**eWALL Cloud Middleware**



**This document presents an overview of eWALL that can be used by the eWALL Open Source Community. The document belongs to eWALL consortium and any use of the document requires the reference to eWALL Open Source (http://gitghub.com/ewallprojecteu)**

# Table of contents

[Table of contents 2](#_Toc505251535)

[1. Cloud Middleware positioning in the eWALL architecture 4](#_Toc505251536)

[2. eWALL Cloud Infrastructure Layer 6](#_Toc505251537)

[2.1 Hardware dedicated to the eWALL cloud 7](#_Toc505251540)

[2.2 eWALL cloud components 7](#_Toc505251541)

[2.3 Access to the eWALL cloud resources 10](#_Toc505251542)

[2.3.1 HTTP access to the eWALL cloud management dashboard 11](#_Toc505251543)

[2.3.2 SSH direct access to the eWALL cloud virtual machines 12](#_Toc505251544)

[2.3.3 HTTP(S) access to the eWALL services endpoints 13](#_Toc505251545)

[2.3.4 Cache services 14](#_Toc505251546)

[2.3.5 Monitoring and alerting services 15](#_Toc505251547)

[2.3.6 Object storage service 15](#_Toc505251548)

[2.4 Migrating OpenStack deployment to Amazon Web Services 16](#_Toc505251549)

[2.4.1 Converting OpenStack image to OVA format 16](#_Toc505251550)

[2.4.2 Uploading OVA file to Amazon S3 storage 17](#_Toc505251551)

[2.4.3 Import VM to Amazon EC2 18](#_Toc505251552)

[2.4.4 Prerequisites 18](#_Toc505251553)

[3 eWALL Platform within overall eWALL system architecture 20](#_Toc505251554)

[3.1 Data flows between home and cloud platform components 23](#_Toc505251555)

[4 Authentication and Authorization framework 26](#_Toc505251556)

[5 eWALL Platform Implementation 28](#_Toc505251557)

[5.1 Remote Proxy 28](#_Toc505251558)

[5.1.1 Black box description 28](#_Toc505251559)

[5.1.2 Requirements 28](#_Toc505251560)

[5.1.3 Features 29](#_Toc505251561)

[5.1.4 Local Data Manager 30](#_Toc505251562)

[5.1.5 Black box description 30](#_Toc505251563)

[5.1.6 Requirements 30](#_Toc505251564)

[5.1.7 Features 31](#_Toc505251565)

[5.2 eWALL Portal 32](#_Toc505251566)

[5.2.1 Black box description 32](#_Toc505251567)

[5.2.2 Requirements 32](#_Toc505251568)

[5.3 Notification Manager 35](#_Toc505251569)

[5.3.1 Black box description 35](#_Toc505251570)

[5.3.2 Requirements 35](#_Toc505251571)

[5.3.3 Features 36](#_Toc505251572)

[5.4 Profiling Server 37](#_Toc505251573)

[5.4.1 Black box description 37](#_Toc505251574)

[5.4.2 Requirements 37](#_Toc505251575)

[5.4.3 Features 38](#_Toc505251576)

[5.5 Login service 39](#_Toc505251577)

[5.5.1 Black box description 39](#_Toc505251578)

[5.5.2 Requirements 39](#_Toc505251579)

[5.5.3 Features 40](#_Toc505251580)

[5.6 Cross-component functionalities 41](#_Toc505251581)

[5.6.1 Black box description 41](#_Toc505251582)

[5.6.2 Features 42](#_Toc505251583)

[5.7 Artefacts developed within other WP4 tasks 43](#_Toc505251584)

[5.7.1 Data Manager 43](#_Toc505251585)

[5.7.2 Cloud Gateway 43](#_Toc505251586)

[5.7.3 Intelligent Decision Support System 44](#_Toc505251587)

[5.7.4 Commons data model 44](#_Toc505251588)

[6 Data Manager 46](#_Toc505251589)

[6.1 Positioning of the Data Manager block 46](#_Toc505251590)

[6.2 Data Manager functionality 46](#_Toc505251591)

[6.3 Data Manager interfaces 47](#_Toc505251592)

[7 Cloud Gateway 49](#_Toc505251593)

[7.1 Cloud Gateway functionality 49](#_Toc505251594)

[7.2 Cloud Gateway specification 49](#_Toc505251595)

[7.3 Cloud Gateway deployment 50](#_Toc505251596)

[7.4 Cloud Gateway interfaces 51](#_Toc505251597)

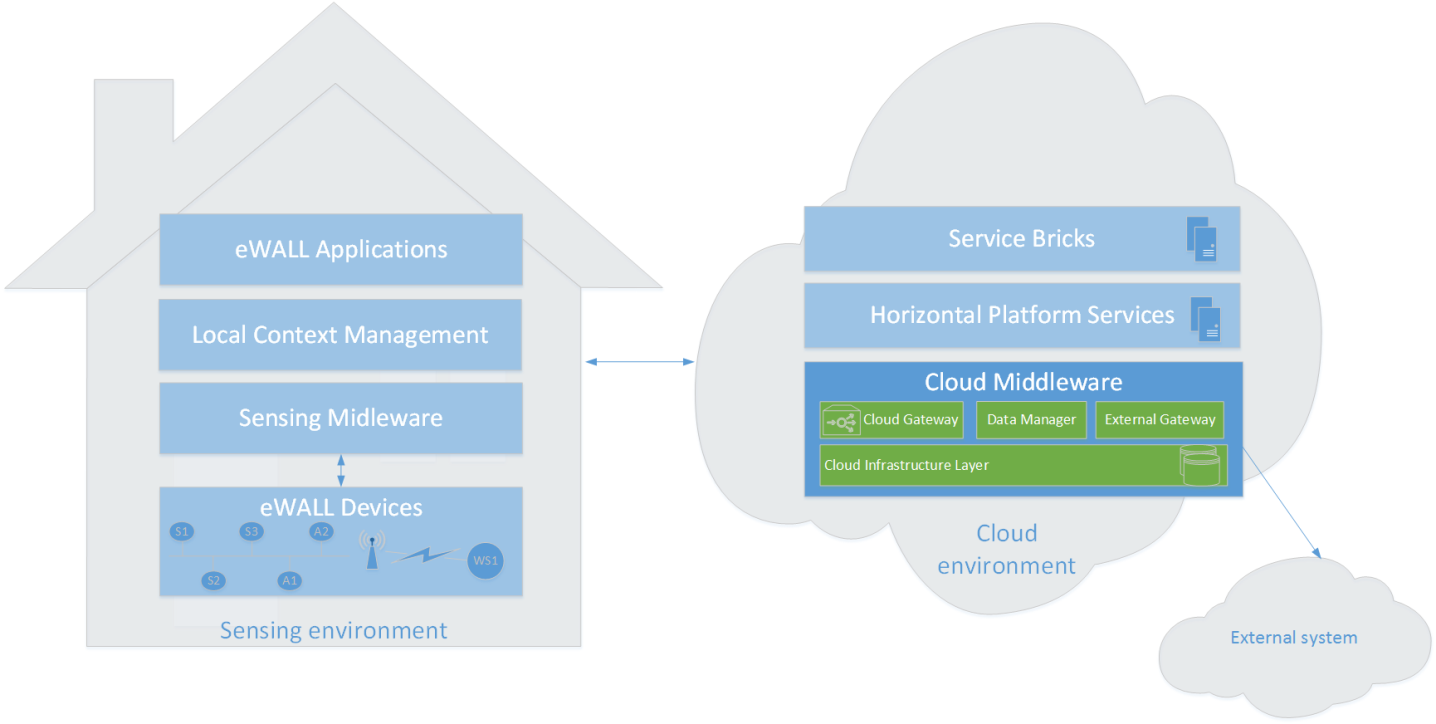
[8 Bibliography 54](#_Toc505251598)

# Cloud Middleware positioning in the eWALL architecture

The Cloud Middleware (CM) is one of the main architectural building blocks of the eWALL cloud. It is responsible for:

1. providing all cloud communication infrastructure, including communication within different service components within the cloud, but also communication with the Home Sensing Environment,
2. data abstraction and all data handling in the cloud (storing, retrieving, handling data integration from multiple sources etc.) taking into account privacy and security concerns,
3. hiding the heterogeneity and distribution of cloud hardware environment and management of physical resources and the provisioning of virtual resources - the service components that run in cloud just communicate with the Cloud Middleware and they may be unaware of the actual cloud hardware configuration that they are running on.

It interfaces Sensing Middleware to handle the exchange of information between eWALL sensing environments and cloud environment. This mainly relates to processed, indexed and congregated sensor data from sensing environment to cloud, but also device control data (i.e. actuator control) from cloud towards sensing environment. This interface is also responsible for the exchange of configuration and system control data, such as device and environment registration and remote configuration of sensing environment. Furthermore, it interfaces Horizontal Platform Services and Service Bricks to leverage the information exchange with higher layer blocks in cloud environment. This includes access to lower level data from sensing environments stored in the cloud, and storing and accessing higher-level data (output of various higher layer blocks, such as reasoning).



**Figure 1: T4.3 Cloud Middleware components in high level eWALL architecture**

As can be seen from **Figure 1**, the main functional components of this block are:

1. **Cloud Infrastructure Layer (CIL)**: provides the eWALL cloud infrastructure and the related services, and acts as an abstraction layer for the physical resources used in the eWALL backend for computational, communication and storage purposes. The CIL implements the Infrastructure as a Service (IaaS) cloud paradigm
2. **Data Manager (DM):** offers several features like basic data processing, format exchange, and input validation of received data. It uses Cloud Infrastructure Layer to seamlessly access physical storage facilities (databases) for data persistency.
3. **Cloud Gateway:** responsible for interconnecting eWALL cloud with Sensing Environments (Home and Mobile [1]). It is responsible for receiving processed and indexed measured sensor data, to send control and configuration data to Sensing Environments and for bidirectional application data. It directly communicates with Remote Proxy and supports both message based pull and push communication.
4. **External Gateway (EGw)**: responsible for interconnecting eWALL Cloud with other external sources such as epSOS. It is responsible for sending and receiving data to and from the external sources, and for bidirectional application data. It directly communicates with the Data Manager and Data Fusion blocks and supports both message based (pull/push) and streaming communication.

More details about each component are provided in next sections.

# eWALL Cloud Infrastructure Layer

As described in the deliverables “*D2.3, Preliminary system architecture* [1] and *“D2.7 Final user and system requirements and architecture* [2], one of the architectural components which enable the provisioning of the eWALL services is the Cloud Infrastructure Layer (CIL). The CIL provides the eWALL cloud infrastructure and the related services, acting as the abstraction layer for the physical resources used in the eWALL backend for computational, communication and storage purposes.

The CIL implements the *Infrastructure as a Service (IaaS)* cloud paradigm [3]. In this paradigm, a service provider offers the computing, storage and network resources on-demand to end users, together with the possibility to scale up and down the services which run on top of them dynamically, according to the requirements.

IaaS environments are typically offered by commercial cloud service providers, which own large datacenter infrastructures and sell their services to customers. In the eWALL project, in place of relying on an external provider, the Consortium opted for the installation of a private IaaS environment (the “eWALL private Cloud”), which allows for a higher degree of freedom in the management of the physical and virtual resources with respect to commercial services offered by external providers.

Having a private Cloud available allowed the Consortium to prepare and configure parallel execution environments and software lifecycle management tools for automating the software builds and the deployment of the software artefacts, hence reproducing execution scenarios which resemble almost exactly the final production environment. It also allowed to reach high degree of automation in the process of integration testing.

The eWALL cloud infrastructure is based on OpenStack[[1]](#footnote-2), which is the reference open source solution for creating private clouds. Among the main functionalities offered by OpenStack, it is possible to split logically the cloud environment into a set of so called “Projects” (or “Tenants”), which are independent resource areas where users, based on policies defined by an administrator, can manage virtual computational resources such as: virtual machines, virtual networks, virtual storage, etc.

Within eWALL, a set of “parallel” environments have been set up, each one dedicated to a specific phase of the software lifecycle management:

* The *alm* environment: dedicated to the hosting of virtual machines and tools which enable the whole process of Application Lifecycle Management
* The *dev* environment: dedicated to the hosting of a set of virtual machines that allow to test the software artifacts in the development phases
* The *int-level-1* environment: integration environment dedicated to the hosting of a set of virtual machines that allow (nightly) automated testing of the software artifacts released on the code repository
* The *int-level-2* environment: integration environment dedicated to the hosting of a set of virtual machines that allow end to end testing of the eWALL platform before releasing it into a final environment
* The *prod* environment: environment dedicated to the hosting of a set of virtual machines that allow the execution of the eWALL platform to be used by trial end users

More details on the parallel environments can be found in the eWALL deliverable *D6.1 Integration report* [1]*.*



## Hardware dedicated to the eWALL cloud

The OpenStack framework consists of a multitude of components that can be deployed according to a very flexible set of hardware scenarios. The most complex deployment scenario is represented by a cloud provider, where the hardware infrastructure consists of one or more datacenters with thousands of servers, and the components are deployed and replicated on dedicated high-level servers, as they must satisfy the workload generated by thousands of customers who pay for such services, with specific service level agreements ruled by contractual obligations.

The simplest deployment scenario, on the other hand, is represented by a set of virtual machines running on a laptop, where the OpenStack components are deployed and run using the very limited set of physical resources available. This kind of deployment is obviously usable only for proof-of-concept demos and for experimenting the functionalities of the platform.

In eWALL, we address an intermediate scenario where we set up a real, fully functional instance of the OpenStack platform into one single, but still very powerful, high-level server. This allows running a private cloud which satisfies all the requirements for the development, testing, application lifecycle management and runtime phases of the project. Anyway, the platform is open to extensions with additional physical resources.

The server dedicated to the eWALL private cloud infrastructure is hosted by the Telecommunication Department of the University Politehnica of Bucharest (UPB), and has the following characteristics:

* Model: HP PROLIANT DL180G6 server
* CPU: 12 x 2,80 GHz, Intel Xeon processors
* Storage: 3 TB HDD
* RAM: 96 GB RAM DRR3

## eWALL cloud components

The OpenStack cloud infrastructure consists of the following set of OpenStack modules, for which we provide a very high level description:

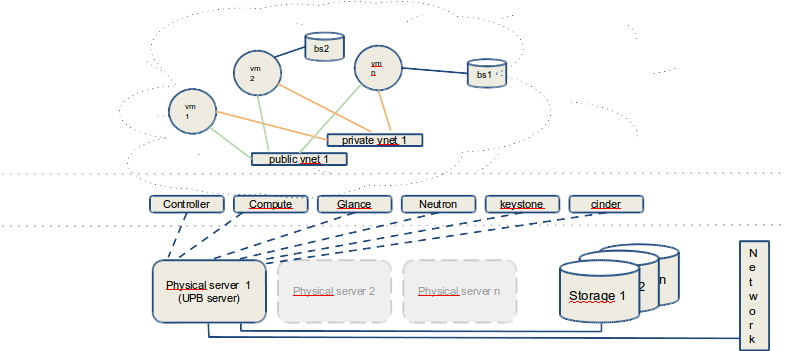
* **Compute**: the Compute module is installed on every physical server which is dedicated to run virtual machines. It basically interacts with the operating system hypervisor and manages all the operations related to the handling of the virtual resources on it.
* **Glance**: the Glance module is responsible for the management (creation, maintenance, provisioning) of virtual machine images, which are then made available to the users of the cloud platform to launch instances of virtual machines based on a specific image.
* **Neutron**: the Neutron module is responsible for the management of all the network-related operations of the cloud: creation of virtual networks, management of IP addresses, definition of access rules and firewalls, etc.
* **Keystone**: the Keystone module is the OpenStack module which provides identity, token, catalog and policy services throughout all the other OpenStack modules.
* **Cinder**: the Cinder module is responsible for the provisioning of on demand, self-service access to Block Storage resources to be used within virtual machines.
* **Swift**: the Swift module is responsible for the provisioning of on demand, self-service access to Object Storage resources to be used as remote storage areas (addressing the needs of applications which require the management of multimedia content).
* **Controller**: the Controller module is responsible for providing a set of management functionalities, which allow orchestrating the various OpenStack services provided by all the other modules. It hosts all the centralized functions like the cloud status database, the message broker, the compute and the storage schedulers, API endpoints, authentication services, image catalogue, orchestration engine, monitoring and accounting functions, the web dashboard server, etc.;

As mentioned in section 1, in eWALL we installed all the OpenStack modules on a single, powerful server. Therefore, the deployment view of the eWALL cloud reflects the one depicted in Figure 2, where the lower layer represents the physical resources, the mid layer represents the OpenStack components and the upper layer represents the virtual resources running in the cloud infrastructure.

In the physical layer, the “Physical server 1” hosts all the OpenStack components, as represented by the dotted line. In principle, the platform can be extended by adding new physical servers and the deployment of the components can be changed, spreading them over multiple servers. This concept is represented by the greyed “Physical server n” elements. The physical server is connected to the physical storage and network, via the corresponding drivers, interfaces and physical connections.

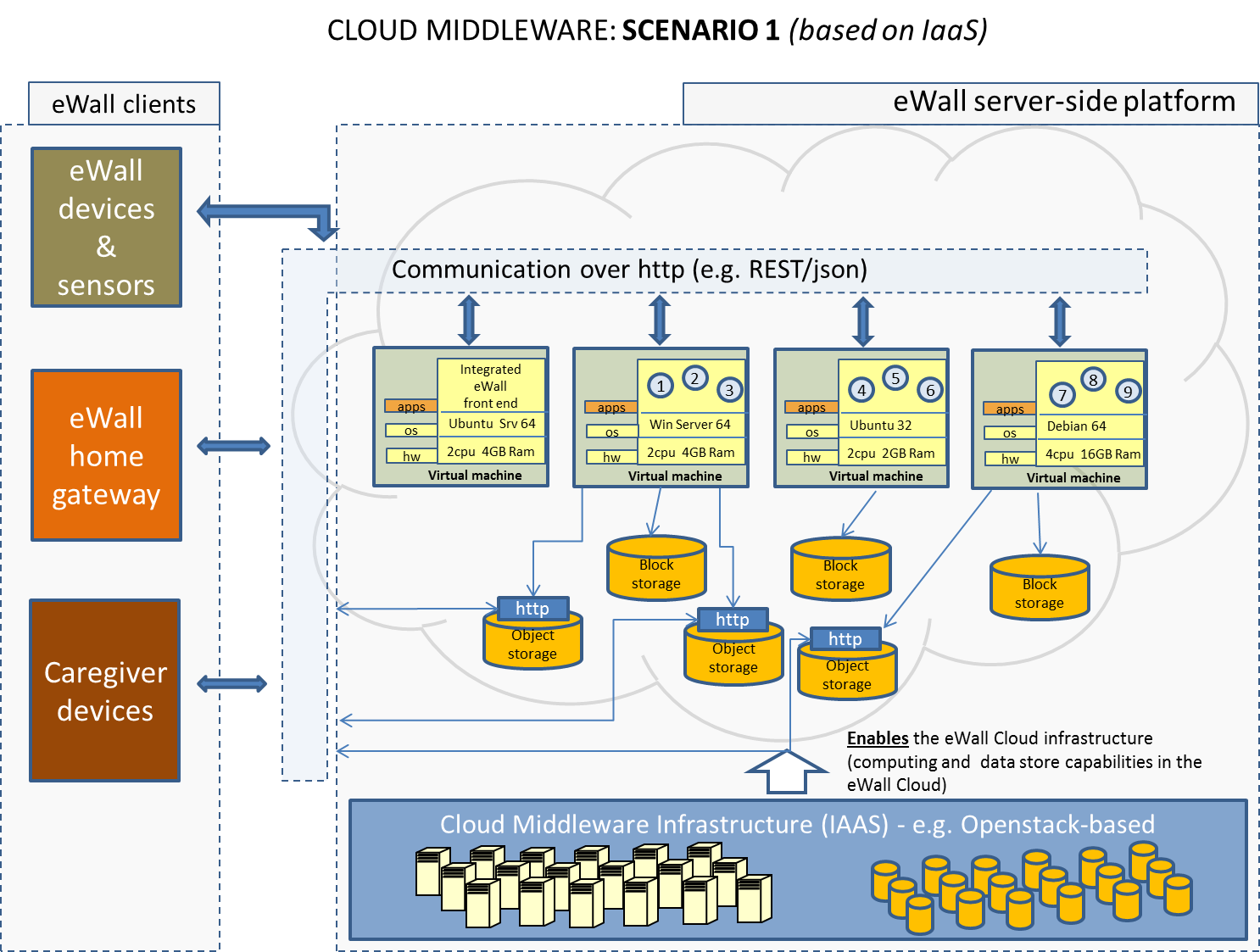
In the mid layer, the OpenStack components described at the beginning of this section are represented.

In the upper layer, a sample representation of a set of virtual resources that can be run on the infrastructure is provided: a set of virtual machines *v1*, *v2*, *vn* are running in the cloud. They all share a private network which allows communication only within the cloud subnetwork *private vnet 1*, and a public network *public vnet 2* which allows to interact with other networks (internal or external to the cloud). Some of the virtual machines have a dedicated block storage (*vm2-bs2*, *vmn-bs1*).



**Figure 2: eWALL OpenStack IaaS - single node deployment**

A more concrete representation of how the cloud resources are used in the whole eWALL project is provided in Figure 3 (also taken from [1]). The virtual machines are represented with their virtual hardware, operating system and with the functional services they provide. As an example, the leftmost virtual machine is providing an integrated front end to access the eWALL services via HTTP protocol, and consists of a 2CPU, 4GB RAM virtual machine running the Ubuntu Server x64 operating system. Such virtual resources are provisioned, together with the related block storage, by the underlying eWALL IaaS, thanks to the OpenStack services installed on the physical resources available.



**Figure 3: IaaS model in eWALL**

## Access to the eWALL cloud resources

Both for security and resource limitation reasons, the eWALL cloud exposes to the outer world (on the public internet) only a selected set of endpoints, namely the endpoints which are invoked for the provisioning of the eWALL services, over the HTTP(S) protocol.

In addition to such endpoints, the platform also provides means to developers and system administrators to access the resources for performing deployment of new components, configurations, maintenance activities, etc.

Therefore, the eWALL cloud can be accessed at three different levels:

* HTTP access to the management dashboard.
* Direct access to virtual machines via SSH protocol.
* Direct HTTP(S) access to service endpoints.

Access at levels 1 and 2 requires an active connection via SSH tunnelling towards the server hosted by UPB. To get this access working, it is necessary to have a valid user account on the physical server and a valid user account on the cloud infrastructure.

Access at level 3 is publicly available at URLs directly reachable from the internet. This type of access is used by the eWALL components for inter-communication and from the end users’ devices for accessing the eWALL services.

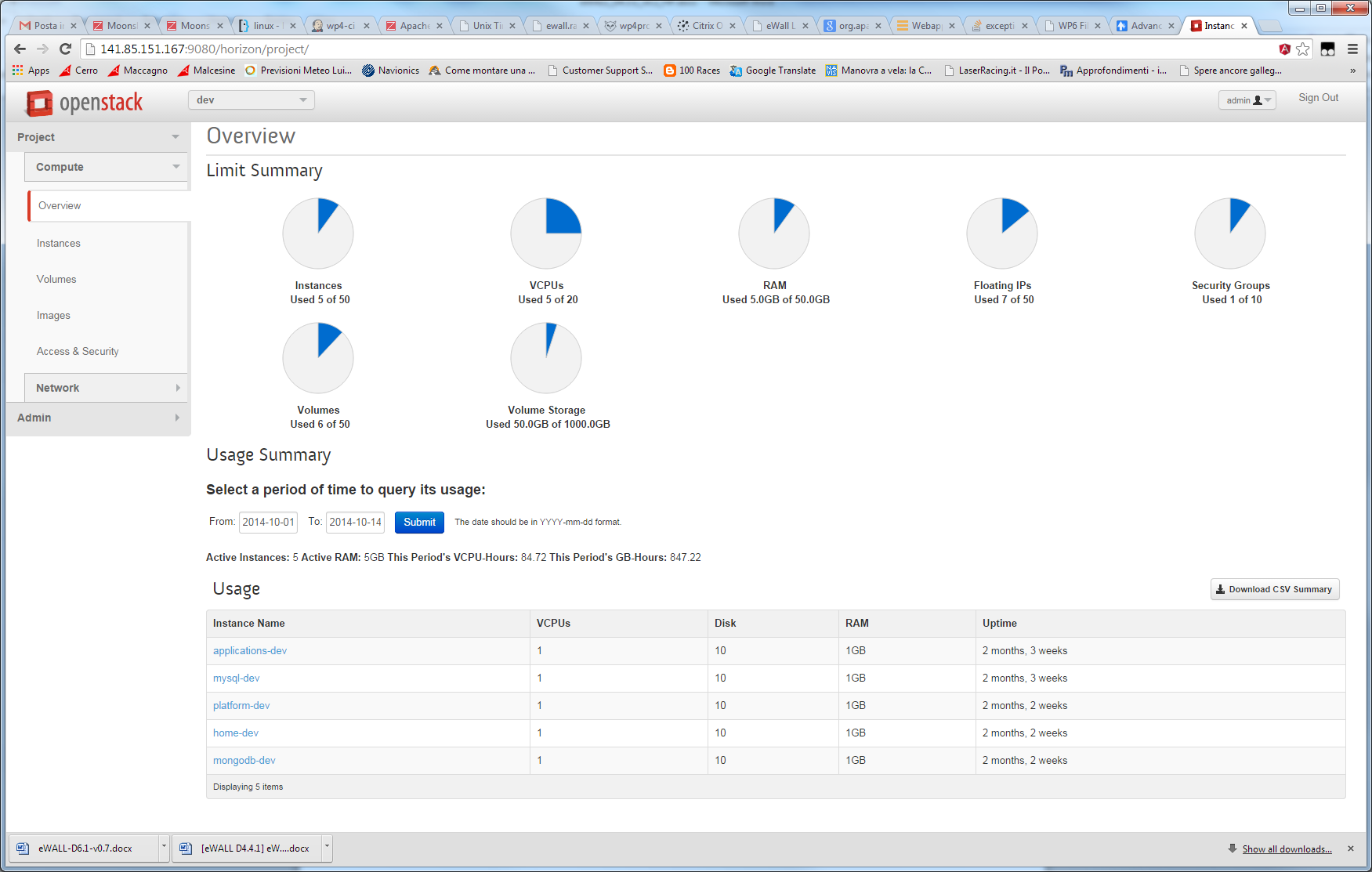
### HTTP access to the eWALL cloud management dashboard

OpenStack provides a web-based management dashboard named Horizon, which allows users to monitor the environments and perform actions based on their role. The eWALL Horizon dashboard is available to authorised users at the url:

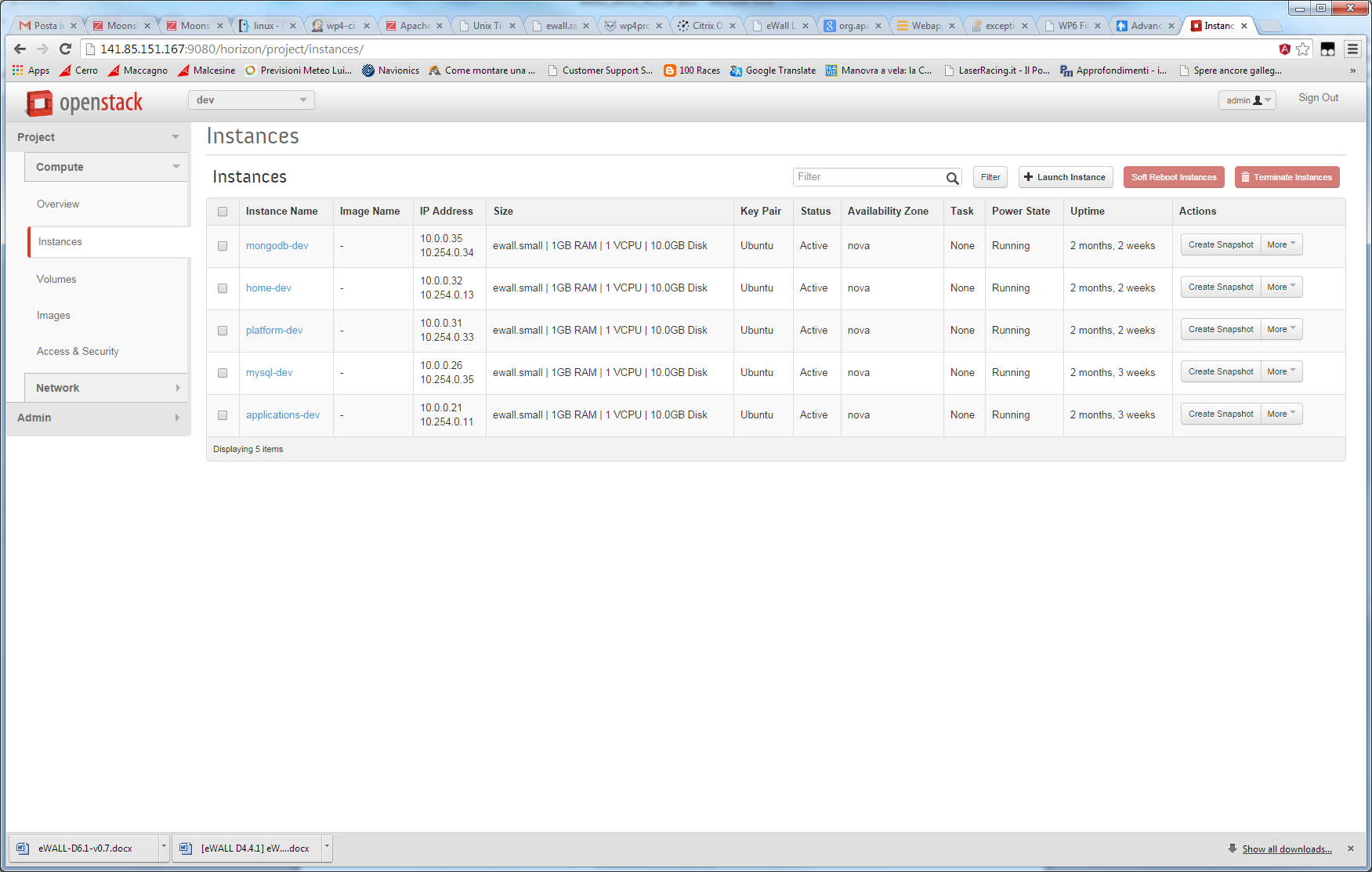
*http://141.85.151.167:9080/horizon*

The Horizon dashboard allows a lot of management operations on the cloud resources. Some examples are shown in the following pictures.

Figure 4 represents the resource overview of one of the environments (the development environment) that have been set up for eWALL.

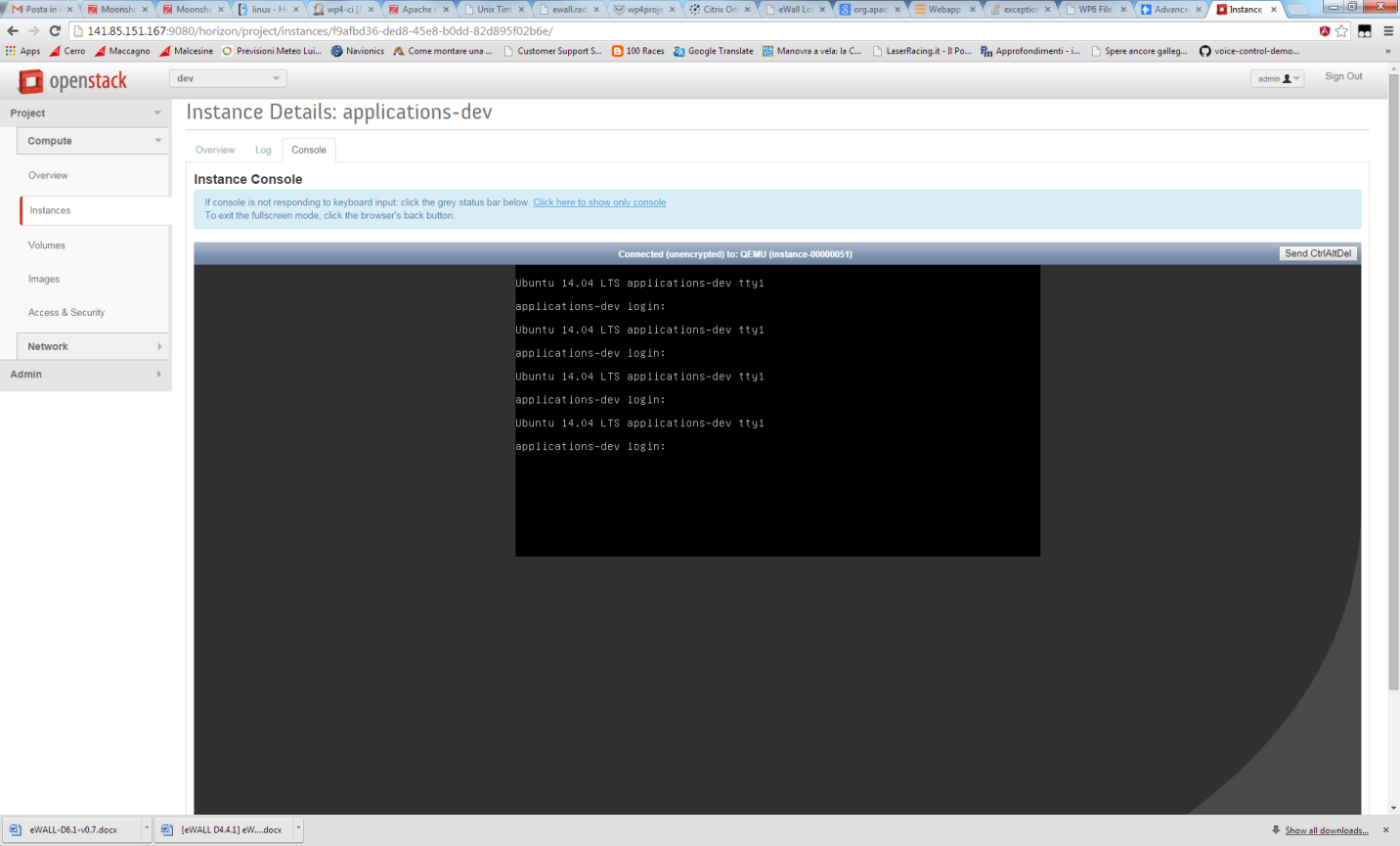


**Figure 4: eWALL Horizon web dashboard (resource view)**



**Figure 5: eWALL Horizon web dashboard (instance view)**

Figure 5 shows the instance view, a listing of the set of virtual machines instances running in the development environment, and providing management functionalities such as creating snapshots, starting new virtual machines, stopping, rebooting or deleting running instances, etc.



**Figure 6: eWALL Horizon web dashboard (access to VM console)**

Figure 6 shows the console view, which allows to connect via ssh to the terminal of a virtual machine from within the web dashboard.

### SSH direct access to the eWALL cloud virtual machines

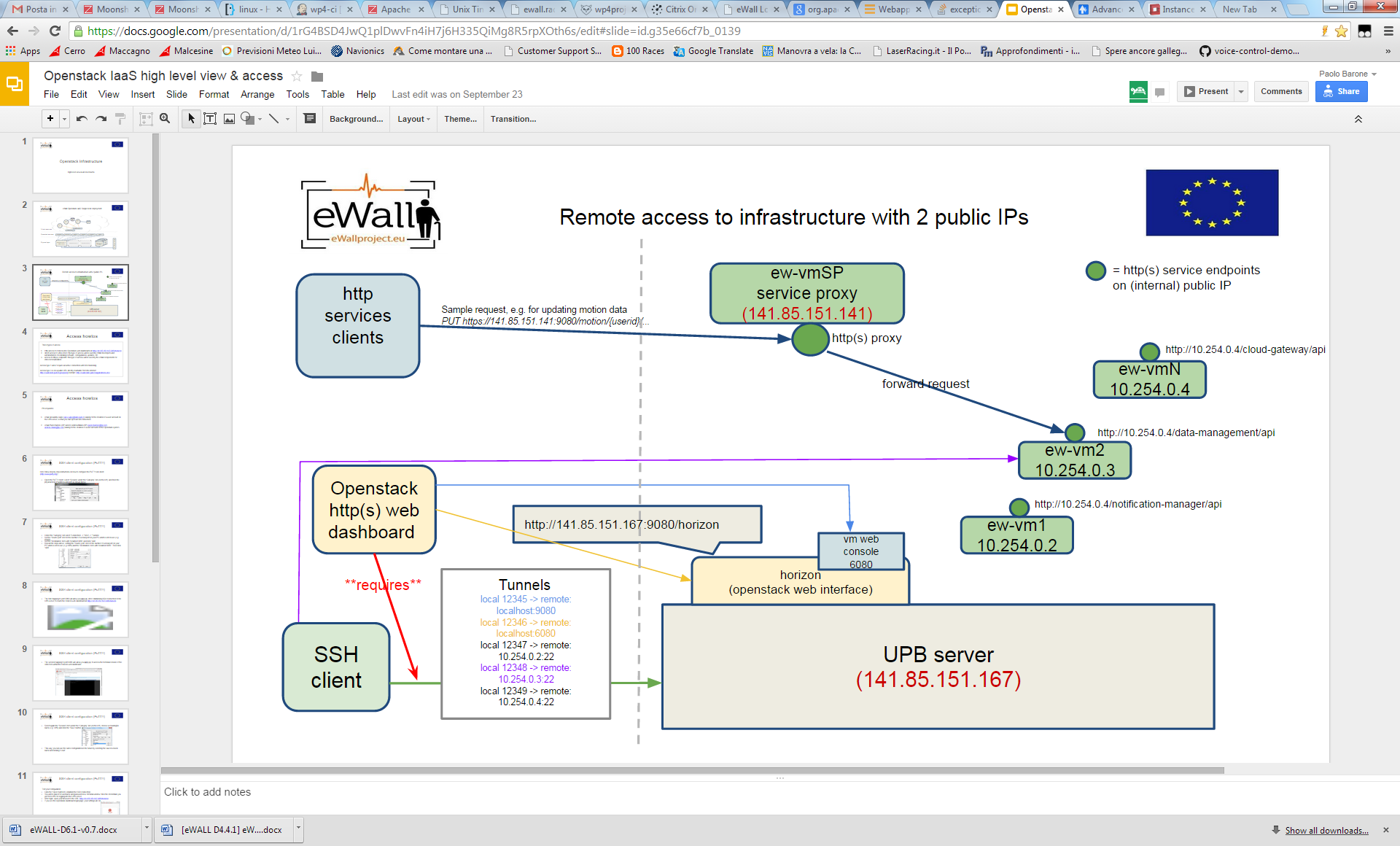
This type of access is used by eWALL developers and administrators for installing software, handling configurations, managing updates, etc.

Due to resource limitations (limited number of public IP addresses available in the hosting environment), the whole infrastructure was set up leveraging only two public IP addresses.

One of the public IP address is assigned to the UPB physical server. The second public IP address is assigned to a dedicated virtual machine, hosted in an administrative section of the eWALL cloud that was appositely created (the Application Lifecycle Management area), which acts as a web-proxy and enables the exposure of the HTTP(S) services on the internet as described in section 2.3.3.

Having only one IP address available for accessing the cloud infrastructure, we set up a specific network configuration on the eWALL environment which allows to perform tunnelling from the IP address of the physical server to the virtual IPs of the virtual machines hosted in the eWALL cloud.

**Figure 7** provides a high level overview of how this tunneling can be used from an SSH client to log into a vitual machines, and how an HTTP client can access the eWALL services provisioned by the different virtual machines through the web-proxy.



**Figure 7: Remote access to infrastructure with only 2 public IPs**

### HTTP(S) access to the eWALL services endpoints

The HTTP services deployed on the VMs, as already mentioned, are made available to the outer world via a dedicated VM which acts as a web-proxy. It forwards back and forward the HTTP requests mapping one single address to different endpoint URLs.

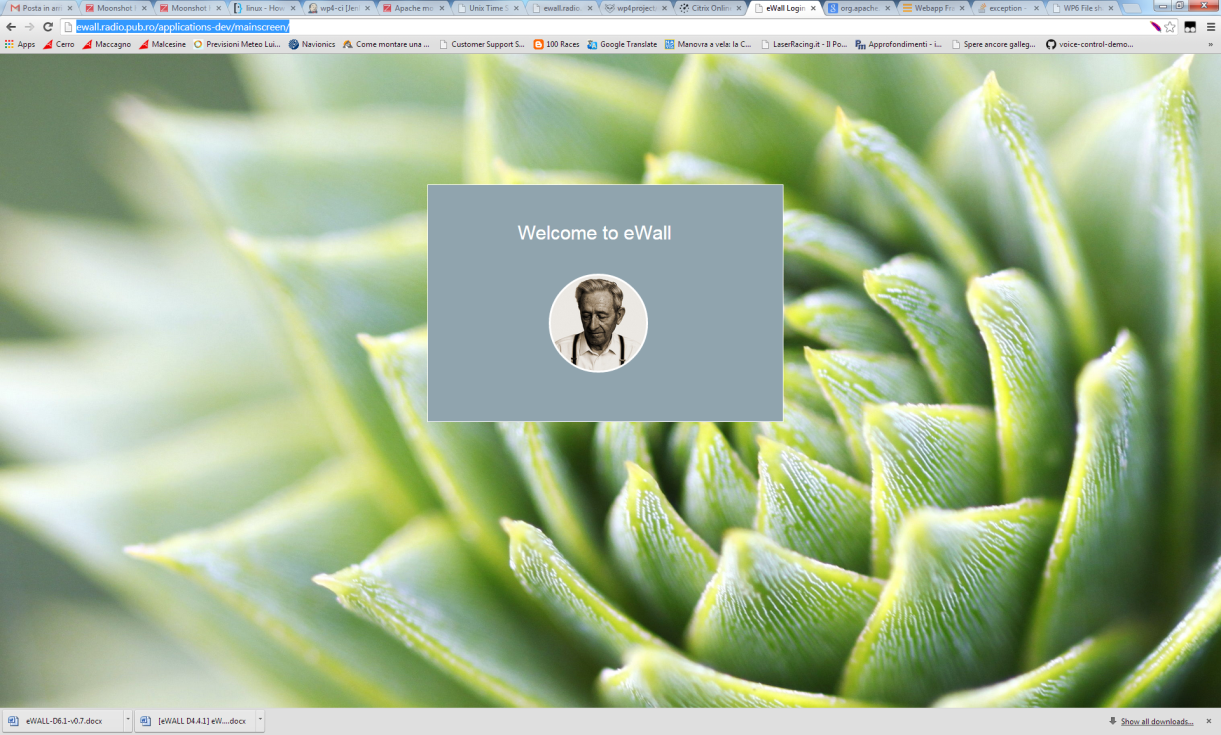
For this to happen, we have set up and configured a local DNS and a proxy server, which allow to reach the HTTP services at URLs with the following format:

*http://ewall.radio.pub.ro//[vm-name]/[service\_endpoint]*

As an example, the current prototype GUI for the end user from the development environment can be retrieved from:

*http://ewall.radio.pub.ro/applications-dev/mainscreen/*

getting the page shown in Figure 8.

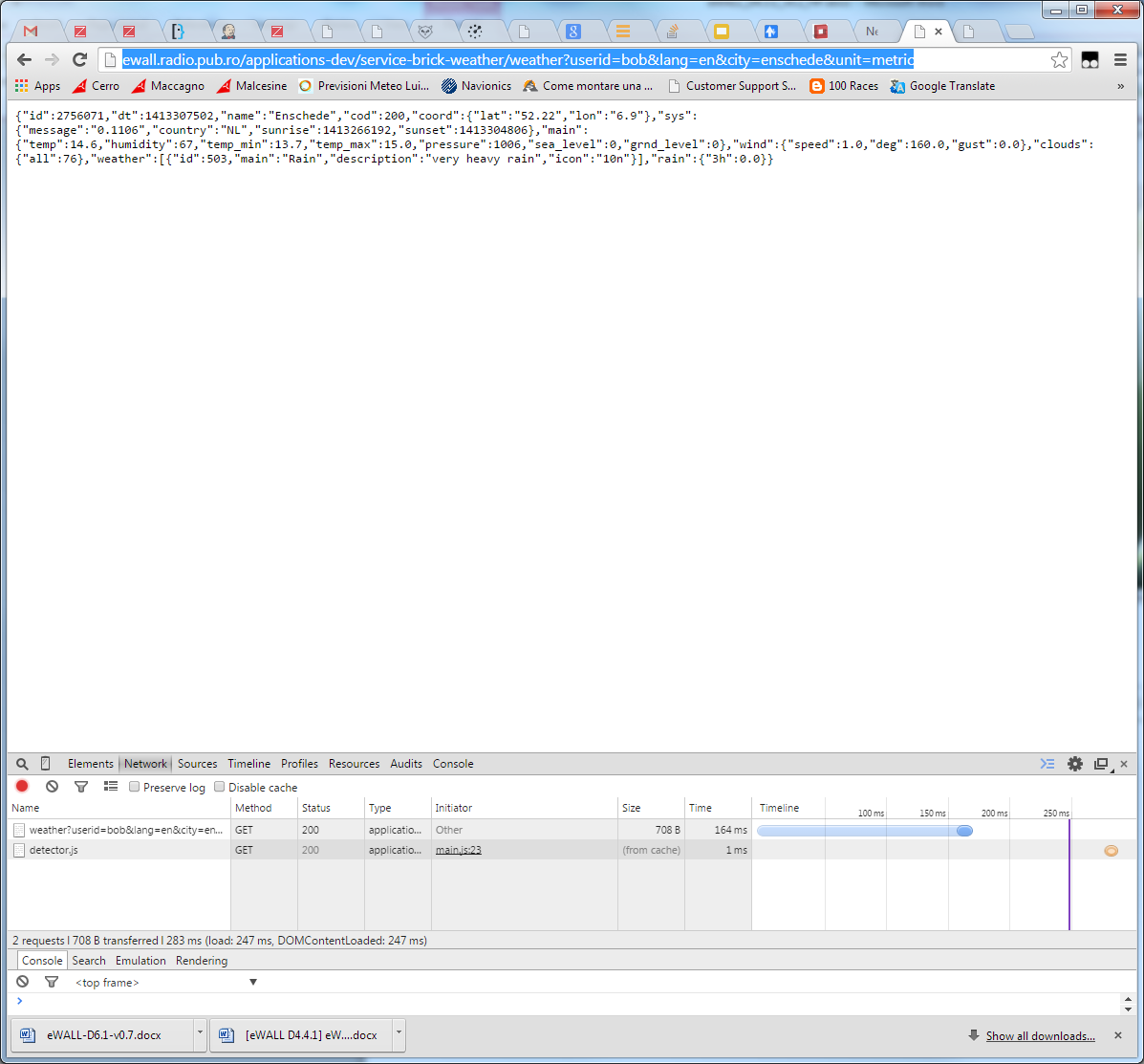


**Figure 8: Example (1) of a service endpoint publicly available on the internet**

Or, as an example of more specific service endpoints, you can get data from the weather service brick deployed on the “application-dev” VM in the “dev” environment at the URL:

*http://ewall.radio.pub.ro/applications-dev/service-brick-weather/weather?userid=janesmith@ewall.eu*

getting the text (JSON format) shown in **Figure 9**, which can be used by other eWALL services to compose an application.



**Figure 9: Example (2) of a service endpoint publicly available on the internet**

### Cache services

To improve the performance of the services deployed on the virtual machines running on the eWALL Cloud Infrastructure, a caching mechanism has been selected and included in the platform. Components that perform a lot of data retrieval operations on behalf of a client, when possible, can leverage the cache services to save temporarily response data, so that if the same data is needed by a following request it is not necessary to retrieve them again. The caching service selected for the eWALL infrastructure is based on Memcached[[2]](#footnote-3), an open source, high-performance memory object caching system particularly suitable for web based applications. Details on how it has been used in eWALL are provided in deliverable D4.4.3 [2]

### Monitoring and alerting services

The Cloud Infrastructure Layer and the set of virtual machines running on it constitute an ecosystem of services which have many interdependencies. To work effectively, it is mandatory to maximize the uptime of the virtual machines, and to verify the connectivity and responsiveness of the services provided. Therefore, the ecosystem requires continuous monitoring for verifying that all the interdependencies are statisfied, and automated alerting to notify responsible actors about issues. For this purpose, a monitoring and alerting infrastructure based on the open source Zabbix[[3]](#footnote-4) solution has been installed and configured. Details on the specific tasks configured in Zabbix for eWALL are described in D4.4.3 [2].

### Object storage service

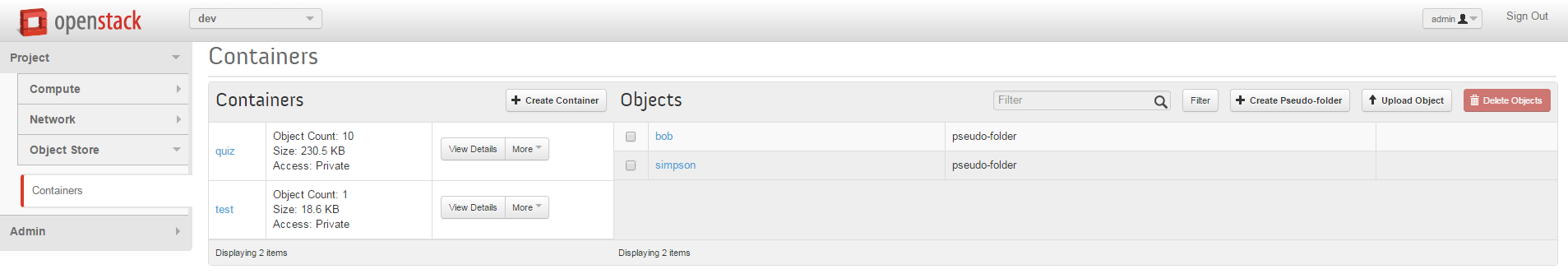
Object storage, already introduced in Section 2.2, is *“a storage architecture that manages data as objects, as opposed to other storage architectures like file systems which manage data as a file hierarchy and block storage which manages data as blocks within sectors and tracks. Each object typically includes the data itself, a variable amount of metadata, and a globally unique identifier. Object storage can be implemented at multiple levels, including the device level (object storage device), the system level, and the interface level. In each case, object storage seeks to enable capabilities not addressed by other storage architectures, like interfaces that can be directly programmable by the application, a namespace that can span multiple instances of physical hardware, and data management functions like data replication and data distribution at object-level granularity”*[[4]](#footnote-5).

For the eWALL platform, an Object Storage service is needed to fulfil the requirements of applications and services that must manage personalized multimedia contents (images, videos, etc.) in a flexible and reliable manner, to enable enhanced content personalization features and adequate level of privacy.

The Openstack platform provides and implementation of an Object Storage service called Swift, *“an open source software for creating redundant, scalable data storage using clusters of standardized servers to store petabytes of accessible data. It is a long-term storage system for large amounts of static data that can be retrieved, leveraged, and updated. […] Object Storage is ideal for cost effective, scale-out storage. It provides a fully distributed, API-accessible storage platform that can be integrated directly into applications or used for backup, archiving, and data retention.”*[[5]](#footnote-6)

**Figure 10** shows how the Openstack Object Storage can be accessed via the Horizon console. Within the Object Storage, every tenant can create so called “Containers”, which are http accessible storage area where users with proper permissions can “drop” generic contents. Every item which is put in that area, can be later accessed via an http URL. This mechanism is the same as the one provided by popular remote storage systems like Dropbox[[6]](#footnote-7) or Google Drive[[7]](#footnote-8).

Having a similar service for eWALL in a private cloud environment is important, as it allows to have full control on where the data are, and on the access level to grant to every user of the system.



**Figure 10: Openstack Object Storage view from the Horizon console**

The object storage service is also active in the eWALL prod environment based on Amazon Web Services (see Section 4.4), with the same principles being applied. The only difference is that, instead of being based on OpenStack Swift, it is based on Elastic Block Storage[[8]](#footnote-9)

## Migrating OpenStack deployment to Amazon Web Services

In order to get past the limitation of having one single physical server and to ensure redundancy and reliability to the eWALL production environment, the decision was taken to migrate the *prod* environment to a cloud hosting provider. Following an internal survey, Amazon Web Services (AWS) was selected as the cloud provider for the eWALL prod environment. AWS has been found by the EC to be fully compliant with EU Data Protection Directives[[9]](#footnote-10) therefore satisfying all the requirements for sensitive medical data.

In order to migrate the existing OpenStack VMs to AWS, and therefore preserve the already installed environment, which has taken time to set into place, several major steps had to be taken:

* Convert raw image to OVA format to be imported in Amazon EC2
  + Convert image to VirtualBox vdi disk file
  + Create VirtualBox VM from vdi file
  + Export VirtualBox VM to OVA format
* Upload OVA file to Amazon S3 storage bucket
* Import VM to Amazon EC2
* Configuring VMs for the new environment

Brief descriptions will be provided next for each of the steps



### Converting OpenStack image to OVA format

Unfortunately, OpenStack does not provide a direct way of exporting VM image to an OVA format, suitable for uploading to AWS EC2. Therefore, an intermediate conversion to VirtualBox VM was deemed necessary.

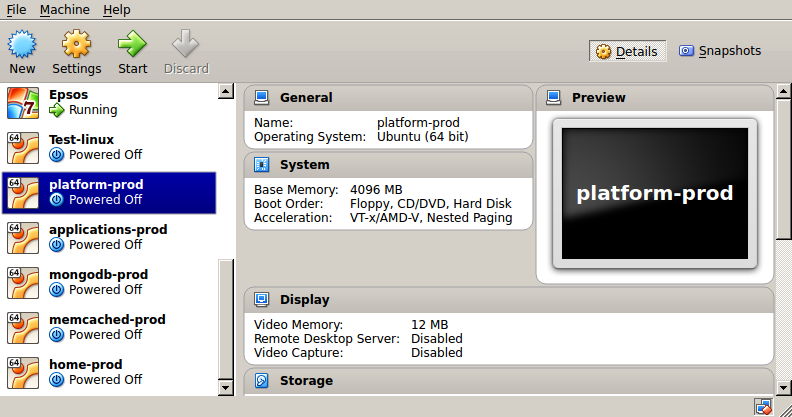
1. The OpenStack image was converted to VirtualBox[[10]](#footnote-11) virtual disk file, using the qemu-img command

$ qemu-img convert -f raw –O vdi *imageId* *filename*.vdi

*imageId* – the image UUID obtained via the *glance image-list* command

*filename –* the name of the VirtualBox virtual disk file

1. The VDI file was imported in the VirtualBox using VirtualBox GUI (**Figure 11**). This was done by creating a new VirtualBox VM and using the VDI image created in step 1 as a virtual hard drive file.



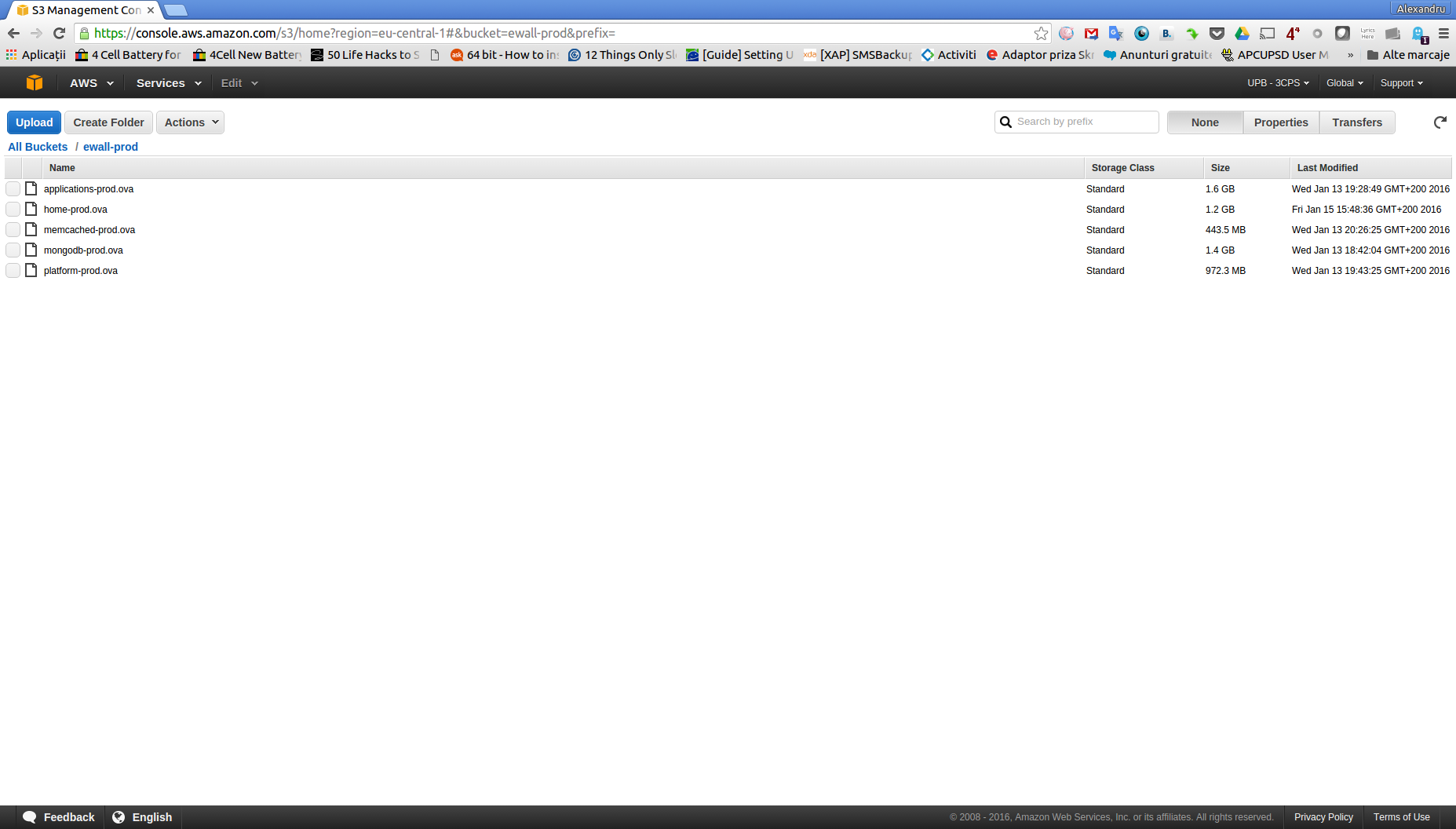
**Figure 11 VirtualBox VMs**

1. VirtualBox VMs were next exported to OVA format. This provides an OVF package as a single file archive with *.ova* extension. An OVF package is composed of metadata and file elements that describe virtual machines.

Prior to exporting, the disk files were compacted in order to save space occupied by the virtual disk, therefore enabling a quicker upload to Amazon S3 storage.

### Uploading OVA file to Amazon S3 storage

Amazon S3 is storage for the Internet. Amazon S3 provides APIs for creating buckets and uploading objects. The OVA files were uploaded via the S3 console in the *ewall-prod* bucket



### Import VM to Amazon EC2

In order to import the OVA image to Amazon EC2, the AWS Command Line Interface (CLI) tools were used. The tool accepts the Amazon S3 **bucket** and **path to the disk file** as a JSON representation. For instance, to upload *filename.ova* to ewall-prod bucket, the following example request was made:

$ **aws ec2 import-image --cli-input-json "{ \"Description\": \"Description of image\", \"DiskContainers\": [ { \"Description\": \"*Description of task*\", \"UserBucket\": { \"S3Bucket\": \"*ewall-prod*\", \"S3Key\" : \"*filename.ova*\" } } ]}"**

The request received a response in the form of a JSON representation, of which, the important parameter is the *taskId*. The *taskId* can be used to check the status of the import, using another command. For instance, for a taskId of *import-ami-fglsvl1f*, the following call has to be made

**$ aws ec2 describe-import-image-tasks --cli-input-json "{ \"ImportTaskIds\": [\"import-ami-fglsvl1f\"], \"NextToken\": \"abc\", \"MaxResults\": 10 } "**

* + 1. **Configuring VMs in AWS**

The configuration of the set of virtual machines in the *prod* environment in AWS is done in a similar fashion to the one depicted in **Figure 2** and **Figure 3**. Nevertheless, the main difference is that there is not only one physical server, but the resources are distributed in the AWS cloud. There is also one load balancer configured that automatically distributes incoming application traffic across Amazon EC2 instances in the cloud.

### Prerequisites

In order to use AWS CLI tools and import VMs to EC2, several prerequisites have to be met.

1. *Credentials*. Access to AWS resources can be given on a per-user basis. This is managed in the IAM (Identity and Access Management) console. The user that is given access receives an access key and a secret, that can be used to make AWS CLI requests. These can be configured using the following command

**$ aws configure**

1. *Roles*. An entity that is defined as a role contains a set of permissions for the user with that specific role. We defined a role named *vmimport* with a set of policies given in a separate file.

**$ aws iam create-role --role-name vmimport --assume-role-policy-document file://trust-policy.json**

1. *Policies.* A policy specifies actions that are permitted and the resource that the user is allowed to access. We created a set of policies that were defined in a separate file and assigned to the *vmimport* role defined previously.

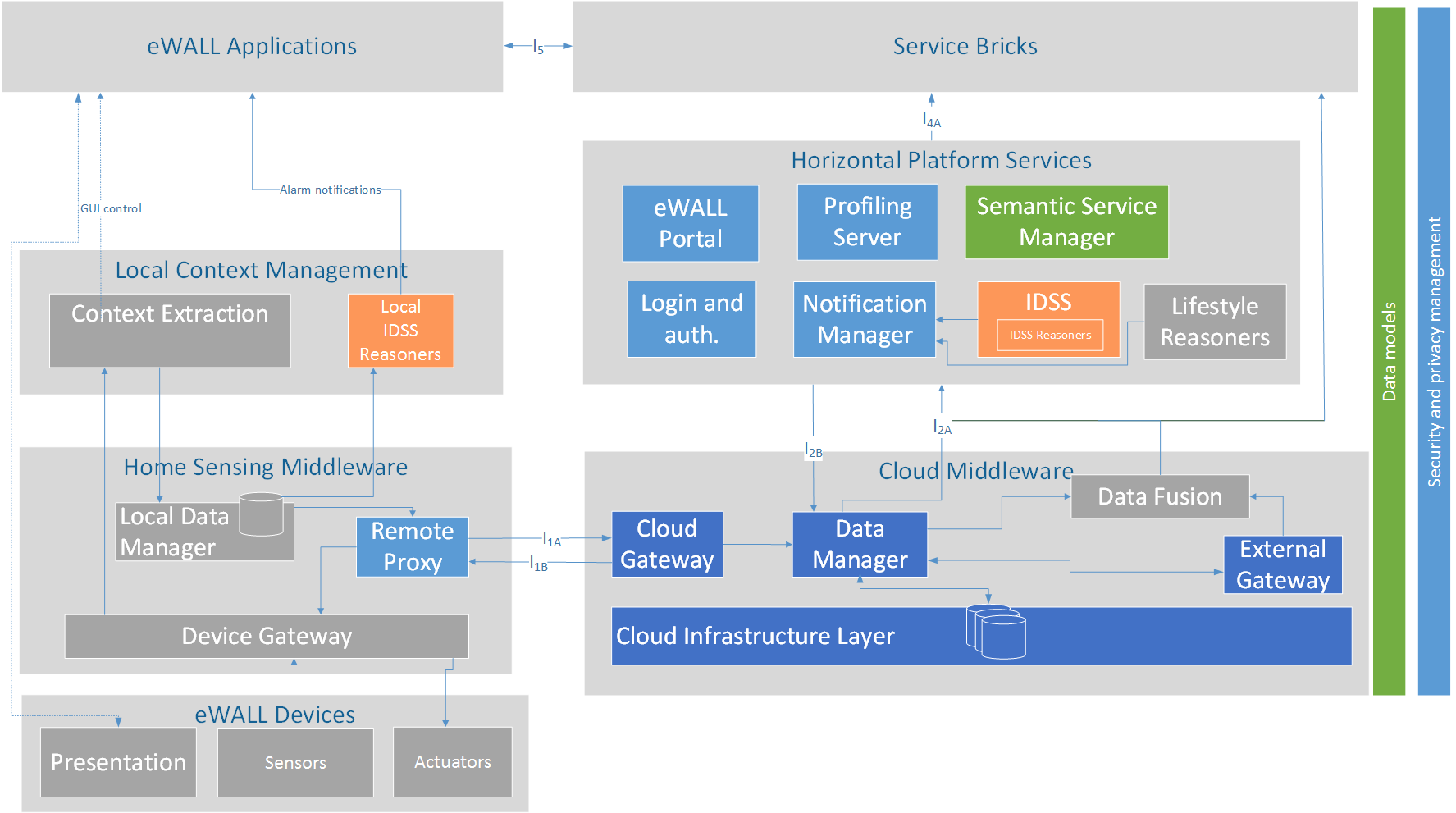
**$ aws iam put-role-policy --role-name vmimport --policy-name vmimport --policy-document file://role-policy.json**

# eWALL Platform within overall eWALL system architecture

The eWALL system involves an ecosystem of sensing devices, computing resources and display devices. The interaction within the elements of the ecosystem is enabled by the central component of the system, eWALL Platform with core running in cloud environment and which aggregates data coming from sensors, analyses them, transforms them into specific formats suitable for the user/system that needs them and infers new higher level knowledge by applying advanced reasoning on data. The objective of the ecosystem is providing added values services to the end users.

The purpose of this chapter is to describe eWALL Platform and its positioning within the overall eWALL system. In a broad sense of the word, the platform refers to as all developed software that is used as a base upon which all applications can run on top of any devices. However, the term eWALL Platform is also used as organizational unit of work under the responsibility of WP4. Based on this, we consider the eWALL Platform as a set of software artefacts developed within work on WP4 that are used as a base upon which other eWALL artefacts, namely devices and processing components from WP3 on one side, and personalized services and applications from WP5 on the other side, are implemented, integrated and deployed. The eWALL Platform provides core services to eWALL system which include secure and reliable communication between home and cloud environment, scalability, data management, security and privacy handling. Beside these core services, eWALL Platform also provides generic reusable set of services which ease the development of personalized services and applications, namely profiling, notification support, intelligent decision making, provisioning and configuration.

Based on eWALL multi-tier architecture described in eWALL Reference Model in D2.7 [2], the core functionality of eWALL Platform is positioned on Data Management and Business Services tiers. The Data Management tier provides global persistence services and various adapters that provide access, search and update services to databases and its data stored in a database management system. The Business Services tier provides enterprise-level services, and is also responsible for protecting the integrity of enterprise resources at the business logic level. Since eWALL Platform is also responsible for secure and reliable connection between home and cloud environment, the part of Data Sources tier, responsible for transfer of data to Data Management tier is also considered as part of eWALL Platform. The rest of Data Sources tier functionality, namely representing all sensors and gateways that aggregate and process data from sensing environment is related to the work of WP3.



**Figure 12: eWALL Platform and other WP4 components within the eWALL architecture**

The basic eWALL Platform block diagram is presented in Figure 12. Main artefacts of the eWALL Platform implemented are:

* Remote Proxy (T4.4) - responsible for connecting home sensing environment with eWALL Cloud,
* Local Data Manager (T4.4) - responsible for retrieving sensing data from local storage and sending this data to eWALL Cloud using Remote Proxy if allowed by local privacy policy,
* eWALL Portal (T4.4) - responsible for providing interface and support for administrative and technical management of the eWALL Platform,
* Login and authentication service (T4.4) - responsible for authenticating users/clients and to verify access and permissions
* Notification Manager (T4.4) - processing entity that is capable of managing (i.e. storing, forwarding or deleting) all eWALL notifications,
* Profiling Server (T4.4) - manages user profile information represented by user profile ontology,
* Data Manager (T4.3) - responsible for basic data processing, format exchange, and input validation of received data,
* Cloud Gateway (T4.3) - responsible for interconnecting eWALL Cloud with Sensing Environments (Home and Mobile),
* Intelligent Decision Support System (T4.2) - responsible for decision making activities in eWALL system,
* Data Model (T4.1) - represents data model for eWALL Platform.

The functionality of these artefacts is based on the functional architecture of eWALL system, that was presented in deliverable D2.7 [5]. Details about current version of RESTful APIs are presented in *Appendix A: eWALL Platform RESTful interfaces*

The above Figure 12 presents also how the work on the eWALL Platform was divided among tasks and partners on WP4. As mentioned earlier, this deliverable describes implementation and integration process that lead to development of the second version of eWALL Platform, and implementation of artifact that have been implemented within the work of this deliverable (illustrated light blue), namely Remote Proxy, eWALL Portal, Login and authentication service, Notification Manager, Profiling Server and Security and privacy management. The other artefacts have been implemented in the scope of other WP4 tasks and implementation is described in related deliverables; D4.1.3 Semantic model of eWALL middleware services [3], D4.2.3 Intelligent support system for eWALL [4](illustrated orange), and D4.3.3. Cloud middleware services for eWALL [2]**Error! Reference source not found.** (illustrated dark blue). Components illustrated grey are result of the work in WP3 and WP5.

## Data flows between home and cloud platform components

Basic eWALL system interaction flows that present most common interactions between different parts of eWALL system, such as system initialization flow, data push flow, data pull flow, notifications flow, are presented in D2.7 [5]This section describes in more details only data flows between home and cloud platform components. The flow presented on Figure 13 show basic flow of data at local platform components initialization.



**Figure 13 Data flow between home and cloud platform components at initialization**

First, Remote Proxy sends credentials to login to the system and obtain authentication token that is then used in all subsequent HTTP requests towards eWALL Cloud. Afterwards, Remote Proxy registers sensing environment with Point of Contact with details on local platform version, status, expiration, etc. Cloud Gateway checks whether sensing environment is provisioned and enabled in the (via eWALL Portal), and if it is not already online. If successfully, Remote Proxy obtains devices configuration data, such as devices types, room name etc. This information is then used by Local Data Manager to create subscriptions for receiving data per each device from local database. Next, Remote Proxy performs synchronization sync check in case there was an unexpected system shutdown of Home PC. For pushing data from cloud to home environment, eWALL uses a technique call HTTP long polling. This is started at the end of initialisation phase.



**Figure 14 Data push flow from local platform to cloud platform**

After initialization, local platform components are ready to start sending data to cloud and to receive commands from cloud. As presented on Figure 14, when new processed data from WP3 is stored in local database, the data is immediately send to Local Data Manager via database continuous changes API. LDM adapts this data to commons data model (D4.1.3), checks privacy policy and if allowed sends data to cloud via Profiling Server and Cloud Gateway as designed in D2.7 data push communication model. Prior to that, if there is a Local Reasoner (D4.2.3) subscribed to this data type, the data will be process by Local Reasoner. If alarm needs to be send to cloud Local Reasoner uses Remote Proxy and Cloud Gateway to send notification to Notification Manager.

For receiving commands from cloud Remote Proxy uses a technique called HTTP long polling. In principle, the Cloud Gateway holds the request from Remote Proxy open until new event occurs. Once event occurs, the Cloud Gateway responds and sends the new data in response. When the RP receives the new data, it immediately sends another request, and the operation is repeated. This is used as server push feature and it is used to send actuator commands from cloud to home environment, as well as for remote configuration of local platform via eWall Portal, as presented in diagram in Figure 15.



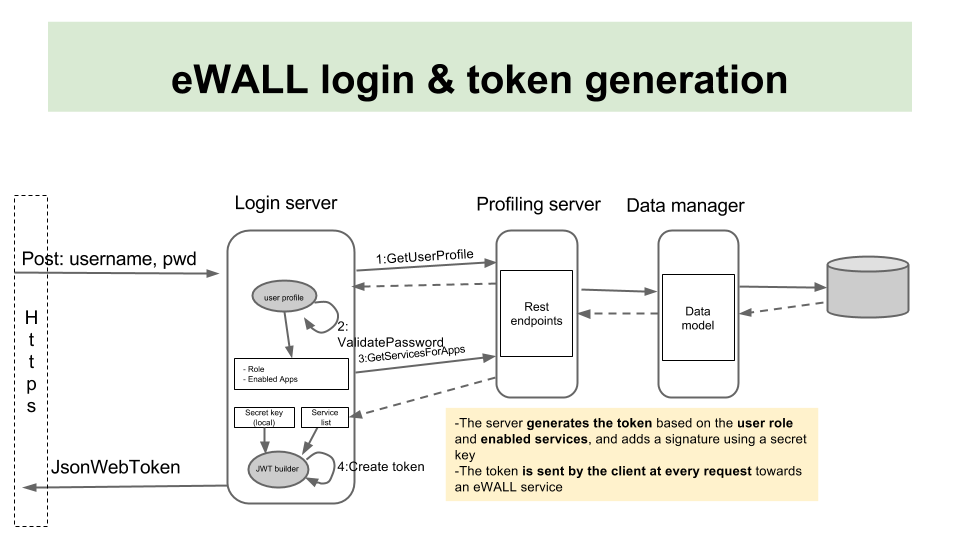
**Figure 15 Actuator control command and remote configuration data push from cloud to local platform**

# Authentication and Authorization framework

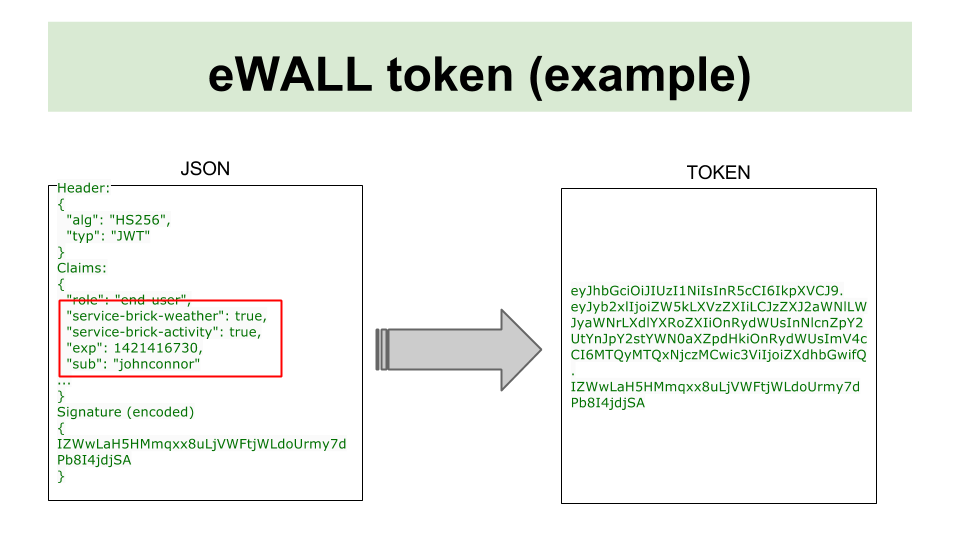
The eWALL platform is obtained by composition of a set of loosely coupled web services, talking each other via REST/HTTP protocol. This kind of architecture, which is very modular and allow for a high degree of flexibility in the service provisioning, requires special handling for what concerns authentication and authorization.

In traditional three-tiers, web-based systems, where services are provisioned by one (or a cluster of) front-end server, the server manages subsequent requests coming from the same client by creating, after authentication (e.g. via credentials), a “session” object. Such object allows the server to maintain a status between calls, and to share with the client a temporary identifier (session id) whose lifetime lasts until explicit logout or time expiration.

In the case of eWALL, the platform services are exposed by a set of different servers, independent from each other (what we can call “micro services” architecture) and the consumers can invoke each of them directly. No temporary information such as session ids or status data are stored between following invocations; rather, every call is self-contained (stateless). In this scenario, the best approach for identifying whether a client can access a service leverages the so called “token-based” authentication and authorization mechanism. This means that, after a user/app logs in to a login service (providing credentials), the login server will generate and return to the client a token (encoded string), which must be transmitted by the client in every subsequent request to each service provider. The token is “self-contained”, meaning that it contains information stating which services the user is entitled to use and for how long. The login server, based on the credentials provided by the user and by retrieving his profile, detects which services the client can access and builds the token accordingly. After creation, the token content is signed using the token itself and a “secret” key, which is known by the login server and by the service providers. This token generation workflow is represented in Figure 16, while Figure 17 depicts the representation of a simple token in JSON format and in the correspondent encoded format.



**Figure 16: Token generation workflow**



**Figure 17: Example of a simple token, from the json representation to the encoded representation**

Every service provider will check, at every request, whether the token is valid and contains the proper permissions for using the service. Since the token is signed with a secret key, if a client tries to modify it (to get more permissions than allowed), the service provider will detect the modification and will deny access. The related workflow is represented in Figure 18.



**Figure 18: Token transmission form client to service provider, and token verification at the provider side**

Details on how the authentication and authorization mechanism has been implemented in eWALL are provided in D4.4.3 [1].

# eWALL Platform Implementation

This chapter describes the implementation of eWALL Platform artefacts. As already mentioned, the eWALL Platform is composed of artefacts that were developed in scope of this deliverable (which are in more detail presented in this chapter) but also of artefacts that were developed within other WP4 tasks (these are just briefly listed in this chapter with a reference to the related deliverable with more implementation details).

Each artefact has its own ID card (table) with links to source, documentation, deployment location, version, responsible partners and task, and black box description. Additionally, all T4.4 artefacts also have description of requirements and features.

All eWALL Platform cloud components are built as Web Application Archive (WAR file) and deployed and integrated at Apache Tomcat Web servers running on eWALL OpenStack IaaS (see D4.3.3 [2] for more details).

## Remote Proxy

|  |  |
| --- | --- |
| Artefact: Remote Proxy | |
| Maven artefact | http://serv2.radio.pub.ro/gitlab/wp4/wp4project/tree/stable/remote-proxy/pom.xml |
| Source (Git address) | http://serv2.radio.pub.ro/gitlab/wp4/wp4project/tree/stable/remote-proxy |
| Build (Jenkins) | http://ewall.radio.pub.ro/jenkins/job/wp4-release/eu.ewall.platform.remoteproxy$remote-proxy/ |
| Binaries (Nexus) | http://ewall.radio.pub.ro/nexus/content/repositories/releases/eu/ewall/platform/remoteproxy/remote-proxy/1.0.0/ |
| Documentation (Javadoc) | http://ewall.radio.pub.ro/jenkins/job/wp4-release/eu.ewall.platform.remoteproxy$remote-proxy/javadoc/ |
| Version | 1.0.0 |
| Responsible partner(s) | ENT, UPB |
| Related to task | T4.4 |

### Black box description

The Remote Proxy (RP) is responsible for connecting the Home Sensing Environment with the eWALL Cloud. It provides necessary interfaces for remote access and configuration of sensor and actuators in sensing environment, and is responsible for transmission of sensor data to the eWALL Cloud. It is responsible for sending processed and indexed measured sensor data, to receive control and configuration data to Sensing Environments and for bidirectional application data. It directly communicates with Cloud Gateway component from Cloud Middleware and supports both *message based* and *pull and push* communication. Currently, RP is implemented as standalone Java component that can run on any JVM.

### Requirements

List of related requirements from D2.7 [5]:

* Gen\_001: Flexibility - ability to support a variety of market available or eWALL developed user and network devices
* Gen\_002: Scalability - ability to easily scale the eWALL platform to all envisioned use-cases
* Gen\_003: Traceability - ability to log and track the taken actions throughout the platform operation
* Gen\_004: Extensibility - ability to easily integrate novel devices in the platform (transparent protocol formats and protocol messages)
* Gen\_005: Reliability - ability to provide reliable communication within the platform and always on-time reaction
* Gen\_006: Compatibility - ability to integrate various information from various devices in a user transparent manner
* Gen\_018: Maintainability and configurability - ability to easily maintain and configure the system after deployment
* Spec\_009: Remote accessibility – ability to provide remote access to the eWALL platform
* Spec\_023: Communication **–** ability to enable inter-component message-based (or event-based) and call-based communication between distributed components
* Fun\_003: Continuous reporting – ability to provision continuous communication of user data among all envisioned interfaces – continuous message communication (including communication to/from cloud environment)

### Features

Main features of Remote Proxy currently implemented are:

* Performs registration and configuration of Remote Proxy components and Sensing Environment at eWALL Cloud via REST interface.
* Sending point of contact information from Sensing Environment via REST interface that is used for reaching Sensing Environment from remote.
* Responsible for realization of communication protocol and adaptation, especially in regards to the quality, security, capacity and reliability of the communication between the home and cloud environments. Sending all measurements data coming from WP3 device components, including accelerometer data, temperature, humidity, luminance, movement, bed sensor etc. via REST interface and AMQP messages.
* Besides implemented RESTful interface for message exchange between the Sensing Environment and eWALL Cloud, a second communication mechanism, based on Advanced Message Queuing Protocol (AMQP) has been developed for high volume/critical sensor data upload. Registration and sending of measurements from WP3 devices is currently implemented. The measurements are sent to a RabbitMQ server in different queues (one for each measurement type) and with different routing keys. The routing keys will enable future differentiation between messages coming from different eWALL systems, sensing environments or devices. More about AMQP/RabbitMQ can be found in D4.3.2 [5].
* Responsible for adaptation of data being transferred between Sensing Environment and eWALL Cloud.
* Interfaces Device Gateway to request raw/unprocessed data directly from sensor and control actuators.
* Provides security features such as token based authentication and authorization, and encryption of sent data
* Support for basic remote host system monitoring

Features that are planned to be implemented in next cycle:

* Improved support for remote administration and monitoring (alarm generation)
* Support for subscription/notification based communication

### Local Data Manager

|  |  |
| --- | --- |
| Artefact: Local Data Manager | |
| Maven artefact | http://serv2.radio.pub.ro/gitlab/wp4/wp4project/blob/stable/local-data-manager/pom.xml |
| Source (Git address) | http://serv2.radio.pub.ro/gitlab/wp4/wp4project/tree/stable/local-data-manager |
| Build (Jenkins) | http://ewall.radio.pub.ro/jenkins/job/wp4-release/eu.ewall.platform.local.datamanager$local-data-manager/ |
| Binaries (Nexus) | http://ewall.radio.pub.ro/nexus/content/repositories/releases/eu/ewall/platform/local/datamanager/local-data-manager/1.0.0/ |
| Documentation (Javadoc) | http://ewall.radio.pub.ro/jenkins/job/wp4-release/eu.ewall.platform.local.datamanager$local-data-manager/javadoc/ |
| Version | 1.0.0 |
| Responsible partner(s) | ENT, UPB |
| Related to task | T4.4 |

### Black box description

The Local Data Manager (LDM) is a component that provides the trust and privacy support of the local data being processed or kept at the Home Sensing Environment (namely Home PC). The passing, receiving and managing the data products is handled by the LDM based on a set of ethical, privacy and security requirements and policies described addressing the requirements in D2.7 [5] and D2.4 The LDM will be mainly driven by the locally running eWALL sensor data processing outputs for local context extraction and the local IDSS reasoners, applications and services (see Figure 12). The communication of the LDM with the cloud will go through to the Remote Proxy component. The functionality also includes trust monitoring and key or certificate management as well as authentication and authorization for local running eWALL components.

### Requirements

List of related initial requirements from D2.7 [5]:

|  |  |
| --- | --- |
| Gen\_014 | **Privacy** – ability to keep personal information from being disclosed and shared with unauthorized parties |
| Gen\_020 | **User data separation** – ability to create pseudo identifiers for privacy protection supported by eWALL user and network devices |
| Gen\_022 | **Anonymity** – ability to switch off eWALL sensors and devices and manage the deletion of raw data from these sensors |
| Spec\_009 | **Remote accessibility** – ability to provide remote access to the eWALL platform |
| Spec\_010 | **Priorities management** – ability to handle different simultaneous requests and messages with different priority levels |
| Spec\_017 | **Identification, authentication and authorization** – ability of system components to identify, authenticate and authorize an entity (human users and other system components) that wants to use them before allowing them access to resources |
| Spec\_018 | **Confidentiality** - ability to maintain confidentially (the way in which the information disclosed or managed by the system is treated) of identifiable data, including controls on storage, handling, and sharing of data |
| Spec\_019 | **Integrity** – ability to detect data modifications and prevent unauthorized modifications, especially related to service user data, sensor data and commands sent to actuators. |
| Spec\_020 | **Non-repudiation** – ability to trace back every action on sensitive assets to the person or system component that performed it |
| Spec\_021 | **Auditing** – ability of a system to log all actions on sensitive assets, including failed access attempts |
| Spec\_022 | **Consent specification** – ability to provide a usable interface to capture the consent of the end-user about sharing data with services |
| Spec\_023 | **Communication –** ability to enable inter-component message-based (or event-based) and call-based communication between distributed components |

### Features

The LDM is built based on the security and privacy components, defined in D2. [6]:

* User Identification – Responsible for recognizing a valid user's identity
* Authentication – This component provides authentication for entitles which send inbound data or which consume data. Authentication is the process of verifying the claimed identity of a user.
* Authorization – Provides authorization for an entity (user or component) with specific role for what component or data are allowed to be sent or consumed. Basically, it checks that the given entity has the right permissions to access a certain component or data.
* Consent – It can be described as privacy management module that allows user to manage access to user related data to other entities (users)
* Encryption - Provides encryption/decryption for both data transmission (e.g. SSL) and stored data
* Audit logger - Provides functions for security audits.
* Configuration – Responsible for handling all security and privacy related configuration.

## eWALL Portal

|  |  |
| --- | --- |
| Artefact: eWALL Portal | |
| Maven artefact | http://serv2.radio.pub.ro/gitlab/wp4/wp4project/blob/stable/ewall-portal/pom.xml |
| Source (Git address) | http://serv2.radio.pub.ro/gitlab/wp4/wp4project/tree/stable/ewall-portal |
| Build (Jenkins) | http://ewall.radio.pub.ro/jenkins/job/wp4-release/eu.ewall.platform.portal$ewall-portal/ |
| Deploy | http://ewall.radio.pub.ro/platform-dev/ewall-portal/ |
| Documentation (Javadoc) | http://ewall.radio.pub.ro/nexus/content/repositories/releases/eu/ewall/platform/portal/ewall-portal/1.0.0/ |
| Version | 1.0.0 |
| Responsible partner(s) | HP, ENT |
| Related to task | T4.4 |

### Black box description

The eWALL Portal is responsible for providing interface and support for administrative and technical management of the eWALL Platform. It is designed for:

* Healthcare System Administrators: personnel belonging to or representing the healthcare system administrative staff, which manage the policies for the provisioning of the healthcare services to end users (e.g. for defining which caregivers must/can assist end users) in a specific geographical or administrative region
* Technical Administrators: personnel trained for managing the technical aspects of the system (e.g. technical configuration, initial addition of users and assignment of roles, etc.)

From the administrative or technical personnel’s perspective, the system allows them to manage policies or technical aspects of the platform. It will have a mechanism where:

* the responsible actors can be registered into the system and the relationships among them can be set
* the applications/services related to a specific user can be configured
* the sensing environment (set of sensing devices installed at a target end user’s home) can be associated to a specific user in the system
* the relevant events related to a specific user can be monitored

It will also have specific dashboards showing aggregated information, which depends on the role of the active user, e.g. related to system status for administrators, to target end user events for caregivers, etc. The ideal way to provide the above mentioned functionality is by means of a web portal, which allows access from everywhere in a modality which is independent from the client operating system used. If properly designed it might also offer access via mobile.

In the following we provide a description of the overall envisioned functionalities, which will be then defined at a finer level in terms of technical specifications.

### Requirements

List of related requirements from D2.7 **Error! Reference source not found.**:

* Gen\_001: Flexibility - ability to support a variety of market available or eWALL developed user and network devices
* Gen\_003: Traceability - ability to log and track the taken actions throughout the platform operation
* Gen\_006: Compatibility - ability to integrate various information from various devices in a user transparent manner
* Gen\_007: Responsiveness – ability to dynamically react and/or reconfigure eWALL platform elements
* Gen\_008: Multiuser capability – ability to support multiple eWALL users with guaranteed profiling
* Gen\_011: Security – ability to secure the eWALL users’ data from obtrusive and accidental eavesdropping
* Gen\_018: Maintainability and configurability - ability to easily maintain and configure the system after deployment
* Spec\_009: Remote accessibility – ability to provide remote access to the eWALL platform
  + 1. **Features**

Currently implemented features:

* Basic Support for eWALL system provisioning of sensing environments, users, devices, sensors and actors via REST interface
* Web based access for provision of sensing environments and associated primary users
* Creation and management of “Regions”: The eWALL system will be adopted in different Countries, each one with specific regulations and policies with regard to the healthcare services provisioning. The portal provides means for creating so-called “Regions”, each one representing a healthcare system specific to a geographical or administrative area. Such kind of initial, top level configuration is made by a Super User, defined at the time of the deployment of the platform by the eWALL technical staff.
* Region Administration: Every Region needs administration; therefore, within each Region the Portal makes possible to add users with administrative roles. The Region Administration section of the portal provides means for creating and managing (add/delete/modify) actors with the “region administrator” role for a specific healthcare system and creating and managing (add/delete/modify) actors with the “technical administrator” role for a specific healthcare system
* End User registration: Every end user (both target end users and caregivers) must be properly registered into the system by setting adequate profile information, and the relationship among them (e.g. “who are the caregivers who are assisting a specific target end user?”) must also be defined. In fact, all the back-end components which perform behavioural patterns analysis, messaging, notification triggering, etc. must know information about specific users and their profile, and also the relationship between the caregivers and the end users, to know e.g. who must be notified about events, when needed. Therefore, the Portal provides the following functionalities:
  + Creating and managing (add/delete/modify) new target users
  + Creating and managing (add/delete/modify) new caregivers (formal or informal)
  + Associating a target end user to one or more caregivers
  + Associating a target end user with a “sensing environment”. This step enables the platform to accept data coming from a remote home

In addition, for each user the Portal provides the management of profile data.

* Applications setting: every application interacts with a set of web service endpoints (“micro services”), each one providing specific functionality or data retrieval. The Portal allows, for each application, to configure information like the name and the endpoint for access and, most importantly, to specify the exact set of back end services required. This information is crucial because enables the token-based authentication and authorization mechanism implemented in eWALL (c.f. sections 5.5 and 5.6).
* Services setting: the Portal allows to configure the back end services deployed on the system, by specifying name and endpoint.
* Application assignment: For each target end user, a responsible actor must be able to define the set of applications that the user is entitled to use. Therefore, for each user, there is a related area in the portal where all the required configuration and settings are filled in and saved into the central platform. This area, which is accessed from the user profile page, allows:
  + Visualization of the list of applications available in the platform,
  + For each target end user, visualization of the list of the applications in the platform that can be assigned to him (e.g. because he/she has a specific set of sensors in the home),
  + Assignment of applications to an end-user.
* Event monitoring and history: Relevant events must be tracked within the platform and the actors in the system must be able to monitor them. Depending on their role, they might be able to see different subsets of such data. While the “event tracking” is dedicated to visualization of instant (recent) data, all the events history should be saved and made available for later access. This feature is under evaluation and will be considered for the next project cycle.

## Notification Manager

|  |  |
| --- | --- |
| Artefact: Notification Manager | |
| Maven artefact | http://serv2.radio.pub.ro/gitlab/wp4/wp4project/blob/stable/notification-manager/pom.xml |
| Source (Git address) | http://serv2.radio.pub.ro/gitlab/wp4/wp4project/tree/stable/notification-manager |
| Build (Jenkins) | http://ewall.radio.pub.ro/jenkins/job/wp4-release/eu.ewall.platform.notificationmanager$notification-manager/ |
| Binaries (Nexus) | http://ewall.radio.pub.ro/nexus/content/repositories/releases/eu/ewall/platform/notification-manager/1.0.0/ |
| Documentation (Javadoc) | http://ewall.radio.pub.ro/jenkins/job/wp4-release/eu.ewall.platform.notificationmanager$notification-manager/javadoc/ |
| Documentation (Swagger) | http://ewall.radio.pub.ro/platform-dev/notification-manager/index.html |
| Version | 1.0.0 |
| Responsible partner(s) | UKIM |
| Related to task | T4.4 |

### Black box description

The notification manager is a processing entity that is capable of managing (i.e. storing, forwarding or deleting) all eWALL notifications. It is a single central block where all notifications are being managed, so the user is not overloaded by all information coming randomly from independent notification sources. The main role of the notification manager is to manage the notifications (alarms) and messages that originate from a specific set of processing blocks, e.g. service bricks and reasoners.

### Requirements

The notification manager exhibits the following system requirements:

* Gen\_001: Flexibility - ability to support a variety of market available or eWALL developed user and network devices
* Gen\_002: Scalability - ability to easily scale the eWALL platform to all envisioned use-cases
* Gen\_003: Traceability - ability to log and track the taken actions throughout the platform operation
* Gen\_004: Extensibility - ability to easily integrate novel devices in the platform (transparent protocol formats and protocol messages)
* Gen\_005: Reliability - ability to provide reliable communication within the platform and always on-time reaction
* Gen\_006: Compatibility - ability to integrate various information from various devices in a user transparent manner
* Gen\_018: Maintainability and configurability - ability to easily maintain and configure the system after deployment
* Spec\_009: Remote accessibility – ability to provide remote access to the eWALL platform
* Spec\_023: Communication **–** ability to enable inter-component message-based (or event-based) and call-based communication between distributed components
* Fun\_003: Continuous reporting – ability to provision continuous communication of user data among all envisioned interfaces – continuous message communication (including communication to/from cloud environment)
* Gen\_007: Responsiveness - ability to dynamically react and/or reconfigure eWALL platform elements
* Gen\_008: User mobility input: ability to track the position and the movement of users in-house (important for various eWALL services)
* Spec\_018: Confidentiality: ability to maintain confidentially (the way in which the information disclosed or managed by the system is treated) of identifiable data, including controls on storage, handling, and sharing of data

### Features

The Notification Manager acts as a communication gateway/interface that forwards all necessary notifications (alarms) and messages, which are triggered by the respective processing blocks, to the specific notification user interface(s) UI(s). The Notification Manager is designed to combine and prioritize notifications based on their nature origin and destination. For the highest priority notifications and alarms the Notification Manager sends the notification/alarm content via email to the respective caregivers and primary user. Moreover, it can decide to discard specific ones in case of possible information overload. Additionally, the Notification Manager is capable of sending the required information to a specific, i.e. target, UI device based on the user’s (patient’s) location. The Notification Manager also handles caregiver specific notifications that are fed to the eWALL caregiver application.

Currently implemented features:

* Receiving notifications from IDSS reasoners and distributing them towards the corresponding user interface.
* Prioritization of notifications (prioritized scheduling and sending of notifications) based on the priority coefficient carried in the notification message. *Comment: Higher priority coefficients reflect a lower overall notification priority.*
* Sending highest priority notifications/alarms via email. The recipients of these notifications are the primary user, informal and formal caregivers. The e-mail addresses of all recipients are gathered from the corresponding user profile via the profiling server.
* Sending notifications to a target UI device of interest based on the user’s location. The user location is acquired from the user’s measurement data via the profiling server. *Example: If the user is in the same room as the eWALL main screen (i.e. the living room), then the Notification Manager will send all notifications to it. If the user leaves the room (and has a mobile device with an installed notification UI), then the Notification Manager will send all future notifications to the hand held device. In this manner, the user can be always fed with important information regardless of its actual whereabouts.*
* Cross-component artefacts compatibility of the Notification Manager. (i.e. Token-based authentication and authorization compatible interfaces*,* component configuration compatibility, automatic REST API documentation generation compatibility).
* Management (storing and handling) of the notifications in the cloud data base
* Handling caregiver based notifications
* Interfacing and sending notifications with caregivers’ application
* Policy based reasoning for sending notifications (e.g. send only specific notifications to the underlying caregiver/user based on the caregiver’s/user’s preferences)

## Profiling Server

|  |  |
| --- | --- |
| Artefact: Profiling Server | |
| Maven artefact | http://serv2.radio.pub.ro/gitlab/wp4/wp4project/blob/stable/profiling-server/pom.xml |
| Source (Git address) | http://serv2.radio.pub.ro/gitlab/wp4/wp4project/tree/stable/profiling-server |
| Build (Jenkins) | http://ewall.radio.pub.ro/jenkins/job/wp4-release/eu.ewall.platform$profiling-server/ |
| Binaries (Nexus) | http://ewall.radio.pub.ro/nexus/content/repositories/releases/eu/ewall/platform/profiling-server/1.0.0/ |
| Documentation (Javadoc) | http://ewall.radio.pub.ro/jenkins/job/wp4-release/eu.ewall.platform$profiling-server/javadoc/ |
| Documentation (Swagger) | [http://ewall.radio.pub.ro/platform-dev/profiling-server/index.html](https://www.google.com/url?q=http://ewall.radio.pub.ro/platform-dev/profiling-server/index.html&usd=2&usg=ALhdy29I0XtTV5QIAIqrofhjTPPVHAKmwA) |
| Version | 1.0.0 |
| Responsible partner(s) | ENT |
| Related to task | T4.4 |

### Black box description

Profiling Server manages user profile information represented by user profile ontology, described in Profile ontology section within eWALL ontology and implemented within commons-data-model artefact, both described in eWALL deliverable D4.1.2 [3].

### Requirements

List of related requirements from D2.7 [5]:

* Gen\_001: Flexibility - ability to support a variety of market available or eWALL developed user and network devices
* Gen\_002: Scalability - ability to easily scale the eWALL platform to all envisioned use-cases
* Gen\_003: Traceability - ability to log and track the taken actions throughout the platform operation
* Gen\_004: Extensibility - ability to easily integrate novel devices in the platform (transparent protocol formats and protocol messages)
* Gen\_005: Reliability - ability to provide reliable communication within the platform and always on-time reaction
* Gen\_006: Compatibility - ability to integrate various information from various devices in a user transparent manner
* Gen\_018: Maintainability and configurability - ability to easily maintain and configure the system after deployment
* Spec\_023: Communication **–** ability to enable inter-component message-based (or event-based) and call-based communication between distributed components
* Gen\_007: Responsiveness - ability to dynamically react and/or reconfigure eWALL platform elements
* Gen\_008: User mobility input: ability to track the position and the movement of users in-house (important for various eWALL services)
* Spec\_018: Confidentiality: ability to maintain confidentially (the way in which the information disclosed or managed by the system is treated) of identifiable data, including controls on storage, handling, and sharing of data

### Features

User profile information is stored in central place, eWALL cloud database (MongoDB[[11]](#footnote-12)) and is being exposed by the Profiling Server via REST interface. Except the data contained within (user) profile ontology different user related measurements are also exposed. Alongside data retrieval minimal user information storing functionality is also exposed via REST interface as a start towards more profound provisioning.

Main currently implemented features are:

* Receiving requests on REST interface for storing, modifying, obtaining and deleting user profile information,
* Receiving requests on REST interface for storing, modifying, obtaining and deleting eWALL applications’ information,
* Receiving requests on REST interface for storing, modifying, obtaining and deleting eWALL services information,
* Receiving requests on REST interface for storing, modifying, obtaining and deleting user related measurements data (e.g. health, visual)
* Receiving requests on REST interface for storing, modifying, obtaining and deleting environm related measurements data (e.g. gases, humidity, illuminance, mattress pressure, temperature, movement)
* Receiving requests on REST interface for storing and deleting user credentials data needed for the authentication with the eWALL system. The passwords are hashed and stored to the separate part of the eWALL MongoDB database.
* Receiving requests on REST interface for checking if the given user is allowed/authenticated to access the eWALL system.
* Communicating with the eWALL cloud (MongoDB) database containing the data,
* Receiving requests on REST interface and storing of user profile information.

Features that are planned for implementation in the next period:

* More elaborate support for profile data (both storing and retrieving) in the sense making available different API calls etc.
* Connection with Data Manager component (described both this document and in D4.3.2 [4]) and clear separation of concerns.

## Login service

|  |  |
| --- | --- |
| Artefact: eWALL Platform Login | |
| Maven artefact | http://serv2.radio.pub.ro/gitlab/wp4/wp4project/blob/stable/ewall-platform-login/pom.xml |
| Source (Git address) | http://serv2.radio.pub.ro/gitlab/wp4/wp4project/tree/stable/ewall-platform-login |
| Build (Jenkins) | http://ewall.radio.pub.ro/jenkins/job/wp4-release/eu.ewall.platform$ewall-platform-login/ |
| Binaries (Nexus) | http://ewall.radio.pub.ro/nexus/content/repositories/releases/eu/ewall/platform/ewall-platform-login/1.0.0/ |
| Documentation (Javadoc) | http://ewall.radio.pub.ro/jenkins/job/wp4-release/eu.ewall.platform$ewall-platform-login/javadoc/ |
| Documentation (Swagger) | http://ewall.radio.pub.ro/platform-dev/ewall-platform-login/ |
| Maven artefact | 1.0.0/ |
| Responsible partner(s) | HP |
| Related to task | T4.3 |

### Black box description

The Login service allows clients to authenticate to the eWALL platform via credentials, and in case they are authenticated returns them a token, a self-contained data structure which describes the role and the permissions that a client has on the system. Such token must be transmitted by the client in every subsequent request in form of http header. Since the token represents the operations that the client is allowed to do on the system, it is also used by the applications to create/delete dynamically GUI elements. If the user profile is changed by an operator, e.g. adding or removing permissions for using an application, the token contents change accordingly. Based on the change, the GUI reacts by redefining the elements which allow to access applications.

### Requirements

List of related requirements from D2.7 [5]:

* Gen\_001: Flexibility – ability to support a variety of market available or eWALL developed user and network devices
* Gen\_002: Scalability – ability to easily scale the eWALL platform to all envisioned use-cases
* Gen\_003: Traceability - ability to log and track the taken actions throughout the platform operation
* Gen\_004: Extensibility – ability to easily integrate novel devices in the platform (transparent protocol formats and protocol messages)
* Gen\_006: Responsiveness – ability to dynamically react and/or reconfigure eWALL platform elements
* Gen\_007: Multiuser capability – ability to support multiple eWALL users with guaranteed profiling
* Gen\_011: Security – ability to secure the eWALL users’ data from obtrusive and accidental eavesdropping
* Gen\_015: Context information - ability to provide context information that is useful for services to adapt themselves according to the needs, preferences and situation of the user
* Gen\_016: Service orientation – ability of a system to ensure reusability and composability of services and service components
* Gen\_017: Semantic interoperability – ability to enable semantic interoperability between applications and services for ensuring the highest degree of decoupling (enables an open system and facilitates reuse of existing services and applications)
* Gen\_019: Multi-modal user interaction – ability to support multi-modal user interaction
* Spec\_009: Remote accessibility – ability to provide remote access to the eWALL platform
* Spec\_017: Identification, authentication and authorization – ability of system components to identify, authenticate and authorize an entity (human users and other system components) that wants to use them before allowing them access to resources

### Features

The eWALL Login service is the entry point for all the eWALL users to the eWALL platform. Every user which is entitled to access the platform is assigned a username and password. Before accessing an eWALL service, users must authenticate by providing their credentials toward the Login service. Then, the Login service checks for the validity of the credentials (existence of the user and matching with a hashed password). After this preliminary verification, the service retrieves the user profile from the Profiling Server and extracts from it the user role and the list of the applications which such user is authorized to access. This setting (definition of the set of applications per user) is made from the portal by an administrator when inserting a new user (or modifying an existing one) into the system. From the list of applications, the Login service retrieves the list of all the needed backend services. Finally, the Login service builds a “token”, a string containing a set of information which identify the end user and the set of services he/she can access. The token is compliant with the JSON Web Token (JWT) specification[[12]](#footnote-13), which defines a standard structure for representing claims to be transferred between two parties. The token contains information such as the username, the expiration date (the token has limited lifetime), the set of services accessible, and the user role. After creation, the token is encoded and signed using the content of the token itself and a secret key, which is known only by the login service and by the backend services which builds up the eWALL platform. The token is then sent back to the client over https. In all following requests towards every eWALL service endpoint, the client must provide in an http header the token obtained after login. The service providers will extract the token from the request and check it for validation before serving the request. The signing mechanism grants that the token, if modified by a client (to get more permissions than stated by the claims), will be considered invalid by the service providers. In fact, the change will be immediately detected, due to the mismatch between the signature in the token and the signature built for verification by the server by applying the secret key to the token contents received by the client.

Since the token has an expiration time, the Login service provides to clients an endpoint which allows them to renew an existing token before its expiration.

In summary, the Login service provides two REST endpoints for:

* Authenticating a client, taking as input username and password, and returning a token in case of successful authentication
* Renewing a valid token, taking as input a valid token and returning a new token with updated contents and a new expiration date

Thanks to the renewal mechanism, a client can maintain the token live for the time required by requesting a renewal before the expiration time. Moreover, since the token is dynamically generated based on the user profile, if a modification in the profile happens, at the next renewal the client can detect the change and react accordingly, e.g. by modifying an application GUI to reflect the current set of applications that a user can access.

## Cross-component functionalities

|  |  |
| --- | --- |
| Artefact: eWALL Common Libraries | |
| Maven artefact | http://serv2.radio.pub.ro/gitlab/wp4/wp4project/blob/stable/ewall-common-libs/pom.xml |
| Source (Git address) | http://serv2.radio.pub.ro/gitlab/wp4/wp4project/tree/stable/ewall-common-libs/ |
| Build (Jenkins) | http://ewall.radio.pub.ro/jenkins/job/wp4-stable/eu.ewall.platform$ewall-common-libs/ |
| Binaries (Nexus) | http://ewall.radio.pub.ro/nexus/content/repositories/releases/eu/ewall/application/ewall-common-libs/1.0.0/ |
| Documentation (Javadoc) | http://ewall.radio.pub.ro/jenkins/job/wp4-stable/eu.ewall.platform$ewall-common-libs/javadoc/ |
| Maven artefact | 1.0.0/ |
| Responsible partner(s) | HP |

### Black box description

The eWALL REST services are provisioned by software components which share the same high-level structure:

* They are based on java servlets that:
  + are packaged in wars
  + leverage the Spring framework, and in particular the Spring boot features
  + load configuration properties from files
* They expose REST endpoints which must be invoked by other components, hence need to provide clear API definition
* They must share the same authentication/authorization mechanism, which allows to detect if a request can be served depending on the requestor’s role and permissions
* They should share the same configuration and documentation generation mechanisms, so that they are easily maintainable and have a uniform documentation

Therefore, a set of cross-component functionalities have been implemented and released in form of libraries, packaged into a “ewall-common-libs” artefact, to be included in the eWALL components.

### Features

The “ewall-common-libs” artefact provides the following features:

* ***Component configuration***: a uniform way to configure components and services, based on a Spring framework ”@PropertySource” annotation which allows to load automatically and transparently a set of property files, without the need for a developer to specify anything else in the code.
* ***Token-based authentication and authorization verification***: Automatically adds a servlet filter to the service, so that every request received is checked against the auth token to allow/deny access. A developer can use this feature by just setting a specific Spring ”@Import” annotation for a JwtAuthConfig class of the library
* ***Token-based requests:*** Provides an *ewallRestClient* object to make authenticated requests towards internal components (micro-services interaction, e.g. to the Profiling Server) in a transparent way for the developers. A developer can use this feature by just setting a specific Spring ”@Import” annotation for a CommonConfig class of the library
* ***Memory object caching system:*** enable caching of http requests and related response on the eWALL Memcached server via annotation. Requires only to set a specific Spring ”@Import” annotation for a CacheConfig class of the library, on top of the methods which implement REST endpoints. All the communication with the Memcached server and the related connection management and data storage are transparent to the developers.
* ***Automatic documentation generation*** for REST API: adds configuration for the Swagger framework (a tool for generating documentation on REST endpoints and to test them interactively. By just adding a Spring ”@Import” annotation for a SwaggerConfig class of the library, web documentation for all the endpoints of the service are automatically generated at build time and deployed together with the service

## Artefacts developed within other WP4 tasks

### Data Manager

The Data Management (DM) is responsible for basic data processing, format exchange, and input validation of received data. It uses Cloud Infrastructure Layer to seamlessly access physical storage facilities (databases) for data persistency. More details about implementation of this artefact can be found in *D4.3.3 Cloud middleware services for eWALL* [2].

|  |  |
| --- | --- |
| Artefact: Data Manager (DM) | |
| Maven artefact | http://serv2.radio.pub.ro/gitlab/wp4/wp4project/blob/stable/data-manager/pom.xml |
| Source (Git address) | http://serv2.radio.pub.ro/gitlab/wp4/wp4project/tree/stable/data-manager |
| Build (Jenkins) | http://ewall.radio.pub.ro/jenkins/job/wp4-release/eu.ewall.platform.middleware.datamanager$data-manager/ |
| Binaries (Nexus) | http://ewall.radio.pub.ro/nexus/content/repositories/releases/eu/ewall/platform/middleware/datamanager/data-manager/1.0.0/ |
| Maven artefact | 1.0.0 |
| Responsible partner(s) | ENT, UPB |
| Related to task | T4.3 |

### Cloud Gateway

The Cloud Gateway (CGw) is responsible for interconnecting eWALL Cloud with Sensing Environments (Home and Mobile). It is responsible for receiving processed and indexed measured sensor data, to send control and configuration data to Sensing Environments and for bidirectional application data. It directly communicates with Remote Proxy and supports both message based pull and push communication. More details about implementation of this artefact can be found in *D4.3.2. Cloud middleware services for eWALL* [2].

|  |  |
| --- | --- |
| Artefact: Cloud Gateway (CGw) | |
| Maven artefact | http://serv2.radio.pub.ro/gitlab/wp4/wp4project/blob/stable/cloud-gateway/pom.xml |
| Source (Git address) | http://serv2.radio.pub.ro/gitlab/wp4/wp4project/tree/stable/cloud-gateway |
| Build (Jenkins) | http://ewall.radio.pub.ro/jenkins/job/wp4-release/eu.ewall.platform.middleware.cloudgateway$cloud-gateway/ |
| Binaries (Nexus) | http://ewall.radio.pub.ro/nexus/content/repositories/releases/eu/ewall/platform/middleware/cloudgateway/cloud-gateway/1.0.0/ |
| Documentation (Javadoc) | http://ewall.radio.pub.ro/jenkins/job/wp4-release/eu.ewall.platform.middleware.cloudgateway$cloud-gateway/javadoc/ |
| Documentation (Swagger) | http://ewall.radio.pub.ro/platform-dev/cloud-gateway/index.html |
| Version | 1.0.0 |
| Responsible partner(s) | ENT, UPB |
| Related to task | T4.3 |

### Intelligent Decision Support System

The Intelligent Decision Support System (IDDS) is responsible for decision making activities in eWALL system. It uses inference to discover new relationships, automatically analyses the content of the data, and manages knowledge (i.e. perform reasoning based on corresponding or appropriate rules). More details about implementation of this artefact can be found in *D4.2.3 Intelligent support system for eWALL* [4]**Error! Reference source not found.**.

|  |  |
| --- | --- |
| Artefact: Intelligent Decision Support System (IDSS) | |
| Maven artefact | http://serv2.radio.pub.ro/gitlab/wp4/wp4project/blob/master/idss-core/pom.xml |
| Source (Git address) | http://serv2.radio.pub.ro/gitlab/wp4/wp4project/tree/stable/idss-core |
| Build (Jenkins) | http://ewall.radio.pub.ro/jenkins/job/wp4-release/eu.ewall.platform.idss$idss-core/ |
| Binaries (Nexus) | http://ewall.radio.pub.ro/nexus/content/repositories/releases/eu/ewall/platform/idss/idss-core/1.0.0/ |
| Documentation (Javadoc) | http://ewall.radio.pub.ro/jenkins/job/wp4-release/eu.ewall.platform.idss$idss-core/javadoc/ |
| Version | 1.0.0 |
| Responsible partner(s) | RRD, UOZ |
| Related to task | T4.2 |

### Commons data model

The commons data model artefact represents data model for eWALL (cloud) platform. It is used by Profiling Server, Data Manager, Cloud Gateway and all other software artefacts (also service bricks from WP5) that store, read and use eWALL data. More details about implementation of this artefact can be found in *D4.1.2 Semantic model of eWALL middleware services* [3].

|  |  |
| --- | --- |
| Artefact: Commons data model | |
| Maven artefact | http://serv2.radio.pub.ro/gitlab/wp4/wp4project/blob/stable/commons-data-model/pom.xml |
| Source (Git address) | http://serv2.radio.pub.ro/gitlab/wp4/wp4project/tree/stable/commons-data-model |
| Build (Jenkins) | http://ewall.radio.pub.ro/jenkins/job/wp4-release/eu.ewall.platform$commons-data-model/ |
| Binaries (Nexus) | http://ewall.radio.pub.ro/nexus/content/repositories/releases/eu/ewall/platform/commons-data-model/1.0.0/ |
| Documentation (Javadoc) | http://ewall.radio.pub.ro/jenkins/job/wp4-release/eu.ewall.platform$commons-data-model/javadoc/ |
| Version | 1.0.0 |
| Responsible partner(s) | ENT |
| Related to task | T4.1 |

# Data Manager

## Positioning of the Data Manager block

Figure 19 shows the main interacting building blocks of the Cloud Middleware (CM) entity. As described in “D2.7 Final user and system requirements and architecture” [2], Data Manager (DM) is the bridge between different eWALL Cloud components and data coming from remote locations such as Sensing Environments. It is thought as an abstraction layer between various entities in the cloud domain, providing data services in both directions - marshalling requests from upper layers and processing data on behalf of distant sensor applications.



**Figure 19: Main building blocks of the Cloud Middleware entity and position of Data Manager**

The Data Manager block thus has several features like data processing, validation and exchange. It receives data from the Cloud Gateway, processes them, stores them in the Cloud Database and makes them available to higher level data processing components (such as IDSS [3], Lifestyle Reasoners, Service Bricks) and other eWALL components.

## Data Manager functionality

The Data Manager block offers multiple functionality features like data processing, validation and exchange. Basic functionality is offered as a create, read, update and delete (CRUD) layer in maintaining at least one data persistence layer covering the basic data model. The persistence layer has an implementation of storage as a MongoDB[[13]](#footnote-14) document database/model set with a core segment that handles POJO (plain old Java object) model persistence. It is extendable and further configurable for different database management system (DBMS) implementations as per requirement (i.e. Relational DBMS – RDBMS, Distributed DBMS – DDBMS, etc.) offering mirroring, replication or other services if needed. Also available is an expandable and dynamic extended object/model set that offers complex data serving in context of a wider model definition as required by usage requirements definitions and data formats expected by use cases, e.g. complex combined objects of interest to service bricks, which are together and further exposed through a RESTful API. The same layer takes care of internal data validation, (re)formatting/translation and processing with transformations where and when required, on the fly or within persistence operations for performance or other reasons.

**Table 1: Data Manager ID card**

|  |  |
| --- | --- |
| Component: Cloud Gateway | |
| Maven artefact | http://serv2.radio.pub.ro/gitlab/wp4/wp4project/blob/master/data-manager/pom.xml |
| Source (Git address) | http://serv2.radio.pub.ro/gitlab/wp4/wp4project/tree/master/data-manager |
| Build (Jenkins) | http://ewall.radio.pub.ro/jenkins/job/wp4-data-manager/ |
| Documentation (Javadoc) | http://ewall.radio.pub.ro/jenkins/job/wp4-data-manager/javadoc/ |
| Deploy (Nexus) | http://ewall.radio.pub.ro/nexus/ |
| Version | 1.0.0-SNAPSHOT |

## Data Manager interfaces

As mentioned earlier, the DM component receives data from the Cloud Gateway, processes them, stores them in the cloud database and makes them available to higher level data processing components. Therefore, communication interfaces from this block to other modules have been defined. The Southbound Interface represents the interface between DM and Cloud Storage, while the Northbound Interface represents the interface between DM and other components like Cloud Gateway and Profiling Server.

Figure 20 presents a high level overview of Data Manager communication interfaces. Data Manager can have more northbound APIs, depending on the current working context, upper layer component which is making the call and the type of the communication. The first API category is used for data pull and push communication.



**Figure 20 Data Manager - Communication Interfaces**

The southbound interface represents the interface of the DM component to the Cloud Storage, namely interface between DM and the MongoDB database. The interface enables basic SCRUD (Search, Create, Read, Update, Delete) operations. The REST API implemented as a Mongo web service offers plain and parameter query enabled endpoints that serve basic data validated and formatted as needed. API offers HATEOAS[[14]](#footnote-15) (Hypermedia as the Engine of Application State) links implementation within produced consumable JSON documents to enable for easy searchable data relations structure and operations discoverability enabling API self-sufficiency as per REST formal definitions. The API can, in later stages, further offer integrated controllable fine-grained authentication and authorization mechanisms to offer direct or cascade control of resources at disposal of accessing entities.

# Cloud Gateway

## Cloud Gateway functionality

The Cloud Gateway (CGw) is responsible for interconnecting eWALL Cloud with Sensing Environments. It is main contact point in communication to and from components running at eWALL Home. It is responsible for receiving processed and indexed measured sensor data, to send control and configuration data to Sensing Environments and for bidirectional application data. It directly communicates with Remote Proxy and supports both message based pull and push communication.

The interface is based on the principles from ETSI M2M Mid interface [4], which use bidirectional, near-real-time, and low-latency communication. To ensure that all communication between eWALL cloud and Sensing Environment is over an encrypted communication channel, all communication is performed over HTTPS (in the case of REST/HTTP communication).

Table 2 presents Cloud Gateway id card with links to source, build file, documentation, deploy location, and version information.

**Table 2: Cloud Gateway ID card**

|  |  |
| --- | --- |
| Component: Cloud Gateway | |
| Maven artefact | http://serv2.radio.pub.ro/gitlab/wp4/wp4project/blob/master/cloud-gateway/pom.xml |
| Source (Git address) | http://serv2.radio.pub.ro/gitlab/wp4/wp4project/tree/master/cloud-gateway |
| Build (Jenkins) | http://ewall.radio.pub.ro/jenkins/job/wp4-cloud-gateway/ |
| Documentation (Javadoc) | http://ewall.radio.pub.ro/jenkins/job/wp4-cloud-gateway/javadoc/ |
| Deploy (Nexus) | http://ewall.radio.pub.ro/nexus/ |
| Version | 1.0.0-SNAPSHOT |

## Cloud Gateway specification

The Cloud Gateway is implemented as **RESTful Web Service** with optional support for **AMQP based messaging** using Spring Framework[[15]](#footnote-16) and Java EE platform.

Main features currently implemented are:

* **Registration** and **deregistration** of Sensing Environments via REST interface and AMQP broker.
* Support for local sensing environment **configuration data retrieval and synchronization**
* **Handling** of **point of contact information** from Sensing Environment via REST interface and AMQP message listener that are used for reaching Sensing Environment from remote.
* **Receiving** and **storing** all **sensing** data coming from local sensing environments, including activity data, environmental sensing (temperature, humidity, luminance, gas levels, movement, and presence), furniture sensing data, appliance sensing data, speaker sensing data, visual sensing and vitals data, using REST/HTTP and AMQP interface
* Acting as **message broker** between local reasoners in local environments and Notification Manager
* **Actuator control support** that allows sensing actuator commands towards actuators in local sensing environment.
* **Provisioning support** (adding, updating, reading deleting) of information about, sensing environments and devices configuration on eWALL cloud.
* **Storing** and **reading** all **data** to and from Mongo database through Data Manager.
* **Full support for AMQP communication** with Remote Proxy.
* Support for displaying **local platform version** on the cloud via eWall Portal.
* Support for **obtaining user's system preferences in local environment**.

Main groups of subcomponents of Cloud Gateway are:

* **Services** – set of service components that realize functionality such as handling registration and deregistration, point of contact information, handling of all measurements from Sensing Environments, actuator control and full configuration and provisioning support.
* **Controllers** – various controllers that provide REST interface towards service components.
* **Data Access Objects** – handle communication towards other cloud components.
* **Receivers** – AMQP counterparts of REST Controllers. They provide the interface between Cloud Gateway and Remote Proxy via AMQP queues.

The Cloud Gateway uses commons data model specified in D4.1.3 [5] for domain specification and representation of all data.

## Cloud Gateway deployment

The Cloud Gateway is a Web Service developed on top of Spring Framework and built as Web Application Archive (WAR) file and deployed at Apache Tomcat Web server running on eWALL OpenStack IaaS (see Section 2 for details on Cloud Infrastructure Layer). As mentioned, the server dedicated to the eWALL private cloud infrastructure is hosted by the Telecommunication Department of the University Politehnica of Bucharest (UPB), and Cloud Gateway services are available through platform development environment available at:

*http://ewall.radio.pub.ro/platform-dev/cloud-gateway/*

The Cloud Gateway has also been successfully tested in integrated environment (see more details in D6.1 Integration report [1]).

## Cloud Gateway interfaces

The Cloud Gateway has different set of software interfaces used for exchange of commands and data primarily from and to Remote Proxy and that can be grouped to the following categories (as presented on Figure 21):

* Control and configuration interfaces (I1)
* Sensing data interfaces (I2)
* Notifications interface (I3)
* Actuator interfaces (I4)



**Figure 21 Cloud Gateway main software interfaces**

Control and configuration Cloud Gateway interfaces (I1) are used for receiving registration and point of contact information (version, status, etc.) from remote proxy running in local home environments, synchronization of local and cloud data, obtaining device configuration data from cloud and remote configuration of local platform.

Sensing data interfaces (I2) are used for receiving all sensing data from home environments and storing them using Data Manager. This includes user activity data, environmental sensing data (temperature, humidity, luminance, gas levels, movement, and presence), furniture sensing data, appliance sensing data, speaker sensing data, visual sensing and vitals data.

Notification interface (I3) is used for exchanging notification messages from local reasoners in local environments and Notification Manager.

Actuator interface (I4) is used for control and sensing actuator commands from cloud components towards local home environment.

While the main communication mechanism for sensor data upload (towards the Cloud) is implemented as a RESTful Web Application, as described in Section 7.2 and in D4.4.3 [2], we considered as important to offer the possibility of a secondary message exchange infrastructure thought both as a fallback layer in case of Cloud Server failure and, in an evolutionary system version, as a reliable channel for binary data / critical data upload.

The adoption of REST as the predominant method to build public APIs has over-shadowed any other API technology or approach in recent years. The vast majority of API developers have opted for REST as their approach and JSON as their preferred message format, which also the main approach taken in eWALL.

However, the need for asynchronicity is especially prevalent within the Internet of Things; connected devices need to publish status-updates, sensory data, etc. to other applications that are listening – instead of those applications having to poll them continuously, which would impose unnecessary bandwidth and processing requirements on the devices themselves (which are often extremely restricted).

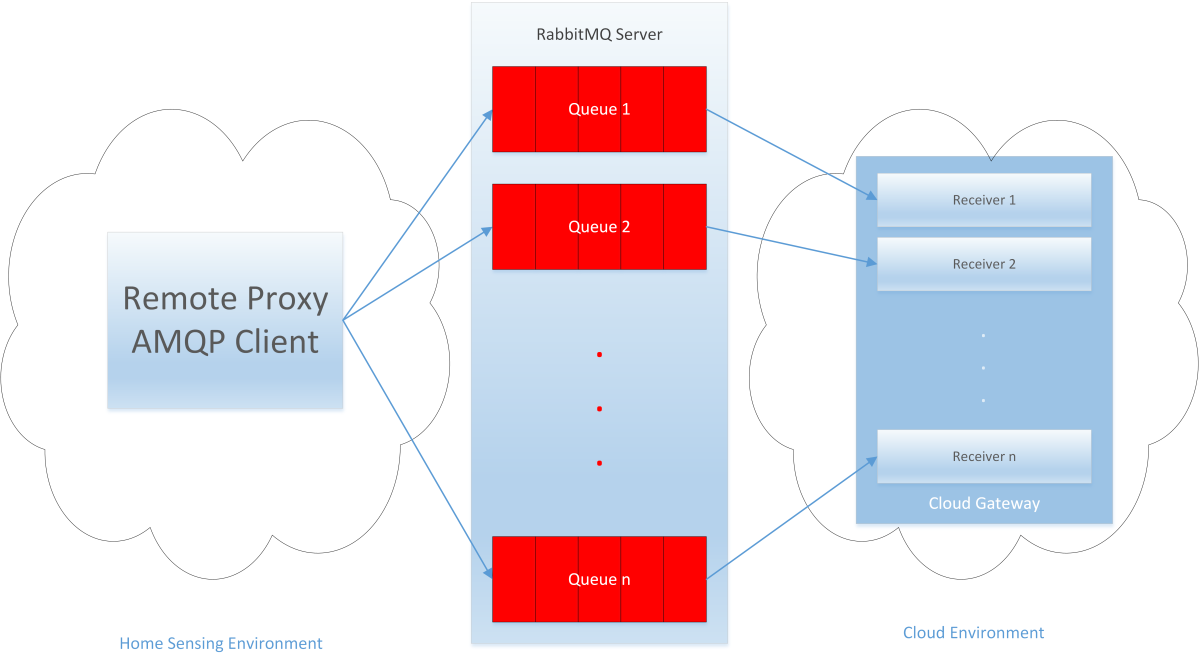
The core of this alternative asynchronous connection is constituted by an AMQP Server (in our case we considered the open source RabbitMQ[[16]](#footnote-17) implementation of such). AMQP (Advanced Message Queuing Protocol) is a protocol that was originally designed for high performance data communication between financial institutions. Several versions were defined and implemented over the years, the last one, AMQP 1.0, massively extended and completed the previous ones. It is designed as a middleware layer to facilitate easy interconnectivity among client applications improving their interoperability. The standard offers a set of interfaces implemented in corresponding drivers, that abstract away implementation and language differences between clients.

In this final iteration of the eWALL Cloud Middleware (CM) the aforementioned AMQP fallback layer is enhanced and most functionality is implemented, with the main elements shown in Figure 22, and details in Table 3.

Data transfer between the two environments is done in an asynchronous fashion. RabbitMQ is, actually, a messaging broker - an intermediary for messaging. It gives eWALL applications a common platform to send and receive messages, and the messages a safe place to live until received. The main benefit of this approach is that the communicating entities (HSE and CM, or, more specifically, RP and CGw) are decoupled at the transport level. The different types of messages from the sensing environment are stored in corresponding queues on the RabbitMQ server. In the CGw, message listeners (called, in our system, AMQP Receivers) retrieve, from the queue, the measurements as soon as they are received. Since the queuing service works message oriented, the actual message handling is done by RabbitMQ, more specifically by its Spring AMQP implementation[[17]](#footnote-18). The Sensing Environment (Remote Proxy) counterpart is described in D4.4.3 [2].

**Table 3 AMQP Receivers implemented in CGw**

|  |  |  |
| --- | --- | --- |
| **No** | **Receiver** | **Role** |
| 1 | Simple Measurement Receiver | Receives generic data (temperature, humidity, illuminance etc.) |
| 2 | Accelerometer Measurement Receiver | Receives accelerometer data |
| 3 | Mattress Pressure Sensing Receiver | Receives mattress pressure sensing data |
| 4 | Vitals Sensing Receiver | Receives vitals sensing data (heart rate, oxygen saturation, blood pressure) |
| 5 | Visual Sensing Receiver | Receives visual sensing data |
| 6 | Speaker Sensing Receiver | Receiver speaker sensing data |
| 7 | Appliance Power Sensing Receiver | Receives appliances power sensing data |
| 8 | Notifications Receiver | Receives notifications |
| 9 | Timestamp Receiver | Receives messages related to last timestamp a record in the database has |



**Figure 22: Message exchange between Home Sensing Environment and Cloud Environment using AMQP**

# Bibliography

|  |  |
| --- | --- |
| [1] | The eWALL Consortium, «D6.1 Integration Report,» eWALL for Active Long Living FP7 project. |
| [2] | «D4.4.3 - eWALL Platform,» 2016. |
| [3] | «D4.2.3 Intelligent support system for eWALL,» 2016. |
| [4] | TS 102 921, «Machine-to-Machine communications (M2M); mIa, dIa and mId interfaces,» ETSI, 2012. |
| [5] | «D4.1.3 Semantic model of eWALL middleware services,» 2016. |
| [6] | The eWALL Consortium, «D2.3 Preliminary System Architecture,» eWALL for Active Long Living FP7 Project, Apr, 2014. |
| [7] | The eWALL Consortium, «D2.7 Final user and system requirements and architecture,» eWALL for Active Long Living FP7 Project, Febr, 2015. |

1. http://www.openstack.org/ [↑](#footnote-ref-2)
2. http://memcached.org/ [↑](#footnote-ref-3)
3. http://www.zabbix.com/product.php [↑](#footnote-ref-4)
4. <https://en.wikipedia.org/wiki/Object_storage>, accessed on 14 Jan 2016 [↑](#footnote-ref-5)
5. <http://docs.openstack.org/admin-guide-cloud/objectstorage_intro.html>, accessed on 14 Jan 2016 [↑](#footnote-ref-6)
6. https://www.dropbox.com/ [↑](#footnote-ref-7)
7. https://www.google.com/drive/ [↑](#footnote-ref-8)
8. http://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html [↑](#footnote-ref-9)
9. http://www.cnpd.public.lu/en/actualites/international/2015/03/AWS/AWS-3-6-15.pdf [↑](#footnote-ref-10)
10. https://www.virtualbox.org/ [↑](#footnote-ref-11)
11. http://www.mongodb.org/ [↑](#footnote-ref-12)
12. http://self-issued.info/docs/draft-ietf-oauth-json-web-token.html [↑](#footnote-ref-13)
13. http://www.mongodb.org/ [↑](#footnote-ref-14)
14. <http://spring.io/understanding/HATEOAS> [↑](#footnote-ref-15)
15. Spring, http://spring.io/ [↑](#footnote-ref-16)
16. <http://www.rabbitmq.com> [↑](#footnote-ref-17)
17. <http://projects.spring.io/spring-amqp/> [↑](#footnote-ref-18)