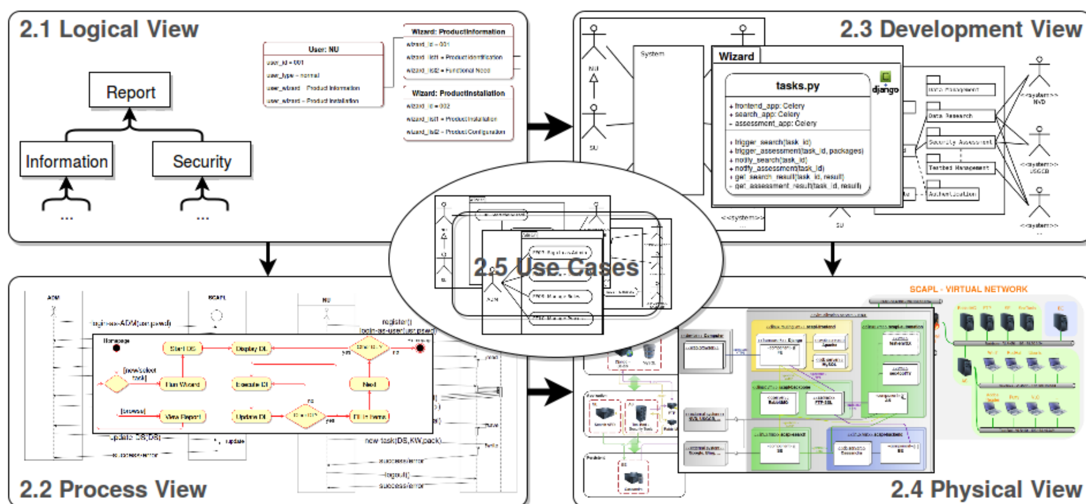
G SDD EXTRACT

G.1 **System Overview** 1
G.1.1 Logical View 1
G.1.2 Process View 3
G.1.3 Development View 4
G.1.4 Physical View 5
G.1.5 Use Cases 8
G.2 **System Architecture** 10
G.2.1 Architectural Design 10
G.2.2 Decomposition Description 12
G.2.3 Design Rationale 14
G.3 **Data Design** 14
G.3.1 Data Description 14
G.3.2 Data Dictionary 15
G.4 **Component Design** 16
G.4.1 Generic Structure 16
G.4.2 Front-End 17
G.4.3 Search Engine, Automation System 17

G.1 System Overview

This section provides an overview about system's context and design, based on the 4+1 architectural views model, addressing each view in the order depicted in the following image.



G.1.1 Logical View

First, regarding the objectives, the final outcome, that is, a report about an analyzed software product, must be defined on a hierarchical basis. Figure G.1 depicts how a report is to be dissected according to the information to be found within.

A *Report* object can be divided in two parts :

1. *General* : It relates to purely informational results, that is, information that can be found by simply using a public search engine such as Google, Bing or Yahoo!. Organizationally, the task of

searching for the general information about a software product can be delegated to non-security related personnel such as managers, architects, technicians, ...

2. *Security* : It refers to more specialized results, obtained from using specific tools, applying particular controls and so forth. As a consequence, security assessment tasks have to be delegated to specialized personnel such as auditors, security assessors, accreditators, ...

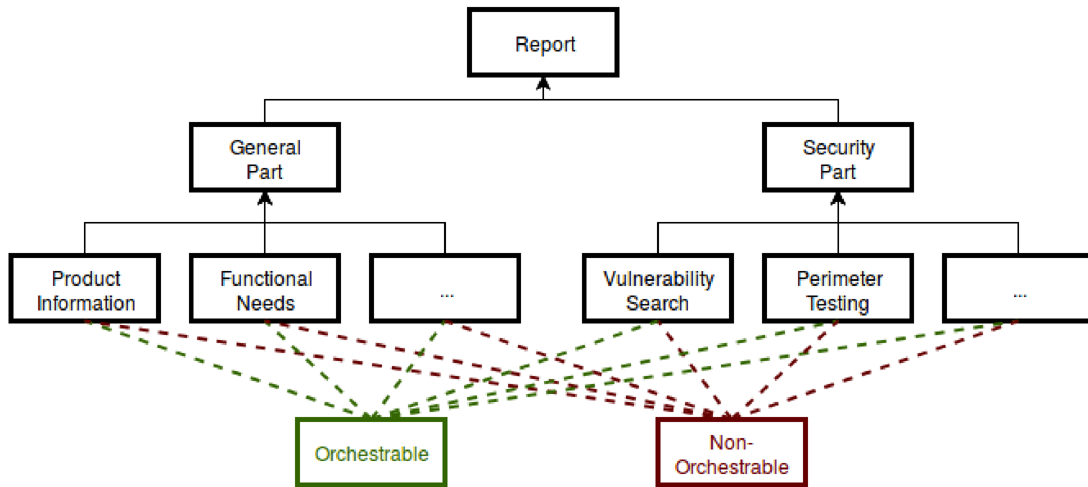


Figure G.1: Object diagram for a report

A very important thing to note is that one can expect to be able to automate some tasks but it's already sure that it will not be possible for all of these. That's why the design of SCAPL requires a sufficient granularity in its objects to be able to distinguish manual and automated tasks.

So what ? How can a report be generated with SCAPL based on the given object definition ?

In its objectives, SCAPL's ambition is to provide the end-user a user-friendly interface to work out the assessment of a software product. A possible way to solve this problem is to provide the user a wizard that can help him to get quickly to the point. So, coming back to the object definitions and now that Report is defined, a relationship must be defined between a end-user and the tasks to be triggered in order to generate a report. The following figure depicts this relationship in a hierarchical way.

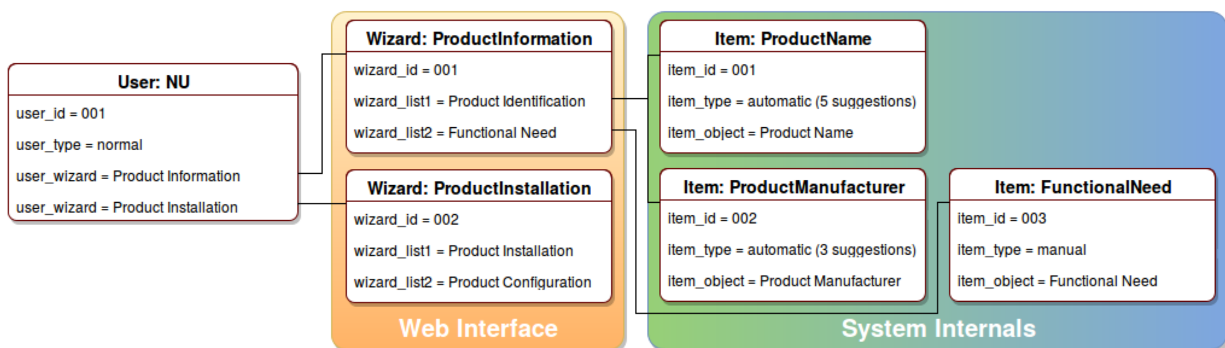


Figure G.2: Object diagram with the relationship between a user and the available tasks

In Figure G.2, *User* does obviously not include every user-related attributes but the main ones are the identifier, the type and the wizards he/she is allowed call (as NU and SU must not be able to trigger the same ones). *Wizard* objects describe lists of information items that must be handled. Task objects simply hold the automated functionalities on the information items at an atomic level. In this example, one can see that *User 001* of type *normal* is able to access wizards *001* (*ProductInformation*) and *002* (*ProductInstallation*). These wizards in turn refer to tasks *001*, *002* and *003* to get the related information.

G.1.2 Process View

In order to explain what is happening when a user browses his/her homepage and wants to run a wizard, the following activity diagram depicts the high-level activities performed to produce a record from which a report can be generated.

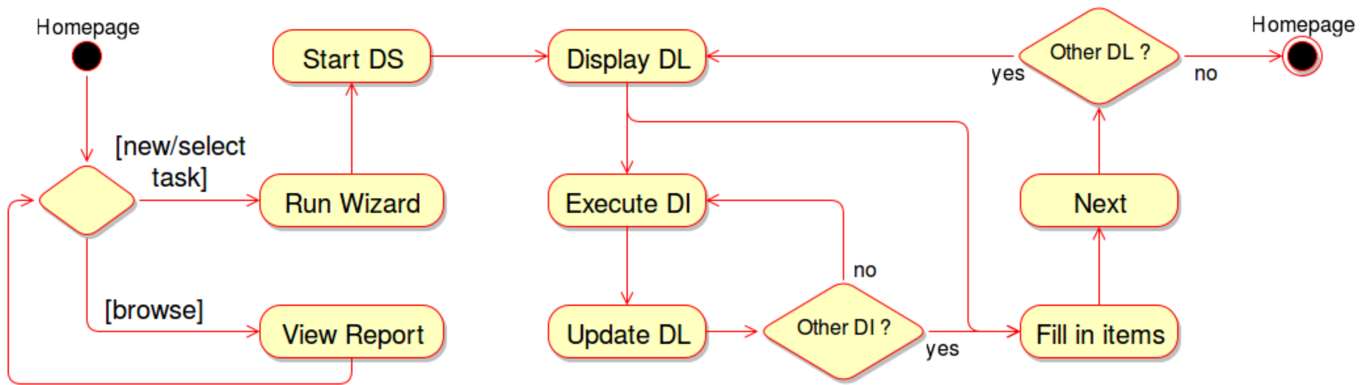


Figure G.3: Activity diagram for browsing reports and starting a wizard

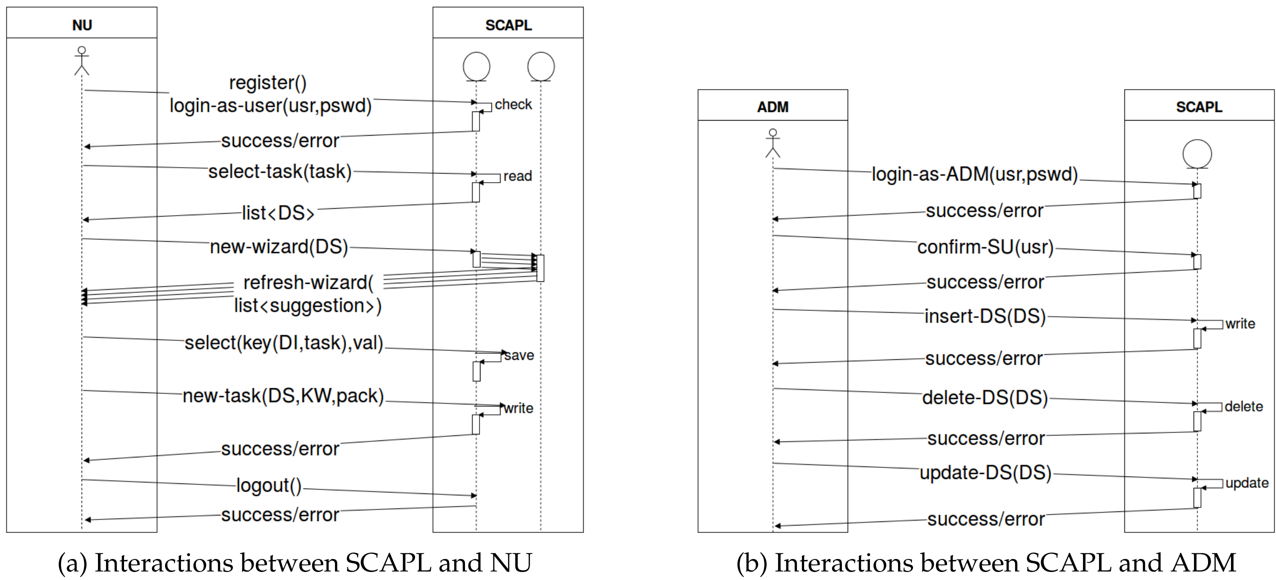
As the diagram in Figure G.3 shows, the starting point of this example is the user's homepage. From this point, the user can choose to browse existing reports or to select an existing task or to create a new one. Note that a task relates to all activities to be performed around a single software product to assess. Once the wizard is started :

- *Start DS*: is the activity for displaying the wizard itself as a sequence of steps
- *Display DL*: shows a form with all the items to process at each step of the sequence
- *Execute DI*: as the user can now also perform the activity Fill in items, the system starts processing data items in order to propose suggestions to the user
- *Update DL*: each data items is started asynchronously and triggers the update of the data list once done until there remains no more data item
- *Next*: as all the data items of the list have been processed and the right information has been selected, the user can go to the next step until the last is done, then redirecting to the homepage (at this point, the system is responsible to generate the report based on the data stored after following the wizard)

Another representation of the interactions between end-users and the system is sketched in Figure G.4. Note that these give an idea of how the system should handle various events, that will not necessarily be the exact sequence of implemented operations.

In Figure G.4a, a NU registers and selects an existing task to work on a specific software product to assess. He/she starts a new wizard based on a DS available for his/her role and then receives suggestions based on what was already processed by another user (if relevant, as the task already existed when starting the wizard) and on other DI's not processed yet. He/she selects a proposed data on a given DI whose value is stored in the back-end based on a combination of the current task identifier and the DI identifier. He/she then leaves the current wizard to create a new task (for another software product to assess) and finally log out (without starting a new wizard for this new task).

In Figure G.4b, an ADM logs in (on a separate interface) and first confirms a SU registration before performing operations on DS's. He/she then consecutively inserts, removes and updates a DS.



(a) Interactions between SCAPL and NU

(b) Interactions between SCAPL and ADM

Figure G.4: Sequence diagrams of multiple possible interactions

G.1.3 Development View

First, system's context is depicted in the context diagram in Figure G.5 in order to identify the actors.

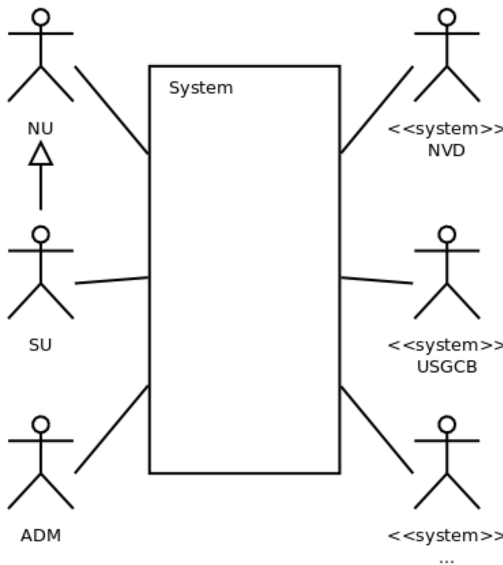


Figure G.5: Context diagram of SCAPL

The diagram on the left depicts the system in its context with its primary and secondary actors.

The **primary actors** consist of **NU** and **SU** inheriting functionalities of **NU** and then owning specific security ones. **ADM** is a separate actor independent from **NU** and **SU**.

The **secondary actors** are simply the external systems contacted by the system for mining relevant information.

Afterwards, system's packages are sketched in the package diagram in Figure G.6.

Admin: is aimed to provide to an administrator the functionalities to manage the data scheme and users and is completely independent from other from the other packages for the purpose of segregating system users' roles.

Wizard: contains the functionalities for running a wizard based on user's role and relies on *Authentication* package for user log in.

Profile: contains the functionalities for the user to manage its profile and to view the exposed data such as existing reports.

Authentication: is a separate package owning sign-in and sign-up functionalities.

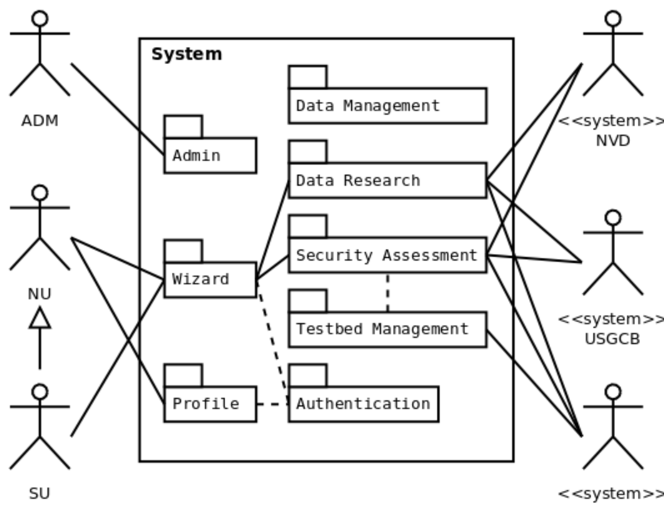


Figure G.6: Package diagram of SCAPL

Data Management: contains the report generation feature and eventually a tool to manage BE's database. This package is independent from any other as the report generation feature is an automated task and database management should be done by a system administrator, independently from any end-user.

Data Research: contains the search API's to be managed by a specialized search component.

Security Assessment: contains the security perimeter testing features to be managed by a specialized automation component.

Testbed Management: contains the virtualization features to be managed by a specialized automation component.

In this package diagram, one can see that :

- ADM only accesses features from the *Admin* package, meaning that a physical person playing the role of administrator should have a separate profile as NU or SU for using other features. This is an application of the security principle of role segregation.
- Except a particular functionality of the *Wizard* package, SU inherits all functionalities of NU. However, NU and SU do not have the same rights to trigger system features as the role determines the available ones (NU can trigger search tasks whereas SU can also trigger assessment tasks).
- Both links from *Authentication* to *Wizard* and *Profile* packages are sketched as dashed because they inherently rely on *Authentication*.
- The link from *Security Assessment* to *Testbed Management* is sketched as dashed because it inherently relies on *Testbed Management* for making test environments in order to be able to perform security assessment tasks.
- *Wizard* package is able to trigger functionalities from *Data Research* and *Security Assessment* packages whose inputs rely on the secondary actors that are essentially data repositories.
- In addition, *Data Research*, *Testbed Management* and *Security Assessment* interact with secondary actors (that is, external sources) such as data repositories but also installation package repositories.

G.1.4 Physical View

Now that SCAPL's internals are defined, a physical implementation is proposed based on the software servers and libraries chosen in the SRS (see Appendix F). The characteristics of this implementation are the followings :

- [*scapl-frontend*] As FE relies on the Django framework, it will require an Apache web server running. Moreover, FE's data will be managed through MySQL as stated in the SRS. In the Process view, the sequences of events are presented without any consideration of which user input is required to make SCAPL work. This problem is overcome by using an FTP server in the Application tier for storing uploaded packages of the software products to be tested. For the final implementation of

SCAPL, it is expected to use SFTP for transferring files. Moreover, as also stated in the Process View, asynchronous tasks are triggered by DI's and the mechanism used by Celery imposes to use a specific server for message queuing, that is, a RabbitMQ server. The interaction with this server is managed through AMQP [1].

- [*scapl-backbone*, *scapl-search*, *scapl-automation*] At least one backbone machine is required to make the RabbitMQ and FTP servers run. SE and AS are separated in two other machines with their specific underlying tools and libraries. They both also require an AMQP link to the RabbitMQ server. AS also requires an FTP link to the FTP server in order to download software packages for setting up test environments. SE and AS both require a DATA link to the back-end machine in order to save their findings.
- [*scapl-backend*] BE runs in a separated machine relying on a NoSQL database management system (e.g. Apache Cassandra or MongoDB) and thus requires two DATA links to SE and AS.

All these considerations are depicted in the following deployment diagram.

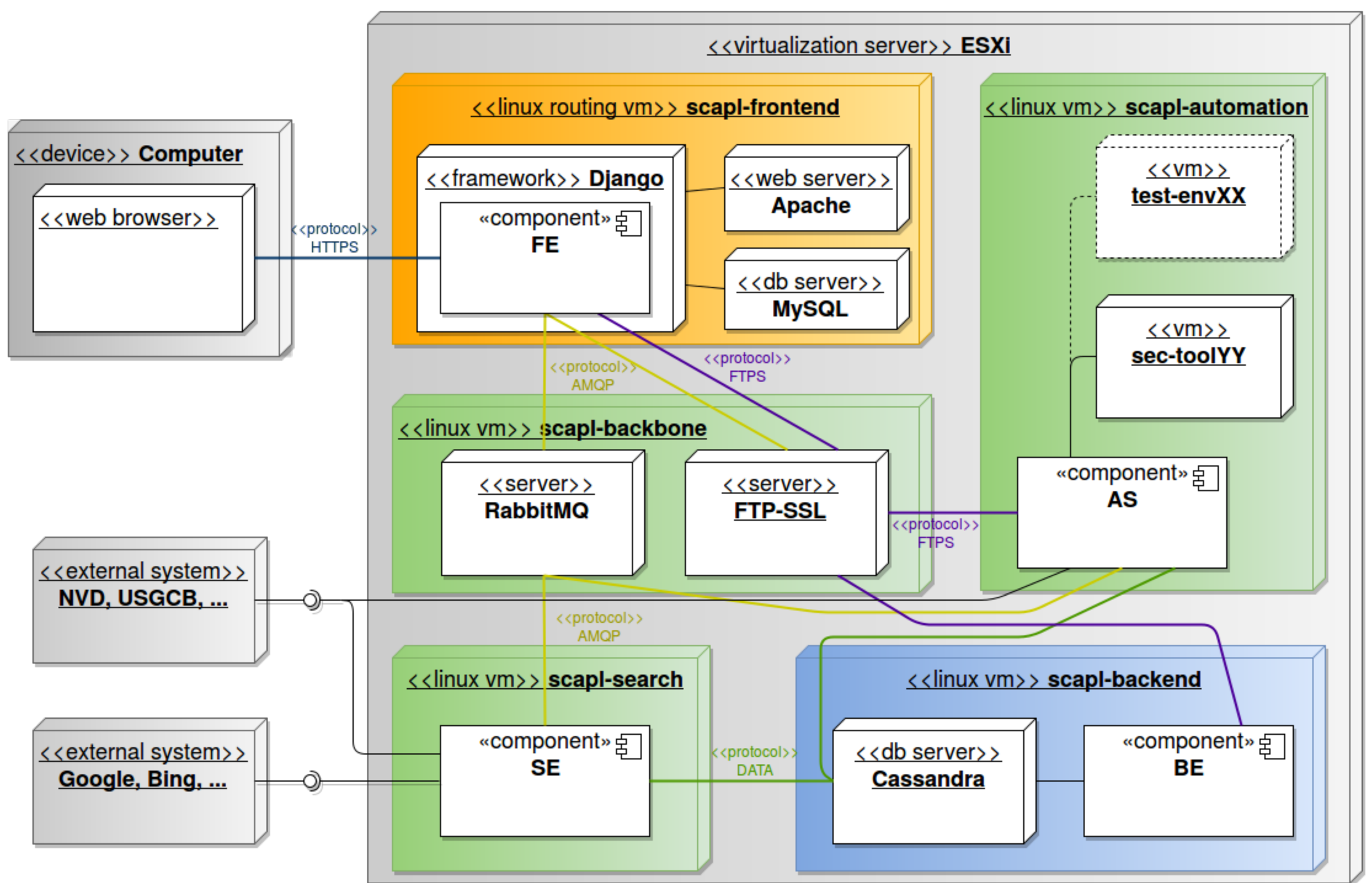


Figure G.7: Deployment diagram of SCAPL

First, for the sake of simplicity, SCAPL will be developed based on a simple virtual network architecture that does not provide any security for SCAPL's components. This virtual network is illustrated in the following figure.

Afterwards, the expected end-state of SCAPL is an architecture that judiciously separates components, baselines and test environments and carefully controls every access of each VM to the Internet. Such a segregation prevents potentially infected test environments to interact with any system's permanent machine. This way, a software product whose packages come with a malware cannot defeat the entire system.

[1] Advanced Message Queuing Protocol

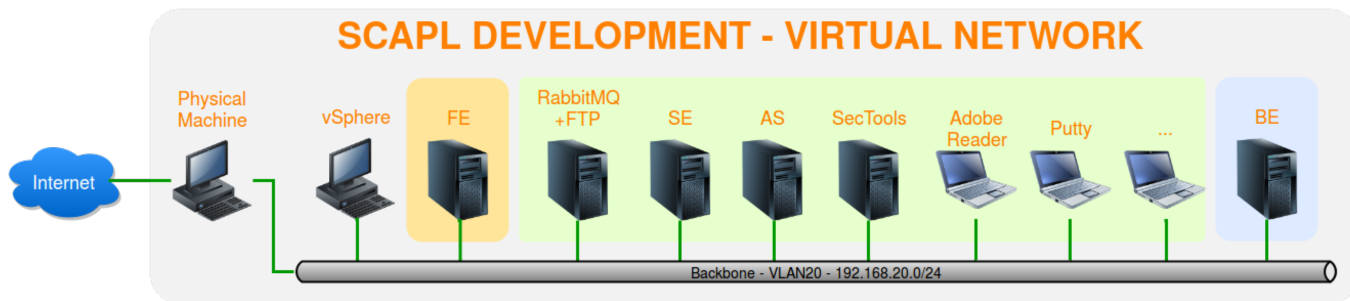


Figure G.8: Virtual network architecture for SCAPL (Development version)

These considerations are illustrated in the figure hereafter. One can point out that :

- A Proxy controls accesses to the outside and a Firewall/IDS performs routing and filtering from subnetworks from SCAPL's internals. This practice makes VLAN10 a DMZ with the Web service inside, providing security for the rest of the system if the front-end server was infected.
- Moreover, for the sake of monitoring, baselines (VLAN30) are separated from the backbone network (VLAN20 ; holding most of SCAPL's components) as these VM's are more volatile (the goal being to regularly deploy baselines to let them perform their updates and then to re-box them as template machines).
- A vSphere management machine is foreseen (in Windows 7), as it is required for managing the ESXi server supporting the whole infrastructure.

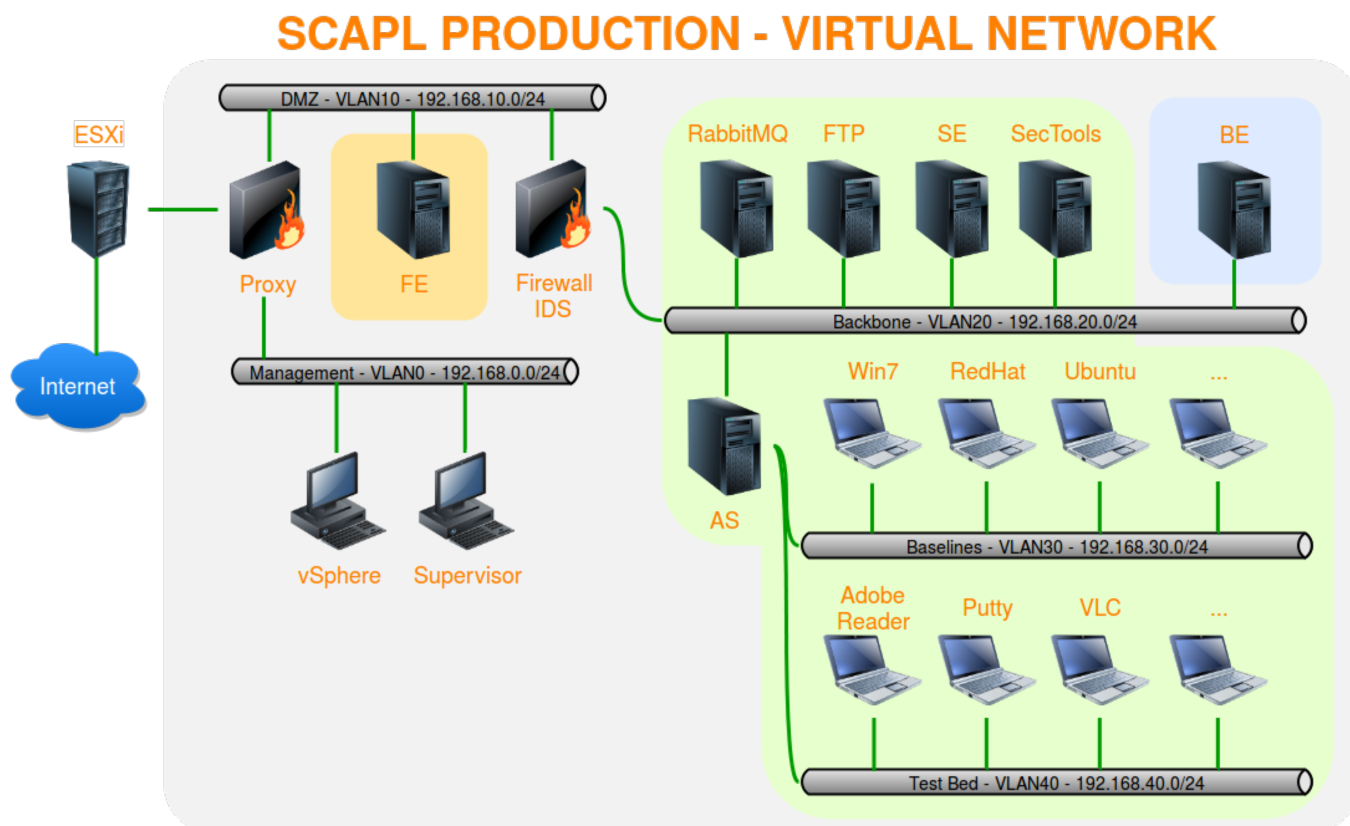


Figure G.9: Virtual network architecture for SCAPL (Production version)

G.1.5 Use Cases

The following UML Diagrams depict use cases for each package, identified by their requirement identifier from the SRS (provided in Appendix F).

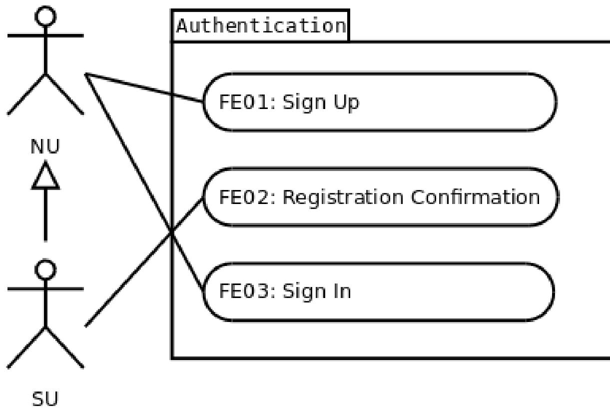


Figure G.10: Use case diagram for package Authentication

Authentication contains the functionalities to manage the authentication to the web interface of the system.

Note that NU does not need a Registration Confirmation whereas SU does. This design comes from the fact that NU's do not trigger heavy and/or harmful tasks in the system and could be more numerous (registration confirmation could block progress in APL tasks if too much NU's register). In contrast to NU's, SU's should be less numerous and could trigger sensitive tasks and therefore their management requires a confirmation for registering.

Profile contains the functionalities to manage user customizable items such as the profile or view of existing reports.

All the functionalities handled by this package are common to both NU and SU (as SU inherits from NU).

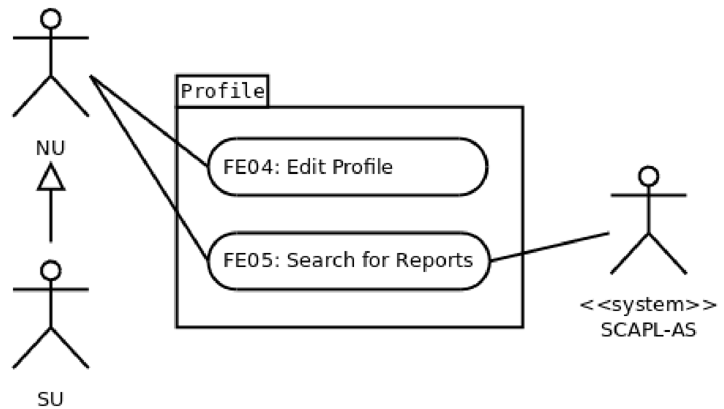


Figure G.11: Use case diagram for package Profile

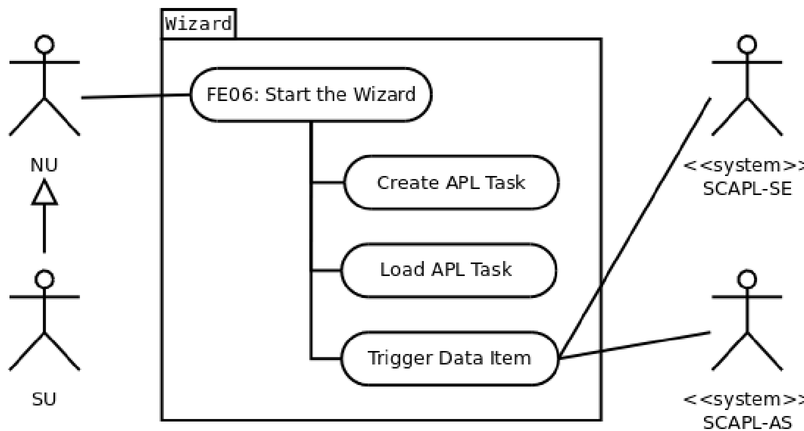


Figure G.12: Use case diagram for package Wizard

Wizard contains a primary functionality hiding three secondary ones for allowing the user to participate to APL tasks through a wizard.

The secondary functionality *Trigger DI* allows to trigger asynchronous task sending to system's internal components.

Admin contains the functionalities for ADM to manage all entities of the system from the users to the report structure.

As ADM is completely isolated from NU and SU, it has its own authentication feature.

As *Manage Users* handles profile management, *Manage Roles* relates to the association between user profiles and the data the users are able to act on.

Separately from these functionalities, the entire APL process can be refactored through the *Manage Process* package by adding, removing or reordering items in the data structure.

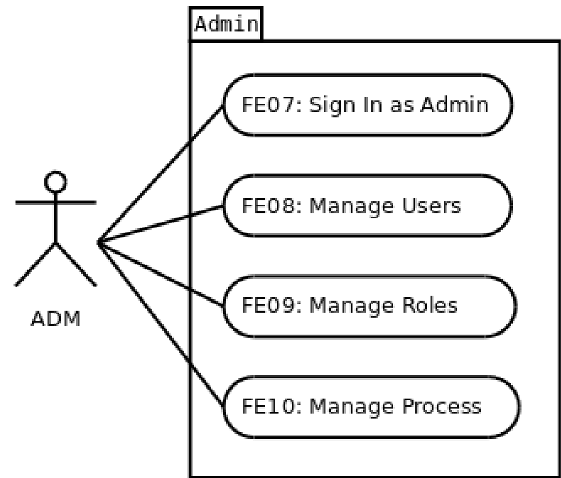


Figure G.13: Use case diagram for package Wizard

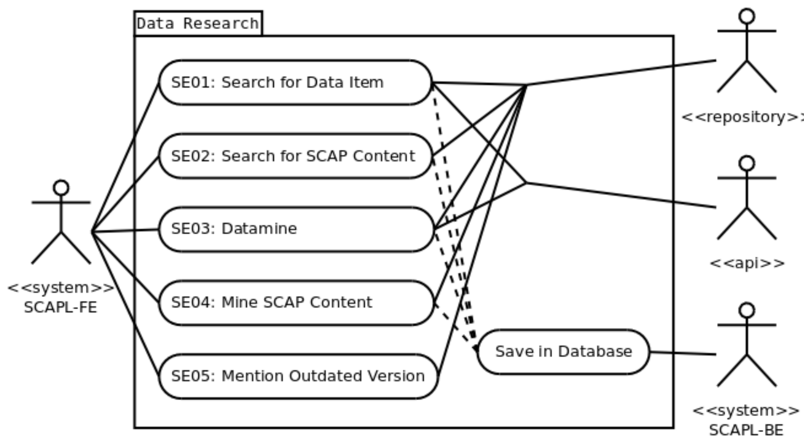


Figure G.14: Use case diagram for package Data Research

Data Research contains all the functionalities related to the search capabilities of the system.

The secondary functionality *Save in Database* is aimed to keep track of the data queried on the secondary actors (as a cache for queried information and as a storage for user's selected information).

The secondary actors are repositories (e.g. for SCAP) and API's (e.g. Google or Bing). Note that all the primary functionalities point to the repository generic actor (and not only SCAP ones), meaning that they could all rely on repositories for their search.

Testbed Management contains the functionalities to manage and deploy baselines as test environments.

Note that automating the baseline management is not a priority. Therefore, AS01 to AS03 are considered optional.

AS01 and AS02 point to a secondary actor as it requires package repositories to prepare a baseline.

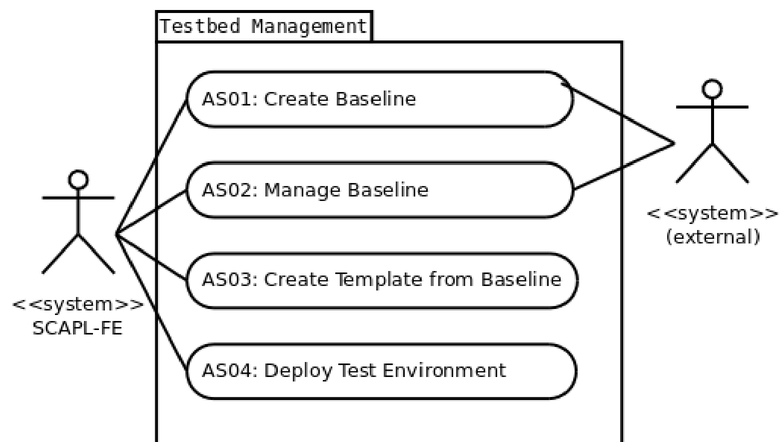


Figure G.15: Use case diagram for package Testbed Management

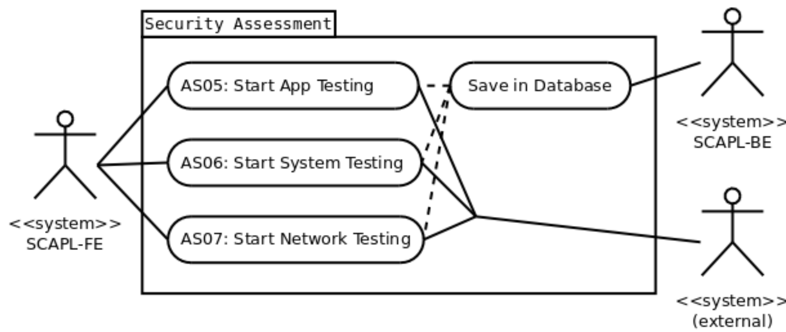


Figure G.16: Use case diagram for package Security Assessment

Security Assessment contains the functionalities related to the security task automation capabilities of the system.

Just like *Data Research*, *Save in Database* is the functionality that provides the interface to the back-end database for storing queried information.

In this diagram, the secondary actor external refers to created test environments but also to independent systems that could participate to the testing (e.g. a vulnerability assessment system such as OpenVAS).

Data Management contains the functionalities to configure the report generation feature but also to manage the database.

Note that BE02 is optional as, depending on the chosen DBMS[2], database management tools should certainly already exist.

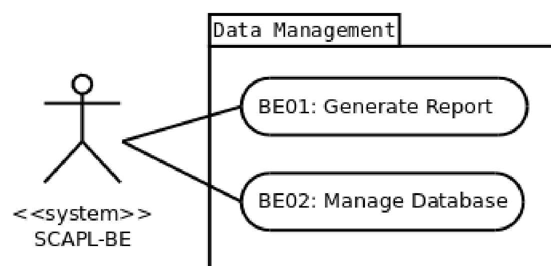


Figure G.17: Use case diagram for package Data Management

G.2 System Architecture

This section presents system's architecture through an architectural design figure, a decomposition in subsystems and the justifications of the design decisions.

G.2.1 Architectural Design

Principles:

- From a security point of view, a **3-tiers architecture** is a best practice (easier to harden, more difficult to cause data leakage).
- **Secure protocols** are used between each component.
- **Portability** : is a design and implementation constraint, therefore each sub-system is made to run on a VM such that it can be deployed on any operating system supporting a virtualization environment.
- **Scalability** is also a design and implementation constraint, therefore it should be possible to distribute the components across multiple separated hardwares.
- **Extensibility** is again a design and implementation constraint, therefore the component chains could be fully or partially isolated for easier integration of parallel projects (e.g. FE-SE-BE chain, or also an additional chain such as FE-[Project]-BE).

[2] DataBase Management System

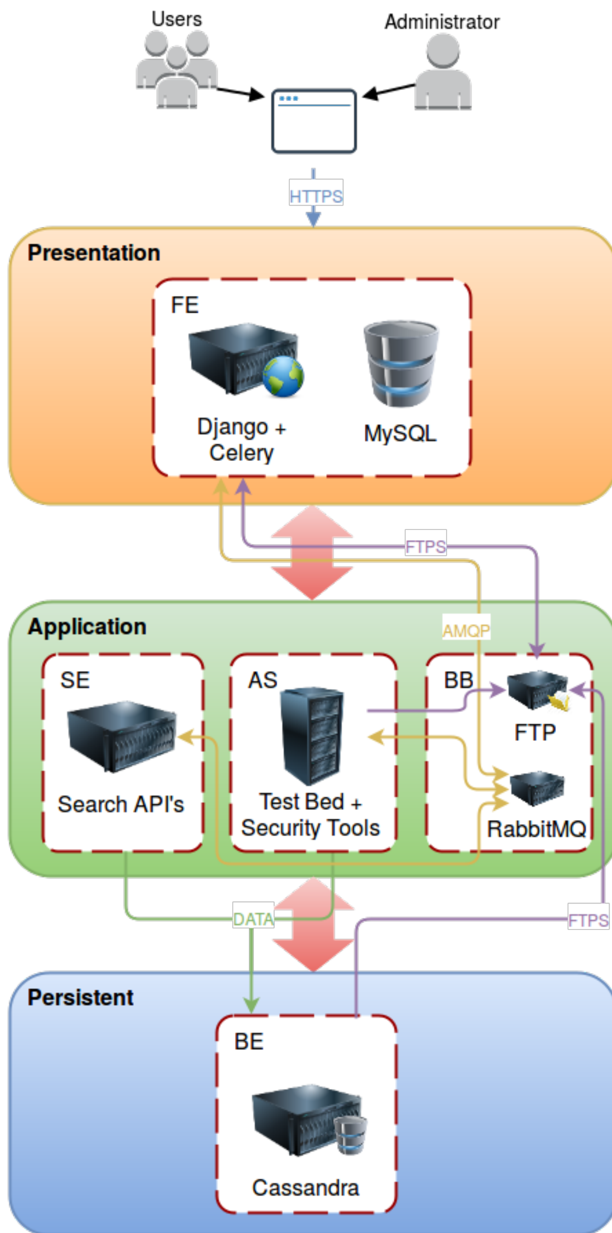


Figure G.18: 3-Tiers Architecture of SCAPL

Presentation: consists of a component hosting the web server (implemented with the Django framework) with its dedicated relational database MySQL or SQLite) aimed to hold the data scheme for the dynamic wizard to be exposed to the users.

Application: can be implemented in many different ways to provide strict independence between its components (FTP + RabbitMQ servers inside the components) or to facilitate deployment and management (centralized FTP + RabbitMQ servers, as depicted in the figure on the left) and contains multiple processing components for supporting FE data item tasks.

Persistent: owns the storage database for the data collected by the Application layer.

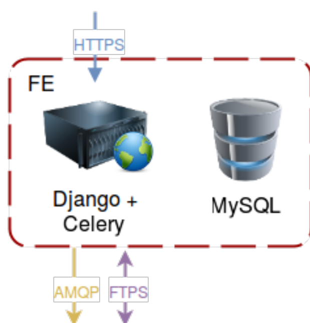
The responsibilities of each layer and component is explained in the following table :

Layer	Component	Responsibility	Description
Presentation	FE	Web Interface	FE exposes a web interface to the users via HTTPS, providing a wizard based on a data scheme stored on FE's database, in order to help them fulfil software security assessments.
Presentation	FE	User inputs	A wizard for a new assessment always includes user inputs for keywords (as a base for the future researches by SE) and packages (for installing the product to be tested in AS).

Presentation	FE	Asynchronous task invocation	Each time a DI is invoked by FE, an asynchronous task is sent via AMQP to the RabbitMQ server in the Application layer.
Application	BB	Task management	When a task is invoked, it is distributed to the responsible component by the RabbitMQ server.
Application	BB	File storage	The uploaded packages received from FE are to be stored on the FTP server. The reports generated from the data stored in BE are also stored in this FTP server for availability to FE.
Application	SE	Data research	When a task is received, SE cares for : • researching information via its API's • saving the collected relevant information into BE
Application	AS	Security assessment	When a task is received, AS cares for : • setting up a new test environment • triggering security (sub-)tasks for assessing the installed product • saving the collected relevant information into BE
Persistent	BE	Data recording	BE provides an API to SE and AS in order to save required information.
Persistent	BE	Report generation	BE regularly computes PDF reports from its stored data and sends these to the FTP server in the Application layer for availability to FE.

G.2.2 Decomposition Description

As already sketched in the figure in the previous subsection, the system is decomposed into 5 components. This subsection is aimed to give more details on each separate component. The proposed configuration is considered as a starting point but could be refined/scaled.



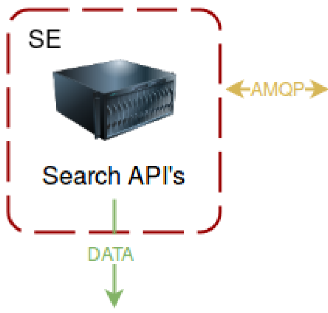
Configuration:

- Single VM with Ubuntu Server, Python (with Celery and Django framework modules) and MySQL (not required if using SQLite) installed

Characteristics:

- This component uses a typical Django project structure with the particular packages integrated
- Wizard package holds the data scheme (discussed in the next section) about DI, DL and DS and the Celery code for triggering asynchronous tasks
- Profile package holds the data scheme (discussed in the next section) about NU, SU and ADM

. . . .



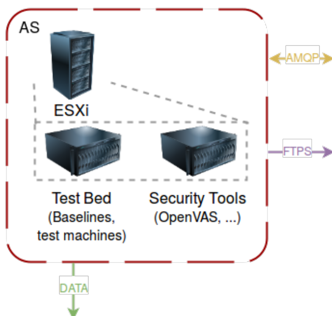
Configuration:

- Single VM with Ubuntu Server and Python (with Celery and Cassandra /MongoDB client modules) installed

Characteristics:

- This component contains a modular project structure allowing to easily add new features accessible from FE via the DI's
- The search API's are gathered in the *Data Research* package

---



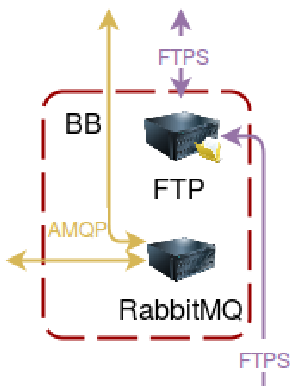
Configuration:

- Physical machine with Ubuntu Server installed for supporting the virtualization environment (ESXi Server)
- VM with Ubuntu Server and Python (with Celery and Cassandra /MongoDB client modules) installed for supporting the automation scripts (Test Bed)
- If required, one VM per security project installed (Security Tools)

Characteristics:

- This component contains a modular project structure allowing to easily add new features accessible from FE via the DI's
- The automation features are gathered in the *Testbed Management* package
- The testing features and interfaces are gathered in the *Security Assessment* package

---



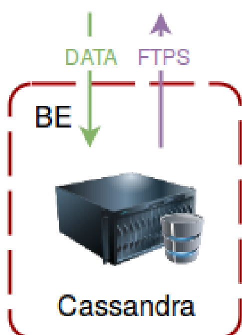
Configuration:

- Single VM with Ubuntu Server, RabbitMQ and an FTP server installed

Characteristics:

- This component coordinates the asynchronous tasks coming from FE and intended for SE and AS
- This component does not require any particular package and just needs to be adequately configured

---



Configuration:

- Single VM with Ubuntu Server and Python (with Cassandra or MongoDB client modules) installed

Characteristics:

- This component contains a daemon for regularly generating reports and eventually DB tools
- The related features are gathered in the *Data Management* package

G.2.3 Design Rationale

The base principles are already explained in Subsection G.2.1. One must point out that, regarding the configuration of AS, the ESXi Server will in fact be used to support the entire architecture of SCAPL as depicted in Subsection G.1.4 in Figure G.9, meaning that AS will in fact be a VM controlling the creation/deletion of test environments through Vagrant and Ansible.

Referring to non-functional requirements and design and implementation constraints from the SRS, PERF0X, ERF0X and SECU0X are ensured at FE level. For the aforementioned constraints,

- **Extensibility**, related to adding new core features (by creating a new Application-layer component), is enforced by separating the Application components as much as possible (by putting the RabbitMQ and FTP servers at the component level and not at the Application layer level). Within the components, extensibility is also achieved by implementing packages in such a way that new Python scripts can be easily added and linked inside FE through DI's.
- **Scalability** is ensured by splitting responsibilities as explained in the table in part *Architectural Design*. Indeed, at FE level, a load balancing can be set up by simply duplicating the web server and configuring it to do so. At SE and AS levels, several instances can be contacted on different message queues managed by the RabbitMQ server(s) to distribute processing. At BE level, Cassandra/MongoDB is naturally designed to scale to multiple distributed systems.
- **Security** is ensured, according to the Defence-in-Depth principle, by using a 3-tiers architecture. Indeed, servers at the different tiers can be more easily hardened given their roles (e.g. by using a host-based firewall only allowing required protocols) and the Presentation tier can never contact the Persistent tier in order to prevent an attacker from altering stored data. The only weak point is the relational database at FE level and FE's security settings must be carefully configured.

Note that this last consideration explains why the FTP server is located at the Application level. As it could be considered as persistent data storage, it actually only handles data in support of AS (and possibly SE in the future) or FE and not long-term data processed by the system such as the data stored in the NoSQL database (Cassandra or MongoDB). Indeed, it holds the required packages provided by the user through FE for testing purpose and the reports generated by BE at the disposal of FE for consulting purpose.

G.3 Data Design

This section gives details about the data scheme underlying the system and describes the relevant data structures. It explains the DS-DL-DI scheme for supporting the dynamic wizard and explains user and APL task schemes for supporting user and report managements.

G.3.1 Data Description

At the relational database level, in the **Presentation** layer, the data scheme is split in **two** main **categories** :

1. (Nearly-) **Static** : corresponds to the data scheme supporting the dynamic wizard. This is qualified as dynamic for emphasizing the fact that, through this particular data scheme, the final report structure can be customized and tailored to organization's needs. It is translated at the data level into a scheme that, normally, should not change very often, hence the category (nearly-)static.
2. **Volatile** : corresponds to the data scheme for supporting user and tasks management. Indeed, user profiles have to be stored somewhere and available for FE. Moreover, once a wizard is started with keywords matching a new software product, a new task (with its metadata) has to be recorded in order to keep track of it. Therefore, in contrast to the previous category, the data is volatile.

The Entity-Relationship model for the **static data scheme** is depicted in Figure G.19.

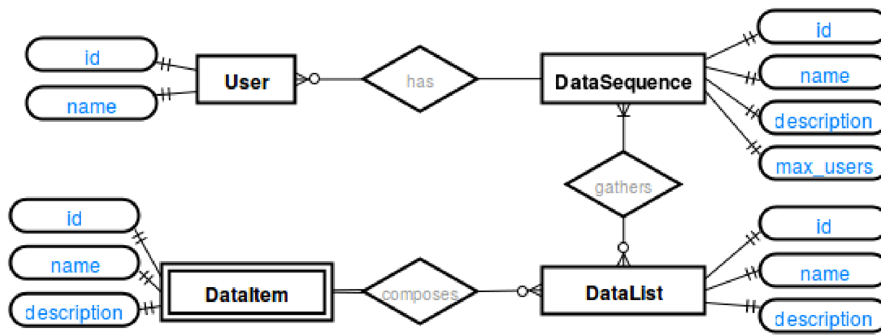


Figure G.19: Entity-Relationship model for the static data scheme

Note that there obviously exists a link between the static and volatile data schemes at the user level as the wizard has to be adapted according to NU and SU. The previous figure mentions a generic User entity, which is the parent of NU and SU.

The Entity-Relationship model for the volatile data scheme is not depicted as it does not require to be detailed due to its simplicity (as it can be seen in the next part).

G.3.2 Data Dictionary

The EER[3] model for the volatile and static data schemes, more suitable for an ORM[4] (required for writing models into the Django web server), is depicted in Figure G.20 and gathers all relevant data entities.

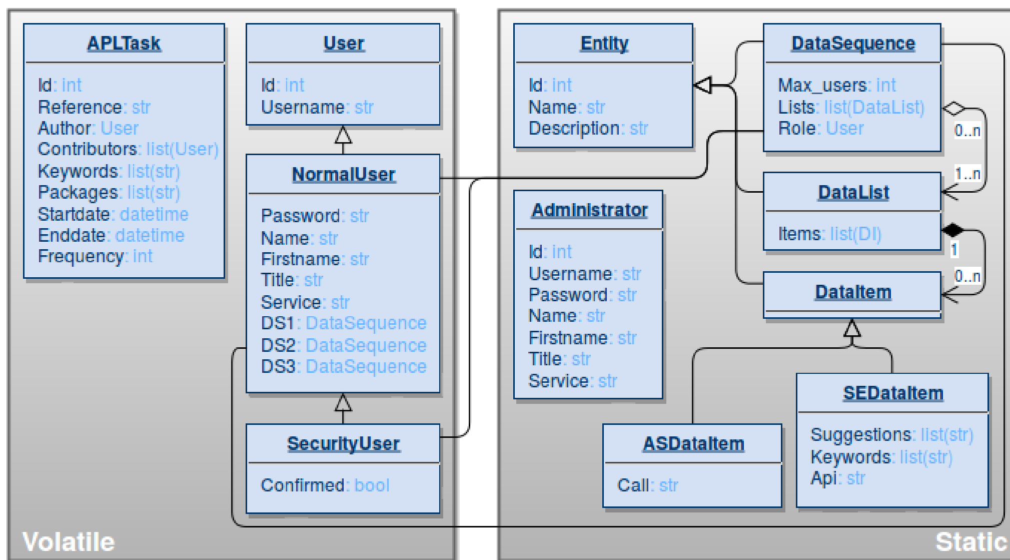


Figure G.20: EER model for the volatile and static data schemes

In this model :

- **APLTask** : holds the data related to a started *APL task* (to be distinguished from the asynchronous tasks triggered by the DI's).
- **User** : is a generic entity aimed to keep the base information of the possible high-level users, that is, NU, SU, ADM but also a predefined one with ID 0 named System, useful for tasks periodically generated (cfr *Frequency* of an APLTask).

[3] Enhanced Entity-Relationship
[4] Object-Relational Mapping

- **NormalUser** : represents a basic user, that is, NU which only accesses research functionalities.
- **SecurityUser** : represents an assessor, that is, SU which also accesses automated assessment functionalities in addition to these of NU.
- **Administrator** : represents a system administrator, that is, ADM which lies in the static scheme for the same reason as for the *DataSequence*, *DataList* and *DataItem* and is physically isolated from the user interface as it is part of the model of the *Admin* package.
- **Entity** : is a parent entity for inheriting base attributes, that is, *Id*, *Name* and *Description*.
- **DataSequence** : represents an ordered list of *DataList*'s, invocable by a defined maximum number of users having a specific role, such that a DS is a composition of at least one DL.
- **DataList** : represents an ordered list of *DataItem*'s such that a DL can be referenced by zero or more DS.
- **DataItem** : is a parent entity for defining data items intended for either SE or AS.
- **SEDataItem** : represents a DI aimed to research the related information providing suggestions based on given keywords and calling a specified API.
- **ASDataItem** : represents a DI aimed to process an assessment information given a call to a specific tool.

Note that, as depicted in the EER model, the Administrator entity is isolated in the static scheme and therefore does not interact with the APL process. That is a design choice aimed to segregate roles on the front-end. Separating the *Admin* package and the *Administrator* role like this allows to better secure the administration part as NU/SU and ADM do not share the same authentication mechanism.

G.4 Component Design

This section gives details about the component internals. It explains how each component is design to fulfil its requirements and interact with the other components.

G.4.1 Generic Structure

Each component of SCAPL is built according to a template, as shown hereafter.

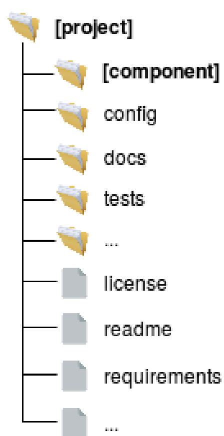


Figure G.21: Generic component internal structure

- **[component]** : is the folder containing the source code of the component (see next subsections).
- **config** : contains the settings of the supporting system.
- **docs** : holds the relevant documentation for the given component.
- **tests** : is folder with unit tests for assessing component's source code.
- **...** : designates other possible folders, specific to the component (e.g. **static** and **media** files for FE).
- **license** : holds the copyright and, if relevant, an open-source license such as GPL[5].
- **readme** : is the common basic help document for any project.
- **requirements** : is the list of required packages of the component for Python.
- **...** : designates other possible files, specific to the component (e.g. **tasks.py** and **worker.py** files for SE and AS for the communication with the RabbitMQ server).

[5] GNU Public License, a typical license for free software

G.4.2 Front-End

The front-end is built using the Django framework. It then imposes a way to structure the project. This structure is depicted in Figures G.22 and G.23.

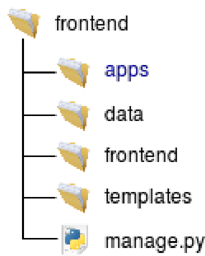


Figure G.22: FE internal structure

- **apps** : contains the applications (see details in the next figure) corresponding to the packages presented in the package diagram.
- **data** : contains some initial data.
- **frontend** : is named like the project by convention in Django and essentially holds the settings.
- **templates** : contains the HTML templates processed by Django according to the MVT[6]model.
- **manage.py** : is the script for running the Django server.

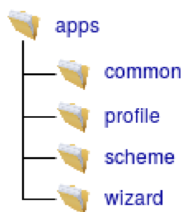


Figure G.23: FE internal applications

- **common** : corresponds to *Authentication* package and is renamed to avoid name clash with Django built-in module *auth*.
- **profile** : is the feature package as described in Subsection G.1.5.
- **scheme** : corresponds to the part of the *Admin* package identified by FE10 as a great part of the administration features is already available via the built-in module *admin* of Django.
- **wizard** : is the feature package as described in Subsection G.1.5.

Note that this structure is mostly inspired from a documentation maintained by an experimented web developer available at `http://django-project-skeleton.readthedocs.io`.

G.4.3 Search Engine, Automation System

Both components follow the same structure such as depicted in Figure G.24.

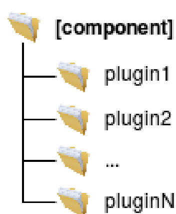


Figure G.24: SE and AS internal structures

The structure of each component is very simple and only consists of one folder type.

pluginN : contains the source code of a plug-in project adapted to provide an API for the asynchronous tasking, that is, for the communication from FE to the component through the RabbitMQ server.

Note that, at the root of the component (corresponds to **[project]**), the files tasks.py and worker.py are present respectively for defining the tasks to be asynchronously run and for running a listener that collects and sends messages from/to the RabbitMQ server.

[6] Model – View – Template ; a web development design pattern similar to the Model – View – Controller (MVC) model