
140         String result = uriTemplate;
141         for (Entry<String, Object> e : dictionary.entrySet())
142             result = result.replace(getAsExpandee(e.getKey()), valueToString(e.getValue()));
143         return result;
144     }
145 
146     /**
147      * @param o
148      *            the object
149      * @return the string representation of an object
150      */
151     private static CharSequence valueToString(Object o) {
152         return o.toString();
153     }
154 
155     /**
156      * @param expandee
157      *            the element to be expanded
158      * @return the expandee in the form it appears on an uriTemplate string
159      *         (e.g. uriTemplate = "/someuri/{id}" dictionary = {id:"0"} ->
160      *         expandee = id is converted to {id})
161      */
162     private static String getAsExpandee(String expandee) {
163         return "{" + expandee + "}";
164     }
165 
166 }
167 
```

Appendix C

Experiment files

C.1 Experimental Plan

Experiment Plan

Context

With this experiment we intend to compare two tools that generate client SDKs for consuming RESTful APIs. Swagger Codegen is an existent generation tool for, a popular RESTful API definition language, the Open API Specification (OAS). HEADREST Codegen is the tool that we developed for the HEADREST (HR) specification language.

OAS: https://swagger.io
HR: http://rss.di.fc.ul.pt/tools/confident/

Experiment

We want to find out if our client generation from HR is more useful than OAS in what concerns the client side development. For this we want to create a working client, generating from both HR and OAS.

The RESTful API is relative to a contacts management API, with two main resources Contact and ContactList which are represented by ContactRep and ContactListRep, respectively. ContactPostData and ContactPutData are used as data for the available POST and PUT operations.

The final structure of the project should look as follows:

UI

Developed in Swing, it makes controller calls for the various operations, giving input and expecting a string as return to present to the user.

Controller

To be completed. Provides methods that represent operations, receiving necessary information via arguments and returning a string. The user is expected to complete the controller by grabbing the input and the correct Client API call if the input is valid.

Client API

Generated by either the *Swagger Codegen* or the *HEADREST Codegen*, it is meant to be used by the controller to consume the API.

Server

Server files already generated and working, providing the web service.

JUnit

A set of tests to exercise some use cases on the developed Controller.

Use Cases

- User gets (GET) the list of contacts, which is empty
- User adds (POST) a valid contact
- User attempts to add (POST) an invalid contact
- User updates (PUT) an existent contact
- User gets (GET) the list of contacts, which is not empty

C.2 Work paper

Work

Scenario

Our clients want to develop a RESTful API relative to a contacts management API, with two main resources Contact and ContactList which are represented by ContactRep and ContactListRep, respectively.

They have provided the specification of the system via HEADREST/OAS files, which were used to generate **client code** via HEADREST Codegen and Swagger Codegen respectively. They then created a stub Controller class which is meant to serve as interface for a User Interface, which they require that you complete.

Overview of the final structure of the project should look as follows:

UI

Developed in Swing, it makes controller calls for the various operations, giving input and expecting a string as return to present to the user.

Controller

To be completed. Provides methods that represent operations, receiving necessary information via arguments and returning a string. The user is expected to complete the controller by grabbing the input and calling the correct Client API call if the input is valid.

Client API

Generated by either the *Swagger Codegen* or the *HEADREST Codegen*, consumes the API.

Server

Server files already generated and working, providing the web service.

JUnit

A set of tests to exercise some use cases on the developed Controller.

Use Cases

- User gets (GET) the list of contacts, which is empty
- User adds (POST) a valid contact
- User attempts to add (POST) an invalid contact
- User updates (PUT) an existent contact
- User adds (POST) another valid contact
- User gets (GET) the list of contacts, which is not empty
- User attempts to add (POST) a contact with invalid information
- User attempts to update (PUT) a contact with invalid information
- User attempts to add (POST) a contact with incomplete information (nulls)
- User attempts to update (PUT) a contact with invalid information (nulls)

Generated project packages

- src/main/java – main java package
    - io.headrest/swagger.client.ui
        - the user interface
    - io.headrest/swagger.client.controller
        - the controller used by the user interface
    - io.headrest/swagger.client.api
        - contains the DefaultApi.java – the Client API
    - io.headrest/swagger.client.model
        - contains the model files
- src/main/test – test java package

What to do?

You will do a sort of Test Driven Development where your objective is to clear all the tests from src/main/test/io.headrest/swagger.client.controller/ControllerTest.java while taking note of how much time it takes until all tests run green and, in the end, using a diff tool to count to number of written LOC, as well as provide the final project.

The generated file you start with will be OAS first and HR last, and you will have to make both HR and OAS generated client controllers.

You will also be asked to participate in a qualitative/voluntary survey where you will be asked a couple of answers - yes/no, range from x to y.

Survey link: https://goo.gl/forms/HZSnZJPcgDznBlVJ2

Survey password: **aAbBcCdD4321**

Output

- On experiment (for each client project)
    - Total time taken
    - LOC (lines of code)
    - Project
- After experiment
    - Survey

Feel free to ask any questions and good luck! ☺

C.3 Presentation

Client generation with HEADREST and OAS/Swagger

Telmo Santos

Context

- Client created a UI
- Client has a working server
- Client needs you to create the Controller
  - The UI gets and gives everything as strings
  - Wants to abstract from request data structure

User Interface

GET /contacts

GET

(for state check)

POST /contacts

Id: 1

Name: Maria João

Email: mail@mailhost.com

Birth: 31121996

NewsSubscriptionType: ANUAL

NewsSubscriptionWhen: 100

POST

PUT /contacts/{id}

Name: Maria João

Email: mail@mailhost.com

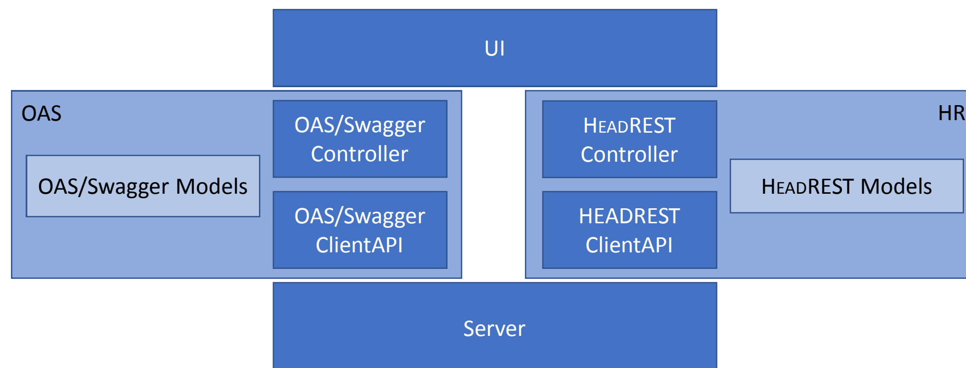
NewsSubscriptionType: ANUAL

NewsSubscriptionWhen: 100

PathId: 1

PUT

View



OAS/Swagger Specification File

- Explain OAS/Swagger Spec
- Go over the generated code

Client API –OAS/Swagger

- ClientAPI found as DefaultAPI class
  - Provides methods do consume the API

Models –OAS/Swagger

- Models are found on the io.(…).model package
  - Provide the data structure

OAS/Swagger development

- Let the users create the Controller for the OAS/Swagger generated files

HEADREST Specification File

- Explain HEADREST Spec
- Go over the generated code

Client API – HEADREST

- ClientAPI found as DefaultAPI class
  - Provides methods do consume the API
  - **Provides success precondition verification methods (HR only)**
    - Uses special classes to return precondition success or failure (with cause)
      - **BooleanValidation**
      - **JustifiedInvalid** (extends BooleanValidation to make it false and carry the cause)

```
/**
 * @returns BooleanValidation with true if the success precondition is verified, JustifiedInvalid with reason otherwise
 */
public BooleanValidation personPostRequiredSuccessPrecondition(Person body) {
    boolean pre1 = Person.validate(body);
    return pre1 ? new BooleanValidation(true) : new JustifiedInvalid(getReasonString(pre1) + " failed.");
}
```

Models – HEADREST

- Models are found on the io.(…).model package
  - Provide the data structure
  - **Provide validation method for instances of those models**

```
/**
 * Method used to validate an instance of this type
 *
 * @param instance
 *            the instance
 */
public static boolean validate(Person instance) {
    return instance.getAge() > 18;
}
```

Usage of the extra generated code

```
{
    request in {body: PersonPostData} &&&
    (forall p: Person .  !p.hasName(request.body.name))
}
    POST /person
{
    response.code == SUCCESS &&
    response in {body: PersonRep}
}
```

Usage of the extra generated code

```
public String personPost(String name, String age){
    PersonPostData pp = new PersonPostData();
    pp.setName(name);
    pp.setAge(Integer.valueOf(age));

    if(PersonPostData.validate(pp)){
        PersonRep pr = API.personPost(pp);
        return pr.toString();
    } else {
        return "Error :C";
    }
}
```

- Server-side verifications cannot happen client-side
  - basically state verifications
    - (forall p: Person . !p.hasName(request.body.name))
- Part of the precondition can be verified client-side
  - mainly data structure validation

Usage of the extra generated code

{

request in {body: PersonPostData} &&&

request.body.age > 20 &&

(forall p: Person . !p.hasName(request.body.name))

}

POST /person

{

response.code == SUCCESS &&

response in {body: PersonRep}

}

- Extra restrictive, not only does the person have to be over age (to be a valid PersonPostData) but also over 20 years old in order to be SUCCESS

Usage of the extra generated code

```
public BooleanValidation personPostRequiredSuccessPrecondition(PersonPostData body) {
    boolean pre1 = PersonPostData.validate(body) && body.getAge() > 20;
    return pre1 ? new BooleanValidation(true) : new JustifiedInvalid(getReasonString(pre1) + " failed.");
}
```

```
public String personPost(String name, String age) {
    PersonPostData ppd = new PersonPostData();
    ppd.setName(name);
    ppd.setAge(Integer.valueOf(age));

    BooleanValidation pprsp = API.personPostRequiredSuccessPrecondition(ppd);

    if (pprsp instanceof BooleanValidation && pprsp.isValid()) {
        PersonRep pr = API.personPost(ppd);
        return pr.toString();
    } else if (pprsp instanceof JustifiedInvalid){
        return "personPost" + ((JustifiedInvalid)pprsp).getReason();
    }

    return "Something went wrong !";
}
```

- Abstracts the precondition validation
- Easier error management

HEADREST development

- Let the users create the Controller for the HR generated files

Appendix D

Experiment Specification files

D.1 OAS Spec

```yaml
swagger: "2.0"
info:
  version: "1.0.0"
  title: "ContactsWithLogic"
basePath: "/headrest-jaxrs-resteasy-server"
paths:
  /contacts/{id}:
    put:
      parameters:
      - in: "body"
        name: "body"
        required: true
        schema:
          $ref: "#/definitions/ContactPutData"
      - name: "id"
        in: "path"
        required: true
        type: "integer"
        format: "int32"
      responses:
        200:
          description: "OK"
          schema:
            $ref: "#/definitions/ContactRep"
  /contacts:
    get:
      parameters: []
      responses:
        200:
          description: "OK"
          schema:
            $ref: "#/definitions/ContactListRep"
    post:
      parameters:
      - in: "body"
        name: "body"
        required: true
        schema:
          $ref: "#/definitions/ContactPostData"
      responses:
        201:
          description: "Created"
          schema:
            $ref: "#/definitions/ContactRep"
          headers:
            Location:
              type: "string"
        409:
          description: "Conflict"
definitions:
  ContactRep:
    type: "object"
    required:
    - "birthDate"
    - "email"
    - "id"
    - "name"
    - "newsSubscriptionType"
    - "newsSubscriptionWhen"
    - "registerDate"
```

```
    properties:
      id:
        type: "integer"
        format: "int32"
        description: "id must be positive"
      registerDate:
        type: "integer"
        format: "int32"
        description: "must be greater than birth date"
      birthDate:
        type: "integer"
        format: "int32"
        description: "must less than register date"
      email:
        type: "string"
        description: "string that contains @"
      newsSubscriptionType:
        type: "string"
        enum:
        - "ANUAL"
        - "SEMANAL"
        - "DIARIO"
      name:
        type: "string"
      newsSubscriptionWhen:
        type: "integer"
        format: "int32"
        description: "natural (includes 0) and newsSubscriptionType ANUAL
goes to 365, SEMANAL goes to 7, DIARIO goes to 24"
  ContactPutData:
    description: "must comply with ContactRep"
    type: "object"
    required:
    - "email"
    - "name"
    - "newsSubscriptionType"
    - "newsSubscriptionWhen"
    properties:
      name:
        type: "string"
      newsSubscriptionType:
        type: "string"
        enum:
        - "ANUAL"
        - "SEMANAL"
        - "DIARIO"
      email:
        type: "string"
      newsSubscriptionWhen:
        type: "integer"
        format: "int32"
  ContactListRep:
    type: "object"
    required:
    - "contacts"
    properties:
      contacts:
        type: "array"
        items:
          $ref: "#/definitions/ContactRep"
```

```
ContactPostData:
  description: "must comply with ContactRep"
  type: "object"
  required:
  - "birthDate"
  - "email"
  - "id"
  - "name"
  - "newsSubscriptionType"
  - "newsSubscriptionWhen"
  properties:
    id:
      type: "integer"
      format: "int32"
    email:
      type: "string"
    newsSubscriptionType:
      type: "string"
      enum:
      - "ANUAL"
      - "SEMANAL"
      - "DIARIO"
    name:
      type: "string"
    birthDate:
      type: "integer"
      format: "int32"
    newsSubscriptionWhen:
      type: "integer"
      format: "int32"
Uri:
  type: "string"
```

D.2 HRSpec

```
specification ContactsWithLogic

// Resource declaration
resource Contact {
        pred hasId(integer),
        pred hasName(string),
        pred hasEmail(string),
        pred hasBirthDate(integer)
}

resource ContactList {
        pred contains(Contact)
}

// Type declaration
type ContactRep = (cr: {
        id: integer,
        name: (x : string where x.length > 2),
        email: (e: string where contains(["@"],e)),
        birthDate: integer,
        registerDate: integer,
        newsSubscriptionType: (subType: string where subType in ["ANUAL"] || subType in ["SEMANAL"] || subType in ["DIARIO"]),
        newsSubscriptionWhen: integer
} where
        (cr.newsSubscriptionWhen > 0) &&
        (cr.birthDate < cr.registerDate) &&
        (cr.newsSubscriptionType == "ANUAL" => cr.newsSubscriptionWhen <= 365) &&
        (cr.newsSubscriptionType == "SEMANAL" => cr.newsSubscriptionWhen <= 7) &&
        (cr.newsSubscriptionType == "DIARIO" => cr.newsSubscriptionWhen <= 24)
)

type ContactPostData = (cpd: {
        id: integer,
        name: (x : string where x.length > 2),
        email: (e: string where contains(["@"],e)),
        birthDate: integer,
        newsSubscriptionType: (subType: string where subType in ["ANUAL"] || subType in ["SEMANAL"] || subType in ["DIARIO"]),
        newsSubscriptionWhen: integer
} where
        (cpd.newsSubscriptionWhen > 0) &&
        (cpd.newsSubscriptionType == "ANUAL" => cpd.newsSubscriptionWhen <= 365) &&
        (cpd.newsSubscriptionType == "SEMANAL" => cpd.newsSubscriptionWhen <= 7) &&
        (cpd.newsSubscriptionType == "DIARIO" => cpd.newsSubscriptionWhen <= 24)
)

type ContactPutData = (cpd: {
        name: (x : string where x.length > 2),
        email: (e: string where contains(["@"],e)),
        newsSubscriptionType: (subType: string where subType in ["ANUAL"] || subType in ["SEMANAL"] || subType in ["DIARIO"]),
        newsSubscriptionWhen: integer
} where
        (cpd.newsSubscriptionWhen > 0) &&
        (cpd.newsSubscriptionType == "ANUAL" => cpd.newsSubscriptionWhen <= 365) &&
        (cpd.newsSubscriptionType == "SEMANAL" => cpd.newsSubscriptionWhen <= 7) &&
        (cpd.newsSubscriptionType == "DIARIO" => cpd.newsSubscriptionWhen <= 24)
)

type ContactListRep = {
        contacts: ContactRep[]
}

/**
 * Link representation with resource, defining a predicate
 */
contact:ContactRep represents Contact{
        hasId(var1) => var1 == contact.id,
        hasName(var1) => var1 == contact.name,
        hasEmail(var1) => var1 == contact.email,
        hasBirthDate(var1) => var1 == contact.birthDate
}

contactList:ContactListRep represents ContactList{
        contains(contact) => (exists i:(x: integer where x >= 0 && x < contactList.contacts.length) . contact.hasId(contactList.contacts[i].id))
}
```

```
// Constant declarations
def SUCCESS = 200
def CREATED = 201
def CONFLICT = 409

// Axioms
{
	true
}
	GET /contacts
{
	response.code == SUCCESS &&
	response in {body: ContactListRep} &&&
	(exists l: ContactList .
		response.body representationof l
	)
}

{
	request in {body: ContactPostData} &&&
	(forall c: Contact . !c.hasId(request.body.id))
}
	POST /contacts
{
	response.code == CREATED &&
	response in {body: ContactRep, header: {Location: URI}} &&&
	(exists l: ContactList .
		(exists c: Contact .
			l.contains(c) &&
			response.body representationof c &&
			response.header.Location uriof c &&
			c.hasId(response.body.id) &&
			c.hasName(response.body.name) &&
			c.hasEmail(response.body.email) &&
			c.hasBirthDate(response.body.birthDate)
		)
	)
}

// add contact, CONFLICT
{
	request in {body: ContactPostData} &&&
	(exists c: Contact . c.hasId(request.body.id))
}
	POST /contacts
{
   response.code == CONFLICT
}

// update contact, SUCCESS
{
	request in {body: ContactPutData, template:{id:integer}} &&&
	(exists c: Contact . c.hasId(request.template.id))
}
	PUT /contacts/{id}
{
	response.code == SUCCESS &&
	response in {body: ContactRep} &&&
	(exists l: ContactList .
		(exists c: Contact .
			l.contains(c) &&
			response.body representationof c &&
			c.hasId(response.body.id) &&
			c.hasName(response.body.name) &&
			c.hasEmail(response.body.email) &&
			c.hasBirthDate(response.body.birthDate)
		)
	)
}
```

Appendix E

Experiment classes

E.1 Controller

```java
1 package io.headrest.client.controller;
2
3 import io.headrest.client.ApiException;
7
8 /**
9  * Class that serves as interface to the UI
10  */
11 public class Controller {
12
13     /**
14      * The generated client api
15      */
16     private static final DefaultApi API = new DefaultApi();
17
18     /**
19      * Executes the <i>GET /contacts</i> operation
20      *
21      * @return the result of the operation
22      */
23     public String contactsGet() {
24         try {
25             ContactListRep result = API.contactsGet();
26             if(result.getContacts().size() == 0)
27                 return "No contacts added yet.";
28             return result.toString();
29         } catch (ApiException e) {
30             return e.getMessage();
31         }
32     }
33
34     /**
35      * Executes the <i>PUT /contacts/{id}</i> operation
36      *
37      * @param name
38      *            the name to update to
39      * @param email
40      *            the email to update to
41      * @param newsSubscriptionType
42      *            the newsSubscriptionType to update to
43      * @param newsSubscriptionWhen
44      *            the newsSubscriptionWhen to update to
45      * @param pathId
46      *            the id of the path
47      * @return the result of the operation
48      */
49     public String contactsIdPut(String name, String email, String newsSubscriptionType, String newsSubscriptionWhen,
50             String pathId) {
51         try {
52             // TODO complete
53             ContactRep cr = API.contactsIdPut(null, null);
54             return cr.toString();
55         } catch (ApiException e) {
56             return e.getMessage();
57         }
58     }
59
60     /**
61      * Executes the <i>POST /contacts</i> operation
62      *
63      * @param id
64      *            the contact id
65      * @param name
66      *            the contact name
67      * @param email
68      *            the contact email
69      * @param birthDate
70      *            the contact birth date
71      * @param newsSubscriptionType
72      *            the contact newsSubscriptionType
73      * @param newsSubscriptionWhen
74      *            the contact newsSubscriptionWhen
75      * @return the result of the operation
76      */
77     public String contactsPost(String id, String name, String email, String birthDate, String newsSubscriptionType,
78             String newsSubscriptionWhen) {
79         try {
80             // TODO complete
81             ContactRep cr = API.contactsPost(null);
82             return cr.toString();
83         } catch (ApiException e) {
84             return e.getMessage();
85         }
86     }
87 }
88
```

E.2 Controller Test

```
1 package io.headrest.client.controller;
2
3 import static org.junit.Assert.assertTrue;
8
9 /**
10  * Tests for the controller
11  *
12  * @author Telmo Santos
13  *
14  */
15 @FixMethodOrder(MethodSorters.NAME_ASCENDING)
16 public class ControllerTest {
17
18     /**
19      * Controller
20      */
21     private static final Controller CONTROLLER = new Controller();
22
23     @Test
24     public void test0_contactsGetTestEmptyContactsList() {
25         // when
26         String result = CONTROLLER.contactsGet();
27         // then
28         assertTrue(
29                 "contactsGetTestEmptyContactsList failed..."
30                 + " Server already has contacts ... "
31                 + "this test is meant to be run immediatly after the start of the server",
32                 "No contacts added yet.".equals(result));
33     }
34
35     @Test
36     public void test1_contactsPostMaria() {
37         // when
38         String result = CONTROLLER.contactsPost("0", "Maria Joao", "mail@mailhost.com", "02121990", "ANUAL", "100");
39         // then
40         assertTrue(
41                 "contactsPostMaria failed ..." + "\nGot:\n" + result + "\nExpected:\n" + "class ContactRep {\n"
42                         + "    id: 0\n" + "    registerDate: 10102018\n" + "    birthDate: 2121990\n"
43                         + "    email: mail@mailhost.com\n" + "    newsSubscriptionType: ANUAL\n"
44                         + "    name: Maria Joao\n" + "    newsSubscriptionWhen: 100\n" + "}",
45                 ("class ContactRep {\n" + "    id: 0\n" + "    registerDate: 10102018\n" + "    birthDate: 2121990\n"
46                         + "    email: mail@mailhost.com\n" + "    newsSubscriptionType: ANUAL\n"
47                         + "    name: Maria Joao\n" + "    newsSubscriptionWhen: 100\n" + "}").equals(result));
48     }
49
50     /*
51      * System.out.print of the string expected above ^
52      *
53      * class ContactRep { id: 0 registerDate: 10102018 birthDate: 2121990 email:
54      * mail@mailhost.com newsSubscriptionType: ANUAL name: Maria Joao
55      * newsSubscriptionWhen: 100 }
56      */
57
58     @Test
59     public void test2_contactsPostMariaAgain() {
60         // when
61         String result = CONTROLLER.contactsPost("0", "Maria Joao", "mail@mailhost.com", "02121990", "ANUAL", "100");
62         // then
63         assertTrue("contactsPostMariaAgain failed ..." + "\nGot:\n" + result + "\nExpected:\n"
64                 + "Conflict. Id already exists", ("Conflict. Id already exists.").equals(result));
65     }
66
67     @Test
68     public void test3_contactsIdPutMaria() {
69         // when
70         String result = CONTROLLER.contactsIdPut("Maria Joao 2", "mail2@mailhost2.com", "SEMANAL", "4", "0");
71         // then
72         assertTrue(
73                 "contactsPutMaria failed ..." + "\nGot:\n" + result + "\nExpected:\n" + "class ContactRep {\n"
74                         + "    id: 0\n" + "    registerDate: 10102018\n" + "    birthDate: 2121990\n"
75                         + "    email: mail2@mailhost2.com\n" + "    newsSubscriptionType: SEMANAL\n"
76                         + "    name: Maria Joao 2\n" + "    newsSubscriptionWhen: 4\n" + "}",
77                 ("class ContactRep {\n" + "    id: 0\n" + "    registerDate: 10102018\n" + "    birthDate: 2121990\n"
78                         + "    email: mail2@mailhost2.com\n" + "    newsSubscriptionType: SEMANAL\n"
79                         + "    name: Maria Joao 2\n" + "    newsSubscriptionWhen: 4\n" + "}").equals(result));
80     }
81
82     /*
83      * System.out.print of the string expected above ^
84      *
85      * class ContactRep {
86      *     id: 0
87      *     registerDate: 10102018
88      *     birthDate: 2121990
89      *     email: mail2@mailhost2.com
90      *     newsSubscriptionType: SEMANAL
91      *     name: Maria Joao 2
92      *     newsSubscriptionWhen: 4
93      * }
94      */
95
96     @Test
97     public void test4_contactsPostJoao() {
98         // when
99         String result = CONTROLLER.contactsPost("1", "Joao Maria", "mail@mailhost.com", "23111980", "ANUAL", "200");
100        // then
101        assertTrue(
102                "contactsPostJoao failed ..." + "\nGot:\n" + result + "\nExpected:\n" + "class ContactRep {\n"
103                        + "    id: 1\n" + "    registerDate: 11102018\n" + "    birthDate: 23111980\n"
104                        + "    email: mail@mailhost.com\n" + "    newsSubscriptionType: ANUAL\n"
105                        + "    name: Joao Maria\n" + "    newsSubscriptionWhen: 200\n" + "}",
106                ("class ContactRep {\n"
107                        + "    id: 1\n" + "    registerDate: 11102018\n" + "    birthDate: 23111980\n"
108                        + "    email: mail@mailhost.com\n" + "    newsSubscriptionType: ANUAL\n"
```

```
109                         + "    name: Joao Maria\n" + "    newsSubscriptionWhen: 200\n" + "}").equals(result));
110     }
111 
112     /*
113      * System.out.print of the string expected above ^
114      *
115      * class ContactRep {
116      *      id: 1
117      *      registerDate: 10102018
118      *      birthDate: 23111980
119      *      email: mail@mailhost.com
120      *      newsSubscriptionType: ANUAL
121      *      name: Joao Maria
122      *      newsSubscriptionWhen: 200
123      * }
124      */
125 
126     @Test
127     public void test5_contactsGetTestTwoContactsOnContactsList() {
128         // when
129         String result = CONTROLLER.contactsGet();
130         // then
131         assertTrue(
132                 "contactsGetTestTwoContactsOnContactsList failed ..." + "\nGot:\n" + result +
133                         "\nExpected:\n" + "class ContactListRep {\n" +
134                         "    contacts: [class ContactRep {\n" +
135                         "        id: 0\n" +
136                         "        registerDate: 10102018\n" +
137                         "        birthDate: 2121990\n" +
138                         "        email: mail2@mailhost2.com\n" +
139                         "        newsSubscriptionType: SEMANAL\n" +
140                         "        name: Maria Joao 2\n" +
141                         "        newsSubscriptionWhen: 4\n" +
142                         "    }, class ContactRep {\n" +
143                         "        id: 1\n" +
144                         "        registerDate: 11102018\n" +
145                         "        birthDate: 23111980\n" +
146                         "        email: mail@mailhost.com\n" +
147                         "        newsSubscriptionType: ANUAL\n" +
148                         "        name: Joao Maria\n" +
149                         "        newsSubscriptionWhen: 200\n" +
150                         "    }]\n" +
151                         "}",
152                 ("class ContactListRep {\n" +
153                 "    contacts: [class ContactRep {\n" +
154                 "        id: 0\n" +
155                 "        registerDate: 10102018\n" +
156                 "        birthDate: 2121990\n" +
157                 "        email: mail2@mailhost2.com\n" +
158                 "        newsSubscriptionType: SEMANAL\n" +
159                 "        name: Maria Joao 2\n" +
160                 "        newsSubscriptionWhen: 4\n" +
161                 "    }, class ContactRep {\n" +
162                 "        id: 1\n" +
163                 "        registerDate: 11102018\n" +
164                 "        birthDate: 23111980\n" +
165                 "        email: mail@mailhost.com\n" +
166                 "        newsSubscriptionType: ANUAL\n" +
167                 "        name: Joao Maria\n" +
168                 "        newsSubscriptionWhen: 200\n" +
169                 "    }]\n" +
170                 "}").equals(result));
171     }
172 
173     /*
174      * System.out.print of the string expected above ^
175      *
176      * class ContactListRep {
177      *      contacts: [class ContactRep {
178      *          id: 0
179      *          registerDate: 10102018
180      *          birthDate: 2121990
181      *          email: mail2@mailhost2.com
182      *          newsSubscriptionType: SEMANAL
183      *          name: Maria Joao 2
184      *          newsSubscriptionWhen: 4
185      *      }, class ContactRep {
186      *          id: 1
187      *          registerDate: 11102018
188      *          birthDate: 23111980
189      *          email: mail@mailhost.com
190      *          newsSubscriptionType: ANUAL
191      *          name: Joao Maria
192      *          newsSubscriptionWhen: 200
193      *      }]
194      * }
195      */
196 
197     @Test
198     public void test6_contactsPostInvalidMario() {
199         // when
200         // invalid id, mail and newsSubscriptionWhen
201         String result = CONTROLLER.contactsPost("-1", "Mario Maria", "mailATmailhost.com", "02121990", "ANUAL", "400");
202         // then
203         assertTrue(
204                 "contactsPostInvalidMario failed ..." + "\nGot:\n" + result + "\nExpected:\n" + "contactsPost - " + "Precondition1 failed.",
205                 ("contactsPost - " + "Precondition1 failed.").equals(result));
206     }
207 
208     @Test
209     public void test7_contactsIdPutInvalidMariaUpdate() {
210         // when
211         // invalid mail
212         String result = CONTROLLER.contactsIdPut("Maria Joao 3", "mailATmail.com", "ANUAL", "200", "0");
```

```
213         // then
214         assertTrue(
215                 "contactsPutInvalidMariaUpdate failed ..." + "\nGot:\n" + result + "\nExpected:\n" + "contactsIdPut - " + "Precondition1 failed.",
216                 ("contactsIdPut - " + "Precondition1 failed.").equals(result));
217     }
218 }
219
```

E.3 Contact Representation

```
1 /*
2  * ContactsWithLogic
3  * No description provided (generated by Swagger Codegen https://github.com/swagger-api/swagger-codegen)
4  *
5  * OpenAPI spec version: 1.0.0
6  *
7  *
8  * NOTE: This class is auto generated by the swagger code generator program.
9  * https://github.com/swagger-api/swagger-codegen.git
10  * Do not edit the class manually.
11  */
12
13 package io.headrest.client.model;
14
15 import java.util.Objects;
22
23 /**
24  * ContactRep
25  */
26 public class ContactRep {
27     @JsonProperty("id")
28     private Integer id = null;
29
30     @JsonProperty("registerDate")
31     private Integer registerDate = null;
32
33     @JsonProperty("birthDate")
34     private Integer birthDate = null;
35
36     @JsonProperty("email")
37     private String email = null;
38
39     /**
40      * Gets or Sets newsSubscriptionType
41      */
42     public enum NewsSubscriptionTypeEnum {
43         ANUAL("ANUAL"),
44
45         SEMANAL("SEMANAL"),
46
47         DIARIO("DIARIO");
48
49         private String value;
50
51         NewsSubscriptionTypeEnum(String value) {
52             this.value = value;
53         }
54
55         @JsonValue
56         public String getValue() {
57             return value;
58         }
59
60         @Override
61         public String toString() {
62             return String.valueOf(value);
63         }
64
65         @JsonCreator
66         public static NewsSubscriptionTypeEnum fromValue(String text) {
67             for (NewsSubscriptionTypeEnum b : NewsSubscriptionTypeEnum.values()) {
68                 if (String.valueOf(b.value).equals(text)) {
69                     return b;
70                 }
71             }
72             return null;
73         }
74     }
75
76     @JsonProperty("newsSubscriptionType")
77     private NewsSubscriptionTypeEnum newsSubscriptionType = null;
78
79     @JsonProperty("name")
80     private String name = null;
81
82     @JsonProperty("newsSubscriptionWhen")
83     private Integer newsSubscriptionWhen = null;
84
85     public ContactRep id(Integer id) {
86         this.id = id;
87         return this;
88     }
89
90     /**
91      * Get id
92      *
93      * @return id
94      **/
95     @ApiModelProperty(required = true, value = "")
96     public Integer getId() {
97         return id;
98     }
99
100     public void setId(Integer id) {
101         this.id = id;
102     }
103
104     public ContactRep registerDate(Integer registerDate) {
105         this.registerDate = registerDate;
106         return this;
107     }
108
109     /**
110      * Get registerDate
```

```
111      *
112      * @return registerDate
113      **/
114     @ApiModelProperty(required = true, value = "")
115     public Integer getRegisterDate() {
116         return registerDate;
117     }
118
119     public void setRegisterDate(Integer registerDate) {
120         this.registerDate = registerDate;
121     }
122
123     public ContactRep birthDate(Integer birthDate) {
124         this.birthDate = birthDate;
125         return this;
126     }
127
128     /**
129      * Get birthDate
130      *
131      * @return birthDate
132      **/
133     @ApiModelProperty(required = true, value = "")
134     public Integer getBirthDate() {
135         return birthDate;
136     }
137
138     public void setBirthDate(Integer birthDate) {
139         this.birthDate = birthDate;
140     }
141
142     public ContactRep email(String email) {
143         this.email = email;
144         return this;
145     }
146
147     /**
148      * Get email
149      *
150      * @return email
151      **/
152     @ApiModelProperty(required = true, value = "")
153     public String getEmail() {
154         return email;
155     }
156
157     public void setEmail(String email) {
158         this.email = email;
159     }
160
161     public ContactRep newsSubscriptionType(NewsSubscriptionTypeEnum newsSubscriptionType) {
162         this.newsSubscriptionType = newsSubscriptionType;
163         return this;
164     }
165
166     /**
167      * Get newsSubscriptionType
168      *
169      * @return newsSubscriptionType
170      **/
171     @ApiModelProperty(required = true, value = "")
172     public NewsSubscriptionTypeEnum getNewsSubscriptionType() {
173         return newsSubscriptionType;
174     }
175
176     public void setNewsSubscriptionType(NewsSubscriptionTypeEnum newsSubscriptionType) {
177         this.newsSubscriptionType = newsSubscriptionType;
178     }
179
180     public ContactRep name(String name) {
181         this.name = name;
182         return this;
183     }
184
185     /**
186      * Get name
187      *
188      * @return name
189      **/
190     @ApiModelProperty(required = true, value = "")
191     public String getName() {
192         return name;
193     }
194
195     public void setName(String name) {
196         this.name = name;
197     }
198
199     public ContactRep newsSubscriptionWhen(Integer newsSubscriptionWhen) {
200         this.newsSubscriptionWhen = newsSubscriptionWhen;
201         return this;
202     }
203
204     /**
205      * Get newsSubscriptionWhen
206      *
207      * @return newsSubscriptionWhen
208      **/
209     @ApiModelProperty(required = true, value = "")
210     public Integer getNewsSubscriptionWhen() {
211         return newsSubscriptionWhen;
212     }
213
214     public void setNewsSubscriptionWhen(Integer newsSubscriptionWhen) {
```

```
215         this.newsSubscriptionWhen = newsSubscriptionWhen;
216     }
217 
218     @Override
219     public boolean equals(java.lang.Object o) {
220         if (this == o) {
221             return true;
222         }
223         if (o == null || getClass() != o.getClass()) {
224             return false;
225         }
226         ContactRep contactRep = (ContactRep) o;
227         return Objects.equals(this.id, contactRep.id)
228                 && Objects.equals(this.registerDate, contactRep.registerDate)
229                 && Objects.equals(this.birthDate, contactRep.birthDate)
230                 && Objects.equals(this.email, contactRep.email)
231                 && Objects.equals(this.newsSubscriptionType, contactRep.newsSubscriptionType)
232                 && Objects.equals(this.name, contactRep.name)
233                 && Objects.equals(this.newsSubscriptionWhen, contactRep.newsSubscriptionWhen);
234     }
235 
236     @Override
237     public int hashCode() {
238         return Objects.hash(id, registerDate, birthDate, email, newsSubscriptionType, name, newsSubscriptionWhen);
239     }
240 
241     @Override
242     public String toString() {
243         StringBuilder sb = new StringBuilder();
244         sb.append("class ContactRep {\n");
245         sb.append("    id: ").append(toIndentedString(id)).append("\n");
246         sb.append("    registerDate: ").append(toIndentedString(registerDate)).append("\n");
247         sb.append("    birthDate: ").append(toIndentedString(birthDate)).append("\n");
248         sb.append("    email: ").append(toIndentedString(email)).append("\n");
249         sb.append("    newsSubscriptionType: ").append(toIndentedString(newsSubscriptionType)).append("\n");
250         sb.append("    name: ").append(toIndentedString(name)).append("\n");
251         sb.append("    newsSubscriptionWhen: ").append(toIndentedString(newsSubscriptionWhen)).append("\n");
252         sb.append("}");
253         return sb.toString();
254     }
255 
256     /**
257      * Convert the given object to string with each line indented by 4 spaces
258      * (except the first line).
259      */
260     private String toIndentedString(java.lang.Object o) {
261         if (o == null) {
262             return "null";
263         }
264         return o.toString().replace("\n", "\n    ");
265     }
266 
267     /**
268      * Method used to validate an instance of this type
269      *
270      * @param instance
271      *            the instance
272      */
273     public static boolean validate(ContactRep instance) {
274         return instance.getNewsSubscriptionWhen() > 0 && instance.getBirthDate() < instance.getRegisterDate()
275                 && (!(Objects.equals(instance.getNewsSubscriptionType().value, "ANUAL"))
276                         || instance.getNewsSubscriptionWhen() <= 365)
277                 && (!(Objects.equals(instance.getNewsSubscriptionType().value, "SEMANAL"))
278                         || instance.getNewsSubscriptionWhen() <= 7)
279                 && (!(Objects.equals(instance.getNewsSubscriptionType().value, "DIARIO"))
280                         || instance.getNewsSubscriptionWhen() <= 24)
281                 && instance.getEmail().contains("@") && instance.getName().length() > 2;
282     }
283 }
284 
```

E.4 Client API

```
1 package io.headrest.client.api;
2
3 import java.util.ArrayList;
20
21 public class DefaultApi {
22     private ApiClient apiClient;
23
24     public DefaultApi() {
25         this(Configuration.getDefaultApiClient());
26     }
27
28     public DefaultApi(ApiClient apiClient) {
29         this.apiClient = apiClient;
30     }
31
32     public ApiClient getApiClient() {
33         return apiClient;
34     }
35
36     public void setApiClient(ApiClient apiClient) {
37         this.apiClient = apiClient;
38     }
39
40     /**
41      *
42      *
43      * @return ContactListRep
44      * @throws ApiException
45      *             if fails to make API call
46      */
47     public ContactListRep contactsGet() throws ApiException {
48         Object localVarPostBody = null;
49
50         // create path and map variables
51         String localVarPath = "/contacts".replaceAll("\\{format\\}", "json");
52
53         // query params
54         List<Pair> localVarQueryParams = new ArrayList<Pair>();
55         // headers params
56         Map<String, String> localVarHeaderParams = new HashMap<String, String>();
57         // form params
58         Map<String, Object> localVarFormParams = new HashMap<String, Object>();
59
60         final String[] localVarAccepts = {
61
62         };
63         final String localVarAccept = apiClient.selectHeaderAccept(localVarAccepts);
64
65         final String[] localVarContentTypes = {
66
67         };
68         final String localVarContentType = apiClient.selectHeaderContentType(localVarContentTypes);
69
70         String[] localVarAuthNames = new String[] {};
71
72         // if (!contactsGetRequiredSuccessPrecondition())
73         // throw new ApiException();
74
75         GenericType<ContactListRep> localVarReturnType = new GenericType<ContactListRep>() {
76         };
77         return apiClient.invokeAPI(localVarPath, "GET", localVarQueryParams, localVarPostBody, localVarHeaderParams,
78                 localVarFormParams, localVarAccept, localVarContentType, localVarAuthNames, localVarReturnType);
79
80     }
81
82     /**
83      *
84      *
85      * @param body
86      *            (required)
87      * @param id
88      *            (required)
89      * @return ContactRep
90      * @throws ApiException
91      *             if fails to make API call
92      */
93     public ContactRep contactsIdPut(ContactPutData body, Integer id) throws ApiException {
94         Object localVarPostBody = body;
95
96         // verify the required parameter 'body' is set
97         if (body == null) {
98             throw new ApiException(400, "Missing the required parameter 'body' when calling contactsIdPut");
99         }
100
101         // verify the required parameter 'id' is set
102         if (id == null) {
103             throw new ApiException(400, "Missing the required parameter 'id' when calling contactsIdPut");
104         }
105
106         // create path and map variables
107         String localVarPath = "/contacts/{id}".replaceAll("\\{format\\}", "json").replaceAll("\\{" + "id" + "\\}",
108                 apiClient.escapeString(id.toString()));
109
110         // query params
111         List<Pair> localVarQueryParams = new ArrayList<Pair>();
112         // headers params
113         Map<String, String> localVarHeaderParams = new HashMap<String, String>();
114         // form params
115         Map<String, Object> localVarFormParams = new HashMap<String, Object>();
116
117         final String[] localVarAccepts = {
118
119         };
120         final String localVarAccept = apiClient.selectHeaderAccept(localVarAccepts);
```

```
121
122         final String[] localVarContentTypes = {
123
124         };
125         final String localVarContentType = apiClient.selectHeaderContentType(localVarContentTypes);
126
127         String[] localVarAuthNames = new String[] {};
128
129         // if (!contactsIdPutRequiredSuccessPrecondition(body, id))
130         // throw new ApiException();
131
132         GenericType<ContactRep> localVarReturnType = new GenericType<ContactRep>() {
133         };
134         return apiClient.invokeAPI(localVarPath, "PUT", localVarQueryParams, localVarPostBody, localVarHeaderParams,
135                 localVarFormParams, localVarAccept, localVarContentType, localVarAuthNames, localVarReturnType);
136
137     }
138
139     /**
140      *
141      *
142      * @param body
143      *            (required)
144      * @return ContactRep
145      * @throws ApiException
146      *             if fails to make API call
147      */
148     public ContactRep contactsPost(ContactPostData body) throws ApiException {
149         Object localVarPostBody = body;
150
151         // verify the required parameter 'body' is set
152         if (body == null) {
153             throw new ApiException(400, "Missing the required parameter 'body' when calling contactsPost");
154         }
155
156         // create path and map variables
157         String localVarPath = "/contacts".replaceAll("\\{format\\}", "json");
158
159         // query params
160         List<Pair> localVarQueryParams = new ArrayList<Pair>();
161         // headers params
162         Map<String, String> localVarHeaderParams = new HashMap<String, String>();
163         // form params
164         Map<String, Object> localVarFormParams = new HashMap<String, Object>();
165
166         final String[] localVarAccepts = {
167
168         };
169         final String localVarAccept = apiClient.selectHeaderAccept(localVarAccepts);
170
171         final String[] localVarContentTypes = {
172
173         };
174         final String localVarContentType = apiClient.selectHeaderContentType(localVarContentTypes);
175
176         String[] localVarAuthNames = new String[] {};
177
178         // if (!contactsPostRequiredSuccessPrecondition(body))
179         // throw new ApiException();
180
181         GenericType<ContactRep> localVarReturnType = new GenericType<ContactRep>() {
182         };
183         return apiClient.invokeAPI(localVarPath, "POST", localVarQueryParams, localVarPostBody, localVarHeaderParams,
184                 localVarFormParams, localVarAccept, localVarContentType, localVarAuthNames, localVarReturnType);
185
186     }
187
188     /**
189      * @returns BooleanValidation with true
190      */
191     public BooleanValidation contactsGetRequiredSuccessPrecondition() {
192         boolean pre1 = true;
193         return new BooleanValidation(pre1); // will always be valid
194     }
195
196     /**
197      * @returns BooleanValidation with true if the success precondition is
198      *          verified, JustifiedInvalid with reason otherwise
199      */
200     public BooleanValidation contactsIdPutRequiredSuccessPrecondition(ContactPutData body, Integer id) {
201         boolean pre1 = ContactPutData.validate(body);
202         return pre1 ? new BooleanValidation(true) : new JustifiedInvalid(getReasonString(pre1) + " failed.");
203     }
204
205     /**
206      * @returns BooleanValidation with true if the success precondition is
207      *          verified, JustifiedInvalid with reason otherwise
208      */
209     public BooleanValidation contactsPostRequiredSuccessPrecondition(ContactPostData body) {
210         boolean pre1 = ContactPostData.validate(body);
211         return pre1 ? new BooleanValidation(true) : new JustifiedInvalid(getReasonString(pre1) + " failed.");
212     }
213
214     /**
215      * Returns the booleans that failed in a prepared string
216      *
217      * @param booleans
218      *            the list of booleans
219      * @return the prepared string
220      */
221     public String getReasonString(boolean... booleans) {
222         StringBuilder sb = new StringBuilder("");
223         for (int i = 0; i < booleans.length; i++)
224             if (!booleans[i])
```

```
225                 if (sb.length() == 0)
226                     sb.append("Precondition" + (i + 1));
227                 else
228                     sb.append(" && Precondition" + (i + 1));
229         return sb.toString();
230     }
231 
232 }
233 
```

E.5 Boolean Validation Class

```
1 package io.headrest.client.api.validation;
2
3 /**
4  * Basic boolean validation
5  */
6 public class BooleanValidation {
7
8     /**
9      * If is valid
10     */
11    private boolean valid;
12
13    /**
14     * Construct
15     * @param valid if is valid
16     */
17    public BooleanValidation(boolean valid) {
18        this.valid = valid;
19    }
20
21    /**
22     * @return true if valid, false otherwise
23     */
24    public boolean isValid() {
25        return valid;
26    }
27 }
28
```

E.6 Justified Invalid Class

```
1 package io.headrest.client.api.validation;
2
3 /**
4  * Justified invalid with reasoning
5  */
6 public class JustifiedInvalid extends BooleanValidation{
7
8     /**
9      * The reason
10     */
11    private String reason;
12
13    /**
14     * Construct
15     * @param reason the reason
16     */
17    public JustifiedInvalid(String reason) {
18        super(false);
19        this.reason = reason;
20    }
21
22    /**
23     * @return the reason
24     */
25    public String getReason() {
26        return reason;
27    }
28
29 }
30
```

Appendix F

HeadREST Codegen Form

HeadRest Codegen Form

HeadRest Codegen is a tool for generating server stubs and client SDKs for a RESTful API specified with HeadRest.

In what follows, we will conduct a survey regarding your opinion on the usage of the tool, done previously.

Participation in this survey is voluntary. Participants may withdraw by not submitting the survey at any time. Depending on your answers, this survey takes approximately 5 to 15 minutes to complete.

No identifying information will be collected and all participants shall remain anonymous. Collected data are planned to be published along with a MSc thesis.

By clicking through the consent statement and submitting the completed survey, individuals are indicating their willingness to participate.

We really appreciate your time and input.

* Required

1. **Accept and continue** *

   Mark only one oval.

   ◯ Yes

   ◯ No *After the last question in this section, stop filling out this form.*

2. **Password** *

   ____

Context questions

Here we will question you about your past experience with RESTful APIs and tools

3. **Did you know about RESTful APIs before the experiment ?** *

   Mark only one oval.

   ◯ Yes

   ◯ No

4. **Did you know the Open API Specification (OAS/Swagger) language ?** *

   Mark only one oval.

   ◯ Yes

   ◯ No

5. **Did you know the HEADREST (HR) language ?** *

   Mark only one oval.

   ◯ Yes

   ◯ No

6. **Have you used Swagger Codegen before ?** *
   Mark only one oval.
   - Yes
   - No

Languages questions

Here we ask a series of questions relative to the languages used

7. **How easy was OAS specification to understand ?** *
   Mark only one oval.

	1	2	3	4	5	6	
very hard							very easy

8. **How easy was HR specification to understand ?** *
   Mark only one oval.

	1	2	3	4	5	6	
very hard							very easy

Tools questions

Here we ask a series of questions relative to the tools used

9. **How easy was using the Swagger-Codegen ?** *
   Mark only one oval.

	1	2	3	4	5	6	
very hard							very easy

10. **How easy was using the HEADREST-Codegen ?** *
    Mark only one oval.

	1	2	3	4	5	6	
very hard							very easy

11. **How useful was Swagger-Codegen generated code ?** *
    Mark only one oval.

1	2	3	4	5	6	
						very useful

12. **How useful was HEADREST-Codegen generated code ?** *
Mark only one oval.

	1	2	3	4	5	6	
	⬭	⬭	⬭	⬭	⬭	⬭	very useful

Comparison

Here we compare both languages and tools

13. **Which tool was the most useful ?** *
Mark only one oval.

⬭ HR
⬭ OAS

14. **How easy was it to understand the OAS generated code ?** *
Mark only one oval.

	1	2	3	4	5	6	
very hard	⬭	⬭	⬭	⬭	⬭	⬭	very easy

15. **How easy was it to understand the HR generated code ?** *
Mark only one oval.

	1	2	3	4	5	6	
very hard	⬭	⬭	⬭	⬭	⬭	⬭	very easy

16. **Which client ended up with more quality ?** *
Mark only one oval.

	1	2	3	4	5	6	
OAS	⬭	⬭	⬭	⬭	⬭	⬭	HR

Additional remarks

Here we ask for any additional remarks relative to the experiment

17. **Remarks** *

Powered by

Bibliography

[1] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifier (URI): Generic syntax. RFC 3986, Network Working Group, 2005. https://tools.ietf.org/html/rfc3986.

[2] Lorenzo Bettini. *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition*. Packt Publishing, 2nd edition, 2016.

[3] Gavin M. Bierman, Andrew D. Gordon, Cătălin Hriţcu, and David Langworthy. Semantic subtyping with an SMT solver. *J. Funct. Program.*, 22(1):31–105, 2012.

[4] Bill Burke. *RESTful Java with Jax-RS*. O'Reilly Media, Inc., 1st edition, 2009.

[5] Luca Cardelli and John C. Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1:3–48, 1991.

[6] F. Ferreira, T. Santos, F. Martins, A. Lopes, and V. Vasconcelos. Especificação de Interfaces Aplicacionais REST. In *Actas do 9º Encontro Nacional de Informática, INFORUM 2017, Aveiro, Portugal*, 2017.

[7] Fábio Ferreira. Automatic Tests Generation for RESTful APIs. Master's thesis, Faculdade de Cilencias da Universidade de Lisboa, 2017.

[8] Roy T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[9] Roy T. Fielding and Richard N. Taylor. Principled design of the modern Web architecture. In *ICSE*, pages 407–416, 2000.

[10] Ira R. Forman and Nate Forman. *Java Reflection in Action (In Action Series)*. Manning Publications Co., Greenwich, CT, USA, 2004.

[11] J. Gregorio, R. Fielding, M. Hadley, M. Nottingham, and D. Orchard. URI template. RFC 6570, Internet Engineering Task Force, 2012. https://tools.ietf.org/html/rfc6570.

[12] Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 26-28, 2010, Hagenberg, Austria*, pages 13–24, 2010.

[13] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *POPL*, pages 131–142. ACM, 1991.

[14] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[15] Markus Lanthaler and Christian Guetl. Hydra: A vocabulary for hypermedia-driven web apis. In *Proceedings of the WWW2013 Workshop on Linked Data on the Web, Rio de Janeiro, Brazil, 14 May, 2013*, 2013.

[16] A. Monnox. *Rapid J2EE Development: An Adaptive Foundation for Enterprise Applications*. Hewlett-Packard professional books. Prentice Hall PTR, 2005.

[17] Richard Warburton. *Java 8 Lambdas: Pragmatic Functional Programming*. O'Reilly Media, Inc., 1 edition, 2014.

[18] Jim Webber, Savas Parastatidis, and Ian Robinson. *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly Media, Inc., 1st edition, 2010.