Automated Measurement and Analysis of Open Source Software

Project Number 318772

D5.5 – REST API

Version 1.0
10 October 2014
Final

Public Distribution

University of York

Project Partners: **Centrum Wiskunde & Informatica**, **SOFTEAM**, **Tecnalia Research and Innovation**, **The Open Group**, **University of L'Aquila**, **UNINOVA**, **University of Manchester**, **University of York**, **Unparallel Innovation**

Project Partner Contact Information

Centrum Wiskunde & Informatica Paul Klint Science Park 123 1098 XG Amsterdam, Netherlands Tel: +31 20 592 4126 E-mail: paul.klint@cwi.nl	**SOFTEAM** Alessandra Bagnato Avenue Victor Hugo 21 75016 Paris, France Tel: +33 1 30 12 16 60 E-mail: alessandra.bagnato@softeam.fr
Tecnalia Research and Innovation Jason Mansell Parque Tecnologico de Bizkaia 202 48170 Zamudio, Spain Tel: +34 946 440 400 E-mail: jason.mansell@tecnalia.com	**The Open Group** Scott Hansen Avenue du Parc de Woluwe 56 1160 Brussels, Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of L′Aquila Davide Di Ruscio Piazza Vincenzo Rivera 1 67100 L'Aquila, Italy Tel: +39 0862 433735 E-mail: davide.diruscio@univaq.it	**UNINOVA** Pedro Maló Campus da FCT/UNL, Monte de Caparica 2829-516 Caparica, Portugal Tel: +351 212 947883 E-mail: pmm@uninova.pt
University of Manchester Sophia Ananiadou Oxford Road Manchester M13 9PL, United Kingdom Tel: +44 161 3063098 E-mail: sophia.ananiadou@manchester.ac.uk	**University of York** Dimitris Kolovos Deramore Lane York YO10 5GH, United Kingdom Tel: +44 1904 325167 E-mail: dimitris.kolovos@york.ac.uk
Unparallel Innovation Nuno Santana Rua das Lendas Algarvias, Lote 123 8500-794 Portimão, Portugal Tel: +351 282 485052 E-mail: nuno.santana@unparallel.pt	

Contents

Document Control

Version	Status	Date
0.1	Document outline	1 August 2014
0.2	First draft	15 September 2014
0.3	Second draft	27 September 2014
0.3	For partner review	2 October 2014
1.0	Final Version	10 October 2014

Executive Summary

The purpose of this report is to document the design and the implementation of the REST API, which is used to expose the functionality and data of the OSSMETER platform to external users, and which is developed in the context of deliverable D5.5, which is primarily a software deliverable. This document is an accompanying overview of the delivered software, and it outlines the motivation and implementation of the REST API, as well as how this API is distributed and how it can be used.

In this report, we present a brief discussion on different methods for application integration and data interchange. In this discussion, we present the two main approaches to application interaction, namely HTTP-based transactional architectures and stub/skeleton-based remote procedure call architectures. For both of these types of architectures, we identify their strong points, as well as their shortcomings. Furthermore, we present the web services approach to application integration and more particularly its SOAP and REST variants. Finally, we argue why we decided to use the REST paradigm for providing application integration capabilities to the OSSMETER platform.

After providing the background to the work presented in this report, we present how the OSSMETER API is bundled and distributed. Moreover, we present the specification of this API by discussing high level information such as the targeted audience of this API, as well as low level information for the provided resources and operations. Finally, we present a tool we have implemented for generating client code against the OSSMETER API. Currently, this tool uses the OSSMETER data models to generate implementations in C#, Java, and Python.

To conclude, we relate the requirements laid out by the user case providers in deliverable *D1.1 - Project Requirements* to the work that has been performed for deliverable *D5.5 - REST API*, and show the level of compliance with each requirement.

1 Introduction

The OSSMETER system is an extensible platform for software quality analysis of Open Source Software (OSS). It consists of components for the collection and preprocessing of project data, quality analysis components, as well as visualisation and presentation components. The aim of OSSMETER is to deliver an integrated environment, which will facilitate decision makers to compare and choose between different OSS projects. Moreover, OSSMETER enables managers and community leaders to monitor and manage their OSS projects in an easy and intuitive manner. This is achieved by analysing software engineering data under the hood and exposing to the system users high level information and visualisations. In addition, the architecture of the system allows users to extend or modify the measurement and presentation components of the platform if such need arises.

1.1 Purpose of the Deliverable

The purpose of this report is to document the design and the implementation of the REST API, which is used to expose the functionality and data of the OSSMETER platform to external users. Figure 1 illustrates the architecture of the OSSMETER system, which is presented in detail in deliverable *D5.1- Platform Architecture Specification*. According to this architectural specification, the REST API layer is a layer atop the OSSMETER system, which encapsulates the specificities of the platform enabling external users to interact with the system and retrieve data produced by the platform at a higher level of abstraction.

D5.5 is primarily a software deliverable and this document is an accompanying overview of the delivered software. It outlines the motivation and implementation of the REST API, as well as how this API is distributed and how it can be used.

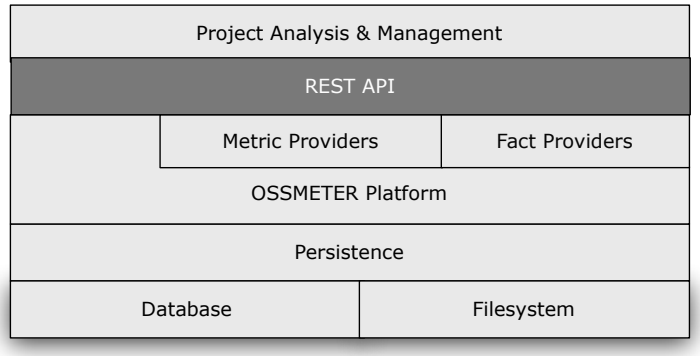

Figure 1: OSSMETER system architecture.

The work presented in this document is done in the context of work package (WP) 5, whose objective is to develop the OSSMETER system and integrate the components developed in work packages 2-4. The results presented in the following sections are related to the following task (from the OSSMETER DoW):

Task 5.4: REST API Design and Implementation - In this task, a REST API will be designed to allow external developers to retrieve the calculated metrics for individual OSS projects in platform-neutral formats (e.g. JSON, SOAP, plain XML) and exploit them in custom applications. To ease the adoption and integration of this API with custom applications, clients for a number of programming languages (at least Java, Python and C#) will also be provided using MDE techniques (they will be generated from a common specification).

1.2 Structure of the Document

The rest of the document is structured in the following way. In Section 2.1 we present related work on application interaction and we argue why we have chosen to use the REST architectural style for allowing interaction of external applications with the OSSMETER platform. Section 3 presents the OSSMETER REST API. More particularly, Section 4 presents high level implementation details, while Section 5 provides the specification of the API. In Section 6 we present a code generator, which enables the automatic generation of client libraries from the WADL specification of the REST API. Finally, Section 7 associates the work presented in this document to the project's requirements, while Section 8 concludes this report by summarising the report's contents.

1.3 Relationship to Other OSSMETER Deliverables

This document presents the REST API and its integration into the OSSMETER system. The OSSMETER platform is developed within WP5 and its architecture is presented in deliverable *D5.1-Platform Architecture Specification*. The component integration is described in *D5.3-Component Integration Interim Report* and *D5.5-Component Integration Final Report*. Furthermore, the REST API presented in this deliverable is used by the OSSMETER web client, which is presented in deliverable *D5.6-End-User Web Application*. Finally, deliverable *D1.1-Project Requirements* of WP1 lists the requirements, which are related to the work described in this document.

1.4 Contributors

The main contributor of this deliverable is University of York. All project partners contributed to this document by providing system requirements, as well as by providing feedback and suggestions for editing and refinements of this document.

2 Background

One of the main goals of the OSSMETER platform is to allow easy access to the calculated measurements and raw data by external applications. Such applications can possibly be deployed on the machine the OSSMETER system is deployed on or they can be deployed on a different machine connected to the network.

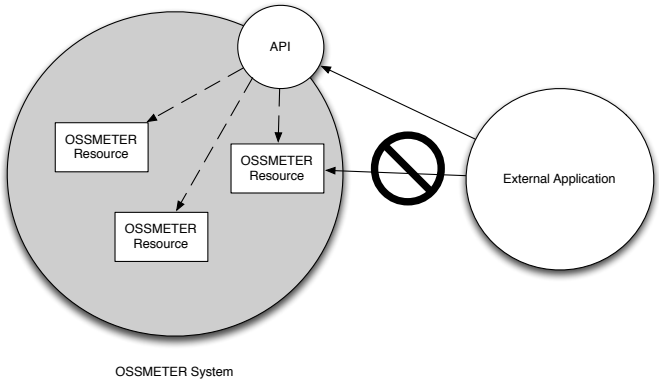

Figure 2: Platform interaction with external applications.

Figure 2 illustrates how we envisage the platform to connect to external tools. More particularly, external applications should not access OSSMETER resources directly, but through a dedicated and well-defined interface. In this way, client developers should not have to worry about the intricacies of the platform or of the data storage but they should focus on using the data provided by the platform to solve the problem at hand. Moreover, if platform resources are accessed only via a well-defined API, they are protected from getting corrupted accidentally by clients.

In this section we briefly review the different alternatives we considered for providing integration with external applications. Moreover, we provide an explanation on why we have chosen to use REST Web Services.

2.1 Application Interaction Architectures

Various architectures exist for providing application interaction and data interchange over a network. These architectures can be grouped into two categories:

- HTTP-based transactional architectures
- Stub/Skeleton-based remote procedure call architectures

2.1.1 HTTP Based Transactional Architectures

Architectures belonging to this category operate by having an application running on the server and clients communicate with this application through HTTP. After a transaction is complete, the con-

nection between the client and the server is terminated. Examples of such approaches are JSP[1], ASP[2], and CGI[3].

The main advantage of these architectures is that the communication takes place over the standard HTTP ports. On the other hand, since the connection between the client and the server is not persistent, there is a significant overhead creating and terminating connections in scenarios where clients make a number of discrete connections. Moreover, these types of architectures lack a directory service such as UDDI, and thus clients must be aware of a service in advance, of what data the service expects and how the service expects to receive the data. Finally, there is no standard way to handle and report errors in such architectures.

2.1.2 Stub/Skeleton-Based Architectures

Architectures belonging to this category provide programmatic access to a remote service as though it is a local entity. In other words, clients treat objects and methods of the remote server application as if they are local. The communication and the data interchange is taken care from the architecture. Such applications rely to two components, the stub and the skeleton. The stub module is part of the client and it identifies, which methods are available on the server, as well as it encodes and decodes the calls to the server. The skeleton module is part of the server application and is responsible for making available, which parts of the application are available to clients. Moreover, it handles requests from clients. In stub/skeleton architectures, once a client connects to the server it stays connected until the client is terminated. Examples of such architectures are CORBA[4], RMI[5], and Component Object Model (COM)[6].

Although, these approaches do not have the disadvantages of the HTTP-based transactional architectures, they have limitations as well. RMI technology locks the user into a purely Java solution, while CORBA can be hard to understand since it does not rely on XML for translating data, but on an Interface Definition Language (IDL). Moreover, the aforementioned technologies use binary protocols for transmitting the data over the network. There is no standard protocol and each of these technologies uses its own. Some of them can interact with each other (e.g. RMI and CORBA) but others cannot. Moreover, since these protocols are binary, specific ports have to be opened on firewalls in order to enable communication over the Internet [2].

2.2 Web Services

Web Services (WS) is a third approach to application interaction, and it is considered to be a combination of the two aforementioned architectural groups. A service is a software component provided through a network-accessible endpoint. More particularly, WS allow clients to invoke procedures,

[1] http://www.oracle.com/technetwork/java/javaee/jsp/index.html
[2] http://msdn.microsoft.com/en-us/library/aa286483.aspx
[3] http://www.w3.org/CGI/
[4] http://www.corba.org/
[5] http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html
[6] https://www.microsoft.com/com/default.mspx

functions, and methods on remote objects using an XML-based protocol. In the WS realm, the system that makes a request is called a service requester, while the system that processes this request is called a service provider.

Figure 3: WS protocol stack.

Figure 3 illustrates the protocol stack of WS. The bottom layer of this stack is responsible for transporting messages between communicating applications. The main protocol of this layer is HTTP. The messaging layer is responsible for encoding messages, so that they can be understood. Traditionally, such messages have been encoded in XML. In 2005 Yahoo[7] started offering its WS in JSON[8]. Nowadays, both XML and JSON are widely used for encoding WS requests and replies. The *Service Description* layer is responsible for describing the public interfaces of WS. Technologies used to describe such interfaces include the Web Service Description Language (WSDL[9]) and the Web Application Description Language (WADL[10]). Finally, the *Service Discovery* layer is responsible for registering and locating WS applications.

Since WS are accessible via standard protocols like XML and HTTP, they can be combined in a loosely coupled way to achieve complex operations. Therefore, WS do not tie developers to particular programming environments like RMI technology does neither to specific representations like IDL. Moreover, WS do not suffer from the limitations of binary protocols for the interchange of data since they rely on XML, which is textual. Another advantage of WS over traditional HTTP-based interoperation mechanisms is the use of XML-based languages for specifying an interface. In the case of traditional HTTP-based architectures cooperation between the provider and the consumer of the data is required in order to built a functioning system.

On the other hand, since WS often rely on HTTP for communication over the network, they suffer from the limitation that the connection between a client and a server is not persistent. However, the benefits of using WS over traditional approaches outweigh the limitations. Therefore, the technology

[7] https://www.yahoo.com/

[8] http://web.archive.org/web/20071011085815/http://developer.yahoo.com/common/json.html

[9] http://www.w3.org/TR/wsdl

[10] https://wadl.java.net/

used for exposing the data of the platform to external clients in the context of OSSMETER is WS. In the following section, we will briefly introduce the main variants of WS.

2.3 "Big" vs REST web services

There are two types of WS, namely "Big" and REST. "Big" web services use XML messages, which follow the Simple Object Access Protocol (SOAP) standard. This standard defines the message architecture and the messages formats. A human readable description of such services is often provided expressed in WSDL. The main advantage of SOAP is the support of multiple protocols for transporting the XML messages. Therefore, SOAP can use almost any transport protocol to send requests and receive replies, from HTTP to SMTP or Java Messaging Service (JMS). Moreover, SOAP provides guarantees for non-functional requirements such as security or reliability. However, these guarantees do not come for free. The SOAP XML messages are verbose and complicated in order to support such features, and this causes an overhead.

The alternative to SOAP web services is REST (Representational State Transfer) web services. REST web services follow four basic design principles:

- Use HTTP methods explicitly.
- Be stateless.
- Expose directory structure-like URIs.
- Transfer XML, JSON, or both.

REST is considered to be a light-weight alternative to SOAP. This is because SOAP relies exclusively on XML for requests and replies, while REST can use less verbose message formats such as JSON. On the other hand SOAP provides error handling, as well as security and reliability guarantees. Moreover, the use of SOAP requires dedicated middleware, while REST relies solely on HTTP.

The main goal of using web services in the context of OSSMETER is to enable external tools to use OSSMETER data in an easy manner. To this end, we choose to use REST web services due to the simplicity of the REST approach compared to the SOAP one. Since we do not expect critical applications to be built atop the OSSMETER platform, the security and reliability features of SOAP are not so important as the reduced complexity of REST.

3 OSSMETER REST API Bundle

The OSSMETER REST API bundle is packaged and distributed as part of the main OSSMETER platform. The API is implemented in the following plugin:

- `org.ossmeter.platform.client.api`

This plugin has a dependency on the following OSSMETER components:

- `org.eclipse.core.runtime`
- `org.restlet`
- `org.ossmeter.platform`
- `org.ossmeter.repository.model`
- `org.jackson.all`
- `com.googlecode.pongo.runtime`
- `org.ossmeter.platform.visualisation`

Dependencies on external libraries such as Restlet[11], which is a REST framework, or Jackson [12], which is a JSON parser, are bundled together with the rest of the platform. All platform dependencies are distributed under EPL friendly licences. The OSSMETER platform, and consequently the REST API component are distributed under the EPL 1.0 license[13].

4 API Implementation

The REST API of the OSSMETER platform is developed using the Restlet framework[14]. Restlet is a lightweight open source RESTful web API framework for the Java platform. In OSSMETER two main framework packages are used:

- `org.restlet`
- `org.restlet.ext.osgi`

[11] `http://restlet.com/`
[12] `http://jackson.codehaus.org/`
[13] `http://www.eclipse.org/legal/epl-v10.html`
[14] `http://restlet.com/`

The `org.restlet` package contains the core of the Restlet framework. The main feature of this package is the Restlet API, a compact and portable Java API, that embodies major REST concepts such as *Connectors*, *Representations*, and *Components*.

The `org.restlet.ext.osgi` package is an extension to the core Restlet package and it allows the deployment of Restlet applications as OSGi services. The ability to being deployed as an OSGi bundle is a very important feature in the context of OSSMETER, since the OSSMETER platform is distributed as a set of OSGi bundles running within an Equinox container.

5 API Documentation

The OSSMETER REST API is a REST, resource-oriented API accessed via HTTP that uses JSON based representations for information interchange. The API is designed to provide access to the following aspects of the OSSMETER platform:

- Importing, retrieving and updating project's metadata.
- Retrieving metric provider's data and visualisations.
- Retrieving factoid's data.

5.1 Intended Audience

The OSSMETER REST API is intended for use by engineers, who wish to integrate external tools with the OSSMETER platform or to extract platform's data. This report documents this API. To use it, the reader should have a general understanding of:

- REST web services
- HTTP 1.1
- JSON data serialisation format

5.2 Resource Summary

Figure 4 provides a summary of the available OSSMETER REST resources[15]. In the following sections, we will provide the resource definitions, the applicable HTTP verbs and examples for each resource.

5.3 Authentication

Authentication is not supported in Version 1.0 of the OSSMETER REST API.

[15]In the REST terminology a resource can be anything of potential interest that is serializable in some form. In the context of OSSMETER, resources are the results of queries on the OSSMETER data storage.

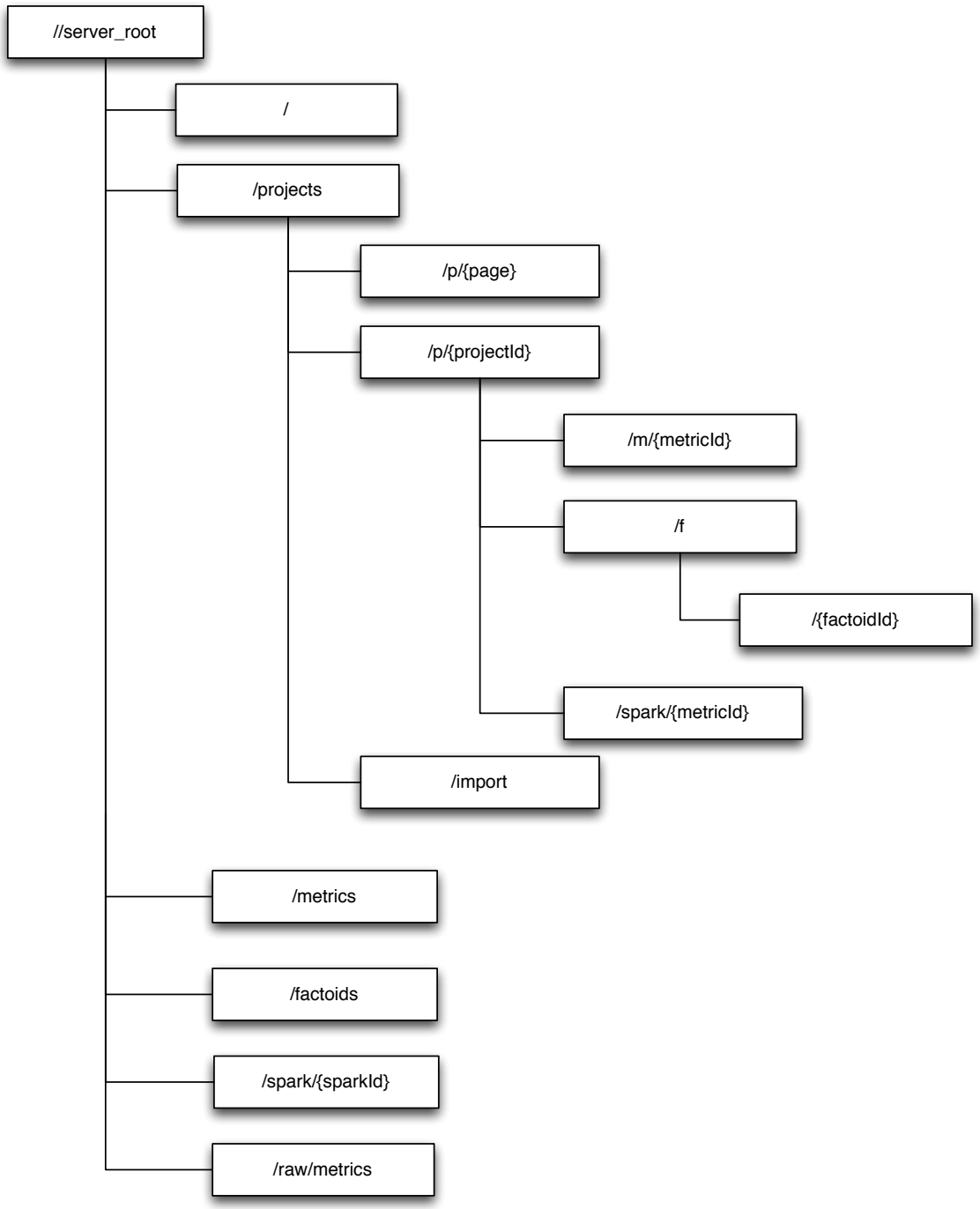

Figure 4: Summary of OSSMETER resources.

5.4 Representation Format

The OSSMETER API supports JSON-based representation format for both requests and responses. This is specified by setting the `Content-Type` header to `application/json` MIME type, if the request/response has a body.

5.5 Representation Transport

Resource representation is transmitted between client and server by using HTTP 1.1 protocol, as defined by IETF RFC-2616. Each time an HTTP request contains payload, a Content-Type header shall be used to specify the MIME type of wrapped representation. In addition, both client and server may use as many HTTP headers as they consider necessary.

5.6 Resource Identification

The resource identification for HTTP transport is made using the mechanisms described by HTTP protocol specification as defined by IETF RFC-2616.

5.7 Limits

There is no limit on the number of requests the API can serve.

5.8 Versions

Querying the version is not supported in version 1.0 of the API.

5.9 API Operations

In this section we provide the details for each REST operation giving the expected input and output for each URI.

5.9.1 API Liveness

GET	/

Description: This is used to check if the API is online. Returns 200 if API is online.

Sample result:

```
200(OK)
Content-Type: application/json
{'status':200}
```

Listing 1: API Liveness

5.9.2 Projects List

GET	/projects

Description: This is used to retrieve a list of monitored projects.

Parameters:

Name	Description	Details
page	This parameter specifies the number of pages to be returned by the server.	Optional, integer value.
size	This parameter specifies the size of a page	Optional, integer value.

Sample result:

```
200(OK)
Content-Type: application/json
[{
  _type: "org.ossmeter.repository.model.Project",
  parent: {},
  vcsRepositories: [... ],
  communicationChannels: [... ],
  bugTrackingSystems: [... ],
  licenses: [... ],
  persons: [... ],
  companies: [],
  name: "Epsilon",
  shortName: "epsilon"
  "description" :"Epsilon is a family of languages ..."
},{
  _type: "org.ossmeter.repository.model.github.GitHubRepository",
  parent: {},
  vcsRepositories: [... ],
  communicationChannels: [... ],
  bugTrackingSystems: [... ],
  companies: [],
  languages: [],
  contents: [],
  downloads: [],
  forks: [],
  milestones: [],
  description: "A pre-commit hook for Git that creates GIFs ...",
  full_name: "jrwilliams/gif-hook",
  shortName: "gif-hook",
  name: "gif-hook",
  _private: false,
  fork: false,
  git_url: "git://github.com/jrwilliams/gif-hook.git",
  html_url: "https://github.com/jrwilliams/gif-hook",
  clone_url: "https://github.com/jrwilliams/gif-hook.git",
  ssh_url: "gitgithub.com:jrwilliams/gif-hook.git",size:  3928]
```

Listing 2: Projects List.

5.9.3 Project Import

POST	/projects/import

Description: This is used to import a new project to be monitored. It accepts a JSON object with a 'url' field that can be used by the importers. Returns an "Unable to import project" message if it fails, or returns the project's metadata.

Sample result:

```
200(OK)
Content-Type: application/json
{
  _type: "org.ossmeter.repository.model.Project",
  parent: {},
  vcsRepositories: [... ],
  communicationChannels: [... ],
  bugTrackingSystems: [... ],
  licenses: [... ],
  persons: [... ],
  companies: [],
  name: "Epsilon",
  shortName: "epsilon"
  "description" :"Epsilon is a family of languages ..."
}
```

Listing 3: Project import.

5.9.4 Project

GET	/projects/p/{projectId}

Description: This is used to retrieve the metadata of a specific project.

Parameters:

Name	Description	Details
projectId	This parameter specifies the string ID of a specific project.	Required, String value.

Sample result:

```
200(OK)
Content-Type: application/json
X-My-Header: The Value
{
   "parent" :{},
   "vcsRepositories" :[
```

```
      {
         "name" :"epsilon",
         "url" :"http://dev.eclipse.org/svnroot/modeling/
                org.eclipse.epsilon"
      }
   ],
   "communicationChannels" :[
      {
         "name" :"eclipse.epsilon",
         "url" :"news.eclipse.org/eclipse.epsilon",
         "authenticationRequired" :true,
         "port" :80,
         "lastArticleChecked" :"-1"
      }
   ],
   "bugTrackingSystems" :[
      {
         "url" :"https://bugs.eclipse.org/bugs/xmlrpc.cgi",
         "product" :"Epsilon",
         "component" :"Core"
      }
   ],
   "persons" :[],
   "licenses" :[],
   "name" :"modeling.epsilon",
   "shortName" :"modelingepsilon",
}
```

Listing 4: Project Resource.

5.9.5 Project Metric Visualisation

GET	/projects/p/{projectId}/m/{metricId}

Description: This is used to retrieve the visualisation of a specific project metric provider. Visualisations are defined in the *MetVis* language presented in *D5.4 - Component Integration Final Report*.

Parameters:

Name	Description	Details
projectId	This parameter specifies the string ID of a specific project.	Required, String value.
metricId	This parameter specifies the string ID of a specific metric provider.	Required, String value.

Sample result:

```
200(OK)
Content-Type: application/json
{
  "id": "averageSentiment",
  "name": "Average Sentiment",
  "description": "...",
  "type": "LineChart",
  "datatable": [
```

```
    {
    Date: "20071127",
    "Average Sentiment": 0
    },
    {
    "Date": "20071129",
    "Average Sentiment": 0
    },...
  ]
  "timeSeries": true,
  "categorical": false,
  "x": "Date",
  "y": "Average Sentiment"
}
```


Listing 5: Project Metric Visualisation.

5.9.6 List of Project Factoids

GET	/projects/p/{projectId}/f

Description: This is used to retrieve a list of project factoids. Factoids are components, which are used to aggregate low level metric providers. Factoids are presented in detail in *D5.4 - Component Integration Final Report*

Parameters:

Name	Description	Details
projectId	This parameter specifies the string ID of a specific project.	Required, String value.

Sample result:


```
200(OK)
Content-Type: application/json
[
  {
    "id": "cocomo",
    "name": "Cocomo",
    "summary": "....",
    "dependencies": []
  },{
    "id": "org.ossmeter.factoid.bugs.channelusage.BugsChannelUsageFactoid",
    "name": "Bug Channel Usage",
    "summary": "...",
    "dependencies": [
      "org.ossmeter.metricprovider.historic.bugs.newbugs.model.BugsNewBugsHistoricMetric",
      "org.ossmeter.metricprovider.historic.bugs.comments.model.BugsCommentsHistoricMetric",
      "org.ossmeter.metricprovider.historic.bugs.patches.model.BugsPatchesHistoricMetric"
    ]
  }
]
```


Listing 6: List of Project Factoids

5.9.7 Project Factoid

GET	/projects/p/{projectId}/f/{factoidId}

Description: This is used to retrieve the data of a specific factoid for a project.

Parameters:

Name	Description	Details
projectId	This parameter specifies the string ID of a specific project.	Required, String value.
factoidId	This parameter specifies the string ID of a specific factoid.	Required, String value.

Sample result:

```
200(OK)
Content-Type: application/json
{
   "_id" :"1d6e42c0-73e5-4e03-b2e2-9b2f8837c365",
   "_type" :"org.ossmeter.platform.factoids.Factoid",
   "metricDependencies" :[
      "org.ossmeter.metricprovider.historic.bugs.newbugs.model.BugsNewBugsHistoricMetric",
      "org.ossmeter.metricprovider.historic.bugs.comments.model.BugsCommentsHistoricMetric",
      "org.ossmeter.metricprovider.historic.bugs.patches.model.BugsPatchesHistoricMetric"
   ],
   "metricId" :"org.ossmeter.factoid.bugs.channelusage.BugsChannelUsageFactoid",
   "name" :"Bug Channel Usage Factoid",
   "stars" :"FOUR",
   "factoid" :"The project is associated with 1bug tracking system. In the last year, it has
       received high attention. 59new bugs, 265new comments and 46new patches have been posted, in
       total. In the last month, 3new bugs, 11new comments and 1new patches have been posted to the
       bug trackers of the project."
}
```

Listing 7: Project Factoid

5.9.8 Sparkline of Project Metric

GET	/projects/p/{projectId}/s/{metricId}

Description: This is used to compute sparkline related data for a metric. See deliverable *D5.4 - Component Integration Final Report* for a full discussion on how sparklines are generated.

Parameters:

Name	Description	Details
projectId	This parameter specifies the string ID of a specific project.	Required, String value.

metricdId	This parameter specifies the string ID of a specific metric provider.	Required, String value.

Sample result:

```
200(OK)
Content-Type: application/json
{
  "id": "averageSentiment",
  "name": Average Sentiment",
  "description": "...",
  "low": -0.375,
  "high": 0,
  "first": 0,
  "last": -0.2885679602622986,
  "firstDate": "27/11/2007",
  "lastDate": "14/10/2014",
  "months": 82,
  "spark": "/spark/55865160-b51c-4bd5-8d49-f06e64f2e585",
  "metricId": "averageSentiment",
  "projectId": "epsilon"
}
```

Listing 8: Sparkline of Project Metric

5.9.9 Sparkline Image

GET	/spark/{sparkId}

Description: This retrieves the sparkline image with the given identifier. See deliverable *D5.4 - Component Integration Final Report* for a full discussion on how sparklines are generated.

Parameters:

Name	Description	Details
sparkId	This parameter specifies the string ID of a specific sparkline image.	Required, String value.

Sample result:

```
200(OK)
Content-Type: image/png
```

Listing 9: Sparkline Image

5.9.10 List of Metrics

GET	/metrics

Description: This is used to retrieve a list of the visualisable metric providers supported by the platform. It includes information about how the metric provider should be visualised - in the *MetVis* format (see deliverable *D5.4 - Component Integration Final Report*).

Sample result:

```
200(OK)
Content-Type: application/json
[
  {
    "id": "bugOpenTime",
    "name": "Bug Open Time",
    "description" :" ... ",
    "type": "LineChart",
    "datatable": {
      "cols": [{
        "name": "Date",
        "field": "$__date"
      },{
        "name": "Bug Open Time",
        "field": "$avgBugOpenTime"
      }]
    },
    "x": "Date",
    "y": "Bug Open Time",
    "timeSeries": true
  },
  {
    "id": "sentimentAtThreadEnd",
    "name": "Sentiment at thread end",
    "description" :" ... ",
    "type": "BarChart",
    "datatable": {
      "cols": [{
        "name": "Date",
        "field": "$__date"
      },{
        "name": "Sentiment At Thread End",
        "field": "$overallSentimentAtThreadEnd"
      }]
    },
    "x": "Date",
    "y": "Sentiment At Thread End",
    "timeSeries": true
  },...
]
```

Listing 10: List of Metrics

5.9.11 List of Factoids

GET	/factoids

Description: This is used to retrieve a list of factoids supported by the platform.

Sample result:

```
200(OK)
Content-Type: application/json
[
  {
    "id": "cocomo",
    "name": "Cocomo",
    "summary": "....",
    "dependencies": []
  },{
    "id": "org.ossmeter.factoid.bugs.channelusage.BugsChannelUsageFactoid",
    "name": "Bug Channel Usage",
    "summary": "...",
    "dependencies": [
      "org.ossmeter.metricprovider.historic.bugs.newbugs.model.BugsNewBugsHistoricMetric",
      "org.ossmeter.metricprovider.historic.bugs.comments.model.BugsCommentsHistoricMetric",
      "org.ossmeter.metricprovider.historic.bugs.patches.model.BugsPatchesHistoricMetric"
    ]
  }
]
```

Listing 11: List of Factoids

5.9.12 List of Raw Metrics

GET	/raw/metrics

Description: This is used to retrieve a list of the all of the metric providers supported by the platform. This resource is different from the resource described in 5.9.10, which returns only the visualisable metric providers of the platform.

Sample result:

```
200(OK)
Content-Type: application/json
[{
  "name": "Commits",
  "type": "org.ossmeter.metricprovider.trans.commits.CommitsTransientMetricProvider",
  "description": "Transient metric provider for commits."
},{
  "name": "Commits by day",
  "type": "org.ossmeter.metricprovider.trans.dailycommits.DailyCommitsMetricProvider",
  "description": "Commits group by the day on which they occurred."
```

```
},{
  "name": "SourceForge importer",
  "type": "org.ossmeter.metricprovider.trans.importer.sourceforge.SourceForgeImporterProvider",
  description: "This provider enable to update a projects calling a importProject from sourceforge
      importer"
},{
  "name": "GoogleCode importer",
  "type": "org.ossmeter.metricprovider.trans.importer.googlecode.GoogleCodeImporterProvider",
  "": "This provider enable to update a projects calling a importProject from google code importer"
}, ...]
```

Listing 12: List of Raw Metrics

5.9.13 Project Metric Provider Data

GET	/raw/projects/p/projectId/m/metricId

Description: This is used to retrieve the raw data of a project's metric provider. This is essentially a JSON dump of the metric provider's MongoDB collection.

Parameters:

Name	Description	Details
projectId	This parameter specifies the string ID of a specific project.	Required, String value.
metricId	This parameter specifies the string ID of a specific metric provider.	Required, String value.

```
200(OK)
Content-Type: application/json
[
  {
    _id: "a3614b4e-1e71-430b-9ea1-16161e0ac4db",
    _type: "org.ossmeter.metricprovider.historic.bugs.sentiment.model.BugsSentimentHistoricMetric",
    overallAverageSentiment: 0,
    overallSentimentAtThreadBeggining: 0,
    overallSentimentAtThreadEnd: 0,
    __date: "20071127",
    __datetime: {
      $date: "2007-11-27T00:00:00.000Z"
    }
  },{
    _id: "b5ebbffe-7d85-404f-a6c7-fa1628e3cc73",
    _type: "org.ossmeter.metricprovider.historic.bugs.sentiment.model.BugsSentimentHistoricMetric",
    overallAverageSentiment: 0,
    overallSentimentAtThreadBeggining: 0,
    overallSentimentAtThreadEnd: 0,
    __date: "20071129",
    __datetime: {
      $date: "2007-11-29T00:00:00.000Z"
    }
  }, ...
]
```

Listing 13: Project Metric Provider Data

6 Generation of API Clients

The REST API provided by the OSSMETER platform can be consumed by external applications. Such external applications can be implemented in different programming languages and on various platforms. To enable the easy implementation of clients against the provided API we have built a code generation facility, which generates the necessary code for interacting with the API. This code generator is implemented using model-driven engineering technologies, and more particularly using the Epsilon model management framework [1] and the Eclipse Modelling Framework [4]. The corresponding Eclipse plugin is the following:

- `org.ossmeter.platform.client.generator`

Moreover, we have generated client code for C#, Java, and Python. The generated code can be located in the following OSSMETER plugins;

- `org.ossmeter.platform.client.csharp`
- `org.ossmeter.platform.client.java`
- `org.ossmeter.platform.client.python`

The generated Java client is used by the OSSMETER web application, described in deliverable *D5.6 - End-User Web Application*, to extract data from the platform.

The code generator uses the models defined in the various OSSMETER plugins as well as the WADL specification of the REST API to perform the code generation. A high level overview of the code generator is illustrated in Figure 5. The code generator consists of two components, namely a language-specific and a language-agnostic. The language-specific component contains templates for the supported languages, while the language-agnostic one contains utility operations, which contains common functions that can used by the language-specific templates.

The code generator is distributed as an Eclipse project and it depends on the Eclipse Modelling Framework and the Epsilon model management framework. The structure of the generator project is illustrated in Figure 6.

The `lib` directory contains the Apache Ant[16] library, which can be used to orchestrate Epsilon model management tasks. The `util` directory contains utility Epsilon operations, that are used in the language-specific generators, while the `models` directory contains the models used to generate the code from. This set of models consists of the metamodels defined in the OSSMETER manager

[16] `http://ant.apache.org/`

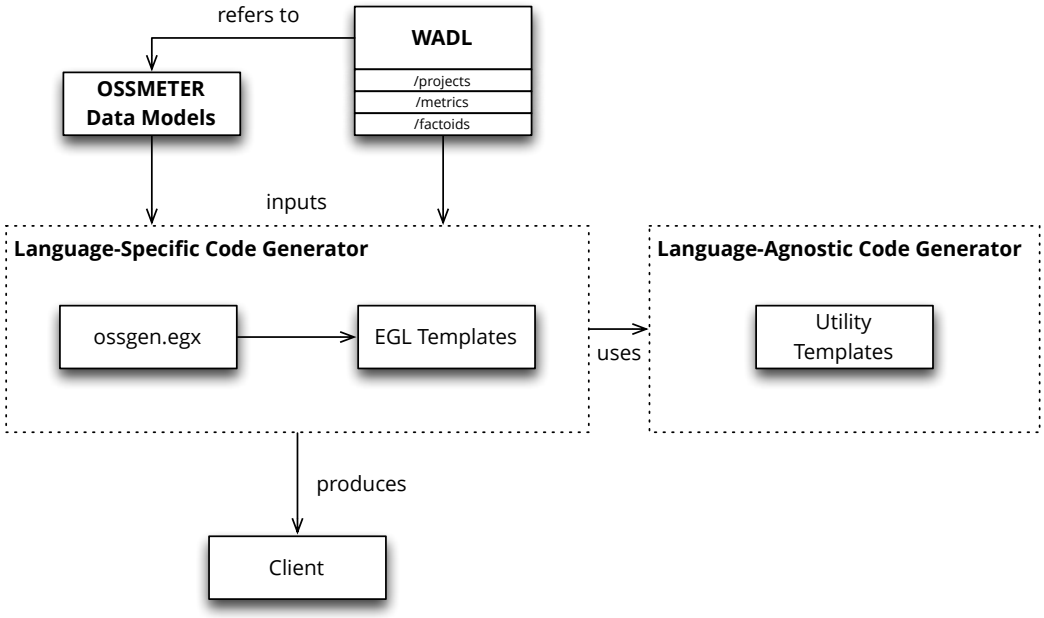

Figure 5: Conceptual diagram of the OSSMETER generator.

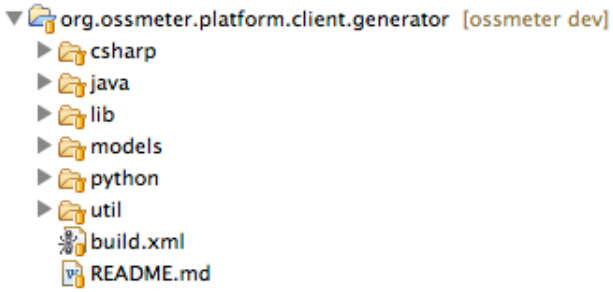

Figure 6: The structure of the code generator project.

plugins, as well as a WADL specification, describing the OSSMETER REST API. This means, that if the platform is extended with a new manager, then the data model of this new manager has to be copied in this model folder. In future versions of the platform, this process will be automated. The contents of the `models` file are illustrated in Figure 7. Finally, there is one directory per supported language. These directories contain the EGL [3] templates for generating code for a given language.

The code generation process is driven by the `build.xml` ant script. This script loads all the ecore models in memory and then executes the corresponding templates. If a user wants to execute the generator, he or she has to run this ant script by right-clicking on it and selecting *`Run As -> Ant Script`*. If an engineer wants to provide support for addition languages, he or she has to follow the following steps:

1. Create a folder with the language's name, e.g. "haskell", and include the EGL templates for the new language.

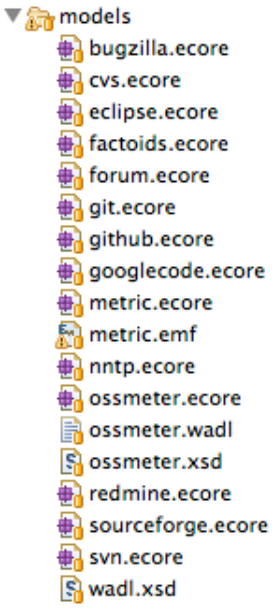

Figure 7: The models used by the OSSMETER code generator.

2. Edit line 113 of the `build.xml` to include the new language in the list, e.g. "`<for list="java,csharp,haskell" ...`".

7 Compliance to OSSMETER Requirements

In work package D1.1 [5] the requirements of the OSSMETER project are listed. In this section, we show how the work described in this document relates to the requirements that are relevant to work package 5. Table 9 defines a colour-coding scheme for expected requirement compliance. In Table 10 we list the requirements of the OSSMETER platform provided by the project's research partners, and colour them using the scheme defined in table Table 9 to show whether they have been satisfied by the work presented in this document. Moreover, in Table 12 we list the requirements provided by the project's use case providers.

Table 9: Color-coding scheme for requirement compliance.

Compliance	Description
Full Compliance	Full compliance is achieved
Partial Compliance / Not yet known	Partial compliance is achieved or expected compliance is not yet known.
No Compliance	No compliance is expected

Table 10: Requirements from research partners.

Workpackage	ID	Description	Priority	Status
WP5	78	The user shall only see the result of predefined queries.	Shall	This is supported by the provided REST API.
WP5	81	The user shall be able to export metrics to external data formats (e.g. CSV).	Shall	Users can export data in JSON format by using the REST API.
WP5	82	The platform shall provide a web-based API through which 3rd party applications will be able to connect to it.	Shall	A REST API has been implemented.
WP5	84	The user shall be able to visualize the metrics.	Shall	This is supported by having the REST API return visualisation-specific data.

8 Summary

This document has presented the motivation and specification for the OSSMETER REST API. First, we presented the two main approaches to application interaction, and we justified why we used the REST paradigm to provide such functionality to the OSSMETER platform. Moreover, we presented how the REST API is distributed, and the different resources it provides. Next we introduced a code

Table 12: Requirements from use case partners.

Workpackage	ID	Description	Priority	Status
WP5	4	Provide a user interface to OSSMETER services that uses a web browser	SHALL	A user interface is provided in the form of a web application. This application uses the platform's REST API.
WP5	67	Provide an open API to make available OSSMeter services to other systems	MAY	A REST API has been implemented.

generator, which can be used to quickly generate client code against the OSSMETER REST API. Finally, we related the requirements set out in D1.1 to the work delivered in work package 5, and showed the level of compliance with each requirement.

References

[1] D. Kolovos, L. Rose, R. Paige, and A. Garcıa-Domınguez. The epsilon book, Dec 2010.

[2] Mares and Peter. Art of Java Web Development. *Computer Journal*, 48(2):253, Mar. 2005.

[3] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. C. Polack. The Epsilon Generation Language. In *Proceedings 4th European Conference on Model Driven Architecture Foundations and Applications*, Berlin, Heidelberg, June 2008. Springer-Verlag.

[4] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[5] The Open Group. *D1.1 - Project Requirements*. Technical report, The Open Group, June 2013.

Abbreviations

ASP Active Server Pages

CGI Common Gateway Interface

COM Component Object Model

CORBA Common Object Request Broker Architecture

HTTP Hypertext Transfer Protocol

JSON JavaScript Object Notation

JSP Java Server Pages

RMI Remote Method Invocation

UDDI Universal Description, Discovery and Integration

WADL Web Application Description Language

WS Web Service

WSDL Web Service Description Language

XML Extensible Markup Language