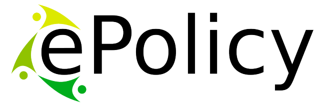

Engineering the Policy-making Life Cycle

Deliverable D8.3
Functional and Performance Evaluation of the integrated decision support system

Document type:	Deliverable
Dissemination Level:	Public
Editor:	Ambra Molesini and Federico Chesani
Document Version:	1.0
Contributing Partners:	UNIBO
Contributing WPs:	WP8
Estimated P/M (if applicable):	A number
Date of Completion:	30 September 2014
Date of Delivery to EC	30 September 2014
Number of pages:	26

ABSTRACT

Deliverble D8.3 consists of the evaluation of the prototype of the decision support system by investigating technical aspects, exploiting classical software engineering techniques like coverage of the functional/ non functional requirements identified within Task 2.1. Also performance evaluations of the overall prototype will be analysed, to provide a comprehensive evaluation of the system as a whole.

The project is co-funded by the European Community under the Information and Communication Technologies (ICT) theme of the Seventh Framework Programme (FP7/2007-2013). Grant Agreement n°288147.

Authors of this document:

Ambra Molesini[1], Federico Chesani[1], Michela Milano[1]

[1]: DISI, University of Bologna
email: {ambra.molesini, federico.chesani, michela.milano }@unibo.it

Copyright © by the ePolicy Consortium
The ePolicy Consortium consists of the following partners: University of Bologna; University College Cork, National University Ireland, Cork; University of Surrey; INESC Porto, Instituto de Engenharia de Sistemas e Computadores do Porto, Fraunhofer – Gesellschaft zur Foerderung der Angewandten Forschung E.V.; Regione Emila-Romagna; ASTER – Società Consortile per Azioni; Università degli Studi di Ferrara.

Possible inaccuracies of information are under the responsibility of the project team. The text reflects solely the views of its authors. The European Commission is not liable for any use that may be made of the information contained therein.

Contents

1 Introduction **4**

2 Evaluation of Functional Requirements **6**

2.1 Requirements derived from the Roles management: the "Login page" and the security issues 6

2.2 Requirements derived from the "policy maker" view 9

2.3 Requirements derived from the "Domain Expert" view 10

2.4 Requirements derived from the "Administrator" view 11

2.5 Functional Requirements Matching 17

3 Performance Evaluation **19**

3.1 DSS Architecture: brief recap 20

3.2 Performance comparison 22

A Changes from previous versions **26**

Section 1

Introduction

Work Package 8 is devoted to the integration of the single ePolicy components into a comprehensive framework. While the single components advance the state-of-the-art in their respective research field, a key aspect of ePolicy is to merge these components into a unified framework, so as to provide the policy maker a single and unique perspective on the policy process model.

WP8 addressed a number of key challenges. On one side, there was the need of allowing each project's partner to pursue the best research direction, without posing unnecessary technical constraints. On the other side, there was the need to "merge" the components in a comprehensive framework able to guide the policy maker towards informed decisions. Moreover, the application of software engineering practices has been made difficult by the research flavor of ePolicy, and (non-)functional requirements have been often revised on policy makers' inputs: once made aware of new possibilities opened up by the projects' partial results, policy makers suggested new possibilities and goals for the project, in an iterative process. The outcome of WP8 research activities is the integrated decision support system documented in D8.2[1].

This deliverable has a two-fold aim: on one hand, it shows how functional and non-functional requirements of the integrated architecture have been addressed; on the other hand, it provides some performance evaluation about the integrated framework.

Regarding the requirements, deliverable D2.3[2] proposed a number of objectives to be achieved within the project. Among them, a number of those objectives have been specified in terms of functional and non functional requirements, enlisted in the Deliverable D8.2[1]. Section 2 enlists the requirements, together with some discussion explaining how they have been addressed.

About the performance evaluation of the integrated framework, Section 3 focuses on the impact that the technical choices have on the single components. While the performances

[1] D8.2 "Integrated Policy Decision Support System Implementation"
[2] D2.3 "Means of Project Evaluation"

of each component have been addressed in the related work packages[3], in this deliverable we are more interested to document how the technological and architectural choices have an impact over the single components. From the performance viewpoint, the most important choice is related to the decision of having each component running in a server, plus (a) a server acting as a central orchestrator and offering a security layer and authentication services; and (b) a dedicated server running the web-based graphical front-end and the visualization tools. Such architecture implies that an important amount of data is transferred via network connections. Section 3 provides some evaluation of how the distribution of data across a network increases the time delays: the comparison has been done by confronting the average time of using each component as a stand-alone service (locally invoked directly on a server), rather than be used across the network. Also, the delay introduced by the use of Visual Interface documented in Deliverable D7.4[4] has been assessed.

In this document we do not provide any evaluation about the scalability of the integrated framework. The current technological solution envisages a server for each component, plus an orchestrating server. The envisaged business model foresees one installation of the whole architecture for each public body (being it a European region, a city council or a nation-wide body). For each installation, a limited number of users are expected (no more than ten, let's say). The server-side communication has been build using the Spring Framework[5], and services have been exposed following a REST approach. This architecture has been proved to be overly sufficient and robust for the identified business model, and it can easily support the scaling up to hundreds of users. Given such premises, we did not consider meaningful to investigate the scaling up issues.

[3]See for example Deliverable D3.3, Section 2.2, for some figures about performances and scalability of the Global Optimizer component.

[4]D7.4 "Evaluation of Visual Analytics Prototypes (Version 2)"

[5]`http://spring.io/`

Section 2

Evaluation of Functional Requirements

In this chapter we enlist the functional requirements that emerged both from the discussion based on D2.1, and from the GUIs mockup presented in D8.2.

2.1 Requirements derived from the Roles management: the "Login page" and the security issues

The principal aim of the ePolicy Decision Support System is to aid policy makers to develop plans and implementation policies for specific aspects. This prototype in particular will be fitted for the energy related field, and for alternative energy sector in specific. Beside the policy maker, we envisage at least other two possible type of users that are involved within the policy making process: the administrator, responsible for the correct functioning of the whole prototype, and a domain expert.

Due to the nature of the ePolicy prototype, the main security issue that has been managed is the user authentication. In particular, since the data managed in the system might be considered as sensible data (thus not public available), the access to the services will be controlled and restricted to authenticated users. Generally speaking, access control is aimed at allowing authorised users to access the system resources they need, while preventing un-authorised users to do the same. In the case of the ePolicy prototype the suitable access control model is the Role Base Access Control (RBAC).

RBAC is a NIST standard [6] and specifies security policies in terms of organisational abstractions (users, roles, objects, operations, permissions, and sessions) and their relationships [1]. In particular, users are assigned to roles, and roles to permissions. A *role* is understood as a job function within the context of an *organisation* with some associated semantics regarding the authority and responsibilities conferred to the user which plays the role at a given time. A *permission* is an approval to perform an *operation* on some protected *objects*: the exact semantics of "operation" and "object" depends on the specific case. A session is a mapping between a given user and the subset of its currently active

roles: so, each session is associated with a single user, while a user can be associated to one or more sessions. Organisation rules are defined in terms of relationships between the above elements—namely, between roles and permissions, and between roles and users; inter-role relationships are also introduced to specify *separation of duties*. More precisely, *static separation of duty* (SSD) is obtained by enforcing constraints on the assignment of users to roles, while *dynamic separation of duty* (DSD) is achieved by placing constraints on the roles that can be activated within or across the given users' session(s).

This discussion leads to the definition of a number of requirements tied to the users authentication and roles assignment. In particular:

FR1. Support for an authentication mechanism based on the concept of *username* and *password*.

FR2. Support for the concept of *user role*, where to different roles correspond different set of rights for executing actions.

FR3. Support for the role of *policy maker*.

FR4. Support for the role of *domain expert*.

FR5. Support for the role of *administrator*.

FR6. Support for a *dummy user*.

FR1. Requirement: Today, access control is typically designed by clearly separating the definition of a suitable *access policy* – i.e., the set norms for granting / refusing access to resources – from the hardware & software *mechanisms* used to implement and enforce it. This requirement specifies the kind of mechanism adopted for the ePolicy prototype access control. In order to satisfy this requirement, we have developed two different login pages. The first one, showed in Figure 2.1, represents the main login page, while the second, showed in Figure 2.2, represents the login page for the administration services. Both pages collect the user credentials and send them to the ePolicy server, where a Spring Security [5] mechanism checks the user credentials and associates the right role to the authorized user .

FR2. - FR6. Requirements: These requirements specify an high level access control policy. We implement these policies through a Spring Security mechanism: for each service provided by the ePolicy Web Service we have specified an RBAC *permission* that associate the authorisation for doing a certain *operation* on some protected *resource*. Then, these permissions have been associated to the different roles according to the specific necessities, and each role is associated to an user. The *dummy user* (Requirement **FR6.**) can navigate and query the ePolicy DSS through the specification of the data in suitable fields such as data for the social simulator or data for the global optimisation, but this user cannot access to the reserved data and policies introduced/ created by the authorised roles/ ePolicy DSS.

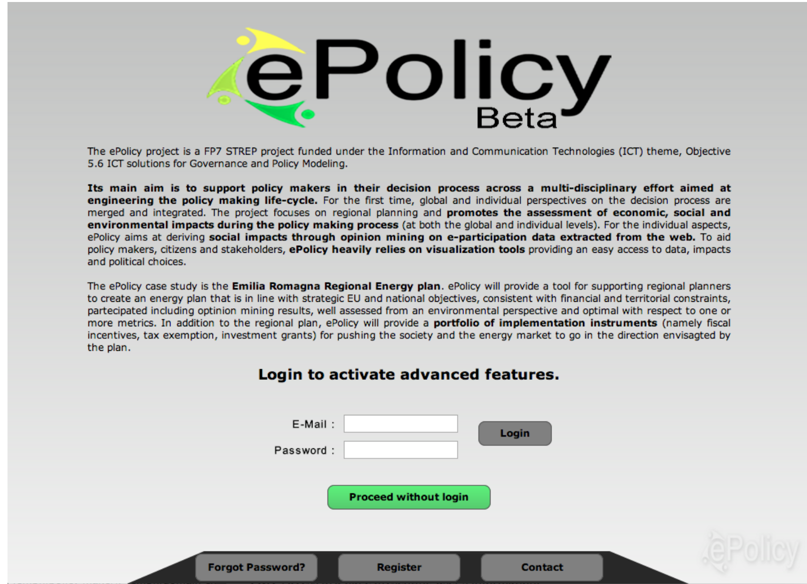

Figure 2.1: Login page of the ePolicy Decision Support System

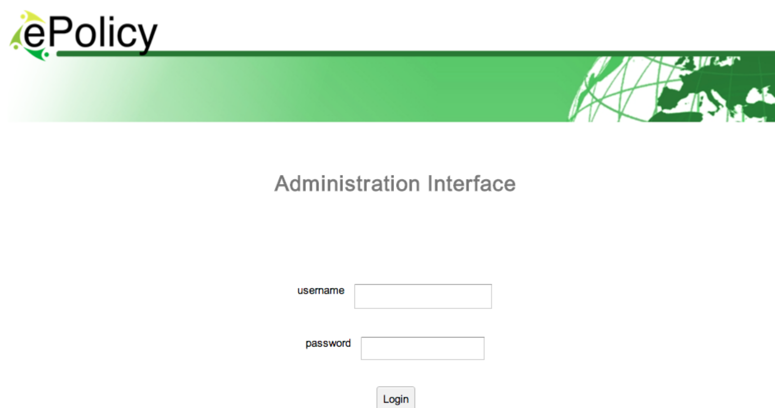

Figure 2.2: Login page of the ePolicy Decision Support System Administration

The following code represents and extract of the ePolicy Web Services. In particular, the code manages the *getAccounts* request. The red line represents a Spring Security check for verifying the authorisation. When the *getAccounts* service is requested, the Spring Security mechanism takes several actions:

- it recovers the user from the session,
- it recovers the role and all the permissions associated to the user that are stores in a chain of so called *GranthedAuthority*
- if the user has the requested permission the *getAccounts* service is invoked and a result is returned, otherwise the ePolicy Web Service notify the security violation to the browser.

```
1        @PreAuthorize("hasRole('/admin/getAccounts')")
2            @RequestMapping(value = "/admin/getAccounts",
3              method={RequestMethod.GET,RequestMethod.POST})
4             public @ResponseBody List<Account> getAccounts() {
5                log_db("/admin/getAccounts");
6                 logger.info("getAccounts invoked!");
7                 return userService.getAccounts();
8                 }
```

2.2 Requirements derived from the "policy maker" view

The following requirements have been identified:

FR7. Access to a page presenting user's information.
FR8. Support to the notion of a user profile.
FR9. Save generated plans.
FR10. Load and inspect previously saved plans.
FR11. Delete previously saved plans.
FR12. Inspect saved plans.
FR13. Save, load, inspect and delete information about the social simulation and best incentive/rewarding mechanism about a specific plan.

FR7. - FR13. Requirements: These requirements allows policy maker to view his/her profile and to manage the Energy plans. We have addressed these requirements providing several different APIs for storing, retrieving and changing the data from a suitable DataBase. In order to do this, we have exploited the Spring framework functionalities that allow both to easily manage the Database access, and, as said previously, to manage the system security.

2.3 Requirements derived from the “Domain Expert” view

The following requirements have been identified, in addition to the requirements identified for the policy maker role:

FR14. For each service, the domain expert must be able to access, modify, save and delete configurations/models.

FR15. Data saved by domain expert should be related to the configuration/model used, and comparison between data computed with different “current models” should not be supported.

In the following paragraph we discuss the different requirements.

FR14. Requirement: The domain expert should be able to access to all services accessed by the policy maker.

We accomplish this requirement assigning to domain expert the same access permissions already assigned to the policy maker.

FR15. Requirement: The domain expert will be granted the possibility of configuring the various components. This will do through the specification of a domain model where several data such as the budget, the amount of electric power, and a lot of general constraints will be specified. We accomplish this requirement allowing to the domain expert to upload a file where all the data and constraints will be specified. Figure 2.3 shows an example of the GUI adopted for the file upload. In particular, in the top of the page it is possible to choose the file and upload it, while in the bottom a list of all the uploaded files is showed, and the domain expert can also download the different files previously uploaded.

When a new file is uploaded, it is analysed by the ePolicy DSS and the contained data are used for the initialisation of the system variables. If the file contain errors the browser will be notified.

From a technological point of view, the accepted files are excel file formatted according to a specific style provided by the Emilia-Romagna Region (in attached at this Deliverable we provide an example of the file format). The format of the excel file has been decided by the partners involved in the domain modelling activities. Such partners have specifically requested that the excel headers’ columns are in Italian.

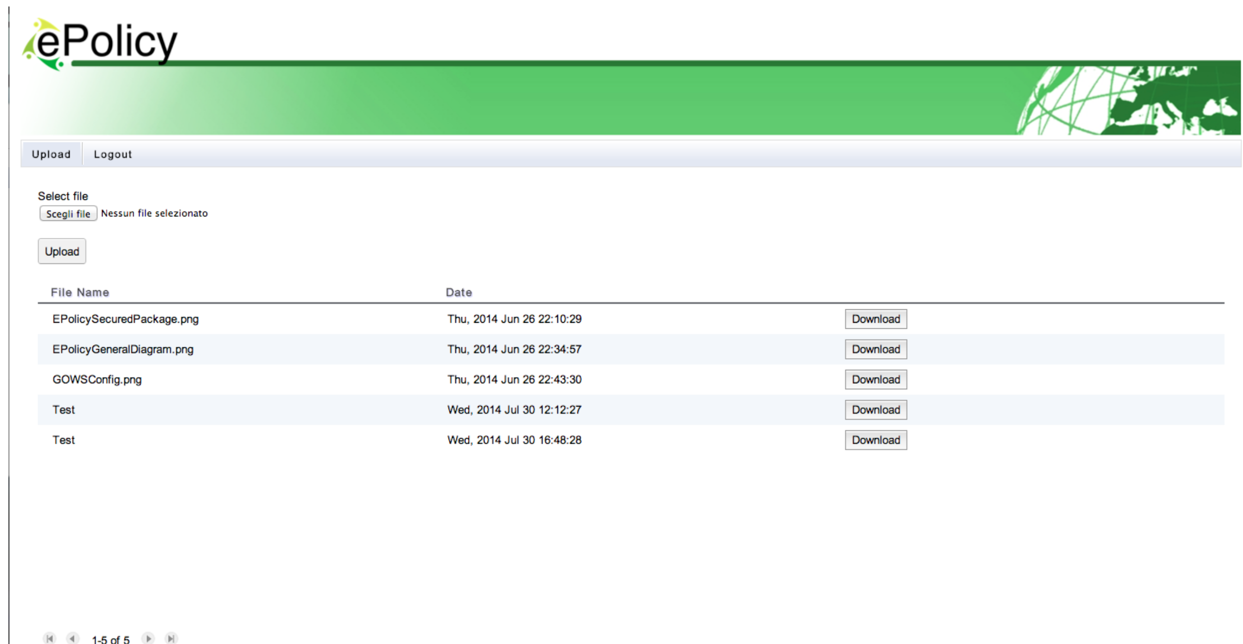

Figure 2.3: The file upload service

2.4 Requirements derived from the "Administrator" view

The administrator is person in charge of the ePolicy DSS. In the case of ePolicy DSS, all the permissions are associated to the administrator role. This because the administrator must manage the RBAC policy and must be able to check the right running of the different services composing the DSS. Here, we discuss only the requirements deriving from the RBAC policy management, since the other requirements are already discussed in the previous Subsection.

For the management of the RBAC policy, the following requirements have been identified:

FR16. Support for adding / deleting / modifying users.

FR17. Support for adding / deleting / modifying roles.

FR18. Support for adding / deleting / modifying permissions.

FR19. Support for adding / deleting / modifying services.

FR20. Identification of a precise set of *permissions*, i.e. a list of actions that can/can not be executed depending on the role assumed by an authenticated user.

FR21. Logging of the actions executed within the system.

FR22. Access to the logs.

All these requirements lead us to design a proper data base in order to store the RBAC data. This DB refers the RBAC model structure. In Figure 2.4 we show a simply version of the Entity-Relationship diagram used for designing the real DB. In particular, we report only the entities, the relationships among them and their cardinalities. From a technological point of view, we chose the Mysql [3] as Database Management System (DBMS). The integration between the java-based components and the persistence layer is based on Object-Relational Mapping (ORM) technologies, and in particular by exploiting the Java

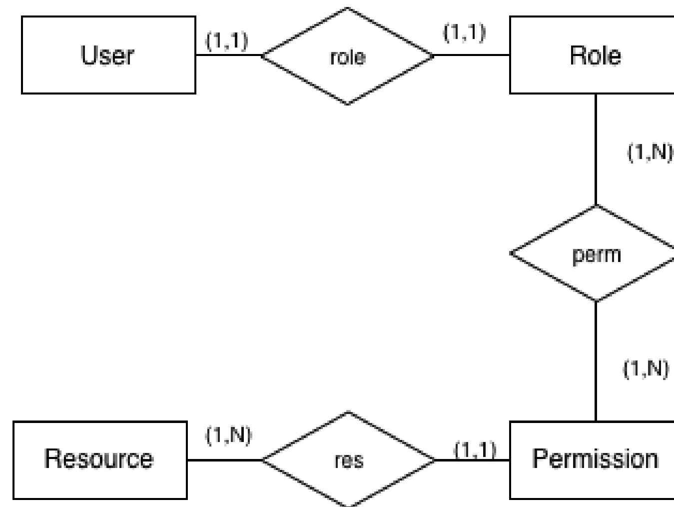

Figure 2.4: The User DB ER Diagram

Persistence Annotation (JPA [2]) standard. Hibernate[7] is the preferred implementation of the JPA standard in the current implementation of the ePolicy Framework. Finally we adopt the Google Web ToolKit (GWT) as a framework for developing the Administration Interface showed in Figures 2.5 - 2.9.

In the following paragraph we discuss the different requirements.

FR16. Requirement: This requirement implies that the administrator should be able to add users and modify users information, and to add/remove users to roles. The administrator should be able also to delete a specific user. We accomplish this requirement through several operations over the DB. In particular, before doing the insertion, modification and deletion of a user, the system verify the actual presence of such a user. In case of insertion, if a user already exists – i.e., the same email and password are already presented in the DB – the insertion is aborted and the browser is notified.

We also developed a specific GUI for the user management. Figure 2.5 shows an example. In particular, this page shows the list of registered users with their relative data (name, surname, email and password) and the role assigned to the user. For security reason the password is not sent from web service to the browser, so the field is empty. Through two different buttons it is possible to store (the *save* button) the user modification or delete (the *Delete* button) the user. In the bottom of the page the *insert* button allows the administrator to insert a new user with respective data and role.

ePolicy

User | Role | Permission | Permission Type | Resource | Log | Upload | Logout

First Name	Surname	Email	Password	Roles		
user	surname	user		☑ ADMIN ☐ DOMAIN_EXPERT ☐ POLICY_MAKER	Save	Delete
user2	surname	user2		☐ ADMIN ☑ DOMAIN_EXPERT ☐ POLICY_MAKER	Save	Delete

1-2 of 2 insert

Figure 2.5: The users management service

ePolicy

User | Role | Permission | Permission Type | Resource | Log | Upload | Logout

Name	Permissions		
ADMIN	☑ /om/getDataDailyAggregation ☑ /om/getDataWeeklyAggregation ☑ /om/getDataMonthlyAggregation ☑ /om/getDataQuarterAggregation ☑ /om/getDataHalfYearAggregation ☑ /ssim/checkExistingInput ☑ /ssim/findInputUniqueIdByScenarioId ☑ /ssim/getStatusOfScenarioId ☑ /ssim/getStatusOfInputUniqueId ☑ /ssim/requestRun ☑ /ssim/viewScenario ☑ /ssim/getSimBrowseData ☑ /ssim/getSimDataForOneInputID ☑ /go/computePlan ☑ /go/computePareto ☑ /go/savePlan ☑ /go/findByPlanName ☑ /go/findPlanByAccount ☑ /go/findPlanById ☑ /go/deletePlanById ☑ /de/uploadFile ☑ /de/downloadFile ☑ /de/getFiles ☑ /admin/getLogs ☑ /admin/checkRole	Save	Delete

1-3 of 3 insert

Figure 2.6: The roles management service

Figure 2.7: The permissions management service

FR17. Requirement: This requirement implies that the administrator should be able to add / delete roles and modify roles' permissions. The administrator should be able also to shows the list of roles. We accomplish this requirement through several operations over the DB. In particular, before doing the insertion and modification of a role, the system verify the actual presence of such a role. In case of insertion, if a role already exists – i.e., the role name is already presented in the DB – the insertion is aborted and the browser is notified. The delete operation is critical, since if a role is associated to an user the deletion is aborted and a notification is sent to the browser: the administrator should remove all the associations user-role before delete a role. This choice might affect the usability for expert administrators, but we deem it as necessary for preventing the accidental role deletion. In addition, only the human administrator is able to choose a new right role for the user.

We also developed a specific GUI for the role management. Figure 2.6 shows an example. In particular, this page shows the list of roles with their associated list of permissions Through two different buttons it is possible to store (the *save* button) the user modification or delete (the *Delete* button) the role. In the bottom of the page the *insert* button allows the administrator to insert a new role with respective permissions. Currently, only three roles have been identified for the ePolicy DSS—policy maker, domain expert, and administrator. However, we provided the insertion operation. In this way, if a new role will be necessary, the administrator will be able to add it without stopping the system.

FR18. Requirement: This requirement implies that the administrator should be able to add / delete permissions and modify their information.

ePolicy

User | Role | Permission | Permission Type | Resource | Log | Upload | Logout

Service Name	Service Address		
OM_getCategories	/om/getCategories	Save	Delete
OM_getDataNoAggregation	/om/getDataNoAggregation	Save	Delete
OM_getDataDailyAggregation	/om/getDataDailyAggregation	Save	Delete
OM_getDataWeeklyAggregation	/om/getDataWeeklyAggregation	Save	Delete
OM_getDataMonthlyAggregation	/om/getDataMonthlyAggregation	Save	Delete
OM_getDataQuarterAggregation	/om/getDataQuarterAggregation	Save	Delete
OM_getDataHalfYearAggregation	/om/getDataHalfYearAggregation	Save	Delete
SSim_checkExistingInput	/ssim/checkExistingInput	Save	Delete
SSim_findInputUniqueIdByScenarioId	/ssim/findInputUniqueIdByScenarioId	Save	Delete
SSim_getStatusOfScenarioId	/ssim/getStatusOfScenarioId	Save	Delete
SSim_getStatusOfInputUniqueId	/ssim/getStatusOfInputUniqueId	Save	Delete
SSim_requestRun	/ssim/requestRun	Save	Delete
SSim_viewScenario	/ssim/viewScenario	Save	Delete

1-49 of 49 insert

Figure 2.8: The resources management service

We accomplish this requirement through several operations over the DB. In particular, before doing the insertion, modification and deletion of a permission, the system verify the actual presence of such a permission. In case of insertion, if a permission already exists – i.e., the permission name is already presented in the DB – the insertion is aborted and the browser is notified. In addition, we have developed another functionality: the administrator should be able also to specify a specific permission type. This functionality is not a requirement, but in a *design for change* perspective, the administrator should be able to specify the kind of operation tied to the permission. Currently, the only permission type presents in the DB is *Invoke*, that specify if a service can be invoked. However, in a future other permission types – such as read or write – could be useful.

We developed a specific GUI for the permission management, and another for the permission type management. Figure 2.7 shows an example of the first one. In particular, this page shows the list of permission with their relative data (name, permission type) and the associated resource. Through two different buttons it is possible to store (the *save* button) the user modification or delete (the *Delete* button) the permission. In the bottom of the page the *insert* button allows the administrator to insert a new permission with respective data.

FR19. Requirement: This requirement implies that the administrator should be able to add resource and modify resource information. The administrator should be able also to delete a specific resource.

We accomplish this requirement through several operations over the DB. In particular,

before doing the insertion, modification and deletion of a resource, the system verify the actual presence of such a resource. In case of insertion, if a resource already exists – i.e., the same resource name is already presented in the DB – the insertion is aborted and the browser is notified. In addition, the delete operation is critical, since if a permission is associated to the resource, the deletion is aborted and a notification is sent to the browser: the administrator should remove all the associations permission-resource before delete a resource. This choice is necessary for preventing the accidental resource deletion. We also developed a specific GUI for the resource management. Figure 2.8 shows an example. In particular, this page shows the list of resources with their relative data (name, web address). Through two different buttons it is possible to store (the *save* button) the resource modification or delete (the *Delete* button) the resource.

FR20. Requirement: This requirement implies that the administrator should able to choose the more suitable set of permissions for each role. We accomplish this requirement showing to the administrator the complete list of all the permissions for each role (see Figure 2.6), then the administrator should choose the set of the permissions he/she wants to associate to the role. There is not a *pre-determined* set of permissions associated to a role when this is inserted. The only assumption we have done during the development is the assumption of *closed-world security environment*: all operations on protected resources are implicitly denied until authorisation policies grant specific operation. This provides security that errs on the side of caution. For example, if the administrator forgets to deploy an authorisation policy, someone will be denied access. While this might be problematic, from a security view it is a preferable approach w.r.t. inadvertently allowing access to resources that should be protected instead. Users who are denied required access will almost certainly ask for corrective action, while users inadvertently granted unauthorised access are unlikely to bring this to the attention of administrators—with potentially disastrous consequences [4].

FR21. Requirement: This requirement implies that the administrator should be able to investigate all the operation inside the ePolicy DSS. We accomplish this requirement providing two types of log. The following code shows an example of the logs. The first log is a command line log printed in the Web server log (line red in the code), while the other (blu line) is stored in the DB and it should be visualised in a browser by the administrator.

```
1  @PreAuthorize("hasRole('/admin/getAccounts')")
2  @RequestMapping(value = "/admin/getAccounts",
3  method={RequestMethod.GET,RequestMethod.POST})
4  public @ResponseBody List<Account> getAccounts() {
5          logger.info("getAccounts invoked!");
6          log_db("/admin/getAccounts");
```

ePolicy

User | Role | Permission | Permission Type | Resource | Log | Upload | Logout

Date	Username	Service Invoked
Tue, 2014 Jun 10 11:38:49	user	/admin/checkRole
Tue, 2014 Jun 10 11:39:03	user	/admin/getResources
Tue, 2014 Jun 10 11:39:37	user	/admin/insertResource
Tue, 2014 Jun 10 11:39:38	user	/admin/getResources
Tue, 2014 Jun 10 11:40:18	user	/admin/insertResource
Tue, 2014 Jun 10 11:40:19	user	/admin/getResources
Tue, 2014 Jun 10 11:40:39	user	/admin/insertResource
Tue, 2014 Jun 10 11:40:40	user	/admin/getResources
Tue, 2014 Jun 10 11:40:57	user	/admin/modifyResource
Tue, 2014 Jun 10 11:40:58	user	/admin/getResources
Tue, 2014 Jun 10 11:41:29	user	/admin/insertResource
Tue, 2014 Jun 10 11:41:29	user	/admin/getResources
Tue, 2014 Jun 10 11:43:16	user	/admin/insertResource
Tue, 2014 Jun 10 11:43:16	user	/admin/getResources
Tue, 2014 Jun 10 11:43:56	user	/admin/insertResource
Tue, 2014 Jun 10 11:43:57	user	/admin/getResources
Tue, 2014 Jun 10 11:44:28	user	/admin/insertResource
Tue, 2014 Jun 10 11:44:29	user	/admin/getResources

1-50 of 11,527

Figure 2.9: The ePolicy DSS logs page

```
7                 return userService.getAccounts();
8         }
```

FR22. Requirement: This requirement implies that the administrator should be able to list all the operation inside the ePolicy DSS. We developed a specific GUI for the logs. Figure 2.9 shows an example. In particular, this page shows the list of log with their relative data: the date of the operation, the user that has requested such operation, and the service invoked.

From a technological point of view, we chose the Mysql [3] as Database Management System (DBMS). The integration between the java-based components and the persistence layer will be based on Object-Relational Mapping (ORM) technologies, and in particular by exploiting the Java Persistence Annotation (JPA [2]) standard. Hibernate[7] will be the preferred implementation of the JPA standard. Finally we adopt the Google Web ToolKit (GWT) as a framework for developing the Administration Interface showed in Figures 2.5 - 2.9

2.5 Summary

The ePolicy prototype provides access to the unified framework of services developed within the project by the consortium. The main requirements are already discussed both in Deliverable D2.1 and D8.2. Here, we have discussed how those requirements have

been addressed in the prototype. In particular, starting from the general requirements presented in D2.1, we have identified a set of roles – policy maker, domain expert, and administrator – and assigned to them the main requirements. We have also developed a specific RBAC model in order to assign to each role the more suitable set of access permissions. Then, we have designed and developed the ePolicy Architecture in order to accomplish the requirements. The ePolicy Architecture will be widely discussed in the next chapter.

The following Table presents a match between the functional requirements and the prototype implementation.

Table 2.1: Requirements Matching

Requirement	**Current prototype**
FR1.	supported
FR2.	supported
FR3.	supported
FR4.	supported
FR5.	supported
FR6.	supported
FR7.	supported
FR8.	supported
FR9.	supported
FR10.	supported
FR11.	supported
FR12.	supported
FR13.	supported
FR14.	supported
FR15.	supported
FR16.	supported
FR17.	supported
FR18.	supported
FR19.	supported
FR20.	supported
FR21.	supported
FR22.	supported

Section 3

Performance Evaluation

The purpose of this Section is to provide an evaluation of the performances of the ePolicy integrated framework. While the performances of the single ePolicy components have been addressed in the specific Work Packages (and the related deliverables), this section focuses in particular on how the performances have been affected by the network-based communication infrastructure.

The framework has been organized as a Service Oriented Architecture, where each component runs as a stand-alone, independent service, while an additional service plays the role of orchestrator, providing the security layer and persistence facilities. Moreover, the web interface offering visualisation techniques has been implemented as a further stand-alone service. Each component, as well as the orchestrator, have been wrapped into a Web Service exposing facilities and following a REST approach, by exploiting the well known Spring Framework[1]; the web-based user interface instead has been built using the GWT[2] framework.

Each component, the orchestrator, and the user interface run in a separated server. The choice of having a server for each component is motivated by the different needs of each ePolicy component. Being based on advances to the state of the art in their respective research field, each component presents its own specific requirements in terms of memory, storage, computational power, operating system and installed libraries. Having separated servers allowed us to overcome possible problems deriving by conflicting requirements, and to adopt a simple yet very neat distributed architecture. However, the coordination and the integration of the components has been achieved through intense network-based communication.

Given the premises above, an important question is *if*, and *how much* the choice of a distributed architecture afflicts the performances of the integrated framework. In particular, delays introduced by the network communication might affect the user experience, thus hindering the usability of the framework. After a brief recap of the modules composing the ePolicy integrated framework (Section 3.1), in Section 3.2 we present and discuss

[1] `http://spring.io/`

[2] `http://www.gwtproject.org/`

the figures about the time required to invoke the services as stand-alone components or rather as part of a distributed system.

The current ePolicy demo is running in a distributed environment: component's servers and the orchestrator are running as virtual servers in a physical node at the University of Bologna, Bologna, Italy, while the user interface and the visualization tools are running in a virtual server in the Fraunhofer IGD research center in Darmstadt, Germany. Hence, the figures discussed in Section 3.2 represent an effective case of a distributed system across Europe.

3.1 DSS Architecture: brief recap

The ePolicy DSS prototype is a distributed system composed by several components developed by projects partners. Figure 3.1 presents a general overview of the prototype architecture. In particular, the components developed by partners have been wrapped by RESTful Web Services orchestrated by a central Web Service (*ePolicySecured* in the Figure 3.1) that ensures the system security.

In detail:

- *ePolicySecured*: it is the main web services that ensures the system security and dispatches the service requests to the specific web service. This server provides also the administrative functionalities allowing the administrator to manage the RBAC policies and the users. The incoming requests are dispatched to the specific web service only if the role associated to the request has the right access permissions.
- *ePolicyDomainSecured*: it represents the domain classes of the prototype. This component is shared among the others in order to provide to all components the same domain entities. This simplifies the interactions among them components.
- *ePolicyClient*: it represents a common interface used by the two GUIs. In particular the GUIs component invoke the ePolicySecured services through the services provided by ePolicyClient.
- *SSimWSRest*: it represents the web service wrapping the Social Simulator component. As it is showed in Figure 3.1 the SSimWSRest uses an external component in order to provide the service. This because the Social Simulator needs a long computation time for producing the simulation results, so the Social Simulator works on a separate virtual machine and it is invoked by SSimWSRest through the java RMI[3] protocol. In particular, the SSimWSRest uses the SocialSimulator component that is an RMI client. SocialSimulator, in turn, dispatches the request to SocialSimulator-Libs, the RMI server, that generates the simulation.
- *OpinionMiningWS*: it represents the web service wrapping the Opinion Mining component.
- *GOWSRest*: it represents the web service wrapping the GlobalOpt component.

[3]http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/

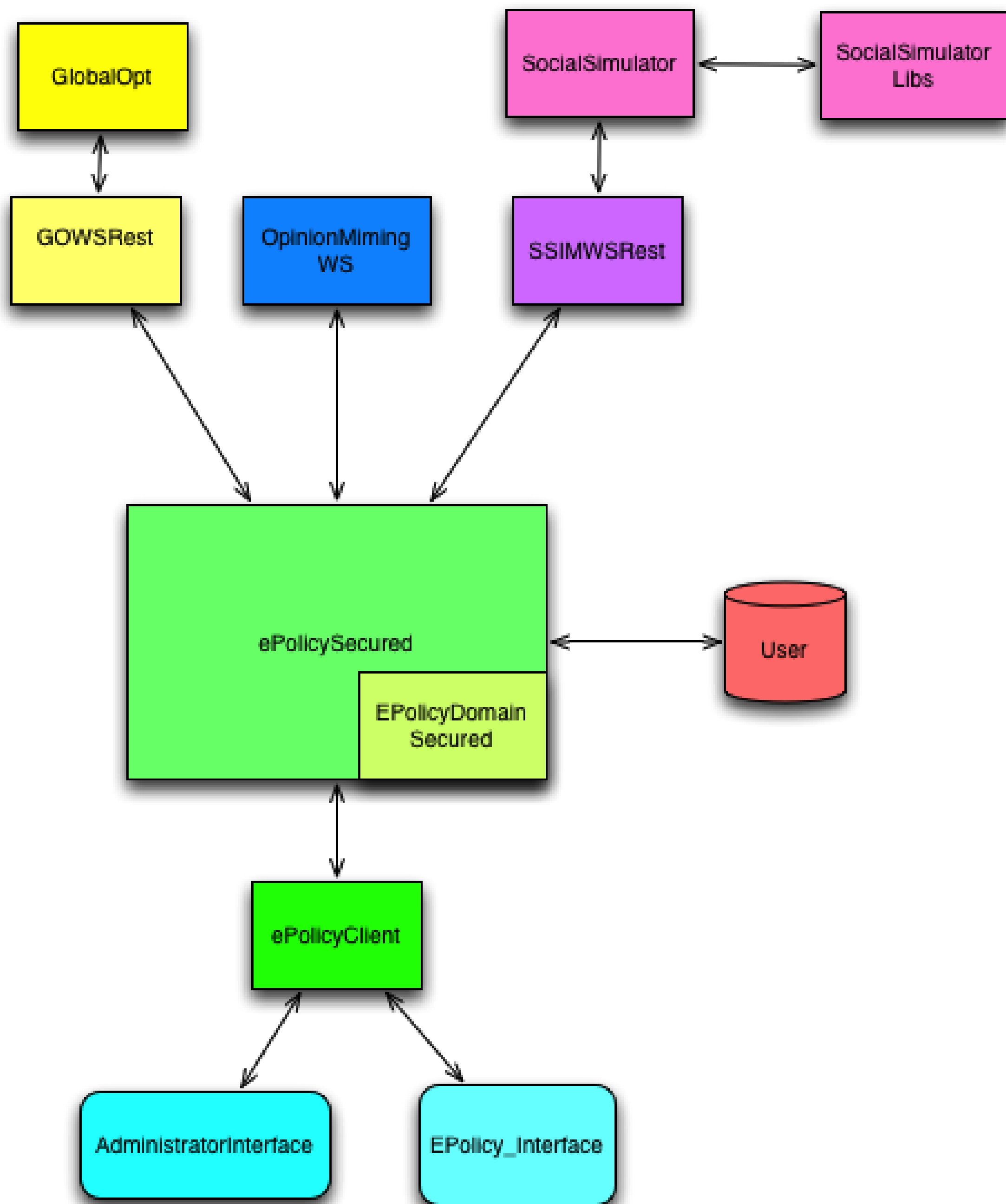

Figure 3.1: ePolicy DSS Architecture

- *AdministratorInterface*: it represents the GUI component for the administration services.
- *EPolicy_Interface*: it represents the main GUI component for the policy maker and the domain expert.

3.2 Performance comparison

Table 3.1 presents the results of our evaluation of the response times of the ePolicy DSS. The values presented in the Table are medium values. For each service, we have conducted several tests (ten tests) in the worst cases – i.e., the more expensive services respect to the response time – and here we provide the medium values.

In particular, in the columns we have:

- first column: the service name;
- second column: the response time of the stand-alone component;
- third column: the response time of the wrapped component, i.e. the component wrapped by the RESTful Web Service, in order to understand how the the network communication impacts the response time;
- fourth column: the response time of the component with the security layer. In this case we have invoked the service through the ePolicySecured Web Service (see Figure 3.1) in order to establish how the security layer impacts the response time;
- fifth column: the response time of the component invoked by the GUIs. In this case we have invoked the service through the ePolicySecured Web Service and the GUIs (show Figure 3.1) in order to assess how the GUIs impact the response time.

Table 3.1: Response Times Evaluation

Component	**Component Invocation**	**Component Invocation by net**	**Component Invocation by net + security**	**Component Invocation by GUI**
Social Simulator	111.16 min	no significant difference	no significant difference	1122 ms
Global Optimizer	1813 ms	1926 ms	2106 ms	2186 ms
Opinion Mining	742 ms	766 ms	948 ms	2754 ms
Admin	249 ms	267.5 ms	270 ms	460 ms

The first row of Table 3.1 shows the response times of the Social Simulator component. The presented data show important discrepancies. In particular, the medium value of the response time of the stand-alone component is 111.16 minutes. This time is composed by the simulation time and by the calculation and aggregation of a lot of different variables

values. We have not investigate beyond the response times for columns three and four, since the delays introduced by both the network communication and the security layer are not appreciable – the delays are around milliseconds — comparing to the response time of the stand-alone component. The value presented in column five, indeed, seems very *low*. This because, the GUI does not invoke directly the Social Simulator component. In fact, due the long time requested for running the simulations, we have decided to run the simulations off-line and to store the calculated data into a data base. So, the results provided by the GUI are already calculated and the response time is acceptable, since accessing it means to query the database with the precomputed solutions. Currently, the GUI provides the results only for a certain subset of input values, but we are working for populating the data base with data covering all the space solution.

The second row of Table 3.1 shows the response times of the Global Optimizer component. The response times have been calculated for the `computePlan` service, that provides an energy plan according to the inputs values. The values in all the columns are comparable among them, and the delays introduced by the network communications, the security layer and the GUI are contained – around 373 ms – considering the widely distributed nature of the architecture.

The third row of Table 3.1 shows the response times of the Opinion Mining component. The response times for the three first columns have been calculated for the `getDataDailyAggregation` service that is the most expensive from the response time point of view. The tests have been conducted onto a data base already populated—the data are stored in a off-line way in order to speed the response times. As showed in the Table, the response times in the firsts three columns are lower respect to the response time in the last column. This because, in the case of the invocation through GUI, the GUI has to do different invocations to the ePolicySecured in order to collect all the aggregated data and shows the relative diagrams for a time spectrum covering twelve years. The values in the first three columns are comparable. The delays introduced by both the network communications and the security layer are around 206 ms that seems acceptable.

Last row in the Table 3.1) presents the tests conducted for the Administration services regarding the invocation of the `getPermissions` service that provides the list of all access permissions stored in the DB. We have chosen this service since it has been the more expensive service from viewpoint of the response time. This because in the ePolicy DSS there is great number of permissions (one for each service, i.e., about one hundred permissions) due to the assumption of closed-world security environment (see 2.4). As showed in the Table, the response times are on average of the order of 250-300 ms in the firsts three columns, while the response time in the last column is a bite more. This because, in the case of the invocation through GUI, the GUI has to do three different invocations to the ePolicySecured in order to obtain the list of permissions, and the lists of permissionTypes, and resources necessary for allowing the administrator to insert new permissions or modify the existing permissions.

As a final remark, we can highlight that all the tests conducted have shown that the net-

work communications and the security layer have a very small impact over the response times of the services. So, the widely distributed nature of the chosen architecture has little effect onto the ePolicy DSS performances.

Bibliography

[1] D.F. Ferraiolo, R. Kuhn, and R. Sandhu. RBAC standard rationale: comments on a critique of the ansi standard on role based access control. *IEEE Security & Privacy*, 5(6):51–53, nov–dec 2007.

[2] Oracle. Jpa home page. https://www.jcp.org/en/jsr/detail?id=317.

[3] Oracle. Mysql home page. http://www.mysql.com/.

[4] Oracle. Security policy overview. http://docs.oracle.com/cd/E12890_01/ales/docs32/policymar

[5] Srping Security Project. Spring security home page. http://projects.spring.io/spring-security/, 2012.

[6] RBAC. American national standard 359-2004 (Role Base Access Control - home page). `http://csrc.nist.gov/rbac/`, 2004.

[7] RedHat. Hibernate home page. http://hibernate.org/.

Appendix A

Changes from previous versions

0.0	Creation of the Document
0.1	Finalisation of Section 2
0.2	Section 3.1 added
0.3	Introduction added
1.0	Version submmitted to EU commission