A simple, lightweight framework for testing RESTful services with TTCN-3

Theofanis Vassiliou-Gioles
Technische Universität Berlin
Weizenbaum Institut
Berlin, Germany
Email: vassiliou-gioles@tu-berlin.de
0000-0002-6990-242X

***Abstract*—Micro-service architecture has become a standard software architecture style, with loosely coupled, specified, and implemented services, owned by small teams and independently deployable. TTCN-3, as test specification and implementation language, allows an easy and efficient description of complex distributed test behavior and seems to be a natural fit to test micro-services. TTCN-3 is independent of the underlying communication and data technology, which is strength and weakness at the same time. While tools and frameworks are supporting micro-service developers to abstract from the underlying data, implementation, and communication technology, this support has to be modeled in a TTCN-3 based test system, manually. This paper discusses the concepts of a TTCN-3 framework on the four different levels of the Richardson-Maturity Model, introducing support for testing hypermedia controls, HATEOAS, proposes a TTCN-3 framework and open-source implementation to realize them and demonstrates its application by a concrete example.**

***Index Terms*—TTCN-3, Software testing, test automation, micro service, RESTful API, web service**

I. Introduction

Cloud computing, the availability of various computing resources over the internet, has become a standard way of using IT resources. In the same way, web services, i.e., services that provide machine-readable documents via the internet, primarily over HTTP [1], have gained attraction and are the dominating components in the software application space. This paper proposes a TTCN-3 framework for testing RESTful web services to support testers, and quality assurance engineers to develop test strategies beyond functional testing.

The World Wide Web Consortium (W3C) defines web services as "[...] a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards." [2]

While the W3C emphasizes on WSDL (Web Services Description Language) [3] and SOAP (Simple Object Access Protocol)) [4] technologies modern interpretations relax the interface description (or description protocols) beyond WSDL, for example, to the usage of OpenAPI Specification (OAS) [5], the successor of Swagger [6][1]

This machine-to-machine interaction over the network has led to extensive web-service infrastructures, and in more recent times, to the emerging of micro-service based software architecture, with REST (Representational state transfer) [7] being one of its prominent software architecture paradigms.[2]

"We can identify two major classes of Web services:

- REST-compliant Web services, in which the primary purpose of the service is to manipulate XML representations of Web resources using a uniform set of "stateless" operations; and
- arbitrary Web services, in which the service may expose an arbitrary set of operations." [2]

RESTful services use a variety of different data representations with XML and JSON [8] being the most prominent representatives.

Various mappings for data definition languages like ASN.1 [9], XML [10] or JSON [11] have been defined and standardized for TTCN-3 [12]. While these mappings define the data structures, we propose on how to map the functional specifics of RESTful web services to TTCN-3 and propose a framework. While the terms REST and RESTful are widely used for mainly any kind of HTTP based Application Programming Interface (API), it becomes apparent that there are different understandings of what a RESTful API is. REST, just like web-services it not a technique but an architectural style. The Richardson Maturity Model (RMM) evaluates a REST-API with respect to its quality. It defines four levels, starting from level 0, using HTTP as the transport model, only but no other (standardized) web mechanism. The highest level, level 3, adds hypermedia controls, or HATEOAS (Hypertext As The Engine Of Application State) as introduced by [7].

The four different levels of RMM are defined as follows:

- level 0 - Using HTTP as transport protocol only. At this level, HTTP is being used to transport data (formatted as XML documents, JSON encoded, or in any other format)

[1]The OpenAPI Initiative has adopted the Swagger 2.0 specification. Today's version is v.3.0.3.

[2]In this paper we use web service and micro services synonoumesly

978-1-7281-8915-4/20/$31.00 ©2020 IEEE
DOI 10.1109/QRS-C51114.2020.00089

to and from a server. No dedicated web technologies are being used.
- level 1 - Applications on this maturity level use specific resources in the form of URIs (Uniform Resource Identifiers) to manipulate data at or retrieve data from servers. As a result, information is migrating from the transported data (i.e., the message body of an HTTP request/response) to the HTTP level.
- level 2 - Instead of using individual HTTP verbs like GET or POST to merely transport documents, HTTP verbs are now being used as intended by the HTTP specification and differentiating safe data retrieval and ”unsafe” data or resource manipulation.
- level 3 - Finally, the highest level of the RMM, introduces hypermedia as a way to navigate through an interface and application. [13] claims that RESTful APIs have a level 3 maturity as a precondition. However, this is often ignored in practice, when naming APIs RESTful.

As level-3 REST APIs have far less practical applications than level 1 or 2, we will first focus on REST APIs up to level 2 of the RMM. Finally, we will give an outlook on how RMM level-3 RESTful APIs can be tested for their specific RMM-level-3 properties.

II. Testing web services

In industry, two different development approaches to expose API functionality via RESTful APIs can be identified that require a specific *web-service centric* test approach.

- First, a remotization (bottom-up) approach where locally developed functionality is made accessible remotely.
- Second, a webservice-native (top-down) approach where resources and functionality are first designed via (remotely accessible) API specifications and being implemented in a second step.

In both approaches, the remotely accessible APIs are being specified and serve as the test basis. A specific web-service-centric test approach emphasizes but does not limit, the test functionality on the specific properties of the exposed resources and operations. It assumes that functional testing has been performed locally, for example, via unit testing. A web-service-centric test approach should abstract from the actual transport semantics and should focus on the data and operation-specific aspects. In addition it should enable fine-granular access to transport-related elements, if required.[3]

We will first introduce a TTCN-3 framework capable of addressing the properties of an RMM-level-2 REST API by supporting HTTP, URIs, and different HTTP operations. Build on this, we extend the framework to support RMM-level-3 REST APIs in the following section.

As outlined earlier web-service is not a particular technology but an architecture style that can be realized by using various technologies.

[3]Examples of transport-related elements are `Header-Fields` or URL query parameters.

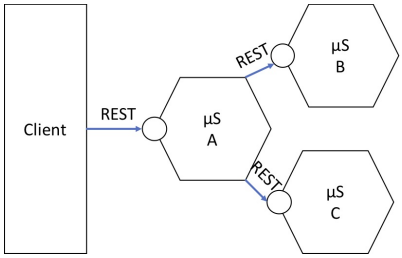


Fig. 1: Abstract webservice architecture with three webservices A-C

Fig. 1 shows a simplified web-service architecture with a CLIENT and three web services (A, B, and C). The respective API of each service expose their service via a RESTful API.

The presented framework focuses on web services that are using the following web service protocol stack and technologies

- HTTP as application transport protocol at the REST API. Other application transport protocols like FTP, SMTP, etc. are not considered.
- JSON as messaging protocol. XML is also used frequently as an alternative. While not considered in this paper, we plan to extend the proposal also to cover XML as messaging protocol. JSON encoded data is embedded in the HTTP message body at a REST API.
- OpenAPI as description protocol. The description protocol defines the public API to the webservice. In this paper, we consider web service APIs to be specified using a description protocol. While technically not part of the TTCN-3 framework OpenAPI based web services have been used to design the presented TTCN-3 framework. Other description protocols like protocol-buffer [14] have influenced the design.

With the selection of HTTP as the transport protocol, HTTP elements play an essential role in the service description.

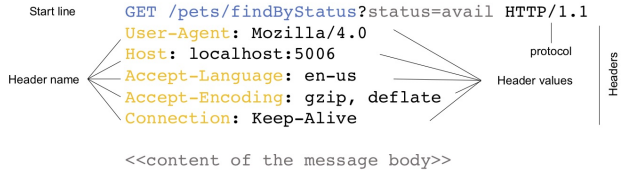
- HTTP method or verb:
- URL: describing the endpoint and parameters of a web service function
- HTTP Header-Fields: HTTP-header fields are identified by their name and can transport additional information for the usage, such as authentication or data format selection as a header value
- HTTP-message body data format: As web services are targeted for machine-to-machine-communication, machine-processable data formats are transported. While not limited to widely accepted formats are JSON or XML.

Fig. 2a and Fig. 2b display the anatomy and essential elements of a HTTP message for a hypothetical webservice that is available via `https` at `localhost` at port `5006`.

III. Testing with TTCN-3

TTCN-3, the Testing and Test Control Notation, is the test specification and implementation language defined by the

(a) Anatomy of URL including method for a HTTP message



(b) Elements of a HTTP message

Fig. 2: HTTP message

European Telecommunications Standards Institute (ETSI) for the precise definition of test procedures for black-box testing of distributed systems. It fits well with the test requirements of RESTful-based applications with possibilities to efficiently describe complex distributed test behavior through parallel stimuli and responses.

One of the benefits of TTCN-3 is that it enables the specifications of tests in a language and platform-independent manner, thus following the micro-service paradigm to choose the technology that fits best for the implementation of a particular. Our goal is to consider a TTCN-3 test system as another micro-services, with highly specific targets.

A. Architecture

According to the different RMM levels, we have structured or TTCN-3 framework for testing RESTful API based web services in three levels.

- RMM-0/1 support - We define message structure together with port definitions and operations that support the sending and receiving of simple, HTTP based messages. As basic support, we consider support for GET messages, the ability to specify the URL, and the ability to handle information in header fields for sending and receiving efficiently.
- RMM-2 support - RESTful API on this maturity level make HTTP specific use of HTTP operations. Our framework adds supporting functionality for the different HTTP operations and some testing functionality to tests on HTTP specific properties, like the idempotency of operations like GET, or PUT.
- RMM-3 support - In addition to the features already introduced support RMM 3 support introduced some test functionality to work with the hypermedia aspects of RMM 3. Framework components of all levels can be combined.

B. Core Functionality for REST services RMM level 0/1

As core functionality of the TTCN-3 framework a TTCN-3 module `Lib_HTTP` together with HTTP-to-TTCN-3 mapping conventions have been defined. A TTCN-3 module supporting this TTCN-3 framework should indicate `encoding "RESTful"` for generic HTTP support and flexible HTTP message body encoding, `encoding "RESTful/json"` for JSON message body encoding or `encoding "RESTful/XML"` for XML encoded message bodies.[4] as module attributes.

`"RESTful"` defines a HTTP request as a **`record`** with fields for message content, like query parameters, HTTP header fields or message body, for HTTP requests that support message bodies. For the specification of HTTP operations, URIs and associated resources TTCN-3 **`encode`** and **`variant`** attributes are being used.

Listing 1 and Listing 2 show an example for a HTTP request and the type specification for a generic HTTP response taken from `Lib_HTTP`, respectively.

```
1   type record ReadField {
2       string channelId,
3       string fieldId,
4       string api_key optional,
5       integer results optional,
6       string _start  optional,
7       string traceId optional
8   }
9   with {
10      encode "REST/get";
11      variant "path:/channels/{channelId}/fields
    /{fieldId}.json";
12      encode (api_key) "query";
13      encode (results) "query";
14      encode (_start) "query:start";
15      encode (traceId) "header:Trace-ID";
16  }
```

Listing 1: A templated and parameterized GET operation

```
1   type record HTTPResponse {
2       StatusLine statusLine, set of Header
    headers, Body body optional
3   }
4   type record StatusLine {
5       integer statusCode, string reasonPhrase
    optional
6   }
7   type record Body {
8       string messageBodyTxt optional,
    octetstring messageBodyRaw optional
9   }
10  type record Header {
11      string name, string val
12  }
```

Listing 2: A generic HTTP response as defined by `Lib_HTTP`

Listing 1 shows support for different HTTP operations (line 10), URI templating (line 11), support of non-hierarchical parametrization via query components (lines 12-14), support of header fields (line 15) and rewriting of templates, queries, and headers (lines 14-15).

As the third component of basic HTTP support, we bind the execution of test cases to a specific webservice implementation at runtime, via parametrization of the **`map`** operation.

A typical runtime binding can be seen in Listing 3 where the local port `service` is mapped to the system port `server`

[4]The current implementation only supports JSON message body encoding. However we are planning to extend the support to support also XML encoding. Therefore the `encoding "RESTful/XML"` has been reserved

. Also, the `baseUrl`, which consists of `scheme`, `host`, optional `port`, and `basepath` is being provided to the test system implementation. For this a predefined data type `RESTAPIconfig` has been defined in *Lib_HTTP*.

```
module TestSuite {
    import from Lib_HTTP all;
...
type port HTTPPort message {
    in HTTPResponse;
    out ReadField;
    map param (RESTAPIconfig config);
}

type component RESTComponent {
    port HTTPPort service; }
type component TSRestService {
    port HTTPPort server; }

testcase test_ReadSingleData() runs on RESTComponent
     system TSRestService {
        map(mtc:service, system:server) param (
    RESTAPIconfig:{
            baseUrl := BASE_URL,
            authorization := omit
        });
...
}}
```

Listing 3: Runtime binding of abstract test case to concrete webservice implementation

C. Extended functionality for REST services at RMM level 2

REST-based applications with an RMM level 2 make use of HTTP operations beyond pure GET or POST functionality, like PUT and DELETE, to follow the CRUD (Create, Read, Update, Delete) principle. In addition to these operations, the presented framework also supports HEAD, OPTIONS, PATCH, etc. These extensions are supported in the same way as the GET operation with the usage of the **`encode`** `" "` attribute.

HTTP Operation	Idempotent	Safe	encode attribute
GET	yes	yes	`encode "REST/get"`
HEAD	yes	yes	`encode "REST/head"`
OPTIONS	yes	yes	`encode "REST/options"`
PUT	yes	no	`encode "REST/put"`
DELETE	yes	no	`encode "REST/delete"`
POST	no	no	`encode "REST/post"`
PATCH	no	no	`encode "REST/patch"`

TABLE I: Overview of currently supported HTTP operation and their classification with respect to idempotency and safeness [1]

Typical applications on this RMM level use a protocol description, like an OpenAPI specification [5], to define the RESTful API. The TTCN-3 framework provides a mapping of a RESTful API and a set of helping functionality and separates them into three modules:

- Operations - specifies the API's endpoints together with their HTTP operations
- Models - specifies the data model and the data structures that can be used in the operations and the responses
- Components - specifies the test system architecture and binds the operations and responses to the ports

While the separation into modules is technically unnecessary, it's relation to RESTful protocol descriptions terminology like in [5], and related tooling, the separation with appropriate module naming has shown to be helpful. For each module, the **`encode`** `"RESTful"` attributes specifies the compliance of the mapping to this RESTful TTCN-3 mapping. For the support of OpenAPI v2 specifications, a supporting library *`openapiv2`* is provided. The library contains OpenAPI specific types like *`string`*, **`double`** or *`date`*.

As an example to illustrate the mapping concepts we will use the Petstore webservice as introduced in [15].

```
/pet/findByStatus:
  get:
    summary: "Finds Pets by status"
    operationId: "findPetsByStatus"
    produces:
    - "application/json"
    parameters:
    - name: "status"
      in: "query"
      description: "Status values that need to be
  considered for filter"
      required: true
      type: "array"
      items:
        type: "string"
        enum:
        - "available"
        - "pending"
        - "sold"
        default: "available"
      collectionFormat: "multi"
    responses:
      200:
        description: "successful operation"
        schema:
          type: "array"
          items:
            $ref: "#/definitions/Pet"
      400:
        description: "Invalid status value"
```

Listing 4: A YAML based operation definition in Swagger. [15], shortened

1) Operations: In HTTP based RESTful API specifications an operations is represented by an endpoint together with the HTTP-operation. Fig. 2a displays this as *path* and *operation* respectively.

Listing 4 shows a typical operation specification using the YAML format of the OpenAPI specification v2 (OASv2). Fig. 2b display one possible resulting status line as defined by this operation.

The TTCN-3 framework maps each REST operation, and its response to two respective TTCN-3 structures, the first structure, a **`record`**, follows the mapping as specified in III-B. The second TTCN-3 structure, a **`union`**, summarizes the information about the different specified HTTP responses for the request.

Listing 5 shows the elements of the proposed TTCN-3 mapping. Each REST operation is modelled as a TTCN-3 **`record`**. The specific REST properties are specified using

TTCN-3 attributes. **encode** "REST/get" identifies this record as a GET operation for the endpoint as specified by **variant** "path: /pet/findByStatus". As the operation has only one array of strings parameters the resulting record contains one mandatory element of type StatusElement (a type restricted *string* defined in the models section.). **encode** (status) "query" identifies the status field as a query parameter to the GET operation.

```
type record FindPetsByStatus {
    record of StatusEnum status
}
with {
    encode "REST/get";
    variant "path: /pet/findByStatus";
    encode (status) "query"
}
```

Listing 5: A simple parametrized GET operation

```
// 200 - successful operation
// 400 - Invalid status value
type union FindPetsByStatusResponse {
    record of Pet _200,
    noContext _400
}
with {
    encode "REST/getResponse";
    encode (_200) "body/json"
}
```

Listing 6: A simple TTCN-3 response

An operation also specifies responses, with potentially specific return codes. Listing 6 shows the mapping of the HTTP response to the respective TTCN-3 type. The **union** is enriched with encoding information for the mapping of different status codes and potential response bodies. For each response code, an alternative shall be specified (lines 4 - 5). Each alternative field references to one response code. The response code 200 maps to the field name _200, as identifier naming restrictions in TTCN-3 prohibiting number literals as identifiers. The example in Listing 4 defines two responses, a 200 and a 400. The content of the 200 is specified as an array of pets (a data structure that is referenced and defined somewhere later in the YAML file). The content of the later one is not further defined; we call this status unspecified. The supporting library *openapiv2* provides the predefined type noContext, a generic HTTP response to handle this type of generic, potentially underspecified response.

Send and receive templates can be specified to describe test behaviour and awaited responses.

```
module PetStoreTemplates {
 import from Operations all;
 template FindPetsByStatus findPetByStatus := {
    status := {"available", "sold"}}
 template FindPetsByStatusResponse
    findPetByStatusResponseSucc := { _200 := ?}
 template FindPetsByStatusResponse
    findPetByStatusResponseInv := { _400 := ?}

}
```

Listing 7: Simple templates for GET request and corresponding responses

When being sending findPetByStatus to the webserice this would be encoded to a HTTP request with the status line GET http://localhost/v2/pet/findByStatus?status=available&status=sold HTTP/1.1

2) Models: Different to unstructured parameters that can be encoded in the start line of an HTTP request (e.g., as query parameters), the message body carries (encoded) structured information. JSON and XML are widely used as encodings. Their usage is indicated as MIME type in the HTTP Content-Type header field. The current version of the framework supports JSON encoded message bodies in requests and responses. Listing 6 introduced the usage of encoded message bodies in responses, Listing 8 visualises the usage in requests. For sending or receiving data structures (Pet) are being referenced that have been specified the OpenAPI specification of the web services. Listing 9 displays a possible mapping of the structured object to TTCN-3. Our predefined TTCN-3 library *openapiv2* provides predefined OpenAPIv2 data types (e.g. *string* or *int64*).

```
type record AddPet {
    Pet body
}
with {
    encode "REST/post";
    variant "path: /pet";
    encode (body) "body/json"
}
```

Listing 8: A POST request operation with message body, JSON encoded

```
import from openapiv2 all;

type set Pet {
    int64 id optional,
    Category category optional,
    string name,
    record of string photoUrls,
    record of Tag tags optional,
    string status optional
}
type set Category {
    int64 id, string name
}
type set Tag {
    int64 id, string name
}
type string StatusEnum ("available", "pending",
"sold");
```

Listing 9: Referenced TTCN-3 data structures

```
Pet:
    type: "object"
    required:
    - "name"
    - "photoUrls"
    properties:
      id:
        type: "integer"
        format: "int64"
      category:
        $ref: "#/definitions/Category"
      name:
        type: "string"
      photoUrls:
```

```
        type: "array"
        items:
          type: "string"
      tags:
        type: "array"
        items:
          $ref: "#/definitions/Tag"
      status:
        type: "string"
        enum:
        - "available"
        - "pending"
        - "sold"
```

Listing 10: OpenAPI specification of structured object

In our framework and reference implementation, we have applied a simplified mapping from JSON encoded OpenAPI object specifications to TTCN-3, which we call POTO (Plain Old TTCN-3 Objects). We consider this mapping a subset of [11], which defines a full TTCN-3 to JSON. For the focus of testing RESTful APIs that carry JSON encoded structured data, supporting a subset of the TTCN-3 to JSON mapping has been proved efficient. Other than POTO encoding rules can be referenced by using the respective `encoding` attribute values.

3) Components: In TTCN-3 **port**s are being used for communication with the system under test. Test components have **port**s, and the actual test behavior is being executed on components. As third element of the RESTful API TTCN-3 framework **port**s and **component**s are being defined as introduced by Listing 3.

```
module PetStoreTestSuite {
    import from Lib_HTTP, openapiv2 all;
    import from Models, Operations, Components all;

    // Configuration
    ...
    // Testdata
    template FindPetsByStatus findPetByStatus := {
        status := {"available", "sold"} }

    testcase SendFindByStatus()
        runs on PetStoreClient system PetStoreServer
     {
        map(mtc:service, system:server) param (
    configNoAuth);

        timer t; t.start(5.0);
        service.send(findPetByStatus);
        alt {
        [] service.receive(FindPetsByStatusResponse
    :{_200 := ?}) {
                t.stop; setverdict(pass);
            }
        [] service.receive(FindPetsByStatusResponse
    :{_400 := ?}) {
                t.stop; setverdict(inconc);
            }
        [] service.receive(HTTPResponse:?) {
                t.stop; setverdict(inconc);
            }
        [] service.receive {
                t.stop; setverdict(fail);
            }
        [] t.timeout {
                setverdict(fail, "Server was not
    responding. Check base url");
            }
        }
}}
```

Listing 11: A simple TTCN-3 test suite testing a RESTful API

4) Assertions: RMM level 2 and above applications are defined by respecting more strictly the actual semantics of HTTP operations. To maintain the targeted maturity level, assuring properties like idempotency increase the interoperability between clients and servers as clients.

By using the proposed TTCN-3 test framework, this class of assertions can be implemented efficiently, as shown by Listing 12. Similarly, also other deviations of idempotency can be implemented.[5]

```
function AssertIdempotency(FindPetsByStatus request,
     integer retries)
    runs on PetStoreClient return boolean {
    var template FindPetsByStatusResponse r1, r0;
    for (var integer i := 0; i < retries - 1; i := i
     + 1) {
        service.send(request);
        alt {
        [i == 0] service.receive(
    FindPetsByStatusResponse:?) -> value r1 {
                r0 := r1;
            }
        [i > 0] service.receive(
    FindPetsByStatusResponse:?) -> value r1 {
                if (not (match(r0, r1))) {
                    setverdict(fail);
                    return false;
                }
            }
        [] service.receive(HTTPResponse:?) {
                setverdict(fail);
                return false;
            }
        }
    }
    return true; }
```

Listing 12: A sample idempotency assertion

D. RESTful APIs on RMM level 3

As stated earlier, true RESTful APIs have RMM-level 3 maturity, i.e., hypertext media usage as a precondition. However, only very few publicly available APIs support hypermedia as ”the engine of application state” [13] (HATEOAS) which is one of its four constraints. For example, the ”freeplan” web service as provided by the Deutsche Bahn AG[6] describes the API as ”A RESTful webservice to request a railway journey [...]”. An analysis of the swagger[7] specification reveals that no

[5]Example: For repeated, identical DELETE requests, a successful DELETE request might return as a first response another status-code than subsequent requests. Nevertheless, the status of the resources (deleted) is the same for the first and all following identical requests. Therefore this type of behavior can also be considered ”idempotent”.

[6]https://developer.deutschebahn.com/store/apis/info?name=Fahrplan-Free&version=v1&provider=DBOpenData

[7]freeplan swagger specification

hypermedia is being used. The absence of hypermedia does not constitute a ”defect” or limitation in the API’s usability, but an architectural design decision. It just serves as an example of widespread incorrect use of the term *RESTful*.

Applications that use hypermedia in RESTful APIs embed, in addition to the resource data, links to describe transitions between resource states. A client only needs to *understand* the link names, the so-called relations, to interact and manipulate an APIs resources. As relations are central to the hypermedia concept and ”a REST API should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state [...]” [13], we will examine the fitness of our proposed framework by exploring the compatibility to ”HAL - Hypertext Application Language” [16] as an example. Beyond the usage of using standard relations as maintained, for example, by IANA[8] HAL is one way to define relations and encapsulates it in an own media type `application/hal+json`.

1) HAL - Hypertext Application Language: HAL target is to give ”a consistent and easy way to hyperlink between resources” in an API [17]. While HAL supports JSON and XML to express hyperlinks, we are currently considering the support of HAL and JSON only. Listing 13 shows a hypothetical response for the request of a `pet` as introduced earlier. It contains the status of the pet resource (line 12-13) as well as links to documentation (line 4) and operations (line 2-11). According to [17] `"_links"`, `"self"`, and `"curies"` should be included in every response.

```
1  {    "_links": {
2          "curies": [{ "name": "ps", "href": "http://
       petstore.io/docs/rels/{rel}", "templated": true
       }],
3          "self": { "href": "/pet/123" },
4          "next": { "href": "/pet/124" },
5          "previous": { "href": "/pet/122" },
6          "ps:findByStatus": {
7              "href": "/pet/findByStatus{?status}",
8              "templated": true
9          }
10     },
11     "petsName": "MyDog",
12     "status": "available"
13 }}
```

Listing 13: A (hypothetical) HAL compatible response for requesting a per resource

2) Testing HAL based applications: Link relations are the central element of RESTful API on RMM-level 3. Thus we mandate a test framework for RMM-level 3 RESTful APIs to verify the correctness or at least the availability of the advertised link relations. As the semantic of the link relations strongly depends on the used media type, we have started investing the applicability for HAL based applications. `Lib_HAL` includes predefined data types to support the modeling of HAL link relations. Listing 14 visualises the usage of the predefined data types *`Href`* and *`Curies`* to map the HAL specification into a TTCN-3 data type, together with `encoding` annotations to indicate the actual usage at execution time. We introduce a specific `Accept-Header` (line 5) and expect in a response a `Content-Type: application/hal+json` (line 12). In addition to the resource data, we have introduced in the message body of a `200 response` (line 11) a `_links` fields to capture the link relations (line 16-19).

```
1  module PetStoreHal language "TTCN-3:2018 Object-
       Oriented features" {
2      import from Lib_HAL all;
3  ...
4      type record GetPetById_HAL extends GetPetById {
5        string accept
6        } with { encode (accept) "header:Accept"; }
7
8        // 200 - successful operation
9        type union GetPetById_HALResponse
10           extends GetPetByIdResponse {
11     PetHal _200
12       } with { encode (_200) "body/hal+json"; }
13
14       type set PetHal {
15       string petName, string status,
16       record {
17           Href _self, Curies curies,
18           Href next optional, Href previous optional
19       } _links
20     }
21
22     template GetPetById_HAL GetPetHal := {
23       accept := "application/hal+json",
24       petId := 22
25       }
26  ...
27 }
```

Listing 14: A TTCN-3 type system for a HAL extended GetPetById operation

By using such types, variations of application-level testing can be performed. For example, it can be possible to define assertions that assert the validity of individual link relations (like **`self`**) or all link relations for each received response. This approach could also be combined with idempotency assertions, as introduced earlier. The different application-level testing approaches are currently under validation. Results on the effectiveness could be published in a later phase.

E. Implementation and validation

Fig. 3 outlines the structure and functional components of our proposed RESTful API test framework, `RJ-Plugin`[9]. It fully respects the specification of the TTCN-3 test system architecture, as defined in [18] [19] [20]. It consists of a set of mapping rules (a), a collection of predefined TTCN-3 libraries (b), as well as the respective TTCN-3 system adapters, platform adapters, and codecs (SA, PA, CD), packaged as TTworkbench plugins. With such a framework available, a broad range of new applications for TTCN-3 is opened in micro-service testing. For example, TTCN-3 can be used for micro-service unit testing, as well as integration testing. Applying the parallelization concepts of TTCN-3 scalability

[8]https://www.iana.org/assignments/link-relations/link-relations.xhtml

[9]Published as open-source for Spirent’s TTworkbench under the MIT license, available for review

and performance tests could be generated on the same test basis as for the functional tests.

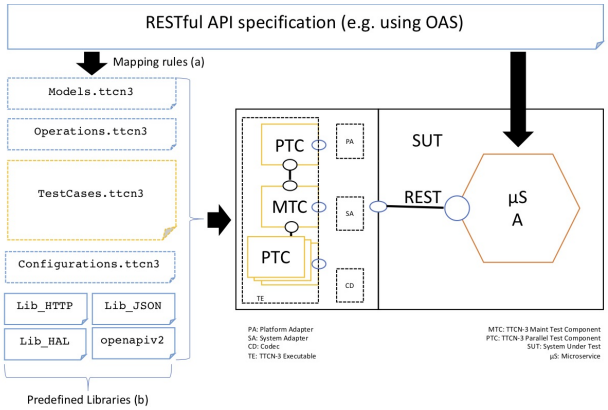


Fig. 3: Components and structure of the framework (RJ-Plugin)

While under constant development the framework offers the following features

- Multiple components support, to enable multiple, parallel executing test components
- Basic and API-Key authentication support.
- Flexible JSON encoding via POTO mapping or, if available via "JSON" or "JSON RFC7159"
- Functional support for RMM-level 2 aspects like testing *idempotency*
- Initial support for `application/hal+json` media type as a representation of hypermedia in RESTful APIs on RMM-level 3

We validated our approach by testing various publicly available API in OpenData[10], IoT[11], and smart home systems[12]. Some APIs offered an OpenAPI, aka swagger, specifications, others did not. For APIs that did not offer an OpenAPI spec, we specified one and applied the mapping rules.

IV. Results and Outlook

We have defined a TTCN-3 framework by defining a mapping between HTTP/JSON based RESTful API operations. We provide a reference implementation for the Spirent's TTworkbench and apply it to various applications in various domains. One of the key findings was that by using a structured test approach, we could identify underspecifications, like undocumented status code for responses. While all APIs used the term REST API, according to our assessment, none of the tested APIs achieved RMM-level 3 maturity. Only one (LaMetric) made some use of hypermedia concepts.

We are planning to support the automatic generation of the TTCN-3 type system for OpenAPI v2/v3 specifications and adding XML support.

[10]e.g. Deutsche Bahn APIs
[11]e.g. Mathworks ThingSpeak API
[12]e.g. LaMetric API or Timeular API

Acknowledgment

Funded by the German Federal Ministry of Education and Research(BMBF) - NR 16DII113. We are also grateful to the anonymous reviewers for their valuable suggestions to improve the quality of the presented ideas in this paper.

References

[1] R. Fielding and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content - RFC 7231," Jun. 2014, library Catalog: tools.ietf.org. [Online]. Available: https://tools.ietf.org/html/rfc7231

[2] W3C, "Web Services Architecture - Relationship to the World Wide Web and REST Architectures," 2004. [Online]. Available: https://www.w3.org/TR/ws-arch/#relwwwrest

[3] ——, "Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language," 2007. [Online]. Available: https://www.w3.org/TR/wsdl20/

[4] ——, "Simple Object Access Protocol (SOAP) 1.1," 2000. [Online]. Available: https://www.w3.org/TR/2000/NOTE-SOAP-20000508/

[5] OpenAPI Initiative, "OpenAPI Specification (OAS) v3.0.3," Feb. 2020. [Online]. Available: http://spec.openapis.org/oas/v3.0.3

[6] Smartbear Software, "OpenAPI Specification - Version 2.0 Swagger." [Online]. Available: https://swagger.io/specification/v2/

[7] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Doctoral Thesis, University of California, Irvine, 2000. [Online]. Available: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[8] ECMA International, "The JSON Data Interchange Syntax," Dec. 2017. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[9] ETSI. European Telecommunications Standards Institute (ETSI), "ETSI Standard (ES) 201 873-7 V4.7.1: The Testing and Test Control Notation version 3; Part 7: Using ASN.1 with TTCN-3," 2018, pages: 1-320 Volume: 1 Place: Sophia- Antipolis, France.

[10] ——, "ETSI Standard (ES) 201 873-9 V4.10.1: The Testing and Test Control Notation version 3; Part 9: Using XML schema with TTCN-3," 2019, pages: 1-320 Volume: 1 Place: Sophia- Antipolis, France.

[11] ——, "ETSI Standard (ES) 201 873-11 V4.8.1: The Testing and Test Control Notation version 3; Part 11: Using JSON with TTCN-3," 2018, pages: 1-34 Volume: 1 Place: Sophia- Antipolis, France.

[12] ——, "ETSI Standard (ES) 201 873 V4.11.1: The Testing and Test Control Notation version 3; Parts 1-11," 2019, pages: 1-320 Volume: 1 Place: Sophia- Antipolis, France. [Online]. Available: http://www.ttcn-3.org/index.php/downloads/standards

[13] R. T. Fielding, "REST APIs must be hypertext-driven," Oct. 2008, library Catalog: roy.gbiv.com. [Online]. Available: https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

[14] Google, "Protocol Buffers Version 3 Language Specification | Google Developers." [Online]. Available: https://developers.google.com/protocol-buffers/docs/reference/proto3-spec

[15] Smartbear Software, "Petstore - A sample server." [Online]. Available: https://petstore.swagger.io/

[16] M. Kelly, "HAL - Hypertext Application Language," Sep. 2013. [Online]. Available: http://stateless.co/hal_specification.html

[17] ——, "JSON Hypertext Application Language," library Catalog: tools.ietf.org. [Online]. Available: https://tools.ietf.org/html/draft-kelly-json-hal-08

[18] ETSI. European Telecommunications Standards Institute (ETSI), "ETSI Standard (ES) 201 873 V4.11.1: The Testing and Test Control Notation version 3; Part: 1: TTCN-3 Core Language," Apr. 2019.

[19] ——, "ETSI Standard (ES) 201 873-5 V4.8.1: The Testing and Test Control Notation version 3; Part: 5: TTCN-3 Runtime Interface (TRI)," May 2017.

[20] ——, "ETSI Standard (ES) 201 873-6 V4.12.1: The Testing and Test Control Notation version 3; Part: 6: TTCN-3 Control Interfaces (TCI)," May 2020.