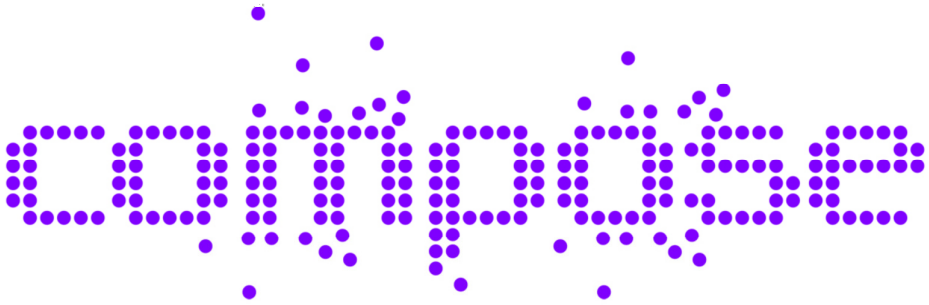

# Collaborative Open Market to Place Objects at your Service


## D3.1.2.1

## Service recommender – First prototype

| | | |
|---|---|---|
| **Project Acronym** | COMPOSE | |
| **Project Title** | Collaborative Open Market to Place Objects at your Service | |
| **Project Number** | 317862 | |
| **Work Package** | WP3.1 Service Management | |
| **Lead Beneficiary** | OU | |
| **Editor** | Luca Panziera | OU |
| **Reviewer** | Dave Raggett | W3C |
| **Reviewer** | Juan David Parra Rodriguez | UNI PASSAU |
| **Dissemination Level** | PU | |
| **Contractual Delivery Date** | 30/04/2014 | |
| **Actual Delivery Date** | 30/04/2014 | |
| **Version** | V1.0 | |

# Abstract

*The service recommender is a subcomponent of the service management infrastructure. The aim of this module is to assess which are the best services or service objects according to several criteria that matches users' needs. By taking a set of candidate services as input, the recommendation is implemented through a three-step process composed of filtering, scoring and ranking activities. The recommender returns a ranked list of services, which is a subset of the input list. Top-ranked services are the best according to recommendation criteria.*

*For the first prototype of the recommender, filtering and scoring is implemented for evaluating the following criteria: (i) service trust, (ii) service usage and (iii) sensor status of service objects.*

*This document describes the high level architecture of the recommender and the novel algorithms and techniques that implement the overall recommendation process for the first prototype.*

# Document History

| Version | Date | Comments |
|---|---|---|
| V0.1 | 16/04/2014 | First draft |
| V1.0 | 29/04/2014 | Final version |

# Table of Contents

# List of Figures

# Acronyms

| Acronym | Meaning |
|---|---|
| COMPOSE | Collaborative Open Market to Place Objects at your Service |
| NFP | Non-Functional Property |
| QoS | Quality of Service |
| REST | Representational state transfer |
| API | Application programming interface |
| OWL - DL | Web Ontology Language - Description Logic sublanguage |
| USDL | Unified Service Description Language |

# 1 Introduction

The service management work package is based on three main components (see Figure 1):

1. An advanced linked services discovery engine, whose job is to discover distributed and heterogeneous COMPOSE entities. The service discovery engine is layered on top of a service registry, which exploits information retrieval and semantic search and storage technologies.
2. An advanced service recommender system, which is in charge of suggesting new relevant services based on users' previous interactions, similarity between services, and other non-functional properties such as performance, trust, etc. This sub-component, emphasized in Figure 1, is *the focus of this deliverable*.
3. An assisted service composition engine, which is meant to help users create new composite services by (semi) automatically combining existing services to obtain the desired functionality.

Both the service recommender and the service composition engines leverage the service discovery engine.

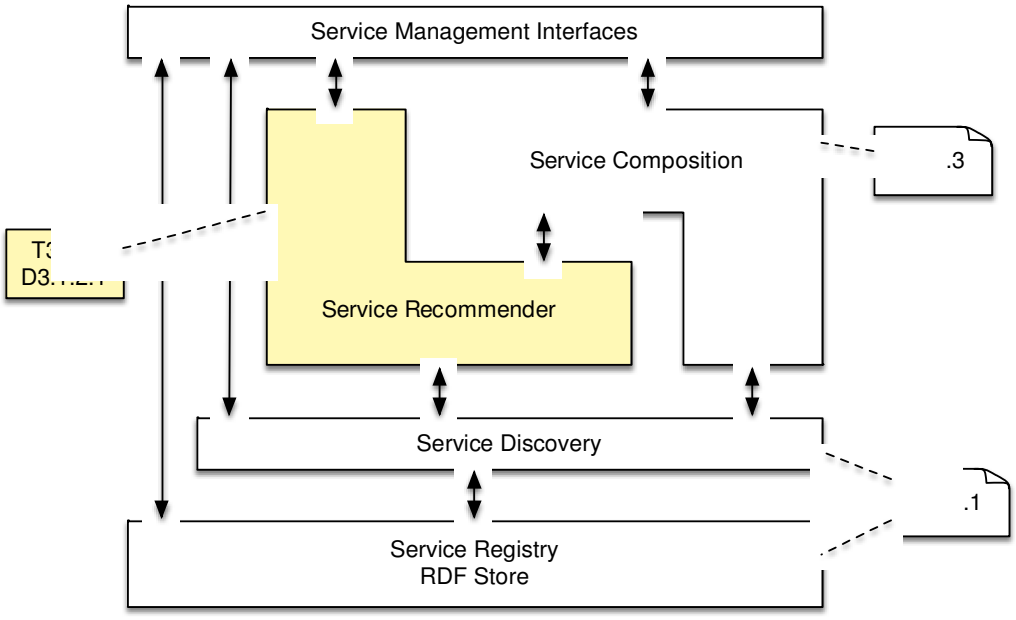


**Figure 1: A simplified logical view of service management components**

According to deliverable D1.3.1, services are software components, which provide functionalities through the Web. Services can be external Web APIs, sensor, actuators and smart objects. Services can be also results of service compositions that are implemented through the COMPOSE platform.

The objective of the service recommender is to provide functionalities for suggesting the best service according to criteria that are relevant for users. The service recommendation is implemented through a ranking of services, which are returned by the discovery engine, based on an evaluation of several service characteristics. The first prototype of the recommender will evaluate the following characteristics of services:

- trust level;
- service usage;
- sensors status associated with a service.

The recommender has been designed through a modular and flexible architecture that allows COMPOSE platform providers to remove and integrate service evaluators in a seamless way. As well as the composition engine, the recommender is managed as a plugin by the discovery engine based on iServe (details in deliverable D3.1.1.1).

The scientific contributions of the development of this prototype are the following:

- a generic and extensible approach for a service trust computation on personalized[1] trust criteria;
- a novel metric for estimating service usage based on the exploitation of Web data;
- a specific metric for sensor quality based on a combination of data analytics techniques on sensor streams.

# 2 Prototype overview - short description

This section provides details about the design of the service recommender. The design is based on the following approach. As an initial step, actors that interact with the recommender are identified. Then, an overall architecture of the software component is introduced. Finally, descriptions of the main functionalities and main design choices are provided.

## 2.1 Architectural responsibilities and interactions

Four actors that interact with the service recommender are identified, as show in Figure 2: developer interface, composition engine, discovery engine and platform provider. A graphical user interface will provide recommendation functionalities to developers that use the COMPOSE platform. The composition engine (T3.1.3) uses the same recommendation functionalities.

Service recommendation includes three activities: filtering, scoring and ranking. Filtering excludes services that do not meet particular criteria that are relevant for users. Scoring will assign a numeric value to each service, which assesses the service quality according to specific criteria (e.g., usage). Finally, the ranking will return an order list of services based on filtering

[1] By the personalization, we mean a customization of trust criteria according to the service users' preferences, which are given by the users or developers explicitly.

and scoring results. The configuration of there three activities, therefore of the recommendation logic is defined by the provider of the COMPOSE platform.

The role of the discovery engine is to provide lists of services that match specific functional characteristics (e.g., weather services). The recommender takes one of these lists as input, in order to recommend the best services that provide similar functionalities.

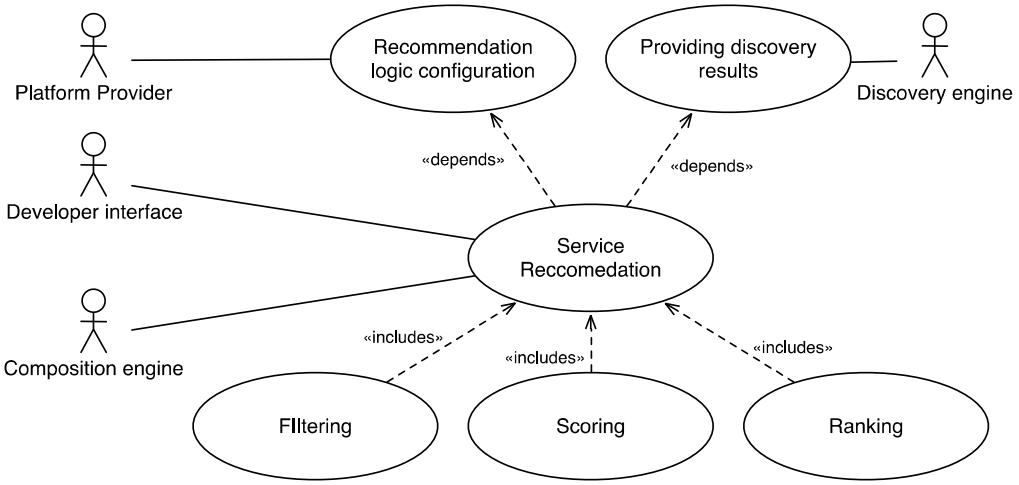


**Figure 2: Actors and interactions with the recommender**

## 2.2 High level design

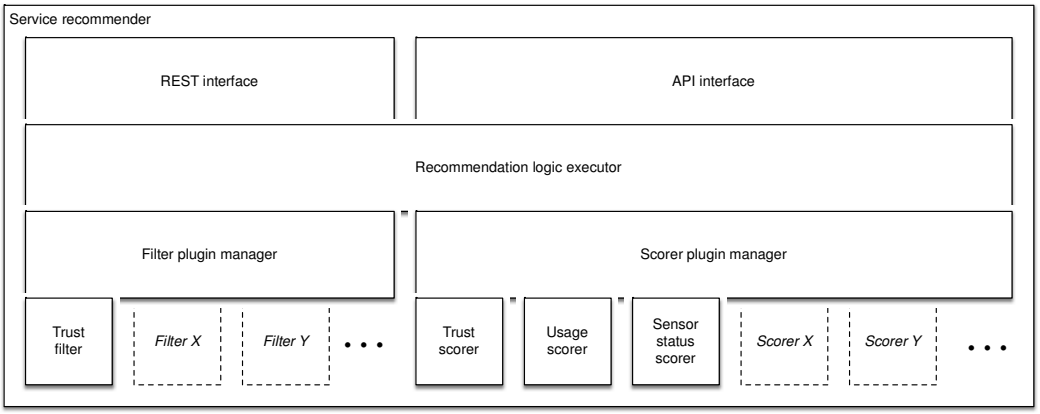


**Figure 3: Service recommender architecture**

The architecture of the service recommender is composed of three layers, as shown in Figure 3. The first layer provides the interface to the component, which is implemented through REST and APIs of the language adopted for the implementation. The RESTful interface provides

access to the recommendation functionalities to third party software as a service. In contrast, the interface based on native APIs allows COMPOSE platform modules a more efficient communication with the service recommender.

The second layer is the executor of the recommendation logic. This logic implements filtering, scoring and ranking according to the configuration defined by the provider of the COMPOSE platform.

The filter plugin manager and the scorer plugin manager are the components of the third layer. The first module manages filters. Filters are components with a common interface. Each filter assesses if a service meets specific criteria (e.g., a minimum trust level), therefore if the service must be excluded by the recommendation results. The latter manager runs scores. Similarly to filters, scores share a common interface. Each scorer evaluates a characteristic of each specific service (e.g., service usage). The result of the evaluation is a score that represent the quality of the service characteristic. Scores are combined in a global score that will be used to rank services.

Both mangers will invoke, respectively, filters and scores according to the recommendation logic defined by the platform provider. The management of filters and scorers as plugins has several advantages:

- to improve the efficiency and effectiveness of the recommender by adding or substituting filtering and scoring strategies;
- to adopt different recommendation approaches based on the service domain;
- to tune the recommendation in order to fulfil better user needs.

## 2.3 Main functionalities introduced

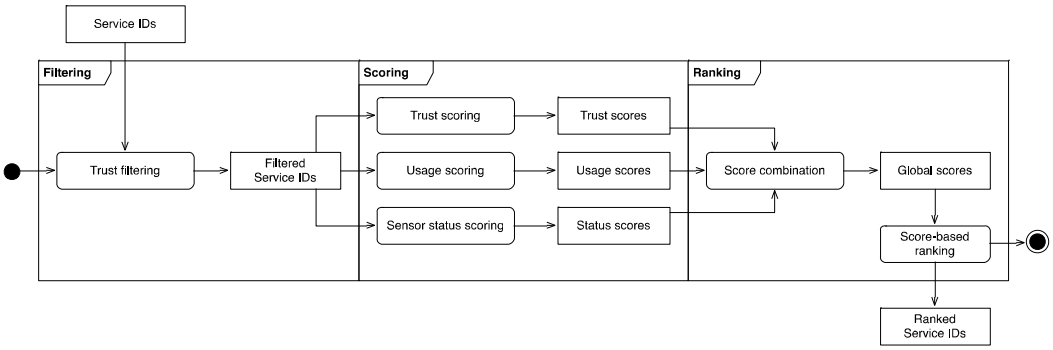


**Figure 4: Service recommendation process**

The recommendation process is composed of three main activities, as introduced in section 2.1: filtering, scoring and ranking. These activities are sequential and composed of sub-activities as shown in Figure 4. For this first prototype, the first phase performs only trust filtering, which returns a filtered list of Service IDs.

The second phase performs three different scoring strategies. For each service, scores are returned for evaluating (i) service usage, (ii) status of sensors that are service components and (iii) trust.

The ranking phase combines all the scores computed for each service trough an average in order to produce a global score for each service. Each global score is exploited for ranking services.

## 2.4 Main Design choices

The service recommender is implemented in Java by exploiting Guava libraries. Guava provides efficient manipulation functions of data structures therefore it allows an efficient implementation of the recommendation process.

### 2.4.1 Filters and Scorers

Filters and scores are the main modules that implement the recommendation logic. These components are implementations, respectively, of the `Filter` and `Scorer` Java interfaces. The following listing shows the two interfaces.

```
public interface Filter {
  public boolean apply(URI serviceId);
}

public interface Scorer {
  public Double apply(URI serviceId);
}
```

Each Filter implements its filtering logic through the `apply` method. This method returns `false` if a service must be excluded because it does not meet some criteria (e.g., an acceptable trust level), otherwise it returns `true`.

A scorer apply method implements the algorithm that assess a particular service characteristic (e.g., service usage). The method returns a value between 0 and 1. The higher the returned value, the higher is the assessment of the characteristic. For instance, a scorer that assesses usage of services returns a value close to 1 for services, which are often very used. Instead, the scorer returns a value close to 0 if a service has low usage.

### 2.4.2 Trust evaluation

"Trust (or, symmetrically, distrust) is a particular level of the subjective probability with which an agent will perform a particular action, both before it can monitor such action (or independently of his capacity of ever to be able to monitor it) and in a context in which it affects the agents' own action"[1]. In the scope of the COMPOSE vision, trust is understood in a

similar way, as an evaluated expectation a user of a service[2] has about the service behavior, before the service is used in a particular context. Being subjective, the trust perception may, therefore, change in different contexts, or be different in the same context for different users, given users past experiences and skills.

A first prototype of the COMPOSE trust module is built on top of a trust goal classification approach introduced in [2]. In that approach, trust promises of services are matched against a trust requirement by a classification technique to identify services that fit (classify into) the requirement. The trust requirement, or trust criteria, is a set of trust criterion. A trust criterion is a triple `<trust attribute/indicator, value, relevance>`.

For example, a trust criterion in a context of leisure services (e.g. a weather info service), stated as "I trust services having a good reputation and being popular", may be specified as trust criteria shown below:

```
{<'reputation', 'good', '1'>, <'popularity', '0.5', '1'>}
```

Or, another example, in a context of services dealing with sensitive data where a service security aspect is an important trust indicator: "I trust services having high reputation, ensuring data confidentiality using TSL/SSL protocol, but better if TSL protocol, and having authorization in means of Tokens. Security is more relevant than reputation"

```
{<'reputation', high, '0.5'>, <'confidentiality', 'tls', '1'>,
<'confidentiality', 'ssl', '0.8'>, <'authorization', 'token', '1'>}
```

An evaluation of service trustworthiness is a process of collecting trust related information about services, and then calculating a trust index, in regards to the given trust criteria.

Particularly, by the trust related information (so, trust indicators) we may consider:

- reputation, including accumulated service popularity and ratings in COMPOSE platform;
- service security properties, such as stated guarantees about an authorization, authentication, or data encryption, and/or results of intrinsic security evaluation (e.g. inspection of security states in information flow analysis, or detections of compliance of service contracts with service runtime behavior);
- QoS guarantees such as a service availability or a service object's data accuracy or data freshness.

Important to note, the COMPOSE trust module relies on other COMPOSE components that actually will be providing data, or values or indexes of these trust indicators. It is out of scope of this deliverable how each of these is being evaluated. In particular, the WP5 Reputation Server, Security Analysis and Activity Monitoring will be providing a service reputation index,

[2] Or, composite services, or simple or composite service objects. However, in short writing, they are all named as services.

popularity and rating data, and security properties and contracts. Then, WP3.2 Service monitoring is expected to provide data about QoS guarantees.

Therefore, as the first prototype of the trust module is being delivered in a scope of this deliverable, which is in the terms of milestones before the Reputation Server, Security Analysis, and Service Monitoring prototypes, the first prototype focuses on (1) the implementation of a trust model, (2) trust index computation of simple service, and (3) high-level specification of integration interfaces. A second version of the prototype, which is expected to be integrated with other WP components, will address rest of concerns.

### High Level Requirements

The trust module has to support answering these questions:

- Is a simple service trusted, based on available information about it and given trust criteria?
- Is a composite service trusted, based on available information about it, information about services it is composed of, and given trust criteria?
- Is a service trusted, in a scale relative to others given services?

Trust evaluation should be feasible in the context of subjective (user-provided) or global trust criteria, on an absolute scale or scale relative to other service. When a service user does not provide a trust criterion, then a global trust criterion is set and used.

### High Level Design

The prototype has three high-level components, namely Collectors, Trust Scoring and Ranking, and Knowledge base, as shown in Figure 5.

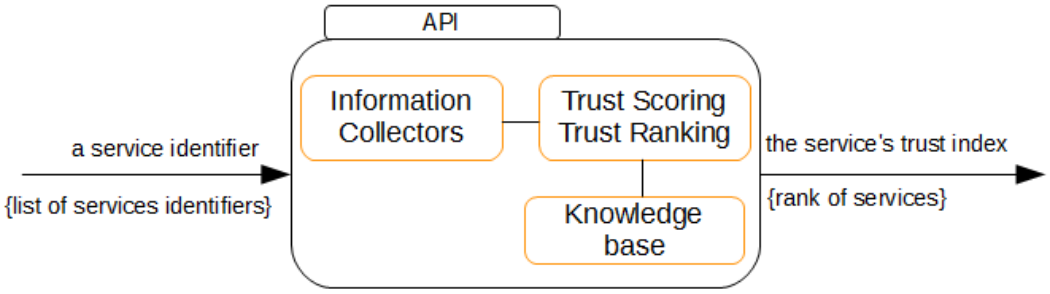


**Figure 5: A high-level design of trust module prototype**

Upon a trust evaluation request, the Collectors collect descriptions and behavioural information about the services, or other COMPOSE entities (e.g. user profiles), by invoking COMPOSE Reputation Server, Security Analyzers, and COMPOSE Service Monitor APIs. Then, the trust scoring and trust ranking algorithms resolve the trust index of an entity, either on an absolute scale or relative to other entities, by evaluating the gathered information with the trust criteria.

Knowledge base contains a semantic trust model including other models, such as a model for service security aspects.

### What is in the prototype

**(1) Models.** A generic semantic trust model (in OWL-DL) is developed to allow uniform and semantic representation of both trust criteria and trust evaluation related attributes of services. Identified initial set of trust attributes is organized into semantic taxonomy of non-measurable and measurable attributes. For example, service security properties such as authentication, certificate, or assurance of confidentiality, are non-measurable, descriptive attributes. A reputation index may be considered as a measurable, quantified attribute, with the measure on an ordinal scale (e.g. ['bad', 'medium', 'high']).

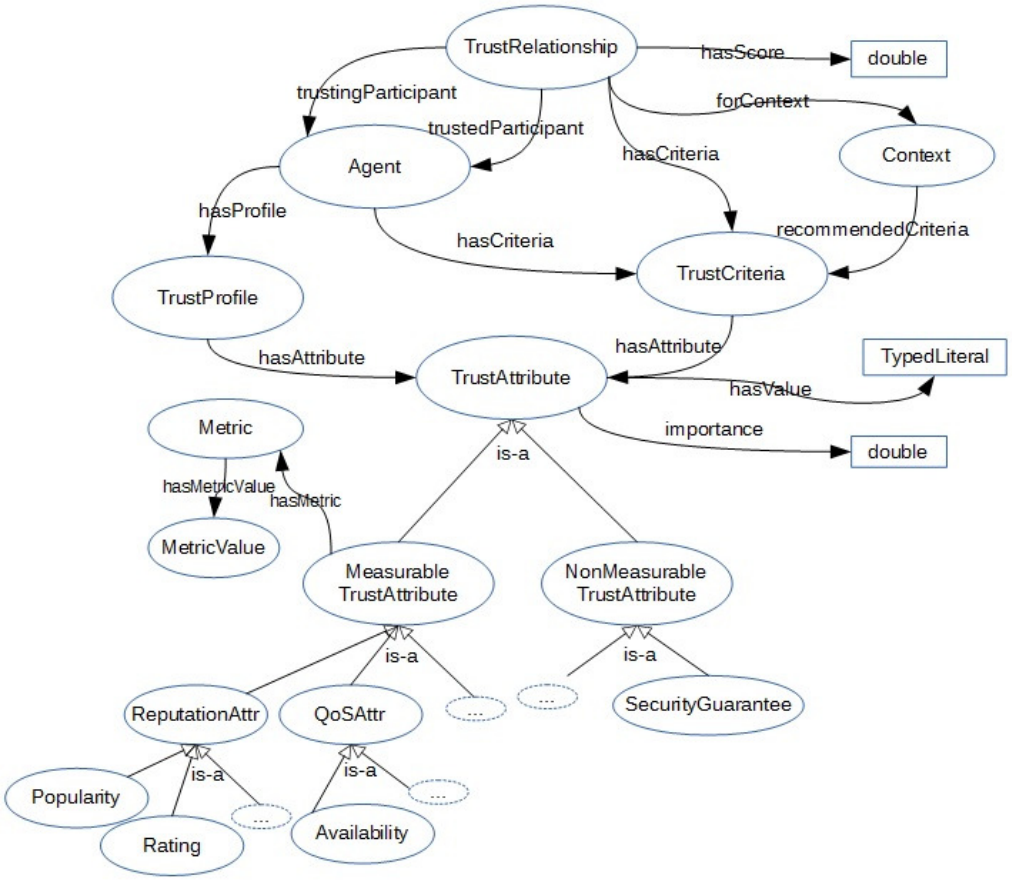


**Figure 6: Semantic trust model**

As for the service security properties, the first prototype relies on USDL-Sec [3]. USDL-sec is a lightweight semantic vocabulary for describing service security guarantees such as authorization or confidentiality. A decision to support USDL-Sec in the prototype is because of its simple, however, informed expressivity. Particularly, USDL-sec makes security guarantees described in different levels of an abstraction including *security goals* (e.g., Authorization, Authentication, or

Confidentiality), then specific *security mechanisms* (e.g. Token, Encryption) to achieve the goals, and finally, concrete *security technologies* (e.g. OAuth, OpenID, or SSL) that implement the mechanisms. Such level of granularity is suitable for evaluating the trust on different levels of trust criteria details. As USDL-Sec does not provide a taxonomy of security technologies, for the purpose of the trust module prototype implementation we built an initial one. All the developed semantic models are available at < http://goo.gl/mscAUQ>

**(2) Scoring.** The trust index (*t)* is a real number in the range [0,1]. It quantifies the match between the requested trust attributes and those corresponding service characteristics, defined by semantic descriptions. The *t=1* means that given service is fully trusted, while *t=0* mean it is distrusted, in respect to the trust criteria. The first prototype of trust module implements the computation of trust index of simple services. The computation is defined by the following relation (1).

$$t = \frac{\sum_{tp}^{n} \left( Ev(tp, D) * relevance(tp) \right)}{\sum_{tp}^{n} relevance(tp)}; for each tp \in C. t \in [0,1] \quad (1)$$

*C is a trust criteria. Tp is a trust criterion. D is a dataset of service/ serviceobject NFPs*

The relation (or expression) is a weighted sum of the values obtained by Ev(tp, D) function. *Ev(tp, D)* evaluates a service attributes (i.e. non-functional properties - NFPs) with regards to the given trust criterion *tp* and returns a normalized value on a scale between 0 and 1. Normalization is required as properties or criteria can be of different dimensions. We use a linear normalization $\frac{value(tp, D)}{valueMax(tp)}$, where *value(tp, D)* is a value of a service property corresponding to the *tp*, and *valueMax(tp)* is the maximum [possible] value of *tp* indicator.

For example, assume a reputation as a trust property, with reputation index on an ordinal ['bad', 'medium', 'high'] scale with relative degree of difference between values. Then, if a trust criterion is 'reputation, at least ‘medium', and service has reputation index ‘medium, then 0.66 is returned, as $\frac{relativeDegreeOf\ 'medium'}{relativeDegreeOf\ 'high'} = \frac{2}{3} = 0.66$. Or, if popularity is in a range of [1, 10], and a service popularity index is 5, then normalized value of the popularity will be $\frac{5}{10} = 0.5$

Semantic similarity computation evaluates matches between service security guarantees and corresponding trust criterion, both expressed using USDL-Sec. The semantic similarity is also a value in the range of [0,1]. The prototype uses a SML - Semantic Measurement Library [4]. SML implements a number of different algorithms for measuring semantic similarity. Provided by the SML, the trust prototype uses the Lin's [5] measure, as suitable for evaluating similarity of concepts in lightweight IS-A taxonomies such as the developed taxonomy of security technologies.

The Java API for trust scoring is defined as follows:

- `Double` **`obtainTrustIndex`**`(URI resource, TrustCriteria criteria)` - Answers a trust index for a resource in regards to the specified trust criteria passed by a resource consumer

- `Double` **`obtainTrustIndex`**`(URI resource)` - Answers a trust index for a resource according to the global trust criteria set in a trust module.
- `Boolean` **`isTrusted`**`(URI resource)` - Answers if resource is trusted in terms of the global trust criteria and trust index threshold value of 0.5
- `Boolean` **`isTrusted`**`(URI resource, TrustCriteria criteria)` - Answers if resource is trusted in terms of the given trust criteria and trust index threshold value of 0.5
- `void` **`setGlobalTrustPerception`**`(TrustCriteria criteria)` - Sets global trust perception parameter of the trust module.

**(3) Ranking**. In addition, the trust module implements ranking to compute service trustworthiness on a scale relative to other given services. The ranking is defined as a multi-criteria decision making (MCDM) problem of different alternatives selection, with services being considered as alternatives and with a given trust criteria. The ranking is implemented using a TOPSIS (Technique for Order of Preference by Similarity to Ideal Solution) approach for MCDM [6]. Using TOPSIS, the prototype ranks a set of services by two steps. First, normalising and weighting evaluated score for each trust criterion, for each service. Second, resolving a service rank by calculating for the each given service its relative geometric distance from the ideal positive and ideal negative solution. The ideal positive solution is one with the best score in each criterion, while the ideal negative is one with the worst score in each criterion.

The Java API for trust-based ranking is defined as follows:

- `List<Tuple2<URI,Double>>` **`rankResources`**`(List<URI> resources, TrustCriteria criteria, boolean excludeIfAttributeMissing, OrderType order)` - Answers ranking of resources and their scores, in regards to the trust criteria, using TOPSIS decision strategy. If a 'excludeIfAttributeMissing' parameter is set true, then resources evaluated as not having at least one requested trust attribute are excluded from the ranking.

**(4) Integration Interfaces.** The `Collector` Java interface is defined for integration with COMPOSE components that will be providing mechanisms and information to the trust evaluation. Each specific `Collector` (e.g. `ReputationCollector`) must be registered in the trust module and implement a method with the following signature.

```
Model collectInformation(String resourceIdentifier)
```

The method, for a given resource identifier (e.g. a service URI), obtains the requested information for trust evaluation, and returns a `Model` with the information populated. Actual request can be a Web service/API call to retrieve a JSON object, or in other similar means. For an example, invocation to the WP5 Reputation server to resolve reputation information will be as follows.

```
GET http://localhost:8080/Reputation/runningservice/{serviceId}
```

Finally, the trust module implements a `Filter` and a `Scorer` interfaces (see Section 2.4.1) for integration with the Service Recommender. The two implementations are respective `TrustFilter` and `TrustScorer`.

### External API

The first prototype of the trust module is provided as a web service, too. It can be invoked with the following URL pattern:

GET: *http://host:port/trusthings?srvcid={serviceId}* - for trust scoring

or,

GET: *http://host:port/trusthingsrank?srvcid={serviceId}* - for trust ranking

where *{serviceId}* is a parameter list of service identifiers. The web service returns a JSON object. E.g.

```
Response Header:
Content-Type:application/json; charset=UTF-8
Status Code: 200
{
  "success" : "true",
  "indexes" : [ {
    "serviceUri" : "www.compose-project.eu/iotservices/example1",
    "index" : 0.6,
    "rank" : 1
  }, {
    "serviceUri" : "www.compose-project.eu/iotservices/example2",
    "index" : 0.325,
    "rank" : 2
  } ]
}
```

A returned JSON object contains a trust index of the service (attribute “index”), and the rank (attribute “rank”) in case of a list of service URIs sent to the web service. In a case of error, a JSON response is sent

```
Response Header:
Content-Type:application/json; charset=UTF-8
Status Code: 500
{
  "success" : "false",
  "message" : "error message text"
}
```

The trust web service is implemented using the Vert.X [7] polyglot application framework for web applications. The trust web service as a Vert.X module is developed for Java Virtual Machine. Developed module is available for download online at http://goo.gl/mscAUQ.

### 2.4.3 Usage evaluation

For this first prototype, the usage evaluation is performed for Web APIs available on the Web, which can be loaded in the service registry of the COMPOSE platform.

In the literature [8,9], usage measurement of Web APIs is used for estimating the popularity of services. The most adopted approach for estimating usage is counting Web API occurrences in composite applications, such as mashups. For instance, if a Web API is a component of 300 mashups, it is more used than one that appears in 50 mashups. This approach can be applied for composite services on the COMPOSE platform, but it suffers of a cold start problem that is typical of recommender systems. Until the amount of composite services stored in the platform is not statistically representative for each Web API, it does not provide a trustworthy estimation of usage.

In order to solve this issue, state-of-the-art solutions measure usage by exploiting ProgrammableWeb[3], an online repository of more than 11000 Web API descriptions [8,9]. Nowadays, this website also provides descriptions of more than 7000 mashups. Unfortunately, only 1300 services are components of these mashups, therefore usage cannot be measured for most Web APIs.

The novel approach implemented by the first prototype of the recommender is based on the exploitation of Web data that allows the estimation of criteria that drives Web API usage by developers. Through the measurement of these criteria, we estimate usage.

In order to figure out these criteria, a questionnaire has been submitted to 56 Web developers. Survey results show that the two main criteria are (i) popularity of the provider and (ii) quality of community that support developers' issues. The first criterion highlights that users prefer Web APIs with popular providers. The second one shows that users prefer Web APIs, which provide communities, through forums or mailing lists, that better support developers.

The provider popularity is measured as the number of daily visitors to providers' websites. Our assumption is that the more a provider is popular, the more users visit their website. Alexa[4], a service that monitors network traffic of websites, is used for evaluating provider popularity.

The quality of community support is measured as the number of daily active community members. This metric, which we call "community vitality" for short, is defined on the base of an additional survey. The users of the former survey evaluated the utility of eight metrics proposed in the literature for measuring the health of a community [10]. Survey participants assessed that community vitality is the most useful metrics for assessing support quality. For these measurements, data have been collected from 110 Web API support forums available on Google Groups.

To verify that provider popularity and community vitality can estimate usage, correlation metrics have been computed. On data collected between 2005 and 2013, the correlation

[3] http://www.programmableweb.com/

[4] http://www.alexa.com

evaluations of measurements between the two metrics and usage, computed as number of mashups, show that:

- Provider popularity is a good usage estimator for Web APIs that are not provided by organizations which offer services in multiple domains (e.g., Google, Yahoo!, Microsoft, Amazon, etc.);
- Community vitality is a good estimator of usage for most Web APIs.

On the base of the correlation results, the usage $u_a$ of a Web API $a$ is computed as follows:

$$u_a = \begin{cases} \frac{v_a}{\max(V)} & \text{if } v_a \in V \\ \frac{p_{\rho_a}}{\max(P)} & \text{if } p_{\rho_a} \in P \\ 0 & \text{otherwise} \end{cases}$$

where $v_a$ is the vitality of the community that support $a$ and $p_{\rho_a}$ is the popularity of the provider $\rho$ of $a$. $V$ is the set of community vitality measurements and $P$ is the set of popularity measurements for providers that focus on a single domain. Both metrics are exploited for estimating usage because communities' data is not always available or collectable due to licensing restrictions. In addition, Alexa offers data for almost all of Web API provider websites available.

Each measurement of $u_a$ is defined as follows:

$$v_a = \left|U_{c_a,t}\right| \text{ and } p_{\rho_a} = \left|\beta_{\rho_a,d}\right|.$$

$U_{c_a,t}$ is the set of users that posted a message on the forum or mailing list provided by the Web API $a$ in the time period $t$. $\beta_{\rho_a,t}$ is the set of visitors on $\rho_a$'s website during the day $d$. The size of $t$ is 60 days and $d$ represents the most recent day in $t$. Vitality is measured on large time intervals because the amount of daily active community users has a big variance, therefore measurement are affected by noise. By evaluating vitality in a big interval, the noise is smoothed, therefore the measurement better figures out community behaviours. The size of $t$ has been defined empirically based on the Google Groups dataset. We observed that the maximum delay between two posts in the dataset is 27 days. Therefore, the $t$ size has been defined at least twice this period in order to have a statistically representative measurement composed of at least one active user of the community.

### 2.4.4 Sensor status evaluation

The objective of this module is to set a score on the sensor quality by applying a combination of data analytics operations on the data streams a sensor carries. The main idea is to provide a data-driven quality measurement, which does not require further attributes but the data itself. This is, of course, intended to behave as an additional metrics to be analysed together with further attributes.

Thus, in order to keep the decoupled approach and a minimalistic set of requirements, the quality scorer relies solely on the very basic attributes of a sensor data stream: the value of the

measurement and its timestamp. Typically, this information extracted from sensors forms a time series, which is a list of consecutive measures or samples characterized by these two basic parameters. A description of the implementation of such analytical flow is provided later in this section.

For the analysis, the first step was gathering data to develop the analysis so, as a first round, we have gathered data from environmental sensors from Xively[5]. In particular, and due to the disparity and poor quality of the air pollution measurements, we have run the first trials using the following amount of sensors:

- 18 humidity sensors
- 22 temperature sensors
- 12 air pressure sensors

For a timeframe ranging from 2011-01-01 to 2014-01-01 the total amount of collected data has reached 450 MBs. Together with the availability and the relevance for the overall project domain, additional criteria has been evaluated when choosing environmental data as the dataset for building the prototype. Phenomenon described by environmental data follows the cyclic patterns on which the applied analytic operations rely. In addition, the variation between two consecutive samples of sensor data is expected to be gradual and soft, so assumptions on the expected behaviour of data can be defined and verified.

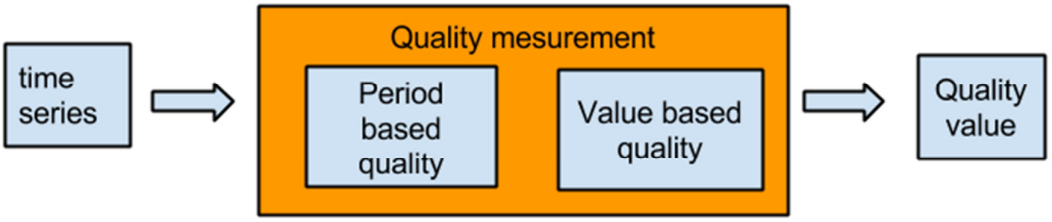


**Figure 7: High-level schema of the data quality algorithm**

Following this reasoning, the algorithm defined for sensor quality measurement is based on the temporal and metrical precision of the time series. The results range between 0 and 1, being 1 the value expressing a higher sensor quality and 0 the lower one. Actually, a value of 1 would mean that sensor is working perfectly both in terms of regularity and behaviour.

Next sections provide a description of the two separate analytic processes the algorithm embodies.

### Period based quality computation

[5] www.xively.com

In this process the temporal accuracy of the time series is used to evaluate its quality. We assume that a valuable time series is one that continuously delivers its measurements (no missing values) with a continuous period between measurements (no variation in its periodicity). The computation process is based on the following steps:

At first, an array containing the intervals between each two consecutive measurements is created:

$$L[n] = T[\,i+1] - T\,[\,i\,]\;\forall i : T(i)$$

Being:

- L[n] is interval array
- T [i] is the time series

Using this array, the measuring period (expressed in seconds) is estimated assuming that it should be the most common value in the array, in other words, is assumed that the mode of the array is the measuring period used to read the data. Once the period is calculated using this formula:

$$Q\ (Ts) = 1 + \log\ \left(\frac{1}{1+PL\,(\,L\,[\,n\,]\,)}\right)$$

Being:

- *Ts* is the time series
- *L[n]* is the interval array
- *Q(Ts)* is The quality of the time series ranging from 0 to 1
- *PL(L[n])* is the Precision loss : a value that computes the loss of precision of the time series.

*PL(L[n])* is calculated by:

$$PL\,(\,L[n]) = \left(\frac{\sigma}{Mo\,(\,L)}\right) \cdot R$$

Being:

*PL (L[n])* is the precision loss of Time series *L [n]*
**σ** is the standard deviation of time series *L*
*Mo (L)* is the mode of time series *L*
*R* is precision loss factor = 0.1

The described schema is summarized as shown in figure 8.

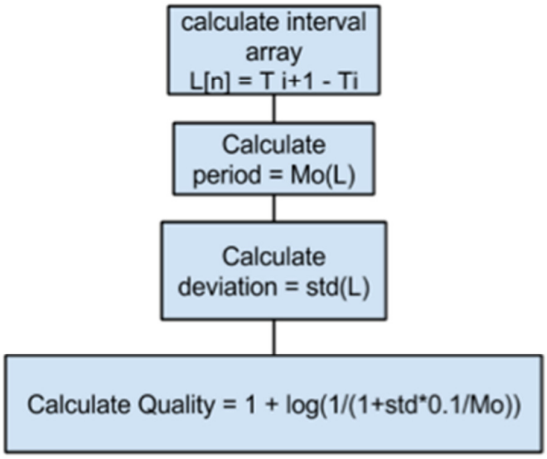


**Figure 8: Data quality measurement flow**

### Value-based quality computation

The value-based quality computation is based on the implementation of an anomaly detection process on the time series of the sensor streams. The first step is the creation of an array containing values of each measurement. Since we're using environmental sensors providing quite continuous measurements, we assume that a properly working sensor should provide data having a smooth variance between consecutive measures. This is checked by using an outliers[6] detection trough the interquartile method.

The **interquartile range** (IQR) is a measure of variability, based on dividing a data set into quartiles.

Quartiles divide a rank-ordered data set into four equal parts. The values that divide each part are called the first, second, and third quartiles; and they are denoted by Q1, Q2, and Q3, respectively.

- Q1 is the "middle" value in the *first* half of the rank-ordered data set.
- Q2 is the median value in the set.
- Q3 is the "middle" value in the *second* half of the rank-ordered data set.

The interquartile range is equal to Q3 minus Q1.

[6] An outlier is a value that "lies outside" (is much smaller or larger than) most of the other values in a set of data.

If this method is applied on a sorted list with the values, all the outliers should be places on the rest of the array (not inside the interquartile range). Once the sorting has been done, each outlier is identifies using:

$$Q3 + OF \cdot IQR < x < Q3 + EVF * IQR$$

Or

$$Q1 - EVF \cdot IQR \leq x < Q1 - OF \cdot IQR$$

Being:

- Q1 is the 25% quartile
- Q3 is the 75% quartile
- IQR is the Interquartile Range, difference between Q1 and Q3
- OF is the Outlier Factor, that is used with value 3
- EVF is the Extreme Value Factor, that is used with value equals to 3.5

Using this process, the algorithm identifies the outliers and the value returned of this calculation is the division between the number of outliers and the time series length. Since 1 is the best score, the final formula is expressed as:

$$1 - (Num_{outliers} \ / Num_{samples})$$

**Aggregation of both results**

The aggregation criterion penalizes those series with a low score as a result of any of the modules. So we directly use the product of both scores, as this is the best formula between the different options tried:

$$(Period\ Quality) * (Value\ Quality)$$

**Code developed**

The final tool has been developed in Java. Core class is called Scorer, and the entry point is the apply method, which triggers the execution of the analytics process. Sensor is identified by an URI, which acts as unique ID.

The current implementation provides a mock-up implementation of such identification mechanism. Currently, this URI is mapped to a directory that contains a set of JSON files containing the sensor measurements. The JSON files are defining according to the following schema:

```
{"data":[{"phenomenon":"temperature",
          "timestamp":"2011-01-28T18:25:04.332300Z",
          "observationId":4,
          "sensorId":"15749_1",
          "value":-6.00}…]}
```

Once the mapping from URI to folder path is done, the system scans the entire folder by reading de-serializing every JSON file it contains into the model classes (Timeseries and Samples). This is based on the assumption that every folder holds information of just one sensor. Once completed, previously described calculations are performed individually and then merged.

For value-based quality computation, the process uses a Weka[7] Interquartile Range approximation to process the outlier detection. Since it's also assumed that a single file is used for each day of sensor measurements (and therefore system must process multiple files for each sensor), the final evaluation score is the mean of the scores of each archives inside the sensor's folder.

# 3 Future directions

After the current deliverable, the main focus of the next prototype will be the integration of the service recommender with the COMPOSE platform and the analysis and improvement of performance in terms of scalability and effectiveness. Evaluation techniques implemented by scorers and filters will be also improved as shown in the following subsections.

## 3.1 Trust evaluation

A future realization of the COMPOSE trust module prototype will focus on: (1) implementation of the algorithm for the trust index calculation for composite services; (2) integration with the other COMPOSE components, from WP5 and WP3.2, that will be providing information and/or scores for trust index computation; and (3) scalability and performance, which has not been so far estimated in the first prototype of the trust module, mainly because the module has yet to be integrated with the rest of COMPOSE components.

## 3.2 Estimation of usage

According to the results provided by the first survey introduced in section 2.4.3, quality of documentation is an additional criterion that drives the adoptions of Web APIs by users. Metrics for evaluating documentation quality for Web APIs will be exploited for improving usage estimation.

## 3.3 Sensor Status evaluation

Two main research directions are identified in the sensor data analytics efforts described through section 2.4.4 in the current document.

On the one side, in terms of further analytics techniques to be applied two main options have been evaluated: the use of classifier algorithms for getting an automated classification of poorly

[7] http://www.cs.waikato.ac.nz/ml/weka/

identified incoming data streams and the development of prediction models on the accuracy of sensor data.

Being a challenging and yet widely open scenario, some initial approach on the predictive modelling on sensor streams is already defined. Given a set of ubiquitous data from sensor streams with a consistent amount of historical data, it is possible to develop prediction methods for short-term forecasting of the above-mentioned processes. Such prediction system can be based on machine learning methods that use regularities and patterns in historical data as well as contextual information (weather, major events, etc.) to forecast the evolution of the observed process along with uncertainty estimates and, in addition, to develop an auto-evaluation method that continuously provides a measurement of the prediction uncertainty in different time ranges.

# References


1. D. Gambetta (1988). Trust: Making and Breaking Cooperative Relations, Oxford: Basil Blackwell.

2. S. Galizia, A. Gugliotta, and J. Domingue (2007). A trust based methodology for web service selection. International Conference on Semantic Computing, IEEE, 2007.

3. USDL-Sec Vocabulary. http://forge.fi-ware.eu/plugins/mediawiki/wiki/fiware/index.php/FIWARE.OpenSpecification.Security.USDL-SEC

4. http://www.semantic-measures-library.org/

5. D. Lin (1998). An Information-Theoretic Definition of Similarity (), In 15th International Conference of Machine Learning

6. C.L. Hwang, K. Yoon, (1981). Multiple Attribute Decision Making: Methods and Applications Springer-Verlag, New York

7. http://vertx.io/

8. Gomadam, K., Ranabahu, A., Nagarajan, M., Sheth, A., Verma, K.: A faceted classification based approach to search and rank Web APIs. In: proc. of the IEEE International Conference on Web Services (ICWS). pp. 177–184 (2008)

9. Tapia, B., Torres, R., Astudillo, H.: Simplifying mashup component selection with a combined similarity-and social-based technique. In: proc. of the International Workshop on Web APIs and Service Mashups (MASHUPS). p. 8 (2011)

10. Rowe, M., Alani, H.: What makes communities tick? community health analysis using role compositions. In: proc. of International Conference on Social Computing (SocialCom). pp. 267–276. IEEE (2012)