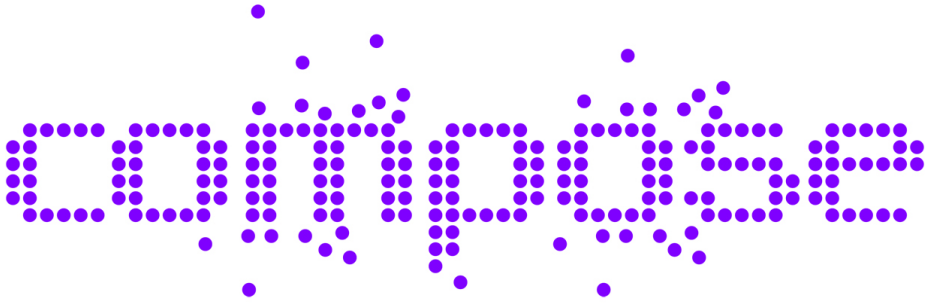

Collaborative Open Market to Place Objects at your Service

D2.1.1

Design of the object virtualization specification

Project Acronym	COMPOSE	
Project Title	Collaborative Open Market to Place Objects at your Service	
Project Number	317862	
Work Package	WP2 Object as a Service	
Lead Beneficiary	BSC	
Editors	Vlad Trifa, Iker Larizgoitia	EVRYTHNG
Reviewer	David Carrera	BSC
Reviewer	Dave Raggett	W3C
Dissemination Level	PU	
Contractual Delivery Date	31/10/2013	
Actual Delivery Date	30/10/2013	
Version	V1.0	

Abstract

The COMPOSE project aims to perform research leading to the development of an IoT platform that will easily enable relevant stakeholders to be engaged. Stakeholders include (i) developers, who wish to develop services and applications based on real-world smart objects (ii) Smart objects providers and owners who wish their smart objects to be exposed and available to developers, and (iii) end-users who wish to make use of existing services and applications. The goal is to create such a platform that will automatically take the burden off the identified stakeholders and enable each one to concentrate on their areas of expertise while leaving all systems related aspects and more for the COMPOSE platform to take care.

The main aim of this document is to present the requirements and protocol specification of two key parts of the COMPOSE relation to physical objects, mainly Web Object and Service Objects. In summary, a Web Object is any web-enabled device that can be directly connected to the COMPOSE platform. This connection is done by mapping the Web Object information to a Service Object, the internal representation of objects (sensors in general) inside the platform.

We present these elements in terms of the minimal requirements they must fulfil in order to be part of the COMPOSE platform. Additionally we provide two main aspects, first the information model followed by these entities and then the communication protocol, based on standard web standards that will govern the relation between the Web Object, the Service Object and any other component that communicates with them.

Document History

Version	Date	Comments
V0.8	19/09/2013	Merged version 0.7 with draft google docs. Structure and preliminary content ready for discussion.
V0.9	15/10/2013	Integrated comments from reviewer (W3C) Extended section 2 Added JSON objects for model and protocol
V0.95	29/10/2013	Integrated comments from reviewer (BSC)
V0.98	29/10/2013	Summary added.
V1.0	30/10/2013	Final formatting.

Table of Contents

List of Figures

List of Tables

Acronyms

Acronym	Meaning
COMPOSE	Collaborative Open Market to Place Objects at your Service
HTTP	Hyper Text Transfer Protocol
REST	Representational State Transfer
WO	Web Object
SO	Service Object
API	Application Programming Interface
NFC	Near Field Communication
CRUD	Create Read Update Delete
CSO	Composite Service Object
JSON	Java Script Object Notation

1 Introduction

The COMPOSE platform is defined to act as an enabler for the creation of applications for the Internet of Things. As a first step, in *D1.2.1 Initial COMPOSE Architecture*, the general architecture of the COMPOSE platform is presented.

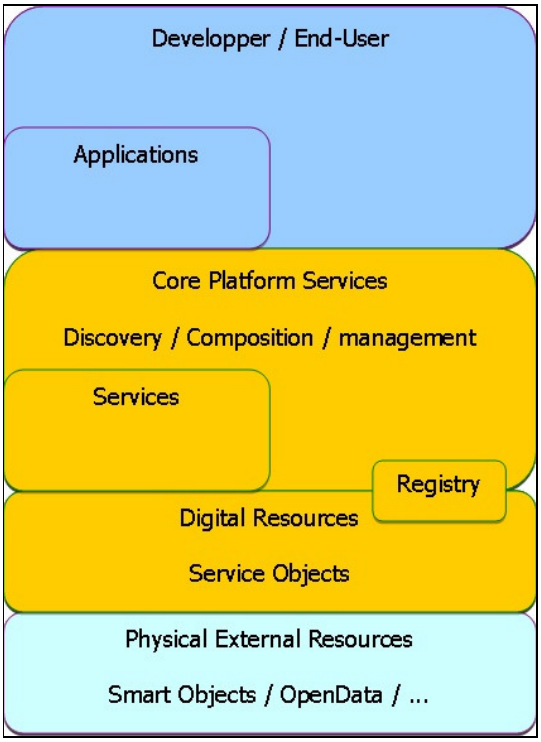

Figure 1: Architectural high level components

Figure 1 defines the whole stack of components defined in the COMPOSE platform. From the Developer / End user point of view to the low-level physical objects, which COMPOSE will enable to integrate.

In this deliverable we will focus on defining the connection between the physical objects and the COMPOSE platform, based on concepts and components defined as part of the architecture. In a nutshell, we will start from the description of an object, a physical entity that has some observable properties, readable by some mechanism. To connect to COMPOSE, we will define a Web protocol; therefore the objects need to speak Web protocols (mainly HTTP). If an object is able to speak HTTP, we define it as a Smart Object. If an object does not speak HTTP, it will require a proxy that does. For a Smart Object to be plugged into COMPOSE it needs to speak a specific REST Web based protocol that we will define. This is what we call a Web Object, i.e. external devices that are able to speak our protocol. Once they are speaking our protocol (called Web Streams) they will have a virtual identity inside the COMPOSE platform, what defines a Service Object. Service Objects can be combined inside the platform for mainly data processing related tasks, defining the Composite Service Objects and can be also integrated in bigger applications and services.

We will now summarize in more detail the key terminology we have adopted for COMPOSE, in order to have a precise understanding of the elements we are going to specify throughout the deliverable. For the purpose of this deliverable we will focus on the lower components of COMPOSE, mainly the Web Objects, the Service Objects. The Composite Service Objects will be presented in detail in *D2.3.1 Design of the object composition specification and components)*.

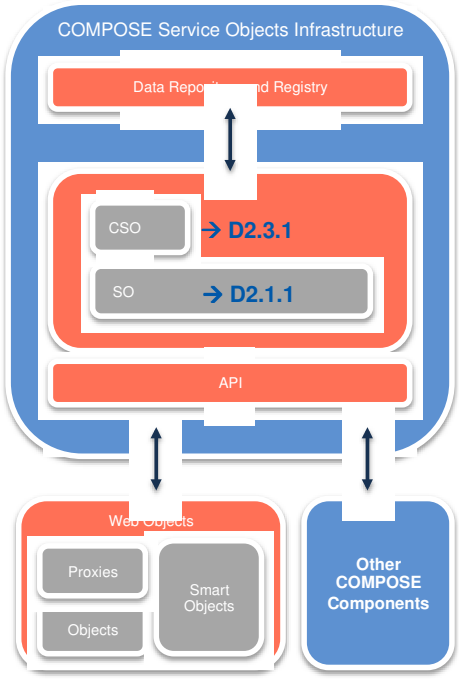

Figure 2: COMPOSE Service Objects infrastructure

1.1 Objects and Smart Objects

Creating a network of real-life interconnected objects is the principal idea behind the Internet of Things. COMPOSE is therefore seen as an enabler for the Internet of Things, which takes a step further in the integration of different objects that are able to provide information of certain kind, or even control the environment where they are located (the sensor/actuator paradigm). These objects will hold a virtual identity in COMPOSE, which will act as their representatives for mainly data collection (acting as sensors), actuation and also as a link to more advanced services. IoT is based on the interconnection of devices, we take a step further and we talk about a Web of Things, where objects are able not only to communicate among themselves, but also they do so utilizing standard web-enabled protocols. For an object (eventually a device) to be web-enabled we consider that at least it can speak web protocols, mainly HTTP (also under some requirements, as we will explain later). This is where we make the following classification:

- ***Object***: an Object in COMPOSE is a physical entity (object or device) that provides some kind of dynamic information, either connected to the Internet or not. Objects cannot either comply with the protocols that are part of COMPOSE or not be directly

connected to Internet. This type of objects will require a proxy to connect to COMPOSE.

- ***Smart Object***: if an Object is a device with web-enabled communication, we define it as a Smart Object. A Smart Object is a device that already holds the capabilities to talk to the COMPOSE platform directly.

1.2 Web Objects

Once the physical entity is web-enabled (either directly or by using a proxy), they have to implement a COMPOSE specific protocol to be integrated in the platform. Any of the abovementioned physical entities, as soon as they implement the COMPOSE specification we are describing in this deliverable, they become a ***Web Object***. A Web Object is therefore characterized by sitting on the edge of the COMPOSE platform and being able of using a predefined web-based protocol to communicate with the COMPOSE platform.

1.3 Service Objects

Once Web Objects are available, they need to be linked to the COMPOSE platform, this is where Service Objects are required. Service Objects are the standard internal COMPOSE representations of Web Objects.

COMPOSE specifies how these objects are expected to communicate with Web Objects, in order to obtain data from them, or set data within them. Either the Smart Object itself or a defined proxy will implement this interface in order to connect to the corresponding COMPOSE Service Objects.

A Service Object exposes the same interface (API) also internally towards the rest of the components within the COMPOSE platform. That interface is needed in order to streamline and standardise internal access to Service Objects, which can in turn represent a variety of very different Web Objects providing very different capabilities.

Furthermore, in order for the Service Object to be a usable component within the platform, upon creation it is enhanced by semantic metadata and is stored in a registry. The enriched description can be used later by COMPOSE discovery mechanisms to supply external users with reference to the Service Object based on its characteristics.

1.4 Composite Service Objects

A Composite Service Object is just like another Service Object that can subscribe to data streams from other Service Objects and perform some computation on them (they run some code that does processing and are created “manually” by COMPOSE developers). They output also data streams that can be accessed by other Composite Service Objects or clients. Composite Service Objects follow the same interface as Service Objects, adding some capabilities for the stream processing. These objects are defined separately in deliverable *D2.3.1 “Design of the object composition specification and components”*.

2 Anatomy of Web Objects and Service Objects

Web Objects are the key element that provides data flows into the COMPOSE platform. They are linked to physical entities and communicate with the platform following a standard web-based protocol. Web Objects have a virtual counterpart in COMPOSE, the Service Object. Both Web Objects and Service Objects communicate in order to stream the information generated by the Web Objects into the platform to be used by the rest of the components.

Web objects, and by extension the Service Objects will follow the same data model (section 4) and will comply with the same protocol (section 5). Service Objects will provide extended functionality as they are the representation of the Web Object inside the platform, but extending the already defined data model (e.g. adding provenance information to the received information or extending the data with semantic links).

As Web Objects come inherently from physical objects, heterogeneity of scenarios arises, depending on the connectivity and capabilities of the corresponding devices. Web Objects might have different types of connectivity and different types of capabilities for web support.

Based on the capabilities for web support two types of Web Objects can be considered:

- ***Server-based Web Objects*** that can run a Web server accessible from the COMPOSE platform, which accepts HTTP requests and can therefore implement the server-based aspects of the Web Streams protocol.

- ***Client-based Web Objects*** that do not support incoming HTTP requests, or are hidden behind a firewall, therefore they must always initiate the connection. In this case, they implement the client-based aspects of the Web Objects protocol and coordinate with the correspondent Server Objects in the COMPOSE Platform. Alternatively, a client-based Web Object could implement Web Sockets as a workaround to provide the server-based functionality of the protocol. The mapping of the protocol to Web Sockets is outlined in the appendixes.

In regards to the type of connectivity, many network configurations are possible:

- ***Object without connectivity:*** The object is a physical object with no network device attached, nevertheless a proxy can be created (depending on the domain) to connect to a Service Object inside COMPOSE.

- ***Object with non-web connectivity:*** The object has a network device with on-board support for a non-web protocol, creating then the need for a proxy to connect to COMPOSE.

- ***Smart object***: Web Objects by definition have on-board support for web protocols, mainly HTTP, therefore they do not require a specific proxy, they just require to implement the COMPOSE protocol.

Web Objects will support different communication paradigms, which are conditioned by the network topology and the type of Web Object (as defined before). If we consider communication from the application/client point of view we can define the following alternatives:

- ***Push interactions***: Service Objects receive updates on the Web Objects information directly from them. This is the default option for client-based Web Objects, as they do not support incoming connections.
- ***Polling interactions***: Server Objects inside COMPOSE are able to ask the Web Objects to provide information on their status.
- ***Subscription mechanism***: advanced Web Objects might include a subscription mechanism based on those provided by the HTTP protocol itself (HTTP callbacks, Web Sockets) or others of choice

The following table shows a relation of the different possible configurations for Web Objects with some examples of their configuration:

Table 1: Connectivity vs. Web-style examples for Web Objects

Connectivity/Web-style		**Client**	**Server**
		A PROXY must be created complying with our Web Object definition. The PROXY in this case has only client capabilities, therefore it can only update its state by connecting to the Service Object directly.	*A PROXY must be created complying with our Web Object definition. The PROXY in this case has server capabilities so it can support the full functionality.*
Object without connectivity	*A physical entity without any connectivity.*	A bottle of wine with a QR code. A scanner in the check-out is implemented as a COMPOSE PROXY and just updates on the stock.	A book with a NFC tag. A reader in a library is implemented as a COMPOSE PROXY. Service Objects can be subscribed to be notified when certain book is taken.
Object with connectivity	*A physical entity with network connectivity but not web-based.*	A set of temperature sensor nodes that communicate in Zigbee. A COMPOSE PROXY collects the sensed information and updates every hour on the values.	A set of light sensors and actuators on the city lights that communicate with BLE. A COMPOSE PROXY can access the information and operate on the lights.
Smart Object	*An object with web-enabled communication.*	A smartphone which implements the COMPOSE Web Streams protocol and advertises directly to its Service Object the updates on its location.	A Raspberry PI device which is connected to a fan and provides control over its speed.

Regardless of the type and configuration, we need a common framework for this devices to communicate with COMPOSE and share its data. And this is what the Web Streams protocol we define in this deliverable does; it provides a common data model and common operations (based on a HTTP REST-based protocol) to enable the communication of Web Objects with COMPOSE, as well as applications and services with their virtual representations within COMPOSE, the Service Objects.

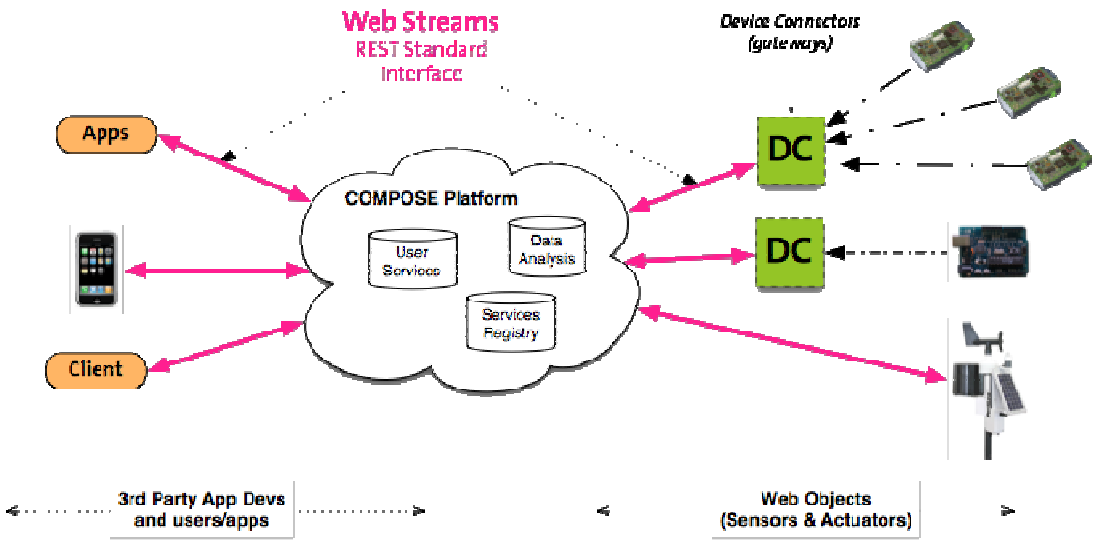

Figure 3: Web Streams general view

In COMPOSE we are based on the idea of the Web of Things[1], being in our case the basic requirements for a device to comply with the Web Streams requirements. Smart Objects supporting RESTful interactions are out-of-the-box part of the Web of Things. Web Objects Requirements (section 3) is simply a list of constraints and recommendations on how to leverage the basic Web architecture (mainly HTTP protocol) for devices to be part of the Web of Things and simplify the integration with the COMPOSE Service Objects. It introduces very little coupling beyond classic HTTP (because it is only basic HTTP), and it is only a set of design choices that can be implemented in any standard HTTP server. This only describes how devices/services can talk to each other, not "how to understand what they are saying".

Once a device is a Web Object and is able to talk HTTP, it can then also implement the Web Streams Protocol. Web Streams Protocol (section 7) is a proposal for a protocol on top of HTTP, which describes how to interact with the objects to extract information from them (sensors) or even to delegate actions (actuators) following well established REST principles.

[1] http://www.webofthings.org, http://www.w3.org/community/wot/

3 Web Objects requirements

In this section we define requirements for Web Objects (WO). As we explained before, some Web Objects will be restricted to a client-based version, for which some of the requirements will not be applicable. The peculiarities of client-based WO affect the protocol in ways that will be explained in section 5. To clarify this aspect, in each requirement we will highlight whether it applies to WO client-based, WO server-based or both.

3.1 Level 0 - MUST

R0.1 - WO MUST support HTTP version 1.1

Web Objects MUST be able to communicate using HTTP 1.1. This can be either as a mere client that pushes data to an endpoint (in COMPOSE, this would be a Service Object) and/or by enabling a HTTP server.

Applies to: WO client-based, WO server-based

R0.2 - WO MUST support UTF8 encoding, for requests and responses

The supported format for encoding all requests and responses sent and received from the WO MUST be UTF8, to avoid problems with special characters and localized content.

Applies to: WO client-based, WO server-based

R0.3 - WO MUST support GET, POST and DELETE HTTP verbs

The aim of the protocol to be defined for WO is to follow REST principles. For this idea to come to reality, support certain HTTP verbs of the specification is a must. Considering the REST paradigm is based on resources and CRUD operations on them, the WO must support GET for reading operations, POST for creation and DELETE for removal.

The update of information of resources is usually done with the PUT/PATCH verbs, but as we are considering the possibility of restricted devices (e.g. the client-based WO), PUT/PATCH is in reality not widely supported in certain libraries, therefore POST shall be used instead for updates.

Applies to: WO client-based, WO server-based

R0.4 - WO MUST support JSON as the default representation mechanism

Information exchanged or requested from/to WO MUST be represented in JSON or any compatible variant. JSON might not be the only format supported but it will be the default one, so the information provided by the WO is always available in this format.

Applies to: WO client-based, WO server-based

R0.5 - WO MUST have a unique identifier represented by a dereferenceable slash HTTP URI

It is important for a WO to have unique Web identifier and a common access point, a so called "root resource" which represents the entry point for the Web Object and enables the interaction with it.

Examples of acceptable HTTP URIs are:

http://gateway.api.com/devices/TV

http://kitchen-raspberry.device-lab.co.uk

https://192.168.10.10:9002

https://kitchen:3000/fridge/root

Note: the root URL does not require the device to be connected and accessible "publicly" over the Web. The URL works equally well inside a local area network (LAN).

Applies to: WO server-based

R0.6 - WO MUST implement HTTP status codes 200, 400, 500

Full HTTP 1.1 support is advisable, however, considering the range in nature of the WO (some of them could be restricted devices) only a minimum subset is mandatory for a successful action, error on the client side and error on the server side.

Applies to: WO server-based

R0.7 – WO MUST provide a description document at their entry point address

A WO MUST have an address to which a simple HTTP GET can be done and a whole description about the WO can be retrieved with no further effort.

Applies to: WO server-based

R0.8 - WO MUST use secure HTTP connections (HTTPS)

WO must use always a secure connection HTTPS. This is enforced via security policies that are defined inside COMPOSE (see D5.1.1 "Security requirements and architecture for COMPOSE").

Applies to: WO client-based, WO server-based

3.2 Level 1 - SHOULD

Any Web Object SHOULD support these rules (in the limits of reason and ability of devices). So unless there are technical or practical limitations for not adhering to these constraints, they should be followed.

R1.1 – A WO description document should be provided by just using the IP address of the device and a default web port.

In other words, accessing the IP address of the Web Object and the default port 80 for HTTP, or 443 if HTTPS should return the full description of the WO. This feature is envisioned for an easy mechanism for discovery of devices at the network level without further protocol needs.

Applies to: WO server-based

R1.2 - WO SHOULD support PUT/PATCH for resource updates

In particular, if possible the use of PUT and PATCH in addition to POST should be used for changing the state of a device (or sending a request to an actuator).

Applies to: WO server-based

R1.3 - WO SHOULD support additional HTTP status codes

Especially for errors, WO internal HTTP implementation should support the most common HTTP status codes (415 Invalid media type, 403 Forbidden, 201 Created, 503 Unavailable). When applicable, any other code from the HTTP specification SHOULD be supported as appropriate.

Applies to: WO server-based

R1.4 - WO SHOULD be able to provide subscription mechanisms to their information

Ideally, WO provide subscription mechanism so that interested parties can get notifications of the updates on their state (e.g. the Service Objects in the COMPOSE platform), being able to subscribe to streams and not have to continuously poll WOs for new data.

Subscriptions should be also configurable ideally to optimize the communication between WO and their subscribers. As the WO is HTTP based, alternatives for this subscription mechanism are, but not restricted to, HTTP Callbacks or Web Sockets.

Applies to: WO client-based, WO server-based

3.3 Level 2 - MAY

Any Web Object MAY support these rules, but not expected to. It depends on the capabilities of the devices and implementation aspects.

R2.1 - WOs MAY have a default machine-readable documentation

Web Objects may decide to provide their information in any machine-readable format available. As the only requirement is JSON it is left to the Web Object developers to provide any alternative machine-readable format, like JSON-LD or JSON-schema.

R2.2 - WOs MAY offer an HTML interface/representation (UI)

Some WO may come with a pre-programmed HTML representation or UI that could be used under some conditions to integrate WO in dashboard environments.

Applies to: WO server-based

R2.4 - WOs MAY provide additional representation mechanisms (RDF, XML, JSON-LD)

WO can provide additional representations of the data they provide and even their own descriptions. Being JSON the only mandatory representation, the rest of the formats are up to the implementers if needed for additional purposes. The protocol does not define the mappings to other representations so far, we may consider adding them in future revisions if needed.

Applies to: WO client-based, WO server-based

R2.5 - WOs MAY provide streaming mechanisms

WO sensed information might require the use of special streaming protocols. For this situations WO can provide special subscription channels that will be tailored to the concrete streaming protocol.

Applies to: WO server-based

4 Web Streams Model

The idea behind distinguishing between Objects, Web Objects and Service Objects is to establish the different levels and representations involved in the process of taking a physical object or device and making it part of COMPOSE. Regardless of this distinction, we aim at providing a unified data model that can be used throughout all the components. For this purpose we see the data model as a stack of information, meaning that a Web Object defines some properties that still hold for the correspondent Service Object. Additionally, the Service Object will have extended properties that complement the description. Ultimately, Composite Service Objects add more extensions to this model.

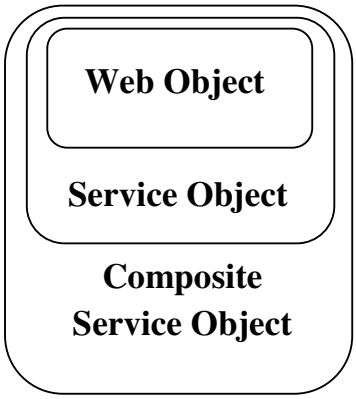

Figure 4: Nested view of the Web Streams data model.

4.1 Web Objects

Note that the Web Object requirements do not state anything about the actual interaction between WO and the SO, i.e. the API description, conventions, or data models used by the WO. This section proposes an actual data model to be used on top of Web Objects as defined in the previous chapter.

The WO Model is defined by 5 sections Properties, Streams, Actions, and Subscriptions. The general JSON structure is as follows:

Table 2: Web Object Model summary.

```
{
	"properties":{},
	"streams":[ ],
	"actions":[ ],
	"subscriptions":[ ]
}
```

4.1.1 Properties

Properties are static information about the WO. Each WO can have a set of properties which describe general properties and the static state of the device. Information represented as properties are also considered not to have a history of how the value changed over time. The underlined attributes are mandatory:

- ***id***: unique identifier of the Web Object.
- ***name***: human readable name of the Web Object.
- ***description***: human readable description of the Web Object.
- ***schema***: link to the schema definition of the particular Web Object. Some aspects of the WO are not standardized as they depend on the concrete type, therefore the schema of the missing parts must be provided here.
- ***help***: pointer to additional information of the Web Object, how it works and how to use it if needed. It could be either human or machine readable.
- ***ui***: pointer to a web based interface of the Web Object if available.
- ***mode***: describes if the Web Object is client-based or server-based. By default it is considered to be client only.
    - client: the WO can only work as a HTTP client.
    - server: the WO can work as HTTP server and implements the server related functionality of the WO protocol.
- ***subscriptionOptions***: an array defining which subscription methods are available, if any. WO could support either HTTP callbacks or websockets.
- ***customFields***: extension point to define WO specific properties.

Table 3: Example - Properties of a WO.

```
{
  "properties":{
       "id":"WebObject-2423529879879875",
       "name":"Shopping Cart",
       "description":"Shopping Cart that updates location information as it
                      moves",
        "schema":"http://myshoppingcart.com/doc/shoppinCartSchema.json"
       "help":"http://www.myshoppingcart.com/shopppingCart.html",
       "ui":"http://www.myshoppingcart.com/shoppingCartUI.html",
       "mode":"server",
       "subscriptionOptions":[
              "http.callback",
              "http.websocket"
       ],
       "customFields":{
              "creator":"COMPOSE Consortium Hardware Team"
       }
  …
}
```

4.1.2 Streams

Any WO can have 0-n sensors that can collect data periodically or sporadically and possibly store that data locally. Each sensor has an associated data stream (a sensor stream) where sensor readings are pushed and made available (each reading is called a sensor update). There might be additional streams for other purposes, such as debugging, etc., therefore we refer to streams as any piece of information, the value of which changes over time and is exposed by the WO. Each stream has the following schema:

- ***name***: a given name to the stream.
- ***lastUpdate***: this value is simply the timestamp (Unix epochs) of last time the readings have been updated. Updates are synchronized by stream; all the channels will share the update time.
- ***channels***: each stream might have one or more dimensions, which are dependent on the specific domain. We call these dimensions channels. The definition of the channels is domain specific, but typically, each channel will have a set of custom fields which are multiple key-value pairs for that particular reading. No model is suggested/enforced/validated (other than it must be a set of key value pairs) - the semantics should be established by the WO creator and exposed in the documentation for the integration in COMPOSE.
- ***description***: a human readable description of the stream.
- ***type***: an open field to classify the stream if needed. Default value is sensor..
- ***customFields***: each stream has a customFields attribute, which is simply a set of key-value pairs to store current information about the sensor (not historical, updates are not stored). A sensor can have multiple channels and each channel has only one dimension (k-v pair, with its own customFields).

Table 4: Example: Stream info of a WO

```
{
…

"streams":[
	{
		"name":"location",
		"description":"Location of the shopping cart",
		"type":"sensor",
		"lastUpdate":998239224,
		"channels":[
			{
				"name":"longitude",
				"unit":"degrees",
				"type":"numeric",
				"current-value":30
			},
			{
				"name":"latitude",
				"unit":"degrees",
				"type":"numeric",
				"current-value":70
			}
		],
```

```
            "customFields":{
                  "model":"ModelX245-A Location Sensor",
                  "measureError":"1%",
                  "technology":"BLE"
            }
      }
],
…
}
```

4.1.3 Actions

Each WO could have 0-m actions, representing predefined commands that can be sent for the WO to perform some kind of activity. These actions will have normally a set of parameters that can be configured in a key-value fashion. The description of these parameters depends on the WO and their schema should be provided in the schema definition. Actions are usually defined by the manufacturer to map the actuation capabilities of a device. The schema of the actions has the following properties:

- ***name***: identifier of the action to be used in the protocol when sending the request to the WO.
- ***description***: human readable description of the action
- ***status***: defines if the action is being executed and its last status (“success”, “failed”, ”in progress”)
- ***lastUpdate***: the timestamp when the action was invoked the last time.
- ***parameters***: set of parameters that are needed for the action. The definition of the parameters is dependent of the action.

Table 5: Example: Action of a Web Object

```
{
…

      "actions":[
            {
                  "name":"reboot",
                  "description":"Reboots the device",
                  "status":"success",
                  "lastUpdate":23123123,
                  "parameters": {
                        "delay":"integer",
                        "requester":"string"
                  }
            }
      ]

…

}
```

4.1.4 Subscriptions

Information provided by a WO could potentially be accessed in many ways, as it will be defined by the WO protocol. Some WO might support a subscription mechanism, so we need to include in the WO model a way to represent subscriptions, regardless of the underlying mechanism. Subscriptions are made against streams in a WO. Subscriptions can only be defined for Web Objects that support server capabilities. A subscription is defined by the following attributes:

- ***subscriberId***: an identifier of the subscriber.
- ***subscriptionId***: an identifier for the specific subscription to the stream.
- ***type***: defines the type of the subscription, mainly the mechanism for the notification (http callbacks, websockets…).
- ***delay***: the minimal time in seconds between two notifications. Notifications are sent only when a new sensor reading is available, but at most once every delay seconds. If set to 0, then notifications are sent as soon as a new sensor reading has been done.
- ***expire***: it is the duration (in seconds) the subscription will be valid. The value can be between 0 (never) and 31536000 (365 days, max allowed).
- ***forceUpdates***: if set to false, notifications will be sent only if the sensor reading has changed since last notification. If the value is true: notifications will be sent each time (either periodically or sporadically).
- ***stream***: the stream to which the subscription is connected.
- ***customFields***: here the specificities of the type of subscription can be added as a key value set of properties, for example, the url where to push the notifications can be described here. Or additional authentication tokens if needed.

Table 6: Example: Subscriptions in a WO

```
{
…
      "subscriptions":[
            {
                  "subscriberId":"ServiceObject-123213213",
                  "subscriptionId":"ServiceObject-123213213#location",
                  "type":"http.callback",
                  "delay":0,
                  "expire":10000,
                  "forceUpdates":false,
                  "stream":"location"
                  "customFields":{
                        "callbackUrl":"http://www.compose-
project.eu/so/ServiceObject-123213213/callback"
                  }
            },
            {
                  "subscriberId":"ServiceObject-123213213",
                  "subscriptionId":"ServiceObject-123213213#location02",
                  "type":"http.websocket",
                  "delay":0,
                  "expire":10000,
                  "forceUpdates":false,
                  "stream":"location"
```

```
                    "customFields":{
                          "websocketUrl":"wss://www.compose-
project.eu/WebObject-2423529879879875/streams/location"
                    }
               }
       ]…
}
```

4.2 Service Objects

As Service Objects are the virtual representations of the Web Objects inside the COMPOSE platform, the Service Object Model is naturally an extension of the Web Object schema. A Service Object will be described in the same way as a Web Object, but including additional attributes that are inherent to the capabilities of the platform (like semantic links, extended subscription mechanisms or provenance information).

4.2.1 Extended Subscription Mechanism

Service Objects will support additional subscription mechanisms provided by the COMPOSE platform. This is reflected in the schema by allowing additional types of subscription in the properties of the Service Object. For example, there is already an on-going effort in WP4 (summarized in *D4.2.2 Initial prototype of the COMPOSE communication infrastructure*) where novel subscription mechanisms are defined, which can be easily integrated in the Service Object information model.

4.2.2 Provenance

Provenance information is used within the COMPOSE platform to keep track of the origins of the data and ensure correct security and access control mechanisms. A Service Object represents an entity providing and collecting data, therefore tracking the movement and processing of this data is important in COMPOSE. Provenance mechanisms are defined as part of WP5, but Service Objects should reflect on their information model the information that is needed for provenance.

Provenance information is provided by stream, therefore the stream schema is extended with a field “provenance” including information about the data streams. Provenance not only defines the origin of the data, also provides information about operations carried out on it and by whom. In COMPOSE, we are interested in the following set of actions:

- Creation of data (WasCreatedBy)
- Copy of data (WasCopiedFrom)
- Modification of data (WasDerivedFrom, PrimarySourceOf)
- Reading data (WasUsedBy)
- Deletion of data (WasInvalidatedBy)

Actions listed previously were taken from the W3C Provenance Model PROV-DM. Data Provenance will be used mainly to support security policy enforcement before data is shared within COMPOSE platform. How to populate this provenance data or even where to store it is out of the scope of the Web Streams protocol, as it is a platform specific aspect. The Web Streams protocol just provides the data model for provenance.

Table 7: Example: Provenance info layout

```
"streams":[
{
	…
	"provenance":{
		"wasCreatedBy":"WebObject-2423529879879875",
		"wasCopiedFrom":"",
		"wasDerivedFrom":[
				],
		"wasUsedBy":[
				"ServiceObjectID1",
				"ServiceObjectID2"
			],
		"wasInvalidatedBy":""
	}
	…
}
```

4.2.3 Linking in Service Objects

The idea of linking Service Objects is to enable the possibility to create provide "semantic" descriptions of them. As the model is JSON based, a linking mechanism in JSON has to be applied. Several alternatives are available to provide links in JSON, such as HAL[2], Collection+JSON[3] or JSON-LD[4]. We do not restrict the use of a specific mechanism; we simply acknowledge the necessity of providing linking capabilities to the Service Object descriptions. Nevertheless, as Service Objects will be integrated further in the project with semantic descriptions, the use of JSON-LD is encouraged.

The semantic information associated to the links section is not static and is developed as part of WP3.1. A request to get the Service Object description will include if available the links section with the semantic link or description. The following is an example of how this section might look like in JSON-LD:

Table 8: Example: Links in SO

```
{
…
	"links":{
	    "@context": "http://www.compose-prject.eu/contexts/SO.jsonld",
	    "@id": "http://superstore.org/resource/shopping_cart",
	    "color": "Red",
	    "owner": "http://superstore.org/resource/megastore" }…
}
```

[2] http://stateless.co/hal_specification.html
[3] http://amundsen.com/media-types/collection/
[4] http://json-ld.org/

5 Web Streams Protocol

The Web Streams Protocol defines a HTTP REST based protocol that governs the access to information provided by Web Objects and Service Objects. Many operations apply as well to the interaction between Service Objects and Services in COMPOSE platform as the provided functionality is the same. The difference between them lies in the parts of the communication and the operations that can be done, sometimes in a slightly different way, but still following the same protocol.

As we are based on REST principles, the Web Objects and Service Objects have a view of resources. Each group of properties is treated as resources too, triggering the functionality depending on the operations performed on these resources, mainly CRUD operations. The following table summarizes the operations that can be done for each type of object:

Table 9: Operations of the Web Stream Protocol

Resource	Operation	Web Object client	Web Object server	Service Object
Object information	CREATE	-	-	POST
	READ	-	GET	GET
	UPDATE	-	-	PUT
	DELETE	-	-	DELETE
Properties	CREATE	-	-	POST
	READ	-	GET	GET
	UPDATE	-	-	PUT
	DELETE	-	-	DELETE
Streams	CREATE	-	-	-
	READ	-	GET	GET
	UPDATE	-	-	-
	DELETE	-	-	-
Channels	CREATE	-	-	-
	READ	-	GET	GET
	UPDATE	-	-	-
	DELETE	-	-	-
Actions	CREATE	-	POST	POST
	READ	-	GET	GET
	UPDATE	-	-	-
	DELETE	-	DELETE	DELETE
Subscriptions	CREATE	-	POST	POST
	READ	-	GET	GET
	UPDATE	-	-	-
	DELETE	-	DELETE	DELETE
Links	CREATE	-	-	-
	READ	-	-	GET
	UPDATE	-	-	-
	DELETE	-	-	-

5.1 Web Objects client-based

Client-based Web Objects only support HTTP client operations, therefore they will not provide direct access to the information of the sensors; they will connect with their corresponding Service Object representation and actively update it with the new values for the channels in the streams the Web Object is managing. A Web Object per se then, does not provide any endpoint; it is seen as a client of the Service Object part of the Web Stream protocol for the following operations:

- CREATE Service Object: the Web Object will connect to COMPOSE by trying to create a virtual representation, i.e. the Service Object.
- UPDATE stream/channel: when new data is available in the Web Object it will invoke the UPDATE operation on the stream.

5.2 Web Objects server-based

Server-based Web Objects support an embedded HTTP server, therefore provide additional REST capabilities. The operations allowed for working with server-based Web Objects are as follows:

5.2.1 Working with Web Objects

Getting a Web Object description

As a Web Object is a resource from the REST point of view, making a GET request to the URL of the Web Object returns its full description.

REQUEST

```
GET /WebObject-2423529879879875 HTTP/1.1
Host: www.mywebobject.eu
Accept: application/json
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 1675
{
       "properties":{
              "id":"WebObject-2423529879879875",
              "name":"Shopping Cart",
              "description":"Shopping Cart that updates location information
as it moves",
              "help":"http://www.myshoppingcart.com/shopppingCart.html",
              "ui":"http://www.myshoppingcart.com/shoppingCartUI.html",
              "mode":"server",
              "subscriptionOptions":[
```

```
                    "http.callback",
                    "http.websocket"
                ],
                "customFields":{
                    "creator":"COMPOSE Consortium Hardware Team"
                }
        },
        "streams":[
                {
                    "name":"location",
                    "description":"Location of the shopping cart",
                    "type":"sensor",

        "documentation":"http://myshoppingcart.com/doc/locationSchema.json",
                    "lastUpdate":998239224,
                    "channels":[
                            {
                                "name":"longitude",
                                "unit":"degrees",
                                "type":"numeric",
                                "current-value":30
                            },
                            {
                                "name":"latitude",
                                "unit":"degrees",
                                "type":"numeric",
                                "current-value":70
                            }
                    ],
                    "customFields":{
                            "model":"ModelX245-A Location Sensor",
                            "measureError":"1%",
                            "technology":"BLE"
                    }
                }
        ],
        "actions":[
                {
                    "name":"reboot",
                    "description":"Reboots the device",
                    "status":"idle",
                    "lastUpdate":23123123,
                    "parameters": {
                            "delay":"integer",
                            "requester":"string"
                    }
                }
        ],
        "subscriptions":[
                {
                    "subscriberId":"ServiceObject-123213213",
                    "subscriptionId":"ServiceObject-123213213#location",
                    "type":"http.callback",
                    "delay":0,
                    "expire":10000,
                    "forceUpdates":false,
```

```
                    "stream":"location"
                    "customFields":{
                          "callbackUrl":"http://www.compose-
project.eu/so/ServiceObject-123213213/callback"
                    }
             },
             {
                    "subscriberId":"ServiceObject-123213213",
                    "subscriptionId":"ServiceObject-123213213#location02",
                    "type":"http.websocket",
                    "delay":0,
                    "expire":10000,
                    "forceUpdates":false,
                    "stream":"location"
                    "customFields":{
                          "websocketUrl":"wss://www.compose-
project.eu/WebObject-2423529879879875/streams/location"
                    }
             }
      ]
}
```

5.2.2 Working with Properties

Getting the properties from a Web Object

The properties of a Web Object can be accessed by making GET requests to the following URL pattern:

webobjectURL/properties/{propertyName}

REQUEST

```
GET /WebObject-2423529879879875/properties HTTP/1.1
Host: www.compose-project.eu
Accept: application/json
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 639
{
      "id":"WebObject-2423529879879875",
      "name":"Shopping Cart",
      "description":"Shopping Cart that updates location information as it
moves",
      "help":"http://www.myshoppingcart.com/shopppingCart.html",
      "ui":"http://www.myshoppingcart.com/shoppingCartUI.html",
      "mode":"server",
      "subscriptionOptions":[
            "http.callback",
            "http.websocket"
```

```
        ],
        "customFields":{
                "creator":"COMPOSE Consortium Hardware Team"
        }
}
```

Individual properties can also be accessed by using the ***name*** in the URL

REQUEST

```
GET /WebObject-2423529879879875/properties/name HTTP/1.1
Host: www.compose-project.eu
Accept: application/json
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 639

{
        "name":"Shopping Cart"
}
```

5.2.3 Working with Streams

A stream of a WO is seen as a resource from the REST perspective so it can be accessed by appending the corresponding name to the URL. The properties of the streams and channels are equally accessible following this pattern.

webobjectURL/streams/{streamName}/{propertyName}

Get the list of streams of a WO

REQUEST

```
GET /WebObject-2423529879879875/streams HTTP/1.1
Host: www.compose-project.eu
Accept: application/json
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 1675

{
        {
```

```
            "name":"location",
            "description":"Location of the shopping cart",
            "type":"sensor",

      "documentation":"http://myshoppingcart.com/doc/locationSchema.json",
            "lastUpdate":998239224,
            "channels":[
                  {
                        "name":"longitude",
                        "unit":"degrees",
                        "type":"numeric",
                        "current-value":30
                  },
                  {
                        "name":"latitude",
                        "unit":"degrees",
                        "type":"numeric",
                        "current-value":70
                  }
            ],
            "customFields":{
                  "model":"ModelX245-A Location Sensor",
                  "measureError":"1%",
                  "technology":"BLE"
            }
      }
}
```

Get channels available in a stream

REQUEST

```
GET /WebObject-2423529879879875/streams/location/channels HTTP/1.1
Host: www.compose-project.eu
Accept: application/json
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 1675

{
      {
            "name":"longitude",
            "unit":"degrees",
            "type":"numeric",
            "current-value":30
      },
      {
            "name":"latitude",
            "unit":"degrees",
            "type":"numeric",
```

```
                "current-value":70
        }
}
```

Some considerations about the channels:

- Each stream is independent; there is no implicit synchronization between streams by default.
- All channels are synchronized, they have the same timestamp (not channel-specific timestamp). The timestamp is stream specific.
- Channel type is dependent on the implementation of the Web Object. They can be defined as a simple string or numeric value, as well as a more complex byte array or binary data.

5.2.4 Working with Actions

Similar to streams in their structure, actions are mostly used to proxy actuators of the device and commands that can be sent to the device. As actions are also resources, the same URL structure applies.

webobjectURL/actions/{actionName}/{propertyName}

Get list of actions of an Web Object

REQUEST

```
GET /WebObject-2423529879879875/actions HTTP/1.1
Host: www.compose-project.eu
Accept: application/json
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 1675

{
        {
                "name":"reboot",
                "description":"Reboots the device",
                "status":"idle",
                "lastUpdate":23123123,
                "parameters": {
                        "delay":"integer",
                        "requester":"string"
                }
        },
        {
        "name":"set-led",
                "description":"Sets the led colour",
```

```
            "status":"in-progress",
            "lastUpdate":23123123,
            "parameters": {
                  "colour":"integer",
                  "requester":"string"
            }
      }
}
```

Send action command to a Web Object

REQUEST

```
POST /WebObject-2423529879879875/actions/reboot HTTP/1.1
Host: www.compose-project.eu
Accept: application/json
Content-type: application/json

{
      "requester":"ServiceObject-1412413241"
      "delay":1000
}
```

RESPONSE

```
HTTP/1.1 201 Created
Content-Type: application/json
Content-Length: 165

{
      "action-id":"ServiceObject-1412413241#reboot"
}
```

Cancel an action

Depending on the type of action, the Web Object might allow the cancelation of it if still in progress. This is done by sending a delete request on the action-id received when the action is posted to the WO.

```
DELETE /WebObject-2423529879879875/actions/reboot/ServiceObject-
1412413241#reboot HTTP/1.1
Host: www.compose-project.eu
Accept: application/json

HTTP/1.1 200 OK
```

5.2.5 Working with Subscriptions

Web Objects might provide subscription mechanisms to the streams of data they update. Subscribing to a stream updates are done by posting a subscription request to the subscriptions resource of a Web Object.

webObjectURL/subscriptions/{subscriptionId}/{propertyName}

Create subscriptions to streams – HTTP callback

REQUEST

```
POST /WebObject-2423529879879875/subscriptions HTTP/1.1
Host: www.compose-project.eu
Accept: application/json
Content-type: application/json

{
	"subscriberId":"ServiceObject-123213213",
	"type":"http.callback",
	"delay":0,
	"expire":10000,
	"forceUpdates":false,
	"stream":"location"
	"customFields":{
		"callbackUrl":"http://www.compose-project.eu/so/ServiceObject-
123213213/callback"
	}
}
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 1675

{
	"subscriptionId":"ServiceObject-123213213#subscription"
}
```

Create subscriptions to streams – Web sockets

REQUEST

```
POST /WebObject-2423529879879875/streams/location/subscriptions HTTP/1.1
Host: www.compose-project.eu
Accept: application/json
Content-type: application/json

{
	"subscriberId":"ServiceObject-123213213",
	"type":"http.websocket",
	"delay":0,
	"expire":10000,
	"forceUpdates":false,
	"stream":"location"
}
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 1675

{
	"subscriptionId":"ServiceObject-123213213#subscription",
	 "customFields":{
		"websocket":"wss://www.compose-project.eu/WebObject-
2423529879879875/streams/location/"
	}
}
```

When opening a WebSockets channel for notifications, the sequence of activities is as follows:

- The subscriber POSTs s subscription request stating that the type of subscription corresponds to a Web Socket.
- If the Web Object accepts the subscription, it provides the WebSocket address of the stream and sends as stated in the subscription the corresponding updates as JSON objects.

Get/Delete/Update information about subscriptions to sensor data

As subscriptions are created they become resources, accessible using the same strategy as any other resource. Sending a GET request on a subscriptionId the details of the subscription can be retrieved. Accordingly, sending DELETE and PUT requests will cancel and update the subscription respectively.

5.3 Service Objects

The Service Objects extension to the Web Streams Protocol provides additional interactions to govern the connection of a WO with a SO, as well as adding additional operations to cover the functionality present in the SO that are not available in the WO.

5.3.1 Working with Service Objects

Creating Service Objects

Web Objects need to be connected to the COMPOSE platform so that a Service Object representation is instantiated and can start being operational. This process could be seen as the registration of the WO within the COMPOSE platform. A Web Object is responsible of establishing the registration of the Service Object on its behalf. As the protocol is REST based and we consider Service Objects as resources, this will be mapped to CRUD operations. How the Web Object knows about the endpoint of the COMPOSE platform is a pre-configuration step that is out of the scope of this protocol.

A Web Object sends its full description to the Service Object creation endpoint. If the correspondent Service Object was not previously created, it will create and start the instance, if it was already created it will just return the assigned ID.

REQUEST

```
POST /ServiceObjects HTTP/1.1
Host: www.compose-project.eu
Accept: application/json
Content-type: application/json

{
       "properties":{
              "id":"WebObject-234980923490823409823 4",
               …
       },
       "streams":[ ],
       "actions":[ ],
       "subscriptions":[ ]
}
```

RESPONSE

```
HTTP/1.1 201 Created
Content-Type: application/json
Content-Length: 165

{
       "id":"ServiceObject-234980923490823409823 4"
}
```

Deleting Service Objects

REQUEST
DELETE /ServicebObject-2423529879879875 HTTP/1.1 Host: www.compose-project.eu Accept: application/json
RESPONSE
HTTP/1.1 200 OK

6 COMPOSE platform issues

Even though the purpose of this protocol is to provide the communication basis and functionality provided by Web Objects and Service Objects, some aspects not covered so far here will be relevant for the actual implementation of the platform. Next we summarize some of these aspects and how we envision they should be dealt with in the implementation of the platform.

6.1 Lifecycle

Life cycle of the WO is not integrated in the lifecycle of COMPOSE, as they are external to the platform and intented to be devices provided by third parties that are provided with the endpoint of a COMPOSE platform where they can register their correspondent Service Object.

Life cycle of the SO is integrated into the global lifecycle management in COMPOSE, as defined in D32.1. The lifecycle of a SO comes defined by two different SO API operations: Deployment and Undeployment of a SO in the platform.

Deployment is activated by a Web Object creating the corresponding Service Object virtual representation, by sending its descriptor. The related storage mechanisms will be activated based on the platform's implementation.

The undeployment of the SO will be done in a similar way, invoking the corresponding API operation. Undeploying a Service Object is a sensitive operation, therefore it must be done on administrative request. In general, a Service Object could potentially be shared among different Composite services or even applications. The undeployment of a Service Object might have a cascade effects on multiple data processing paths within the COMPOSE platform. How to control this process is left to the concrete implementation of the platform.

6.2 Data policies

Not a protocol concern by itself, but as Web Streams defines how to send and update dynamic data with a history, issues surrounding the storage and access of this data will need to be addressed. Either for business model motivations or by ethical concerns, there are some requirements to satisfy about the historical data associated to the Service Objects. At a platform level a set of control knobs to the high level entities in COMPOSE will be provided. By default they are all unlimited unless specified otherwise. The initial set that will be provided is based on the study of the use cases are:

- Max Data Volume
- Max Number of Data Samples
- Deletion policy
- Historical
- Historical Time Window
- Time to Live

- Max Historical Process Time
- Max Incoming Data Process Time

A more detailed description of these parameters can be found in Deliverable *D2.3.1 Design of the object composition specification and components.*

6.3 Data storage

Even though the Web Streams protocol provides a unified data model, this does not imply that all the information is stored in the same way inside the COMPOSE platform. All the information provided by the COMPOSE platform for each Service Object (or Composite Service Object) will be stored around different components. The implementation choice is still to be defined in further work on the platform, but for example, the information coming for the Web Objects, the provenance information and the semantic descriptions of the Service/Composite objects will be likely to be stored separately and combined when using the protocol to request for this information.

6.4 Service Objects Registry

The data and meta-data associated to Service Objects will be stored in the Service Objects repository and registry, correspondingly. In practice both entities will be built on top of the data back-end as defined in the COMPOSE architecture. The technology used to store the data is a document oriented NoSQL database, as it is a best fit for the type of document oriented communication protocol described in this document. JSON documents are the basic data structure in the COMPOSE data plane, comprising Service Objects and Composite Service Objects. Access to the registry and repository will be transparent, as it is done through the Service Objects API, implemented using a RESTful protocol that is specified in this document. No direct access to the data back-end will be offered to external entities. And even COMPOSE components not being directly part of the infrastructure servicing Service Object logic will have to access the data through the API. This way, COMPOSE will guarantee that provenance data is kept updated at all times, which is a critical issue in the platform.

6.5 Ethical Issues

Ethical issues around the Web Streams protocol could arise by the usage of the data that is exchanged by it. Eventually the Web Objects represent physical devices, probably sensing information of the environment, with actuation capabilities in certain cases and most importantly, with a user or several users associated to the data provided by the Web Objects. As this information is mapped to the COMPOSE platform Service Object and further extended by the Composite Service Objects, this poses a serious challenge because different aspects need to be considered (e.g. how long data is stored, how data access is secured, what consent has been given on potential uses of data).

Many of these questions are answered in COMPOSE through the interaction between different components in the platform, being the security module the central element. It is therefore important to provide the means to control such activities. Modifiers, such as the ones explained in section 6.2 will provide the control knobs for other components in the platform to enforce the control on data stored in the SO repository.

Additionally, business models and exploitation plans will have to decide on the very same modifiers to satisfy the needs of any targeted market.

7 Summary of Requirements and influence on the proposed design

The design of the Web Streams specification has been performed in accordance to the requirements identified in D1.1.1. The requirements directly affecting this specification (General, Architectural Component specific or introduced by the use cases) are listed and discussed here in relation to the goals of this document. Some other requirements, not directly linked to this specification but relevant enough for the document, are included also as "indirect requirements" as they introduce co-lateral effects on this document.

7.1 Direct Requirements - Usability

CODE	Title	Description in D1.1.1	How is addressed
US – 1	COMPOSE functionalities should be accessible through open APIs	The interaction with the platform will be made possible through open and well-defined APIs, for both data and services. These APIs will be exposed by the Marketplace for developers and providers.	The design of Web Streams protocol follows a RESTful approach, using JSON and HTTP to transport the operations.
US - 3	Easy developer interaction with the platform to create and register service objects	(…) Developers shall have the support for creating and describing service objects, via a COMPOSE registration interface and GUI, which will be accessible in the COMPOSE platform. Creation of such object is mostly an automatic operation provided by COMPOSE by applying a generic service object wrapper template and an automatic registration phase for new object types. (…)	The REST API allows creating a SO by sending the description of the WO properties to the platform.
US – 7	Comprehensive explanations of security issues and their solution	In case COMPOSE discovers a security problem, i.e. in the service or in a service composition, the user must obtain feedback in which it is relatively easy for him to find the source of the security problem. Possible solutions to the problem should be offered.	The REST API will provide appropriate response codes generated by the COMPOSE security module that will provide feedback to the users in case of any issue.

7.2 Direct Requirements - Scalability

CODE	Title	Description in D1.1.1	How is addressed
SC – 1	COMPOSE shall support in the order of hundreds of thousands of registered service objects	COMPOSE membership component, which keeps track of entities alive and connected to the platform, and the accompanying communication mechanism shall be able to handle the amount of service objects anticipated and detect quickly service objects leaving the system, and discover new services in a reasonable amount of time. The scalability of the platform will be evaluated empirically for up to tens of thousands of service objects, while the projected scalability of the platform should allow for managing a number of service objects	The use of a stateless REST API and the plan to leverage stream-processing technologies in the background provides the means to build a scalable distributed system from a design point of view.

		in the range of a million.	
SC - 4	COMPOSE shall be able to run in a completely geo-distributed scenario	The extremely pervasive nature of the Internet of Things motivates the geo-distribution of COMPOSE components to favour locality and proximity as much as possible. To attain that goal we shall contemplate placing computation close to the edge of the system when appropriate.	The use of stateless REST methods both in the API and in the subscription mechanisms provides a good starting basis for the building of massively distributed systems.

7.3 Direct Requirements - Heterogeneity

CODE	Title	Description in D1.1.1	How is addressed
HT – 1	Integrate a Number of different object technologies into the platform	COMPOSE will support the diversity in the underlying communication infrastructure stemming from interacting with different kinds of smart objects. This will be demonstrated as a part of the piloting activity and scenarios. Examples of such technologies include long lived battery operated devices featuring intermittent communications; streaming from devices such as cameras and microphones; communication via HTTP and WebSockets; IPv6/6LoWPAN; ZigBee; Arduino. It is foreseen that the interaction will be mainly driven through standard interfaces and APIs, such as available through OGC (Sensor Observation Service).	The approach followed for both Services Objects and Composite Service Objects is to assume that Web Objects are their counterparts, and by definition it includes any potential type of device. HTTP and WebSockets will be the main transport protocols. The approach has been validated with Stakeholders and Use Cases.
HT - 2	Platform components will run on a variety of heterogeneous devices, including for example, personal ones (laptops, smartphones) as well as backend systems (servers as well as cloud)	While the crux of the platform will run on server-grade machines, we strive to make use of additional available and distributed computing resources, mainly in the form of smart phones or other rather resourceful mobile devices, in order to perform some computation as close as possible to the relevant information source. The core platform infrastructure is expected to run on a cloud-like infrastructure and the main interaction will be based on Web technologies (for addressing cross-platform and cross-device capabilities). Smart objects resources may be used to carry out computation concerning their derived data, and may further participate in general platform activities such as communication and monitoring.	HTTP and WebSockets provide generic enough means to interact with all kinds of devices.

7.4 Direct Requirements – Data Processing

CODE	Title	Description in D1.1.1	How is addressed
DP – 1	COMPOSE should be able to ingest data flowing into the system with a variety of rates	Support a wide mix of data kinds with different characteristics. From real-time low latency data streams to slowly changing periodic updates, while taking care of historic data as well	The use of a Web Streams protocol in which data is encapsulated as JSON documents allows for a large variety of data types to be supported, with the exception of audio and video streams that would require a different technology to ingest the

			data.
DP - 3	Scalable, distributed and hierarchical data store	Data store should be architected in a manner that enables storing and processing a large amount of data with linear scalability performance, depending on the algorithms involved. It will support geo-distribution and hierarchical storage technologies	The design of the components involved in the SO composition process has been selected to be scalable and distributed: from the RESTful front-end to the NoSQL back-end, as well as the stream-processing pipeline.
DP - 4	Data lifecycle management	COMPOSE will be able to determine when new data is needed, when data has become stale, and the level of certainty that can attributed to different pieces of data	The data model selected allows for the inclusion of fields that define different data storage modifiers (e.g. volume and aging).
DP - 5	Efficiently support both data and metadata stores	Long-lived data and metadata may be associated with COMPOSE entities, such as their description and added semantics. Metadata should be searchable, and support CRUD operations.	The document-oriented back-end to be used provides enough flexibility to support the management of data and meta-data efficiently. The RESTful front-end delivers the CRUD operations.
DP – 7	Provenance Information	Data Management must also be able to securely track the origin and usage history of data. This data may contain user related information, therefore, provenance data must only be accessible by authorized system entities.	As data flows through the CSO processing pipeline, provenance data will be updated and emitted as a part of the output.
DP – 10	Support for PUSH and PULL modes for Object interaction	Data shall be obtained from the objects either in PUSH mode (when the object provides such capabilities) or PULL mode, when the object is not able to initiate the communication with the COMPOSE platform or is hidden behind a passive gateway.	The REST API allows for the use of the PUSH mopde. The use of WebSockets and the WebStreams protocol allows for the use of the PULL mode.
DP – 12	Event-Based computing	The platform shall support subscription services for data changes and allow services to be notified when certain data changes occur.	The design of the CSOs is completely event-driven, being triggered their execution by the arrival of a SU.
DP – 13	Simple and Elastic data models	The data models used in the repository and registry should be simple, reduce to the minimum the need for checks and enforcements on the core platform and allow for dynamic reorganization based on the needs of the services aiming to provide maximum performance as much as possible. The data model shall similarly fit in all the levels of the storage hierarchy. Data checks and validation shall be pushed to the edge as much as possible. The data models will also have to accommodate semantic information provided by the services and objects.	The CSO descriptor is a plain JSON document, and checks will be reduced to the minimum. Semantic information is accommodated through the use of JSON-LD on the links element.
DP – 14	Support for Data Composition and Linked Data	The repository and the data model shall allow for simple ways to plug together objects (data integration) and have proper means to figure out the semantics of the objects. Graph structures shall be supported to improve semantic search over complex structures and combinations of objects. Integration with external sources of data (e.g. Open Data and external service APIs) shall be possible to	The two-level registry in COMPOSE (including the SO registry and the RDF-store developed in WP3.1) will provide access to linked data information in relation to SOs and CSOs.

		produce enriched services in the COMPOSE platform.	

7.5 Direct Requirements - Security

CODE	Title	Description in D1.1.1	How is addressed
SAS – 1	Authentication of Participating Entities	COMPOSE must offer mechanisms which allow the identification and authentication of its entities. This includes authentication for objects, service objects, services, composed services, users, the marketplace, etc. Of course, this requires a specification of how the necessary credentials are distributed.	The API for SOs will support the use of API tokens to identify its users. The management of API tokens and their creation will be defined in WP5.
SAS – 5	Access Control	A comprehensive access control system must be deployed in COMPOSE as not every entity has the same permissions to access other entities. Such system will observe a large variety of possible roles in the system.	Access control will be enforced by the use of API tokens on the operations, and by the coordination with the PDP component developed in the scope of WP5.
SDS – 3	Secure Collection and Storage	In COMPOSE metadata or data which is associated with a user must be protected from other users, services or other third parties. Unauthorized forwarding or usage of data must not be possible without the consent of the user.	Access control will be enforced by the use of API tokens on the operations, and by the coordination with the PDP component developed in the scope of WP5.

7.6 Direct Requirements – Monitoring

CODE	Title	Description in D1.1.1	How is addressed
MN – 4	Provenance	Closely related to usage control is the accumulation of data and service histories. Thus, we require mechanisms which monitor the origin of data and services and the operations performed on data or by services.	As data flows from the WO to the SO and to other components in the platform, provenance data will be updated and emitted as a part of the output of the SO description.

7.7 Direct Requirements – High availability

CODE	Title	Description in D1.1.1	How is addressed
HA – 1	The platform shall exhibit high availability and uninterrupted operation in the presence of faults	As a large scale distributed system, especially when comprised of commodity hardware and more even so with the inclusion of mobile devices, COMPOSE needs to be architected to expect failures and be able to work around them.	SOs will be mapped on top of high-available infrastructures, and their API is designed to be stateless to be fault tolerant.

7.8 Direct Requirements – Ownership

CODE	Title	Description in D1.1.1	How is addressed
OW – 2	Service object ownership policies	The owner of a service object should be able to control the access rights to the object and its related data.	The SO will provide the control knobs to manage data through the use of data store modifiers. These control knobs will be leveraged by other components, particularly from WP5. Access control will be enforced by using API tokens and a close coordination with the PDP component designed in WP5.

7.9 Direct Requirements – Service Objects

CODE	Title	Description in D1.1.1	How is addressed
SEO – 1	Service object representation	Service objects will represent the remote smart object internally to the system; it shall enable access to smart object information (for sensors for example) and act upon them (for actuators)	The SO API will provide access to the data provided by the physical Web Objects.
SEO – 3	Service object registration into the platform	Service objects should be able to announce themselves automatically / programmatically, or have a user manually registering them into the platform.	Service Objects are intended to be registered in the platform automatically by the WO.
SEO – 4	Data access control and Usage Control	Owners of service objects should be able to specify access-control rights for the use of their objects and specify usage control on the resulting data.	Access control will be enforced by using API tokens and a close coordination with the PDP component designed in WP5.
SEO – 5	Service object information sharing	Service objects should easily share their information, such that other interested components can locate and make use of the service object. This process should abide by the specific policies of the objects.	Subscription to SOs will provide this functionality.
SEO – 6	Semantic enhancement	Enrich service objects with semantic information of the smart object they represent, that will help turn them into useful components and building blocks that can be easily located and used by additional parties. In addition, this information can be used for the security analysis of service objects.	SOs will be semantically annotated through the RDF-store developed in WP3.1. The semantic information will be returned by the SO whenever it is requested through the API. It will be included as the links element in the SO descriptor, using JSON-LD to represent it.
SEO – 7	Service objects discovery	Enable discovery and advertisement of newly available service objects.	When deployed, SOs will be visible to other components. The integration with the COMPOSE controller will handle the lifecycle of the

			SOs.
SEO – 8	Service objects search	Support the querying of all available service objects residing in a repository	When deployed, SOs will be searchable. The integration with the COMPOSE controller will handle the lifecycle of the SOs.

7.10 Direct Requirements – Standardisation

CODE	Title	Description in D1.1.1	How is addressed
STD – 1	Object virtualisation	Standardise access mechanisms to smart objects via their virtual service object counterpart, such that accessing such objects is streamlined regardless of the specific protocols used to communicate with the objects or the capabilities supported by the object. This is critical to resilience in the face of heterogeneous services and versioning across releases.	The use of a RESTful API and the HTTP and WebSockets protocols, both of them generic and open technologies, ensures that the platform will be accessible in a standardized way.

8 Summary

The current deliverable is part of the work of WP2, where COMPOSE defines the concepts and methodologies to achieve the Object as a Service concept. This deliverable represents the direct connection between physical objects and the COMPOSE platform. Based on the preliminary analysis of the architecture, we have focused on defining how to integrate real physical objects in the COMPOSE platform.

In the layer closer to the physical objects, COMPOSE defines a series of concepts (Object, Smart Objects, Web Objects, Service Objects, Composite Objects) that have been revisited in this deliverable. Based on this concepts we have outline several aspects to cover the connection and communication of objects to the COMPOSE platform:

- Web Objects requirements: as devices can be heterogeneous in varied ways, we have established that COMPOSE will follow the Web of Things paradigm, where devices must be able to speak web protocols. We have therefore defined the minimum requirements for a device to support direct connectivity to the COMPOSE platform.

- Web Streams Model: based on the components and information that we expect to be exchanged from the connected devices to COMPOSE, we have defined a data model for the Web Objects and the correspondent virtual identity inside COMPOSE, the Service Objects. This model is used to define the characteristics of the objects and the information that is exchanged. It defines how to represent static properties, dynamic data in the form of streams, how to represent actuation capabilities and even how to define subscription mechanisms for this information.

- Web Streams Protocol: this protocol defines in a REST based style the operations that govern the communication between Web Objects and Service Objects, as well as Service Objects in the COMPOSE platform(using the previously defined model).

Next we have presented some platform specific issues that are related to the implementation, but that are going to play an important role in the next steps of the platform development, especially related to the lifecycle, data storage and policies, registration and ethical issues.

Finally we have summarized how the model and protocol defined in this deliverable contribute to the fulfilment of the requisites defined for the COMPOSE platform.

Appendixes provide a complete example of a Service Object description, as well as the associated JSON schema, as well as some guidelines for a generic mapping of our REST protocol for Web Socket support.

The work presented in this deliverable is complemented by *D2.3.1 Design of the object composition specification and components,* where the remaining component from the data plane perspective is defined, the Composite Service Object.

Appendix 1: JSON SO Data Model Example (illustrative)

Table 10: SO Full description example.

```
{

"properties":{
	"id":"ServiceObject-2423529879879875",
	"name":"Shopping Cart",
	"description":"Shopping Cart that updates location information as it
		moves",
	"help":"http://www.myshoppingcart.com/shopppingCart.html",
	"ui":"http://www.myshoppingcart.com/shoppingCartUI.html",
	"mode":"server",
	"subscriptionOptions":[
		"http.callback",
		"http.websocket"
	],
	"customFields":{
		"creator":"COMPOSE Consortium Developer Team"
	}
},
"streams":[
	{
		"name":"location",
		"description":"Location of the shopping cart",
		"type":"sensor",
		 "documentation":
		     "http://myshoppingcart.com/doc/shoppingCartSchema.json",
		"lastUpdate":998239224,
		"channels":[
			{
				"name":"longitude",
				"unit":"degrees",
				"type":"numeric",
				"current-value":30
			},
			{
				"name":"latitude",
				"unit":"degrees",
				"type":"numeric",
				"current-value":70
			}
		],
		"provenance":{
			"createdby":"WebObject-2423529879879875",
			"copiedfrom":"",
			"derivedfrom":[
			],
			"usedby":[
				"ServiceObjectID1",
				"ServiceObjectID2"
			],
```

```
                        "invalidatedby":""
                },

                "customFields":{
                        "model":"ModelX245-A Location Sensor",
                        "measureError":"1%",
                        "technology":"BLE"
                }
        }
],
"actions":[
        {
                "name":"reboot",
                "description":"Reboots the device",
                "status":"success",
                "lastUpdate":23123123,
                "parameters": {
                        "delay":"integer",
                        "requester":"string"
                }
        }
],
"subscriptions":[
        {
                "subscriberId":"ServiceObject-123213213",
                "subscriptionId":"ServiceObject-123213213#location",
                "type":"http.callback",
                "delay":0,
                "expire":10000,
                "forceUpdates":false,
                "stream":"location",
                "customFields":{
                        "callbackUrl":"http://www.compose-
project.eu/so/ServiceObject-123213213/callback"
                }
        }
],
"links":{
        "@context":"http://www.compose-prject.eu/contexts/SO.jsonld",
        "@id":"http://superstore.org/resource/shopping_cart",
        "color":"Red",
        "capacity":"3001",
        "owner":"http://superstore.org/resource/megastore"
}

}
```

Appendix 2: JSON SO Schema

Table 11: SO Data Model JSON schema (ids have been removed for clarity of the code)

```
{
"type":"object",
"$schema":"http://json-schema.org/draft-03/schema",
"name":"Service Object",
"id":"http://compose.so",
"required":true,
"properties":{
	"actions":{
		"type":"array",
		"id":"http://compose.so/actions",
		"required":false,
		"items":{
			"type":"object",
			"id":"http://compose.so/actions/0",
			"required":false,
			"properties":{
				"description":{
					"type":"string",
					"required":false
				},
				"lastUpdate":{
					"type":"number",
					"required":false
				},
				"name":{
					"type":"string",
					"required":true
				},
				"parameters":{
					"type":"object",
					"required":false
				},
				"status":{
					"type":"string",
					"required":false
				}
			}
		}
	},
	"links":{
		"type":"array",
		"id":"http://compose.so/links",
		"required":false
	},
	"properties":{
		"type":"object",
```

```
            "id":"http://compose.so/properties",
            "required":true,
            "properties":{
                  "customFields":{
                        "type":"object",
                        "required":false
                  },
                  "description":{
                        "type":"string",
                        "required":true
                  },
                  "help":{
                        "type":"string",
                        "required":false
                  },
                  "id":{
                        "type":"string",
                        "required":true
                  },
                  "mode":{
                        "type":"string",
                        "required":false
                  },
                  "name":{
                        "type":"string",
                        "required":true
                  },
                  "subscriptionOptions":{
                        "type":"array",
                        "required":false,
                        "items":{
                              "type":"string",
                              "required":false
                        }
                  },
                  "ui":{
                        "type":"string",
                        "required":false
                  }
            }
      },
      "streams":{
            "type":"array",
            "minitems":"1",
            "id":"http://compose.so/streams",
            "required":true,
            "items":{
                  "type":"object",
                  "required":false,
                  "properties":{
                        "channels":{
                              "type":"array",
                              "minitems":"1",
                              "required":true,
                              "items":{
                                    "type":"object",
```

```
                                        "required":false,
                                        "properties":{
                                                "current-value":{
                                                        "type":"number",
                                                        "required":false
                                                },
                                                "name":{
                                                        "type":"string",
                                                        "required":false
                                                },
                                                "type":{
                                                        "type":"string",
                                                        "required":false
                                                },
                                                "unit":{
                                                        "type":"string",
                                                        "required":false
                                                }
                                        }
                                }
                        },
                        "customFields":{
                                "type":"object",
                                "required":false
                        },
                        "description":{
                                "type":"string",
                                "required":true
                        },
                        "documentation":{
                                "type":"string",
                                "required":false
                        },
                        "lastUpdate":{
                                "type":"number",
                                "required":false
                        },
                        "name":{
                                "type":"string",
                                "required":true
                        },
                        "provenance":{
                                "type":"object",
                                "required":false,
                                "properties":{
                                        "copiedfrom":{
                                                "type":"string",
                                                "required":false
                                        },
                                        "createdby":{
                                                "type":"string",
                                                "required":false
                                        },
                                        "derivedfrom":{
                                                "type":"array",
                                                "required":false
```

```
                                        },
                                        "invalidatedby":{
                                                "type":"string",
                                                "required":false
                                        },
                                        "usedby":{
                                                "type":"array",
                                                "required":false,
                                                "items":{
                                                        "type":"string",
                                                        "required":false
                                                }
                                        }
                                }
                        },
                        "type":{
                                "type":"string",
                                "required":false
                        }
                }
        }
},
"subscriptions":{
        "type":"array",
        "id":"http://compose.so/subscriptions",
        "required":false,
        "items":{
                "type":"object",
                "required":false,
                "properties":{
                        "customFields":{
                                "type":"object",
                                "required":false
                        },
                        "delay":{
                                "type":"number",
                                "required":false
                        },
                        "expire":{
                                "type":"number",
                                "required":false
                        },
                        "forceUpdates":{
                                "type":"boolean",
                                "required":false
                        },
                        "stream":{
                                "type":"string",
                                "required":false
                        },
                        "subscriberId":{
                                "type":"string",
                                "required":false
                        },
                        "subscriptionId":{
                                "type":"string",
```

```
                                "required":false
                        },
                        "type":{
                                "type":"string",
                                "required":false
                        }
                }
        }
}
}
```

Appendix 3: Web Socket mapping of Web Streams Protocol

The Web Streams protocol is inherently HTTP-REST based, therefore it defines some requirements that devices supporting this communication model have to fulfil in order to exploit its full potential. There are two common scenarios where this protocol might have difficulties to be implemented. First, not all the devices used for sensing information support HTTP servers. For this situation we have considered a simplified version of the protocol, as described for the client-based Web Objects. Second, network configuration, especially for mobile based environments might prevent devices to be accessible from other networks, making them by force to behave like the client-based version. To overcome this scenario we allow for a simple mapping of the protocol to Web Sockets. If a device is capable of supporting Web Sockets, it can provide the same functionality as the server-based part of the protocol.

The operation of the Web Streams sub-protocol for Web Sockets is as follows[5]:

- The Web Object establishes a Web Socket connection with a web sockets endpoint of the COMPOSE platform.
- When the socket is created, the rest of the protocol applies as defined in the REST interface. Messages are encapsulated in the WebSocket in a JSON object.

The mapping of an HTTP request is done by a JSON object with simple properties representing the HTTP necessary content (method, path, headers and message body). HTTP responses are equally encapsulated in a response object containing the status, the headers and the message body.

Table 12: Example of how to get information about the properties of a Web Object encapsulating the GET request for use in the web socket.

```
{
	"serviceObjectId":"ServiceObject-12312312",
	"request":{
		"method":"GET",
		"path":"/properties/",
		"headers":[
			{
				"name":"Accept",
				"value":"application/json"
			}
		],
		"messageBody":{
		}
}
```

[5] Inspired by https://github.com/wordnik/swaggersocket

Example of how a response, web socket based, to the GET request presented before.

Table 13: Example of how to represent a response, web socket based.


```
{

	"serviceObjectId":"ServiceObject-12312312"

	"response":{
		"status":"200",
		"headers":[
			{
				"name":"Content-type",
				"value":"application/json"
			}
		],
		"messageBody":{
			"properties":{
				"id":"WebObject-2423529879879875",
				"name":"Shopping Cart",
				"description":"Shopping Cart that updates location
					information as it moves",
				"mode":"server",
				"subscriptionOptions":[
					"http.callback",
					"http.websocket"
				],
				"customFields":{
					"creator":"COMPOSE Consortium Hardware Team"
				}
		}
	}

}
```