OGEMA
Introduction of Concepts, Terminology and Framework Services

Version information

This document refers to OGEMA release 2.0. It is not compatible to previous releases of OGEMA.

Copyright

OGEMA

Table of contents

1 Introduction and Terminology

1.1 OGEMA Framework Overview

The core concept of the OGEMA framework is to provide a hardware independent execution environment for energy management applications. The software is designed to be installed on a gateway computer situated between the customer and the smart grid. It acts as a firewall between the public and private communication systems, allowing only certain interactions between the systems as defined by the gateway configuration. Applications installed in OGEMA can obtain access to costumer devices, user displays, smart meters, measuring data, as well as data provided by external market participants, like tariff information or grid parameters. The goal is to provide a platform for Smart Building and Smart Home applications supporting the full range of Smart Grid applications at the customer side with a single efficient hardware platform, which features all the necessary communication connections (see Figure 1). New devices and functionalities can be connected and added in a "plug&play" manner with minimum user interaction.

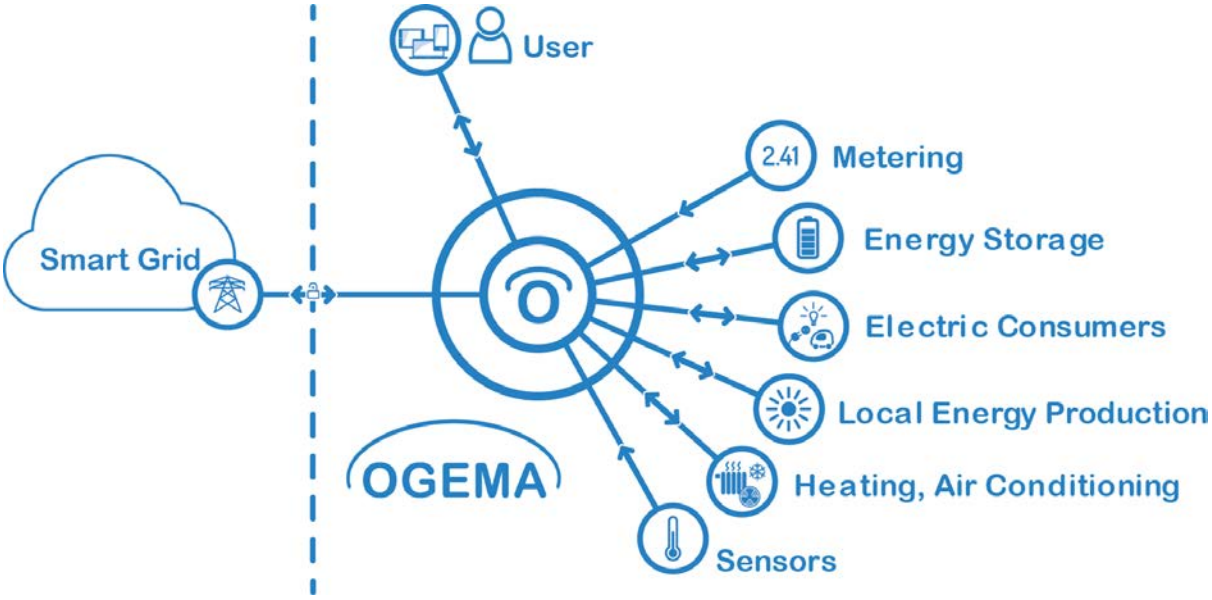

Figure 1: OGEMA framework in a Smart Grid / Smart Building environment. A central OGEMA gateway connects with the user's devices, unifying them into one system even if they come from different vendors and use different communication protocols. On the gateway, different energy or comfort applications may be running, which can be configured by the user connecting to the gateway with a local computer or a mobile device. The framework is logically situated between the user/customer and his/her devices (right side) and the Smart Grid (left side).

The OGEMA software acts as an operating system that allows applications to access different types of connected hardware and remote service providers without having to care about the actual physical realization of the connection. This is achieved by using drivers to take care of device-specifics. Just as in the case of a normal Operating System, say Linux, the drivers are hardware- or communication-specific pieces of software that transfer OGEMA language into commands for the devices or communications. The language of OGEMA is the representation of states in a data graph called resource graph and the manipulation of these data.
The framework itself implements basic functionalities such as management of applications and communication drivers, an API to work with the resource graph, user administration and a REST interface.

However, the real value for the customer comes with the installed applications and drivers that perform services related to energy management and home automation and allow unifying devices from different vendors into one system. Depending on the installed applications, different OGEMA systems can be realized with the OGEMA framework.

The actual OGEMA specification, including the API required for the development of OGEMA applications, is provided in terms of a Javadoc (a set of HTML-pages auto-generated from the annotated Java source code) at https://ogema-source.net/apidocs**. Further instructions and examples for developers can be found on the OGEMA Wiki at** https://www.ogema-source.net/wiki**. A reference implementation developed by Fraunhofer can be downloaded from http://www.ogema.org .**

1.2 OGEMA Applications and Drivers

An application is a piece of software running on an OGEMA framework that has been installed by the framework administrator and performs one or more tasks. For example, an application could display the time-resolved power consumption of a premise, or allow to dim lights. Application tasks can also be automated and be carried out by the framework without any further interaction with the user. They can, for instance, be carried out continuously/periodically, as a result of a framework event (such as the state of a device connected to the system having changed) or schedule driven (e.g. following a charging schedule for an electric car that has been computed by another application or been provided from outside the framework). The tasks an application is allowed to carry out can be restricted by security permissions set upon its installation.

A special case of OGEMA applications are drivers. A driver creates and maintains a connection to a physical device or an external data source and makes information on the physical device or external data source transparent to the framework and the other applications installed (see section 1.3 for an explanation of how information is shared between applications). Similarly, a communication driver can pass information from the framework to the connected system. For example, a driver might enable the control of a connected heating device by applications installed on the system.

1.3 Resources and the Resource Graph

OGEMA is a resource-based system with a central graph like data structure, whose nodes are called resources. The idea of the central resource graph is that it shall represent the current state of the total system, as well as relevant future predictions about the state (forecasts) and relevant past states (log data). A Resource is a representation of states, parameters or communication data as a data structure within the OGEMA framework, so for instance it can represent a physical device, the parameters of a communication driver or data transmitted to the OGEMA gateway from the Smart Grid, such as a price profile. Resources are exposed to all applications on the system, provided security settings grant them the required permissions. Applications interact with the system by reading and modifying the resource graph, irrespective of who provided the data that is read and who takes the modification made as an input. OGEMA applications hence are very modular and portable. For instance, an application that converts a temperature setpoint for a room into an operation schedule for a heater needs not to know which application provides the setpoint; it will be informed by the framework about any changes in the setpoint resource (see 1.5).

The organization of the resource graph can be considered a set of multiple tree-like structures. The difference from real resource trees is that branches of trees can be incorporated into other trees via references (cf. Figure 2). The branch then belongs to both trees. References are transparent for most cases, but in cases where it matters a unique tree exists that it "really" belongs to. For example, in Figure 2, seen from the trees' top-most elements, the top-level resources, two tree structures are seen which are referred to as resource trees. Child nodes of a resource are called sub-resources, the parent node of a resource is called its parent resource. If a tree branch is shared between multiple resource trees, the parent resource of the branch's top-most resource depends on context, i.e. on the resource tree from which the resource is addressed. Note that references can create loops: via references, a resource can be its own sub-resource.

The resource graph is, for the most part, stored persistently on some form of permanent data storage device (e.g. the hard disk of the computer the system runs on). This means that the state of an OGEMA

system is not lost on a system shutdown, but when the system is re-started, the last state stored is recovered. This allows continuing operation after a planned system restart or power failure without having to reconfigure the system. Sensor data are not stored persistently, as their values are usually not expected to equal the last state before the system shutdown.

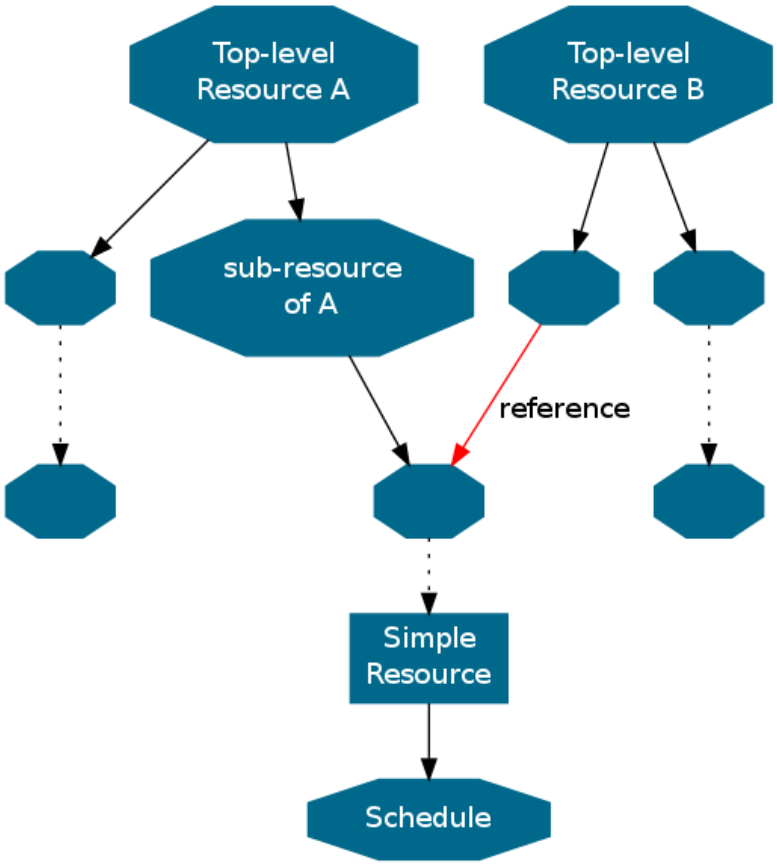

Figure 2: Sketch of an example state of resources in an OGEMA system for illustration of the concepts and terminology introduced in section 1.3. Shown is a case of two resource trees, starting from top-level resource A and B. The two trees share one branch of tree A which is inserted into tree B via a reference. The shared branch ends with a simple resource that stores a value and, in this case, has a schedule on it that contains information about the future values of the simple resource.

1.4 Resource Types and the Data Model

To achieve interoperability, it is indispensable that all applications agree on where exactly in the resource graph information is being placed and how it is encoded. The authority defining this is the OGEMA data model, which defines rules for resource graph's structure. Its most fundamental rule is that every resource has an associated resource type that defines what type of information the resource represents and what its possibly-expected sub-resources are. Apart from purely structural resource types three important classes of resource types exist: Simple resources are resource types whose resources store actual values. A special sub-set of simple resources are unit resources, which are used to represent physical quantities. Schedules are the resource version of time series. They are attached to simple resources and typically provide future information about them (forecasts and programs). Time series are functions over time, which are defined by a set of support points (each with timestamp, value

and quality) and an interpolation mode that defines what the functions' values are between the support points.

The OGEMA data model is designed to be very flexible, ideally being able to encode any piece of information that is available. Three concepts have been chosen to facilitate this. First of all, all elements in the data model are optional elements, meaning that the data model defines where a particular piece of information would be inserted but does not require the piece to be actually present. This allows having a large data model eventually taking care of many different use cases. Secondly, applications can provide their own custom resource types and use them instead or in addition to the OGEMA data model. The use of these custom types should be avoided if possible – it breaks compatibility with all applications not knowing these types. But sensible uses could be an application wanting to persistently store some configuration data (that no other application needs to understand) or a scenario where developers create multiple applications that are intended to work only with another. Finally, as a "light version" of custom resource types, decorators can be used. A decorator is a sub-resource added to a resource that is not defined as an optional element in the resource's resource type. Decorators should be used sparsely for the same reasons as with custom resource types.

1.5 Event Messaging

The resource-based approach used in OGEMA enables a message-oriented style of application programming. Specifically, the framework allows for applications to add listeners of different type on the resource graph. Whenever a modification on the resource graph happens that matches the requested event type, the listener is informed about the change via a callback message. Adding a listener is also referred to as issuing a demand. The listeners related to changes in the resource graph are detailed in section 2.5.

A somewhat unique event listener is a timer listener that can be attached to a timer. A timer is a periodic count-down clock that can be used by applications that need to repeatedly perform a task with an a-priori known frequency. The timer listener invokes a call-back whenever the timer has elapsed. OGEMA pseudo-serializes all callback messages: Different applications run in parallel but for a given application the framework only issues listener callbacks one at a time and waits for the callback to finish before issuing the next one.

2 OGEMA Services and Requirements

Having introduced key terms in the introduction section this section will give a brief overview over the different services that the OGEMA framework offers to applications running in it. The focus of this section is on information and description of concepts that cannot be described in the API itself very well.

2.1 Hardware, Operating System, Java-VM/OSGi Requirements

OGEMA implementations can run on a large variety of devices, including PCs and embedded computers with low energy consumption. The framework is Java based and requires a Java Virtual Machine (JVM) to run on the underlying platform (see Figure 2). The minimum requirement is Java API and Language Specification Version 1.7. Due to binary compatibility of Java 7 with Java 6, OGEMA applications could run within a Java 6 environment, too. OGEMA sources of the reference implementation must be built with the Java compiler version 7 or higher, however, because new Java 7 language features are used.

The application life cycle management is based on an OSGi-Framework, which shall be included in each OGEMA installation, and allows different energy management, control and monitoring applications to run in parallel and to be started, updated or stopped at runtime. The underlying OSGi framework has to be compatible to OSGi Service Platform Core Specification Release 4, Version 4.2.

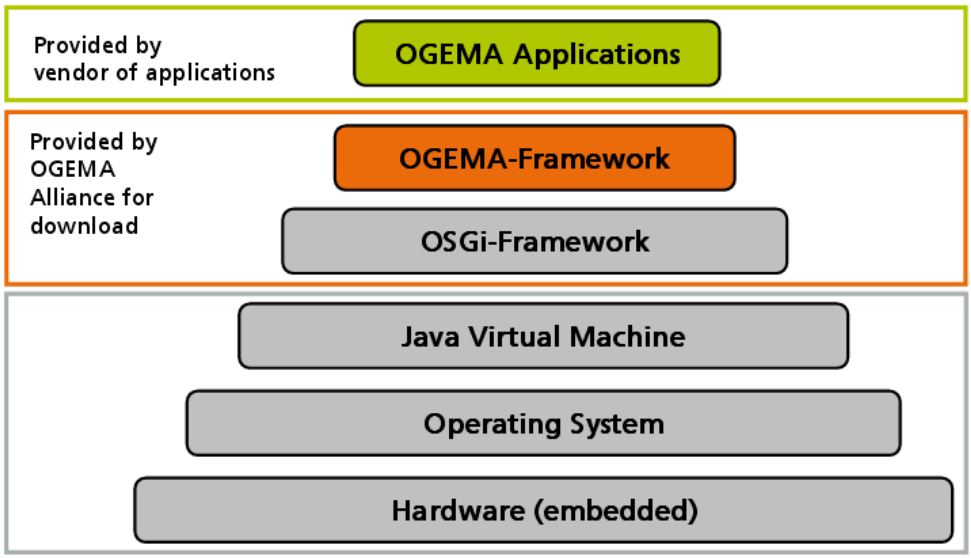

Figure 3: Technology stack that an OGEMA system builds upon. The OGEMA framework runs on top of an OSGi framework, which itself requires a Java Runtime Environment. The latter is readily available for most operating systems.

2.2 Installation and Management of Applications

OGEMA applications and drivers can be installed via marketplaces or from a file folder. The API functionalities for administrative applications that install new applications (including administrator interaction) are defined in org.ogema.installationManager. Before the installation, the required permissions must be declared by the application and must be approved by the administrator (for more details see Section 2.13 Rights / permissions management).

Declaration of required and optional permissions requested is made by two files: permissions.perm and OGEMAPermissions.perm. The permissions.perm file contains Java permissions (extending java.security.Permission) according to the Java specification [1], and OSGi-permissions according to the OSGi API documentation [2]. Its file format is specified in Section 9.10 "Bundle Permission Resource" of the OSGi specification [3]. The OGEMAPermissions.perm file contains OGEMA-specific permission statements that do not extend Java permissions and hence cannot be checked via Java's security mechanisms. The permissions in this file can be attributed as "REQUIRED" which means that the application must have this permission to work properly, or as "OPTIONAL" which means that this permission is used for extra functionality and the application works properly without this permission.

OGEMA applications export the OSGi service application, for which the methods start() and stop() are defined. The framework automatically detects applications based on this service. To start and stop the applications, the respective methods of the application are called. When calling the start method, the application is being passed a reference to an ApplicationManager that serves as the application's entry point to the OGEMA framework. Applications hence do not need to (and should not be permitted to) get OSGi services just for the sake of getting access to OGEMA.

2.3 Driver Structure, Channel Manager

OGEMA offers to develop drivers in two components: A protocol driver or low-level driver that usually implements a certain communication protocol (e.g. Modbus or ZigBee) and a resource driver or high-level driver that employs the low-level driver to establish and maintain the connection between specific devices speaking the protocol and their representations in the resource graph. For the OGEMA framework, high-level drivers are applications, while low-level drivers instead register the ChannelDriver service with OSGi (instead of the Application service).

The interface between the two drivers is the channel manager. It provides and manages a set of generic data channels, which have a naming structure defined by the protocol driver and can generally contain any kind of data. The actual kind of data is again defined by the protocol driver, implying that a resource driver must know the interface of the protocol driver it employs. The channel manager allows to create channels, read and write them, and to register listeners on channels.

2.4 Resource Management

OGEMA resources have resource names. For top-level resources, the name can be freely chosen by the application creating the top-level resource. The names of sub-resources are defined by the resource type of their parent resource – with the exception of decorators, for which again the creating application can choose a name. To avoid naming conflicts between different applications, the method getUniqueResourceName has been defined on the ResourceManager service, which ensures that two applications requesting the same resource name will not get the same name but a unique one that is similar to the requested name.

The nodes in the resource graph can be addressed via paths, starting at some top-level resource (with a unique name) and then following a sub-resource trail down to the respective resource. This path is not necessarily unique for a given resource: different top-level resources may share a common sub-resource which has been inserted into either's resource tree via a reference. In principle, any number of

paths to a resource can exist. But there is always exactly one path leading to the resource that does not contain references. This path is called the location of the resource. Access permissions to a resource are determined by its location, while listeners on nodes of the resource graph are determined by path.

An application can parse and modify the resource graph via the methods defined in the resource management services ResourceManagement and ResourceAccess, as well as Java objects of type Resource. Initial access to top-level resources has to be performed via the resource management services, that return a Resource object (this is true for direct access as well for indirect access via resource demands, which also have to be registered with the resource management). Given an initial Resource object, it is possible to navigate further along the resource graph to which the object is connected. Resource objects are always associated to a path.

On top of the resource access granted to an application in terms of permissions, there are three different access modes to a resource that applications can have, which refer to the values in simple resources. By default, an application is granted shared write access to a resource, which means it can read and write the value. If an application determines that no other application should write the values in a resource, it can demand exclusive write access with some priority. An application that is granted exclusive write access takes the ability to write to the resource from all other applications, leaving them with read access. Higher priorities override lower ones; a demand for shared write access always has a lower priority than any exclusive write demand. Applications can add listeners to resources to be informed about changes in their access states.

Navigation via the resource objects allows navigation into graph nodes that do not exist (yet). Such handles on a non-existing resource are called virtual resources. It is possible to register listeners on virtual resources (e.g. to be informed when someone creates the node) and to call the create() method on them. After creation a resource is inactive. In this state it can be navigated to and be manipulated, but it is invisible to resource demands, and therefore to most other applications. To make it visible the resource must be activated, putting it into the active state. OGEMA does not provide default values for simple resources. An application must provide a sensible value before activating a simple resource. If it cannot do so, it should not activate the resource.

In some cases modifications to the resource graph cannot be performed with a single call to the OGEMA services offered, and a so-existing intermediate state would lead to an illegal state of the resource graph (e.g. re-setting an interval which implies re-setting the minimum and the maximum value). To avoid illegal intermediate states of the resource graph applications can combine multiple commands into transactions. Transactions are objects that can be requested by the ResourceManager and filled with commands. Upon invocation, all of the commands are then executed in one “atomic” operation that guarantees that no application will see an intermediate state in which only part of the commands have been executed.

OGEMA allows defining complex resource patterns. Resource patterns consist of a definition of the root node’s type and a set of other resources with a resource type and a defined path relative to the root node. Applications can use such resource patterns to create instances of it in the resource graph – the root node then is a top-level resource. Also, OGEMA allows registering a resource pattern demand that allows an application to be informed and updated about all the matches of the pattern in the resource graph. In the latter case, the root node does not need to be a top-level resource. See also 1.5: Event Messaging.

2.5 Resource Listeners

The OGEMA API defines a set of different event listeners related to changes in the resource graph.

1. A resource demand listener is registered on a particular resource type, and invokes a callback whenever a resource of the respective type is activated or deactivated. Additionally, it is invoked for each suitable active resource once directly after registration. The resource demand listener is designed for the task of finding and keeping track of all active resources of a given type.
2. Resource structure listeners are registered on individual resources (implying they are registered on a path, not a location). Once registered, they are informed about structural changes of this resource, such as creation/destruction, activation/deactivation, adding/removing of sub-resources or the change of the path's location. A change of a simple resource's value is not reported to a structure listener – the more specialized resource listener has been defined for this type of event.
3. Access mode listeners can be registered on individual resources to inform the registering application about changes of its access mode to the respective resource. This includes access mode changes caused by the application's own change requests and changes caused by another application's requests (e.g. losing write access due to the other application being granted exclusive write access). The change in access mode is not a structure event, since it is specific to each application, while resource structure events are identical for all applications.
4. A resource change listener is registered on an individual resource, and leads to a callback whenever the value or the time series of this resource is written to (even if the write command re-writes the same value or time-series again). The listener can be registered recursively, in which case the listener is invoked whenever the value/schedule of any of the resource's sub-resources changes. Due to the reference and the decorator mechanism of the resource graph, recursive registration can lead to unexpected callbacks in a complicated resource graph. A non-recursive resource change listener registered on a non-simple resource can never be invoked.
5. A resource pattern listener is the equivalent of the resource demand listener registered on a resource pattern instead of a resource type. Instead of keeping the application informed about all active instances of a resource type it keeps the application informed about all active matches of a resource pattern.

2.6 Persistent Data Storage

With the exception of sensor readings OGEMA resources and their values are stored persistently in the OGEMA database (the exceptions are explicitly marked as such in the data model). When the OGEMA framework is re-started after a shutdown, a black-out or a crash, the last state of the resource graph is recovered. Resource demands, resource access modes and virtual resources are considered part of the applications and drivers (which are re-started), so they are not stored persistently and need to be re-issued by the applications. Applications wanting to store data persistently can do so using custom configuration resources.

2.7 Hardware Manager

The hardware access interface provides information on hardware interfaces available as well as hardware devices connected to these interfaces. Devices are connected to the OGEMA gateway mainly via USB. But there are other devices which could be connected via digital IO or native UARTs.

The HardwareManager is responsible to manage a list of currently available hardware. Additional Information is available in HardwareDescriptors that can be accessed via the HardwareManager and contain additional information according to the connection type of the device. When an OGEMA low level driver is installed, the administrator can associate the driver with a device by selecting a device description string from the list of device location identifiers provided by the hardware manager. These associations are stored persistently and compared after each reboot with the existing hardware configuration and updated if necessary. If a device is removed or added, the table of devices is updated so that it reflects the current state of the hardware configuration.

2.8 Logging

Two types of logging are supported by OGEMA, a text logging support for applications and an automatic logging of past resource states. The text logging via the OgemaLogger is created as a direct extension of the slf4j framework [4]. The logger API suggests applications how to perform their logging, but the actual implementation of the log commands is largely left to the framework. Particularly, an administrator application can get access to the loggers via the AdminLogger interface, which allows to configure some of the loggers' behavior.

Some simple resource types can be configured for automatic logging. The log data and their configuration are available from the method getHistoricalData() defined in the resources' interface. Data logging can be configured for periodic logging or for logging whenever the resource's value changes. The historic values are stored persistently as a time series. Access to them can be done via the direct access to the time series' entries or, alternatively, with a filtering function (e.g. the mean value over given time intervals).

2.9 Textual Resource Representations

Resources are given to application in the form of Java objects. For communication with other partners, e.g. external applications or Javascript code, a textual representation of resources is available. The textual representations are in XML format, with the XML structure being defined in the Ogema.xsd file. The conversion from and to text is performed by the SerializationManager.

2.10 REST Interface

To facilitate remote access to the resource graph, the OGEMA frameworks provides a REST interface supporting RESTful communication using XML as well as JSON, and other communication requirements according to RFC2616 [5]. Each request to the REST interface must be https-encrypted and supply a username identifying the user (or application) whose permissions will be used for the access, and the correct password for the user. Requests send via REST are then performed with that user's access rights.

Each OGEMA resource accessible to applications is also accessible via an URI given by .../rest/resources/<path>, where "..." is the base URI of the OGEMA web server, and "<path>" is a valid path to the resource. For schedule resources, up to two additional pseudo-paths /t1/t2 can be appended to the URL, which must be in the form of time stamps (in ms since epoch). If the first extra path is given,

all operations performed on the schedule exclude the entries before t1. If the second pseudo-path is also used, schedule entries whose timestamp equals at least t2 are excluded, too. This way, sub-intervals of schedules can be selected and operated on.

The standard HTTP methods applied to these URIs can be used for interaction with the resources. The GET request corresponds to reading a resource. It returns a textual representation of the resource. For additional configuration of the representation, serialization options can be encoded into the URL in the form "<URL>?<attribute1>=...&<attribute2>=... " . Possible attributes are *depth* (an integer defining the maximum parsing depth starting from the addressed node), *references* (a true/false boolean defining whether references should be parsed as sub-resources (true) or just as links) and *schedules* (boolean defining if schedules should be included (true) or just linked-to). The default is "`?depth=0&references=true&schedules=false`" . Resources whose location is encountered a second time during the processing of a request can be included as links, irrespective of the `references` settings. This terminates the processing of possible loops in the resource graphs and avoids blowing up response messages to a request with a large `depth`.

The PUT request can be sent to any resource URI and requires an attached textual resource representation. If no resource exists at the URL, a 404 error is returned. PUT applies the attached resource representation to the resource at the selected URL as if the message was applied to the resource via the SerializationManager In case of schedules resources, the whole schedule content is replaced with the contents of the message. If only a sub-interval of the schedule was selected via pseudo-paths, the schedule entries outside the selected interval are unaffected by the operation. As a result of the request, the equivalent of a GET on the same URL after the request has been processed is returned.

POST with an attached resource is used to attach a new resource to the URL it is sent to. Contrary to a PUT it can also be sent to .../rest/resources, in which case it creates a new top-level resource. A 406 Error is returned if a top-level resource with the given name already exists. If the request is sent to a resource location the processing of the event depends on whether a sub-resource with the name as the resource in the message exists. If such a sub-resource already exists, a PUT with the attached message is performed on this sub-resource (which may cause an error if the resource types do not match). Otherwise the resource message is attached to the OGEMA resource the request was sent to as a child node, either as an optional field or as a decorator. Type and name of the new resource are inferred from the respective entries in the message. If the message contains a valid xs:ResourceLink, a reference to this resource is created. Sub-resources or decorators are created only if they are explicitly listed as sub-resource in the XML message. If POST was successful, the result of a GET request on the new resource's URL is returned. Otherwise, an error is returned.

Sending a DELETE message to a resource URL attempts to remove the respective OGEMA resource. It is equivalent to calling the delete() method on the resource. A successful delete returns an empty document. An unsuccessful delete returns an HTTP error code.

Access to the log data of resources is available via .../rest/recordeddata/<path>, where again <path> is the path of the resource to access. These URIs accept only the GET request and return a time series (same as a schedule) of all the recorded data. The /t0/t1 pseudo-paths are available for the GET request to only get the data points of a sub-interval.

2.11 User Interface

The definition of how OGEMA applications can provide user interfaces is very loose, leaving a lot of freedom to the application programmers. Applications can register a folder with web resources with the WebResourceManager and provide the URL they want to register the resources under. Access to these registered web resources is then controlled by OGEMA, based on the access rights of users. Specific implementations of the OGEMA specification may define additional GUI-related services.

2.12 Data Models

All resource types are Java interfaces extending org.ogema.model.Resource. The basic resource types containing actual values are defined in the package org.ogema.model.simple; their respective array resources are defined in package org.ogema.model.array. For the representation of physical properties, specialized resource types exist for the most common types of physical properties, which are defined in org.ogema.model.unit. On top of defining the nature of the property represented by the resource, they also define the physical unit the property is measured in.

The OGEMA API only defines the most basic resources. Complex resource types and rules how to construct a resource graph using them are defined in the OGEMA data model. It is contained in the model package; its Javadoc can be found online at http://www.ogema-source.net/apidocs.

2.13 Rights / permissions management

The OGEMA framework assigns rights to user accounts and applications. As OGEMA applications usually operate independently of actual user interaction (most applications are for control and analysis functionalities), user and application permissions are defined independently. For the same reason, user-related permissions are only relevant regarding user interface access. So the only user account-related permission defines the user interface from which applications may be accessed (see org.ogema.administration.UserAccount). All other permissions are granted to applications (see org.ogema.administration.AppAdminAccess). For details about access permissions see also the OGEMA Security Specification document, which is included in the OGEMA releases and available on the OGEMA Wiki.

3 Acknowledgements

This specification has been developed by the Fraunhofer Society in the context of the research project "OGEMA 2.0" funded by the German Federal Ministry for the Environment, Nature conservation, Building and Nuclear Safety and the German Federal Ministry for Economic Affairs and Energy between 2011 and 2015. Participating institutes were the Fraunhofer Institute for Integrated Circuits (IIS), the Fraunhofer Institute for Solar Energy Systems (ISE) and the Fraunhofer Institute for Wind Energy and Energy Systems Technology (IWES).

4 References

[1] [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/security/permissions.html.

[2] [Online]. Available: http://www.osgi.org/javadoc/r4v42/.

[3] The OSGi Alliance, „OSGi Service Platform Core Specification, Release 4, Version 4.2," 2009.

[4] [Online]. Available: http://www.slf4j.org/apidocs/index.html.

[5] „RFC2616," [Online]. Available: http://www.w3.org/Protocols/rfc2616/rfc2616.html.

[6] „RXTX – The Prescription for Transmission," [Online]. Available: http://users.frii.com/jarvi/rxtx/.

[7] „Java Communications API," [Online]. Available: http://www.oracle.com/technetwork/java/index-jsp-141752.html.

[8] „Linux Hotplugging," [Online]. Available: http://linux-hotplug.sourceforge.net/.