HORIZONS 2020 PROGRAMME

Research and Innovation Action – FIRE Initiative

Call Identifier:	H2020–ICT–2014–1
Project Number:	643943
Project Acronym:	FIESTA-IoT
Project Title:	Federated Interoperable Semantic IoT/cloud Testbeds and Applications

Infrastructure for Submitting and Managing IoT Experiments V1

Document Id:	FIESTA-IoT-D4.7-160920-Draft
File Name:	FIESTA-IoT-D4.7-160920-Draft.pdf
Document reference:	Deliverable 4.7
Version:	Draft
Editor:	Rachit Agarwal / Nikolaos Georgantas / Valerie Issarny
Organisation:	Inria
Date:	20 / 09 / 2016
Document type:	R, DEM
Dissemination level:	PU

This document contains information, which is proprietary to the FIESTA-IoT Consortium.

DOCUMENT HISTORY

Rev.	Author(s)	Organisation(s)	Date	Comments
V01	Martin Serrano	NUIG-DERI	2015/02/01	Initial Draft Proposal
V02	Amelie Gyrard	NUIG-DERI	2016/04/25	Table of content
V02	Paul Grace	ITINNOV	2016/05/11	Section: FESTA-IoT Portal Sign In
V03	Rachit Agarwal	Inria	2016/05/25	Table of content
	Ramnath Teja Chekka,	KETI	2016/07/04	Section: Experiment services with NodeRed
	Tiago Teixeira	Unparallel	2016/08/04	Section: Experiment Monitor Services
	Amelie Gyrard	NUIG-DERI	2016/07/12 2016/08/12	Architecture & Methodology & M3 framework sections + Experiment registry and discovery sequence diagrams + explanations + FIESTA-IoT Frond-End platform pictures + explanations + screenshots of the portal and explanations
V04	Rachit Agarwal	Inria	2016/08/17 2016/08/22	Integrated version Redefine TOC, Audience Move section Arch/methodology etc to Deliverable 4.1.1 Section: Google Polymer
V05	Ramnath Teja Chekka	KETI	2016/07/04	Update Section: Experiment Services using Node Red
V06	Rachit Agarwal	Inria	2016/08/23 2016/08/24 2016/08/29 2016/08/31	Documentation of APIs, Integrate contributions Executive summary Conclusion, Experiment Management console, Source Code (EEE subsection), Experiment Deployment Services, Experiment Management services Fixed typos
	Alexander Willner, Ronald Steinke	FOKUS	2016/08/31	Added information about Fed4FIRE portals.
	Tiago Teixeira	Unparallel	2016/09/02	Updated Testbed Monitoring tool
	Martin Serrano, Amelie Gyrard, Joao Bosco Jares	NUIG-DERI	2016/09/02	Experiment Use Case Diagram
V07	Rachit Agarwal	Inria	2016/09/02	Updated TOC based on Nikos comments, Integrated work from

			2016/09/05	FOKUS and Unparallel, Background other tools, Relations with the architecture, Update API Specification Add sequence Diagrams
	Nikos Kefalakis Katerina Pechlivanidou	AIT	2016/09/08	Added existing portals (OpenIoT, VITAL), Existing tools ZKoos, Existing technologies (Node-RED), Platform's Modules Runtime Monitoring, Portal Specification & Implementation, Installation instructions (Portal, modules runtime monitoring).
V08	Rachit Agarwal	Inria	2016/09/08	Integrate AIT contributions
	Nikos Kefalakis	AIT	2016/09/08	Added portal mockup
	David Gomez	UC	2016/09/09	Map based Interface
	Alireza Ahrabien	UniS	2016/09/09	KAT UI
	Ramnath Teja Chekka	KETI	2016/09/09	Update Section: Experiment Services using Node Red
V09	Rachit Agarwal	Inria	2016/09/09	Integrated version
	Amelie Gyrard	NUIG-DERI	2016/09/12	Updated M3 portal section
	Flavio Cirillo	NEC	2016/09/14	Technical Review-1, Added dashboard framework Freeboard in the data visualization tools.
	David Gomez	UC	2016/09/15	Technical Review-2
	Ronald Steinke	FOKUS	2016/09/17	Quality Review
	Nikos Kefalakis	AIT	2016/09/19	Portal Requirements
V10	Rachit Agarwal	Inria	2016/09/19	Address TRs and QR, generate version for submission
V11	Martin Serrano	NUIG	2017/02/16	Format Update and Final Checks
V12	Martin Serrano	NUIG	2017/02/16	Circulated for Approval
Draft	Martin Serrano	NUIG	2017/02/20	EC Submitted

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

TERMS AND ACRONYMS

Acronym	Meaning
API	Application Programming Interface
AuthN	Authentication
AuthZ	Authorization
BPM	Business Process Management
CEP	Complex Event Processing
CLI	Command Line Interface
CSV	Comma Separated Values
CRUD	Create Read Update Delete
DC	Dublin Core
DMS	Data Management Services
DSL	Domain-Specific Language
EaaS	Experiment-as-a-Service
EC	Experiment Controller
EEE	Experiment Execution Engine
EMC	Experiment Management Console
ERM	Experiment Registry Management Component
FCAPS	Fault, Configuration, Accounting, Performance, Security
Fed4FIRE	Federation for Future Internet Research and Experimentation Facilities
FEMO	FIESTA-IoT Experiment Model Object
FIRE	Future Internet Research and Experimentation
FISMO	FIESTA-IoT Service Model Object
FOAF	Friend of a Friend
FRCP	Federated Resource Control Protocol
GCF	GENI Control Framework

GENI	Global Environment for Network Innovations
GR	Good Relations
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
HTTP	HyperText Transfer Protocol
ICO	Internet Connected Objects
IDE	Integrated Development Environment
IERC	European Research Cluster on the Internet of Things
INDL	Infrastructure and Network Description Language
IOT	Internet of Things
JSON	JavaScript Object Notation
KAT	Knowledge Acquisition Toolkit
LOV4IoT	Linked open Vocabularies for Internet of Things
M2M	Machine to Machine
M3	Machine-to-Machine Measurement
MQTT	Message Queuing Telemetry Transport
NDL-OWL	Network Description Language based on the Web Ontology Language
NEPI	Network Experimentation Programming Interface
NML	Network Mark-Up Language
OEDL	OMF Experiment Description Language
OGF	Open Grid Forum
OMF	cOntrol and Management Framework
OML	Orbit Measurement Library
OMN	Open-MultiNet
OMSP	OML Measurement Stream Protocol
OSMO	OpenIoT Service Model Objects
OWL	Web Ontology Language

PPI	Platform Provider Interface
QoS	Quality of Service
RC	Resource Controller
RDF	Resource Description Framework
REST	Representational State Transfer
S-LOR	Sensor-based Linked Open Rules
SD	Service Discovery
SenML	Sensor Markup Language
SFA	Slice-based Federation Architecture
SLA	Service Level Agreement
SSN	Semantic Sensor Network
SWE	Sensor Web Enablement
SWoT	Semantic Web of Things
TLS	Transport Layer Security
W3C	World Wide Web Consortium
WP	Work Package
WYSIWYG	What-You-See-Is-What-You-Get
XHTML	Extensible Hypertext Markup Language
XML	Extensible Markup Language
XML-RPC	XML Remote Procedure Calling protocol
XUL	XML User Interface Language
ZUML	ZK User Interface Markup Language

1 EXECUTIVE SUMMARY

FIESTA-IoT platform is a data store that federates IoT data coming from various sources called testbeds. Experimentation on such data is performed in order to gain more knowledge about the environment the “things” are sensing. FIESTA-IoT provides a mechanism for experimenters that perform experiments over IoT data from various sources. This deliverable provides experimenter an overview of tools and services build in order to facilitate them towards creating and executing their experiment, i.e. services and tools needed to realize the lifecycle of the experiment.

Further, this deliverable covers and report proceedings of Task 4.4 and Task 4.5 within WP4. The Task 4.4 aims to develop portal infrastructure that serves as a single entry point for accessing data, services, and resources to testbeds and experiments. The portal enables experiments to browse resources and data, regardless of their testbed-origin. Furthermore, it integrates the authentication and authorization functionalities developed in Task 4.2, as part of the users’/experimenters’ management module of the portal. In this deliverable we provide current developments with respect to the portal. Note that, the portal also lists and provides WP3 APIs and tools to the end users.

The Task 4.5 aims to develop tools for development, deployment, and management of the experiment. These tools aim to provide necessary services for the successful execution of the lifecycle of the experiment. The tools developed under Task 4.5 are provided as APIs for advanced experimenters and can also be accessed via the portal developed as part of Task 4.4.

To achieve the above-mentioned goals, we start by analyzing requirements coming from the three “in house” experiments (i.e. “Data assembly and services portability experiment”, “Dynamic discovery of IoT resources experiment”, and “Large-scale crowdsensing experiment”. For more information on the experiments readers are directed to [1][2]) and from Task 4.1 (proceedings of which are available via [3]) for the platform and analyzing requirements for the portal. This deliverable focus on the tools made available to the experimenters via the FIESTA-IoT portal and provides an overview of the portal itself.

One of the main functional components developed to fulfill the aim of this deliverable is the Experiment Execution Engine (EEE) that facilitates the scheduling of the described experiment on the Meta cloud (see [4]). The UI tool Experiment Management Console (EMC) supports the EEE and enables experimenters to visualize the status of their experiment and monitor them. Nevertheless, it is also essential to define an experiment. The experiment definition follows the Domain Specific Language (DSL) created in the Task 4.1 (described in [3]). This DSL consists of an entity called FISMO (FIESTA-IoT Service Model Object) that provides value to main parameters to the APIs of the EEE that schedules the experiment on the Meta-Cloud. Together with this, we refer the readers to [3] for the complete knowledge about the workflow and services for the creation of an experiment. FIESTA-IoT customized version of Node-RED tool helps experimenters create the DSL instance that is finally interpreted by the EEE. The created DSL instance provides inputs such as “Scheduling information”, “query to schedule”, etc., to the EEE. As the deliverable also focus on the FIESTA-IoT Portal, the deliverable also reports tools developed to monitor testbeds. Nonetheless, as the FIESTA-IoT

platform has the notion of users, the access to the APIs is secured and ensured by the security component made available via Task 4.2.

In this deliverable, we first provide the relation between the tools developed (and stated in this deliverable) and the FIESTA-IoT reference architecture along with functional requirements addressed and the experiment lifecycle. We then present the background and state of the art tools available in order to build the portal and other components such as EEE. We take inspiration from already existing projects and leverage from partner experience on projects such as OpenIoT, VITAL, etc. We then list services made available via EEE. These services along with those developed in WP3 from the core of the FIESTA-IoT platform and are supported by a user-friendly Graphical User Interface (GUI, also called as UI). We then provide specification of the developed UI tools, such as the experiment management console and the visualization tools, to name a few. The specification of the portal and its implementation follow next. A simple mockup walk through follows the portal specification for a clear understanding. A clear installation steps and how to use components follow the tools section. The conclusion concludes the deliverable.

1.1 Audience

This deliverable is useful to the following audiences:

- **Researchers and engineers within the FIESTA-IoT consortium** will take into account various requirements in order to research, design and implement the APIs needed to support experiments associated to the FIESTA-IoT platform.
- **Experiment owners who wish to join FIESTA-IoT** will be able to use the tools to produce their own experiments. Experimenters will be able to understand how and what IoT data is stored within the FIESTA-IoT Meta Cloud and thus would be able to align their experiments that could utilize such data.
- **Researchers on Future Internet Research and Experimentation (FIRE) focusing on semantically storing data produced by their experiments** will find guidelines to store data produced by their experiments in a semantic manner either in their own repository or utilizing the FIESTA-IoT platform. The researchers will be able use the ontology and the tools as the reference. Further, if they wish to extend/modify the ontology and tools for their own research, they will be able to do so.
- **Members of other Internet of Things (IoT) communities and projects (such as projects of the IERC cluster)** can take this document as an initial reference or inspiration to design and implement their own testbed that also stores data that is semantically annotated.
- **Application Developers** those wish to use the IoT data and create applications that would provide either knowledge about the resources available, sensed phenomenon or both.
- **Open call** participants will be able to better understand the technical details needed for them to join the FIESTA-IoT consortium.
- **Standardization bodies** will have access to this deliverable as it will be a public document and therefore the tools developed can be standardized following the involvement and reach a wider adoption.

2 RELATION WITH THE ARCHITECTURE, REQUIREMENTS AND EXPERIMENT LIFECYCLE

The functional architecture of the FIESTA-IoT platform is described in [4]. In relation to the platform, in this deliverable we are mainly focused on describing a subset of its components, such as the Experiment Execution Engine (EEE) which is a part of the IoT-Process Management Functional Group (FG) and other components within the Management FG such as the WEB Browsing and Configuration Functional Component (FC) and the Performance monitoring FC. In Figure 1 (part of the FIESTA-IoT Architecture), we identify these components (marked in gray). As for the above-mentioned functional components, we present the EEE, Portal and the Experiment Management Console (part of the Web Browsing and Configuration FC) and the Testbed Monitoring tool (part of the Performance Monitoring FC). In the next subsection we report on the EEE requirements.

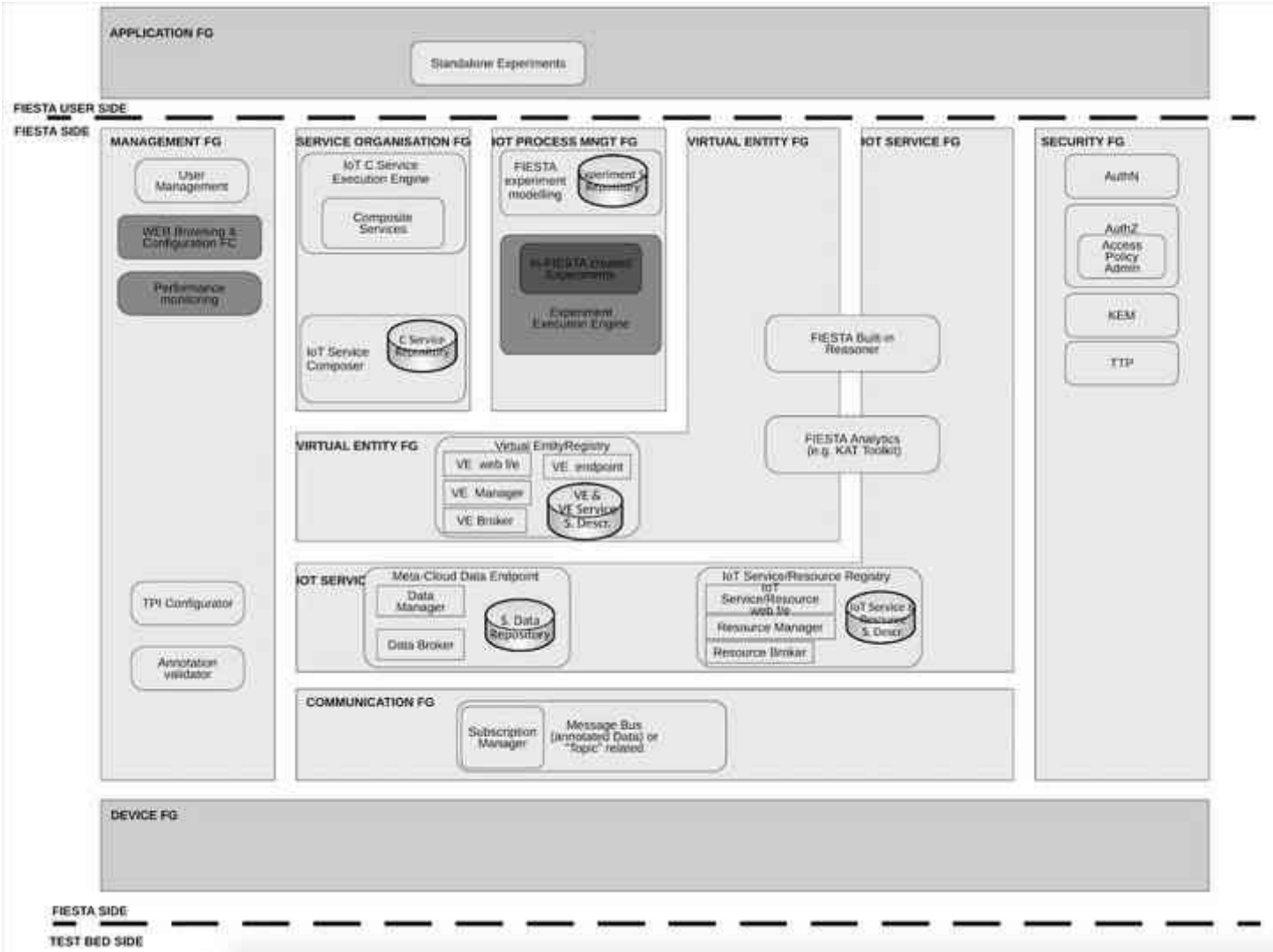


Figure 1: FIESTA-IoT Functional Architecture Components addressed in this deliverable are marked in Grey

2.1 EEE Requirements

Table 1:Requirements addressed by EEE

Requirement ID	Description
11_FR_ACC_Request_data_different_ways	It must be possible for the experimenter to request for data in different ways (e.g. event-based, periodic, and/or autonomous)
23_NRF_ACC_Page_in_subrequests	When a large set of information is requested, it should be possible to page it into different sub-requests
32_NFR_ACC_Provide_dev_deploy_manag_config_tools	FIESTA-IoT should provide development, deployment, management, and configuration tools
41_NFR_PLA_Minimize_processing_delay	Processing delay has to be minimized when requesting information
51_NFR_PLA_FIESTA_hightly_reliable	FIESTA-IoT needs to be highly reliable

The EEE is a component that satisfies part of the non-functional requirement "32_NFR_ACC_Provide_dev_deploy_manag_config_tools" defined in [5] (For the sake of ease in understanding we provide in

Table 1, a description of referred requirements). Further, EEE should be able to:

- Schedule at a defined rate a FISMO as a Job on the Meta Cloud with minimum possible delay: this would require the EEE to read the `QuerySchedule` entity that is a part of FISMO, connect to the Meta Cloud and use the Meta Cloud API to execute the query defined in the `Query` attribute of the FISMO. Note that, a FISMO is a service Model that describes the experiment. It consists of entities such as experiment control, details about the query, etc. More information on FISMO is available in [3]. An experimenter could subscribe to any services (FISMO) on top of their own FISMOs, could request data in different ways for example, based on time period, and could poll for certain data (event based). This would satisfy the "`11_FR_ACC_Request_data_different_ways`" functional requirement. In order to efficiently server the experimenters with the data the EEE should understand the request quickly and should be able to get the data quickly from the Meta-Cloud. Thus EEE along with Meta-Cloud should satisfy the "`41_NFR_PLA_Minimize_processing_delay`" non-functional requirement [5].
- Schedule multiple FISMOs on the Meta Cloud simultaneously.
- Poll a service to get the data: This would require EEE to execute the query defined in the `Query` entity of the FISMO once and on demand.
- Upon scheduling maintain a state variable of the scheduled job: this would enable and help experimenters to know the state of their experiment.

- Maintain a log of executed jobs: this would enable experimenters to know how many times a specified FISMO has been successfully executed.
- Provide a mechanism to the experimenters to subscribe/unsubscribe to a certain already discoverable FISMO: this would enable experimenters to utilize already existing FISMOs in their experiments. In order to subscribe, the experimenter should provide the `experimentOutput` attribute in the FISMO so that the EEE could produce the output accordingly.
- On top of subscription, if a FISMO is deleted by its owner, then all the subscribers will be notified about the deletion: this would allow subscribers to know if the experimenter has to subscribe to any other FISMO or create a new FISMO in order to keep the experiment performing as expected. This requirement is best fulfilled by the Experiment Registry Management Component (ERM).
- For a large `resultset` paging of the result should be provided: this would enable the "`23_NRF_ACC_Page_in_subrequests`" non-functional requirement already specified in [5].
- As this is the core for experiment execution the EEE should be stable and satisfy the "`51_NFR_PLA_FIESTA_highly_reliable`" non-functional requirement [5].

2.2 Portal Requirements

Table 2:Requirements addressed by Portal

Requirement ID	Description
20_FR_SEC_Experimenter_single-sign-on	Single-sign-on mechanism has to be in place
50_NFR_PLA_FIESTA_scalable_extensible_upgradable	The FIESTA-IoT platform must be scalable, extensible and upgradable
70_NFR_SEC_Different_profile_types	FIESTA-IoT must have different profile types (e.g. different kind of experimenters, researchers, etc.). Thereby grant different permissions

The FIESTA-IoT portal is the single entry point for accessing all the visual tools that are being offered from the different FIESTA-IoT tasks for designing, deploying and managing experiments. These tools will provide access to data, services and resources of all the federated IoT testbeds. Further, the FIESTA-IoT portal should be (For the sake of ease in understanding we provide in Table 2, a description of referred requirements):

- Seamlessly integrate all the available visual tools provided from the different tasks within the FIESTA-IoT project,
- As portal is a part of FIESTA-IoT platform it is required to be scalable, extensible and upgradable. This would satisfy a

"50_NFR_PLA_FIESTA_scalable_extensible_upgradable" with respect to the portal [5]. Note that the FIESTA-IoT platform should still be scalable, extensible and upgradable.

- Support a Single-sign-on mechanism (thereby satisfying "20_FR_SEC_Experimenter_single-sign-on" [5]). The user this way will provide his/her credentials only once to the portal and the portal will be responsible to authorize the user with all the visual tools integrated to it.
- Support different profile types (e.g. different kind of experimenters, researchers, etc.) and thereby grant different permissions (thereby satisfying "70_NFR_SEC_Different_profile_types" [5]). This way the portal should dynamically generate the offered tool list based on the user role.
- Provide appropriate menus through which the user will be able to easily navigate through the offered toolset.

2.3 Experiment Lifecycle

The experiment lifecycle consists of 5 phases. These phases are:

- Phase 0: Experiment registration where experimenters identify themselves.
- Phase 1: Discovery of resources.
- Phase 2: Defining the experiment and creating the DSL instance.
- Phase 3: Execution of the experiment.
- Phase 4: Retrieving the results.

We refer the readers to [3] in order for them to understand the experiment lifecycle in details.

2.3.1 Use Case Diagram and Sequence Diagrams

Based on the requirements, in Figure 2 we present a use case diagram for the experimentation infrastructure. The use case is inspired by the Create Read Update Delete (CRUD) principles. On top of CRUD, EEE functionalities are also provided. Specifically the functionalities of the EEE include: scheduling of a job, subscription to other FISMOs, un-subscribing from subscribed FISMOs, manual start and stop of a job, and automatic rescheduling of a job upon server restart. An easy to use UI for the novice experimenter is also provided with which the experimenters can retrieve the execution logs and the status of a job.

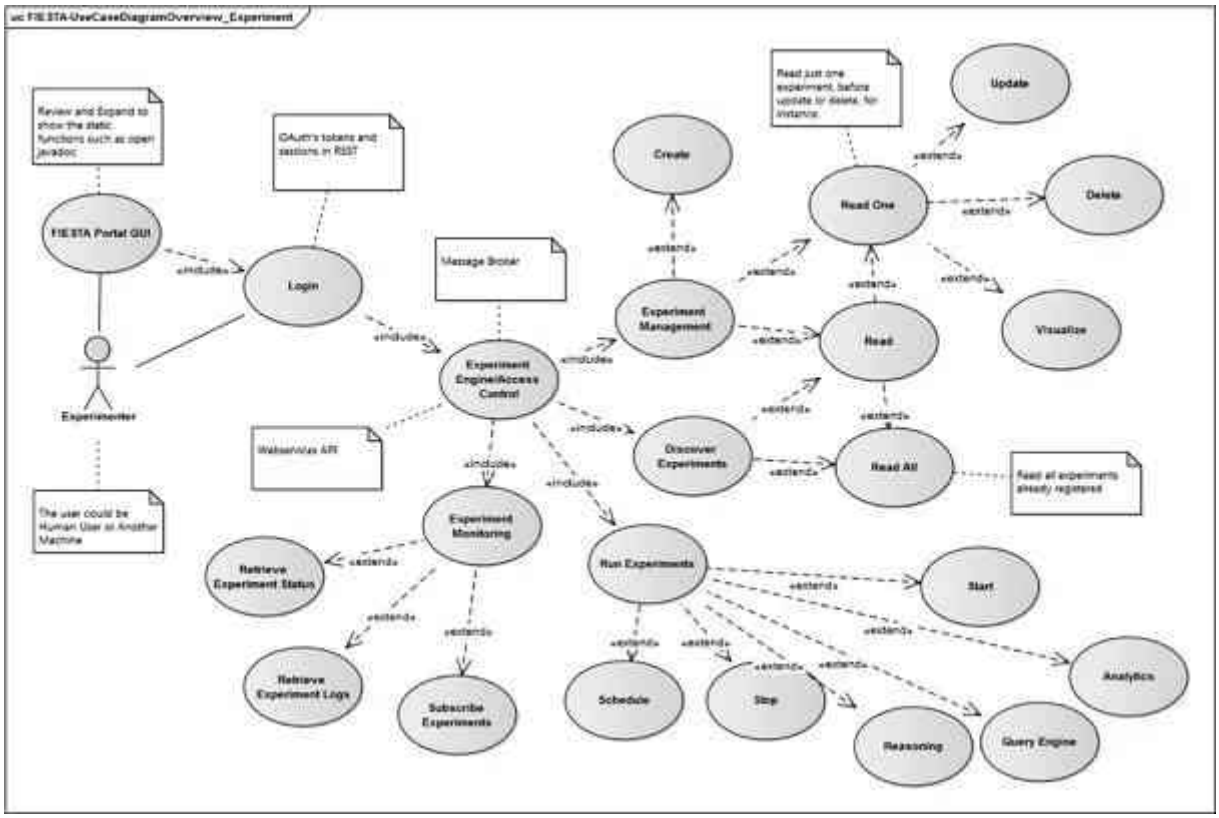


Figure 2: Experiment Use Case Diagram

Below we provide the different sequence diagrams for these use cases, covering the creation of an experiment, modification, subscription, scheduling and stopping an experiment. Figure 3 shows the experiment definition and registration sequence diagram. Here, the experimenter first login into the FIESTA-IoT platform.

The experimenter using ERM's discovery services, discovers existing experiments that are stored in the mySQL database. If, the experimenter is able to get desired experiment that he/she wants, nothing has to be done. Otherwise, the experimenter can either create the experiment specific DSL instance using any XML editor he/she wants and send the DSL instance to the ERM so that it can be stored in the experiment repository or use the Node-RED to create the experiment specific DSL instance. Internally, a library (an extension to the Node-RED) converts the Node-RED specific JavaScript Object Notation (JSON) format to the FIESTA-IoT specific DSL instance and sends the DSL instance to the ERM for storage.

The "save" service in the ERM parses the DSL instance and if everything is OK, the DSL instance is stored in the Experiment Repository. Note that in Figure 3 we only depict one scenario. Once the experiment is stored, an experimenter can request for modifications. For this, the experimenter has to first login to the FIESTA-IoT platform, and then request for the previously stored experiments he/she created.

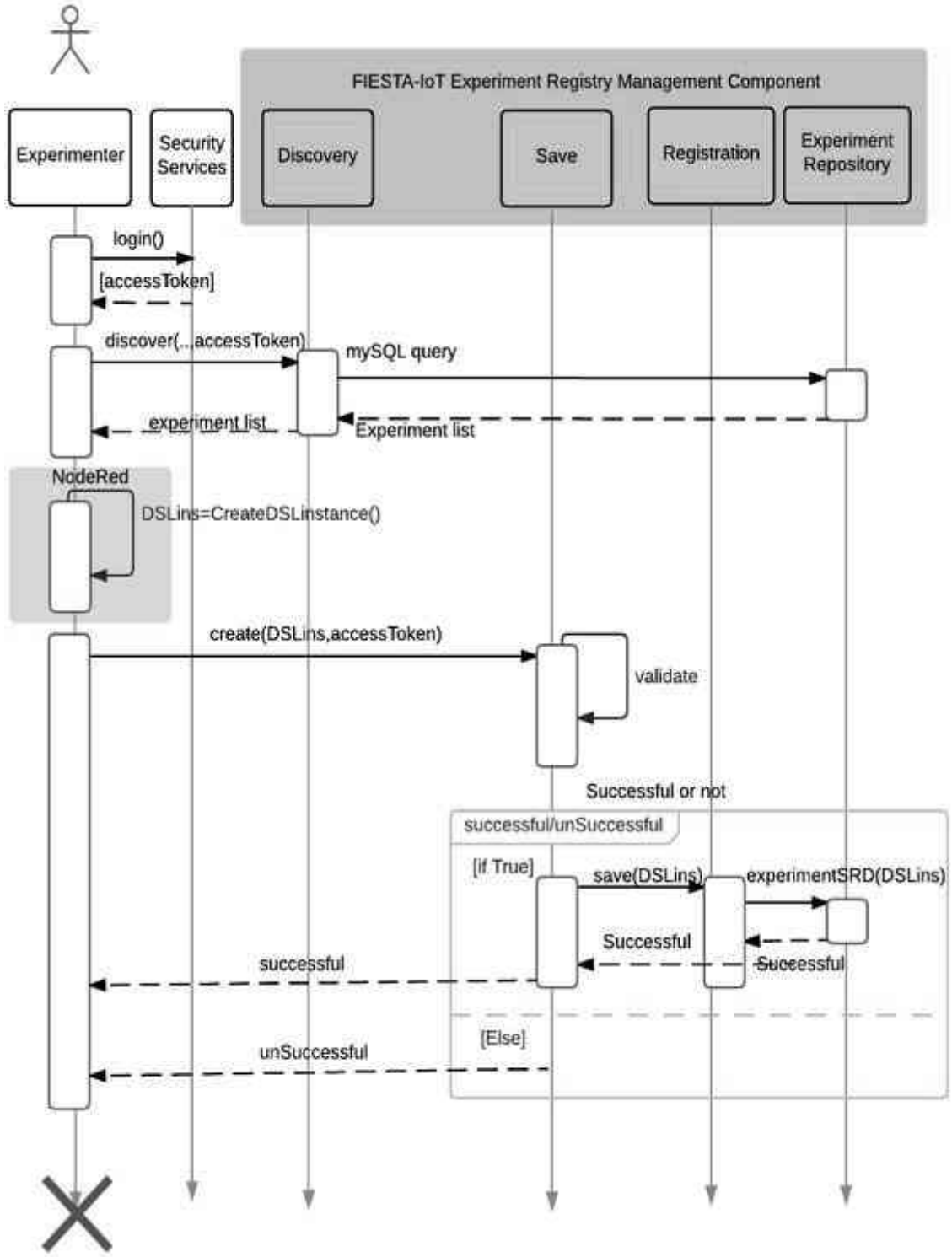


Figure 3: Experiment definition registration

The experiment is then opened in the Node-RED editor in which the experimenter updates their experiment. This would again produce a new JSON file. This JSON file is then converted to the FIESTA-IoT specific DSL and sent for storage. The “save”

service of the ERM parses and instructs the EEE to stop the experiment job if they are running and reschedule the job. Once this is done, the experiment is also stored in the repository. This process is depicted in Figure 4. For unregistering an experiment, a similar process is followed. The same is depicted in Figure 5.

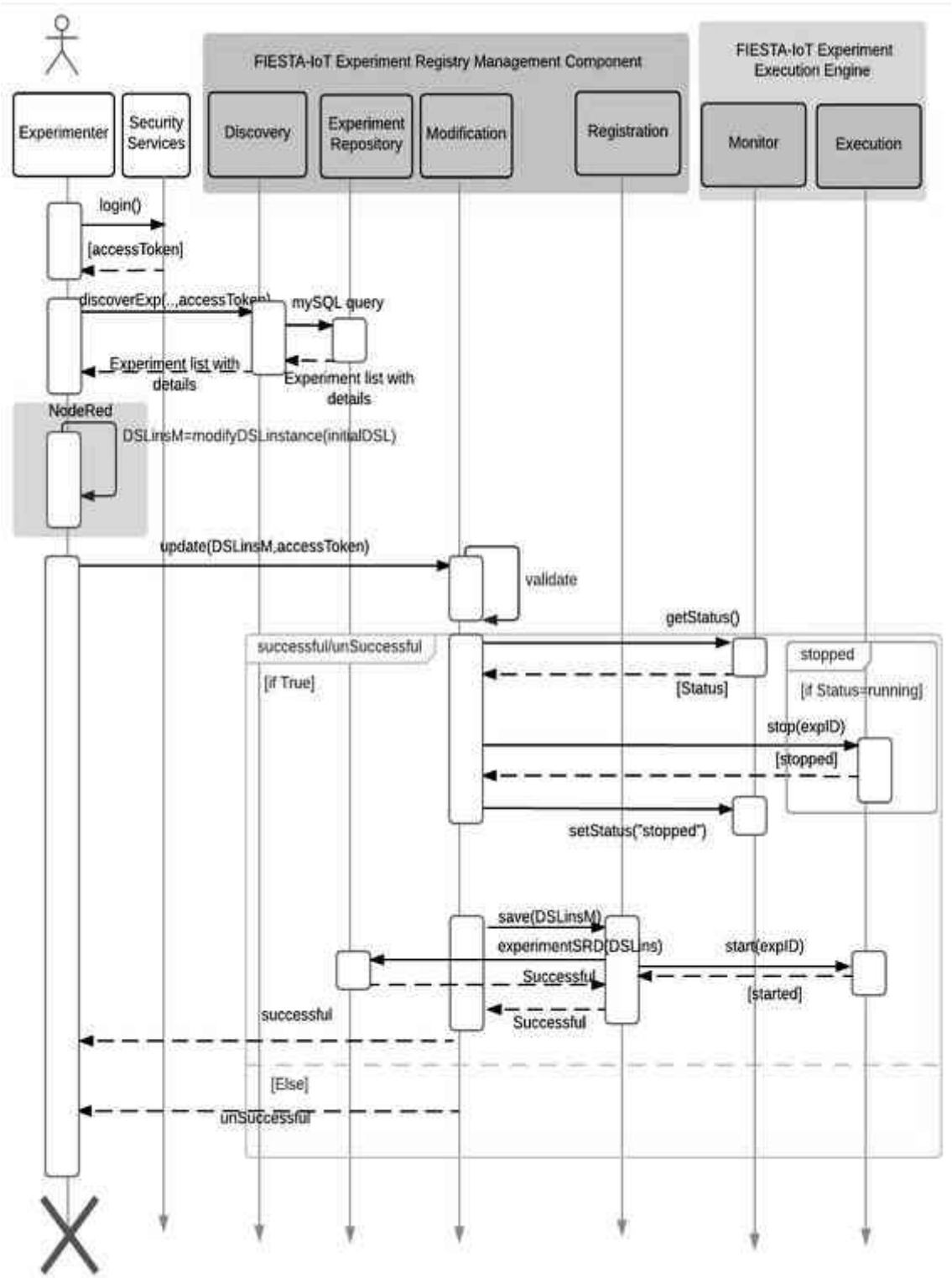


Figure 4: Experiment Modification

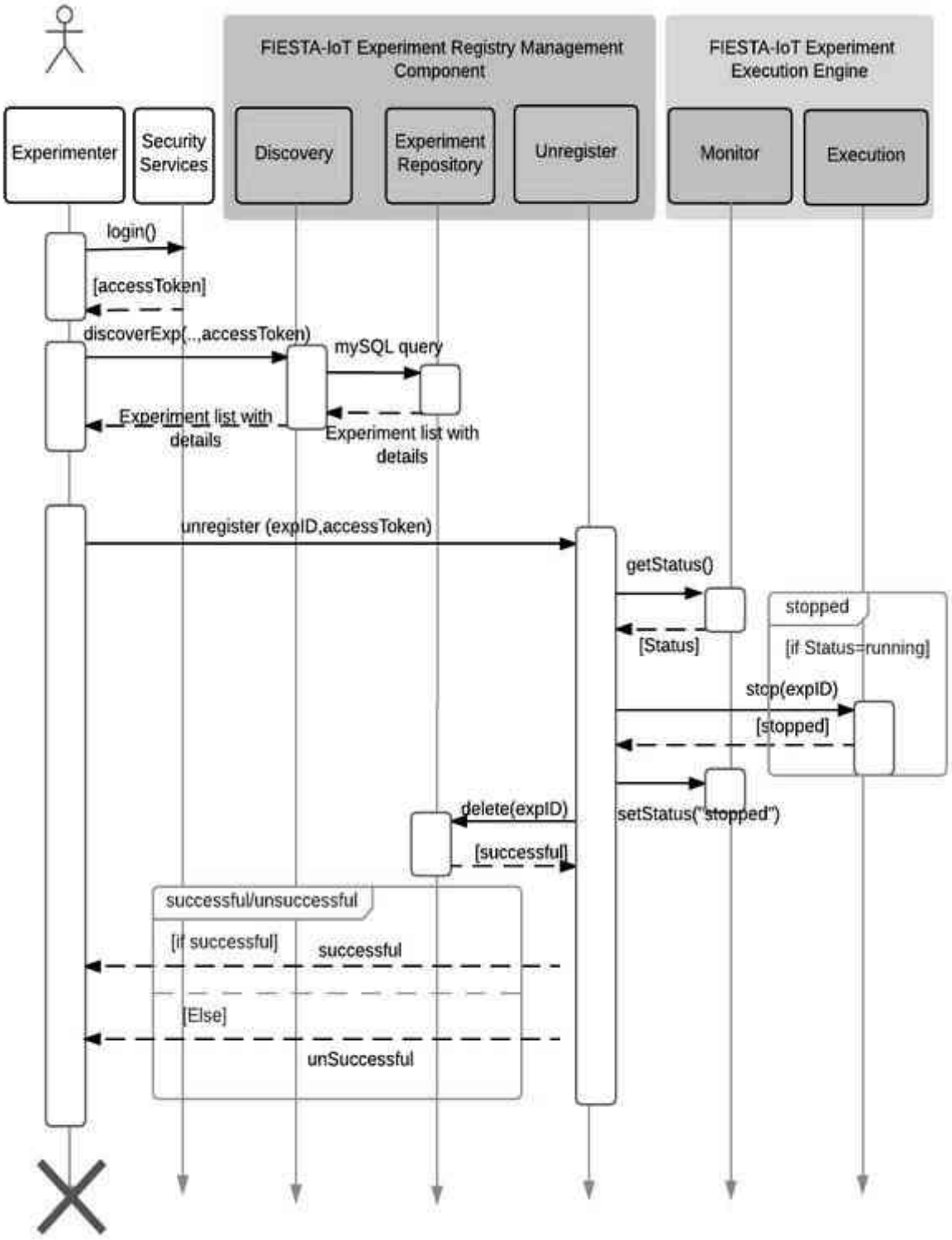


Figure 5: Unregister

The experimenters can subscribe to any available FISMO for their experiment and could request result of the subscribed FISMO to be sent to endpoint specified by them. The subscription would allow experimenters to reuse the existing FISMOs and use the results made available by the FISMO in their experiment. This requires experimenters to first login and then search for available (discoverable) FISMOs. Once the list is available the experimenter selects the `FISMOID` and provides the necessary parameters. The EEE stores this information in the experiment repository and returns the *response* as *Subscribed* to the experimenter. The same is depicted in Figure 6. For unsubscribing, a similar process is followed.

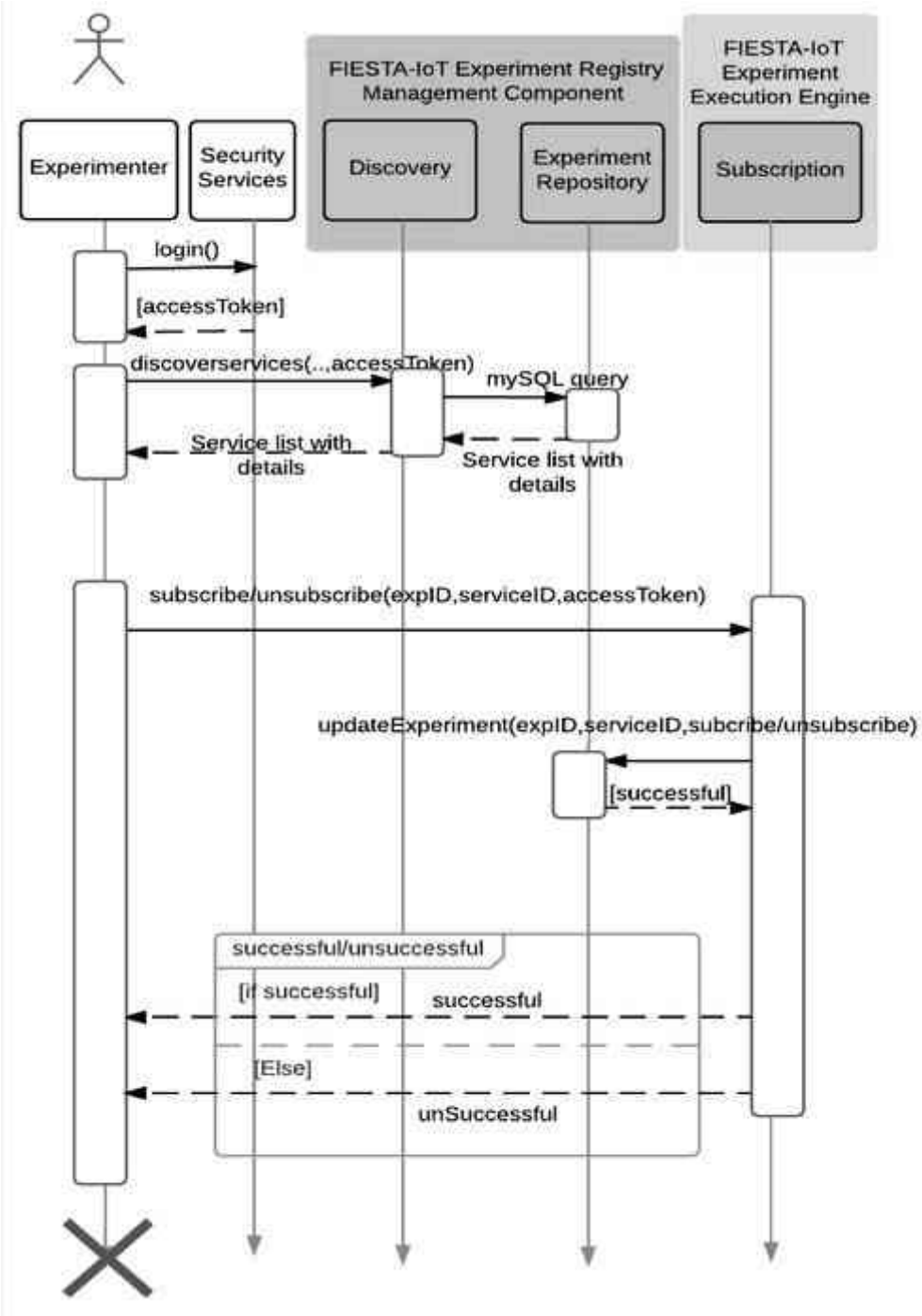


Figure 6: Subscribing/unsubscribing FISMOs

The EEE is responsible for scheduling an experiment services and executing the query. To start a service, the experimenter login into the FIESTA-IoT Platform and searches for the FISMO he/she wants to start. Once the service is identified by the experimenter for starting it is sent to the scheduler. The scheduler checks for the status of the FISMO and identifies if the FISMO is already scheduled or not. If the FISMO is not scheduled it schedules the FISMO, provides a `JobID` to it, sets the status and calls the Meta Cloud repository as and when needed by the service. The results of the execution of the FISMO are sent to the result storage specified by the experimenter in the FISMO object. If the FISMO was already scheduled, an appropriate message is sent. The same is depicted in Figure 7. For stopping, a similar process is followed (see Figure 8).

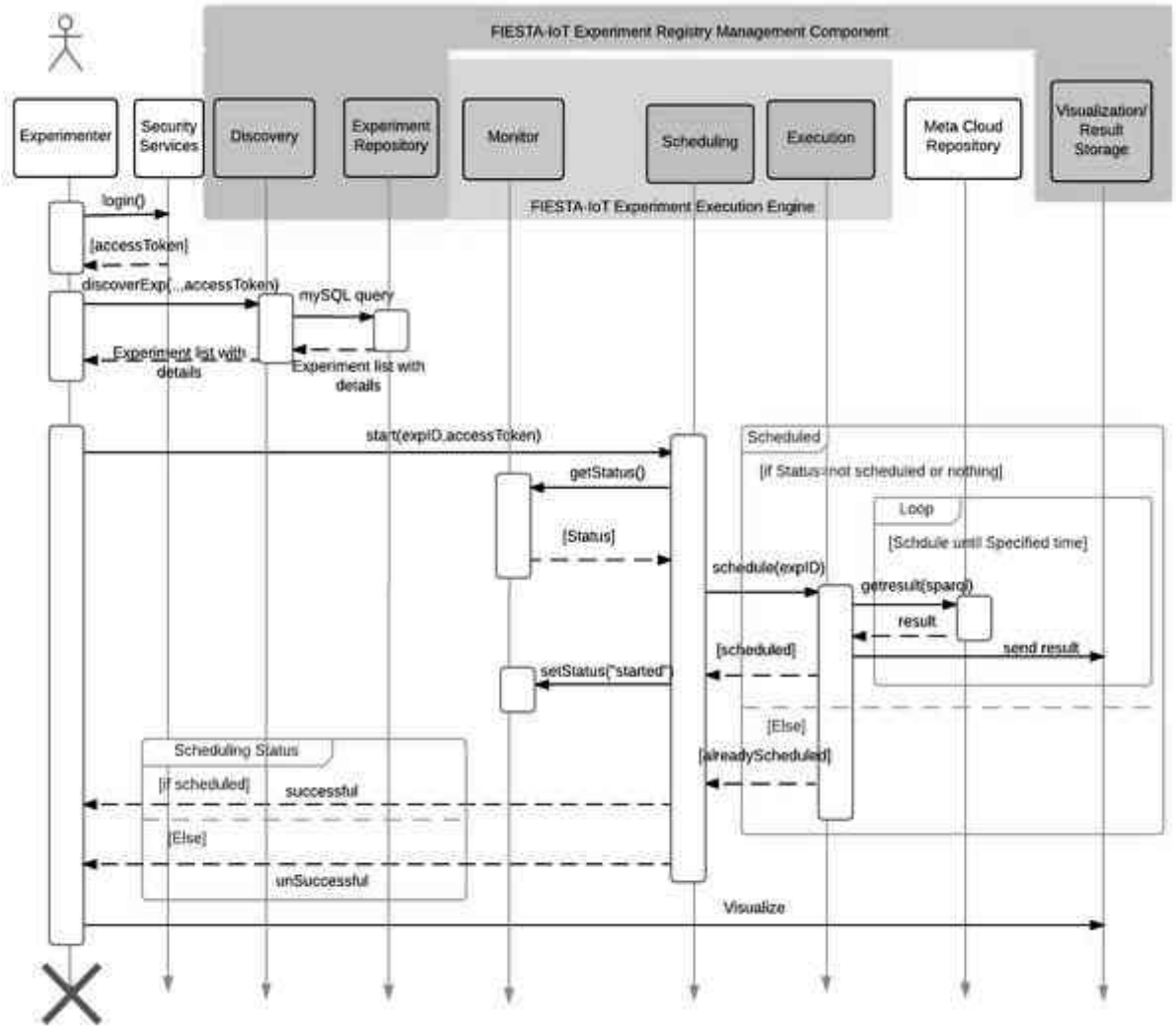


Figure 7: Starting a service

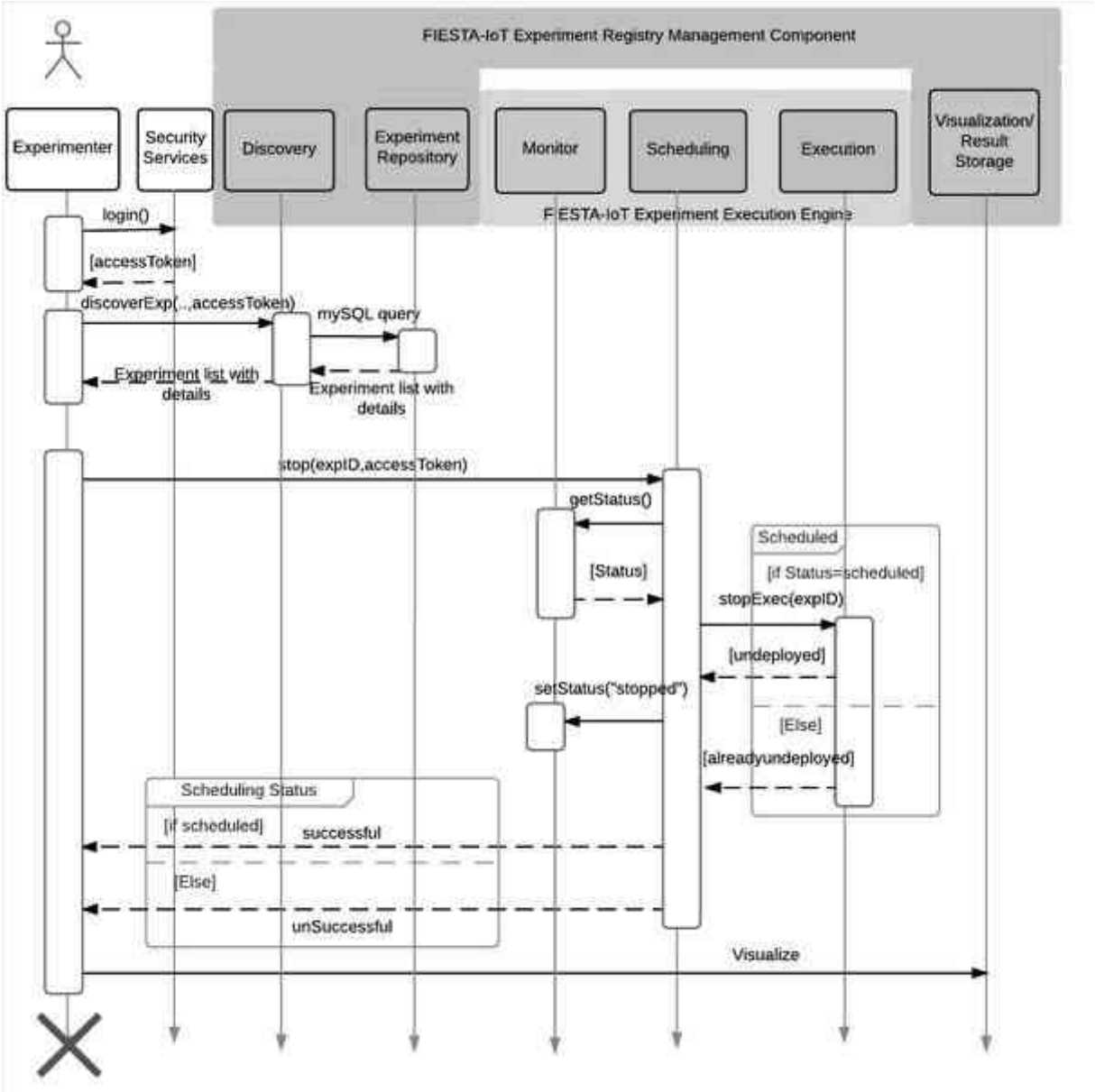


Figure 8: Stopping a service

3 EXISTING TOOLS: BACKGROUND & RELATED WORK

In this section, we investigate about (a) existing portals that are dedicated towards providing semantic services and (b) existing technologies that can be used to build the services and the portal.

3.1 Existing Portals

Below we introduce various off the shelf portals that deal with the provision of semantic-based services.

3.1.1 OpenIoT

The OpenIoT architecture was designed with user-friendliness in mind. This mainly involves hiding the complexity to automate and create an environment as pleasant as possible for all the OpenIoT user interfaces and applications [6]. So the core environment of user interaction with the rest of OpenIoT platform is the OpenIoT Integrated Development Environment (IDE) (see Figure 9). This environment resides in the higher tier of the OpenIoT architecture, and is responsible for the sensors, modules management and enabling the configuration of the OpenIoT service delivery capabilities. It provides a single point of entry for all the OpenIoT tools and applications by utilizing modern Web technologies. The OpenIoT IDE provides the Core Environment, the Request Definition, the Request Presentation, the RDF Schema Editor, and the monitoring tools which are introduced below.

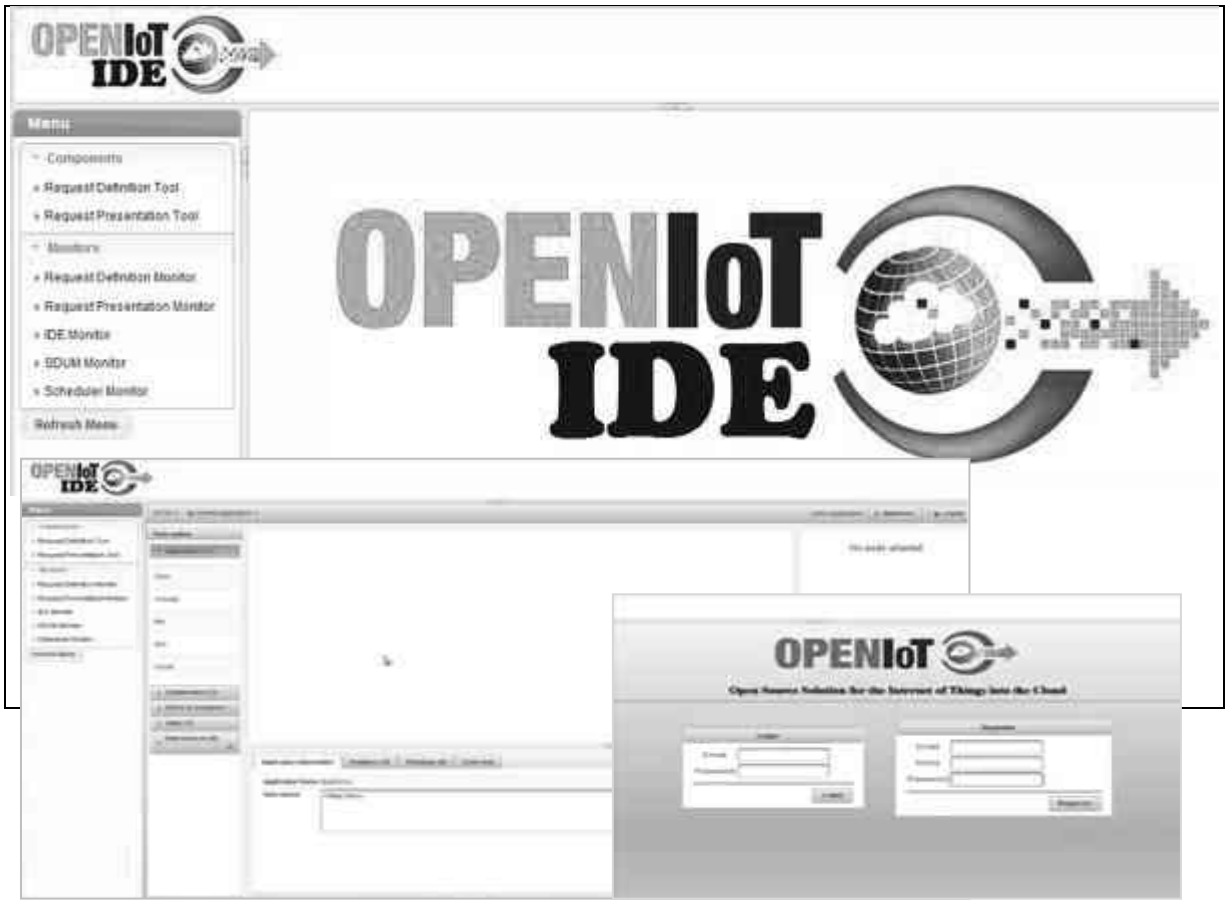


Figure 9: IDE Layout and Main Screenshots [6]

Request Definition

The Request Definition module (see Figure 10) is a Web application that allows end users to visually model their OpenIoT-based services using a node-based WYSIWYG (What-You-See-Is-What-You-Get) UI. Modelled service graphs are grouped into “applications”, i.e., OpenIoT Application Model Objects (OAMOs). These applications are able to group a collection of different services, i.e., OpenIoT Service Model Objects (OSMOs) which comprise/describe a real life application (i.e., weather reports). This enables end users to manage (describe/register/edit/update) different (unrelated) applications from a single point. All modelled services are stored by the OpenIoT Scheduler and are automatically loaded when a user accesses the Web application.

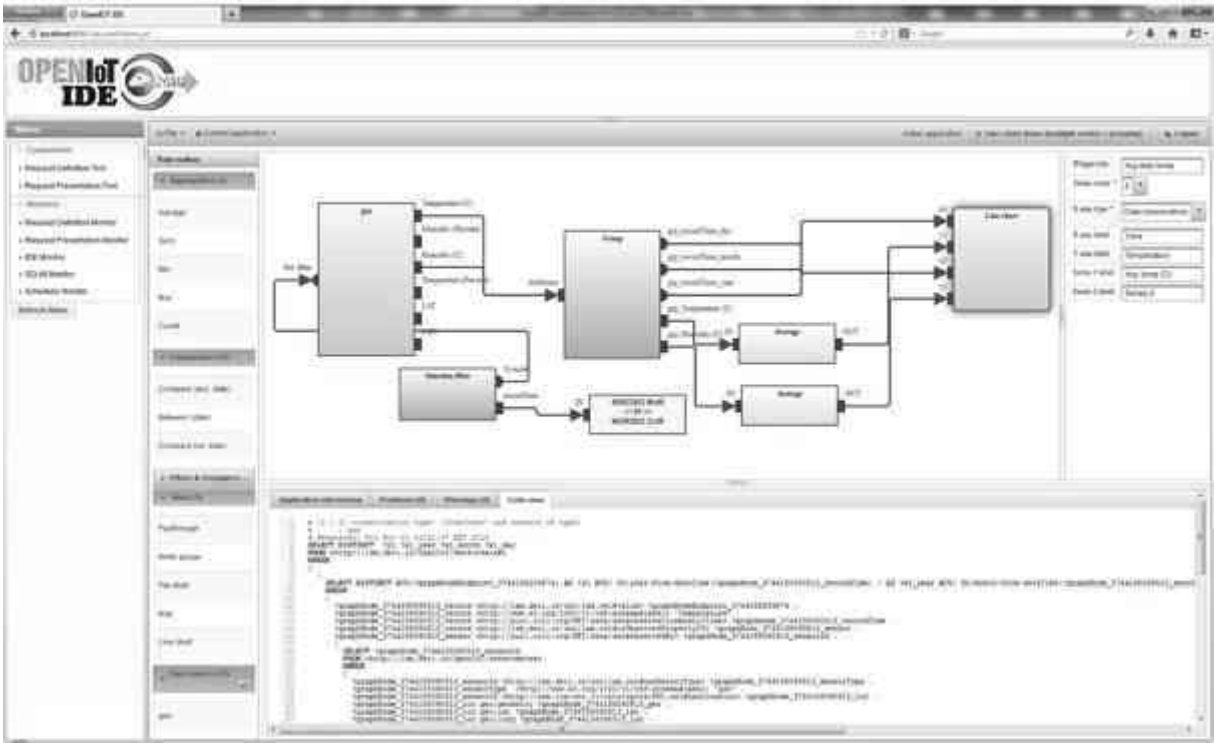


Figure 10: Request Definition view [6]

Request Presentation

The Request Presentation module is a Web application that provides end users with a visual interface to services created using the Request Definition Web application. When a user accesses the Web application, all his/her modeled applications are automatically loaded. Each application contains one or more visualization widgets. The Request Presentation layer parses the application metadata and generates a self-updating widget dashboard (see Figure 11).

Figure 11: Request Presentation visualization widgets example [6]

Sensor Schema Editor

The Sensor Schema Editor supports the average user in annotating sensors and sensor-related data using the OpenIoT ontology and Linked Data principles. The interface automates the generation of Resource Description Framework (RDF) descriptions for sensor node information submitted by users.

The Sensor Schema Editor has two parts, namely the frontend interface (as shown in Figure 12) and the backend LD4Sensors Server. The Schema Editor interface depends on the LD4Sensors Server for generating Linked Data descriptions for sensor metadata and sensor observations. The LD4Sensors Server exposes a REST API that takes as input the sensor metadata and returns the RDF representation.

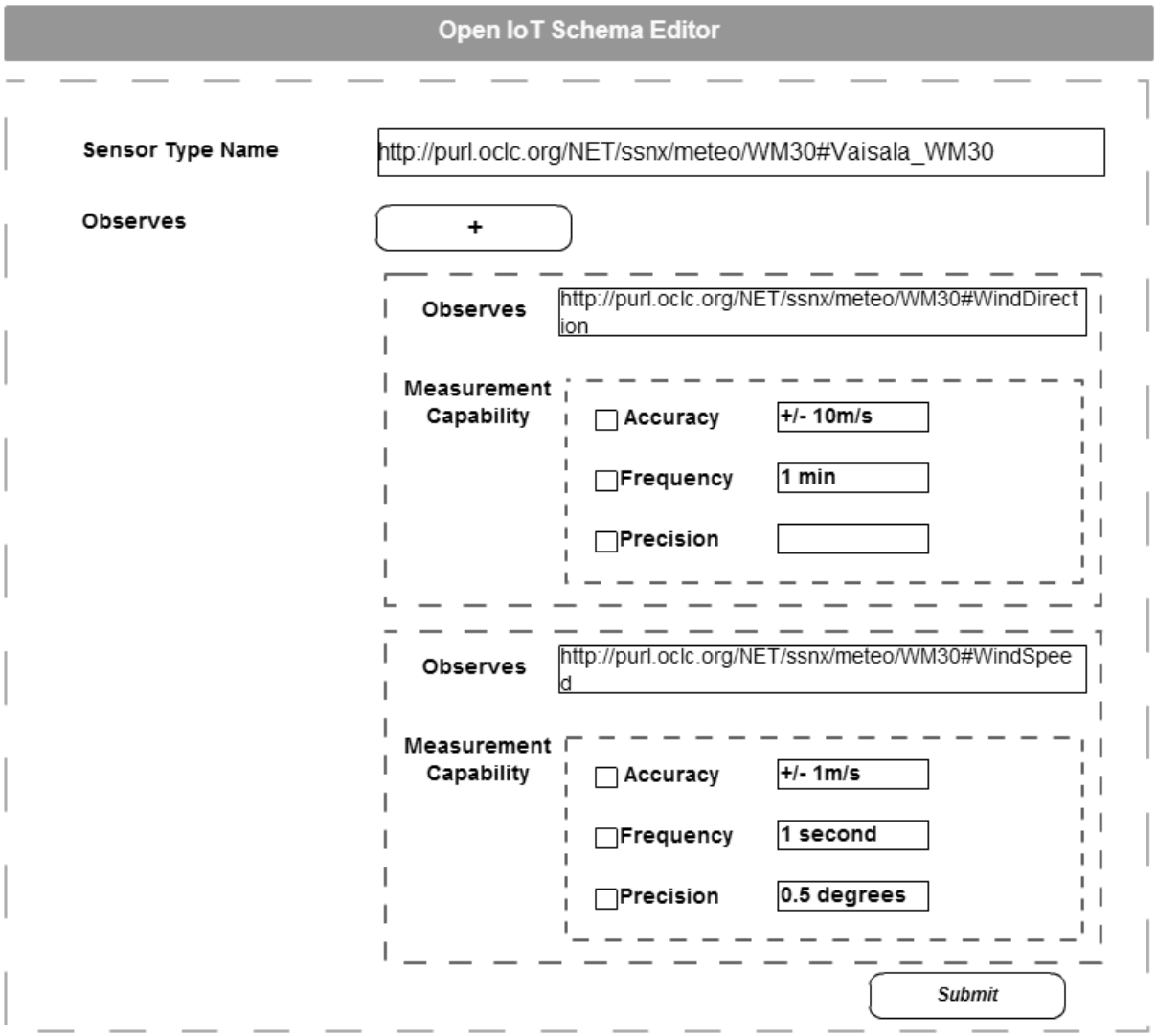


Figure 12: Sensor Schema Editor Interface [6]

Sensor Instance Editor

The sensor instance editor (see Figure 13) uses the sensor type definition created in the previous step using the sensor type editor to pre-populate an interface with sensor observations and properties. The user is then required to fill in the required field to create a sensor instance.

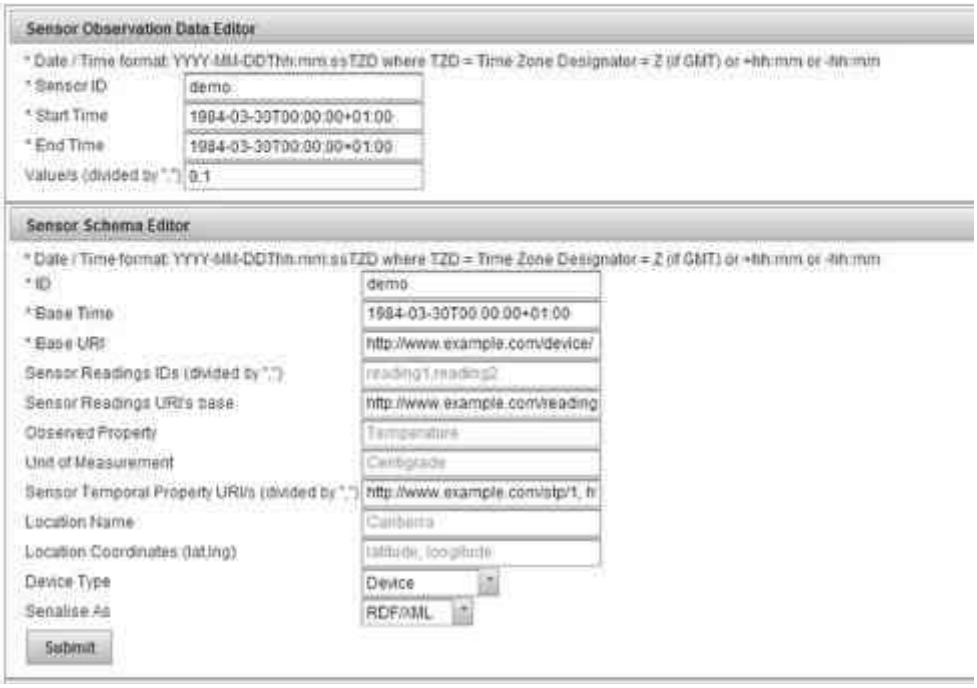


Figure 13: Sensor Instance Editor Interface [6]

3.1.2 VITAL

The VITAL project offers different portals each providing different functionalities. As shown in the VITAL architecture (see Figure 14), the three portals are:

Development and Deployment environment [7]

The VITAL development and deployment environment that contributes to the development, deployment and operation of IoT applications and services by providing developers with:

- A **tool** that they can use to compose the various capabilities that the VITAL framework offers, in order to implement smart city applications, and
- An **environment** where they can deploy the applications they build.

Figure 14 shows the position of the development and deployment environment in the VITAL architecture. The development and deployment environment is integrating the following functionalities provided by the VITAL framework:

- Data Management.
- Resource (i.e. internet Connected Object (ICO) and service) discovery.
- Complex Event Processing (CEP).
- Filtering.
- Business Process Management (BPM).

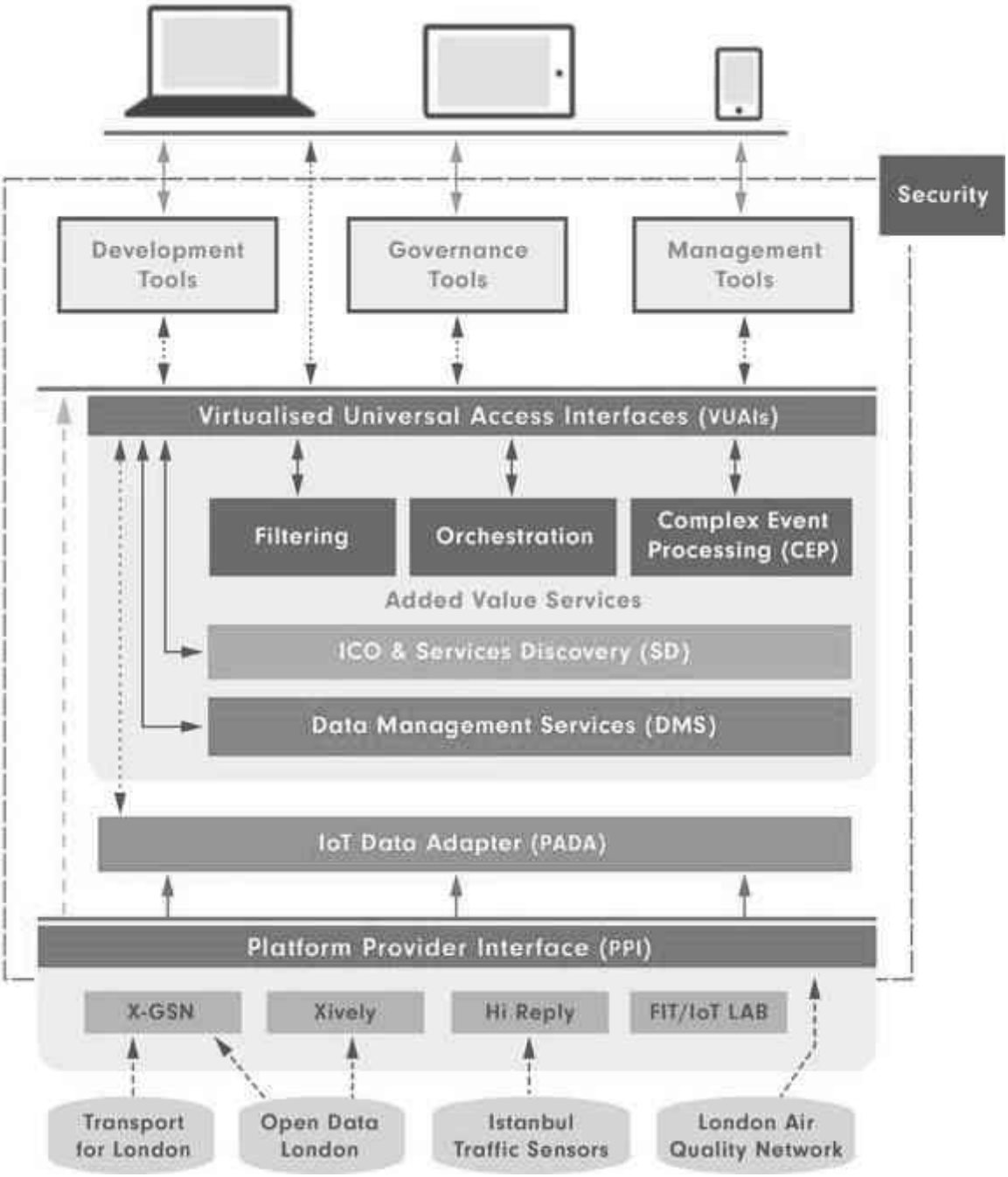


Figure 14: Position of development and deployment environment in the VITAL architecture [7]

Management and Governance tools [8]

VITAL provides to its connected applications a common view of the data & services provided by these underlying systems, unified using a common semantic data model. In a similar fashion the VITAL management layer provides a common management plane that unifies all the management information of all components and systems as well as management functionality that can be performed in a unified way to all parts of the VITAL ecosystem.

More specifically, the VITAL management layer:

- Supports the management of the core VITAL platform and all individual VITAL sub-systems (i.e. software components that are designed and developed by the VITAL project).
- Provides a common, unified management view (monitoring and control if allowed) of the different underlying IoT Systems. It will integrate with any management hooks provided by these systems and translate data and functionality into the common VITAL management model.
- Manages the relationships between VITAL components and subsystems. This is an extension of the monitoring (and control when possible) of the individual component/sub-system operational/health parameters to the monitoring of the interactions between them.

The VITAL management layer handles the following types of objects (VITAL managed objects):

- ICOs and data streams: these are the actual “low-level” ICOs that are typically connected to the VITAL platform via an underlying IoT System. The management layer provides operational information of the ICO basic parameters (including position and trajectory for mobile ICOs) as well as the availability and Quality of Service (QoS) (if available by the IoT systems) parameters of the related data streams. This information is provided by the IoT Systems’ management hooks.
- IoT Systems: these are the underlying IoT Platforms and Applications that are connected to the VITAL platform via the VITAL adapters (PPIs). Depending on what each system supports, the management layer retrieves, processes and uses information on the health, operational parameters, performance, accounting, security and even specific alerts emanating from the connected IoT system.
- VITAL Platform Components: these are the individual loosely coupled sub-systems/components of the VITAL architecture, i.e. the Complex Event Processing (CEP) module, the Service Discovery module, the Data Management layer, the Orchestrator module, etc. The management layer monitors and handles health information as well as data related to all the aspects of the FCAPS (Fault, Configuration, Accounting, Performance, Security) model (wherever applicable) for each component.
- Service Level Agreement (SLA): this refers to the SLAs set between the VITAL deployment and the connected Platform/Data Providers. The management layer will provide monitoring services as well as historical data on these SLAs as part of the Governance Module.
- Security information: this refers to all security related management data, i.e. data related to authenticated sessions and failed authentication or unauthorized access attempts by the applications operating on top of the VITAL platform. It could also extend to other security related information like authentication errors in the communication of VITAL with the underlying IoT Systems (if they support it).
- City-specific configurations: access to this information is realized via the VITAL Governance toolkit/module.

3.1.3 Machine-to-Machine Measurement Portal

The Machine-to-Machine Measurement (M3)[1] is a framework to assist developers in developing Semantic Web of Things applications [9] [10]. A screenshot of its portal is displayed in Figure 15.

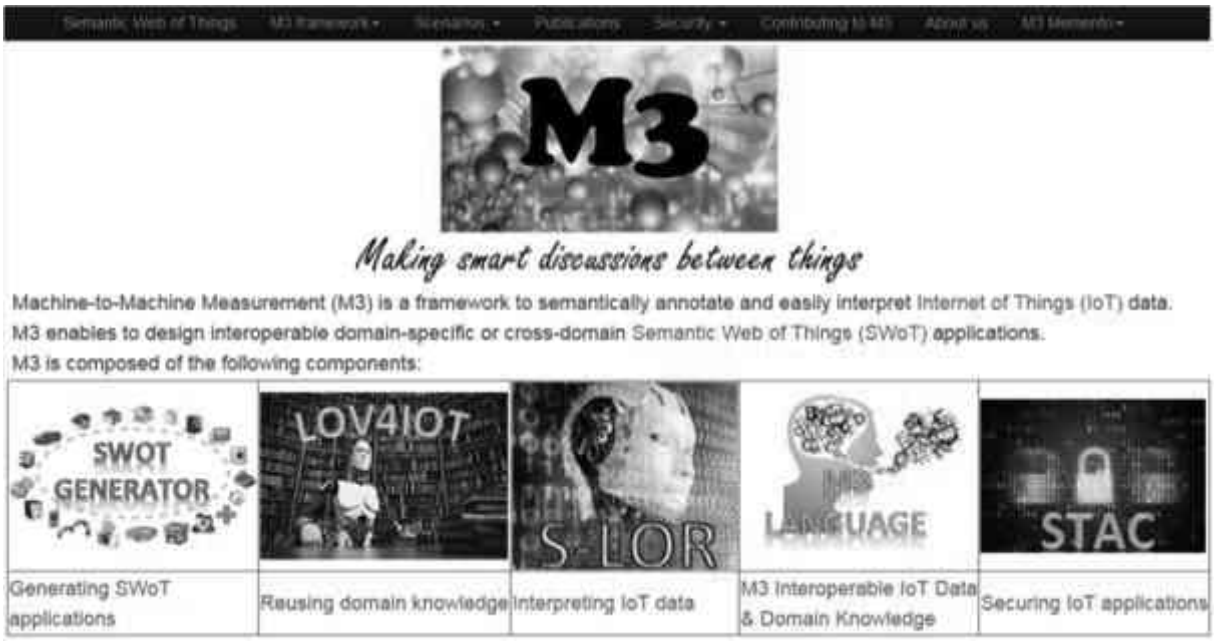


Figure 15: Machine-to-Machine Measurement (M3) portal

The M3 portal provides the following functionalities:

- An experiment generation component, called "**Semantic Web of Things (SWoT) generator**", produces experiment templates to assist the developers to ease their tasks in IoT development to unify heterogeneous sensor data, interpret data, and combine applicative domains. The experiment template contains a set of files to run the experiment: (1) a dictionary to semantically annotate data in a unified way, the dictionary is implemented in the M3 ontology, (2) a set of rules to deduce meaningful information from data, (3) domain knowledge to link sensor data to more knowledge bases to enrich data with additional information, and (4) a pre-defined query to get a subset of sensor data that developers are interested in [10].
- The semantic annotator component annotates data in a unified way thanks to the "**M3 language**" [11]. The semantic annotator is compliant with the M3 ontology (V1) or the M3-lite taxonomy (V2) explained in [12].
- The inference engine executes the dataset of interoperable rules to deduce meaningful information from semantic data. The dataset of interoperable rules is called "**Sensor-based Linked Open Rules (S-LOR)**" [13].
- A dataset and a set of APIs to retrieve domain knowledge is required to build smart and interoperable IoT experiments. The dataset called "**Linked open**

[1] http://www.sensormeasurement.appspot.com/
[2] http://sensormeasurement.appspot.com/?p=documentation

Vocabularies for Internet of Things (LOV4IoT)" referenced more than 300 IoT experiments using semantic web technologies [14][15]. The domain knowledge referenced within, has been manually redesigned to be interoperable with the semantic annotator, the reasoning engine and the query engine. Another benefit of having the interoperable domain knowledge is enabling building cross-domain experiments.

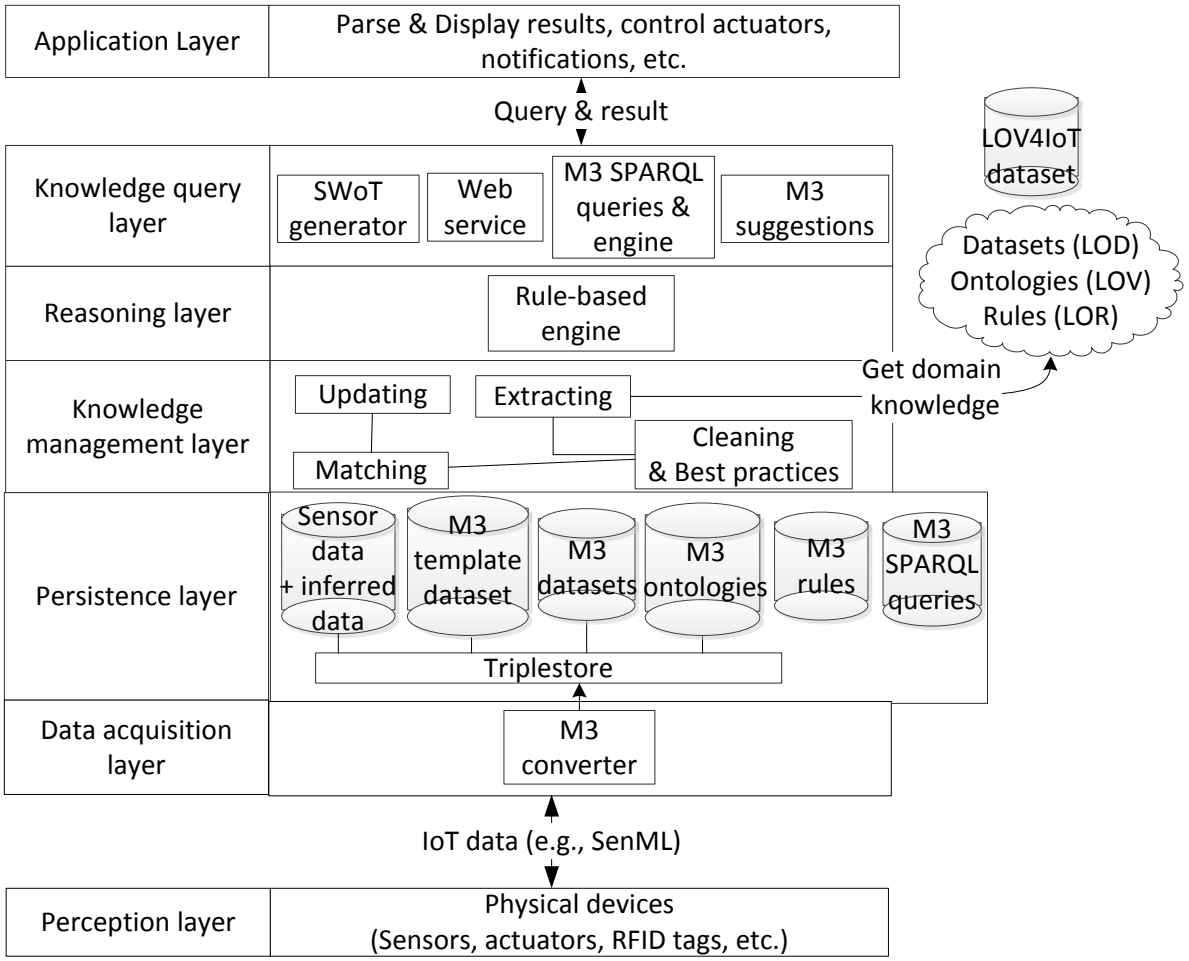


Figure 16: M3 portal architecture

The M3 framework comprises several layers as following (see Figure 16):

- The **perception layer** interacts with physical devices such as sensors, actuators and RFID tags to get their data and control them. In the implementation, this layer is compatible and based on sensor discovery that returns Sensor Markup Language (SenML) data. It could deal with other formats such as Sensor Web Enablement (SWE) and could get data generated by other tools such as Graph of Things.

- The **data acquisition layer** gets sensor metadata such as measurement type, unit, sensor type, value, and domain from M2M devices. An example of sensor data represented in the SenML format is: weather as the domain, cloud cover as the measurement type, 0 as the value, and Okta (a unit to measure the cloud cover) as the unit. Due to the heterogeneity of measurement descriptions, this layer converts sensor metadata in a unified description using semantic web technologies such as RDF/XML. In this layer, sensor metadata is semantically annotated according to the M3 nomenclature that has been implemented in the M3 ontology. For more details on M3 readers are referred to [12].
- The **persistence layer** stores the interoperable M3 domain knowledge (ontologies, datasets and rules), semantic sensor data and inferred sensor data. This layer provides, also, the M3 template dataset to retrieve the M3 domain knowledge to easily build SWoT applications. SPARQL queries have been also designed and are compatible with the M3 domain knowledge to assist developers in querying sensor data. Most of the datasets are stored in a triple store. M3 SPARQL queries and M3 rules are stored in files.
- The **knowledge management** layer is responsible for finding, indexing, designing, reusing and combining domain-specific knowledge (e.g., smart home, intelligent transportation systems, etc.) such as ontologies and datasets to update M3 domain ontologies, M3 datasets and M3 rules which are structured in an interoperable manner. M3 Portal builds a dataset to refer, classify and reuse domain-specific knowledge (LOV4IoT). LOV4IoT is a knowledge base composed of ontology-based projects relevant for IoT. These projects are based on semantic web technologies and provide domain ontologies, datasets and rules that could be theoretically reused to design domain-specific or cross-domain SWoT applications.
- The **reasoning layer** infers high-level knowledge using reasoning engines and M3 rules stored in the persistence layer. M3 rules are extracted from the LOV4IoT dataset and are redesigned to be interoperable with each other. M3 rules are a set of rules compliant with the M3 ontology to infer new knowledge from M3 data. For instance, when the cloud cover is equal to 0 okta, M3 rules can deduce that the sky is blue. This layer produces smarter M3 data.
- The **knowledge query layer** loads M3 ontologies and related datasets. Then, it executes SPARQL queries to provide M3 suggestions. For instance, this layer can suggest activities according to the weather. Indeed, activities are related to weather concepts in tourism and weather datasets.
- The **application layer** employs an application (running on smart devices) that parses and displays the results to end-users. For instance, the M3 framework suggests activities according to the weather forecasting (e.g., catamaran when it is windy). Other treatments can be achieved in this layer such as controlling actuators, sending alerts, etc.

Documentation is provided in the form of numerous publications[2],[3] and tutorials for the easy interaction with the portal and the use of different tools. For instance, a java code skeleton[4] is provided, full of comments to ease the tasks of experimenters to interact with the functionalities and tools provided by the M3 portal.

3.1.4 FED4FIRE

As a successor of the OpenLab[5] project, the Federation for FIRE[6] (Fed4FIRE[7]) project was started in 2012. Fed4FIRE focused on federating European facilities by unifying existing experimentation and management tools and procedures. It was identified that federated testbeds should support all the functions of the experiment life cycle: resource description, resource discovery, resource requirements, resource reservation, resource provisioning (direct or orchestrated), experiment control, facility monitoring, infrastructure monitoring, experiment measuring, permanent storage, and resource release. They should additionally support federated identity management, AuthZ, and SLA management. Analogous to Global Environment for Network Innovations (GENI[8]), Slice-based Federation Architecture (SFA) has been adopted for resource discovery and provisioning; the Federated Resource Control Protocol (FRCP) based tools: cOntrol and Management Framework (OMF) and Network Experimentation Programming Interface (NEPI[9]), for experiment control; and Orbit Measurement Library (OML) Measurement Stream Protocol (OMSP), for transporting monitoring information about experiments, infrastructure and facilities. Furthermore, different possible federation architectures were evaluated. Based on identified characteristics, the heterogeneous federation approach was recognized to be the most suitable for the future Internet experimentation facilities under evaluation. In this architecture, all testbeds run their own native testbed management software.

[2] http://sensormeasurement.appspot.com/?p=documentation
[3] http://sensormeasurement.appspot.com/?p=publication
[4] http://sensormeasurement.appspot.com/?p=end_to_end_scenario
[5] http://www.ict-openlab.eu/project-info.html
[6] https://www.ict-fire.eu
[7] http://www.fed4fire.eu
[8] https://www.geni.net
[9] http://nepihome.org

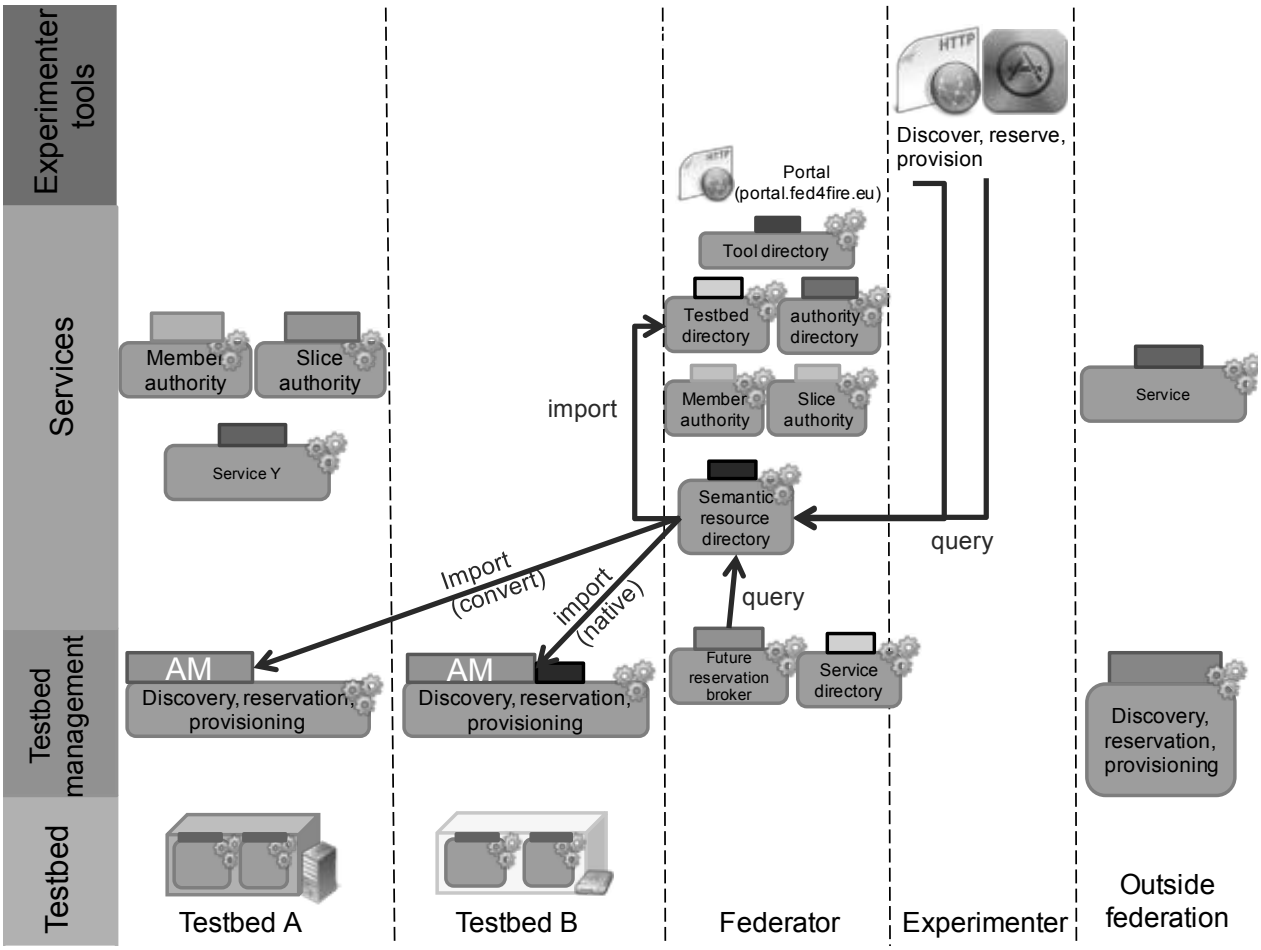


Figure 17: Fed4FIRE Overall Architecture [16] **(based on** [17]**)**

As shown in Figure 17, the portal plays a major role in the overall Fed4FIRE architecture. It includes in particular directories for tools, testbeds, and authorities. As this information is encoded in different ways (HTML and different XML Schemas), it is accompanied by a preliminary semantic version. However, the approach taken by Fed4FIRE is manifold:

- As the first and most important entry point for users, http://www.fed4fire.eu is a set of static HTML pages, which describes the project, how to execute an experiment, the different testbeds, the available tools, and how to add a new facility. It further links to testbed specific documentation.
- Technically, for resource discovery, selection, reservation, provisioning and release, most involved testbeds must support the XML Remote Procedure Calling protocol (XML-RPC) based Slice-based Federation Architecture (SFA), which allows federation across facilities using Transport Layer Security (TLS) based AuthN. As a result, these testbeds can be controlled by various SFA-compliant user tools/portals:

- **MySlice**[10] **(user GUI):** An HTML-based portal deployed within Fed4FIRE[11] to graphically create slices, manage users, link to testbed documentation and join experiments (see Figure 18). In Figure 18 one module of the GUI is shown, which depicts resources available within Fed4FIRE from different testbeds. The list can be filtered by different attributes and finally resources can be reserved and provisioned for further usage (often via SSH, partly via OMF or NEPI – see below). Its functionality can partly be compared with the jFed[12] tool and both tools are tested to work with each other.
- **Flack**[13] **(client GUI)**: developed within ProtoGENI[14]. A Flash-based GUI used for user-friendly selection and provisioning of resources.
- **Omn**[15] **(client Command Line Interface (CLI))**: developed as a collaborative GENI effort. A python-based reference implementation that is also part of the GENI Control Framework (GCF) software package.
- **jFed (client GUI and CLI)**: A Java-based framework that contains automated testing tools, a probe for manual testing and an end-user experimentation-toolkit. In Figure 19, an artificial example is shown in which multiple instances of virtualized interconnected "motors" have been provisioned for an experiment. jFed can also be used to perform automated scenario tests. While not the only one, within Fed4FIRE this is the most prominent and most often used user tool for executing experiments within Fed4FIRE testbeds.
- **SFA Command-Line Interface (client CLI)**: called SFI and developed within PlanetLab[16] and in its core used within MySlice (see Figure 18).

[10] http://myslice.info/
[11] https://portal.fed4fire.eu
[12] http://jfed.iminds.be/
[13] http://protogeni.net/wiki/Flack
[14] http://www.protogeni.net
[15] http://trac.gpolab.bbn.com/gcf/wiki/Omni
[16] https://www.planet-lab.org

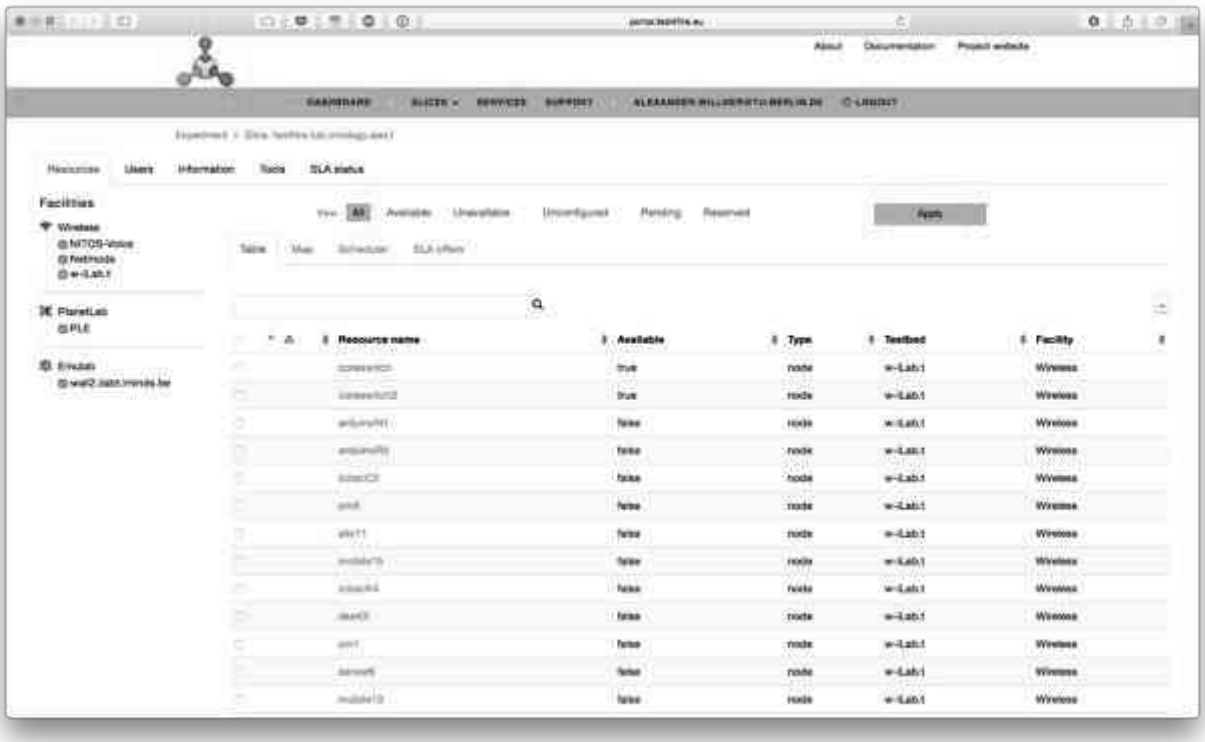


Figure 18: Fed4FIRE Portal based on MySlice

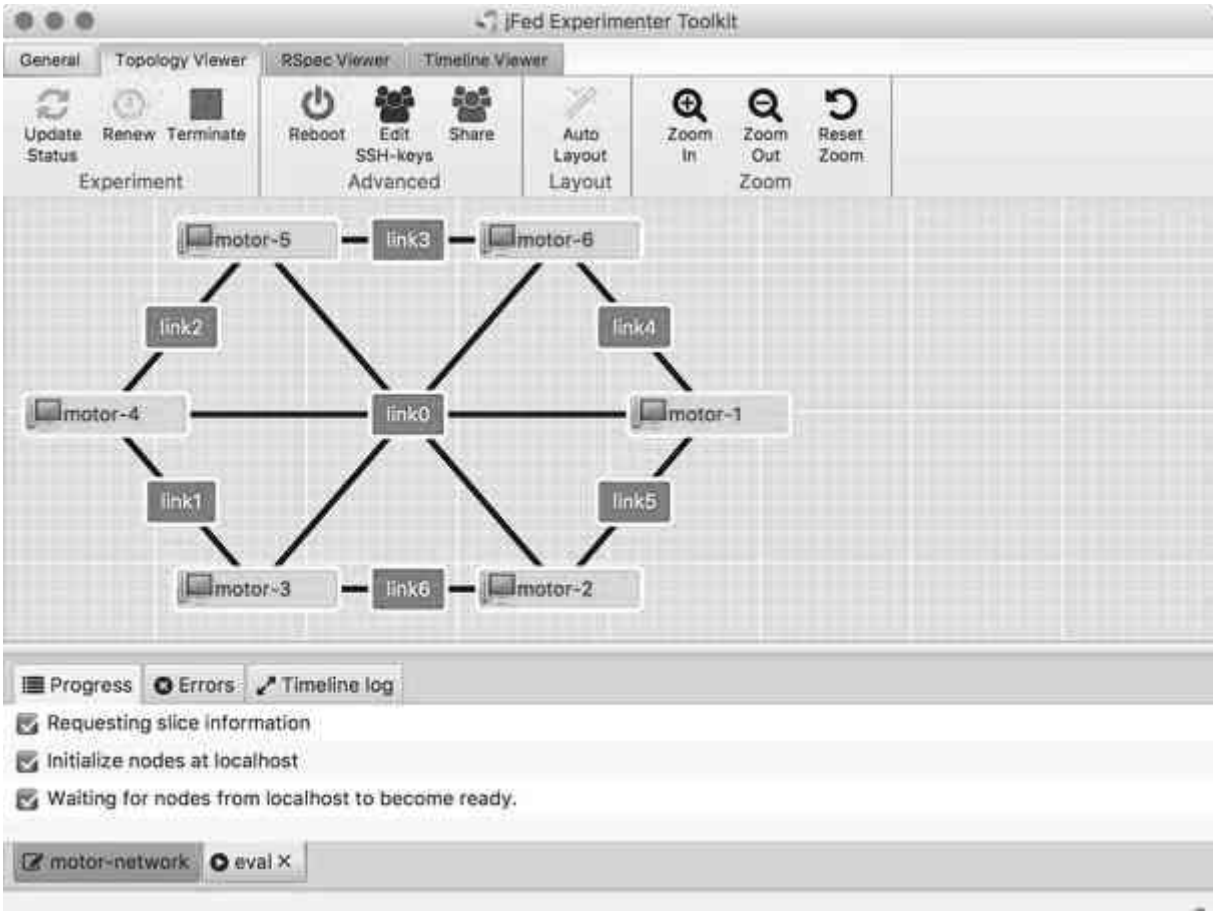


Figure 19: jFed Experimenter GUI [16]

- While SFA mainly addresses issues regarding the description, discovery and provisioning of resources, mechanisms are also needed to describe experiments and to control and monitor resources accordingly. As a result, two experimentation tools have been adopted within Fed4FIRE to execute reproducible experiments:
    - The cOntrol and Management Framework (OMF): Workflows are described using the OMF Experiment Description Language (OEDL), a Domain Specific Language (DSL) based on Ruby. An experimenter using an Experiment Controller (EC) to send the related messages and commands to a Resource Controller (RC). The communication is based on the Federated Resource Control Protocol (FRCP).
    - The NEPI, allows complex experiments to be described and executed, has been enhanced to be compatible with FRCP. NEPI uses its own DSL to define workflows and supports combining resources from three different types of experimentation platforms: simulators, emulators, and real testbeds.
- To reduce complexity, later within the Fed4FIRE project, new testbeds were able to join a so called light-federation. This allowed testbeds to offer other interfaces, such as REST-based API's, as long as the same X.509-based AuthN mechanisms are supported. As a result, other user facing tools have been established. One example is YourEPM[17], a tool focused on the orchestration of Fed4FIRE Application Services. This tool has been developed to let experimenters define and execute complex experiments involving Application Services in the federation by using YourEPM's underlying Activiti Business Process Management (BPM) engine. Activiti[18] is a light-weight workflow and BPM Platform targeted at business people, developers and system admins. YourEPM extends Activiti and provides integration with Fed4FIRE Speaks-for credentials, authentication via GENI Authorization Tool, multi-tenant functionality and integration with Fed4FIRE Application Services via Service Directory and RESTful invocation of external services.
- Additionally, the Open-Multinet (OMN) OWL-based ontologies have been developed and used to semantically describe GENI and FIRE federations of infrastructures, each testbed, and their resources including their status (cf. **Figure 20**). In the centre the Fed4FIRE federation is visualized as a bubble, connected to it different legal entities (such as the Fraunhofer FOKUS), which in turn administer testbeds (such as the Fraunhofer FUSECO Playground). These testbeds offer different interfaces that can be used to discover and provision the according resources (such as the SFA AMv3 API). They can be used within each phase of the experiment life cycle (describe, publish, discover, select, reserve, provision, monitor, control, and release). The ontologies incorporate existing work such as the Open Grid Forum (OGF) Network Mark-Up Language (NML), the Infrastructure and Network Description Language (INDL) or the Network Description Language based on the Web Ontology Language (NDL-OWL) and reuse existing ontologies such as Dublin Core (DC), Friend of a Friend (FOAF), Good Relations (GR), World

[17] http://www.fed4fire.eu/yourepm/
[18] http://activiti.org/

Wide Web Consortium (W3C) Geo and W3C Time. A demonstrator can be found at http://lod.fed4fire.eu. It can answer questions such as “Show me all 802.11 compatible nodes close to the Acropolis”, based on testbed information operated within FIRE and GENI.

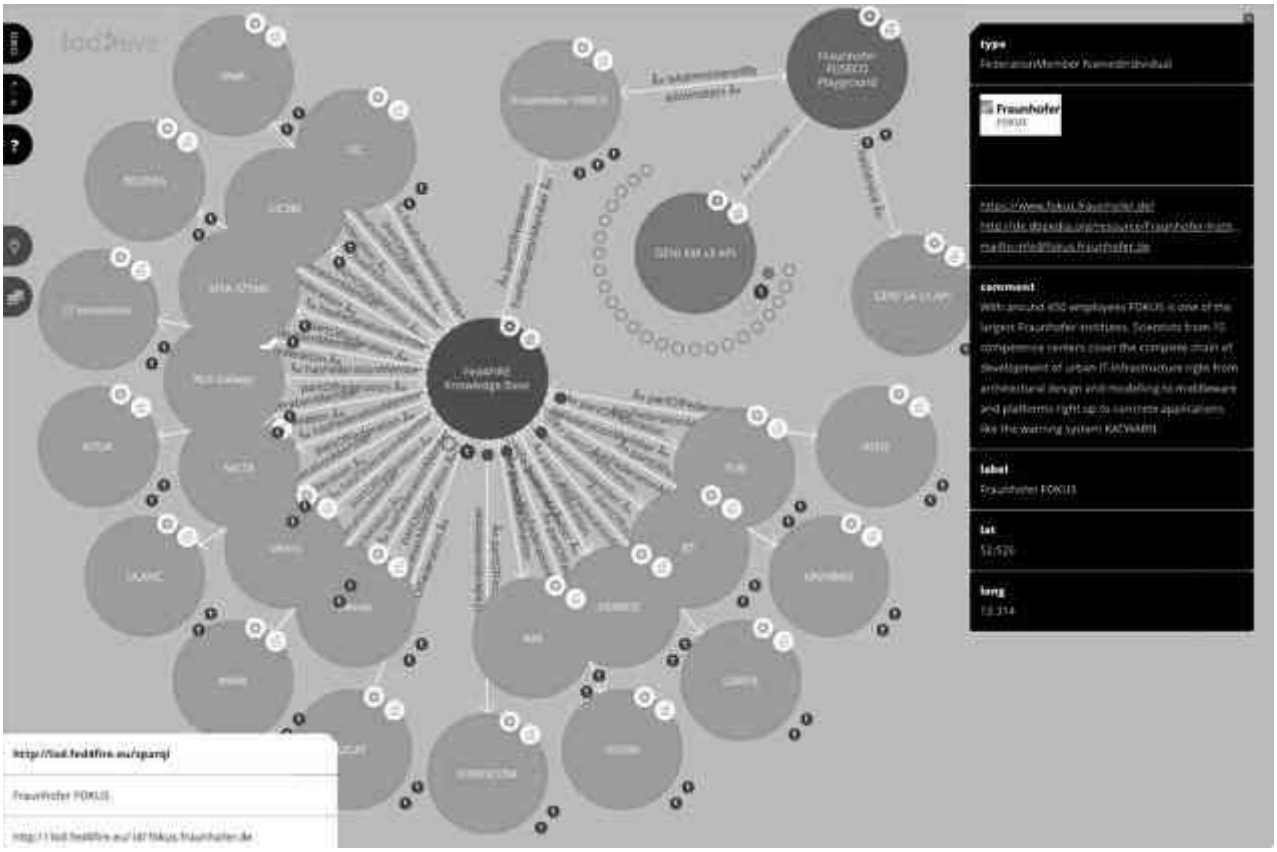


Figure 20: Graphical representation of the semantic-based Fed4FIRE directory [16]

3.2 Existing Tools to build Experimentation portal

Below we provide existing tools that are available towards facilitating the building of the web portal.

3.2.1 Zkoss

Zkoss also known as ZK is a leading enterprise Ajax framework and the easiest way to build great modern Java web applications. ZK is an open-source Ajax Web application framework written in Java that enables creation of graphical user interfaces for Web applications. ZK framework provides:

- Excellent end-to-end productivity boost over other frameworks
- Fast development environment and on-the-fly changes to the UI
- Industry leading Server and Client Architecture offers two methodologies for implementing the UI of applications, in pure Java code or in XML markup
- Event-driven model

- Authentication and authorization with third party security frameworks such as Spring Security

ZK is an open-source UI framework that enables you to build amazing web and mobile applications. Each UI object consists of two parts, a component and a widget[19]. The first one is a Java object that is running at the server side and does not involve a visual part. It represents an UI object that can be handled by a Java application. A widget, on the other hand, is JavaScript-based object interacts with the user, handles several client-side events, it has a visual part and it is running therefore on the client side. The two parts are cooperating in order to provide a fully interactive user experience.

The core of ZK is composed basically of a mark-up language and an Ajax-based event-driven mechanism, over 123 XUL and 83 XHTML-based components. In order to generate the user interfaces designers/programmers use different markup languages in the same file seamlessly, such as feature-rich XML User Interface Language (XUL) and XHTML components. Overall, ZK supports ZK User Interface Markup Language (ZUML) that is a mark-up language for enriched user interface design. ZK downloads only a JSON description of the web page and renders it then into a UI at the client side by managing many client-browser and browser-client requests. The latter is exactly what makes it efficient, since there is no need to get everything at once but create it gradually.

Another important feature of ZK is that the engine automatically performs the event pipelining between clients and servers, and the content synchronization. ZK does not work with the request-response mechanism, such as `HTTP GET` or `HTTP POST` requests. Instead, Ajax requests are sent to the server to update the internal state of each screen widget and the Ajax code is transparent to the programmers. This allows the end users to experience the responsiveness and interactivity of a desktop application. Moreover, the server-client fusion that ZK supports, an optional client-side customization, lets the developer take advantage of the client-side resources to perform customizations and reduce Ajax traffic.

[19] https://www.zkoss.org/wiki/ZK_Getting_Started/Learn_ZK_in_10_Minutes

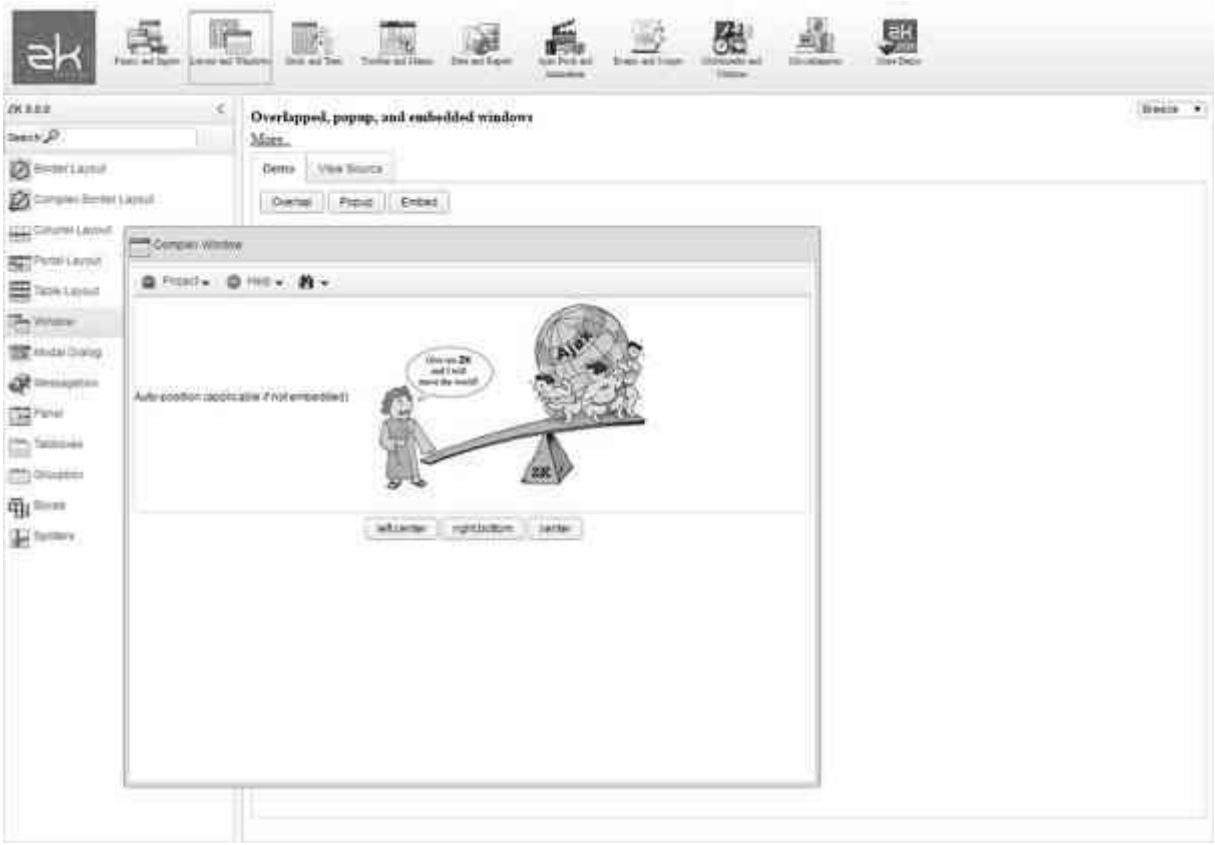


Figure 21: ZK sample screenshot demonstrating a floating window[20].

3.2.2 Google Polymer

Google Polymer[21] is a framework provided by Google Inc. for the development of the dynamic web applications. It is an open source JavaScript library and follows modern design principles. The main features of Google Polymer include data-binding elements, custom elements, computation properties, gesture events and component libraries. Polymer includes various elements to facilitate the building of the web interface. These elements mainly are: App elements, Core elements, material design elements, components for Google's APIs and services, Animation elements, offline, push elements and wrappers for third party libraries. Google Polymer is currently being used in various companies to promote their products[22]. One main drawback of Google Polymer is that it is heavy. Other drawbacks of Google Polymer include learning curve, its installation is not one click and one has to install several dependencies in order to fully install it. Google polymer is still evolving thus has many issues[23] and sometimes lacks[24] documentation. An online tool[25] to develop web applications based on Google Polymer is also available.

[20] https://www.zkoss.org/zksandbox

[21] https://www.polymer-project.org/1.0/
[22] https://github.com/Polymer/polymer/wiki/Who's-using-Polymer%3F
[23] https://github.com/Polymer/polymer/issues
[24] https://github.com/Polymer/docs/issues
[25] https://polymer-designer.appspot.com

A lightweight version of Google Polymer also exists and is called Material Design Lite[26]. It does not rely on any JavaScript frameworks like AngularJS[27] and NodeJS[28] that Google Polymer uses. It supports only limited components like Badges, Buttons, Cards, Chips, Dialog box, Layout, Lists, Loading, Menu, Slider, Snackbar, Toggles, Tables, Text Field and Tooltips. Thus it is good to build web applications that use limited functionalities. Nevertheless, the Material Design Lite lacks few essential components needed for the web development such as “select”.

3.3 Existing tools to build needed Experiment related Components

In this sub section, we provide a brief overview of out-of-the-box tools in order to build experiment-related components. These components basically include Experiment modeling component, Experiment Execution Engine and Experiment Management Console. In [3], we reported tools available to build experiment DSL and reported on Experiment Registry Management component (ERM) that would facilitate the building of experiment and its storage. The specific requirements that were supported by the Experiment Modeling component were also reported. Nonetheless, in this deliverable as we focus on EEE and graphical tool to facilitate building of the Experiment, in this sub-section we report on state of the art tools that are available in order to address the stated requirements.

3.3.1 Tools towards building EEE

Available state of the art schedulers best achieve the above requirements, where EEE’s main objective is to execute the set of defined FISMO’s associated to an experiment on the Meta Cloud. Different programming environments support different schedulers. For example for Python some of the job schedulers available are APSchedular[29], Django-Schedule[30], doit,[31] Gunnery[32], joblib[33], Plan[34], Schedule[35], Spiff[36] and TaskFlow[37]. As we focus on Java as the programming environment, we refrain ourselves in providing details on them.

Nevertheless, besides above-mentioned schedulers, many highly advanced schedulers such as Apache Aurora[38], Chronos[39], Apache ServiceMix[40], JBOSS Fuse[41], and Quartz[42] Job Scheduler exist. A description of Apache ServiceMix,

[26] https://getmdl.io
[27] https://angularjs.org
[28] https://nodejs.org/en/
[29] http://apscheduler.readthedocs.io/
[30] https://github.com/thauber/django-schedule
[31] http://pydoit.org/
[32] https://github.com/gunnery/gunnery
[33] http://pythonhosted.org/joblib/index.html
[34] https://github.com/fengsp/plan
[35] https://github.com/dbader/schedule
[36] https://github.com/knipknap/SpiffWorkflow
[37] http://docs.openstack.org/developer/taskflow/
[38] http://aurora.apache.org
[39] http://mesos.github.io/chronos/
[40] https://servicemix.apache.org/
[41] http://www.jboss.org/products/fuse/overview/
[42] https://www.quartz-scheduler.org/

JBOSS Fuse, and Quartz Job Scheduler is made available in [18] and thus we also refrain ourselves in again describing them here. In this section we briefly describe Apache Aurora and Chronos mainly because they are advanced schedulers.

Apache Aurora is a framework to schedule long lasting processes. Twitter originally developed it and then made it open source. Aurora features include scheduling, crash management, rolling updates and rollbacks, multiuser support, service discovery, SLA Metrics and webhooks to name a few. Aurora is typically used in large platforms that operates over thousands of machines and has numerous services. Aurora is based on Mesos and has rich DSL (also known as service configuration file) that describes job, task and process parameters. The Aurora DSL is python based and is defined using Pystachio[43] tool (a tool used to define specification). Aurora follows a lifecycle as defined in Figure 22. Aurora has dependencies of Apache Zookeeper, Apache Mesos and JDK.

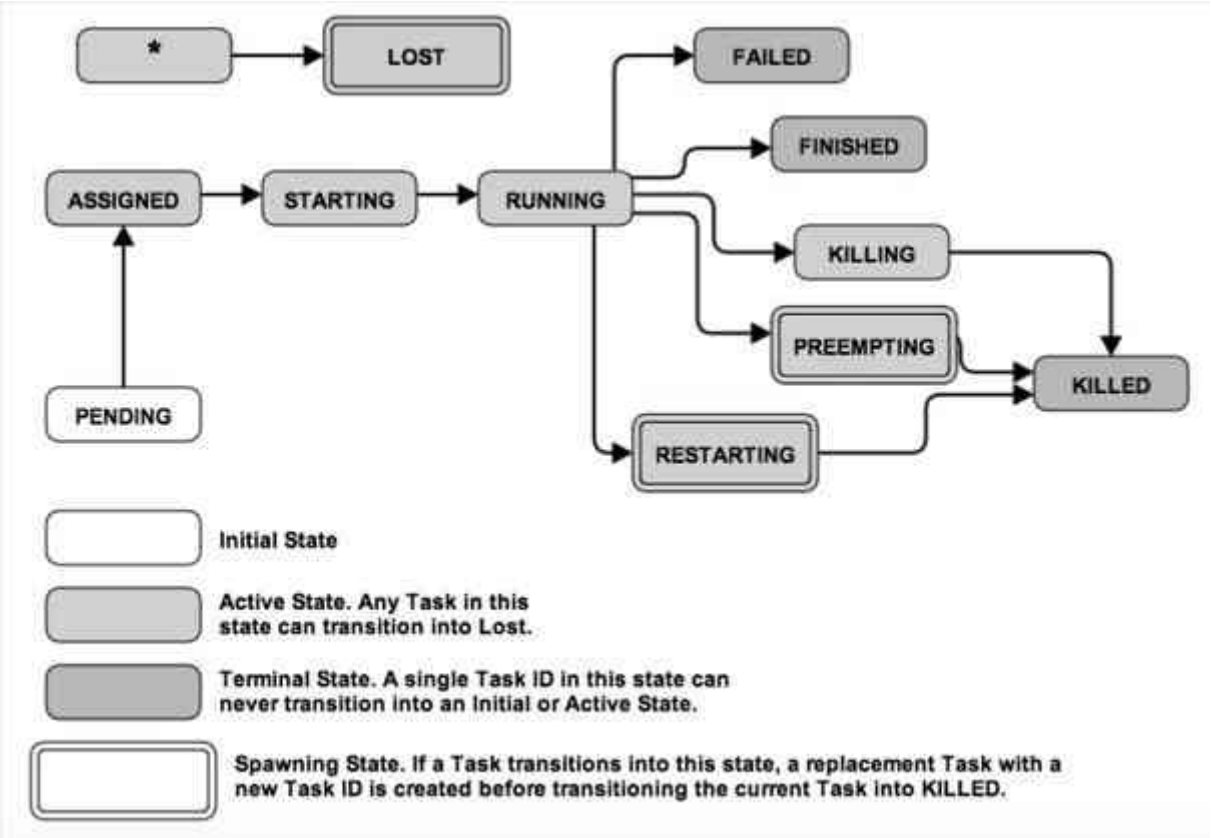


Figure 22: Aurora Job lifecycle[44]

Chronos is another process scheduling tool based on Mesos and can be also largely scalable. Its main features include fault tolerance and job orchestration by supporting arbitrary long dependency chains. As stated, Chronos "allows to schedule jobs using ISO8601 repeating interval notation". Chronos supports REST API for listing, deleting, manually starting, adding, adding dependent job, and many more. Chronos requires Apache Mesos, Apache Zookeeper and JDK.

[43] https://github.com/wickman/pystachio
[44] http://aurora.apache.org/documentation/latest/reference/task-lifecycle/

It is also essential for the process scheduler, once deployed, to provide information to the system administrator. Usually an easy to use and understand UI accompanies the process schedulers. With the UI available, system administrators can know which processes/jobs are scheduled, what is the status of the processes/jobs, define scheduling interval and other metadata, and for some advanced schedulers know the resource utilization. Figure 23 and Figure 24 show the UIs displayed for both Aurora and Chronos.

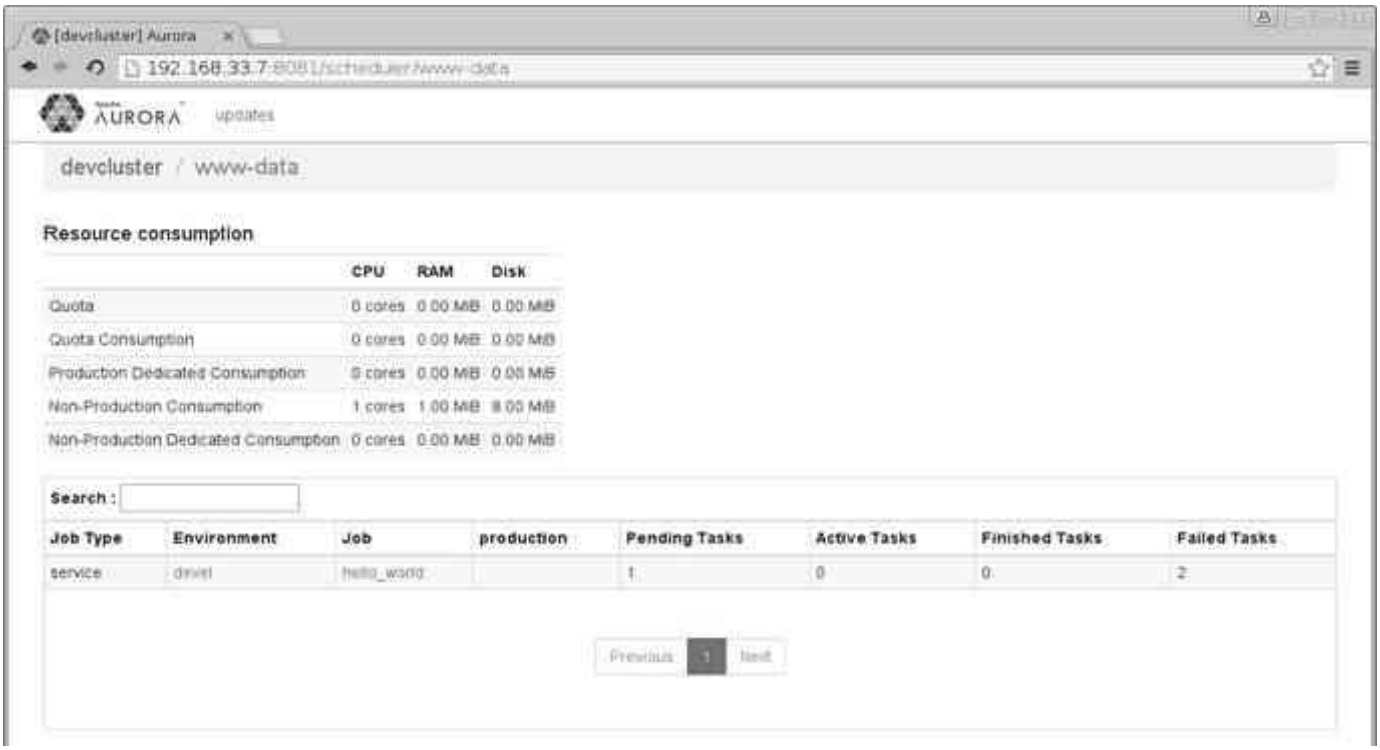


Figure 23: Apache Aurora Interface[45]

[45] http://aurora.apache.org/documentation/latest/getting-started/tutorial/

Figure 24: Chronos Interface[46]

3.3.2 Tools towards building Experiment Editor

There are many editors available that would facilitate the building of the DSL instance. As the DSL instance is an XML, available XML editors can be used such as Oxygen[47], Nodepad++[48], SublimeText[49], Atom[50], etc. However, there are many UI tools available that would facilitate the building of an experiment workflow and would produce needed experiment DSL instance. These mainly are Google Blockly and Node-RED as identified in [3]. In this section we provide in depth analysis of the Node-RED as an experiment Editor.

3.3.2.1Node-RED

Node-RED[51] is an open-source[52] JavaScript based tool for wiring the Internet of Things, created by the Emerging Technology team of IBM. One of its creators, David Conway-Jones, noted in EclipseCon 2014 that Node-RED is for events, what a word

[46] https://mesos.github.io/chronos/img/chronos_ui-1.png
[47] http://oxygenxml.com
[48] https://notepad-plus-plus.org
[49] https://www.sublimetext.com
[50] https://atom.io
[51] http://nodered.org
[52] Distributed under the Apache 2.0 license.

processor is for words, a spreadsheet is for numbers, and a presentation is for ideas: a tool that allows users to coordinate them [19] .

Nodes and flows are the two fundamental concepts in Node-RED. A node is a well-defined piece of functionality. Based on the number of its input and output ports, a node can be of one of the following types:

- an input node, which is a node that has one or more output ports
- an output node, which is a node that has one input port
- a function node, which is a node that has one input port and one or more output ports

Nodes can be wired together into what is called a `flow`. Input nodes sit at the start of a `flow`, output nodes sit at the end of a `flow`, and finally function nodes sit in the middle of a `flow`. `Flows` can be considered as programs, and nodes as the blocks that can be used to build them.

The Node-RED platform comprises two components: (1) a browser-based editor that allows us to design `flows` (i.e. programs), and (2) a light-weight runtime (information about the runtime is made available in Appendix III) where we can deploy and execute our `flows`. Finally, Node-RED can be run as a standalone application, but it can also be embedded into another application.

3.3.2.1.1 Node-RED editor

Node-RED provides a browser-based flow editor for creating flows and deploying them on the Node-RED runtime. Figure 25 shows how the Node-RED editor looks like.

On the left side of the editor is the palette of nodes that we can use to build our `flows`. Node-RED comes with an initial set of nodes that includes, among others, nodes that make HTTP requests, that convert messages in Comma Separated Values (CSV) format to JavaScript objects, that execute system commands, that tweet messages, and nodes that perform sentiment analysis.

In order to create a `flow`, we drag the nodes we need from the palette, drop them into the workspace layout (located in the centre of the editor), set their properties, and wire them together. In order to have more space to work, Node-RED supports multiple, tabbed workspaces. Once we have created our `flows`, we can deploy them to the Node-RED runtime, using the `Deploy` button located at the top right corner of the editor.

In Node-RED we can also define `subflows`, that can be thought of as node templates that can be re-used. `Subflows` can be created, edited and deleted. Each `subflow` is represented as a single node in the workspace. Once created, it appears in the palette, and we can drag and drop it into the workspace as many times as we need, creating a new instance of it each time. Similar to a node, a `subflow` has zero or one input and zero or more outputs.

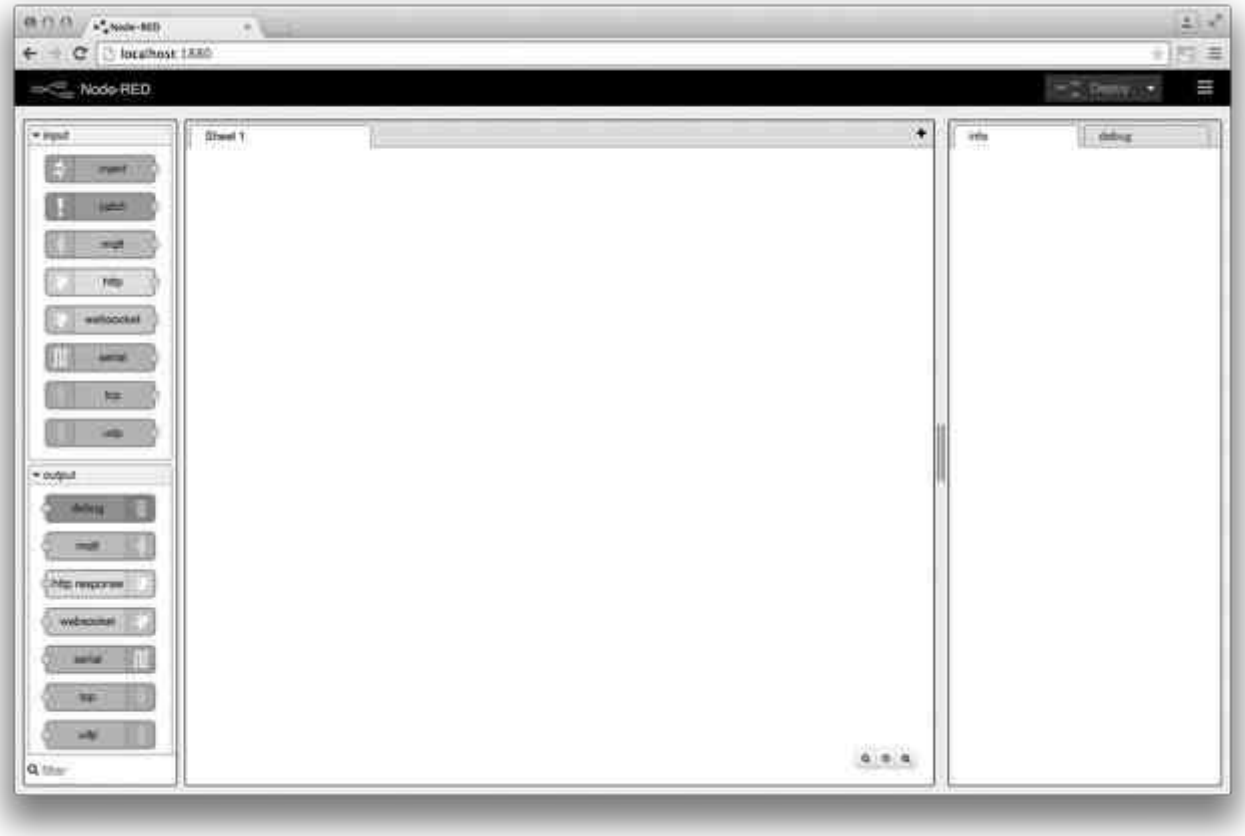


Figure 25: Node-RED flow editor.

`Flows` in Node-RED are represented and stored using JSON. This makes flows re-usable, since they can be easily imported and exported. Node-RED provides a built-in library to import flows from and export flows to. We can also share any flows we have created in the Node-RED Library[53], so that they can be re-used by other users. A description of some core nodes available in Node-RED is made available in Appendix I.

3.3.2.1.2 Extending Node-RED

The Node-RED palette can be extended with new nodes. We can search for new nodes to install in the Node-RED Library, as well as in the `npm` repository (as packages with the keyword `node-red`). Nodes that read and write to MongoDB, or nodes that interact with a Pibrella Raspberry Pi add-on board are some of the nodes that we can find there.

Another way to extend the node palette is by creating our own nodes. Each node is defined in a pair of files: a JavaScript file that defines what the node does (i.e. its runtime functionality), and an HTML file that defines how the node appears in the editor and its help text. A pair of JavaScript and HTML files actually defines a `node set` that contains one or more nodes. Nodes can be packaged as `npm` modules, and then published to the npm repository. Next we provide an example of extentions that can be done with respect to FIESTA-IoT.

[53] http://flows.nodered.org

3.3.2.1.2.1 Discovery of resources

In Node-RED, the discovery of resources can be supported by using a series of nodes such as: `start node/inject node`, `testbed node` and `resourcePanel node` as shown in Figure 26. In the discovery process, a user would be limited to only discover the resources that are registered by the testbeds (i.e. within the IoT Service/Resource Registry). The resources can be viewed at the "`edit testbed node`" popup box of the "`testbed`" node as shown in the Figure 27. Once the experimenter does the edition of the "`testbed`" node, he/she is required to interact with the Node-RED GUI and connect the nodes. In this case, the "`start process`" node with the "`testbed`" node and the "`testbed`" node with the "`resourcePanel`" node. The "`start process`" node sends a value "`true`" to the `testbed` node. Once the "`testbed`" node receives this input, it passes a predetermined API of the selected testbed to be called to the "`resourcePanel`" node. Once the "`resourcePanel`" node receives a valid API, it sends a `HTTP Request` to that API. The result (discovered resources) could then be added to the palette as shown in the Figure 28. However, as this is contradicting with one of the objectives of FIESTA-IoT, we would restrict such functionality.

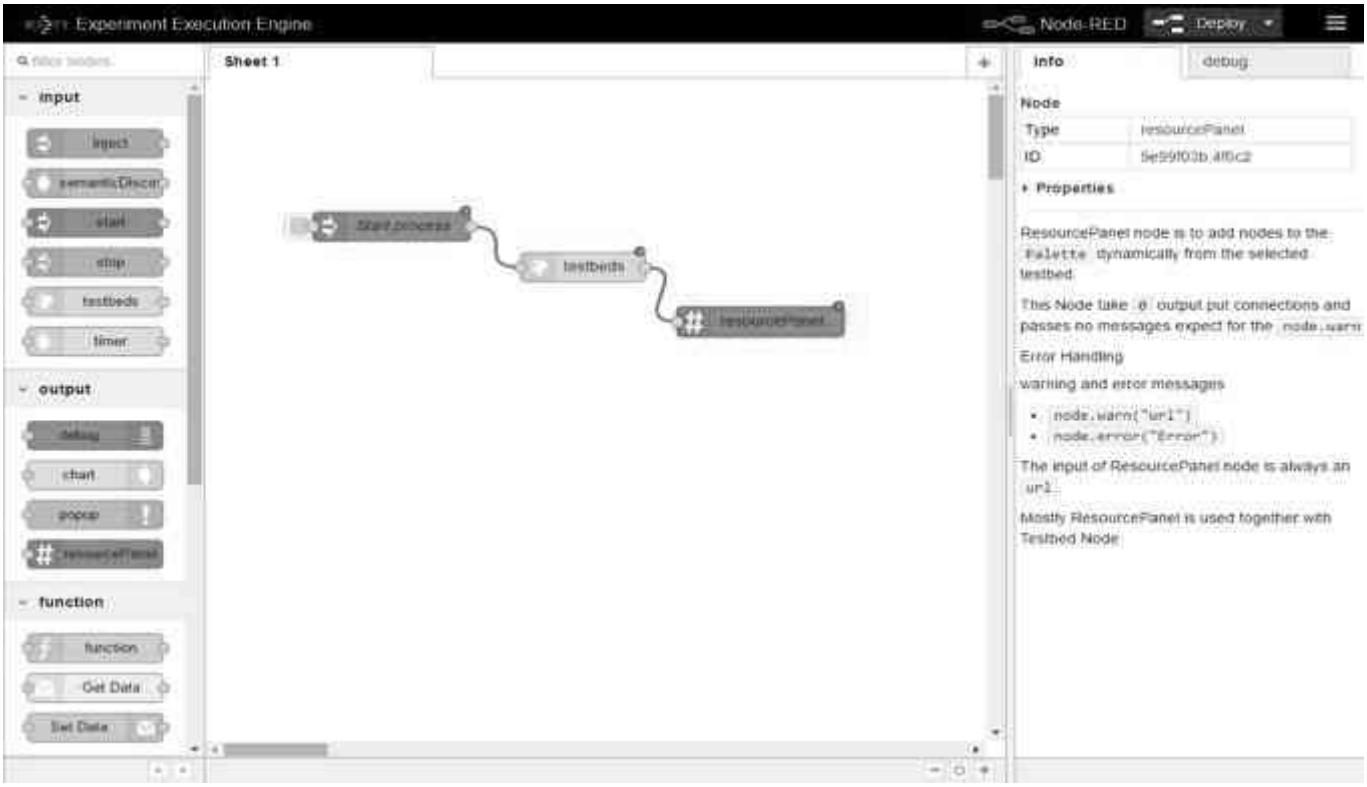


Figure 26: Discovery of the Testbed resources

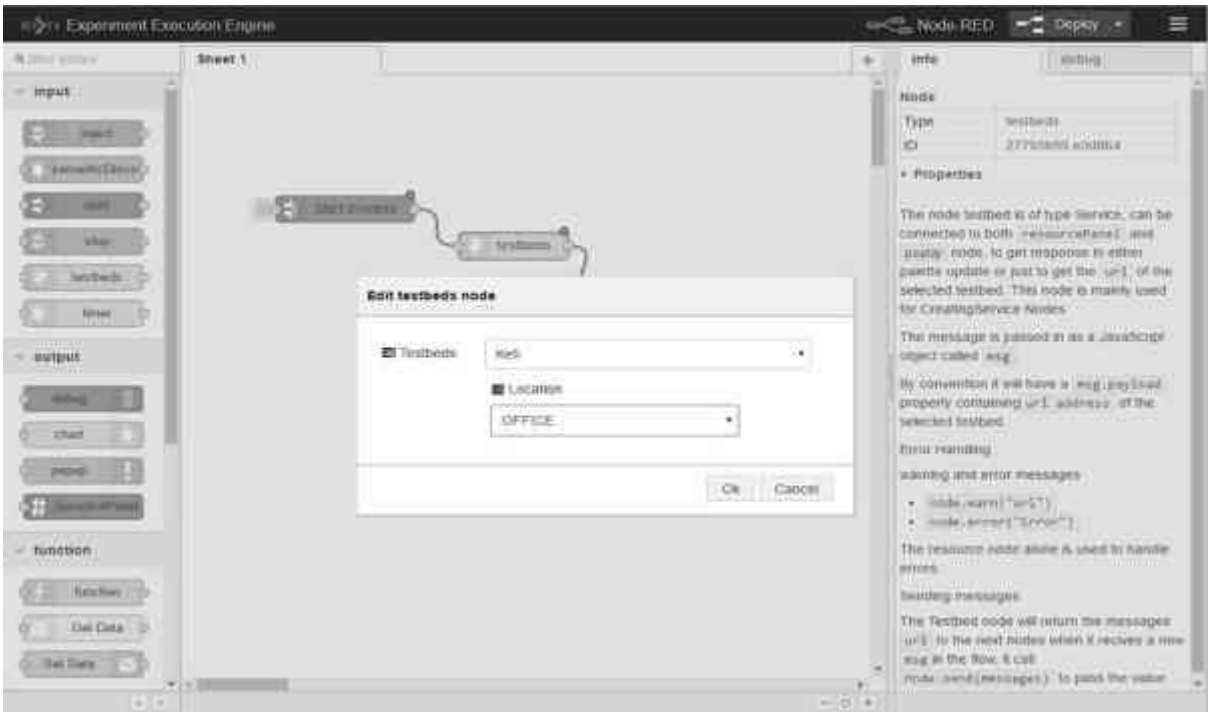


Figure 27: Edit Testbed Node

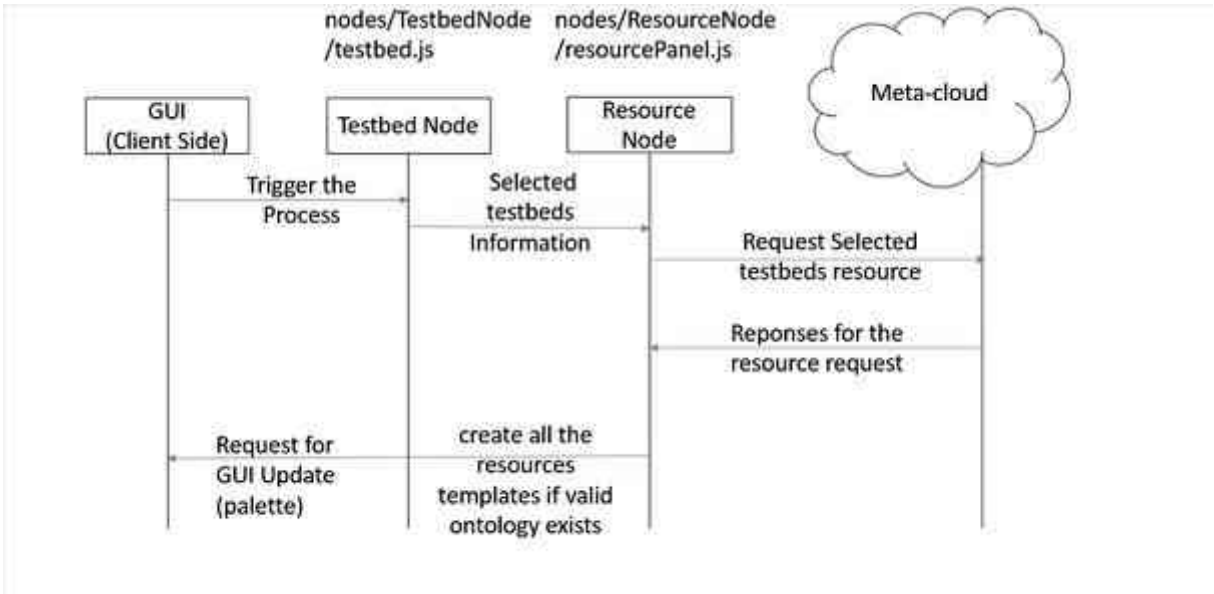


Figure 28: Resource Discovery Experiment Scenario

3.3.2.1.2.2Population

Once the resources are discovered they could be populated into the Node-RED using the `resourcePanel` node's 'add' feature which adds all the registered resources of the selected testbed to the palette as shown in the Figure 29. However, this is not efficient in terms of UI as there might be thousands of resources available with the testbed.

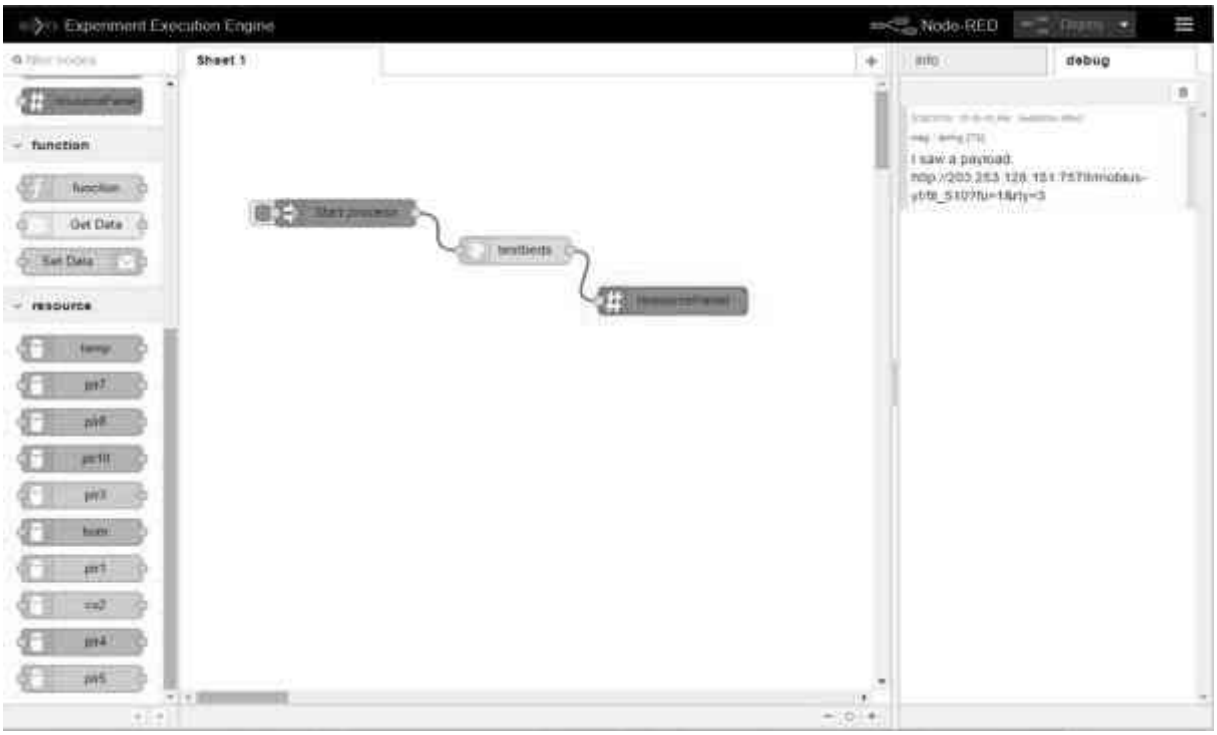


Figure 29: Resource Population into Node Red palette

3.3.2.1.2.3 Depopulation

The depopulation follows the same pattern as population except that `resourcePanel` node's option is changed to 'remove' from the default 'add' and the selected testbed resource will be removed from the palette. To perform successful depopulation the targeted resources should exist in the Node-RED palette.

3.4 The Selected Tools

Based on the goals of FIESTA-IoT and functional advantages of the above tools we choose the following technologies:

- We chose Quartz job scheduler to build the EEE functionalities: this choise is made in order to leverage from Scheduler made available under WP3 and be consistent. Further, Quartz APIs are easy to implement and provide needed functinality. Also as stated in [18], Quartz can support hundreds of complex jobs, have support for persistence and server restarts.
- Due to its features, we have decided to use ZK for the development of the FIESTA-IoT Portal. Nevertheless the UI for different tools such as EMC and Testbed Monitoring tool can use any other technology. These UIs can be invoked from ZK.
- Node-RED as the experiment editor: Node-RED editor is open source and can be easiliy customized to suffice the functionalites needed towards building an experiment.

4 EXPERIMENTATION SERVICES AND API SPECIFICATION

As described in [3] experiment lifecycle consists of resource discovery, experiment definition, experiment execution, and a result retrieval step. The following section will focus on APIs to support Phase 3 and Phase 4 of the experiment lifecycle. The APIs listed below are prone to be modified and changed. There might also be addition of some APIs. Also, we will address the access control by adding security aspects. We will address such mechanism in the next version of the deliverable.

4.1 Experiment Deployment Services

Below we list the experiment deployment related services provided by EEE. These services are services that ensures and target scheduling aspects, subscription and polling.

4.1.1 Scheduling APIs Specification

The `/startFISMOExecution` starts the schedule as specified in the FISMO object. This API upon successful starting returns {“response”: “Job Scheduled”, “jobID”: <JobID>}. The `jobID` and the `status` are stored in a database. The API reads the FISMO object associated with the `FISMOID` and its `QuerySchedule` attribute that contains scheduling information.

API	`/startFISMOExecution`
Description	This API is used to start execution of the experiment service (FISMO). This API provides a jobID to the FISMO upon the successful scheduling on the Meta Cloud. The API uses `timeSchedulePayload` to define the `startTime`, `stopTime` and `periodicity` of the job to be executed.
Method	`POST`
Input	HeaderParam: `String fismoID` HeaderParam: `String timeSchedulePayload` The `timeSchedulePayload` is a JSON string that should contain `startTime`, `stopTime` and `periodicity`. A sample of such JSON is `{"startTime":"2016-09-15 13:57:00", "stopTime":"2016-09-15 16:30:00","periodicity":60}`. Here `startTime` and `stopTime` are in `Date` format (`YYYY-MM-DD HH:mm:ss`) and the `periodicity` is in seconds.
Output	{“response”: “Job Scheduled”, “jobID”: <JobID>} is returned as a `Response` if successful. {“response”: <ERROR>} is returned as a `Response` if unsuccessful. For the possible list of error please see the Errors row below. If the status is scheduled then scheduled `jobID` is also returned.

Produces	application/json
Errors	• "NoSuchServiceModelObjectID": FISMOID is incorrect or does not exist. • "InvalidTImeScheduleStructure": timeSchedulePayload JSON structure is incorrect or does not exist. • "UnParsableDate": either startTime or stopTime is not in the correct format and thus can not be parsed in the required format. • "SchedulerException": is a generic error returned by the Quartz scheduler. • "ImplementationException": is a generic error however with respect to this API it would mean "Failed to execute"

To retrieve the jobIDs for a particular already scheduled FISMO, /getJobIDsfromFISMOID is used.

API	/getJobIDsfromFISMOID
Description	This API is used to get the jobID of a particular already scheduled FISMO. Note that this JobID is the ID given by the Scheduler to the FISMO execution.
Method	GET
Input	HeaderParam: String fismoID
Output	{"jobIDs": [<JobID1>,<JobID2>..]} is returned as a Response if successful. Here the JobIDs is a list of job IDs associated to the FISMOID. A list is returned because there might be subscribers who might have subscribed to a particular FISMOID. Each subscription to a FISMOID, provides a new jobID to the subscription. This is because we consider each subscription to be different. {"response": "No Jobs"} is also returned if there is no Jobs found for a particular FISMO. {"response": <ERROR>} is returned as a Response if unsuccessful. For the possible list of error please see the Errors row below.
Produces	application/json
Errors	• "NoSuchServiceModelObjectID": FISMOID is incorrect or does not exist • "SchedulerException": is a generic error returned by the Quartz scheduler.

	• "ImplementationException": is a generic error however with respect to this API it would mean "Failed to execute the API"

To retrieve the details about a jobID, /getJobIDDetails is used.

API	/getJobIDDetails
Description	This API is used to get the details associated to a particular jobID.
Method	GET
Input	HeaderParam: String jobID
Output	{"JobID": <JobID>, "Group":<GroupID>, "timeSchedule":{"startTIme":<startTime>, "stopTime":<stopTime>, "periodicity":<periodicity>}, "status":<status>} is returned as a Response if successful. Here the groupID is the FISMOID and status is a job status from the list [BLOCKED, COMPLETE, ERROR, NONE, NORMAL, PAUSED]. {"response": "No Job information found"} is also returned if there is no Jobs data. {"response": <ERROR>} is returned as a Response if unsuccessful. For the possible list of error please see the Errors row below.
Produces	application/json
Errors	• "NoSuchJobID": JobID is incorrect or does not exist • "SchedulerException": is a generic error returned by the Quartz scheduler. • "ImplementationException": is a generic error however with respect to this API it would mean "Failed to execute the API"

To retrieve the details about all jobIDs, /getAllJobIDDetails is used. This API is similar to previous one.

API	/getAllJobIDDetails
Description	This API is used to get the details of all the jobIDs.
Method	GET
Input	n/a

Output	{"JobsScheduled": [{"jobID": <JobID1>, "Group" : <GroupID>, "startTIme" : <startTime>, "stopTime" : <stopTime>, "periodicity" : <periodicity>, "status" : <status>}..]} is returned as a Response if successful. Here the groupID is the FISMOID and the status is a job status from the list [BLOCKED, COMPLETE, ERROR, NONE, NORMAL, PAUSED]. {"response": "No Jobs Scheduled"} is also returned if there is no Jobs data. {"response": <ERROR>} is returned as a Response if unsuccessful. For the possible list of error please see the Errors row below.
Produces	application/json
Errors	• "SchedulerException": is a generic error returned by the Quartz scheduler. • "ImplementationException": is a generic error however with respect to this API it would mean "Failed to execute the API"

Further, to get all the jobIDs for all the scheduled FISMOs use /getJobID

API	/getJobIDs
Description	To API is used to get all the existing jobIDs.
Method	GET
Input	n/a
Output	{"jobIDs": [{"jobID": <JobID1>, "FISMOID" : <FISMOID>}..]} is returned as a Response if successful. Here the JobID is the job ID of the scheduled FISMOID. {"response": "No Jobs Scheduled"} is also returned if there is no Jobs data. {"response": <ERROR>} is returned as a Response if unsuccessful. For the possible list of error please see the Errors row below.
Produces	application/json
Errors	• "SchedulerException": is a generic error returned by the Quartz scheduler. • "ImplementationException": is a generic error however with respect to this API it would mean "Failed to execute the API"

The /stopJobExecution stops the job that was already started using the previous defined start APIs. This API takes as an input the JobID and stops the job by deleting it from the scheduler.

API	/stopJobExecution
Description	The API is used to pause the execution of a particular job
Method	POST
Input	HeaderParam: String jobID.
Output	{"response": "Job paused successfully"} is returned as a Response if successful. {"response": <ERROR>} is returned as a Response if unsuccessful. For the possible list of error please see the Errors row below.
Produces	application/json
Errors	• "NoSuchJobID": JobID is incorrect or does not exist • "SchedulerException": is a generic error returned by the Quartz scheduler • "ImplementationException": is a generic error however with respect to this API it would mean "Failed to execute the API"

If a job is paused, it can also be resumed. To resume a job /resumeJobExecution is used.

API	/resumeJobExecution
Description	To API is used to resume the execution of a particular job
Method	POST
Input	HeaderParam: String jobID
Output	{"response": "Job resumed successfully"} is returned as a Response if successful. {"response": <ERROR>} is returned as a Response if unsuccessful. For the possible list of error please see the Errors row below.
Produces	application/json
Errors	• "NoSuchJobID": JobID is incorrect or does not exist • "SchedulerException": is a generic error returned by the Quartz scheduler • "ImplementationException": is a generic error

	however with respect to this API it would mean “Failed to execute the API”

The EEE also provide APIs to reschedule, delete jobs and identify what are the currently executing jobs. This is achieved using `/rescheduleJob`, `/deleteScheduledJob`, `/deleteAllScheduledJobs` and `/getCurrentlyExecutingJobs`.

API	`/rescheduleJob`
Description	This API is used to change the schedule of an already scheduled Job.
Method	`POST`
Input	HeaderParam: `String jobID` HeaderParam: `String startTime` HeaderParam: `String stopTime` HeaderParam: `String periodicity`
Output	{“response”: “Job rescheduled successfully”} is returned as a `Response` if successful. {“response”: <ERROR>} is returned as a `Response` if unsuccessful. For the possible list of error please see the Errors row below.
Produces	application/json
Errors	• “NoSuchJobID”: `JobID` is incorrect or does not exist • “SchedulerException”: is a generic error returned by the Quartz scheduler • “ImplementationException”: is a generic error however with respect to this API it would mean “Failed to execute the API”

API	`/deleteScheduledJob`
Description	This API is used to remove a particular scheduled job from the Scheduler
Method	`POST`
Input	HeaderParam: `String jobID`
Output	{“response”: “Job deleted successfully”} is returned as a `Response` if successful. {“response”: “No Job found”}

	could also be returned. {"response": <ERROR>} is returned as a Response if unsuccessful. For the possible list of error please see the Errors row below.
Produces	application/json
Errors	• "NoSuchJobID": JobID is incorrect or does not exist • "SchedulerException": is a generic error returned by the Quartz scheduler • "ImplementationException": is a generic error however with respect to this API it would mean "Failed to execute the API"

API	/deleteAllScheduledJob
Description	This API is used to remove all scheduled job from the Scheduler. This API will be protected and will be only available to the FIESTA-IoT administrators.
Method	POST
Input	n/a
Output	{"response": "All Job deleted successfully"} is returned as a Response if successful. {"response": "No Jobs found"} could also be returned. {"response": <ERROR>} is returned as a Response if unsuccessful. For the possible list of error please see the Errors row below.
Produces	application/json
Errors	• "SchedulerException": is a generic error returned by the Quartz scheduler • "ImplementationException": is a generic error however with respect to this API it would mean "Failed to execute the API"

API	/getCurrentlyExecutingJobs
Description	This API is used to get all the jobs that are currently being processed. Note that this is different from listing all jobs that are available in the persistence store of the scheduler.
Method	GET
Input	n/a

Output	{“response”: “Currently Executing Jobs.”, “Jobs”:[<jobs>..]} is returned if successful. {“response”: <ERROR>} is returned as a Response if unsuccessful. For the possible list of error please see the Errors row below.
Produces	application/json
Errors	• “SchedulerException”: is a generic error returned by the Quartz scheduler • “ImplementationException”: is a generic error however with respect to this API it would mean “Failed to execute the API”

4.1.2 Subscription APIs Specification

The subscription services (/subscribeToFISMOReport and /unsubscribeToFISMOReport to the discoverable FISMOs) are used so that an experimenter can subscribe to existing discoverable FISMOs or unsubscribe from already subscribed FISMO.

API	/subscribeToFISMOReport
Description	This API is used to subscribe to a particular FISMO’s report
Method	POST
Input	Header Param: String fismoID Header Param: String experimentOutput Header Param: String userID Header Param: String femoID Here the experimentOutput is the ExperimentOutput attribute of the FISMO in the JSON ({“url”: <url>, “optionalFile”: <FILE>, “optionalWidget”: <WIDGET>}). Currently, in this version, the optionalFile and optionalWidget support is not available. A sample of currently valid experimentOutput is {"url":"http://myExperiment.com"}. Further, the userID is the ID of the experimenter, and the femoID is the ID of the experiment to which subscription is to be associated to.
Output	{“response”: “subscribed”, “FISMOID”: <FIMSOID>, “JobID” : <JobID>} is returned as a Response if successful. {“response”: <ERROR>} is returned as a Response if unsuccessful. For the possible list of error please see the Errors row below.

Produces	application/json
Errors	• "NoSuchServiceModelObjectID": FISMOID is incorrect or does not exist • "NoSuchUserID": userID is incorrect or does not exist • "NoSuchExperimentID": FEMOID is incorrect or does not exist • "AlreadySubscribed": FISMOID is already subscribed and associated to the userID. • "InvalidURL": invalid url • "InvalidExperimentOutputJson": invalid Experiment Output Json • "ImplementationException": is a generic error however with respect to this API it would mean "Failed to execute the API or subscription failed"

API	/unsubscribeToFISMOReport
Description	This API is used to unsubscribe from a particular FISMO's report
Method	POST
Input	Header Param: String fismoID Header Param: String userID Header Param: String femoID userID is the ID of the experimenter, and femoID is ID of the experiment to which subscription is to be associated to.
Output	{"response": "Unsubscribed"} is returned as a Response if successful. {"response": <ERROR>} is returned as a Response if unsuccessful. For the possible list of error please see the Errors row below.
Produces	application/json
Errors	• "NoSuchServiceModelObjectID": FISMOID is incorrect or does not exist • "NoSuchUserID": userID is incorrect or does not exist • "NoSuchExperimentID": FEMOID is incorrect or does not exist • "SubscriptionNotFound": FISMOID is not associated to the userID. • "ImplementationException": is a generic error however with respect to this API it would mean "Failed to execute the API or un-subscription failed"

4.1.3 Polling APIs Specification

A polling service is a service using which an experimenter can run the FISMO once without actually scheduling it.

API	`/pollForReport`
Description	This API is used to invoke a previously defined FISMO. A call to this API will only produce one Resultset that will be sent to the URL specified in the `ExperimentOutput` parameter.
Method	`POST`
Input	Header Param: `String fismoID` Header Param: `String experimentOutput` Here the `experimentOutput` is the `ExperimentOutput` attribute of the FISMO in the JSON format ({"`url`": `<url>`, "`optionalFile`": `<FILE>`, "`optionalWidget`": `<WIDGET>`}). Currently, in this version, the `optionalFile` and `optionalWidget` support is not available. A sample of currently valid `experimentOutput` is `{"url":"http://myExperiment.com"}`.
Output	Jena Resultset in XML format
Produces	application/xml
Errors	• "`NoSuchServiceModelObjectID`": `FISMOID` is incorrect or does not exist • "`InvalidURL`": invalid `url` • "`InvalidExperimentOutputJson`": invalid `Experiment Output Json` • "`ImplementationException`": is a generic error however with respect to this API it would mean "Failed to execute the API"

4.2 Experiment Management Services

In this section, we list experiment management service provided by the EEE and the testbed status Monitoring services.

4.2.1 Monitor APIs Specification

Here we list all the APIs that provide "meta" information about an experiment and the associated services (FISMOs).

API	/getJobIDStatus
Description	This API is used to get the status of a particular jobID, i.e., one from the list [BLOCKED, COMPLETE, ERROR, NONE, NORMAL, PAUSED]
Method	GET
Input	HeaderParam: String jobID
Output	{"JobID":<JobID>, "status": <STATUS>} is returned as a Response if successful. Here STATUS is one from the list as described above. Other messages that are returned are {"response": "Job not Scheduled"} {"response": <ERROR>} is returned as a Response if unsuccessful. For the possible list of error please see the Errors row below
Produces	application/json
Errors	• "NoSuchJobID": JobID is incorrect or does not exist • "SchedulerException": is a generic error returned by the Quartz scheduler • "ImplementationException": is a generic error however with respect to this API it would mean "Failed to execute the API"

API	/getAllSubscribersOfFISMOID
Description	This API is used to get a list of subscribers (or the experimenters) that are using a particular FISMO.
Method	GET
Input	HeaderParam: String fismoID
Output	{"UserIDs": [<UserID1>, <UserID2>,..]} is returned as a Response if successful. Here, the "UserIDs" is a list of userIDs that have subscribed to the particular FISMO. It is also possible to get an empty JSON object if there is no user that has subscribed to the given FISMOID. {"response": <ERROR>} is returned as a Response if unsuccessful. For the possible list of error please see the Errors row below.
Produces	application/json
Errors	• "NoSuchServiceModelObjectID": FISMOID is incorrect or does not exist • "ImplementationException": is a generic error

	however with respect to this API it would mean “Failed to execute the API”

API	`/getAllSubscriptionsOfExperimenter`
Description	This API is used to get a list of user subscriptions irrespective of the experiment
Method	`GET`
Input	HeaderParam: `String userID`
Output	`{“FISMOIDs”: [<FISMOID1>, < FISMOID2>,..]}` is returned as a `Response` if successful. Here, the “`FISMOIDs`” is a list of FISMOIDs that the user has subscribed. It is also possible to get an empty JSON object if there are no FISMOIDs that a user has subscribed. `{“response”: <ERROR>}` is returned as a `Response` if unsuccessful. For the possible list of error please see the Errors row below.
Produces	application/json
Errors	• “`NoSuchUserID`”: `userID` is incorrect or does not exist • “`ImplementationException`”: is a generic error however with respect to this API it would mean “Failed to execute the API”

API	`/getMySubscriptionsforExperiment`
Description	This API is used to get a list of user subscriptions with respect to a particular experiment
Method	`GET`
Input	HeaderParam: `String userID` HeaderParam: `String femoID`
Output	`{“Subscriptions”: [{“jobID” : <jobID>, “fismoID”: <FISMOID1>},..]}` is returned as a `Response` if successful. Here, the “`Subscriptions`” is a list of jobIDs and FISMOIDs that the user has subscribed. It is also possible to get an empty JSON object if there are no subscriptions for a particular experiment by the user. `{“response”: <ERROR>}` is returned as a `Response` if unsuccessful. For the possible list of error please see the Errors row below.

Produces	application/json
Errors	• "NoSuchUserID": userID is incorrect or does not exist • "NoSuchExperimentID": FEMOID is incorrect or does not exist • "ImplementationException": is a generic error however with respect to this API it would mean "Failed to execute the API or Subscription Failed"

API	/getJobExecutionLog
Description	This API is used to get the ExecutionLog of a Job. The return is a JSON array with "executionTime" and "dataConsumed" information. Here executionTime is the time it took to successfully execute the Job.
Method	GET
Input	Header Param: String jobID
Output	{"ExecutionLog": [{"executionTime": <time1>, "dataConsumed": <dataConsumed1>}, {"executionTime": <time2>, "dataConsumed": <dataConsumed2>},..]} is returned as a Response if successful. Here, the "ExecutionLog" is a log of successful executions of jobID. It is also possible to get an empty JSON object if there is no ExecutionLog for the jobID. {"response": <ERROR>} is returned as a Response if unsuccessful. For the possible list of error please see the Errors row below.
Produces	application/json
Errors	• "NoSuchJobID": JobID is incorrect or does not exist • "ImplementationException": is a generic error however with respect to this API it would mean "Failed to execute the API"

API	/getJobExecutionCount
Description	This API is used to get the number of times a particular job was executed.
Method	GET
Input	Header Param: String jobID

Output	{"count": <count>} is returned as a Response if successful. Here, the "count" is the number of times the job is executed. Note that the count can also be 0. {"response": <ERROR>} is returned as a Response if unsuccessful. For the possible list of error please see the Errors row below.
Produces	application/json
Errors	• "NoSuchJobID": JobID is incorrect or does not exist • "ImplementationException": is a generic error however with respect to this API it would mean "Failed to execute the API"

4.2.2 FIESTA-IoT Access Mechanism & Testbeds status APIs Specification

The following tables specify the APIs that are intended to be used to retrieve information regarding the status of the FIESTA-IoT access mechanisms, and the status of the testbeds registered within FIESTA-IoT.

In terms of the monitoring of the FIESTA-IoT access mechanisms, there will be a set of methods that will retrieve: information regarding the status of the access mechanism (providing as true/false value as response); information regarding the time interval being used to check the status of the access mechanism, and a method to change the time interval. The specification of these 3 methods is described below:

API	/fiestaStatus
Description	This API will provide information regarding the status of the FIESTA-IoT Access mechanisms.
Method	GET
Input	n/a
Output	Boolean (True in case the access mechanism is working correctly)
Produces	application/json
Errors	• "ImplementationException": is a generic error however with respect to this API it would mean "Failed to execute the API"

API	/getStatusCheckInterval
Description	This API will provide information regarding the time interval being used to recurrently test the FIESTA-IoT access

	mechanisms
Method	`GET`
Input	n/a
Output	Number or String (value in seconds)
Produces	application/json
Errors	• "`ImplementationException`": is a generic error however with respect to this API it would mean "Failed to execute the API"

API	`/setStatusCheckInterval`
Description	This API will provide the possibility of changing the time interval being used to recurrently test the FIESTA-IoT access mechanisms
Method	`PUT`
Input	Integer (value in second)
Output	Boolean (True in the case the value was successfully changed)
Produces	application/json
Errors	• "`ImplementationException`": is a generic error however with respect to this API it would mean "Failed to execute the API"

Another aspect that is going to be monitored is the status of the testbeds registered within the FIESTA-IoT platform. This will enable experimenters to check if FIESTA-IoT is being able to retrieve data from each of the testbeds, in the case they feel they're not receiving the expected amount of data. The specification of the method is described in the following table:

API	`/testbedStatus`
Description	This API will provide information regarding the status of the testbeds registered in FIESTA-IoT Platform
Method	`GET`

Input	n/a
Output	List
Produces	application/json
Errors	• "`ImplementationException`": is a generic error however with respect to this API it would mean "Failed to execute the API"

4.3 Documentation of APIs

There are many tools available for documenting APIs developed. Some of the tools available are Swagger[54], RAML for JAX-RS[55], Apiary[56], DRF Docs[57], Miredot[58] and many more[59]. Due to our previous experience we use RAML for JAX-RS to document the APIs we have built. Some of the APIs that are built as part of WP3 and integrated in the portal as part of Task 4.4 are the APIs to access data from FIESTA-IoT Triple Store. We are also working towards providing a working link of the documentation for these APIs.

[54] http://swagger.io
[55] https://github.com/mulesoft/raml-for-jax-rs
[56] https://apiary.io
[57] http://drfdocs.com
[58] http://www.miredot.com
[59] http://www.mattsilverman.com/2013/02/tools-to-generate-beautiful-api-documentation.html

5 SPECIFICATION OF UI TOOLS

Below we provide a list of planned tools that we envision to build in the second release of the document. Note that rather than the tools mentioned below, we will also gradually enhance the existing tools' behavior and performance.

5.1 Testbed Monitoring tool

The Testbed Monitoring tool is intended to provide all the FIESTA-IoT stakeholders, the current status of the FIESTA-IoT platform, in terms of access mechanisms. This means that it will be recurrently testing if the FIESTA-IoT platform is being accessible from the testbed point-of-view. Besides, it has to verify if the FIESTA-IoT platform is able to access the registered testbeds and provide the FIESTA-IoT stakeholders the current and historical status of each of them.

This monitoring tool can provide essential information to several FIESTA-IoT stakeholders, including:

- Experimenters can understand if there has been an issue with a specific testbed during a limited period of time and therefore the expected amount of data for a specific experiment has not been received;
- Testbed providers can understand possible issues in the connection between their testbed and the FIESTA-IoT platform,
- The FIESTA-IoT development team can monitor the platform behavior in terms of connecting to testbeds.

The Figure 30 presents a mockup of the UI of the Testbed Monitoring tool. Here, the idea is to provide the collected status information in a dashboard-like manner, so that it is easier for the new stakeholders to identify the possible issues related to the FIESTA-IoT platform's ability to get new data from testbeds.

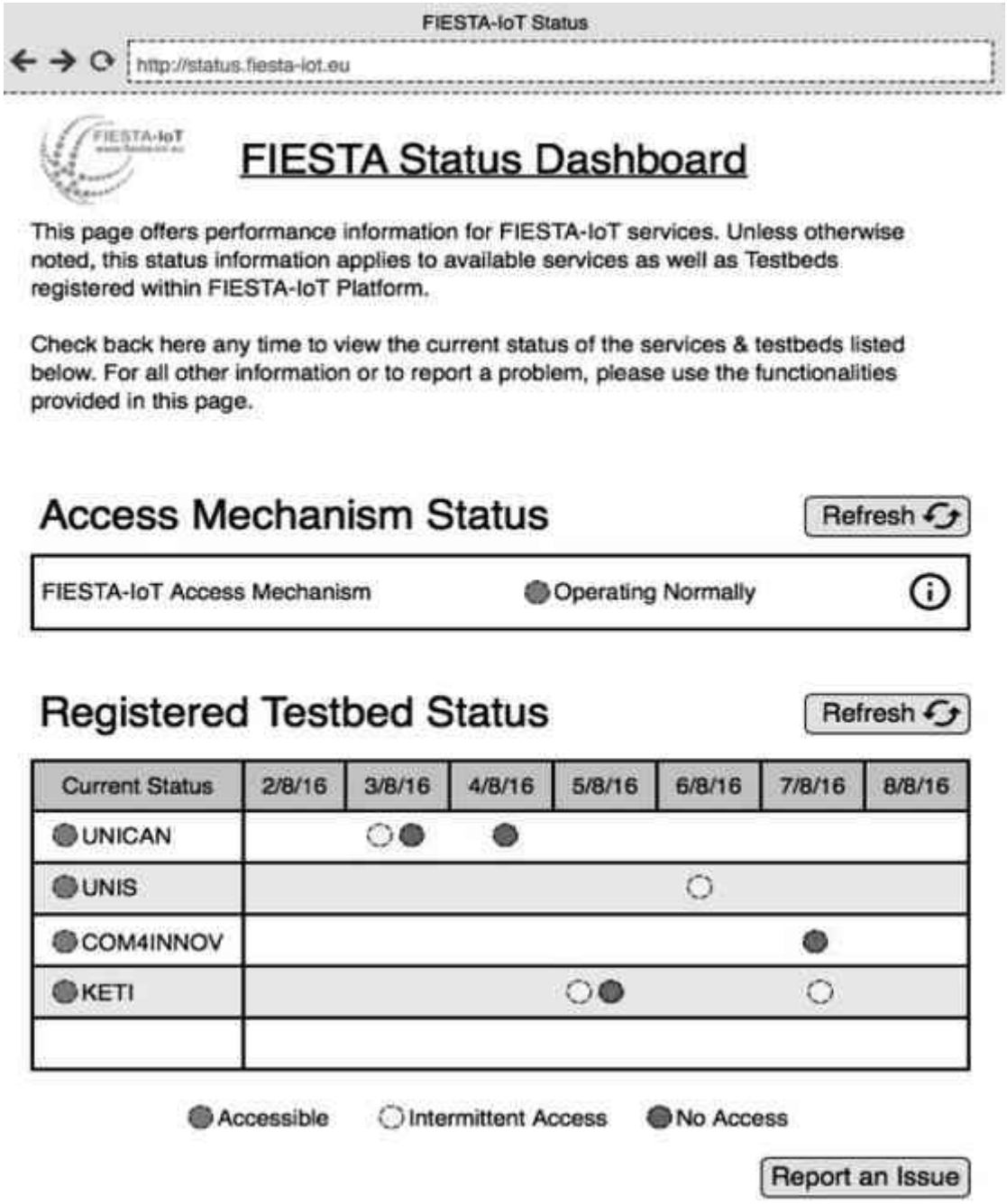


Figure 30:FIESTA-IoT Status Dashboard

5.2 Experiment Management Console

The Experiment Management Console (EMC) is a UI where the experimenter could know about the status of his/her experiment(s). The EMC would list experiments associated to an experimenter. Upon selecting a specific experiment, say "Experiment 2" as in Figure 31, the details of the experiment are presented to the experimenter. This includes experiment details, associated FISMOs and other discoverable FISMOs. An experimenter can chose to update the metadata of the experiment that he/she has created using "`Update MetaData`". This will open a

simple HTML form with prefilled fields. An experimenter can also delete his/her experiment. Upon deleting all references to the experiment would be deleted. This includes: (a) deletion of FISMOs that the experimenter has created, (b) deletion of all the subscriptions to the particular FISMO that was deleted in step (a), and (c) a notification "The FISMO <FISMOID> has been deleted" to all the subscribers of the deleted FISMO. An "`Edit in NodeRed`" button would allow the experimenter to update all the FISMOs that the experimenter has created and are associated to the specific experiment. Note that, here subscribed FISMOs will not be loaded. The "`Associated FISMO`" tab shows the "meta" information about the FISMO. The "meta" information includes if the FISMO was owned or subscribed with the frame of an experiment. The status of the FISMO can be either stopped, scheduled or running. Experimenters can also start/stop a particular FISMO. By default all the FISMOs would have status set to "scheduled" and started. In case of "`owned`" FISMOs, as the FISMO is described using NodeRed, the "`Start Now`" and "`Stop Now`" would only provide the scheduler the information to schedule the respective FISMO. For "Subscribed" FISMOs (as such FISMOs are not owned), the scheduler would only be sending the output of the FISMO to the desired endpoint. Each FISMO also shows "`Past executions`" with information such as `datetime` when the FISMO was successfully executed with the size of the data consumed by the FISMO from the Meta Cloud. Nevertheless, experimenters can delete/unsubscribe a particular FISMO and also poll for results. Note that if an experimenter clicks on "delete" button inside the FISMOID frame, this would open the NodeRed panel. It is then from NodeRed the experimenter would be allowed to delete a particular FISMO. Along with this step if in the Node-RED, the experimenter actually deletes the FISMO, EEE is notified to delete the job associated to it. Lastly, an experimenter would also be able to browse through a list of available FISMOs for new subscriptions.

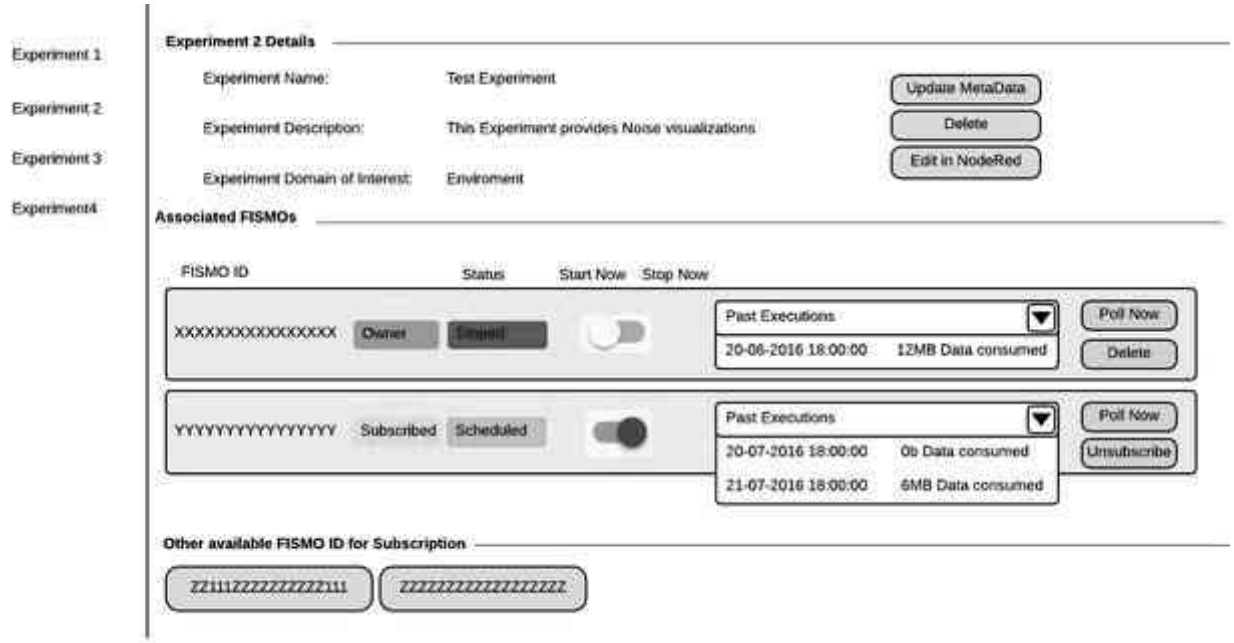


Figure 31:Experiment Management Console

5.3 Data Visualization Tools

On top of all the components that have been described so far, we plan to build a set of features with the clear aim of easing life of external experimenters. This way, they will find a number of graphical tools that are being tailored to cover all the potential needs of future users dealing with the data collected by the FIESTA-IoT platform. Below we summarize the main ones, beside a brief overview of their main functionalities. Nonetheless, the reader has to take into account that they are currently in a preliminary stage and their final layout might be slightly different to the one we are presenting in this section.

5.3.1 Knowledge Acquisition Toolkit

In order to maximise the added value of the data being extracted from the FIESTA-IoT testbeds for the experimenter, it is important to provide data analysis tools and services. As a result, the Knowledge Acquisition Toolkit (KAT) web service is being developed for FIESTA-IoT in order to provide open access data analysis tools for data consumers as a web service. Such a tool provides the following benefits:

- for novice/beginner data consumer, the tools that would enable them to analyze and obtain useful information
- for the more advanced/experienced user providing the most effective tools for a given data set.

For example, such a tool would provide relevant documentation for the beginner data consumer, with examples of data processing work flows; while for the more experienced user the most advanced tools developed in academic institutions can be evaluated by a wide range of users and the most useful data analysis tool for a given data set will be identified. Furthermore, by providing data analysis as a web service, FIESTA-IoT enables a wider range of experimenters to access the FIESTA-IoT system.

A potential example of the layout for the KAT web service GUI is shown in Figure 32. Where on the top left hand corner of the figure, the user specifies the input CSV data file to be processed along with the corresponding directory for storing the processed data. The user can then load the data into the KAT web server, along with selecting the requested methods and corresponding parameters for processing. The user can then visualize the processed data on the right hand panel of the GUI.

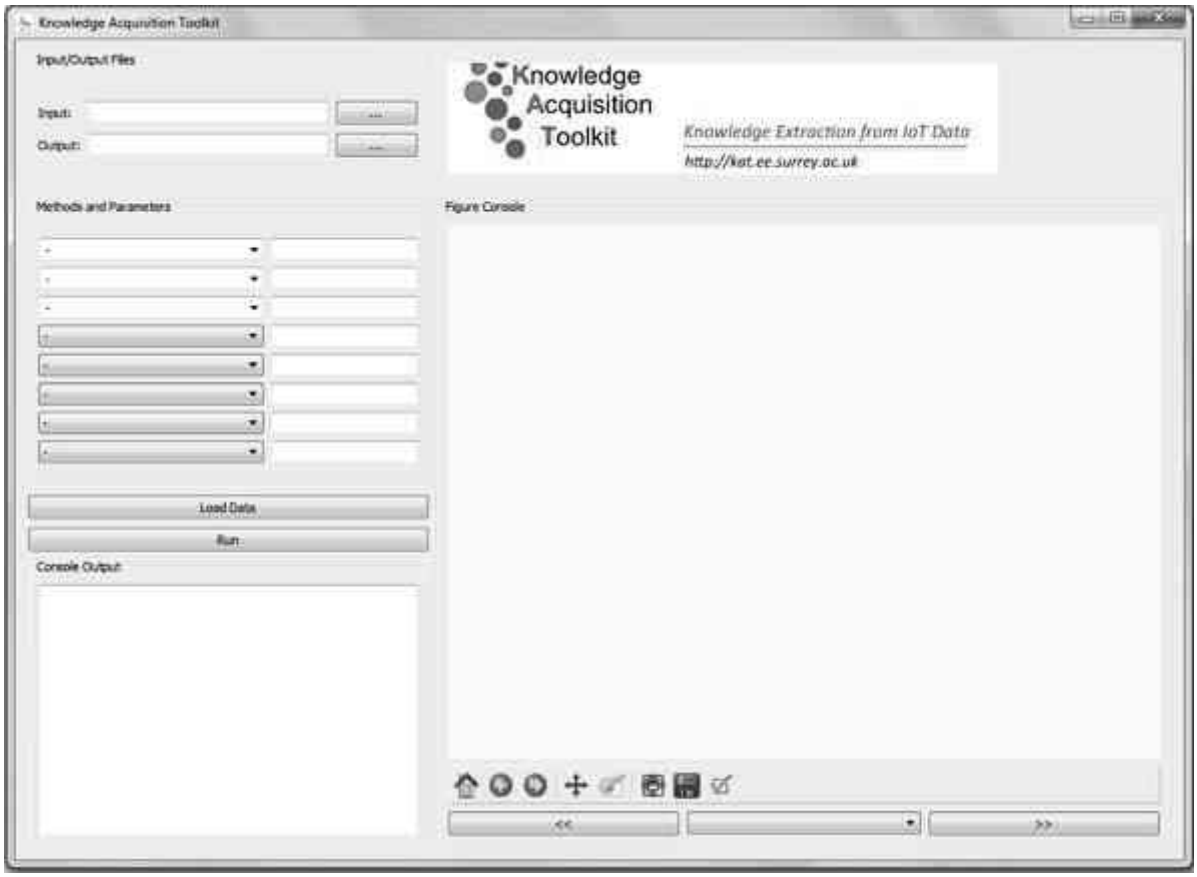


Figure 32: KAT web service UI

5.3.2 Map-based interface

One of the things that come to one's mind when thinking of how to interact with assets would be most likely through a map. Thanks to this visual layout, users do have an instantaneous bidirectional communication channel between them and the underlying machines. On the one hand, all the resources will be displayed on the map, so everyone playing around with the front-end will receive an immediate response of what is there and, more important, where it is; on the other hand, they can also generate information that will be processed in the remote server's premises.

This way, the first aim of this tool is to show the different assets from the different federated testbeds on the corresponding positions onto the map. Nonetheless, apart from this straightforward operation, we have in mind to support more complex features, which will be briefly introduced below. Before providing details, it is worth highlighting that the tool is developed using *Leaflet JS* library[60] ("*an open-source JavaScript library for mobile-friendly interactive maps*"). As long as the core of the platform evolves, more features will be added to this graphical interface.

[60] http://leafletjs.com/

5.3.2.1 Resource Discovery

As stated above, the essential and most basic functionality of the tool is to display all the resources (assets) that are registered with the FIESTA-IoT platform in a testbed agnostic way. Figure 33 presents the user interfaces generated by the `Leaflet` library.

Figure 33: Map user interface based on the Leaflet library

Apart from the map per se, we can see (at the left side of the picture) a set of buttons that allow us to, for instance, zoom in/out (i.e. '+' and '-' symbols) and generate different types of `GeoJSON` objects (polyline, polygon, rectangle, circle or marker, respectively). Bound to these elements that will be overlaid on the map, we can finally observe the last pair of buttons, whose role is to allow the edition of removal of the aforementioned objects. In addition, the top right corner contains a drop-down menu is present that permits the selection of the base map and the rest of the layers (some layers can be: the raw map layer, the markercluster layer, the drawn polygons layer, etc.) that will be plotted on it.

Regarding the FIESTA-IoT's resources, we can see on the map up to four different markers (or group of them, since the zoom level deters from watching every one). As can be easily inferred, they correspond to the four in-house testbeds that have registered their assets into the FIESTA-IoT federation: SmartSantander (Santander, Spain), Com4Innov (Sophia Antipolis, France), Smart ICS (Guildford, UK) and KETI's smart building (Seoul, South Korea). Thus, each of the heterogeneous elements (check [20] to see how the different testbeds natively define their own datasets) are being collected in a testbed-agnostic way.

Behind the UI besides representing the map, what this tool is doing is to utilize the FIESTA-IoT EaaS API to retrieve a list containing all the registered resources. With regards to the specification of this API, it will be included in the forthcoming deliverable D4.3 [21].

In this first approach, experimenters get, as shown on the map, all the resources (assets) that have been registered into the FIESTA-IoT IoT Service and Resource Directory FC. However, they might have in mind not to work with all but only with a subset of the resources. Here is where the previously mentioned `GeoJSON` elements come into the scenes. For a clearer view on this, Figure 34 outlines the usability of the tool. We can define, among different shapes, i.e. polyline, polygon, rectangle and circle (the marker option is pointless here), the particular regions of space within which we do want to e.g. retrieve data from. Then, another list of resources will be managed in order to execute further operations, like the subscription to events or the creation of Virtual Entities[61].

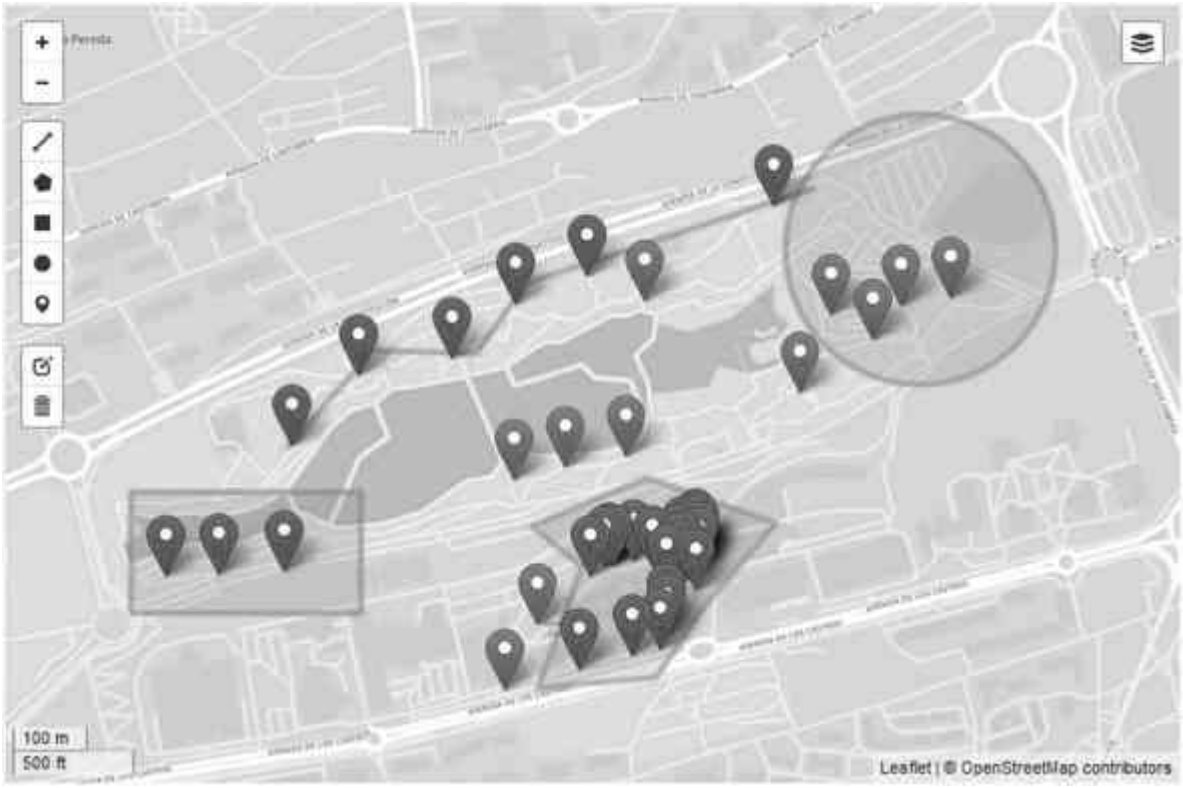


Figure 34: Example of the usage of GeoJSON objects to filter in/out devices

[61] We will include the information on these operations in the second version of the deliverable, moment where all the underlying technologies shall be up and running.

5.3.3 Visualization using a dashboard framework: Freeboard

Freeboard[62] is a dashboard framework specifically conceived for the IoT scenarios (see Figure 35). It comes in two versions: (a) a remote version handled by the Freeboard servers accessed by a user via a web interface, and (b) an open-source version[63] that consists only of the client side. The client side version consists of the graphical application. It does not include any of the server-side code for user management. Further, saving to database and public/private functionality is left to the developer to implement. The open-source version is more than enough for data visualization, since all other features will be handled by the FIESTA-IoT platform.

Freeboard is an HTML-based modular engine. It has a plugin architecture that allows the extension by creating new “datasource” components (those in charge of fetching the data) and new “widget” components (those allow the visualization of the data).

Freeboard is all contained in an HTML static page and it does not need a backend server in order to be executed. Once the Freeboard project has been cloned/downloaded on the local machine, simply opening the index.html on a web browser will show: (a) the freeboard dashboard skeleton which consists of a grid where the widgets can be placed, (b) a top bar environment for defining datasources (on the top right), and (c) dashboard management like saving, load etc. (on the top left).

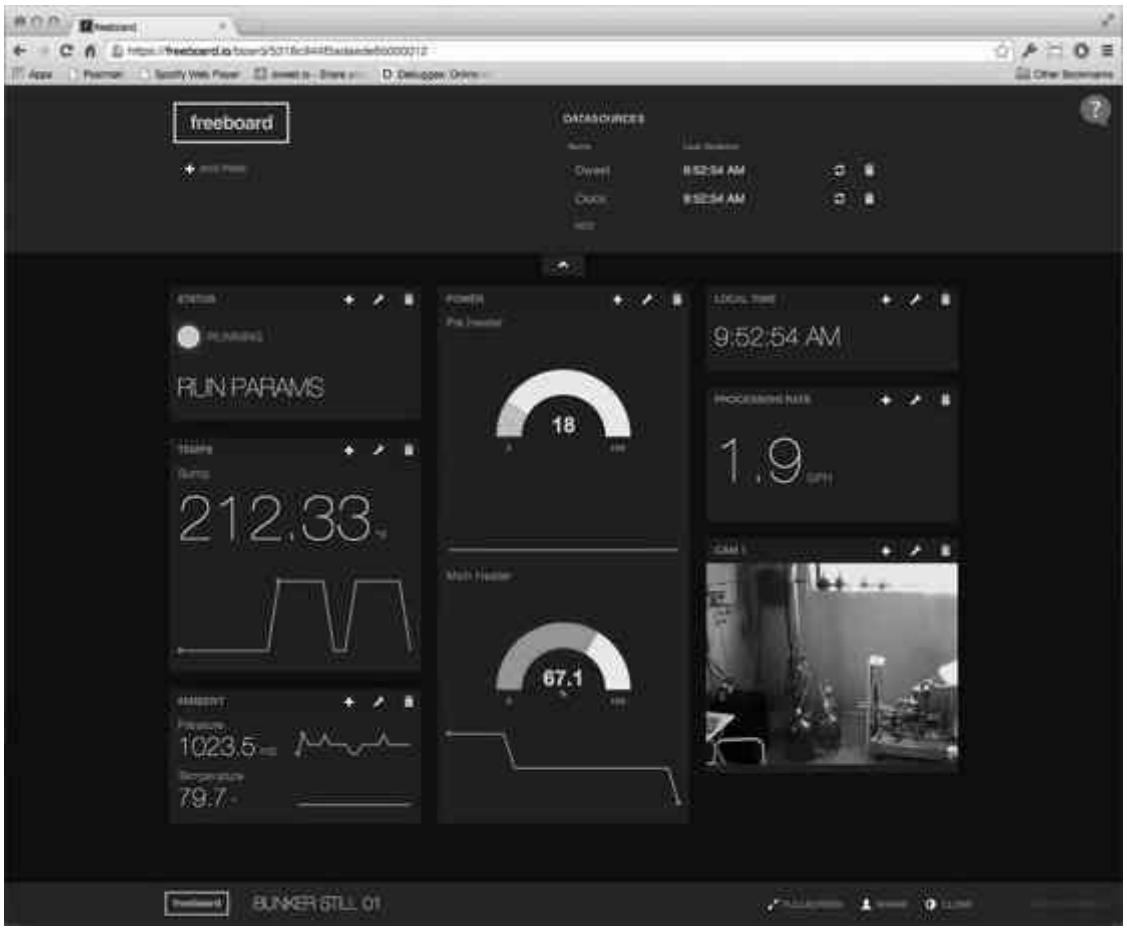


Figure 35 Dashboard example using Freeboard

[62] https://freeboard.io/
[63] https://github.com/Freeboard/freeboard

The default datasources offered by Freeboard are:

- **JSON datasource** that parses a JSON message as response of a request URL (set at JSON datasource instantiation). The request can be set in polling and the polling period is also set at datasource instantiation.
- **Open Weather Map API datasource** that can request weather information to the OpenWeatherMap[64] portal.
- **Dweet.io datasource** that can get date from the dweet.io[65] platform. The latter is a messaging framework for IoT data similar to the Twitter for humans where a data message is associated to a topic.
- **Playback datasource** that reads the inputs from a data file
- **Clock datasource** that reads the machine clock
- **Octoblu datasource** that enables the connection to the Octoblu[66] IoT messaging and automation framework.

The widgets already offered by the Freeboard project are:

- **Text widget** for showing the data as text.
- **Gauge widget** that shows a gauge with a color schema going from green to red passing by yellow, as bigger the value is. The boundaries of the gauge can be set.
- **Sparkline widget** for plotting the history of an attribute. It can show more than one line in the same plot.
- **Pointer widget** for showing 2-dimensional directions, e.g. useful for the wind direction.
- **Picture widget** for showing a picture from an URL.
- **Indicator light widget** that is a graphical representation of a Boolean flag.
- **Google map widget** that embeds the Google map library.
- **HTML widget** that is simply showing a generic HTML (hardcoded at instantiation time or dynamically generated via a JavaScript code).

All the provided widget can be attached with JavaScript code for parsing, extracting or processing the data coming from the datasources.

5.3.4 Visualization Using Node-RED

Visualization is a process of producing a pictorial representation of the experiment's result. To perform visualization in Node-RED, we have chart node powered by Google Charts[67] that could graph a single input into multiple graph formats such as Area, Bar, Column, Scatter, Annotation and Line Charts. We also implement a node that pops up a window to represent the input message.

[64] https://openweathermap.org/
[65] http://dweet.io/
[66] https://www.octoblu.com/
[67] https://developers.google.com/chart/

The visualization in Node-RED is limited to single node input that means: both chart and popup nodes can take input from only one wired node because in Node-RED, nodes do not support multiple inputs. However, we can send the data in a single line format or through a buffer process. The following Figure 36 to Figure 38 represent an experiment where a CO2 sensor value is obtained in regular intervals and visualized using Area chart node.

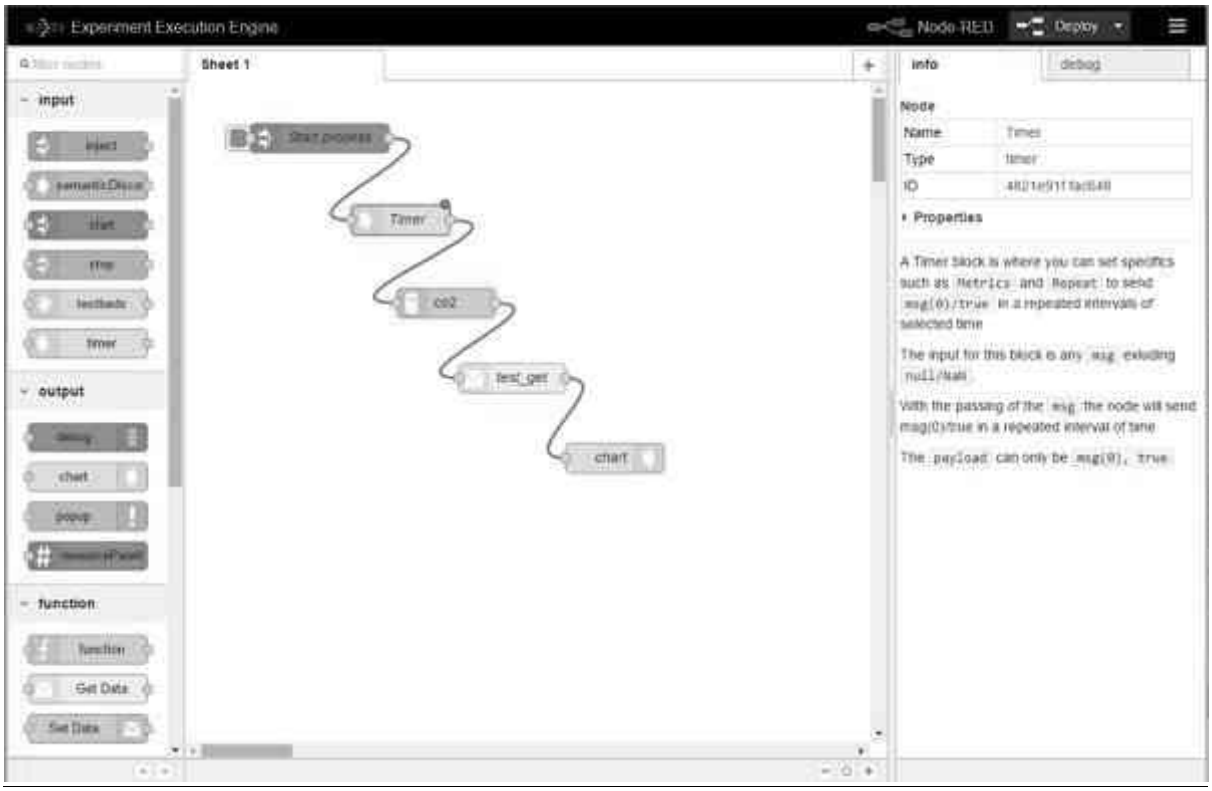


Figure 36: Visualization of the experiment build

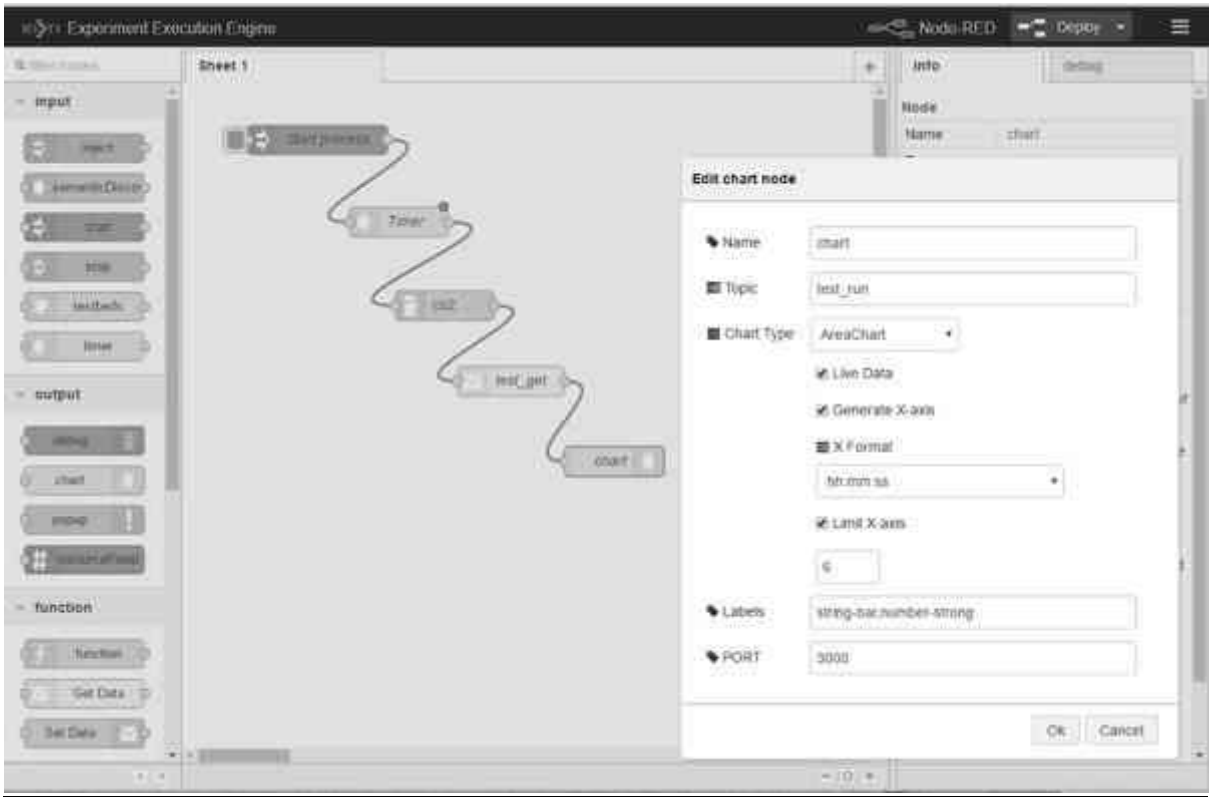


Figure 37: Visualization of the experiment build

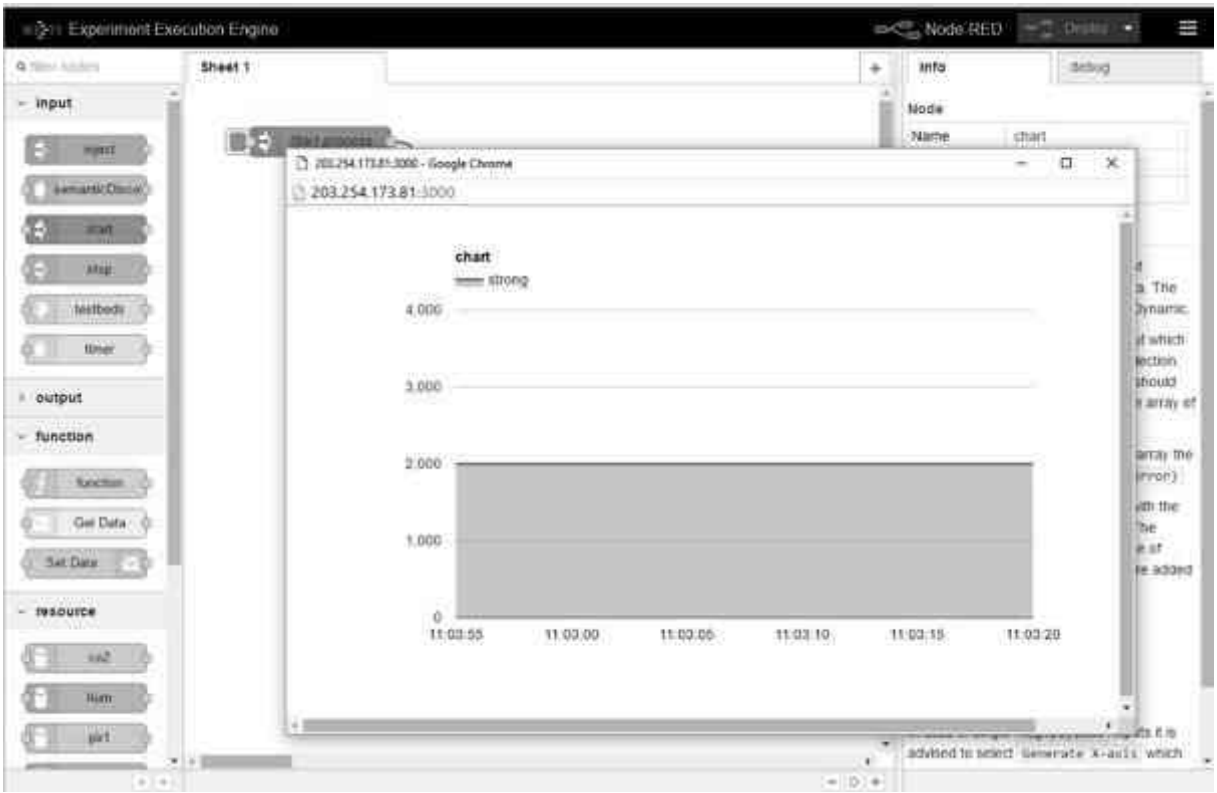


Figure 38: Visualization of the experiment build

5.4 Platform's Modules Runtime Monitoring

In order to implement monitoring features for the various components, the `JavaMelody` library has been integrated into the system. `JavaMelody` has the functionality of monitoring the JVM and the Java EE application server. It is a tool to measure and calculate statistics on the real operation of an application depending on the usage of the application by users[68]. `JavaMelody` is mainly based on statistics of requests and on evolution charts. `JavaMelody` can be configured to provide monitoring information for the entire Host Environment, as well as each individual component. More specifically, `JavaMelody` can:

- provide facts about the average response times and number of executions
- make decisions when trends are bad before problems become too serious
- optimize based on the more limiting response times
- find the root causes of response times
- verify the real improvement after optimizations
- include summary charts which can be viewed on the current day, week, month, year or custom period, showing the evolution over time of the following indicators:
    - Number of executions, mean execution times and percentage of errors of HTTP requests, SQL requests, JSF actions, JSP pages or methods of business façades (if EJB3, Spring or Guice)
    - Java memory
    - Java CPU
    - Number of user sessions
    - Number of JDBC connections

`JavaMelody` also includes statistics of predefined counters (current HTTP requests, SQL requests, JSF actions, JSP pages and methods of business façades for EJB3, Spring or Guice) with, for each counter:

- A summary indicating the overall number of executions, the average execution time, the CPU time, and the percentage of errors.
- The percentage of time spent in the requests for which the average time exceeds a configurable threshold.
- The complete list of requests, aggregated without dynamic parameters with, for each, the number of executions, the mean execution time, the mean CPU time, the percentage of errors and an evolution chart of execution time over time.
- Each HTTP request indicates the size of the flow response, the mean number of SQL executions and the mean execution time.

[68] https://github.com/javamelody/javamelody/wiki

- It also includes statistics on HTTP errors, warnings and errors in logs, on data caches if Ehcache and on batch jobs if Quartz.
- Number of executions, mean execution times and percentage of errors of HTTP requests, SQL requests, JSF actions, struts actions, JSP pages or methods of business façades (if EJB3, Spring or Guice)

By clicking on a monitoring option, we get the screen depicted in Figure 39 below.

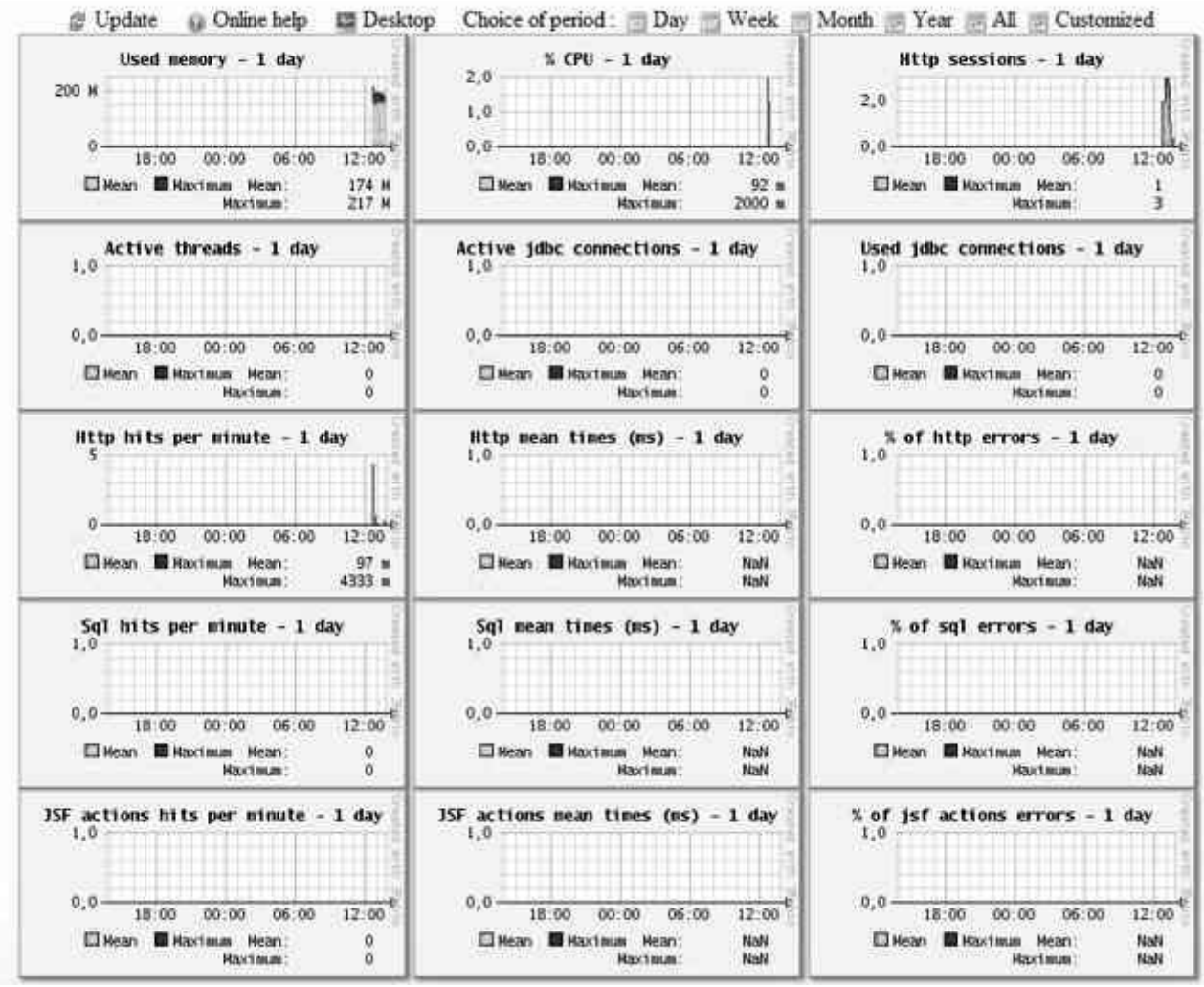
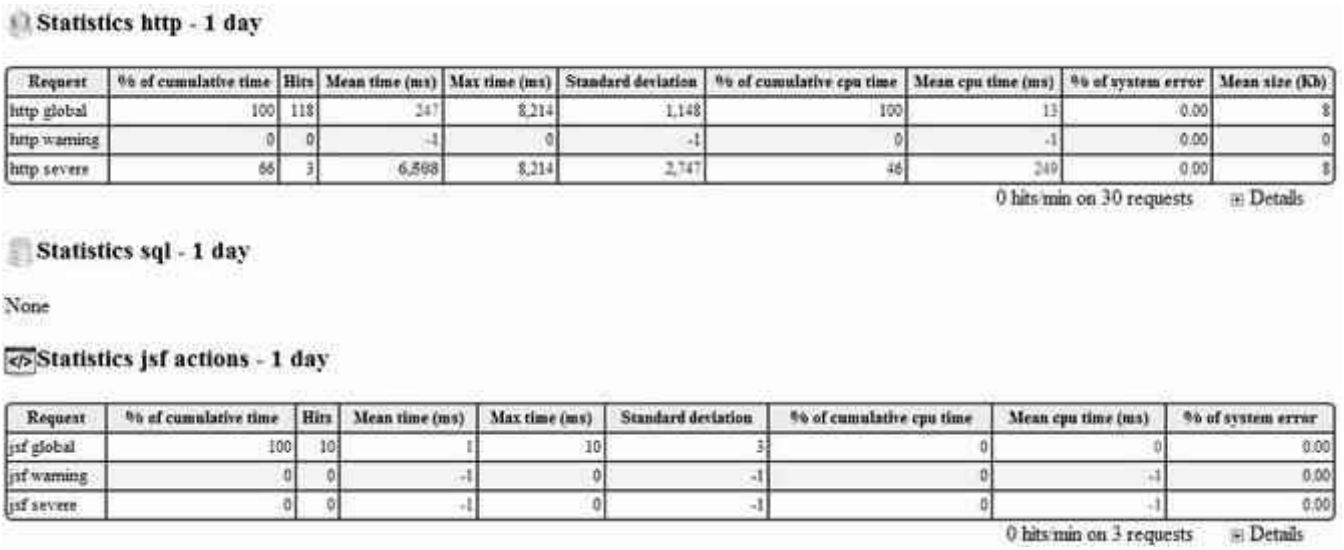


Figure 39: JavaMelody monitoring graphs

These charts (Figure 39) display various statistics concerning a particular module. By clicking on one of the charts we get a more detailed representation. Further down from the initial monitor screen, we can see various statistics collected by `JavaMelody`. An example is presented in Figure 40.



Statistics http - 1 day

Request	% of cumulative time	Hits	Mean time (ms)	Max time (ms)	Standard deviation	% of cumulative cpu time	Mean cpu time (ms)	% of system error	Mean size (Kb)
http global	100	118	247	8,214	1,148	100	13	0.00	8
http warning	0	0	-1	0	-1	0	-1	0.00	0
http severe	66	3	6,868	8,214	2,747	46	249	0.00	8

0 hits/min on 30 requests Details

Statistics sql - 1 day

None

Statistics jsf actions - 1 day

Request	% of cumulative time	Hits	Mean time (ms)	Max time (ms)	Standard deviation	% of cumulative cpu time	Mean cpu time (ms)	% of system error
jsf global	100	10	1	10	3	0	0	0.00
jsf warning	0	0	-1	0	-1	0	-1	0.00
jsf severe	0	0	-1	0	-1	0	-1	0.00

0 hits/min on 3 requests Details

Figure 40: JavaMelody statistics

By clicking on the bottom right corner we can see more details concerning the various statistic groups as shown in Figure 41.

Request	% of cumulative time	Hits	Mean time (ms)	Max time (ms)	Standard deviation	% of cumulative cpu time	Mean cpu time (ms)	% of system error	Mean size (Kb)
http global	100	171	25	1,497	140	100	9	0.00	1
http warning	0	0	-1	0	-1	0	-1	0.00	0
http severe	70	7	440	1,497	555	72	173	0.00	5

3 hits/min on 26 requests ⊟ Details

Request	% of cumulative time	Hits	Mean time (ms)	Max time (ms)	Standard deviation	% of cumulative cpu time	Mean cpu time (ms)	% of system error	Mean size (Kb)
home.jsf GET	70	7	440	1,497	555	72	173	0.00	5
resources/img/openiotlogo_left_IDE.jpg GET	12	5	104	506	224	0	0	0.00	23
home.jsf ajax POST	3	2	60	83	19	7	62	0.00	0
welcome.jsf GET	2	5	21	38	10	3	12	0.00	1
javax.faces.resource/style.css.jsf GET	2	7	14	36	14	3	8	0.00	1
javax.faces.resource/theme.css.jsf GET	1	12	4	5	4	0	0	0.00	0
/ GET	1	4	12	24	10	0	0	0.00	0
javax.faces.resource/jquery/jquery-plugins.js.jsf GET	0	7	5	23	7	0	2	0.00	0
javax.faces.resource/primefaces.js.jsf GET	0	12	3	7	2	1	2	0.00	0
javax.faces.resource/layout/layout.css.jsf GET	0	7	4	9	3	0	0	0.00	0
javax.faces.resource/primefaces.css.jsf GET	0	12	2	3	1	0	1	0.00	0
javax.faces.resource/jquery/jquery.js.jsf GET	0	12	1	3	0	0	0	0.00	0
javax.faces.resource/layout/layout.js.jsf GET	0	7	2	5	1	1	4	0.00	0
javax.faces.resource/images/ui-icons_f9bd01_256x240.png.jsf GET	0	6	3	6	1	0	2	0.00	0
javax.faces.resource/images/ui-bg_inset-hard_100_fcfdfd_1x100.png.jsf GET	0	11	1	2	0	0	0	0.00	0
javax.faces.resource/primefaces-extensions.js.jsf GET	0	7	2	3	0	0	2	0.00	0
javax.faces.resource/images/ui-icons_469bdd_256x240.png.jsf GET	0	6	2	6	1	1	5	0.00	0
javax.faces.resource/layout/toggle-up.gif.jsf GET	0	5	2	4	1	0	0	0.00	0
javax.faces.resource/images/ui-bg_gloss-wave_55_5c9ccc_500x100.png.jsf GET	0	6	1	2	0	0	0	0.00	0
javax.faces.resource/layout/toggle-lt.gif.jsf GET	0	5	1	3	0	0	0	0.00	0
javax.faces.resource/images/ui-bg_glass_85_dfeffc_1x400.png.jsf GET	0	6	1	2	0	0	2	0.00	0
javax.faces.resource/images/ui-bg_inset-hard_100_f5f8f9_1x100.png.jsf GET	0	6	1	3	0	0	0	0.00	0
javax.faces.resource/images/ui-icons_217bc0_256x240.png.jsf GET	0	3	2	2	0	0	0	0.00	0
resources/img/progress.gif GET	0	6	1	2	0	0	2	0.00	5
javax.faces.resource/images/ui-bg_glass_75_d0e5f5_1x400.png.jsf GET	0	3	1	2	0	0	5	0.00	0
javax.faces.resource/images/ui-icons_6da8d5_256x240.png.jsf GET	0	2	1	2	0	0	7	0.00	0

Figure 41: JavaMelody detailed statistics

Further down, we can view (see Figure 42) system and thread information as well as execute some JVM and Container commands such as:

- Execute the Java Garbage Collector
- Generate a Heap Dump
- View memory histogram
- Invalidate HTTP sessions
- View HTTP sessions
- View deployment descriptor
- View MBeans
- View OS processes
- JNDI tree

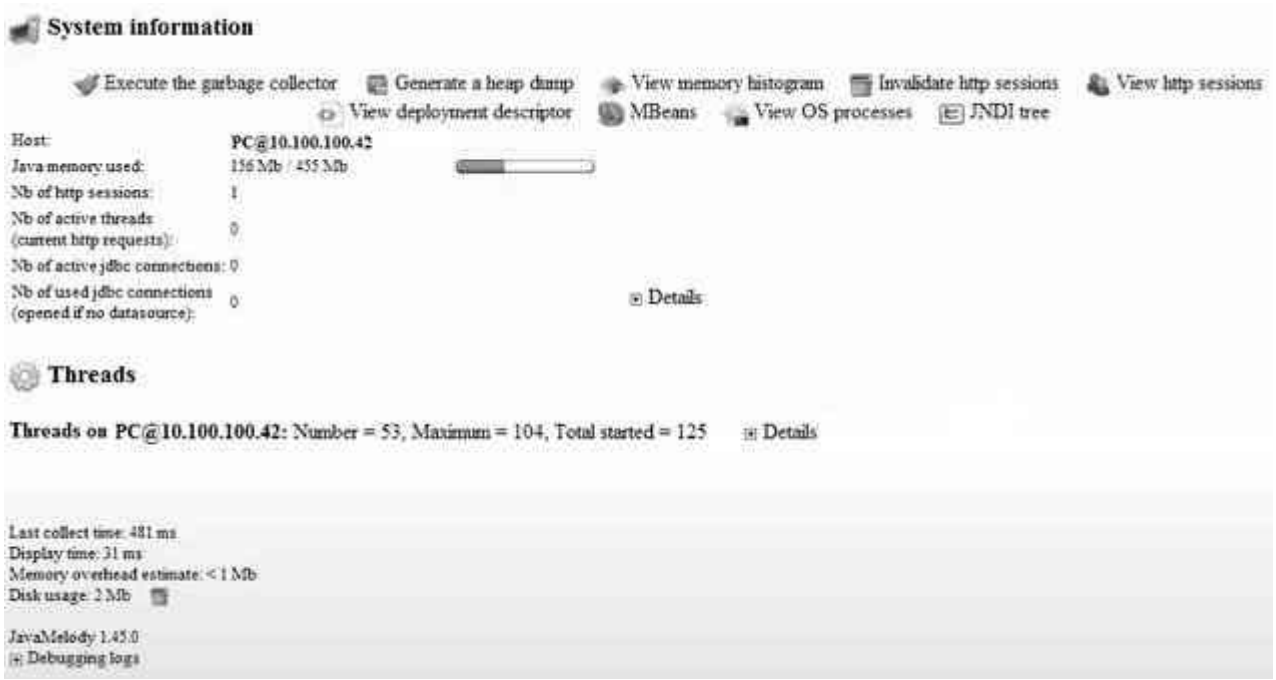
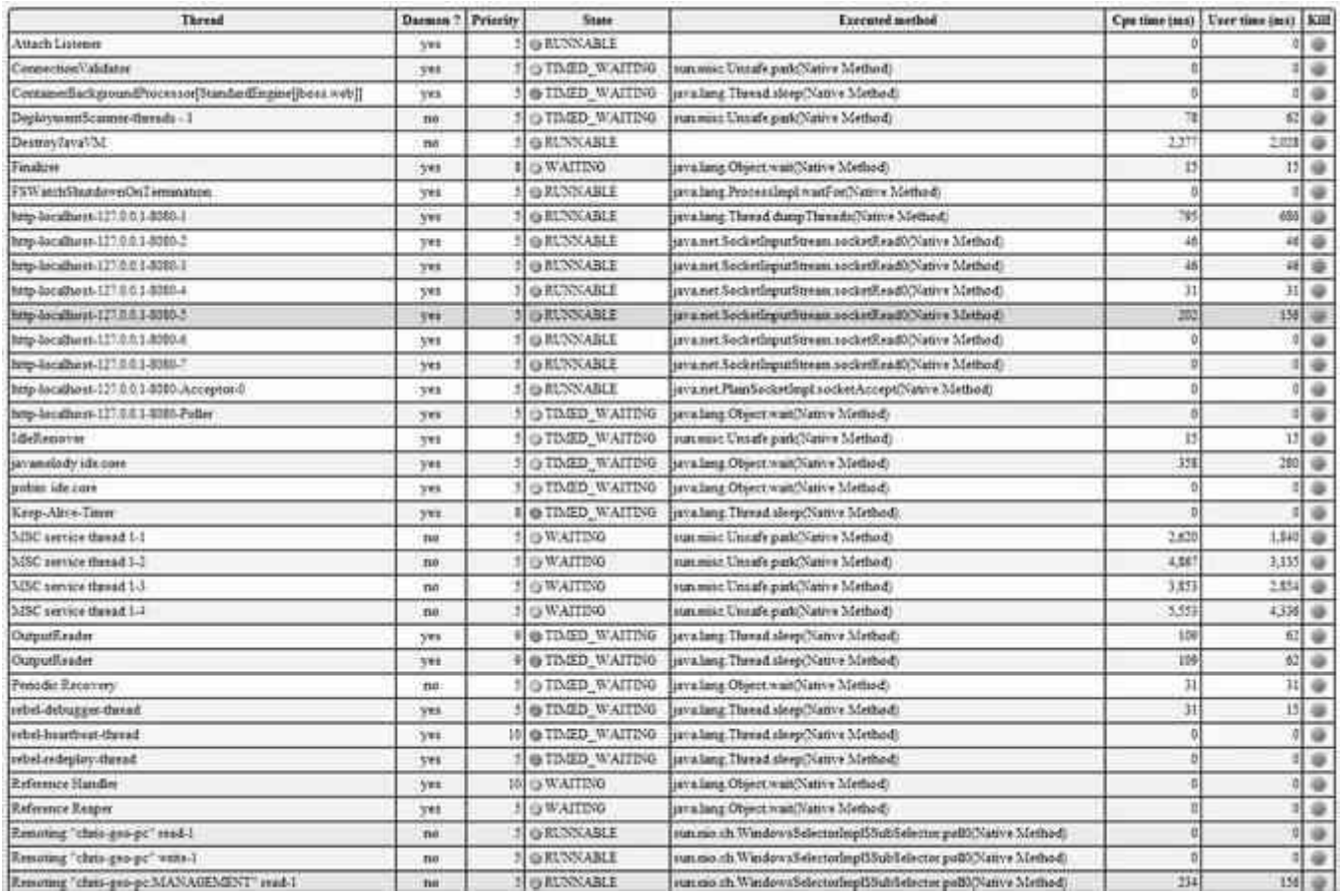


Figure 42: JavaMelody system and thread information

Additionally we can view further detailed information concerning System and Threads. By clicking on “System Details”, we get the screen, shown in Figure 43, which provides us with JVM and OS-related information.

Figure 43: JavaMelody system details

Similarly, we get the screen in Figure 44 by clicking on “Thread Details” which provides information related to the threads running on the OS.



Thread	Daemon ?	Priority	State	Executed method	Cpu time (ms)	User time (ms)	Kill
Attach Listener	yes	5	RUNNABLE		0	0	
ConnectionValidator	yes	7	TIMED_WAITING	sun.misc.Unsafe.park(Native Method)	0	0	
ContainerBackgroundProcessor[StandardEngine[jboss.web]]	yes	5	TIMED_WAITING	java.lang.Thread.sleep(Native Method)	0	0	
DeploymentScanner-threads - 1	no	5	TIMED_WAITING	sun.misc.Unsafe.park(Native Method)	78	62	
DestroyJavaVM	no	5	RUNNABLE		2,277	2,028	
Finalizer	yes	8	WAITING	java.lang.Object.wait(Native Method)	15	15	
FSWatchShutdownOnTermination	yes	5	RUNNABLE	java.lang.ProcessImpl.waitFor(Native Method)	0	0	
http-localhost-127.0.0.1-8080-1	yes	5	RUNNABLE	java.lang.Thread.dumpThreads(Native Method)	795	686	
http-localhost-127.0.0.1-8080-2	yes	5	RUNNABLE	java.net.SocketInputStream.socketRead0(Native Method)	46	46	
http-localhost-127.0.0.1-8080-3	yes	5	RUNNABLE	java.net.SocketInputStream.socketRead0(Native Method)	46	46	
http-localhost-127.0.0.1-8080-4	yes	5	RUNNABLE	java.net.SocketInputStream.socketRead0(Native Method)	31	31	
http-localhost-127.0.0.1-8080-5	yes	5	RUNNABLE	java.net.SocketInputStream.socketRead0(Native Method)	202	358	
http-localhost-127.0.0.1-8080-6	yes	5	RUNNABLE	java.net.SocketInputStream.socketRead0(Native Method)	0	0	
http-localhost-127.0.0.1-8080-7	yes	5	RUNNABLE	java.net.SocketInputStream.socketRead0(Native Method)	0	0	
http-localhost-127.0.0.1-8080-Acceptor-0	yes	5	RUNNABLE	java.net.PlainSocketImpl.socketAccept(Native Method)	0	0	
http-localhost-127.0.0.1-8080-Poller	yes	5	TIMED_WAITING	java.lang.Object.wait(Native Method)	0	0	
IdleRemover	yes	5	TIMED_WAITING	sun.misc.Unsafe.park(Native Method)	15	15	
javamelody idx core	yes	5	TIMED_WAITING	java.lang.Object.wait(Native Method)	358	280	
pubin: idx core	yes	5	TIMED_WAITING	java.lang.Object.wait(Native Method)	0	0	
Keep-Alive-Timer	yes	8	TIMED_WAITING	java.lang.Thread.sleep(Native Method)	0	0	
MSC service thread 1-1	no	5	WAITING	sun.misc.Unsafe.park(Native Method)	2,620	1,840	
MSC service thread 1-2	no	5	WAITING	sun.misc.Unsafe.park(Native Method)	4,867	3,355	
MSC service thread 1-3	no	5	WAITING	sun.misc.Unsafe.park(Native Method)	3,853	2,854	
MSC service thread 1-4	no	5	WAITING	sun.misc.Unsafe.park(Native Method)	5,553	4,336	
OutputReader	yes	6	TIMED_WAITING	java.lang.Thread.sleep(Native Method)	109	62	
OutputReader	yes	6	TIMED_WAITING	java.lang.Thread.sleep(Native Method)	109	62	
Periodic Recovery	no	5	TIMED_WAITING	java.lang.Object.wait(Native Method)	31	31	
rebel-debugger-thread	yes	5	TIMED_WAITING	java.lang.Thread.sleep(Native Method)	31	15	
rebel-heartbeat-thread	yes	10	TIMED_WAITING	java.lang.Thread.sleep(Native Method)	0	0	
rebel-redeploy-thread	yes	5	TIMED_WAITING	java.lang.Thread.sleep(Native Method)	0	0	
Reference Handler	yes	10	WAITING	java.lang.Object.wait(Native Method)	0	0	
Reference Reaper	yes	5	WAITING	java.lang.Object.wait(Native Method)	0	0	
Remoting "chat-geo-pc" read-1	no	5	RUNNABLE	sun.nio.ch.WindowsSelectorImpl$SubSelector.poll0(Native Method)	0	0	
Remoting "chat-geo-pc" write-1	no	5	RUNNABLE	sun.nio.ch.WindowsSelectorImpl$SubSelector.poll0(Native Method)	0	0	
Remoting "chat-geo-pc:MANAGEMENT" read-1	no	5	RUNNABLE	sun.nio.ch.WindowsSelectorImpl$SubSelector.poll0(Native Method)	234	156	

Figure 44: JavaMelody thread details

6 PORTAL SPECIFICATION & IMPLEMENTATION

In this section we provide the specifications of the FIESTA-IoT portal along with technology used to implement the portal.

6.1 Portal Specification

The FIESTA-IoT portal (see Figure 45) should be as user-friendly as possible. The FIESTA-IoT portal provides interaction of the user with the FIESTA-IoT platform. The FIESTA-IoT portal infrastructure will serve as a single entry point for accessing data, services and resources of all the federated IoT testbeds by utilizing modern Web technologies.
The portal host environment is represented by a central Web-based IDE, which provides common accessibility and functionality for the modules of the entire framework of FIESTA-IoT.
For the first version of the portal the menu of the available tools provided is going to follow a static layout common for all the different users (e.g. experimenters, testbed providers, administrators and observers) that could be updated through the component configuration files (see section 6.2.2). In the second version of the portal the role of the user will be taken into account in order to present the appropriate menus dynamically.

Figure 45: FIESTA-IoT portal welcome page

In Figure 46, we can find a sample menu for the Experimenters.

Figure 46: FIESTA-IoT Portal Experimenter menu

In Figure 47, we can find a sample menu for the Testbed Providers.

Figure 47: FIESTA-IoT Portal Testbed Provider menu

Finally in Figure 48, we can find the Management menu for advanced Users (i.e. Administrators).

Figure 48: FIESTA-IoT Portal Management menu

6.2 Portal Implementation

The FIESTA-IoT consortium has identified ZK framework as the most appropriate tool to facilitate the build and set up of the FIESTA-IoT portal. ZK offers a plethora of visualization libraries that not only can help us at building the environment of the FIESTA-IoT portal but also help in the content development of the newly introduced tools of the platform.

6.2.1 How a ZK Component work[69]

A component and a widget work hand in hand to deliver a rich UI experience to a user. The widget traps the user activity and sends an appropriate request to the component. The component then interacts with the developer's application that would respond appropriately telling when the widget should update. This interaction is demonstrated in the following diagram.

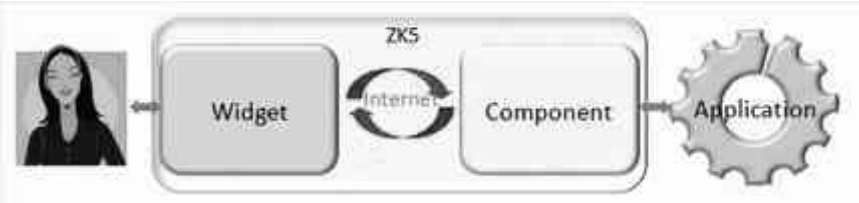


Figure 49: application/component interaction[70]

For example, when an application invokes the `setLabel` method to change the label of a button component, the `setLabel` method of the corresponding button widget (aka peer widget) will be invoked at the client to change the visual appearance (see Figure 50).

69 https://www.zkoss.org/wiki/ZK%20Component%20Development%20Essentials/ZK%20Component%20Overview

70 https://www.zkoss.org/wiki/File:ZKComDevEss_widget_component_application.png

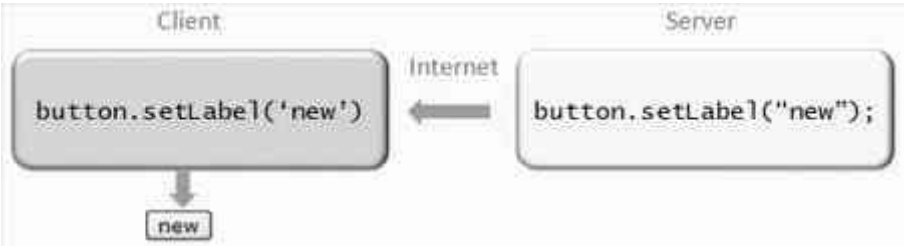


Figure 50: Set label invocation example[71]

When the user clicks the button widget, the `onClick` event will be sent back to the server and notifies the application (see Figure 51).

Figure 51: On click event example[72]

In addition to manipulate a component at the server, it is also possible to control a widget at the client. For example, an application may hide or change the order of the grid columns at the client, while the application running at the server handles the reloading of the grid's content. This technique is called Server+client fusion. It can be used to improve responsiveness and reduce network traffic.

6.2.2 Portal Menu manipulation

The example that follows describes the way to add a new menu and menu items in the current menubar of the FIESTA-IoT portal. Here we want to add a menu, namely "Help", and a menu item named "About Fiesta Portal". The steps that follow show in detail the above task.

Step 1. Create the folder "help" the menu under `webapp/widgets/menu`

[71] https://www.zkoss.org/wiki/File:ZKComDevEss_button_labels.png
[72] https://www.zkoss.org/wiki/File:ZKComDevEss_button_click.png

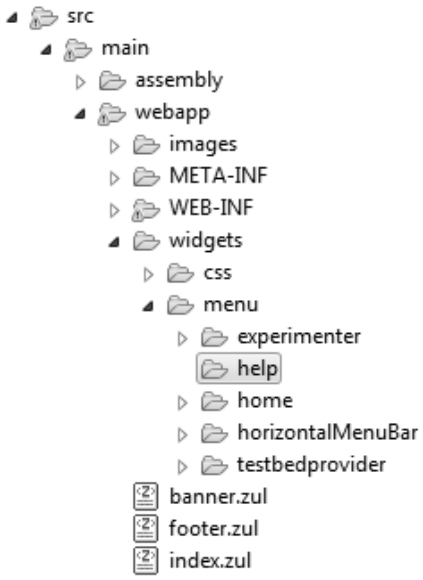


Figure 52: Folder layout

Step 2. Create `zul` pages under `webapp/widgets/menu/help`

You should use the same template as the `templatePage.zul` that can be found under the directory `webapp/widgets/menu`.

- Page 1: help.zul

```
<zk xmlns:n="xhtml">
       <style src="/widgets/css/helpStyle.css"/>

       <div id="container" sclass="container">
              <vlayout id="portalContent" width="100%">
                     <n:h1 class="mo_head">Help</n:h1>
                     <separator bar="true"></separator>
                     <div class="help">
                            <label>We are here to help you</label>
                     </div>
            </vlayout>
      </div>
</zk>
```

Note that you must edit only the red lines. The other lines have to be the same as the sample above.

- Page 2: aboutFiestaPortal.zul

```
 <zk xmlns:n="xhtml">
       <style src="/widgets/css/aboutFiestaPortalStyle.css"/>

       <div id="container" sclass="container">
              <vlayout id="portalContent" width="100%">
                     <n:h1 class="mo_head">About Fiesta Portal</n:h1>
                     <separator bar="true"></separator>
```



```
                    <div class="about">
                            <label>Welcome to Fiesta</label>
                    </div>
            </vlayout>
      </div>
</zk>
```

Note that you must edit only the red lines. The other lines have to be the same as the sample above.

Step 3. Add a css file for the zul pages

You should use the same template as the `templatePageStyle.css` that can be found under the directory `webapp/widgets/css`

- CSS for Page 1:

```
.container{
     padding: 0px;
}

.help{
           width:100%;
     height:850px;
}
```

- CSS for Page 2:

```
.container{
     padding: 0px;
}

.about{
           width:100%;
     height:850px;
}
```

Note that you have to use the above css for each zul page you create. You can add other styles to the components but the main content (for each zul page the main content is the class named container) the main `div` you create (for this zul page the main `div` is the class named about). Moreover, the name “container” of the main content cannot be changed. Otherwise the global style of this div will not be applied.

Step 4. Add the description of the new menu to the config.properties file. The file can be found under src/main/resources.

The description of the menu is in JSON format as follows:

```
menu-4 =
{
```



```
        "menu":"Help",
         "page":"/widgets/menu/help/help.zul",
         "submenu":
         [
                {
                        "name":"About Fiesta Portal",
                        "page":"/widgets/menu/help/aboutFiestaPortal.zul"
                }
        ]
}
```

Note the following:

1. The number in the id of the menu json shows the position of the menu in the menu bar. For example, the id in the json description of our new menu is “menu-4”. This means that it will be the fourth menu element in the menu bar.
2. The object “menu” represents the name of the menu as it will be seen in the menubar.
3. The first object “page” stores the path to the new zul page we created for the menu element. Once “Help” is selected, the `help.zul` page will be shown in the main content of the portal.
4. The object “submenu” describes the menu items that are found under the menu “Help”.
5. The object “name” describes the menu item under the Help menu. Once “About Fiesta Portal” is selected, the `aboutFiestaPortal.zul` page will be shown in the main content of the portal.
6. The object “page” stores the path to the new zul page we created for the menu item.

At this point you should be able to see the new menu and its menu item in the menubar of the FIESTA-IoT portal. It should look like as shown in Figure 53:

Figure 53: MenuBar

And the main content once you select the “About FIESTA-IoT Portal” should look like as shown in Figure 54:

Figure 54: About page

7 DEMONSTRATOR - MOCKUP

The purpose of this section is to provide a FIESTA-IoT portal mockup from the experimenter perspective in order to demonstrate and evaluate the design. Moreover, it serves as a guideline for the FIESTA-IoT developers in order to have a common view towards the end product.

The presented mockup follows the steps of a simple experiment scenario from the experimenter log-in and experiment design up to the experiment execution and visualization.

Step 1. Portal Invocation & User Sign in

The point of entry to the FIESTA-IoT portal is the sign-in screen (as illustrated in Figure 55). The user must authenticate by providing his/her username and password in order to access the FIESTA-IoT portal services. Further, access to the portal is controlled by the user's browser authenticated session Additional to sign-in, the portal also have the following functionality:

- *Forgotten-password*: the user can click the 'forgot password' link on the sign-in screen. The user can then enter their password and request to reset their password. An e-mail with a reset link is then sent to the user, when clicked the user is taken to a FIESTA-IoT security page to enter the new password.
- *Register*: if the user wishes to create a FIESTA-IoT account in order to access the portal and services - they select the register link. They can then fill in their details: name, organization, password etc. A link is then e-mailed to the given address; when the user clicks this link they are identified as the valid user and an account is created in order that they can now log onto the FIESTA-IoT portal (as above).

The FIESTA-IoT portal sign-in is implemented using the `OpenAM` template webpages that act as the front end to the `OpenAM` server[73]. Further, details about the FIESTA-IoT security implementation and the portal sign-in implementation are provided in [22]

[73] https://forgerock.org/openam/

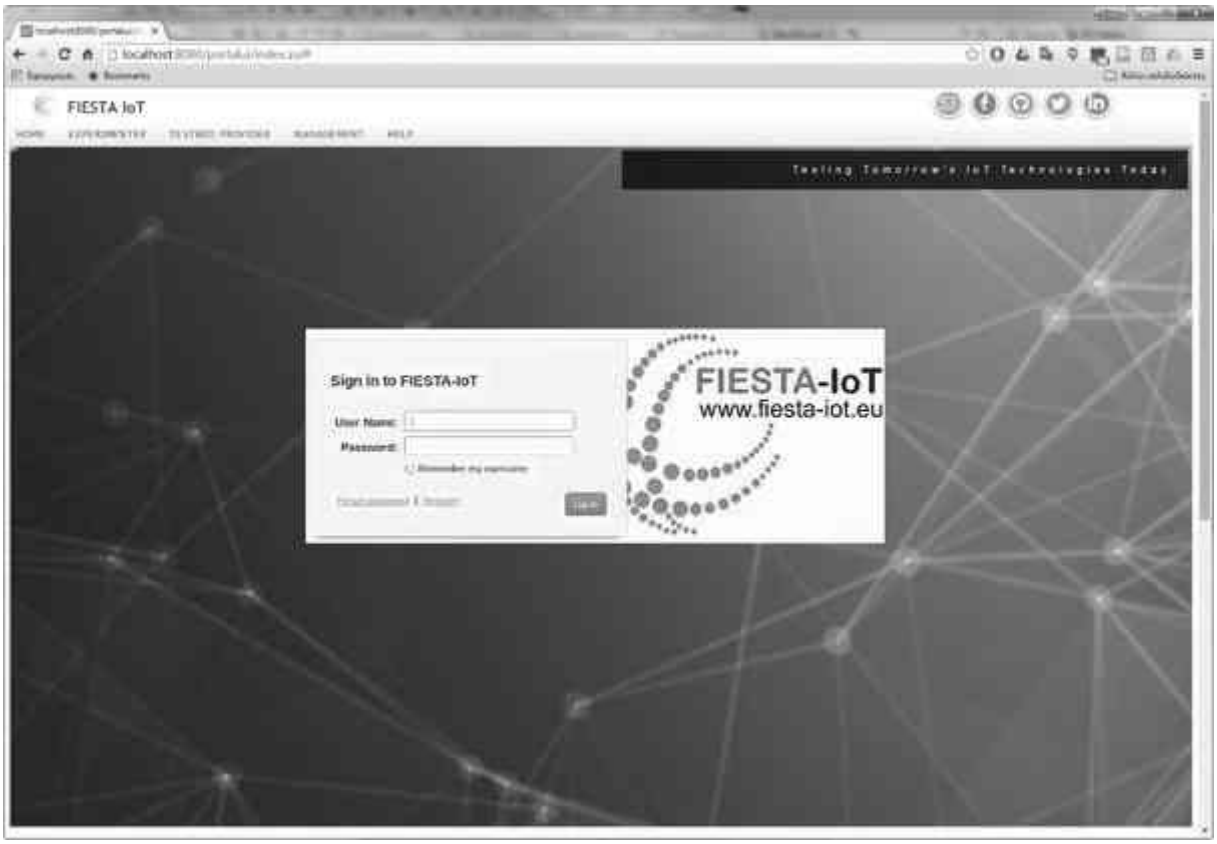


Figure 55: FIESTA-IoT portal Log-in screen

Step 2. User Authentication

As shown in the sequence diagram (see Figure 56) the user after accessing the FIESTA-IoT portal gets redirected to the security component in order to get authenticated and access the experimenter session. Note that, the sequence diagrams presented below provide the interaction of Node-RED nodes with other components within FIESTA-IoT.

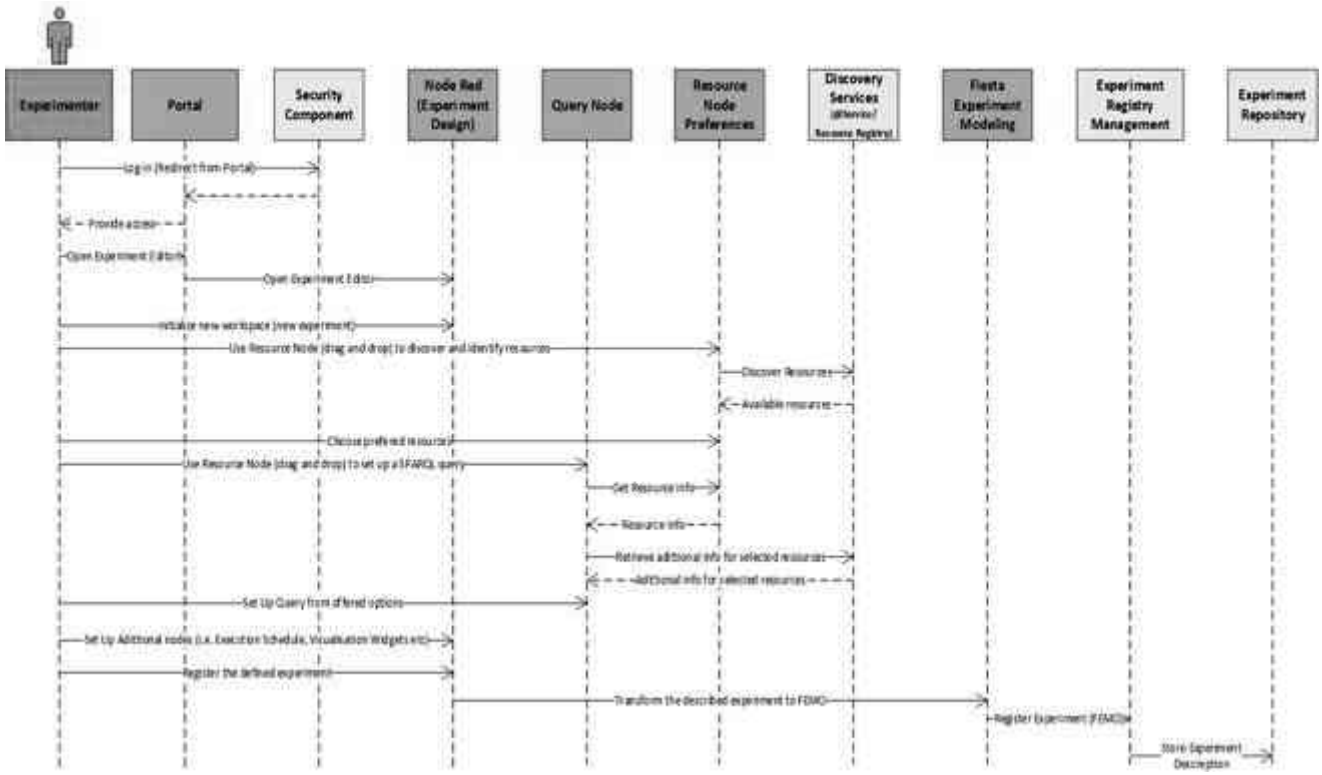


Figure 56 Experiment Definition sequence diagram

As soon as the user gets authorized, he/she is able to access the FIESTA-IoT portal workspace (see Figure 57) and interact with the available menus and tools.

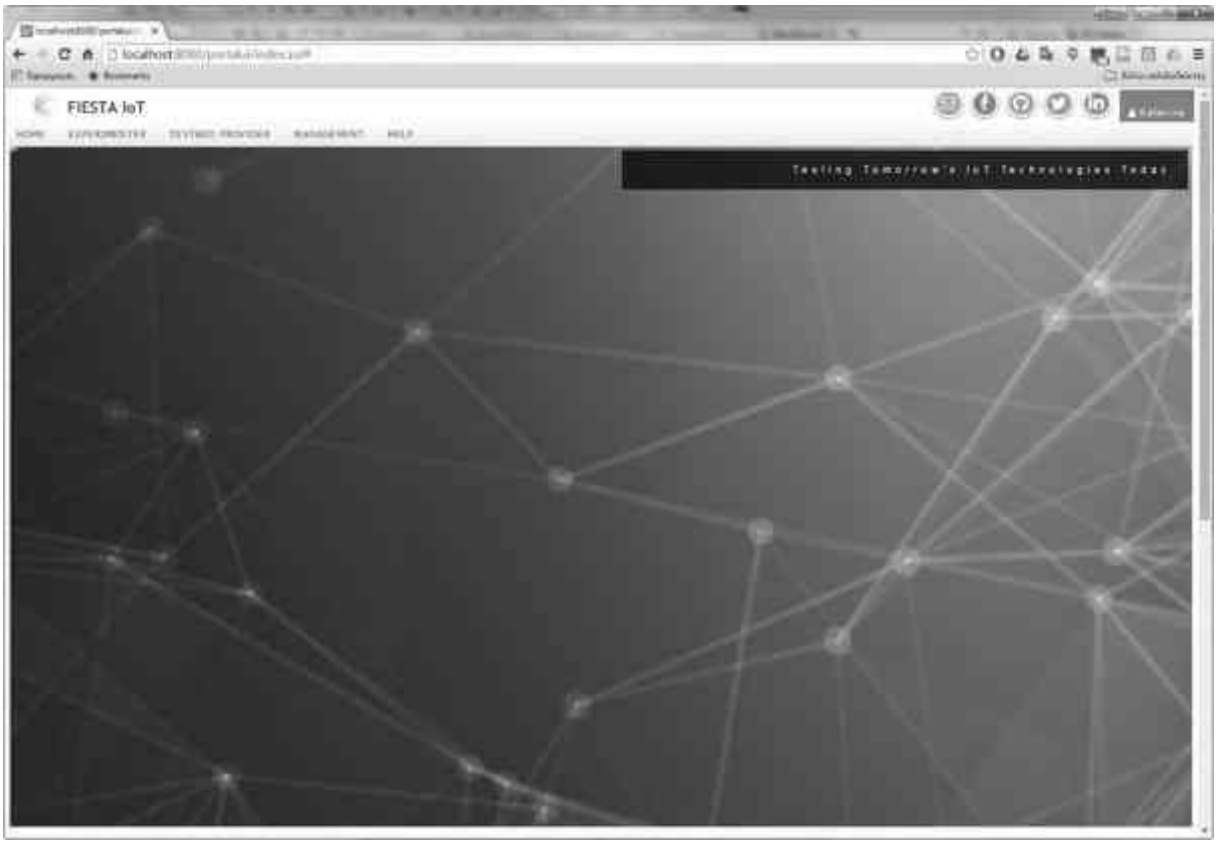


Figure 57: FIESTA-IoT portal empty workspace

Step 3. Initiate Experiment Editor (Node-Red)

The user is now able to access the experimenter menu (see Figure 58) and start a new experiment (see also the sequence diagram in Figure 56).

Figure 58: FIESTA-IoT portal Experimenter Menu

This step opens the Node-RED editor (see Figure 59) with a new tab that hosts the new experiment design (FEMO – FIESTA-IoT Experiment Model Object).

Figure 59: FIESTA-IoT portal Node-RED workspace

Step 4. Experiment Definition

For the experiment definition, the user drags and drops the required nodes to his/her workspace. The first node helps to discover the available resources (sensors) that will be used later on to build the experiment (see Figure 60). Within the discovery options the user will be able to choose from a map the area of interest. The node configuration interface would contact then the FIESTA-IoT Discovery Services (see Figure 56) and retrieve the list of the available resources in order to present them to the user. After listing the available resources the user would be able to choose the ones of interest.

Figure 60: FIESTA-IoT portal Node-RED resource discovery node

The next step in the experiment definition process would be to define the query that is going to be applied at the chosen resources with the help of the `query` node (see Figure 61). The `query` node based on the chosen resources (resource IDs) from the `resource` node would interact with the resource registry (see Figure 56) and retrieve the additional info for the resources (i.e. output type, measurement type) in order to successfully specify the experiment query.

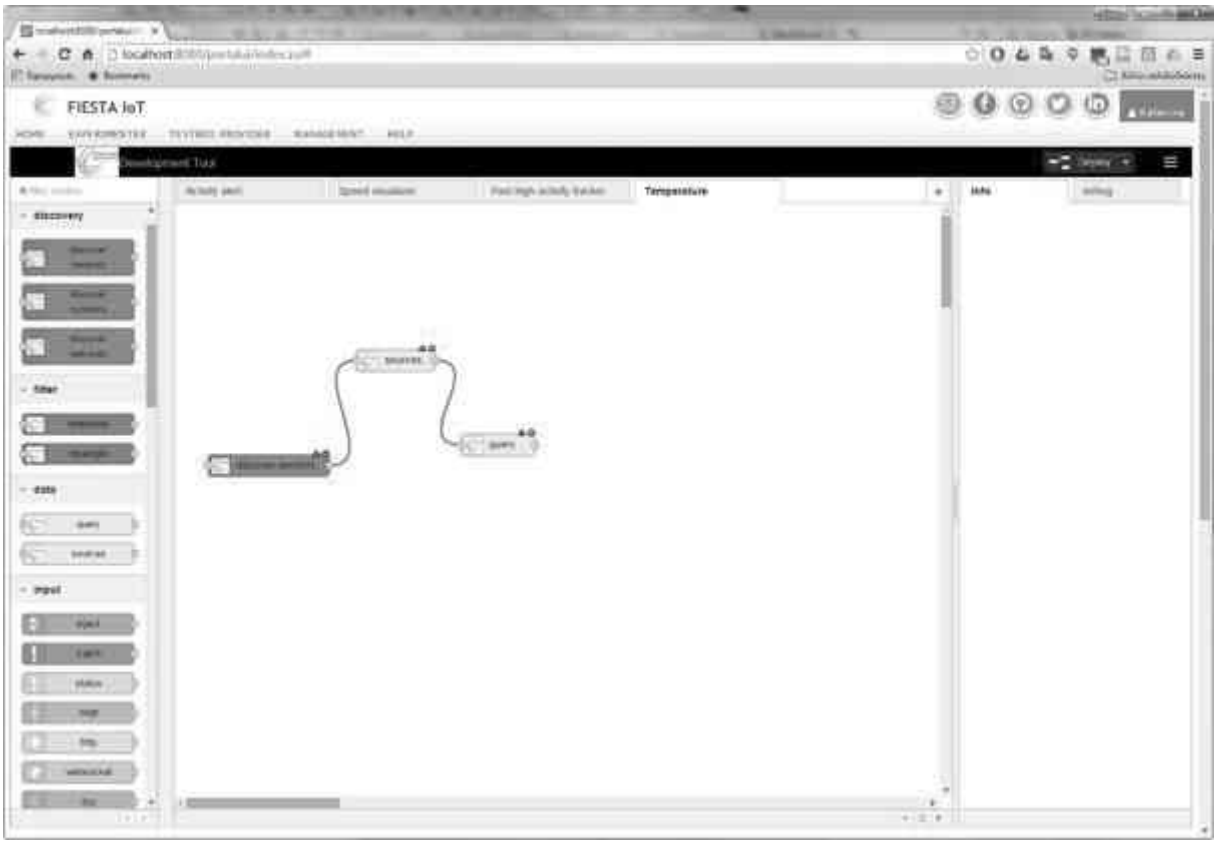


Figure 61: FIESTA-IoT portal query node

Additional to the resource and query nodes, the experimenter can apply reasoning algorithms on top of the query results, define an execution schedule and specify the output format of the results (see Figure 56). These would be achieved by using the equivalent nodes (see Figure 62).

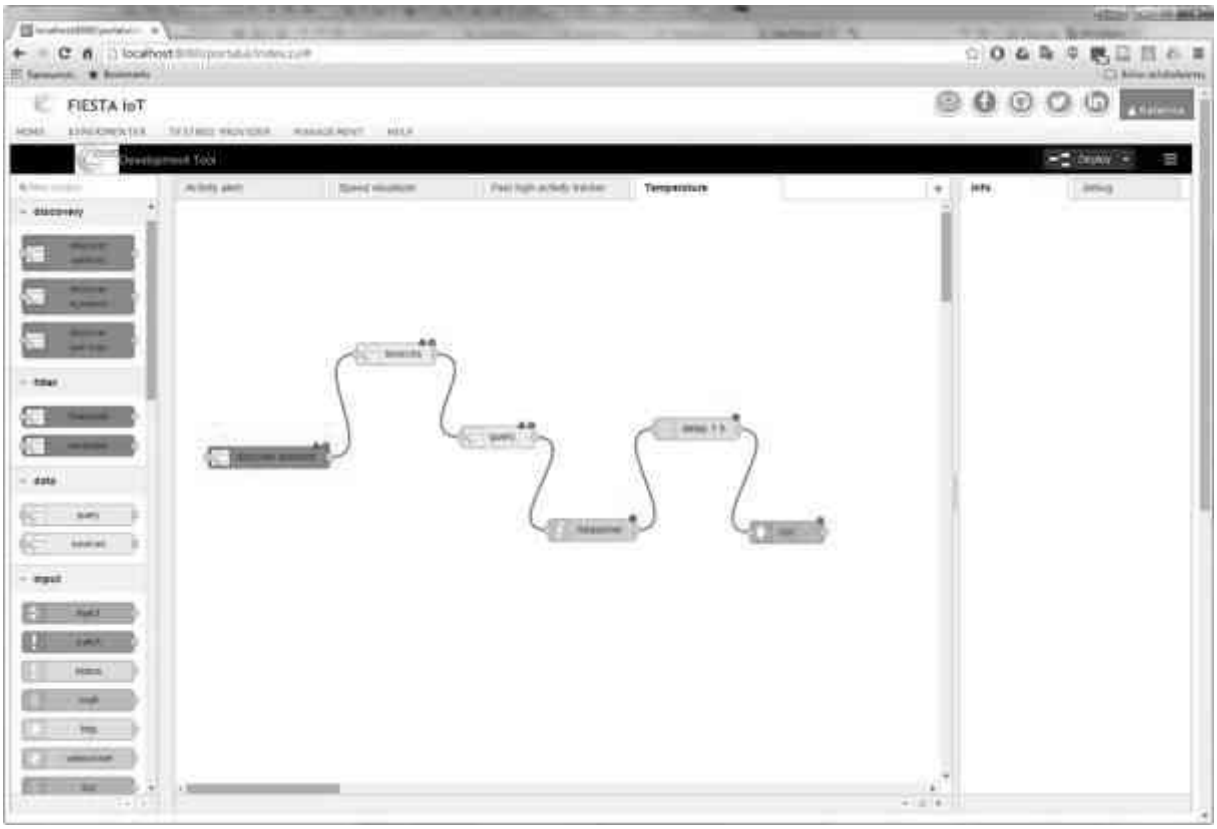


Figure 62: FIESTA-IoT portal scheduling and output nodes

A single workflow will be able to generate a script capable to describe a FISMO that builds an Experiment (FEMO). This includes the following:

- List of service (FISMOs (as a single flow))
- Some of the information that these services will include are:
    - SPARQL script to be executed
    - List of graph URIs that are involved in this experiment.
    - Schedule or trigger for the report to be executed
        - Report if empty functionality will identify if a report will be provided even if there are no new data
    - Output information:
        - Data format (CSV, XLS, JSON, XML, etc.)
        - Presentation widget nodes

Step 5. Experiment Management

After completing the experiment description the experiment should be registered to the experiments repository. This is achieved after it is converted to a FEMO object (from the Node-RED specific script) through the FIESTA-IoT experiment Modeling (see Figure 56). Then, it is sent to the ERM that persists it to the Experiment repository.

Now the experimenter with the help of the EMC (see Figure 63), that can be found under the experimenter menu (see Figure 58), is able to manage his/her defined experiments.

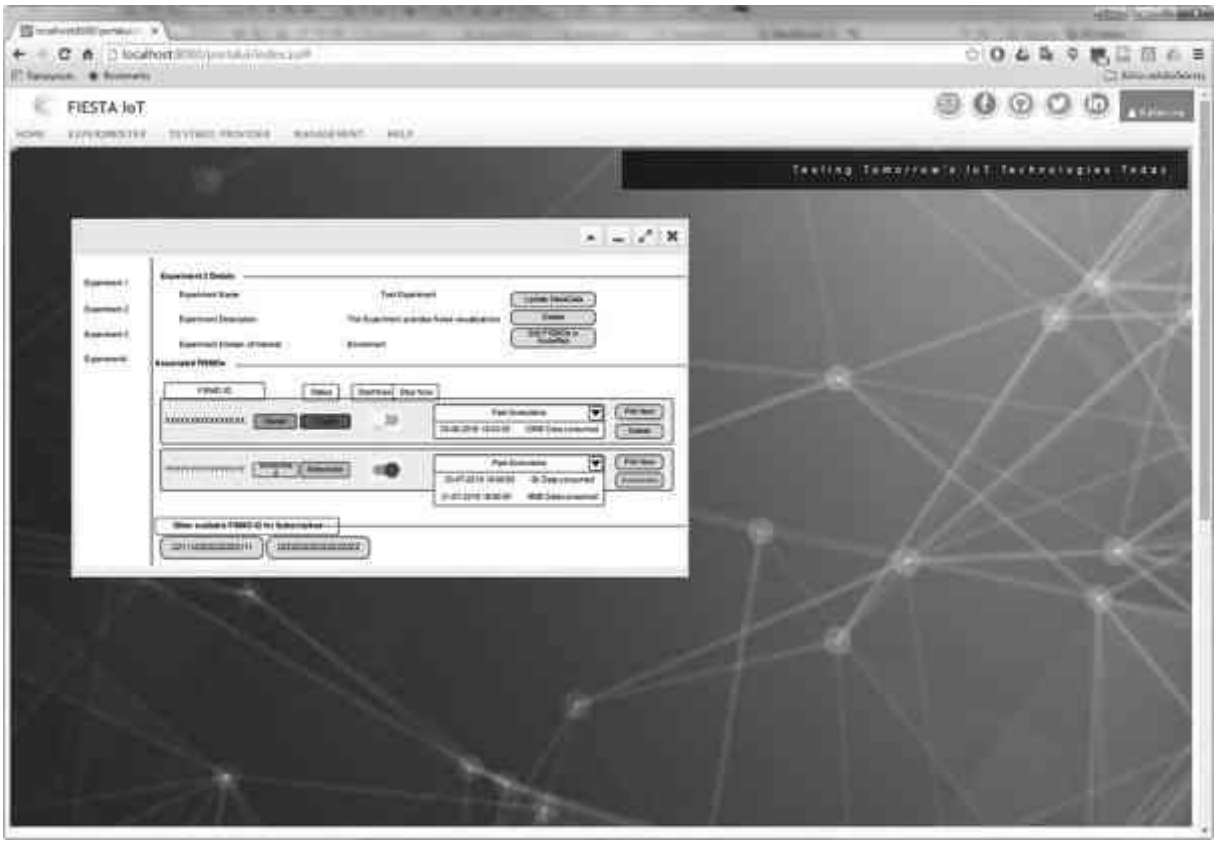


Figure 63 FIESTA-IoT portal Experiment Management within a ZK panel

As shown in Figure 64, the EMC interacts with the Experiment Registry in order to retrieve all the defined experiments. Then the available experiment list is presented to the user where he/she can initiate the experiment process based on the specified schedule at the experiment definition time. This is achieved by forwarding the FEMO configurations to the EEE (see Figure 64) where it gets into a loop (execution schedule) where at the end of each cycle it executes the predefined query and delivers the results till the process is stopped.

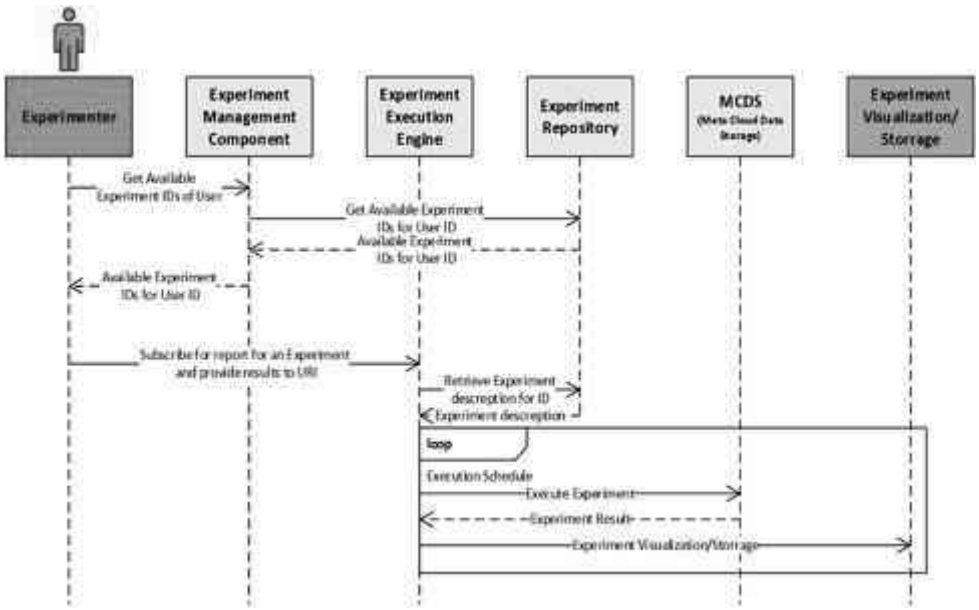


Figure 64 Experiment Execution sequence diagram

Step 6. Result Visualization

The EMC could be utilized by the visualization UI so as to provide the results by using the appropriate widget (predefined by the user at the experiment definition). By choosing the experiment from the EMC, it could open the Visualization UI where the results would be presented (see Figure 65).

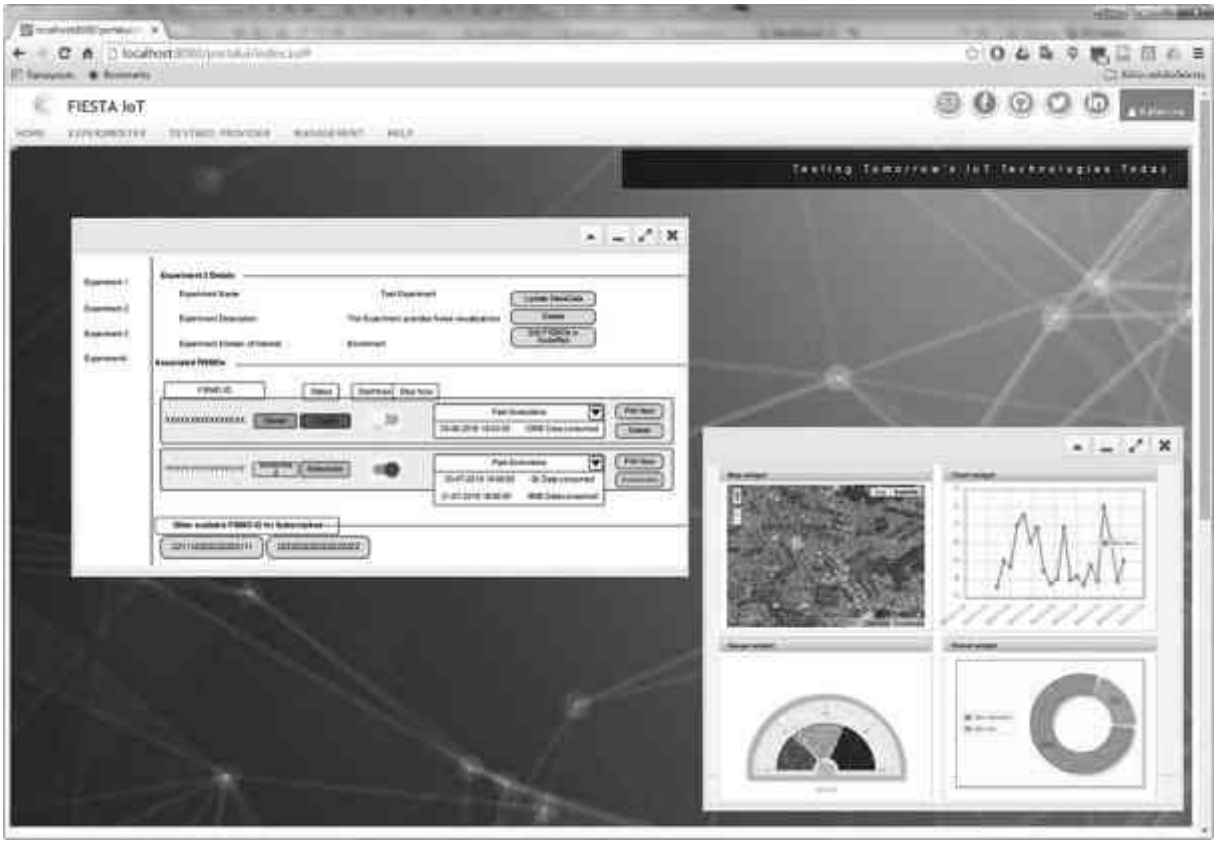


Figure 65: FIESTA-IoT portal Experiment Management and Result visualization within a ZK Panel

8 IMPLEMENTATION - PROTOTYPE

In this section we provide details of the installation procedures for the different components we have built.

8.1 Source Code Availability

The FIESTA-IoT portal, and components referred within this deliverable are offered at the FIESTA-IoT GitLab repository named `core` and is available for download and view at: https://gitlab.fiesta-iot.eu/platform/core/. The latest version of the components is under the `develop` branch[74].

The repository is organized in 4 categories/folders (`doc, modules, ui and utils`) and the components introduced in this deliverable are placed as follows:

- `module`: provides the core modules of the platform
  - `experiment`
    - `eee`: contains APIs relevant to the Experiment Execution Engine. Mainly containing scheduler and polling APIs.
- ui: provides all the modules related to the User Interface
  - `ui.visualization`: contains visualization module
  - `ui.experimentconsole`: contains the user interface for experiment management console
  - `ui.experimentregistry`: contains the user interface for the experiment registry.
  - `ui.portal`: contains the portal environment based on the ZK framework.
  - `ui.experimenteditor`: contains the Node-RED specific source code.
- `utils`: provides utilities related with the platform
- `utils.commons`: include the common utils/objects used in more than 2 projects/components

8.2 Components

All of the described components are `maven` projects and are deployable within `WildFly`[75] container.

8.2.1 Node-RED

8.2.1.1System Requirement

Node-RED requires `Node.js 0.10.x+` and `grunt-cli` to be installed in the target device. `grunt-cli` is used in order to build the application before you can use it.

[74] https://gitlab.fiesta-iot.eu/platform/core/tree/develop

[75] http://wildfly.org/

8.2.1.2Install and Run

After downloading the source code from the FIESTA-IoT Gitlab repository, core pre-requisite modules must be installed by performing following steps:

- `cd [NODE-RED_REPOSITORY]`
- `sudo npm install`
- `sudo npm install –g grunt-cli`

where `[NODE-RED_REPOSITORY]` is the repository of the downloaded Node-RED.

Once installed, we need to build and run the application. To do so, perform:

- `grunt build`
- `node red`

The console will show the results as shown in Figure 66.

```
$ node-red

Welcome to Node-RED
===================

25 Feb 22:51:09 - [info] Node-RED version: v0.13.1
25 Feb 22:51:09 - [info] Node.js  version: v4.2.6
25 Feb 22:51:09 - [info] Loading palette nodes
25 Feb 22:51:10 - [warn] ---------------------------------------
25 Feb 22:51:10 - [warn] Failed to register 1 node types
25 Feb 22:51:10 - [warn] Run with -v for details
25 Feb 22:51:10 - [warn] ---------------------------------------
25 Feb 22:51:10 - [info] User Directory : /home/nol/.node-red
25 Feb 22:51:10 - [info] Server now running at http://127.0.0.1:1880/
25 Feb 22:51:10 - [info] Creating new flows file : flows_noltop.json
25 Feb 22:51:10 - [info] Starting flows
25 Feb 22:51:10 - [info] Started flows
```

Figure 66: Console response at running Node Red

We could now open the Node-RED using `http://[HOST]:[PORT]`. Here `[HOST]` is the host and `[PORT]` is the port in any given web browser.

8.2.2 Portal

The portal is deployed within a `WildFly` container and is using `Maven` as project management. The prototype runs on Windows, Linux, Mac OS X, and Solaris. In order to run the prototype, you need to ensure that `Java 8+` and `WildFly` are installed and/or available on your system. In order to build the prototype you will also need Maven.

Before attempting to deploy and run the prototype applications, make sure that you have started `WildFly`.

More details about the specific versions of the tools and libraries that have been used for the development or that are required for the deployment and execution of the prototypes are given in the section below.

8.2.2.1Install and Run

The prototype has been implemented as a Maven-based web application. Below [WILDFLY_REPOSITORY] indicates the root directory of the WildFly distribution, and [PROJECT_HOME] indicates the root directory of the project.

In order to configure the prototype,

1. make sure that all properties listed in [PROJECT_HOME]/src/main/resources/fiesta-iot.properties have the appropriate values,
2. copy that file into [WILDFLY_REPOSITORY]/standalone/configuration, and
3. issue the following commands:

In order to **build** the prototype, run the following command in [PROJECT_HOME]:

```
mvn clean package
```

Finally, in order to deploy the prototype, run the following command in [PROJECT_HOME]:

```
clean package org.wildfly.plugins:wildfly-maven-plugin:deploy
```

The last step assumes that WildFly is already running on the machine where you run the command.

Alternatively copy the produced (from the build process above) ui.portal-X.war file from the target directory ([PROJECT_HOME]/target/), into the standalone/deployments directory of the WildFly distribution, in order to be automatically deployed.

If the deployment has been successfully completed, you will be able to access the portal using the following URL: http://[HOST]:[PORT]/portalui where [HOST] is the host and [PORT] the port that WildFly uses.

8.2.2.2Deployment instructions for Eclipse

Once you open the project in Eclipse as an existing Maven project, you can deploy the portal in the Wildfly server. Please note that your WildFly server has to be running. You need to follow the steps described below to achieve the deployment.

1. Create a new Run Configuration
2. As the “Base Directory” you have to set the path that navigates to the portal project
3. In the Goals option you have to add the following maven command:
   ```
   clean package org.wildfly.plugins:wildfly-maven-plugin:deploy
   ```
4. Now, apply the changes and run the above configuration you have created.

After this, you can navigate from your browser to http://[HOST]:[PORT]/portalui and see the FIESTA-IoT portal where [HOST] is the host and [PORT] the port that Wildfly uses.

8.2.3 Experiment Execution Engine

8.2.3.1System Requirements

The following Table 3 lists the system requirements that are needed to build and deploy the component. Once the component is successfully deployed its services can be accessed via `http://[HOST]:[PORT]/scheduler/scheduling/` and `http://[HOST]:[PORT]/scheduler/monitoring/` where `[HOST]` is the host and `[PORT]` the port that `Wildfly` uses.

Table 3: System Requirements for EEE

Requirements	Version
Wildfly	10.0.1.Final
Java Platform, Standard Edition	1.8.0_25
Maven	3.1.1
MySQL	5.6.21+

8.2.3.2Dependencies

The EEE requires certain dependencies that form the core of the component. These include those listed in the Table 4 (Note we do not list all the dependencies needed. To know the complete list we redirect the readers to the pom.xml of the component that is made available via https://gitlab.fiesta-iot.eu/platform/core/):

Table 4: Dependencies for EEE Scheduling component

Requirements	Version
Quartz	2.2.1
Quartz-Job	2.2.1
Hibernate	3.2.3.GA
MySQL-connector	5.1.34

8.2.3.3Install and Run

In this section we provide necessary commands to install and run the component successfully. Here we assume that the `Wildfly` is already deployed and is running at `[HOST]` and `[PORT]` with all the configuration setup (as described in the deliverable 3.2.1). Following set of the commands should be followed in-order to successfully install the component.

To install `ActiveMQ` in case it is not installed, perform following steps:

- `cd [WILDFLY_REPOSITORY]/standalone/deployments/`
- `wget http://activemq.apache.org/path/tofile/apache-activemq-x.x.x-bin.tar.gz`

- `tar zxvf activemq-x.x.x-bin.tar.gz`

In the above commands `x.x.x` means the latest version of. To compile the component, we perform the steps as stated in Section 8.2.2.1. Alternatively, we can also perform the following:

- `cd [PROJECT_HOME]`
- `mvn clean package`

To deploy on `Wildfly` perform the following step:

- `mvn wildfly:deploy`

In case the `mvn wildfly:deploy` does not execute, perform the following step:

- `cp [PROJECT_HOME]/target/[PROJECT_NAME].war [WILDFLY_HOME]/standalone/deployments/`

8.2.4 Platform's Modules Runtime Monitoring

8.2.4.1System Requirements

The requirements for this project are `Java SDK 1.8+` and `Maven 3.0+`. The application this project produces is `WildFly` Application Platform 10 compliant and also executes on it.

8.2.4.2Download

`JavaMelody` is already declared as a dependency in the Host Environment (IDE) module's `pom.xml`. Therefore, no particular action is needed to download `JavaMelody` for the specific project.

In order to include `JavaMelody` as a dependency in the IDE, the following code snippet has been added to `pom.xml`.

```
<dependency>
      <groupId>net.bull.javamelody</groupId>
      <artifactId>javamelody-core</artifactId>
      <version>1.60.0</version>
</dependency>
```

In a non-Maven project, the `JavaMelody.jar` files and the dependent libraries may be downloaded[76].

8.2.4.3Deploy From the Source Code

As mentioned above, `JavaMelody` is integrated into the IDE host environment, so no separate deployment is necessary.

[76] https://github.com/javamelody/javamelody/releases

9 CONCLUSION

Via this deliverable, we have provided our advancements with respect to how experimenters could create, deploy and manage experiments, giving as well an overview about the FIESTA-IoT portal with respect to experimenters. Note that the portal is not only limited to the tools that are applicable to experimenters but it also supports tools available for testbed owners (some of which are presented in [18]). Further, other user roles defined within FIESTA-IoT would also use the FIESTA-IoT portal.

To build the tools and the portal as a part of this deliverable, we took inspiration from our experience with building tools within WP3 and already existing EU projects framework. The main outcome of this deliverable is a demonstrator (portal) along with a set of different-purpose tools, such as scheduler and polling that form the core of the EEE are reported. Further, tools such as the Experiment Management Console and the Testbed Monitoring tools that form a part of Web Browsing and configuration FC and Performance Monitoring are also introduced. As it is an on-going effort, some of the tools are not yet available to their full capacity in the first release of this deliverable. As the tools are also to be made available for the gradual Open Calls process, these tools will be well documented and the APIs within will be supported by the documentation where the experimenters can possibly execute the APIs if they have the right credentials.

The tasks within WP4 especially those reported via this deliverable i.e., Task 4.4 and Task 4.5, have a lifespan until month 33 of the project (also the month for the 2nd version of this deliverable). Thus, a lot of future work is envisioned. This basically includes the finalization of the tools that are reported, continuous integration with other components and updates to currently available tools. It is also not to ignore the fact that lot of things will evolve. These would mainly include: the specification of the APIs and portal.

REFERENCES

[1] FIESTA-IoT, “Deliverable 2.3: Specification of Experiments, Tools and KPIs.”

[2] FIESTA-IoT, “Deliverable 5.1: Experiments Design and Specification.”

[3] FIESTA-IoT, “Deliverable 4.1.1: EaaS Model Specification and Implementation,” 2016.

[4] FIESTA-IoT, “Deliverable 2.4: FIESTA-IoT Meta Cloud Architecture,” 2015.

[5] FIESTA-IoT, “Deliverable 2.1: Stakeholders Requirements.”

[6] N. Kefalakis, J. Soldatos, and Others, “OpenIoT Project Deliverable D4.4.2: OpenIoT Integrated Development Environment b,” Jul. 2104.

[7] J. Soldatos, K. Roukounaki, and Others, “Vital Project Deliverable D5.2.2: Development and Deployment Environment V2,” Vital2016b, Jun. 2016.

[8] L. Bracco, P. D. Zovo, and Others, “Vital Project Deliverable D5.1.3: S Management Services over Federated IoT Platforms V3,” Apr. 2016.

[9] A. Gyrard, S. K. Datta, C. Bonnet, and K. Boudaoud, “Cross-Domain Internet of Things Application Development: M3 Framework and Evaluation,” in *3rd International Conference on Future Internet of Things and Cloud*, 2015, pp. 9–16.

[10] A. Gyrard, C. Bonnet, K. Boudaoud, and M. Serrano, “Assisting IoT Projects and Developers in Designing Interoperable Semantic Web of Things Applications,” in *IEEE International Conference on Data Science and Data Intensive Systems*, 2015, pp. 659–666.

[11] A. Gyrard, S. K. Datta, C. Bonnet, and K. Boudaoud, “Standardizing Generic Cross-Domain Applications in Internet of Things,” in *3rd Workshop on Telecommunications Standards, Part of IEEE Globecom*, 2014.

[12] FIESTA-IoT, “Deliverable 3.1.1: Semantic models for testbeds, interoperability and mobility support and best practices,” 2016.

[13] A. Gyrard, C. Bonnet, and K. Boudaoud, “Demo paper: Helping IoT application developers with Sensor-based Linked Open Rules,” in *7th International Workshop on Semantic Sensor Networks, in conjunction with the 13th International Semantic Web Conference*, 2014.

[14] A. Gyrard, G. Atemezing, C. Bonnet, K. Boudaoud, and M. Serrano, “Reusing and Unifying Background Knowledge for Internet of Things with LOV4IoT,” in *4th International Conference on Future Internet of Things and Cloud*, 2016.

[15] A. Gyrard, C. Bonnet, K. Boudaoud, and M. Serrano, “LOV4IoT: A second life for ontology-based domain knowledge to build Semantic Web of Things applications,” in *4th International Conference on Future Internet of Things and Cloud*, 2016.

[16] A. Willner, “Semantic-based Management of Federated Infrastructures for Future Internet Experimentation,” TU Berlin.

[17] B. Vermeulen and Others, “Fed4FIRE Deliverable D2.4: Second Federation Architecture,” Mar. .

[18] FIESTA-IoT, “Deliverable 3.2.1: Specification and implementation of common Testbed interfaces,” 2016.

[19] D. Conway-Jones, “Wiring the Internet of Things with Node-RED,” San Francisco, Mar. .

[20] FIESTA-IoT, “Deliverable 2.2: Analysis of IoT Platforms and Testbeds,” 2015.

[21] FIESTA-IoT, “Deliverable 4.3.1: Tools and Techniques for Managing Interoperable Data sets,” 2016.

[22] FIESTA-IoT, “Delivarable 4.2.1: Techniques for Secure Access and Reservation of Resources,” 2016.

2016 FIESTA-IoT

APPENDIX - I NODE-RED CORE NODE SET

This section gives a short overview of some of the nodes that are available in Node-RED core node set. In order to better understand how these nodes communicate (through messages), refer to Appendix III.

- **http in**

`http in` nodes are input nodes that listen for HTTP requests, thus allowing the creation of simple web services. These nodes only receive HTTP requests; they do not respond back to them. This is something that is done further down the `flow` by http response or function nodes. The messages sent by these nodes contain both the request (that has been received) and the response (that should be later filled and sent).

Some of the properties of `http in` nodes are the `URL`, where the node accepts HTTP requests, and the `HTTP method` that the node supports. The `URL` is relative to a Node-RED configuration property called `HTTP node root` that specifies the root `URL` for nodes that provide HTTP endpoints.

- **http response**

`http response` nodes are output nodes that send responses back to HTTP requests that `http in` nodes have previously received. It is either an `http response` node or a `function` node that can be used together with `an http in` node for the creation of simple web services. The messages sent to these nodes contain information about the response that they should send out.

- **http request**

`http request` nodes can be used to make HTTP requests. The messages received by these nodes contain information about the request they should make, whereas the messages sent by them contain information about the response they got for the corresponding request.

The properties of these nodes include, among others, the `HTTP method` to use, the `URL` to make the request to, what the node should expect to get back as a `response` (UTF-8, binary or JSON), as well as whether basic authentication is required.

- **function**

`function` nodes represent function blocks written in JavaScript that can do practically anything. The message that a function node receives is passed to the function block as a JavaScript object called `msg`. The function block is expected to either return the message(s) that should be passed to the next nodes, or to send them itself.

- **mqtt out**

An **`mqtt out`** node is an output node that connects to a Message Queuing Telemetry Transport (MQTT) broker, and publishes a message to a topic. The message that an `mqtt` out node receives contains details about the message to publish, as well as how and where to publish it.

The properties of `mqtt` out nodes include the QoS level, whether the message should be retained, details about how to connect to the broker, and the topic where the message should be published.

- **template**

`Template` nodes create new messages based on the messages they receive and a provided template. The supported templates are: mustache, HTML, JSON and markdown. A `template` node uses its input message to fill in the gaps in a template, thereby producing its output message. For example, a template node can be used to create simple web pages.

2016 FIESTA-IoT

APPENDIX - II NODE-RED SAMPLE FLOW

This section presents a sample flow built using Node-RED. The flow:

- receives `HTTP GET` requests at `/say-hello`
- retrieves the value of the query parameter name
- responds with an HTML page that says hello

In order to implement this flow, the following nodes are required:

- an `http in` node
- a `template` node
- an `http response` node

We drag the above nodes from the palette into the workspace. We set the method of the http in node to `GET` and its `URL` to `/say-hello`. We provide the following template to the `template` node.

Table 5: Template

```
<html>
  <head>
    <title>Say hello to {{req.query.name}}.</title>
  </head>
  <body>
    <h4>Hello, {{req.query.name}}!</h4>
  </body>
</html>
```

Finally, we wire the output of the `http in` node to the input of the `template` node, and the output of the `template` node to the input of the `http response` node. The final result is shown in Figure 67.

2016 FIESTA-IoT

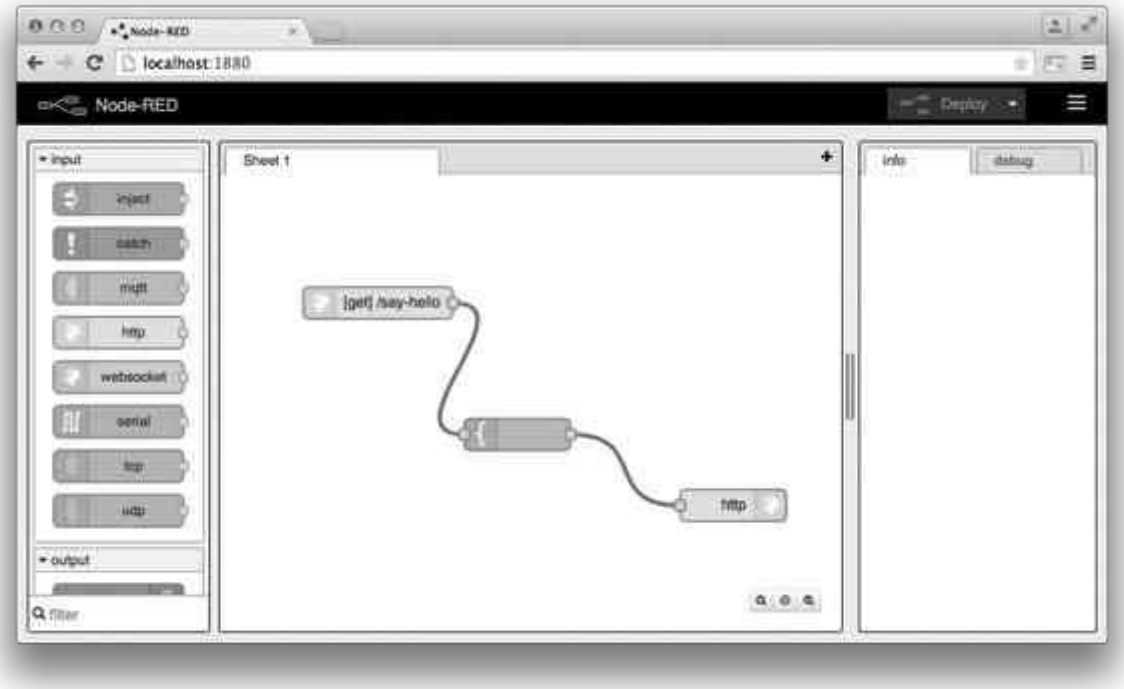


Figure 67: Hello world in Node-RED.

As mentioned in Section 3.3.2.1.1, flows in Node-RED are represented using JSON. The JSON representation of the flow described above follows.

Table 6: JSON representation of Hello world flow.

```
[{"id":"3c3ee40c.c3c11c","type":"http in","name":"","url":"/say-
hello","method":"get","swaggerDoc":"","x":149,"y":122,"z":"a9c02355.
563fe","wires":[["37cbc6d3.c8343a"]]},{"id":"f444ed7.f0bbb1","type":
"http
response","name":"","x":483,"y":313,"z":"a9c02355.563fe","wires":[]}
,{"id":"37cbc6d3.c8343a","type":"template","name":"","field":"payloa
d","format":"handlebars","template":"<html>\n    <head>\n        <ti
tle>Say hello to
{{req.query.name}}.</title>\n    </head>\n    <body>\n        <h4>He
llo,
{{req.query.name}}!</h4>    \n    </body>\n</html>","x":280,"y":246,
"z":"a9c02355.563fe","wires":[["f444ed7.f0bbb1"]]}]
```


APPENDIX - III NODE-RED RUNTIME

A flow designed with the Node-RED editor is nothing more than a sketch; the nodes it contains do not work by themselves (an `http in` node, for example, does not listen for incoming HTTP requests). In order to make it work, the flow needs to be deployed on what is called the `Node-RED runtime` (i.e. the server side of Node-RED).

A `flow configuration` is a JSON-formatted description of one or more flows. This is an example of a configuration that contains the description of the flow we presented in the previous section (marked in red).

Table 7: Flow configuration example.

```
[{"type":"tab","id":"a9c02355.563fe","label":"Sheet
1"},{"id":"13257eec.669069","type":"websocket-
listener","path":"/ws/stations","wholemsg":"false"},{"id":"3c3ee40c.
c3c11c","type":"http in","name":"","url":"/say-
hello","method":"get","swaggerDoc":"","x":149,"y":122,"z":"a9c02355.
563fe","wires":[["37cbc6d3.c8343a"]]},{"id":"f444ed7.f0bbb1","type":
"http
response","name":"","x":483,"y":313,"z":"a9c02355.563fe","wires":[]}
,{"id":"37cbc6d3.c8343a","type":"template","name":"","field":"payloa
d","format":"handlebars","template":"<html>\n    <head>\n        <ti
tle>Say hello to
{{req.query.name}}.</title>\n    </head>\n    <body>\n        <h4>He
llo,
{{req.query.name}}!</h4>    \n    </body>\n</html>","x":280,"y":246,
"z":"a9c02355.563fe","wires":[["f444ed7.f0bbb1"]]}]
```

At any point in time, there is a single `active flow configuration` in Node-RED that describes the deployed flows at that time. So, if we want to change the flows that are deployed on the Node-RED runtime, all we need to do is replace the active flow configuration with a new one.

Every time we start a new session with the flow editor, what we see is the flows that are deployed on the runtime at this particular point in time. If we make any changes to these flows (e.g. if we add a new flow, or if we change a node in an existing flow), and we want to deploy these changes to the runtime, we are given three options:

- to deploy only the modified nodes.
- to deploy only the modified flows.
- to deploy everything.

In all three cases, some nodes need to stop before the new flow configuration can be applied. In the first case, these are only the modified nodes, in the second case these are only the nodes in the modified flows, whereas in the third case all nodes are stopped.

Following that, the runtime parses the new flow configuration, instantiates all the nodes described in it (based on the properties set for each one of them), and starts them. It also saves the configuration in a file, and uses that file during its subsequent startups.

The Node-RED runtime can be thought of as the place where nodes live. Nodes are created, started and stopped. A node can be stopped and re-started more than once

during its lifetime. A node can also exchange messages with other nodes; in particular a node can receive messages from the up-stream nodes in a flow, and send messages to the down-stream nodes. Finally, while running, a node can share its current `status` with the Node-RED editor.

Once we deploy the flow we designed in the previous section (by pressing the Deploy button in the Node-RED editor), we can use it, as shown in Figure 68.

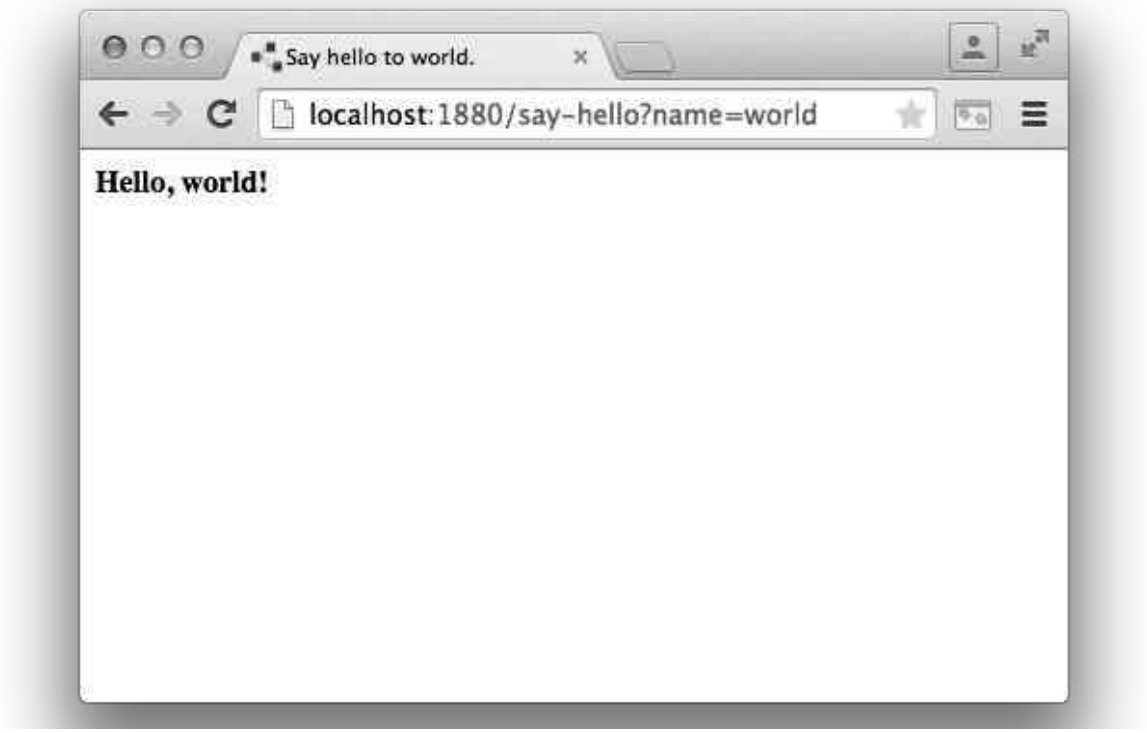


Figure 68: Node-RED "Hello World" example.

APPENDIX- IV NODE-RED EXPERIMENT DESIGN

Experiment Design is a service where the user designs a novel service using the `resource nodes`, `operational` and or `visual` nodes. The experimenter could get the required information of any node that is present in the Node-RED by simply checking the info status of the node located right of the Node Red environment. The following Figure 69 and Figure 70 are few examples of Experiment Design.

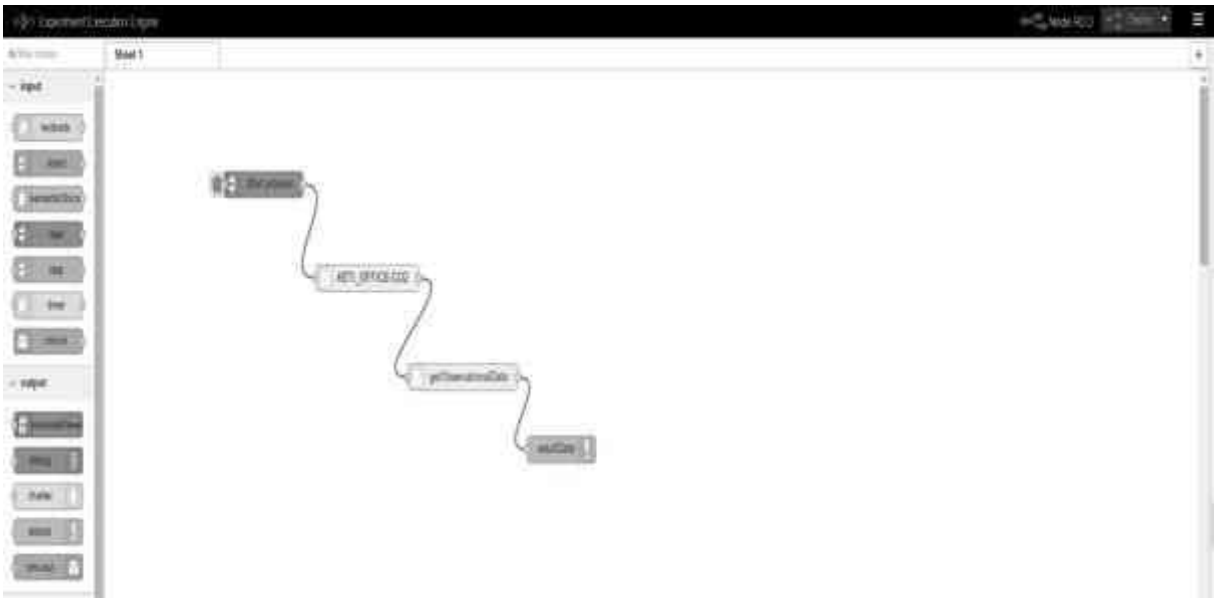


Figure 69: Experiment Design using CO2 resource node and Popup node

Figure 70: Experiment Design using Chart node

2016 FIESTA-IoT

