TÉCNICO LISBOA

Data Center Portal Software Architecture

A NOS Operations and Supervision System

Sérgio Miguel de Albuquerque Alves

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. António Manuel Ferreira Rito da Silva

Examination Committee

Chairperson: Prof. Alberto Manuel Rodrigues da Silva

Supervisor: Prof. António Manuel Ferreira Rito da Silva

Member of the Committee: Prof. Miguel Leitão Bignolas Mira da Silva

October 2015

Acknowledgments

I would like to thank all the members, that are or have been part of the data center team, for their continued support, trust and opportunities given. It is nice to be part of a project which can directly improve other people daily work-life.

I would also like to thank for all the support and feedback given by professor António Rito Silva.

Abstract

NOS, a fusion between Zon, Optimus and most recently Mainroad, is one of the leading corporations in the portuguese telecommunications market. Its quest for providing quality and innovative services, demands for very flexible organic structures and processes, in order to ensure that its business requirements are being fulfilled within the required time to market, and following the required quality standards. In this thesis, we will describe the agile process of designing a software architecture, which started being development in ZON age, with the goal of supporting most of the underlying processes belonging to the Department of Data Center Operations. We will also look into how the architecture evolved, in order to cope with the challenges that rose from merges between Zon, Optimus and Mainroad.

Keywords

Data Center, ITIL, Software Architecture, Portal

Resumo

A empresa NOS, resultado da fusão entre Zon, Optimus e Mainroad, é uma das empresas líderes do mercado de Telecomunicações em Portugal. Com a sua missão de providenciar serviços de alta qualidade e com alto teor de inovação aos seus clientes, necessita internamente da existência de uma estrutura orgânica com processos flexíveis, capazes de dar reposta aos requisitos de negócio, dentro das janelas temporais para o negócio, e dentro dos padrões de qualidade pela a qual se caracteriza. No corpo desta tese, será descrita a arquitetura de software ágil, que nos seus primórdios, começou a ser implementada na ZON, para informatizar a maior parte dos processos pertecentes à Direção de Operação de Centros de Dados. Também será apresentado como é que essa arquitetura evoluíu com o tempo, de forma a conseguir adaptar-se aos universos que surgiram com as fusões das várias empresas.

Palavras Chave

Centro de Dados, ITIL, Arquitetura Software, Portal

Contents

List of Figures

List of Tables

Abbreviations

ABC Architecture Business Cycle

ACID Atomicity, Consistency, Isolation and Durability

ACL Access Control List

AD Active Directory

AJAX Asynchronous JavaScript and XML

API Application Program Interface

BaaS Backup as a Service

BE Back-End

CGI Common Gateway Interface

CI Configuration Item

CMDB Configuration Management Database

CM Change Management

COTS Commercial Off-The-Shelf

CRUD Create, Read, Update, Delete

CSS Cascading Style Sheets

CSV Comma Separated Values

CS Client Server

DAO Data Access Object

DB Database

DC Data Center

DMZ Demilitarized Zone

DRY Don’t Repeat Yourself

EV Easy Vista

FCGI FastCGI

FE Front-End

FIFO First In First Out

HPSM HP Service Manager

HTML HyperText Markup Language

HTTPS Hypertext Transfer Protocol Secure

HTTP Hypertext Transfer Protocol

IaaS Infrastructure as a Service

IM Incident Management

IOC Inversion Of Control

IP Internet Protocol

ISAPI Internet Server Application Programming Interface

ISO International Organization for Standardization

ITIL Information Technology Infrastructure Library

IT Information Technology

JSON JavaScript Object Notation

JS JavaScript

KPI Key Performance Index

LDAP Lightweight Directory Access Protocol

MVC Model View Controller

NaaS Network as a Service

OLA Operational Level Agreement

ORM Object-Relational Mapping

OU Organizational Unit

PM Problem Management

QA Quality Assurance

SCGI Simple Common Gateway Interface

SLA Service Level Agreement

SMTP Simple Mail Transfer Protocol

SOAP Simple Object Access Protocol

SPOC Single Point of Contact

SQL Structured Query Language

SR Service Request

SSJS Server-Side JavaScript

TCP Transmission Control Protocol

TT Template Toolkit

UML Unified Modeling Language

URL Uniform Resource Locator

VIP Virtual IP

VPN Virtual Private Network

WSGI Web Server Gateway Interface

XML Extensible Markup Language

1

Overview

Contents

Before we dive into the technicalities of the created software architecture, it is important to grasp the underlying problems that Zon was facing back in the start of the project, and, how that led to the creation of a system that envisioned to incrementally tackle one by one and solve them.

1.1 Introduction

The Department of Data Center Operations is responsible to ensure the correctness and operationality within all the NOS Data Centers. It also acts as service provider, which exposes data center services, to the rest of the organization. It possesses a wide range of internal clients, which are responsible for different business areas within the company, such as: the Department of Information Systems, responsible for all the applicational layers from systems that support the daily activities within the company; the Department of Product Development, responsible for all the development of systems and services that stride to provide the best TV experience to the clients; the Department of Operation and Supervision, responsible for the systems that support all internet service provider systems; All the departments related to enterprise public offers, which are responsible for creating and maintaining the solutions for external corporate clients.

The services provided by the Data Center can be seen, partly, as Infrastructure as a Service (IaaS), Network as a Service (NaaS) and Backup as a Service (BaaS), where all the provisioning and operations are done manually, rather than in automatized manners, as required by the full definition of IaaS, NaaS and BaaS. This means that, according to IaaS, a data center client can request for the installation of physical & virtual servers, he can requests for their removal, he can request for the addition & removal of computing resources such as cpu/ram/storage, he can request for the installation of a given operating system, he can request for a given software package to be installed/uninstalled, he can request for a given user to be created/removed from the corresponding operating system, etc.

By default, due to security guidelines, network traffic within Data Centers, from one network to another network, is blocked. Therefore, as part of its NaaS strategy, the client is allowed to request for the creation of a new network architecture which are assigned to specific new projects, he can request for the configuration of exception within the firewall flow rules, that unlock traffic from a given source network into a given target network on a specific port, he can ask for Virtual Private Network (VPN) configurations, he can ask for Virtual IP (VIP) balancing of systems, he can ask for the troubleshooting of issues related to connectivities problems.

Likewise, a portfolio of backups is provided to our client to ensure that if necessary, our clients will always have a way to go back in time and recover their data. This is very useful, especially when the clients are upgrading their platforms, and want to have a roll back plan in place, just in the case of something going wrong. Thus, the clients can schedule daily/weekly backups, into given points of their file systems, databases, etc. They can also request for one time backups, which happen only once on the specified date and time. Finally, they can ask for a given scheduled backup to be skipped once, on a given occasion.

All these services are maintained and performed manually by specialized teams within the Data

Center Operations Department. While some of them specialize in virtualization, others specialize in Windows systems, Linux/Unix systems, networking, backups, monitoring and management of data center rooms. The handshake, between the clients and these teams, is done by a delivery team, who is responsible for assuring the quality of the delivered services by sponsoring certain processes such as: the control of the Service Level Agreements (SLAs), the dispatching of work tasks to the intervening operational teams, the early validation of the content provided inside the clients requests, the validation between what was requested and what was implemented before delivering it back to the client, and finally, the documentation of all the necessary operational information and processes within different repositories.

All the teams mentioned behind, except for the delivery team, are also responsible for solving incidents that may arise on the infrastructure and systems managed by the area. Incidents are events which impact a given managed system, in such a way that it affects their normal operationality. They can be caused by the side effects of intervening in the infrastructure, faulty configurations, erroneous human intervention, computing resources usage reaching their limits, etc. They are usually detected by the owner of a given system and reported to team responsible for supporting it. If corrected pre-emptively, by the team responsible for supporting the system, who detects the issue through means of automated monitoring and alarms, the issue will not be raised as an incident.

1.2 Information Technology Infrastructure Library Framework

Most of the methods and processes described before are based in the Information Technology Infrastructure Library (ITIL)[1] framework. This framework is a set of best practices for Information Technology (IT) Service Management that is widely used in telecommunication organizations. The processes it describes are generic enough to allow their implementation on any type of organization that seeks maturity and standardization in its processes and procedures. Since the subjects of its applicability are so wide, organizations can decide which bits to implement first to achieve their goals. A deep dive into the intricacies of the framework is outside the scope of this thesis, thus we shall only lay ground on the parts that were implemented in NOS and that directly affect the data center department.

One of main core concepts introduced by ITIL is the existence of a Configuration Management Database (CMDB)[2]. The CMDB is a repository which holds the contents and relationships of Configuration Items (CIs). Configuration items are IT entities that can range from computers, laptops, blades, racks, virtual servers, networks, to business services, technical services, or any other entity defined within ITIL universe. The idea is to have everything meaningful to the organization registered, in order to be able to nominate it and keep track of all its relationships and history of changes. This is essential because it brings to the organization the quality of traceability within its processes and procedures.

ITIL defines a Request Fulfilment process that is responsible for the handling of Service Requests (SRs). Service requests are initiated by a third party which goes by the name of the requesting

party and, usually end up in the queue of an implementation party. In this context, the data center department is one of the many implementation parties for service requests, while the other departments performing the requests, are seen as the requesting parties. Examples of service requests are the creation of a new user in the a given server, the creation of a new virtual machine, the installation of a given package inside a machine. Therefore, it is quite common that service requests affect the existing configuration items. When such happens, a relationship is created between the service request and the configuration item, relation which documents the intent, so that we may audit it later if required.

The request fulfilment also states that there should be tools for effectively and efficiently handling these service requests. The service requests must have traceable implementation times, which are defined within the Service Level Agreements, so that one may be able to measure the effectiveness and efficiency of a given implementation party. The service requests can also have a life cycle, which may start from the moment they are created, and go on until the moment when they are completed, while having in-between any other state deemed necessary (eg. authorization states, pending information states). The life cycle of a request is continuously monitored so that counter measures are in place, should the SLA be breached. It is also important to evaluate and check the results of a service request, in order to provide quality control on the deliverance of them, and to record useful knowledge for future use.

This leads us to the Change Management (CM) process also described within the ITIL framework. The CM process manages the alterations which will be performed on configurations items. When such alteration is prone to happen, an event named change is initiated. This event may or may not need an approval, and may have an impact and a risk associated. As result, it will trigger changes within the status of the affected configurations items. By definition, a configuration can be anything that provides value to the business, thus, it means that the change management process itself, can also be a configuration item. Hypothetically, if we were to improve the way this process is managed, we would be able to document the improvement within the ITIL framework, by creating a change upon the change management process configuration item.

If we contextualize together the request fulfilment process and the change management process, we have service requests that will result in the creation of corresponding changes. Likewise, we have service requests that will never trigger the creation of a change, since no changes were made to any configuration item. A service request for upgrading the operating system of a given set of configuration items fits the former, while a service request to obtain certain system information from a configuration item fits the later. Therefore, we can say that a change will document what happened to a given configuration item on a given time. It is quite common to have environments[1] associated to configuration items. If we follow the affected configuration items environments and relationships, we can estimate the risk and the impact associated with a given change. It is then possible, to define levels of authorization that fit the business, by conjugating these categorizations.

Service requests are not the only triggers for a change. Incidents and problems, also described

[1] such as development, tests, staging, production

within the ITIL framework, can originate the creation of a change. The creation of an incident is defined within the Incident Management (IM) process. The primary goal of the IM process is to recover the service back to its clients as soon as possible. An incident is usually a ticket where the description of the issue detected can be found, along side with the affected configuration items and the categorization of its impacts and risks. It becomes then possible to prioritize the incidents from different sources, and to assign them different answer/resolution times. The incident is first assigned to a 1st level support team. The 1st Level Support is a Single Point of Contact (SPOC) and acts as the first aid team, since it has enough knowledge to perform simple troubleshooting and to apply workarounds if necessary. If the complexity of the incident surpasses the general knowledge of the 1st Level Support, there is a specialized 2nd Level Support to where it gets reassigned. This new level becomes then the responsible for the recovery of the service. If a correction or a workaround on the root cause cannot be reached by then, the incident is escalated to a problem.

Thus, the Problem Management (PM) process from the ITIL framework comes in to picture. A problem is the source of one or more incidents, therefore the problem management process main goal is to prevent, eliminate and minimize the impact of incidents by preventing and resolving the originating problems. It is also responsible to ensure that any resolution applied to a problem, is correctly implemented adhering to the appropriate means of the framework, such as the change management process presented before. It is also important to maintain a knowledge database where the workarounds and resolution are registered for future use, should a very similar problem arise in the future. It is quite possible that the participation of supplier of a given configuration item will take place in the findings of a solution or workaround for a problem.

An example of an incident that will escalate to a problem, which needs the intervention of its supplier support, is an issue in a given firmware version of a data center network node. Lets assume that the firmware has a glitch in which after the node reaches a certain number of connections, it starts dropping them randomly. At first, incidents will be reported by the owners of the systems whom depend on it. After the 1st and 2nd level support team evaluate the issue, they will come to the conclusion that connections are indeed being dropped even though all the node configuration seems to be correct. A problem in the given network node is then created and reported to the supplier's support. Together, with the supplier's support, the workaround of applying a patch to the firmware will be reached. This update to the firmware will be applied to the node while adhering to the change management process, where one will have to create a change which indicates that the network node configuration item received a firmware update on a given period.

1.3 Maturity Model

Gartner's IT infrastructure and Operation Maturity Model[3] introduced a method that provides the means for organizations to evaluate their maturity in regards to people, processes, technology and management. This is particularly useful because, it allows for an organization to identify where it is standing regarding to standards, and what are the required steps to reach the next maturity level.

The maturity model is divided in six distinct levels of incrementing maturity: Survival, Awareness, Committed, Proactive, Service-Aligned, Business Partnership. These levels behave like a staircase, where an organization reaches the next maturity level, every time it fulfils its defined objectives.

The Survival level is characterized by an organization having no focus in regards to IT infrastructure and operations. As soon as an organization realizes that infrastructure and operations are critical to its core business, beginning to readjust itself internally in order to gain control of it, it reaches the level of Awareness. It becomes committed once it starts to support its daily routines with well defined processes that ultimately improve customer satisfaction. It becomes Proactive by improving service quality through standardization, policy development, governance structures and implementation of organizational wide processes such as the ITIL framework. Once it starts managing infrastructure and operations like a business, becoming completely costumer focused, providing quality proven services, defining competitive SLA, the organization reaches the Service-Aligned level. Finally, the Business Partnership level can be achieved once the IT infrastructure and operations provide increase value and competitiveness to the business itself.

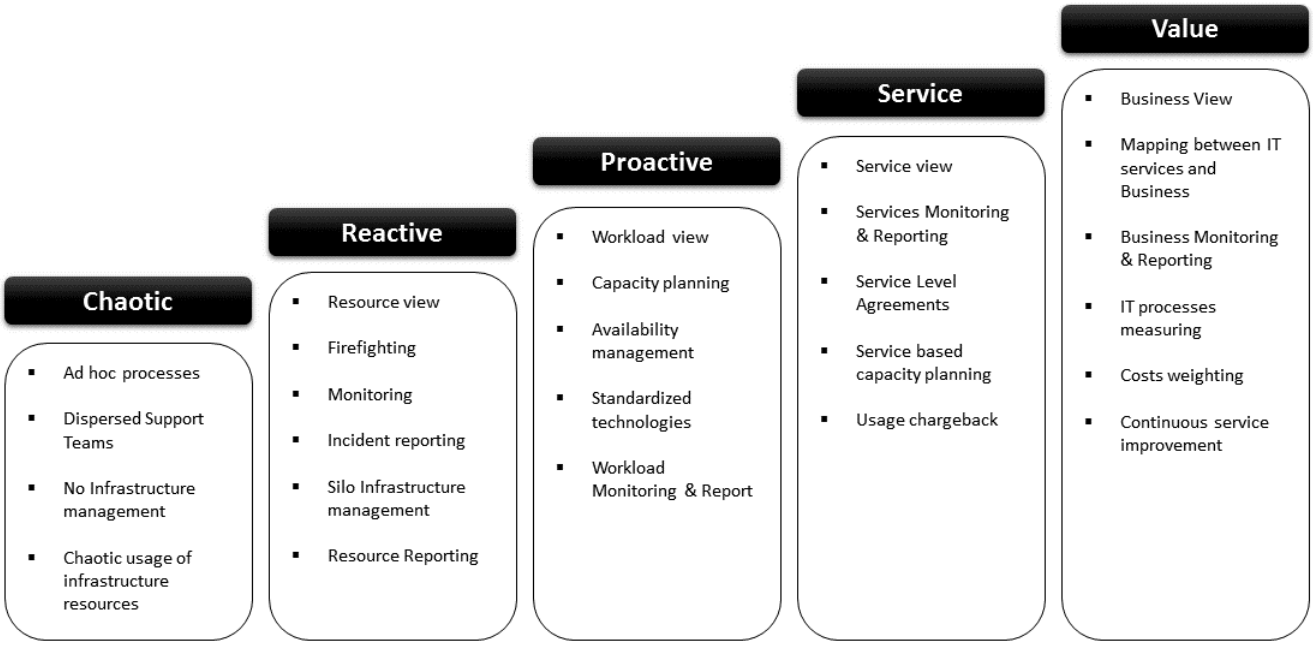

Figure 1.1: Capacity management maturity model levels

TeamQuest, a global leader in IT Service Optimization, developed yet another maturity model based on the previously presented Gartner's model. TeamQuest's Capacity Management Maturity Model emphasizes on applying Gartner's model onto the capacity management. We will be using this model as a reference since it is a bit more specific than Garner's original model, and since it fits neatly in the reality of our organization. This model is also divided in several different levels of incremental maturity: Chaotic, Reactive, Proactive, Service, Value (fig. 1.1). An organization with chaotic maturity level operates in a way that everything is done and thought in an ad-hoc manner, meaning that there are barely processes defined for anything. Thus, there is no infrastructure management, nor centralized support. Different departments grow and consume infrastructure resources on the go without

any planning or attempt to centralize responsibilities. In organizations of these maturity level, it is a big challenge to understand what infrastructure is being used within the organization as a whole.

In the next level, the reactive level, we see the appearance of some of the ITIL procedures. Organizations know how to deal with incidents, how to deal with changes, how to create configuration items in the CMDB. This leads to a better understanding of the available resources to the organization. This perception of the resources and responsibilities may still be separated by department silos, but within each department, there are now procedures in place. The monitoring of configuration items becomes also available, even though, it is used in a reactive way. The means, to identify a possible source for an incident, are in place, although they are only set in place to identify the source of an incident after the damage is done and not before, thus the fire fighting. As an organization steps onto the proactive level, we start to monitoring in order to better plan the future and prevent fires. A big source for incidents is resource depletion. Monitoring allows one to set critical thresholds on which once they have been reached, proactive procedures are set in motion in order to set a given configuration item back into order. Workload, capacity and availability management & planning become the main driver of the organization's decisions. Technology becomes centralized and standardized to make sure that everyone adheres to the same set of chosen tools, procedures and equipment.

Finally, the last two levels are, respectively, the service level and the value level. An organization reaches these levels when it starts managing its infrastructure in terms of the services it provides and in terms of the added value to its core business. Service level agreements that act as a contract between the IT and its clients are defined and introduced. They provide an estimate of how long a given necessity will take to be fulfilled as well as key performance indexes and goals for the teams answering for it. This is only possible because of the various standardized processes that are now embedded in the organization's culture. Also, it becomes possible to track which infrastructure supports a given service and which services support a given business. Thus, the implementation of effective usage charge back across the organization becomes available leading to a better cost planning. On this stage, the organization main focus is the continuous improvement of the delivered services and way of doing business. It is important to mention that an organization might take between five years to a decade to reach such a stage, depending on how much it invests on this field and on how low its current maturity levels are.

1.4 Early Stages

Back in 2010, ZON, one of the companies that merged into NOS, was beginning to take the very first steps away from its chaotic maturity levels. Prior to it, the IT infrastructure management was done completely by an external entity. As infrastructure became more and more critical to the core operation of its business, the decision of bringing its infrastructure management and operation, back in-house was made. This transition had its very own challenges such as, the non existent documentation of the processes and the refusal to share precious know-how. The company itself had already adhered to the ITIL framework. Many of ITIL procedures were already implemented in its

different departments, while the data center was giving the first steps to join them. In the market, there were various tools available to support ITIL processes and procedures, such as BMC Remedy, HP Service Manager (HPSM) and Easy Vista. Each one of them with its different flavours of ITIL, different pricing, different challenges to implement. In our case, ZON had acquired and implemented HP Service Manager, with all its underlying modules for service requests, change management, incident management, problem management and CMDB. Consequently, the new infrastructure management team supported the processes which they are introducing on this mature tool.

The HPSM CMDB was gradually populated with virtual servers, blades, blade centers, storages, backups, switches, firewalls, etc. Tools like HPSM need to be generic enough, in order to allow their implementation in any type of organization domain. But in reality, virtual servers have different attributes than blade servers. Blade servers have different attributes than blade centers. In fact, blade servers live within blade centers. Thus, we have the issue of, depending on the type of configuration items, different sets of attributes may be required. If there are so many differences between attributes of distinct types within a same department, these differences become even more noticeable, when considering types belonging to separate departments. Does the chosen tool allow for different classes? Or does it allow for customization? How hard is it to manually manage the classes and its relationships? How hard is it to create new custom attributes? Thus, instead of modelling a tool to answer the needs of our problems, ZON was left modelling its solutions problems in ways which fit the tool. The end result was that the configuration items were created with general attributes filled with whatever made sense in that case, and the rest of the important details were simply populated into an open description text field in the configuration item.

Another interesting limitation of these tools worth mentioning, is related to how they implement the service request fulfilment. For the very same reasons mentioned above, the content of a service request, requesting the deploy of a new virtual machine, is completely different of the content of a service request, requesting for a new laptop installation to be delivered to an employee. Does the tool allow for the creation of different forms in accordance to each service request? Or does it only allow to write a generic description of what a user requests? Can these forms be detailed enough to fit our needs? Can they be internally created and altered in a timely fashion? Can they trigger actions in other tools automatically so automation becomes possible? The service request fulfilment in HPSM was, also, only a box of text where the requester could describe what he was looking for. That became a problem when one wants to standardize their processes, and make sure that all the necessary information is present when a service request is created. Back then, ZON, who was trying to implement some sort of standardization, was using excel templates per requests, which were distributed on demand to its requesters to be filled and then attached to the service request. Whenever there was a change to that excel template, there was no way to notify its clients that it had change and that they needed to download a new version.

Thus, it became clear that as the organization grew in size, so did it grew in complexity of its underlying domain. It was no longer possible to populate meaningful information, nor to support completely the intricacies of its micro processes, in just one central tool made to fit all. The main

concept behind the in-house creation of the data center portal was to develop a system that serves as the main support to all the management and operational procedures of the data center department. It had to become the main entry point for its external clients and internal teams. The development itself had to follow agile and flexible methods, since the business logic to be implemented, was very domain specific and very personalized. And, all of this had to be achieved while abiding to a very important requirement: that there was only to be one official ITIL tool within the organization, the HPSM. The term System Federation will be used to describe a centralized ecosystem, where there are many surrounding systems, which have their own functions and details, but they all must feed their activities back into the centralized system. The centralized system may not be capable of receiving these feeds directly, but it must be possible to translate these feeds into its domain. These translations may imply the loss of quality or the downgrade of the information, but that is a no problem, as long as, the surrounding systems domain are merely extensions of the centralized system domain.

Following the concept of a federation of systems, we can designate the whatever is the official ITIL tool on an organization as the centralized system, and we can designate all the systems that try to extend on it, the surrounding systems. By ensuring that, no matter what specific domains and procedures are implemented on the surrounding systems, they must be translated back into the centralized one, we guarantee that we are not ditching away all the benefits of adhering to the ITIL framework. If a procedure can be partly translated back into the ITIL tool, it means that we are only expanding on the ITIL universe. Thus, the concepts behind the developing of the data center portal, are to merely expand on the centralized ITIL tool universe, and never to replace it. All the information managed by this portal had to be capable of being translated and mirrored back into the HSPM. After all, HPSM will be the source for any compliance evaluation, ISO certification and official reporting. With that in mind, the first challenge given to this portal was to systematize and standardize the reception of requests. In order to reach higher in organizational maturity, it was important to get rid of the ad-hoc process of submitting a request. Broadcasting excel templates on demand to clients was all but, elegant and efficient, since it led to all sort of unnecessary interactions, such as the ping pong validations from the client to the delivering teams.

1.5 Data Center Portal

Therefore, developing a data center portal, with the different services available to all the clients, was the very first step to help in the urge to reach new levels of maturity. Each service having its own specific set of form, validations and service level agreements. After creating a given request, the client was to be guided on how to access the HPSM tool, in order to follow the official service request fulfilment process, where he will then simply post the ID of the newly created and validated request from data center portal, instead of the open text description or the excel template filled by him. Later on, when the corresponding implementing team notices the newly generated SR ticket in HPSM assigned to them, they were to access the organized and standardized information of the request by simply downloading it through the Data Center (DC) portal. This new described process

was a prototype, whose purpose was to test waters on how would the involved teams react to the creation of such a tool: both in terms of internal teams and external teams. This process was kept simple, as it tried to partly solve the validations issues that haunted the data center teams and its clients. And, as a trade off for that simplicity, it introduced the extra step of requiring the usage of two tools in order to officially submit a request to our teams.

This led to mixed feelings and feedback regarding the experience. Internal teams were happy with the fact that they were now receiving standardized requests on which, the structure they had full control upon. They could easily propose new services, forms, validations, or change the existent ones by interacting with the development team. The DC delivery team was now capable of redirecting its clients to the right services forms, avoiding for most of the cases, the creation of requests through the usage of open text fields. On the other hand, requesting teams whom were used to submit their requests in whatever way they deemed fit, were now greatly constrained by the forms and their validation rules. Albeit, some of them actually welcomed the introduced rigid forms and validations rules, since they guided them on how correctly submit requests with the corresponding available values. Nevertheless, there was a vanguard of users that saw this new way of working as a great obstacle on the way they were used to work. If before they opened a service request in whatever way they wanted, being this request iterated by the data center team until its content became valid, now they had to submit them through the defined catalogue or have them rejected. It took some months of internal politics and tutoring for the data center portal to become widely used and accepted as the official entry point to the data center.

The Portal catalogue was first released with twenty one different services. They covered the necessities of the teams who manage the servers, who operate the operating systems, who operate the backups, who defined and configure Demilitarized Zone (DMZ)[4] in the network, and who deal with the overall processes in data center. Two months later, the number of available services had double to now include the necessities of the teams who deal with the connectivities universe. The connectivities universe included a wide range of services, such as firewall services, VPN related services, tunnelling services and dynamic network load balancing services. Throughout the first year, the varied services and forms were revisited and improved. The prototype was iterated and matured in such a way to support better modifiability and flexibility, in order to be able to answer the constant flow of new features and services. Later on, the needs of the database administration teams were also mirrored in the available catalogue, with the introduction of sixteen new services in the catalogue.

1.6 AD Integration & ACL Engine

There was no restriction on whom had access to the service catalogue of the data center portal. As long as the user was able to reach the server, he was able to create new service requests. Although the portal was inside a DMZ network that blocked unwanted accesses from outside networks, it was also published in the organization corporate proxy. This meant that anyone with that proxy configured in their browser was able to have unrestricted access to the portal. And, not only the user had

unrestricted access through the proxy, but he also had a fake identity, since the portal web server was only able to see the proxy identity, and not the source user identity. If a user generated a service request and accidentally closed his page before copying the id, or if a user generated a service request and then decided not to map it in the HPSM, there would be no obvious way of recovering its content back to the user. So, in order to both create a sense of ownership and a sense of security, it became of the utter most importance to authenticate and authorize its users. In order to avoid the creation of yet another authentication mushroom, an integration with the organization's Active Directory (AD) was followed and implemented.

The Active Directory [5] developed by Microsoft is a structured hierarchy of objects. Objects have their set of attributes, and they represent entities such as users, groups, etc. Since objects are organized in an hierarchy manner, objects such as groups can contain other objects. Usually, the objects held within an AD can be grouped into Organizational Units (OUs). Organizational Units allow for one to resemble the organization real structure. If we set NOS as an example, NOS was created from the fusion between two companies which had their own set of distinct users. There was a point in time where the AD had two distinct OUs, Zon and Optimus OUs. Also, within each object, it is possible to define a set of attributes. It is possible to define user policies regarding to the password renewal process. It is also possible to authenticate against an AD, since it can check a given password against the one set on the user object attributes. The Lightweight Directory Access Protocol (LDAP)[6] is used to perform queries against directories such as the ones from the kind of an active directory. LDAP allows for an authorized user to search the hierarchy of objects in order to find the object which matches the given query. Queries can be a combination of where in the tree an object stands, and what its attributes look like. Binding is another important operation in the protocol. It allows for an LDAP client to authenticate itself as a given user object from the AD. If a binding fails, it means that the given username and password did not match. Thus, it is a way of authenticate against an Active Directory.

Next to authentication, there is authorization. Authorization allows for one to define what is a given authenticated user authorized to do. An effective way of tackling this challenge is through maintaining an Access Control List (ACL)[7]. In general terms, an ACL is a list of permissions of a given object. It defines what operations, can and cannot a user or a group perform within a specific object. Defining an ACL at the user level is the most fine grained control we can get when specifying the subject for the operation. Dealing with ACLs at such level bring a huge burden when it comes to maintain the ACL, thus it is usually ideal to perform the specification on the group level. In terms of what operations can be performed in what object, it is also important to weight its granularity. The more granular we are, the better we can control the access to an object, but the harder it becomes to maintain it. To find a balance between these two is important for both security and usability. The DC portal ACL engine kept this balance in mind, thus it was implemented in such a way that the access was controlled on a group level, while the object operations were coarse enough to allow for their auto discovery.

Finally, we need to have mechanisms in place to be able to empower the Auditing of these accesses. These mechanisms must be able to log every event generated during the authentication and

authorization phases. During the authentication phase, it is important to keep track of every successful or failed login attempt along side with its details. Such as the time stamp of when it happened, which user was tried to authenticate, the cause of failure, the source of this attempt, etc. During the authorization phase, it is important to log what operation in an object was attempted by a given a user. Besides the details mentioned above, we may also include if the access was granted or denied, as well as, the time stamp of when the decision took place. The mechanisms of Authentication, Authorization and Auditing brought an end to the era of uncontrolled access. In order to avoid disrupting abruptly what the data center portal users were used to, the access to all its components was still left open, by the means of mapping its users into a global group with access to everything. Along side with it, it was finally possible to give a sense of ownership of requests to its users.

1.7 Reporting

To keep in check whether or not the Service Level Agreements and the Key Performance Indexes were being followed, a reporting standalone module was developed. Self improvement of processes and systems is unlikely if there are no mechanisms in place that quantify and measure where one is standing at a given moment. The implemented reporting module took advantage of the federation strategy introduced before. Since all important data about the requests was also reflected in the HPSM, the reporting module attacked the HPSM official data warehouse and worked on top of its raw data. The raw data consumed was related to incidents tickets, service requests tickets, change management tickets, problem management tickets and configuration items under the data center management. An augmented table with the business indicators was created for each one of the entities mentioned before. Examples of these business indicators were if a given ticket had fulfilled or failed an SLA, how many linuxes were under the data center management, how many configurations items were affected by a given ticket, etc. These augmented tables were then updated of their differences with the source on a regular basis through a scheduled routine which ran every day on the dead hours to avoid stressing the involved systems during the active hours.

1.8 Workflow Engine

Undesirably, all portal users still had to interact with two tools in order to place a request or to check on their status. An improvement in this process was due, if we were to bring the whole request fulfilment process into the portal. In order to achieve this goal, two major components were amiss, the Workflow Engine and the automatic Integration with HPSM. The Workflow Engine job was to provide ways for one to define workflows that bring automation to our manual procedures. It is possible to define the different phases of a procedure in terms of a State Machine[8]. A State Machine is a set of states and their underlying transitions. These transitions are usually represented by conditions and results, meaning that, if the condition clause of a transition is met, we can transit into the target state while producing a given result. If we place some semantics upon the definition of results, we can implement them as being the action which takes place when a given condition is met, in order

Transition	Condition	Action
C1—A1	User belongs to the group whose the ticket was submitted to	User assigns the ticket to himself
C2—A2	User is the ticket assigned user	User assigns the ticket to another team
C3—A3	User belongs to the group whose the ticket was reassigned to	User assigns the ticket to himself
C4—A4	User is the ticket assigned user	User rejects the submitted ticket
C5—A5	User is the one who rejected the ticket	User resumes the execution of the ticket
C6—A6	User is the assigned user	User completes the implementation of the ticket
C7—A7	Automatic routine after 2 days of inactivity on the ticket	Routine closes the ticket by setting it as completed

Table 1.1: Description of the HPSM request fulfilment transitions

to reach a new state. Thus, defining a procedure becomes nothing more than specifying its several possible states, available actions and conditions.

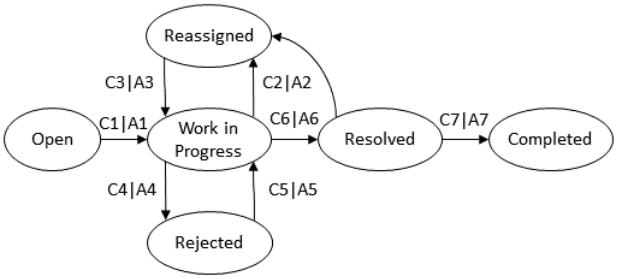

Figure 1.2: HPSM Request Fulfilment State Machine

For instance, the request fulfilment procedure which was followed in HPSM, can be defined as the state machine represented in figure 1.2. It is composed by six distinct states: Open, Work in Progress, Reassigned, Rejected, Resolved and Completed. We can transit from Open into Work in Progress if we meet condition C1 by executing the action A1. In this case, C1 defines that the logged HPSM user must belong to the group from which the given service request ticket was assigned to. A1 represents the action of one user assigning the ticket to himself. The result state of following this particular transition is that the state of the given service request ticket becomes Work In Progress. The detailed description of the several transitions available in the presented workflow can be found in table 1.1. As demonstrated above, the state machine model is a flexible and valid way of describing the complexities behind a workflow. Through the conditions we can make sure that actions only become available when the prerequisites have been met. This prerequisites can be defined as anything, they can be group policies, they can be user validations, they can be entity validations, etc. On the other hand, actions encapsulate behaviour. When transiting from a state to another, it is because something changed. Actions give life to those changes, they implement them, no matter what their scope are. It is up to the workflow designer to decide and implement its own custom behaviour.

Another important characteristic of the Workflow Engine is to keep track of all the workflow past. It must be possible to go back in time and understand when and which transitions happened. Besides the usefulness of having an history of transitions available for general purposes, it also allows for one to build reporting strategies on its raw data. Very much in the likeness of what was done with the reporting collected directly from HPSM. Also, to empower the Workflow Engine with the capability of understanding ownership concepts brings great added value to the engine. Since workflows can be used to define the life cycle of entities, such as what happened in the HPSM request fulfilment, it becomes interesting to imbue it with the means to tell: who performed a given action, to which group the entity was assigned, to which group the entity will be assigned, whom currently has the ownership of the entity. This way, conditions based on the ownership become innate to the engine and do not required extra logic programming in their definition. It also helps the workflow designer to model his workflows having in mind ownership. A given state can only be achieved, if a given action is executed by the group that is suppose to own the process in that specific moment. When an action is executed, as result, the target group of the workflow may change to a new one.

1.9 Invoker Manager

The other major component that was amiss, in order to centralize everything in one tool, was the HPSM automatic integration. Instead of forcing a requester to manually create a service request in HPSM with the reference id of the data center portal, it was important to have this done transparently. Instead of forcing our operators to manually map the Change Management in Service Request Fulfilment, or managing the underlying relationships between configuration items and these processes, it become important to also automatize this bit. If we consider that for every service request created, we have to map at least one configuration item. And that for every service request passed into the 'Work In Progress' state, we have to create a corresponding change, we have to create a relationship between the change and the service request, we have to create a relationship between the change and the configuration items, and etcetera. These manual interactions can easily bulk up to more than 1000 weekly actions that bring no other added value other than ITIL compliance. Thus, automatizing all these interactions, not only helped us step even further in the maturity level, but it also catalysed for significant gains in the efficiency of the daily routines since the bureaucracy burden taken from all teams was extremely high. Also, it allowed for the request fulfilment process to be refined in order to add better clarity on it. In HPSM, it was not clearly what phases if any, a request needed to pass in order to be completed. By migrating all these procedures into the data center portal, we were able to implement our very own phases with our clients need for clarity in mind.

To guarantee that all interactions got mirrored in HPSM, these integrations took advantage of the Workflow Engine mentioned before. After the new request fulfilment workflow was designed, a way to map its state machine into the one in HPSM was drafted. It involved calling the available HPSM web services on each equivalent action. Therefore, if in the DC portal, there was an action where the request finally became assigned to an implementing team, the action would have to additionally

call the corresponding set of HPSM web services that mirrored the states. Using this strategy, it became possible to map distinct but yet somehow similar workflows to best of the efforts. Despite of, the portal service request fulfilment having the responsibility of knowing how to map itself in HPSM through web services calls, it was not its responsibility to invoke them. This responsibility was passed down into yet another component that we will call of Invoker Manager. This component performed the role of a Message Queue[9]. The role of a Message Queue is to provide a way for asynchronous communications protocols. Asynchronous communications are characterized by the fact that applications sending messages, do not need to lock themselves and wait for responses. They can continue processing whatever work they have left, while providing a callback method which will be called once a response for that given request arrives.

The Invoker Manager was developed to function in similar ways. Its job was to receive an object that encapsulated an web service call (invocation) and perform it himself. Once, it received the response, it was then responsible for execute whatever was defined in the invocation callback. This way, the data center portal was able to proceed serving its clients, while in the background, the Invoker Manager made sure that all requested integrations by the portal were being invoked. The Invoker Manager was also responsible for dealing with orders and faults. The submitted invocations carried an unique group id which was used to tell that, within the group of all the invocations which carried that group id, order of arrival was to be respected. No invocation can be invoked before the previous one within its group has succeeded. At the same time, any invocation belonging to a different group id, may be invoked in parallel since they do not overlap one with another. This property is of extremely usefulness especially when dealing with state machines models such as the ones implemented in HPSM. As we saw before, certain actions are only available in given states, thus, we must keep track of the order on which we call these actions.

In the event of a remote service being invoked becoming unavailable, or in the event of a remote service malfunctioning and returning unusual errors, we must ensure that the invocation is periodically retried until it is successful invoked or intentionally dropped. Thus, the Invoker Manager must be able to evaluate the received responses and according to their values, enqueue the invocation again for retry or call the corresponding callbacks for it was successful. This type of events may sound a bit unlikely to happen, but they do happen, more often than rarely. It is easy to come with simple scenarios where even though HPSM is functioning correctly, it might return error messages to its web services invocations. Lets take as an example the scenario where all the licensed sessions are currently being used in HPSM, if the Invoker Manager was to trigger an invocation on that interval, it will receive a response error reporting that the maximum number of sessions were exceeded. Lets say that the Invoker Manager wanted to invoke an update on a given HPSM service request, and that the given HPSM service request was currently locked by another user, the result of that invocation will be yet another error stating such. Therefore, it is of the utmost importance for the Invoker Manager to be well prepared to deal with this type of events.

Integrating with HPSM improved many of the internal and external processes, but there was a little detail that it overlooked. The internal teams still had to interact with HPSM to perform activities

related to the task management. Let the task management be the process where the actual work is dispatched to the varied teams. This means that a given service request created in the data center portal will have to be sliced in one or more tasks for the different implementing teams. On that point in time, the HPSM task module was being used to achieve such. And just like any other module in HPSM, it suffer from the vary same efficiency issues. This led to the adoption of yet another tool that facilitated this process by the operational teams. Since the effort to develop such a module inside the data center portal did not make sense on that particular moment, the Commercial Off-The-Shelf (COTS) of choice became Easy Vista (EV)[10]. There was a period where the data center delivery team was manually dispatching every new portal request into Easy Vista requests, requests which were then further split into tasks. Therefore, in order to improve the process by eliminating this manual creation process, an integration with Easy Vista was also sought and implemented taking advantage of the mechanisms created to integrate with HPSM.

1.10 Data Center CMDB

In 2013, ZON and Optimus were merged into a new company that temporarily went by the name of ZON Optimus. During that period only minor improvements which did not affect the software architecture of the data center portal were made. In the likeness of ZON, Optimus also had its own data center infrastructure, teams, tools and processes. For about an year, those two teams kept working separately, on their own old way and infrastructure. Meanwhile, the processes of both and possible synergies between each other were being studied. Optimus as an organization also based its processes in the ITIL framework. Unlikely ZON, Optimus supported all of its ITIL processes on top of an old version of the BMC Remedy tool[11]. The main entity of this tool was the Trouble Tickets, and they can be seen as the equivalent of incidents tickets in HPSM. These trouble tickets were also being used as service request tickets. Additionally to Remedy, there was another tool that mimicked many of the features found in an ITIL tool, named Infranet. This tool had been developed in a custom manner to fit Optimus IT necessities by an outsource company. It included modules for incident management, service request fulfilment, task management and CMDB. These modules were partly integrated with Remedy, since corresponding TT were created but not updated by the tool. Teams used both tools seamless, depending on whether the source of a TT had been Remedy or Infranet. There were no defined service level agreements and no defined key performance indexes nor reporting being collected.

Later in 2014, NOS the new official brand for Zon Optimus was publicly announced. Internally, the distinction between ZON and Optimus teams, was supposed to cease to exist. This meant that a new strategy was in need to finally merge both realities into a single one. Until that moment, the data center portal had been adapted to allow requests incoming from the Optimus reality, being such requests then manually dispatched to the corresponding teams. The main strategy was maintained, it involved having the data center portal as the entry for everything. While having the data center portal mapping its information onto the relevant systems. It became known to all the areas that a

new ITIL tool, different from HPSM and Remedy, will be officially implemented. Thus, the data center portal was to maintain its integration with the HPSM while ignoring any integration with Remedy, until the appearance of the new tool. This represented a problem itself, since HPSM CMDB had configuration items related to ZON universe, while Infranet CMDB had configuration items related to Optimus universe. Also, the data center portal was validating itself upon the HPSM CMDB, and if we were to simply import Infranet CMDB upon HPSM CMDB, a great amount of relevant information would get lost in the process. This was mainly because that Infranet CMDB was very specific to the data center universe, where HPSM CMDB was way too generic to any universe.

In reality, the Infranet CMDB schema had been developed internally by the ex-Optimus team. Infranet was simply a front-end that fed on top of that information, pretty much in the image of how the DC portal fed on top of HPSM CMDB. We decided to take ownership of this Infranet CMDB and through the methods of federation mentioned previously, to federalize whatever information was possible back into HPSM CMDB. This way complying with the ITIL procedures remained possible while both universe became merged seamless and in a controlled way. To make this possible, the schema of the Infranet CMDB had to be updated to include attributes that were not intersected when comparing to the HPSM CMDB model. Then, an unified configuration item front-end was developed within the data center portal. Thus, the Infranet CMDB became a key new component within the data center portal architecture. Whenever a new configuration item was created or changed through the newly developed front-end, the equivalent invocations were sent to HPSM through web services, in order to maintain both universes consistent. This meant that the DC portal became the only component allowed to perform changes within the items of this Infranet CMDB.

Additionally, this Infranet CMDB was also augmented on its schema of the attributes necessary to introduce the concept of ownership of a configuration item. This was of extreme importance since configuration items started having a workflow life cycle themselves. Until this point, there was no notion of state within a configuration item, they either existed or did not. When in reality, there are many phases through which a configuration item can pass and different actions that can be executed depending on which phase it is. So, we had to expand the influence of the Workflow Engine in order to reach this new type of entity within the Infranet CMDB. From this point on, we will start to referring to this Infranet CMDB as simply being the Data Center CMDB. It became possible to easily track through the DC portal which servers one owned as a requester. Which operating systems they had, what were they specifications, what were their management Internet Protocols (IPs), what were their service IPs, how much storage was deployed on that host, etc. It became also possible to dive into the history of a given server and visualize which portal requests affected it directly. This type of functionalities greatly improved the user experience of both internal and external teams to the data center. After all, a data center operator troubleshooting an incident had now access to very detailed information of a given configuration item without having to access it directly. And by consulting its history, one can try to figure out if there is any relation between the incident and its most recent past of requests.

1.11 Task Management Module

Easy Vista licensing model was revealing itself too expensive, especially when considering that within the new NOS universe, we needed to double the access allowance licenses. If investing on the in-house development of a task management module did not make much sense before, with the definitive merge of companies it was now a main priority. It was important to normalize the way internal teams manage their work and while at it, to reduce the costs derived from the tools supporting that process. Former Optimus team members were still receiving their work in the previous defined ways except for the requests that followed the new standard processes. Those type of requests were arriving to those teams through email. That method of dispatching was not without its very own problems: they easily got lost in the heap of mails, there was no way of measuring their times, it was hard to understand the contents of the request. A possible solution to this problem was to bring these new teams under Easy Vista too, but as mentioned before, the costs of doing such were not justified. Plus, by developing this module in-house, it was possible in a flexible and agile way, to improve the whole process of how people receive, dispatch and perceive work. As processes grew more complex, perceiving the work that needs to get done was no longer a trivial matter.

As the new requests arrived in an orderly manner, they had to have passed through some phases of approval before they reached the point where they start getting implemented. It was when they reached such phase, that the corresponding tasks were then created. By corresponding tasks, we mean that this new system had to map templates of tasks into services. So, a given service had its own set of tasks, with their title, description, Operational Level Agreement (OLA), delivery group, target group and attributes. Around the moment that the system was going to instantiate these tasks from their template, it had to define what extra time to add to their OLA due to the number of configuration items affected on that concrete request. If the request affected many configuration items, the extra time was to be partitioned between all the tasks. A dashboard with pending work was then required, in order for the teams affected by those tasks to be able to identify easily the pending work. Then, when they concluded, cancelled, paused, resumed their tasks, it was important for the corresponding request to show up in the dashboard related to requests, as pending an evaluation of progress. This was mainly because delivery teams needed a mechanism that enabled them to see requests where further actions were required from their part. Take for instance the scenario where a given task was completed within a given request with multiple tasks, the delivery team needed to be able to know that a task had been completed and that the following tasks were pending a submission by their part. There was also the cases where a request had more priority over the others, even though it had been chronologically requested after them. The system needed to allow for one to mark it as urgent, and to change its target date while updating the target date of its corresponding tasks.

It also needed to empower the cases where a given request or a given task was unable to carry on because some necessary details were still missing. In these cases, where the requester needed to be enquired, it was important to update all the SLA, OLA and estimated dates to take into account all the time spent in external teams. A new estimate was to be calculated based on it, and reported

back to the original client. Thus, perceiving the existent work was no longer a matter of managing a First In First Out (FIFO) queue. There were lots of variables and micro processes that needed to be weighted before making the decision of what was to be implemented next. Therefore, it was important to buff the system with the capability of providing all this information in an easy to read way, in order to make the decision making process straight forward. The engine that empowered the DC portal was by then mature enough to easily implement all these kind of functionalities. Thus, removing Easy Vista, not only became a matter of reducing costs but also a matter of simplifying the architecture and computational resources. Since, it was no longer needed to have a connector that implemented the integration with the EV, as well as, the computational resources upon which the EV ran.

1.12 Thesis Outline

Within the following sections, we will expand on the architectural choices and rationale behind the conception of the Data Center Portal. We will also expand on many of the design options that were made when implementing the components defined by the architecture. The order on which we will approach the different phases through which the development passed, will be similar to the one used in the overview section presented before. For each of the phases presented there, we will first propose to introduce the architectural rationale and sought quality attributes, followed by the definition of the quality scenarios. Then, we will detail on how that given system was designed an implemented in terms of both architecture and design decisions. Afterwards, we shall measure and evaluate in whether or not the sought quality attributes were met. At the end of every iteration, we will discuss the decisions that were made and their impacts in the architecture as whole. We will expand on the mistakes that were made and on the possible points of improvement. Since each iteration touches a small subset of requirements that were sought on that given phase of the project, each iteration will slowly build up into a much larger architecture capable of giving answer to all of its requirements.

2

Data Center Catalogue

Contents

In this section the beginning phase of the Data Center Portal will be detailed. We will describe the birth of the Portal DC component whose main purpose is to be the main door for everything. As we saw before, chronologically, at its very beginning, the DC Portal was merely a catalogue where an arbitrary requester was capable of checking the available services and consume them. The challenges of developing such a system may come across as lacking at first glance. But as we will see, key architecture qualities had to be set at this stage, in order to allow increasing the system complexity in the future as predicted.

2.1 Rationale

In all honesty, there was not much architectural planning put into paper when the DC Portal first began to be developed. This was due to the fact that on that stage, it was thought of as a prototype whose purpose was to test grounds on whether or not people would see it with good eyes. Thus, the project started on the wrong track since it was the software designing that led the software architecture and not the inverse, as the best practices recommend. That was partly because the person behind defining the architecture and designing the software was the same, thus the line dividing both responsibilities became blended. Obviously, without much surprise the Data Center Portal was set to be an Web Application which needed to be available within the organization Intranet. Web applications are usually best represented in Client and Server models where we have one or more servers serving an universe of clients. With that in mind, we have then several specializations of this model such as the N-Tier Client Server (CS) architecture[12]. In a N-Tier Client Server, the N defines the number of existent server layers. For example, in an 1-Tier CS there is only one layer of servers which we can denominate as Front-Ends. Within a 2-Tier CS there are two layers of servers: Front-Ends (FEs) and Back-Ends (BEs). These layers allow an architecture to segregate responsibilities of server components at the cost of a bigger round trips response times.

Although, no initial security qualities were sought first, at the applicational level in this stage, the same cannot be said on a component level. This means that, even though the web application was to be available to everyone with no restrictions or control, the underlying architecture had follow the well defined 2-Tier Client Server segregation. A client was meant to be capable of only accessing the front-ends, while the back-ends were meant to be only accessible to the front-ends. This segregation made sure that no other system had direct access to where our important data was going to get persisted. Thus, inter system security was one the first qualities that was natively sought in the conception of the DC Portal. The defined network architecture made sure that the required DMZ was in place in order to avoid any unwanted access to sensitive components. Plus, in order to comply with the network architecture policies of the organization, all the connections of anywhere into our 2-Tier CS architecture were blocked by default. That meant that even clients were blocked of accessing the front-ends by default unless they came from the intranet network.

Another quality attribute that was important to be in place in the early stages of this project was deployability. The deployability quality seeks ease of installation and updating in a system. Due to

the agile nature of this project, features, changes and fixes needed to be deployed into production rather regularly. It was important to have a structure that supported performing such deployments in a transparent and rapid manner. In order to achieve that, the architecture was not composed of just one 2-Tier CS components, but of two 2-Tier CS components. One of these 2-Tier CS component belonged to the production environment of the web application, the other belonged to the staging environment of the web application. The production environment of our web application was only capable of interacting with other production environment within the company, while the staging environment was only capable of interacting with other staging or testing environment in the company. Then, taking advantage of a centralized code repository; features, changes and fixes were to be first integrated in the staging environment. After they had been successfully deployed in the staging environment, it meant that were ready to be pushed into production, since the staging environment was very similar in nature to the production environment.

As more services were brought into the portal catalogue, and changes to previously implemented services were requested, or changes to characteristics that affected the services as whole, a new set of software qualities had to be introduced into the architecture. The system was in dire need of Extensibility and Modifiability. Extensibility and Modifiability can be defined as the ability to easily add or change new features and customizations. At that moment, each new service implied developing its underlying form steps, logic, validation and views. A service was divided in many steps, each step had its own form logic. Fields filled in one step may affect fields belonging to past or new steps. Thus, removing a step had the side effect of requiring one to readjust the boilerplate code related to the service as a whole. Also, adding a new cross-service feature implied to perform code changes on every defined service. If there were fourty distinct services, it implied performing the same change for each one of them. This happened because there was no centralized form framework that was shared among each service in order to cross cut these features. So, as the number of services grew in size, introducing extensibility and modifiability into our architecture by refactoring the existent code base to comply became crucial to the development of the project.

The Model View Controller (MVC) design pattern[13] was chosen as the standard skeleton to follow in the development of the web application. The separation in these three distinct layers of responsibilities still make sense in many of the web applications developed nowadays, and they provide a common ground of knowledge in how to extend and modify a system. In the MVC, the Controllers are responsible for fetching and manipulating data from the Models, and to deliver that data to the views. This way, a web application may have many views from the same set of data, which in our case made sense, since we wanted to present the content of our requests as HyperText Markup Language (HTML), Excel and JavaScript Object Notation (JSON) views. An advantage of designing an web application adhering to MVC is that the pattern simply defines rules and sets general logical barriers. It is then possible to extend the MVC reality to cope with the required complexity of the business logic. Thus, MVC provides a common language to the development teams on how to tackle a problem. This means that the details of a concrete MVC implementation are then left to the developing team. If whether a team decides to implement their Model layer using a specific Object-

Relational Mapping (ORM)[14], whether it decides to implement it through the means of Structured Query Language (SQL) connectors using result sets, whether it decides to simply use the file system, it all boils down to a design decision left open to the development team. Therefore, as a recipe to enact modifiability and extensibility qualities while adhering to best practices, the MVC was adopted in the conception of this portal.

The agile development of the portal where the environment changed more than once with many daily updates was not without its own setbacks. It was not possible to test thoroughly all the incoming features and changes resulting in the introduction of a stream of issues. Let supportability be the architecture quality where an system maintainer can easily identify faults and analyse their root cause through restricted information. We sought to introduce this quality in our software architecture by requiring its underlying components to introduce mechanisms which allow for the system to graciously fail, by informing its affected users that something went wrong, while at the same time registering the error dumps in order to facilitate the problem troubleshooting process performed by the responsible teams. This information was to be accessed in a privileged view within the corresponding component and used to tackle down the issue in a proactive manner, ensuring that problems were fixed automatically without pushing the issue reporting responsibility into this the portal users.

Usability, the quality of empowering the system with means for the user to easily execute its operations was also of the utmost importance to be present in the architecture. After all, at this stage the DC portal was testing waters, or by other words, trying to fall into its users grace. If we take the firewall catalogue as an example, there were requests where a client may want to open more than a thousand of automatically generated firewall flow rules. If the architecture had not been conceived with a mechanism that allowed for the uploading and the parsing of bulks of information, there was no way that users requesting this type of request would stick around. At first glance, this type of feature might look like it does not impact the software architecture, but in reality it does. When we moved from the scenario where the clients always had to fill requests through the web application interface, to the scenario where the clients were able to upload a Comma Separated Values (CSV) file that would then fill the request in accordance, we had to move from in-memory sessions into database persisted sessions. This was mainly because from that point on, users were capable of generating requests so large, that the objects needed to represent them were capable of depleting all the server memory if kept solely in it.

Inside the usability family of qualities, we have the understandability quality. It defines that the users must be able to use the system with very little training. Thus the architecture has to accommodate means to provide intuitive interfaces capable of being self explanatory. Having multi-step forms instead a one-shot large one representing each service is an example of what was sought. The components of the architecture had to allow for gradually submitting information related to forms, remembering the progress done so far, and facilitate navigation between each possible step. It had to support the definition of rules that provide dynamic fields. Dynamic fields are shown or hidden on the basis of whether or not, the fields on which they depend upon have been filled in a specific way. To implement this type of functionality in order to achieve understandability, there is also an impact

on how the architecture must be planned. It sets restrains on what the server must hold in the user session space. If a single-step forms approach had been followed, servers only needed to validate the form data received, process it, and discard it. But instead, when following a multi-step approach, the servers must validate the form data, process it, and save it in session until the request is definitively submitted as a whole. Independently of what type of data store is picked to hold the session (in-memory, files, database, ...), the architectural decision of keeping form data in user session, directly impacts the components resource usage.

2.2 Quality Attribute Scenarios

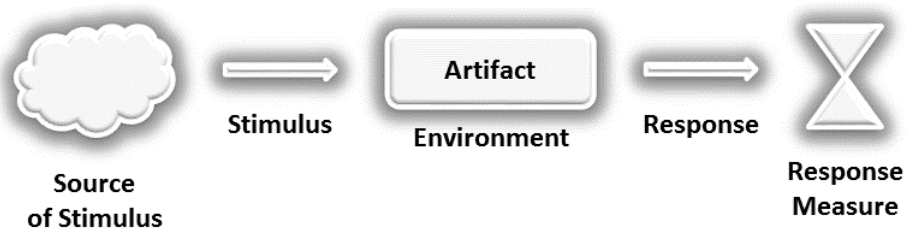

Figure 2.1: Components of quality attribute scenarios

We shall now define a set of quality attribute scenarios[15] that are specific to our architecture. These set of quality attribute scenarios will define the set of requirements that must be respected by the quality attributes mentioned on the previous section. Quality attribute scenarios are a tuple containing the source of a stimulus, the stimulus, the environment, the artifact, the response and the response measure (fig. 2.1). The source of a stimulus is the entity which created a stimulus. A stimulus is the event that arrives to the scenario system. The environment is the specific condition of on where the system stands at the arrival of the stimulus. The artifact is which part of the system is being stimulated, it can be the whole system or just part of it. The response is the action that gets dispatch as result to a stimulus. And the response measure is the measure that will be used in order to test the scenario requirement. Depending on the quality attribute on which we are building the scenario for, the possible values for each component may differ. For example, in a modifiability general scenario, the source of a stimulus may be a developer while in a security general scenario, the source of a stimulus may be an unauthorized system. A concrete quality attribute scenario can then be put together by defining this pieces with concrete values that are valid for a given quality attribute.

On the previous section, we have defined as the main quality attributes present at the very early stages of this architecture: security, modifiability & extensibility, deployability, supportability and usability & understandability. In general terms, the rationale behind them was to have an architecture capable of implementing and deploying new features within the time to market requirements, while setting the back bone for future security requirements, and setting standards to appeal to its future

Scenario	Source	Stimulus	Artifact	Environment	Response	Measure
Sec-1	Internet	tries to access the available services and data	System	DMZ	The access is refused	Probability of an attack that exposes the services to the Internet is very low since it requires reconfiguring all the organizations firewalls
Sec-2	User outside the Intranet	attempts to access the available services and data	System	Production DMZ	The access is refuse since the flow originates from the outside of the organization	Firewall correctly identifies the source of the flow and logs the attempt
Sec-3	Intranet user	attempts to access the available services and data	System	QA/Test DMZ	The access is refused	Firewall correctly refuses access to the QA/Test environment for normal users
Sec-4	Developer Operator	attempts to access the internet to download any type of content	System	DMZ	The access is blocked	Firewall correctly blocks any attempt to access the exterior from within the systems
Sec-5	Unauthorized External System	tries to access the available services and data	System	DMZ	The ports are blocked and therefore any connection is refused	Firewall refuses correctly the flow of information and logs the attempts
Sec-6	Authorized Production System	attempts to access the available services and data	System	QA/Test DMZ	The access to the data and the services is refused since authorized production system can only interact with authorized production systems	Firewall refuses correctly the flow of information and logs the attempts.
Sec-7	Authorized QA/Test System	attempts to access the available services and data	System	Production DMZ	The access to the data and the services is refused since authorized production system can only interact with authorized production systems	Firewall refuses correctly the flow of information and logs the attempts.
Sec-8	Developer Operator	attempts to access the internet from the system to download new packages	System	Production DMZ	Access is denied	Access is correctly blocked and the attempt logged in the Firewall
Sec-9	Developer Operator	attempts to connect to the database remotely and perform Create, Read, Update, Delete (CRUD) operations in either records or tables of it	Database	DMZ	Access to the database is refused	Both the database engine and Firewall correctly refuse connections and log any attempts
Sec-10	Developer Operator	wishes to have access to the services and data available in an unauthorized external system from	System	DMZ	A new flow exception is configured in the Firewall if the request is approved	New flow requests are logged and implemented in the Firewall, requiring an authorization from the competent sources
Sec-11	Malicious Developer or Operator	attempts to obtain privileged information from	System	DMZ	The logs do not expose such information	Privileged information is correctly filtered out of the logs

Table 2.1: Security quality attribute scenarios for our architecture

Scenario	Source	Stimulus	Artifact	Environment	Response	Measure
ModExt-1	System Operator	wishes to add/remove nodes from one of the components presented in the 2-Tier architecture	System	Runtime	Clones the corresponding component, turns it online and adds it to the pool of possible servers	After the clone is done, it must be possible to perform such operation in runtime with no downtime
ModExt-2	Developer	wishes to add/change/remove a new or an existing service	Portal	Design Time	Correctly identifies and alters the corresponding modules without affecting other services	Must be completed within 2 days for medium complexity services
ModExt-3	Developer	wishes to perform alterations in the User Interface	Portal	Design Time	Performs the alterations through the combination of widgets that can be built and reused on demand	Changes to a given UI do not affect other UIs
ModExt-4	Developer	wishes to create/change/remove a reusable widgets	Portal	Design Time	The widget is created, altered or removed without affecting any of the other widgets	Changes to the given widgets must propagate to the UIs that use it
ModExt-5	Developer	wishes to create new forms behaviour (eg. Forms with upload capabilities, with CSV capabilities, etc)	Portal	Design Time	A new form behaviour is created that can be reused by other forms later on	Only forms must be affected by this capability
ModExt-6	Developer	wishes to implement the application business logic	System	Design Time	He must implement the business logic behind the application in Perl	All code can be extended or modified without the additional costs of employing people with know-how in other languages

Table 2.2: Modifiability & Extensibility quality attribute scenarios for our architecture

users. If we consider the properties of general security scenarios, the source of a stimulus can be defined as the person or system who attacks our system with its own motivations. The stimulus can be defined as the attack itself, such as trying to access to an unauthorized resource, trying to forge new tempered information, trying to temper with the current information by updating it, trying to delete records without having permissions for such. The artifact can be the system itself, data within it or users within it, since the attacker may try to perform denial of services, steal information, or pretending to be a person it is not. The environment specifies if whether the attack is happening while the system is online or offline, within a DMZ or outside one, within a secure tunnel or outside one, etcetera. The response can be the counter measure that is applied when such attack happens, such as denying the access to unauthorized information, blocking or banning the access from the attacker to the system, logging the attack attempt, or other types of counter measures which fit the attack. The response measure can be defined in terms of how hard it is to perform such attack, how long does it take to detect such attack, how long does it take to recover or deflect it. In table 2.1, we can find the applied scenarios for the security quality attribute.

If we consider the properties of modifiability & extensibility general scenarios, the source of a stimulus can be defined as the person whom performs the modification or extension. This person is usually the developer, but there may be cases where it is another entity such as the user. The stimulus can be defined as the change necessity itself. It can indicate that a new functionality has to be added, altered and removed. The artifact can be defined as the target of the modification or extension. It can be the line of code, it can be the function, the method, the class, the module, the framework, the system, the quality attribute of the architecture, the resources, and so forth. The environment indicates on which phase it is possible to perform such modification or extension. Runtime, compile

Scenario	Source	Stimulus	Artifact	Environment	Response	Measure
Deploy-1	System Operator	Wishes to deploy a new minor release (e.g. a release that can be easily rollback by simply moving its codebase back in time)	Portal	Production Environment	Easily upgrades the portal with the system online by merging the new release	The downtime between minor releases must be lower than a minute
Deploy-2	System Operator	Wishes to deploy a new major release (e.g. a release which the rollback requires not only going back to a previous code commit, but also to restore a previous backup of the database)	Portal	Production Environment	Schedules a downtime and upgrades the system	The service downtime must be lower than an hour. The number of affected users by the service outage must be minimized
Deploy-3	System Operator	Wishes to test the deploy of a new release into production	Portal	Staging Environment	Tests the upgrade within a controlled environment	If it works on the controlled environment, it must also work on the production environment
Deploy-4	Developer	Wishes to performs a bug correction in a given release	Portal	Developer Environment	Easily upgrades or downgrades to the bugged release and its the corresponding environment	The bug fix must be easily deployable
Deploy-5	Developer	Implement new functionality	Portal	Developer Environment	Easily installs an isolated environment to implement the new functionality	The new functionality must be easily deployable

Table 2.3: Deployability quality attribute scenarios for our architecture

time, build time, design time are examples of specific phases. The response can be defined in terms of whether the point of intervention is identified and implemented smoothly. The response measure can be defined as the effort taken to perform such modification or extension. It can be defined in terms of the affected lines of codes, in terms of time spent in the intervention, in terms of the number of people required to achieve such, in terms of money invested on it, and so forth. In table 2.2, we can find the applied scenarios for the modifiability and extensibility quality attribute.

If we consider the properties of deployability general scenarios, the source of a stimulus can be defined as the developer, system administrator, operator who wants to act upon a new release. The stimulus can be defined as the need to deploy, upgrade, downgrade the system. The artifact is the system itself or part of it on which a software version will be upgraded, downgraded or rollbacked. The parts of a system can range anything as components, modules, frameworks, libraries and etcetera. The environment can be any of the production, staging, test, developer environments. The production environment is usually the environment on which real users access. The staging environment is a controlled environment with identical configurations to the production one, where the impact of new releases can be tested before an actual deploy. The test environment can be seen as the environment where the quality assurance team performs tests upon new releases. The developer environment is the environment where a developer performs his own changes and merges his code base with new releases. We can consider as possible values for the response, the event of successfully deploying a new version, the event of successfully performing a rollback after a deploy went wrong, or any other event related to the deploy of new releases. The measure can be defined as the time it takes to successfully deploy a release, the time it takes to successfully perform a rollback, the environments affected by a new release, the number of codebase merges required to perform such release, etc. In table 2.3, we can find the applied scenarios for the deployability quality attribute.

Scenario	Source	Stimulus	Artifact	Environment	Response	Measure
Support-1	User	Wishes to perform an operation and encounters an issue that prevents him from such	Portal	Malfunctioning	The issue encounter is automatically reported and logged along side with the error message	The error message is specific enough to research and solve the issue within hours
Support-2	Developer Operator	Wishes to find more information about the exceptions that have interfered with users' interactions	Portal	Malfunctioning	Information about exceptions and errors is available on a privileged view in the system only accessible by such	Error list is updated on a real-time basis
Support-3	Developer Operator	Wishes to find more information about the actions performed on the system	Portal	Any	Verifies the available system event log	System event log is updated on a real-time basis
Support-4	Developer Operator	Wishes to restore a previous backup version of the system due to a fatal failure	System	Destroyed	The system is restored through means of a backup that is available within the backup platform	The backup must have data from at least the previous week and before
Support-5	Developer Operator	Wishes to check on whether the network interfaces are functioning correctly, or on whether the system resources consumption is within its limits	System	Any	Information about such is accessed through privileged views on the monitoring systems	The system must run under a 75% threshold of its resources

Table 2.4: Supportability quality attribute scenarios for our architecture

When speaking in terms of supportability in our general scenarios, the source of a stimulus can be defined as a user, operator, administrator or system interacting with ours. The stimulus can be defined as a user accessing the system while encountering some issues. The artifact is the system or part of it that is being accessed, such as component, module, server, network node. The environment can be defined as having slow network, being degraded, having its resources at their limit, throwing exceptions, malfunctioning. The response can be defined as notifying the supporting team that issues are occurring, notifying the user that the system functionality is degraded, logging relevant events from different sources. As a measure unit, we can use how long does it take for the problem to get reported, how long does an issue take to get normalized, how many systems are generating logs, how long does it take to swap these log files. In table 2.4, we can find the applied scenarios for the supportability quality attribute.

Finally, when describing usability & understandability general scenarios, we can define the source of stimulus as being the end user who interacts with the system. The stimulus can be the need to learn the system, need to minimize errors, need to interact with the system efficiently, need to adapt the system or need to get familiar with it. The artifact in this type of scenarios is also the system itself or part of it. The environment can be defined as being either at runtime or at configuration time. Usually the runtime environment is related to normal end users interacting with the common functionalities of the system, while the configuration time environment is related to end users maintaining and operating the system. Responses defined for these type of scenarios must provide means for the end user to achieve what he wants.

If the end user wants to learn the system, then the system must provide a familiar interface. If the end user wants to use the system efficiently, then the system must provide aggregation of commands, smart navigation, simultaneous operations, effective searching, reuse of records. If the end user

Scenario	Source	Stimulus	Artifact	Environment	Response	Measure
Usability-1	User	Needs to be able to easily learn how to navigate within the portal	Portal	Runtime	Interface is familiar to the user	Users recognize the organization look alike within system
Usability-2	User	Needs to be able to fill a table of contents within a form in an efficient manner	Portal	Runtime	Portal allows to fill forms in an aggregated manner	Time taken to submit a form of this kind is reduced to the time it takes to fill a table inside a spreadsheet tool
Usability-3	User	Needs to be able to follow his own pace when filling the information required by a given form	Portal	Runtime	Portal support the user need to feel comfortable by keeping its data so that he may follow his own pace	The time on which the data is kept is on average long enough to support the slowest of the users
Usability-4	User	Needs to be able to navigate between different steps efficiently	Portal	Runtime	Portal provides for efficient navigation within itself	The navigational tree must be lower than two levels
Usability-5	User	Needs to have his errors minimized when filling the information required by a given form	Portal	Runtime	Portal guides the user and recognizes his errors	No forms submissions can have been accepted if they have not been validated

Table 2.5: Usability quality attribute scenarios for our architecture

wants to minimize errors, then the system must provide means to validate data, prevent errors, cancel or undo actions, recover from errors. If the end user wants to adapt to the system, then the system must provide means for customizability such as custom views and filters, custom content, custom local times, internalization. Finally, if the end user wants to get familiar, the system must provide ways to present itself in a simply and comprehensive way, support mechanisms which allow for the end user to interact with it in phases. Finally, we can define some of the measure units as being the time it takes to do a certain type of operation, the number of errors committed by the end user before he was able to achieve a certain operation, surveying user satisfaction, amount of data lost when an error was not prevented, etcetera. In table 2.5, we can find the applied scenarios for the usability and understandability quality attribute.

2.3 Architectural Components

In figure 2.2, we are presented with all the components that composed our architecture in its early stages. The figure is divided in three important boundaries: Intranet, Front-End DMZ and Back-End DMZ; which are enforced by separated firewalls in accordance to our organization security policies. Most of our security quality attribute scenarios are achieved through the enforcing of these boundaries by the firewalls. Let the Intranet be the network to which employees connect their computers, laptops, mobiles and other devices. Let the Front-End DMZ be the network to where the front-ends, that serve our system, will be placed. And let the Back-End DMZ be the network to where the back-ends, that comprise our system, will be deployed. It becomes clear that these three boundaries are the direct implementation of a 2-tier web application architecture. Our clients live inside the intranet boundary, but unlike a typical 2-tier web application implementation, they are only allowed to communicate with the front-ends through a proxy.

This restriction brings extensibility to our architecture, since we can easily upgrade our scenario from having a proxy, to having a load balancer whom efficiently dispatches requests to the available pool of front-ends. Since, the system was not designed to be exposed outside onto the internet,

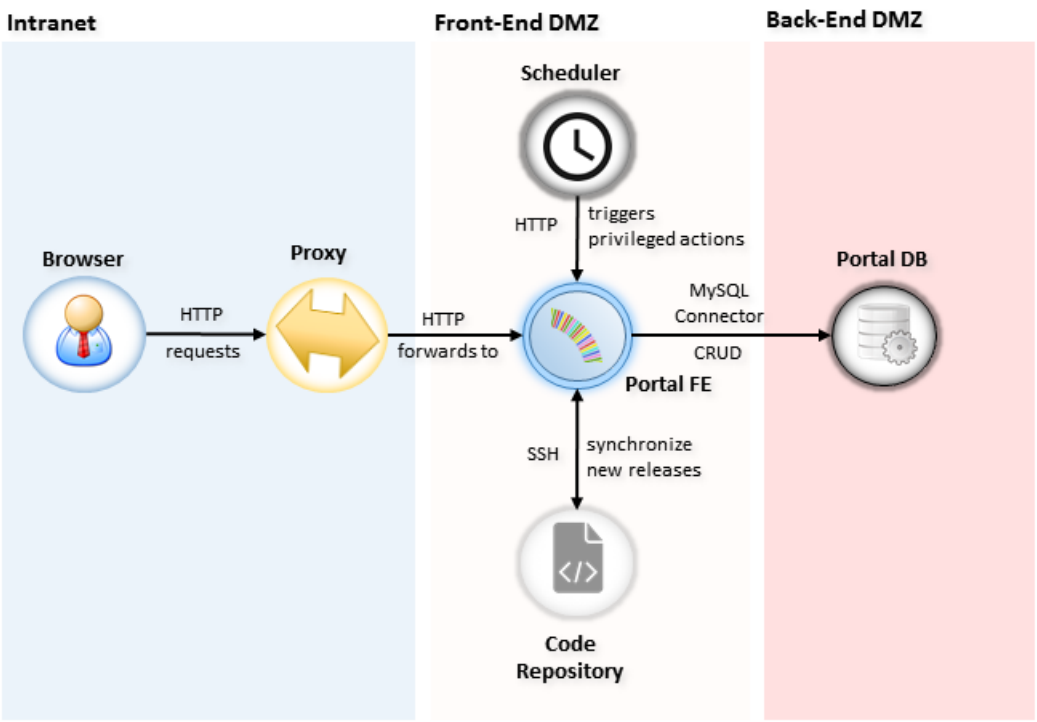

Figure 2.2: Architectural components for the portal catalogue

simply having as the possible universe of users, the number of employees currently collaborating with our organization, scalability was never an issue originally tackled in our architecture. Nevertheless, our `ModExt-1` quality attribute scenario defined that the architecture had to cope with the event of having, scalability introduced, as a new quality attribute. And that is one of the main roles of the given proxy. Not only it helps us segregate our systems as demanded in the security quality scenarios, but it also unlocks the dispatching of the Hypertext Transfer Protocol (HTTP) requests onto different nodes shall the need arise.

Thus, in the presented scenario, the clients are represented by a browser component whom is responsible to perform HTTP[16] requests such as GET, POST, PUT in order to receive data from the front-ends and provide data to them. These requests are not directly delivered by the browser component, they are brokered by the proxy component. The proxy component is the only component that is allowed to cross into the Front-End DMZ by the corresponding firewall. These reduces the exposure of the front-ends to the outside, but it also adds one extra hop into the request round-trip. The Portal FE component is responsible to respond to any forwarded message which arrived through the proxy. This component is actually comprised of two separated sub-components: the web server and the web application. The web server goal is to receive the HTTP requests, dispatch them into the corresponding web application (according to its defined rules) and send the web application response back into its client. The web application goal is to process the client requests according to the implemented business logic of that application into a corresponding response. Examples of softwares that implement web servers are the Apache, TomCat, Lighttpd, Internet Information Services and Nginx. The choice of a given programming language to implement the web application business logic can impact directly the web server technology choice since the web server must be capable of

interfacing with it. This segregation of responsibilities is extremely useful especially when considering the dynamism required within the deployability scenarios. Since the web application is what changes on every release, it is possible to restart the web application without affecting the web server. Also, if we take into account that a web server may be implementing multiple environments, restarting a web application belonging to one of them, does not affect the web application belonging to the other, even though they might be running inside the same web server.

The previously defined `ModExt-6` extensibility scenario, restrained the web application business logic to be implemented through usage of the Perl [17] programming language. This was mainly because the sponsors of the project wanted to reduce the impacts from having the person developing the web application leaving the organization. They believed this risk could be mitigated that way since most of the system administrator that operate the infrastructure were suppose to have a strong background in Perl. Of course, having a strong background in a programming language in order to operate systems, is entirely different than having a strong background in web developing. Nevertheless, no reasoning was possible on this very specific requirement, being the usage of Perl front-end and back-end technologies set on stone, even if it was seen as a niche technology when it comes to web applications. In order to fulfil some of our quality attribute requirements of modifiability, extensibility and usability, a mature, production ready, Perl MVC framework was chosen to implement the web application itself. The most popular Perl MVC framework named Catalyst[18] was chosen due to the complete and clear nature of its documentation, support for Object Oriented Programming, wide availability of rich plugins and easy extensibility.

Still inside the Front-End DMZ, we have the Code Repository component which is mainly responsible for the version control of our system code base and thus, our deployability scenarios depend greatly on it. It acts as a main repository to where specific branches are merged and pushed to. Git[19], a free and open source distributed version control system was the chosen software that implements this component. Its flexible working nature allowed us to implement our architecture deployability necessities into it. Unlike older technologies of version controlling such as SVN[20] and CVS which work on the basis of central repositories, Git was developed to have a peer to peer nature. Every client has a local copy of the repository on its own system. Therefore, performing actions which affect the branches inside a repository is always possible independently of having a connection or not. Since every action has a local nature, the time consumed to perform it is impressively fast. Of course, a version control system is useless, if its clients cannot merge their code bases. Thus, Git allows for its clients to explicitly push or pull branches from other clients in order to synchronize code bases. It was with this scenario in mind, that Git was implemented into our architecture. Our central Code Repository component holds a remote to where all the code bases have to be merged to. Developers developing new features, correcting issues, etc, can perform their job locally until they achieve their goal, after which they must then push the changes into the main repository. Operators who want to deploy new releases into the Portal FE, can pull the corresponding release from the Code Repository into the to be affected Portal FE component, needing only to restart the service afterwards.

Finally, inside the Front-End DMZ boundary, we have the Scheduler component. Its main purpose

is to trigger maintenance and privileged actions on the other systems. For instance, to improve the experience of our users when it comes to fill the available forms, we have divided the forms in many steps. The data submitted by our users must only exist while the session is alive. If an user uploads a temporary file required by a given form but then proceeds to move onto a different form, the temporary file will be indefinitely in our system consuming resources, if no garbage collector is in place. The same can be said of a given user session data, it will live indefinitely unless a garbage collector is deployed to reset it on a timely manner. Thus, the purpose of the scheduler component is to trigger these type of actions which maintain the system within its normal operational parameters. Without this component, implementing the `Usability-3` scenario, could provoke as a side effect, the depletion of all the server resources, thus impacting the negatively the service. These actions may or may not be defined and implemented within the web application business logic. If they are implemented within the web application logic, then the scheduler authenticates itself with the Portal FE and invokes them through HTTP requests. If they are implemented outside the web application logic, then the scheduler accesses and executes bash commands through SSH in the corresponding components.

Inside the Back-End DMZ boundary, only accessible by components existing within the Front-End DMZ, we have the Portal DB component. The Portal DB main purpose is to store and persist all the data related to the web application domain. This component must be capable of ensuring the Atomicity, Consistency, Isolation and Durability (ACID) properties of transactions which act upon its data. There are many database engines which delivery these properties such as MySQL, MariaDB, Postgres, Oracle DB, Microsoft SQL. Each one of them has its own quirks, advantages and disadvantages. MySQL was the adopted technology in our system due to its free licensing model, bootstrapping simplicity, specific SQL additions, rich set of supporting tools and ease of expanding into a cluster architecture in the future. To complement the restrictions enforced by the DMZ firewall, on which only systems residing inside FE DMZ can reach the database engine, MySQL allows one to define rules to block connections from sources that are not specified in its security rules. This kind of features allow us to reduce the impact of the damage, should a DMZ firewall get compromised by a malicious entity.

Additionally, MySQL toolset includes a workbench which facilitates the developer job of designing the database schema. The designing of database schemas within this tool is done through What You See Is What You Get drag and drop interactions. It speeds up greatly the job of setting up new tables, columns, relationships and indexes by automatically generating the corresponding script which implements the designed schema diagram. It also facilitates the synchronization of new changes made within a schema into the corresponding databases. The tool automatically detects the differences and generates a script which upgrades the old schema into its most recent version. These features present in the workbench tool greatly improve the deployability and extensibility of our system. Obviously, this component is then accessed by the portal front-end component through a Perl MySQL connector which implements the necessary communication protocols through TCP connections. The portal front-end component uses then these connections to perform Create, Read, Update, Delete[21] upon the persisted tables. These operations are executed through the usage of prepared

Web Server	HTTPS	Virtual Host	CGI	FCGI	SCGI	WSGI	ISAPI	SSJS
Apache	Available	Available	Available	Available	Available	Available	Available	Available
Lighttpd	Available	Available	Available	Available	Available	-	-	-
Microsoft IIS	Available	Available	Available	Available	Available	-	Available	Available
Nginx	Available	Available	-	Available	Available	Available	-	-

Table 2.6: Characteristics of different web server technologies

statements[22], in order to stop any attempt to perform SQL injection attacks.

2.4 Processes View

As we mentioned before, the Portal FE component is actually made up of two distinct sub components: the web server and the web application. Overall, the choice of one web server over another does not impact that much the architecture. That is mainly because they share many of the main features. It is very easy to achieve on one of the technologies, what is available on the other. It all boils down to what type of interface are we going to use in order to generate the dynamic web pages, the languages that implement those interfaces, operational know-how with a given web server, how lightly it runs and serves pages compared to the others. In table 2.6, we can see a comparison of features and interfaces between some of the web servers mentioned before. Obviously, all of them feature Hypertext Transfer Protocol Secure (HTTPS)[23] and virtual hosting. The HTTPS allows for the secure communication between the clients and the server through the means of secure SSL channels. Although, at this stage, our system did not have any privileged information travelling through the data center channels, it was important to ensure that the web server was capable of switching to these technology, should the necessity arise at future stages.

The virtual hosting allows for one web server to host more than one web application. This is possible through the usage of dispatching rules, which are defined through combinations of source ips, destination ips, destination ports, protocol use, endpoint accessed, domain, etc. Once these have been defined, the web server becomes capable of dispatching requests to different web applications in compliance with the specified rules. If for example, we wanted to take advantage of our resources and deploy the production system and the staging system within the same web server, two virtual host rules would be defined. Virtual hosts are extremely useful when it comes to implement our deployability quality attribute scenarios, while just having one instance of web server deployed. The available interfaces in the web server which web applications must be able to interact with in order to be deployed come next. They also affect the same group of quality attribute scenarios mentioned before. The Common Gateway Interface (CGI)[24] is a standard that defines how can web servers interface with web applications that generate web pages dynamically through programs and scripts installed on the given server. These scripts can be in a varied range of programming languages since they are executed within the server environment, and their result returned to the browser. Since it requires that on every request, the web server has to launch the process which executes the program or the script, its bottleneck becomes the spawning of a new process rather than the processing work

in order to generate the output. This bottleneck becomes even more noticeable if the program or the script is using interpreted languages, because not only it has to launch a new process, but it also has to launch the virtual machine environment where the language will be interpreted.

The FastCGI (FCGI)[25] is the evolution of CGI which seeks to solve its ancestor performance and scalability issues. To achieve that, FCGI bases itself on resource re-usage. Instead of having to go through the hassles of spawning a process on demand every time it receives a request, FCGI pre-launches a pool of processes that will be reused for every request. Thus, the performance impacts of creating and destroying processes context on every request are eliminated. Web applications which implement FCGI, can interface with their web servers in three ways: through a single socket connections within which arriving requests are then multiplexed to different processes within the pool, multiple socket connections which map to different processes within the process pool, or a mix between both of them. The Simple Common Gateway Interface (SCGI) is also an improvement upon its CGI ancestor which follows many of the FCGI ideas, but at the same time, simplifies its protocol in order to be easier to implement. The Web Server Gateway Interface (WSGI) is also a specification for an interface between web servers and web applications. But in this case, the specification is designed towards Python web applications. It seeks out to make Python web application more portable to different web servers, by introducing a new layer which hides away the complexities of implementing CGI, FCGI or other interfaces from the web applications.

The Internet Server Application Programming Interface (ISAPI) is the proprietary tackle on the web server to web application interface problem by Microsoft. This interface allow for the usage of their proprietary programming languages (such as VB.NET, C#, F#) in web applications. These web applications are extensions to the ISAPI web server with are compiled into DLL and loaded by it. These extensions have access to all the functionalities of the web server, such as accessing the object structure of the incoming request, or outputting the corresponding response. ISAPI also provides another way of intercepting the web server behaviour through the definition of filters. The code programmed within these filters is ran on every incoming request and/or every outgoing response. They are basically interceptors that allow one to define special behaviours before delivering a request/response to its destination. Finally, Server-Side JavaScript (SSJS) is a recent interface which empowers the writing of server side web applications in Java Script. There is no standard or specification on how interface must be implemented. Thus, it is quite probable that a JavaScript (JS) web application written to interact with a given web server will need some tweaks in order to be ported into a different one.

In our architecture, we have adopted Lighttpd as our web server of choice since it was one of the many which supported interfacing with perl web application and flexible enough to implement several distinct deploy scenarios. The main idea was to take advantage of its virtual hosting in order to create the separated production and staging environments. In figure 2.3, we are presented with the process view for the Portal FE component. The Catalyst MVC framework allows integration with web servers that support both CGI and FCGI. When the Catalyst FCGI web application process launches, it listens to a specified TCP[26] port. Lighttpd then uses that connection in order to forward the requests it receives. Once the requests have been processed, the Catalyst FCGI web application sends the

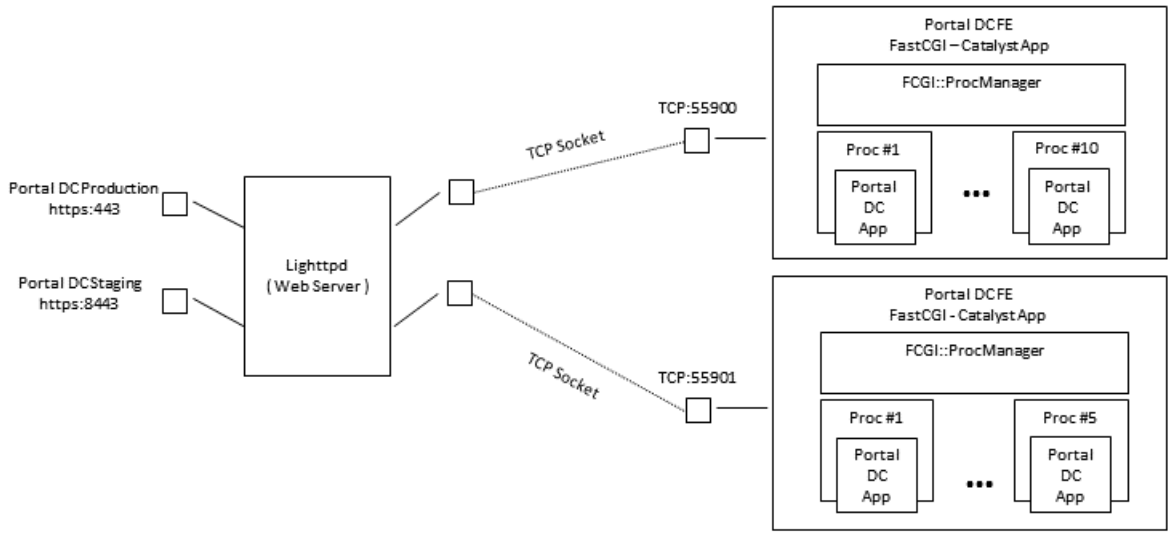

Figure 2.3: Processes view within Portal FE component

back response to the web server through that socket, which then proceeds to send it back into the client. It is possible to create multiple environments within the same web server, by setting virtual hosts that depending on the a defined rule, dispatch to the environment of different launched Catalyst FCGI web applications. This separation between the web server and the web application allows us to shut down or reboot web applications seamless by loading a temporary FCGI web application in the same socket to answer the requests while the web application is down.

Catalyst FastCGI applications are bootstrapped by a module named FCGI::ProcManager, which is responsible to manage the pool of pre-forked processes which answer to requests. This module spawns a pre specified number of processes which will interpret the perl web application. It is during this bootstrap process that the perl interpretor loads up all the necessary packages. Thus, the bottleneck of loading all the packages is reduced to only happen at bootstrap, having the requests dealt with the utmost performance afterwards. It also means, that if we needed to deploy additional processes in order to deal with an increased flux of requests, we would be able to create them on demand with the cost of only forking a new copy of the others process, having no additional time wasted in the re-loading of packages. Since processes are spawned through the operating system fork function, they take advantage of the copy-on-write semantics. When they originally get forked, they will all point to the same memory zone and objects rather than having right away separated memory zones. Then, an actual copy of the objects held in memory only triggers when a given process tries to alter the values held within it. Since most of the framework modules and most of the web application business logic modules simply implement flux control, these zones rarely get rewritten. Therefore, these memory zones rarely go through the process of getting duplicated, leading to a reduced memory footprint of the deployed web application process within the FCGI manager.

In our concrete scenario represented within the figure 2.3, both the production and staging environments are deployed within the same web server. Through the defined routing rules, the web server is capable of distinguish to which FCGI interface does it have to forward the requests. If the client

connection was established through the HTTPS port 443, the request must be forwarded to the production FCGI interface. If the client connection was established through the HTTPS port 8443, then the request must be forwarded to the staging FCGI interface instead. These two FCGI interfaces are deployed on distinct TCP endpoints. These endpoints may or may not live within the same servers. In our case, they are deployed within the same server, since we expected a low usage load on the system being designed. Nevertheless, the architecture modifiability attribute was present in these components. If we saw ourselves in the scenario where the system needed to become scalable, the technologies mentioned above provided the required modifiability to break these components into multiple instances by deploying them in more servers. Needless is to say that, the processes launched through the production FCGI application ran the production branch of code, while the processes launched through the staging FCGI application ran the soon to be production branch of code. Thus, their environments, connections, databases and code bases are completely distinct in order to segregate impacts.

2.5 Layered Module View

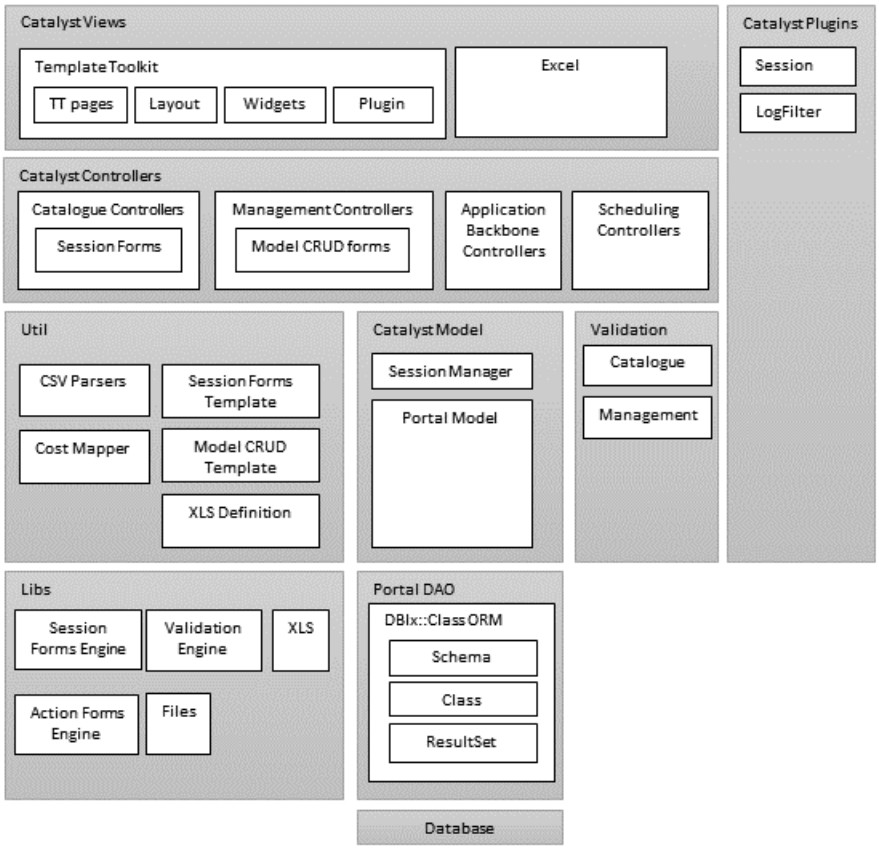

Figure 2.4: The layered module view from Portal DC

We will now describe our system while resourcing to the layered module view presented in figure 2.4. Our layer model supports itself upon the guidelines defined by the Catalyst MVC Framework, while extending that model at the same-time in order to cope with additional complexity. The mod-

ules presented in the layers of our figure are only allowed to directly interact with the adjacent layers which are either bellow or at the side of them. Thus, the layer of `Catalyst Views` may only interact with the layer of `Catalyst Controllers` and with the layer of the `Catalyst Plugins`. The layer of `Catalyst Controllers` may interact with the layers of `Util`, `Catalyst Model`, `Validation` and `Catalyst Plugins`. The layers of `Util`, `Catalyst Model`, `Validation` may interact with the layer of `Libs` and `Portal DAO` as well as between each other. Finally, the layer of `Portal DAO`, may interact directly with the layer of `Database` and `Libs`. These segregations introduce a responsibilities separations which improve the extensibility and modifiability of the system as a whole. We shall now detail the modules within each layer of responsibility in a bottom-up approach. This bottom-up approach will help us to better understand the set boundaries, as well as the need for the uses relationship implicit to the layered model.

2.5.1 Database

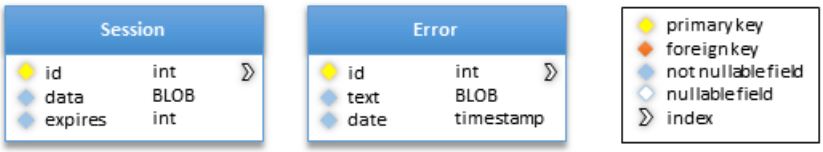

Figure 2.5: Architectural specific tables

The `Database` layer contains the implementation of our web application schema within the chosen database engine. Among the necessary tables to implement our business logic, we have the `Session` and `Error` table, which empower the `Usability-3` quality attribute scenario and the `Support-1` & `Support-2` quality attribute scenarios respectively. These tables and their respective columns, are represented in figure 2.5.

In order to support the web application sessions, we have the `Session` table defined within the schema. This session is simply characterized by the id which uniquely identifies it, followed by a data field which will contain the actual serialization of the session objects. Since this serialization is to be done through marshallers which result in human readable serialized sessions, the data is to be first compressed before being inserted or updated. This guarantees that the memory footprint of a session within our database is minimal, at the cost of being unable to perform queries upon it. If such compression was not in place, it was possible to encounter cases where the serialized version of a given session would exceed the size limits within the given column. An expires time stamp is also presented in the table, to help the scheduled maintenance routines to know when a session can be freed due to inactivity.

Finally, to provide some of the qualities defined within our supportability scenarios, we have the `Error` table. The idea is to have errors automatic dumped into entries of this table, so that operators and developers may act pro-actively on them. This is achieved by having what one could call of a black hole of unexpected exceptions, which graciously presents an error message to the user, while registering the full error stack dump within this table. This table is characterized by an unique

numeric id, followed by a text field which actually holds the corresponding error message dump. The time stamp corresponding to the moment when it happened is also registered, so that the developer tracking the problem may audit the logs in order to perceive the requester order of actions.

2.5.2 Portal Data Access Object

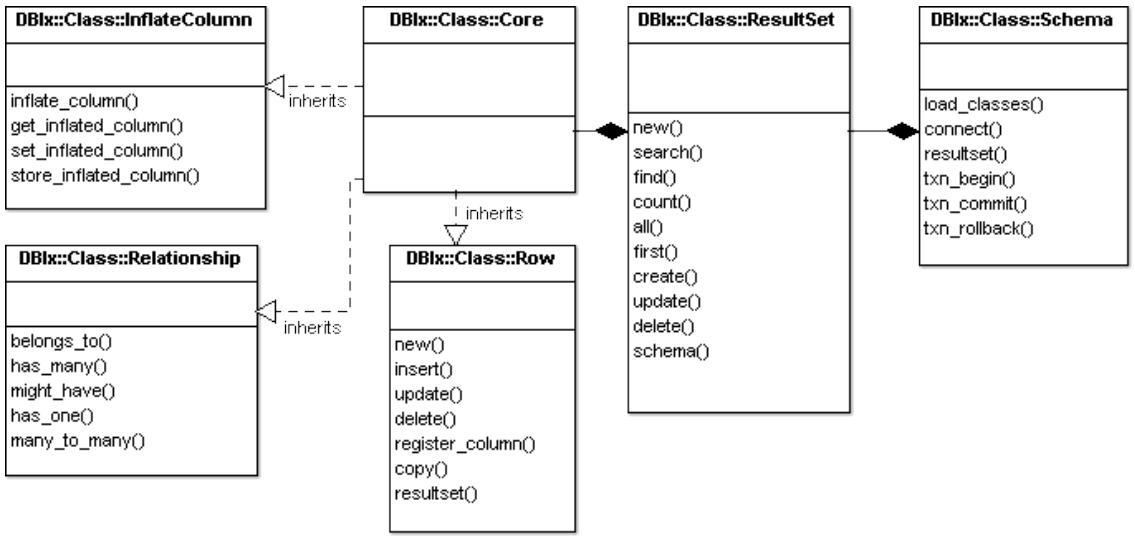

Figure 2.6: DBIx::Class - an extensible and flexible Object Relation Mapper

The bridge between the database persisting the domain and the web application was done through the usage of an Object Relation Mapper (ORM). The goal was to have a layer in-between which mapped our relational domain schema into perl domain objects. By doing such, we are able of improving the modifiability and extensibility of our system, since the SQL intricacies are hidden from the developer implemented the business logic. If we were to change from a MySQL database engine into an Enterprise Oracle database engine, there would be no need to change even a single line in our application. Truth be said, by introducing an ORM in-between, we are just shifting from direct SQL database dependencies, to long lasting ORM dependencies. Nevertheless, the introduction of these new dependencies compromise less the architecture of the system, since it is more likely that, within the life cycle of our system, we have needs to replace or update database engine, rather than redoing the ORM layer. Thus, the ORM abstracts all the underlying complexities which are interchangeable between database engines, such as building and executing SQL queries automatically, fetching data from tables while mapping it into objects, pushing changes done on objects back into the source rows, preventing SQL injection attacks by automatically build the queries through prepared statements, exposing the relationship between tables with relationships between objects, and many other abstractions.

The most mature ORM available in perl is known as `DBIx::Class`[27]. This package is an implementation of the Active Record[14] pattern. The Active Record pattern is characterized by an object which contains both the data and the behaviour of on how to persist information within the database,

along side with extra behaviour which defines the domain logic upon that given data. We have chosen this model in order to iterate faster the development of new functionalities as required in the ModExt-2 modifiability quality attribute scenario. In figure 2.6, we have a simplified UML view from the most relevant classes and methods from this package. A developer whom wishes to map a schema table into a perl class must extend the Core class. This class is a junction of three other classes: Row, Relationship and InflateColumn. The Row class provides the CRUD methods which interface directly with the database. Since an instance of a given class corresponds directly to a row in a table, these internal methods are necessary in order to interface with the database engine. Additionally, this class also allows the developer to register the columns of that given table, so that getters and setters are automatically added to the resulting class.

The Relationship class, buffs our Core class with the means to define relationships such as belongs to, has many, might have, has one, many to many. By invoking these methods with the desired parameters, new accessors are dynamically added into our class, which unlock the ability to traverse or alter the tree of our instance relationships. When one of these newly generated relationship method accessors is invoked, the corresponding SQL query is automatically issued to the database, resulting in the creation of the corresponding result sets or object. Finally, the InflateColumn class provides the means for the developer to define special getters (inflate) and setters (deflate) from a given column. The most obvious example is wanting to access a field which is from the type date time. If no inflation/deflations are performed on the corresponding column of that field, the date and the time will be fetched as a simple string. On the other hand, if one defines a new behaviour for the inflation and deflation of the given field, one will be able to create the corresponding date time object rather than the simple string.

The class ResultSet exposes the basic operations upon a given table. Out-of-the-box, it allows for the developer to create and persist new objects, to search for the objects which match a given specification, to find the object by its identifier, to count all the entries within a given set, to iterate the objects within a set, to update all the objects within a set, to delete all the objects within a set, and many other which are not relevant enough to go into more detail now. All these methods have their own interface and require no SQL to be written, since the ORM will write and issue the corresponding SQL itself. It is possible for the developer to extend this class, in order to implement his business logic related to the manipulation of result sets. Finally, connecting all the dots, we have the singleton Schema class. This class abstracts the database schema itself and the navigation between its different result sets. We define our schema as whole by extending this class and by indicating which of our custom classes, we want to load up. When instancing our newly created schema, we can connect to a specified database engine endpoint of our choice. Then, we can navigate through the different result sets as our business logic requires, and if necessary, control where our transaction start, commit or rollback.

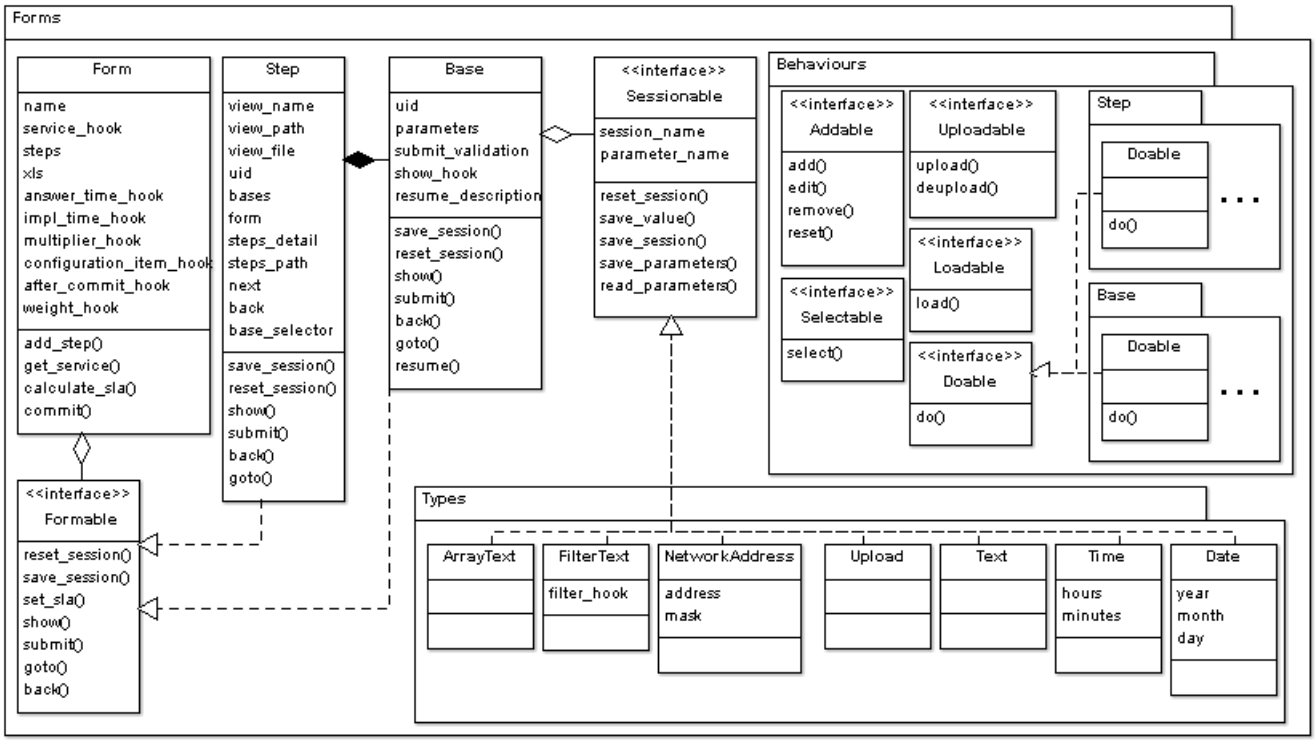

Figure 2.7: Session Forms Engine UML Class Diagram

2.5.3 Libs

In figure 2.7, we have the detailed UML class diagram of the session form engine which empowers most of the catalogue service forms. This module was developed with the concern of easier extensibility and modifiability in mind. The `ModExt-2` and `ModExt-5`, modifiability & extensibility quality attribute scenarios, directly depend on it. It provides the means to quickly develop new full fledge forms or to easily change them by encapsulating common behaviours which can be found in almost every form. If the need for a new common behaviour arises, the engine can easily be extended, in order to start supporting that new scenario. Session forms, are forms where their data lives within the session, until the very final moment where they are truly committed into the system. Only after that, will these forms trigger actions which may change our database state through the manipulation of the available portal domain access objects. The values received through the HTTP parameters are mapped in specific structures within the web application session. The whole module is implemented through the means of Inversion Of Control (IOC)[28], where the developer has well defined intersection points, where he can plug his own logic into the flow of the engine, rather than having him defining both the flow and the underlying logic. If we consider that the engine is working properly, then applying inversion of control strategies helps to reduce the places where the developer may break unintentionally the code.

The Form class is the main entry into this module. This way, a developer only needs to know where can he plug the intelligence he is designing, and all the magic will happen behind scenes. It defines the name of the form, the corresponding service it consumes, the different steps which constitute it and the means to intercept on how the answer time, implementation time, multiplier, configuration items and weights are mapped. It is within this class where the logic of finally committing a group

of session forms, can be programmed. In our system, this logic involves performing a couple of verifications, leading at the end to the creation of new domain request and its respective persistence. In the form engine, steps are classes which implement the interface `Formable`. The interface `Formable` requires that a class implementing it must define the behaviour of: resetting a step session, saving a step session, filling the information to be shown in the step form, stepping forward into the next step by submitting the current one, jumping into a specified step by submitting the current one, going back into the previous step by submitting the current one. These methods are important since they define the our navigation model for the type of forms which we want to provide to the user, as well as, the mapping of the provided HTTP parameters of a given a request into the underlying connection session.

The classes `Step` and `Base` implement the `Formable` interface. The concept behind their design is the implementation of the Composite Design Pattern[29], where they both implement or extend the same interface and where a `Step` is composed by `Base` classes. Whenever a `Step` method is invoked, it invokes the very same method on its `Base` children. This design pattern was rather useful in the designing of these classes, because it was important to empower the concept of sub-forms within the same form step. For instance, the developer may have to design a step where he has a couple of normal text fields which are related to a subject, followed by a multi-entry table of text fields related to a different subject. The underlying logic to fill normal fields, is completely distinct of the underlying logic to manipulate a multi-entry table. When implementing the normal fields, one only needs to submit or clear them, when implementing a multi-entry table, one needs to be able to perform the CRUD actions on it. Also, a group of form fields which share the same logic, may appear in more than just one form, thus, to abide to the best practices of the Don't Repeat Yourself[30], it makes sense to provide ways of reusing these fields implementation.

Therefore, the `Base` class implements that sub-form grouping abstraction. In its essence, a base is an aggregation of form fields which we will denominate as parameters. The `Base` class is responsible to define which parameters does it take care of, fetching and resetting them and validating them on every submission. These parameters are classes which extend the interface `Sessionable`. These interface specifies which methods have to be implemented in order to perform HTTP parameters into session mappings and vice-versa. This means that whenever a base is saving its session, it is simply iterating through its parameters and invoking the corresponding methods. This abstraction allows us to define different type of fields, which map themselves differently depending on what they abstract. Thus, within the package `Types`, we have the specific classes which implement the given interface. The type `ArrayText` knows how to handle an HTTP parameter which provides multiple values. The type `FilterText` knows how to handle simple HTTP parameter, but at the same time, provides the means for the developer to apply a defined filter. The type `NetworkAddress` implements the handling of network addresses. The type `Upload` implements the handling of uploaded files. The type `Text` implements the handling of normal text parameters, but internally it sanitizes the entered text. The type `Time` implements the handling of time parameters, and finally, the type `Date` implements the data handling.

Then, still within the session form module, we have the `Behaviours` package. This package contains interfaces and mixins/traits classes, which expand on what behaviours can a step and a base implement. The bases classes of `Step` and `Base` only implement behaviours related to normal form submission and navigation. Even though, this behaviours may be more than enough for forms with simple interactions, they come short a bit of short when considering more complex scenarios such as the multi-entry table mentioned before. Thus, the `Behaviours` defines the necessary behaviours which can be plugged into the original `Step` and `Base` classes through the usage of traits or mixins. This basically means, that the developer, when instantiating a step or a base, may additionally include the corresponding mixin/trait along with the new instance. This mixing process simply imports all the methods and attributes from the given mixin/trait into the instance. This results on an augmented instance which contains more methods and attributes than its original class. Since, steps and bases are implemented through a composite, whenever we mixin the `Step` instance, we have to mixin its `Base` instances with the class wich implements the corresponding mixin/trait for the `Base`.

Consequently, the interfaces present within the `Behaviours` package, specify which mixins/traits are available and their corresponding interface which needs to be implemented. Then, within the packages `Behaviours::Step` and `Behaviours::Base`, we have the corresponding implementation for the classes `Step` and `Base` respectively. This strategy provides great extensibility to our form engine. If a new type of interactions had to be designed, in order to improve the user interaction, the developer would only need to design a new behaviour and implement its step and base logic. In our engine, the out-of-the-box available behaviours are: `Addable`, `Uploadable`, `Loadable`, `Selectable` and `Doable`. Through the usage of different combinations of these mixins/traits, it is possible to implement most of the forms use cases which we have seen so far. The `Addable` behaviour allows for multi-entry table manipulations. In order to achieve such, it introduces methods which correspond to the adding, editing, removing and resetting of entries. It also introduces new attributes, such as how is the adding validation performing, but we will not go into details since it is beyond our scope. The `Uploadable` behaviour empowers our classes with means to receive uploads and remove them. The `Loadable` behaviours gives our engine the means to load uploaded CSV files into values for the base parameters. The `Selectable` behaviour simply provides the means to have server-side selectables within our forms. Finally, the `Doable` behaviour provides means to implement generic behaviours which take place whenever a submission succeeded or failed. In the appendix A, section A.1, code example 1, we have an excerpt of code where a form step is created through this engine.

In figure 2.8, we have the detailed Unified Modeling Language (UML)[31] class diagram of the actions form engine which empowers forms which work directly with the received parameters from the HTTP requests. This module was also developed with the concern of easier extensibility and modifiability in mind. It is a more versatile spin-off version from the previously presented session form engine. Thus, both engines structures and rationale are very similar in essence, by exploiting the inversion of control model. A developer who designs forms with the session form engine, will be able to easily develop forms using this action form engine. Unlike the session form engine, the actions performed within the action form engine directly impact the entities of our domain. Thus, the usage of

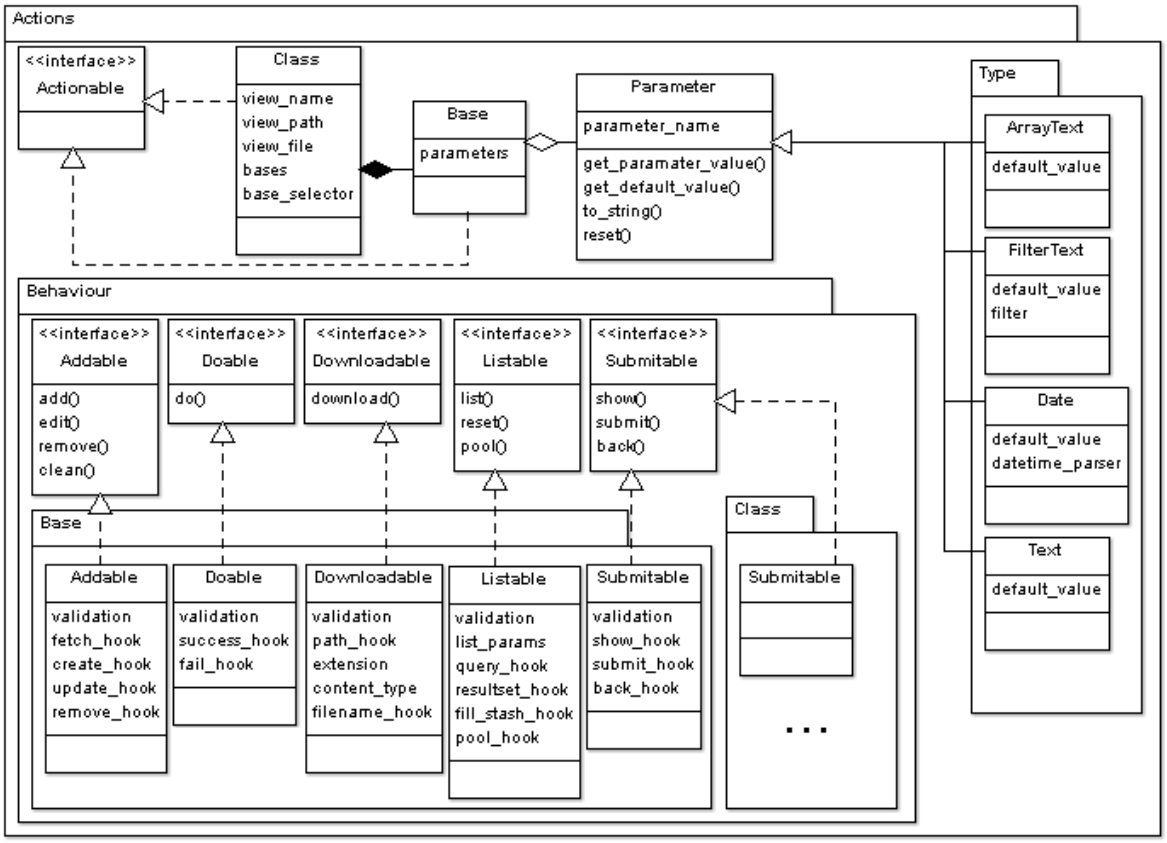

Figure 2.8: Actions Form Engine UML Class Diagram

this engine is advisable when performing CRUD on the data access objects, or when presenting and filtering lists of these entities. Take for instance the service entity: if we were to implement the logic of creation of a new service, then the indicated engine for such would be the session form engine; if we were to update a previously created service, then the indicated engine for such would be the action form engine. In other terms, session form engine boundary is set on soon to be entities, while the actions form engine boundary is set on the manipulation of actual entities.

To quickly explain the action form engine module, we shall simply walk-through the differences comparatively to the session form engine. This engine does not contain a form class abstraction. This is mainly because, even though this module was designed to be used within forms, it is possible to use it in things other than forms, as long as we still need to manipulate passed HTTP parameters somehow. Therefore, there only is an empty interface `Actionable`, interface which other classes will realize in order to have a common interface. `Step` class was replaced with `Class` class. The composite design pattern is still applied through the classes `Class` and `Base`. The `Base` class parameters are now an aggregation of instances which extend the abstract `Parameter` class. This abstract `Parameter` class obliges to define the necessary methods in order to fetch the values from the HTTP request. Likewise, there is a subset of classes within the `Type` package which implement concrete realizations of it, such as the `ArrayText` type, the `FilterText` type, the `Date` type and the `Text` type.

Although, both `Class` and `Base` classes are empty of behaviour since the interface `Actionable` defines none, it is possible to insert the desired behaviours through the usage of the mixins or traits

defined within the Behaviour package. The way they are organized is very identically to the way they were organized in the previous engine. The package itself contains a number of distinct interfaces which need to be implement in order to provide a certain type of behaviour. Then, the concrete implementation is spread between the classes which implement it for the Class class, and the classes which implement it for the Base class. The available behaviours for this engine are: the Submitable behaviour, the Addable behaviour, the Doable behaviour, the Listable behaviour and the Downloadable behaviour. The Submitable behaviour provides the behaviour on which a given form is either submitted or cancelled. If a form is submitted, its contents are first validated, and if successful, whatever is specified within the submission hook is executed. The Addable behaviour provides the means to implement multi-entry tables which affect directly our data access objects. Depending on, whether the action being executed is a creation, an update or a removal, the corresponding defined interception hook is called if the validation was successful. The Doable behaviour implements a generic behaviour where the developer can define success and failure actions which are then executed depending on whether the validation passed or failed. The Listable behaviour provides a way of filtering and listing entities within our domain. The developer defines the available filters parameters, their validation, how is the query built, how are the result sets fetched, how is the pooling to know if new entries appeared done and how to fill the stash which is passed then into the web application. Finally, the Downloadable behaviour provides the equivalent of accessing the file system in order to fetch the specified file within the HTTP request parameter, and sending it back into the source if the validation was successful.

In order to continue the description of the remaining modules inside the libs layer, we shall now tackle the validation engine module, the XLS module, the files module and the template plugins. The validation engine is a simple and yet flexible module which provides the means to encapsulate validation rules into classes. These classes, with defined validation rules, can then be consumed by any module which needs to orderly validate an input. The session and action form engines are an example of module which depend on this validation engine. Examples of what type of rules can be defined with the engine are: if a given field is required, the minimal length of a field, the maximum length of a field, the matching with a given regular expression, matching with the values within a result set, executing a custom code defined verification. If more types of validation rules became necessary, it is possible to easily extend this engine in order to support them. The XLS module contains all the utils necessary in order to generate excel files. These utils know how to create new spreadsheets, fill lines of those spreadsheets with text, build multi column tables with filled text, etc. In short, it is with the help of this module that the later defined excel views will be generated. The files module is responsible to help manage the files which were uploaded within the file system. And finally, the template plugins module is nothing more than extensions which are used within the modules belonging to the views.

2.5.4 Validation

The Validation layer presented in figure 2.4 contains concrete implementations of validation classes which use the validation engine module defined within the Libs layer. These classes are

spread throughout two distinct groups: the `Catalogue` validation group and the `Management` validation group. The `Catalogue` validation group contains all the validation classes that implement the underlying validation rules belonging to forms which make up our catalogue of services. In order to decouple responsibility, for each available service and related forms, we have a corresponding validation class which implements the varied validations rules existent within the different steps of it. This segregation makes it possible to ensure that changes to validation a specific validation class will not affect the validations of unrelated services. Consequently, the `Management` validation group contains all the validation classes that implement the underlying rules of validation belonging to forms that manipulate our portal data access objects. In the appendix A, section A.1, code example 3, we have an excerpt of code where we can see a concrete validation definition.

2.5.5 Catalyst Model

Within the `Catalyst Model` layer of our web application, we have the classes which either instanciate the Catalyst Framework `Component` or `Model` classes. These classes bring predefined implementations which allow for the direct integration with schemas implemented through the `DBIx::Class` ORM framework. Thus, creating a new model within the catalyst framework, simply implies extending the corresponding class, while providing the connection string parameters such as the endpoint, the username, the password, the database, the encoding, etc. The result of performing such is an instanced class which exposes all the methods available within the corresponding schema implementation. Thus, in our case, the `Portal Model` class, is an example of such instanciation, which exposes itself as the proxy to the `Portal DAO` schema. This means that the available controllers within the web application gain access to the portal data access object model through the framework context object. They simply need to access the corresponding method which returns the component `Portal Model` and navigate through the available result sets as they deem fit. On the other hand, the `Session Manager` component is responsible for the maintenance of the resources related to the session itself. Its main goal is to keep track of the files which may have been uploaded throughout the process of filling a form, but that were never actually committed into the system. If no track was kept on this temporary uploads, they would be able to live indefinitely within our system, consuming unnecessary precious resources. Thus, the `Session Manager` component keeps track of all these files and the time when their sessions expires. It provides methods to perform clean-ups on the resources consumed depending on whether or not the session has expired. These methods are then consumed by the respective `Scheduling Controllers`.

2.5.6 Util

Inside the `Util` layer, we have the modules which provide concrete implementations to a series classes which will be consumed by the controllers. For instance, the `Session Forms Template` package contains concrete implementations of forms generated through the `Session Forms Engine` module, which will be shared among many controllers. Lets consider the scenario, where we have a set of forms, where the requester has to fill the multiple affected configuration items of the given request.

Instead of having the logic of creating these step or bases spread throughout the different controllers that implement those forms, we define it on a common place and reuse it at will. In the appendix A, section A.1, code example 2, we have an excerpt of code where a `Session Forms Template` is created. The idea behind the `Model CRUD Template` package is the same, but applied to concrete implementations of forms generated through the `Action Forms Engine` module. The `XLS Definition` package contains all the concrete implementations of the translations between service requests and excel files. The `CSV Parsers` package implements the logic of converting uploaded CSV files into the corresponding values within a form. For every service which allows the auto-filling through the means of uploading a CSV file, it contains the respective class which implements that translation behaviour. The `Cost Mapper` package contains the implementation of the classes which know how to map the fields filled within a form to their corresponding monetary cost.

2.5.7 Catalyst Controllers

The `Catalyst Controllers` layer is the entry point for arriving requests. It is responsible for implementing the business logic by manipulating the model domain and gather the required information to be passed onto the `Catalyst Views`, so that they may generate the corresponding response. The Catalyst MVC framework injects a context object onto all the methods defined within the `Catalyst Controllers`. This context object is the gateway to everything within the framework. Through it, a controller can access: the request object, the response object, the session, the config files, the environment, other controllers, the models, the views, the components. The context object also provides flow control methods such as forwarding the process of the response into another controller action. Additionally, this context object provides a stash that is recycled every request. Inside this stash, the developer may insert whatever key, value combination he sees fit. The passage of contents between the `Catalyst Controllers` and the `Catalyst Views` is achieved through the usage of this stash. The `Catalyst Controllers` layer of our system is composed of four types of controllers which instanciate the framework ones: the `Catalogue Controllers`, the `Management Controllers`, the `Application Backbone Controllers` and the `Scheduling Controllers`.

The `Catalogue Controllers` are responsible to implement the required session forms by using either using the `Session Forms Engine` module, or by using pre-implemented `Session Forms Template`. Additionally, they are also responsible to expose the available actions to the outside. Lets assume that we have a service which is represented by a form containing two simple steps. Then, the corresponding controller has to expose actions which invoke the presentation of the first step, the submission of the first step, the goto of the first step, the presentation of the second step, the go back on the second step, the goto of the second step and the submission of the second step. The `Management Controllers` are responsible to perform the same set of actions but for the universe of forms generated through the `Action Forms Engine` module. The administrative forms of changing the attributes of a service would be an example of where these controllers are to be used. The `Application Backbone Controllers` implement the logic behind navigating between the different available pages within our web application, as well as, the default actions for events such as when an

error has occurred or a page was not found. It is through the implementation of these controllers that we are able to intercept errors which occurred and automatically report them as required in our supportability quality attribute scenarios. The `Scheduling Controllers` are responsible to expose the actions which are meant to be invoked in a scheduled manner. These actions are the endpoint which will be consumed by the `Schedule` component defined in the figure 2.2. In the appendix A, section A.1, code example 5, we have an excerpt of code which shows an example of a `Management Controller` definition.

2.5.8 Catalyst Views

At the topmost level our layered view, we have the `Catalyst Views` layer. One of the packages inside this layer is the `Template Toolkit`, which has a direct impact on how the `Usability-1`, `Usability-4`, `Usability-5`, `ModExt-3` and `ModExt-4` quality attributes scenarios are achieved. The `Template Toolkit` is a flexible and extensible template processing system. It can be applied in the generation of dynamic HTML, but it is not restricted to it. It is possible to define templates for mail messages and other mark-up language through it. Long story short, it provides syntax, directives, variables, virtual methods and filters which unlock the potential of iterating through stash passed objects, and generate dynamic documents based on them. These tools provided by the `Template Tookit` are thoroughly exploited in order to bring reusability to the views implemented by it. This package is implemented by dividing it into three distinct parts: the `TT Pages`, the `Layout` and the `Widgets`.

The `TT Pages` are direct implementation of the unitary pages to be served. For instance, we will have a TT page for each step belonging to a service form. We will have a TT page for each CRUD page which affects a defined data access object of our domain. The `Layout` defines the structure of the template layout. Technically speaking, the unitary TT pages mentioned before are but a piece of a much bigger layout. This way, it is possible to chunk the layout of our web design in many parts by segregating the responsibility of each one. This greatly improves the maintainability of our design and eliminates the necessity to repeat parts of it. For instance, within our current layout, we have the following divisions: header, navigation menu, caption, steps indicator, main content, footer and javascript. The header division defines all the necessary headers and Cascading Style Sheets (CSS) to be used. The navigation menu presents the dynamic navigation menus with all of the available options. The caption division presents a quick description of the current section. The steps indicator is only present in forms which are divided in many steps since it allows for the quick navigation between them. The main content division is where the processed TT pages are inserted. The footer division generates the footer part of the layout. The javascript division includes all the javascript libraries and code required by our system.

Although in the beginnings, the `Widgets` module did not exist, we ended up by introducing it within our system in order to improve the ease of modification of the system. The main idea behind this module is yet again the DRY ideology. When the portal was first in its prototype stage, it became clear that within each page there were parts of HTML code which were constantly repeated, suffering only little changes. It is common to see within a form, tags of the type input text. But, due to the

nature of our usability attribute scenarios, the way we implement our input text cannot be by simple using this HTML tags. Our input text must have a descriptive label, they may or may not have a sign indicating that the field is required, they may or may not have a description on why is it required, they may or may not have an error message which explains the last submission validation error, they may or may not have a tooltip which explains what type of content is expected within it, they may or may not have a place holder which provides visual cues of how to fill it, they may or may not have the last submitted value, they may or may not trigger a calendar which helps picking a date, etc. As we can see in this example, a traditional input text is completely buffed up within our web application version.

Consequently, it became relevant abstracting and encapsulating these structures, in order to create some entity capable of receiving a configuration through its parameters and generating the corresponding HTML boilerplate. These type of entities were named after `Widgets`. The `Widgets` are in their essence isolated TT templates which can be processed on demand. They expect a given configuration input to be passed, input which they will use in order to customize the HTML which they will automatically generate. This way, it is possible to enrich our universe of widgets easily, providing the system with the desired extensibility and modifiability. It also alleviates a lot the process of creating the unitary `TT pages`, since they simple consume these widgets. The readability of the code behind such pages is also greatly improved. There were developed more than twenty distinct widgets following this model, widgets which range from normal input text, input areas, select, radiogroups, checkboxes, checkboxes groups, to the automatic generation of tables with pagination and ordering. As a final remark on this subject, there was a moment in time, where we decided to move from an home-made interface, into a responsive interface powered by Bootstrap[32]. This conversion process was greatly sped up due to the fact that everything was built with widgets. Only two weeks were necessary from a single person, in order to convert more than two hundred distinct pages into Bootstrap. Great part of this conversion work was done by simply converting the widgets into Bootstrap. In the appendix A, section A.1, code example 4, we have an excerpt of a `TT page` being created while taking advantage of the existing `Widgets`.

2.5.9 Catalyst Plugins

The Catalyst framework allows for the extension of its behaviour through the creation of plugins. These plugins are capable of extending the context object methods and attributes, as well as, chaining actions before a request is passed into the controller and after the controller is done processing a response. As a matter of fact, the `Session` plugin takes advantage of the former, while the `LogFilter` plugin takes advantage of the latter. The `Session` plugin augments the context object attributes and methods by on every request checking if whether or not the session cookie is set. Then, if a valid session cookie is set, it proceeds to revive its content by fetching it from the persister. If no valid session cookie is set, then it proceeds to generate a new one, and to allocate the corresponding session in the persister. The `LogFilter` plugin intercepts the logging object from the framework in order to stop sensitive information from being flushed into the logs. This sensitive information is configurable by providing the corresponding specification at instancing time. Filtering the log is

extremely useful in environment where security is sought. If the application is running in debug mode, even the HTTP request parameters get dumped into the log. If the received corresponded to a login, then most likely, the password in clear text will dumped into the application log. Thus, log filtering becomes essential if we are looking to stop sensitive information leaking within our system.

2.6 Closing Comments

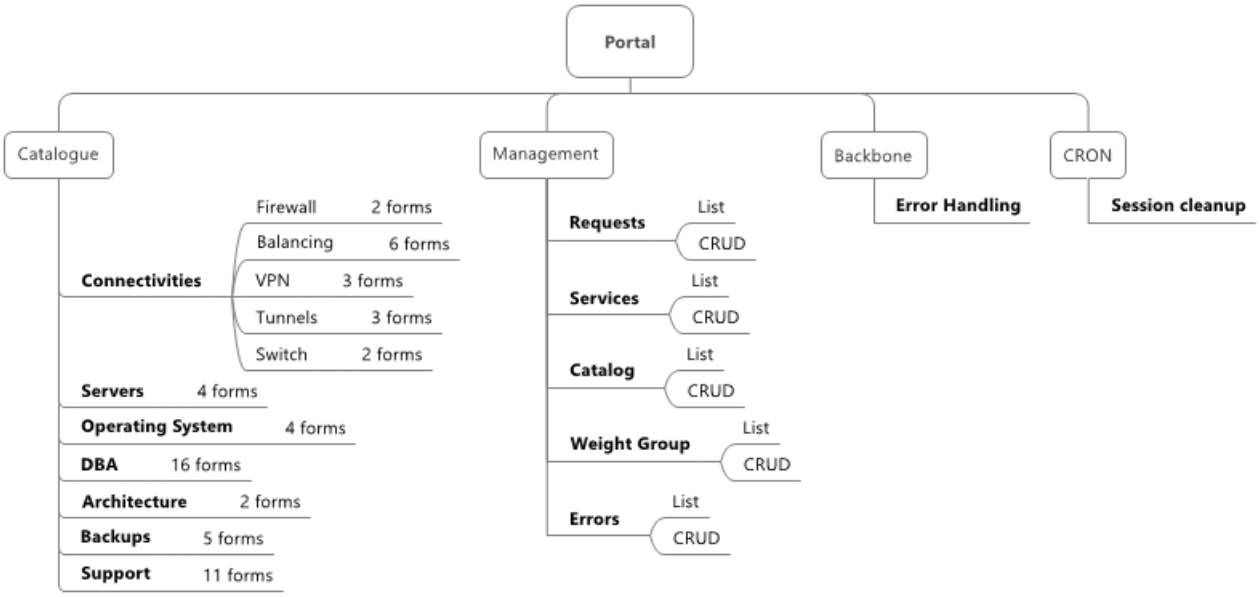

Figure 2.9: Phase 1 - Portal functionalities decomposition

In figure 2.9, we have a functionality decomposition of the features within the DC Portal in its early stages. Its features are divided into four major groups: the catalogue functionalities, the management functionalities, the backbone functionalities and the scheduling functionalities. The catalogue functionalities are composed by all the service forms which are available to a requester. The management functionalities are composed by the privileged operations which the operators and the developer of the portal may execute upon the web application. They involve listing and manipulating the underlying domain of our web application business logic. The backbone functionalities are composed by functionalities which are cross-wide to the whole application such as, intercepting unexpected exception which may occur and register them. Finally, the scheduling functionalities are composed by all the operations which will be consumed by the scheduling component. In the early stages, the only operation that fits such description is the session maintenance cleanup.

In section 2.2, we have defined scenarios for the sought quality attributes of our architecture. They were divided in five main groups: security quality attributes, modifiability & extensibility quality attributes, deployability quality attributes, supportability quality attributes and usability quality attributes. In table 2.1, we have defined our architecture security quality attribute scenarios. We shall now tackle on whether or not they are achieved by the proposed architecture, as well as, which part of the system is responsible for implementing them. All the scenarios between `Sec-1` and `Sec-8` are related to flow

of information between systems within variable environments and our system. They define whether the specific stimulus is to be allowed or not. The component responsible for controlling these flows is the firewall which was represented in figure 2.2 through the means of defined DMZ boundaries. Additionally, in the `Sec-10` scenario is also controlled by the firewall. The `Sec-11` scenario was achieved by a combination of the firewall and the portal database component, since it is possible to define access rules within its engine. On the other hand, the scenario defined within `Sec-10`, was achieved through the introduction of the `LogFilter` within our web application layers as shown in figure 2.4.

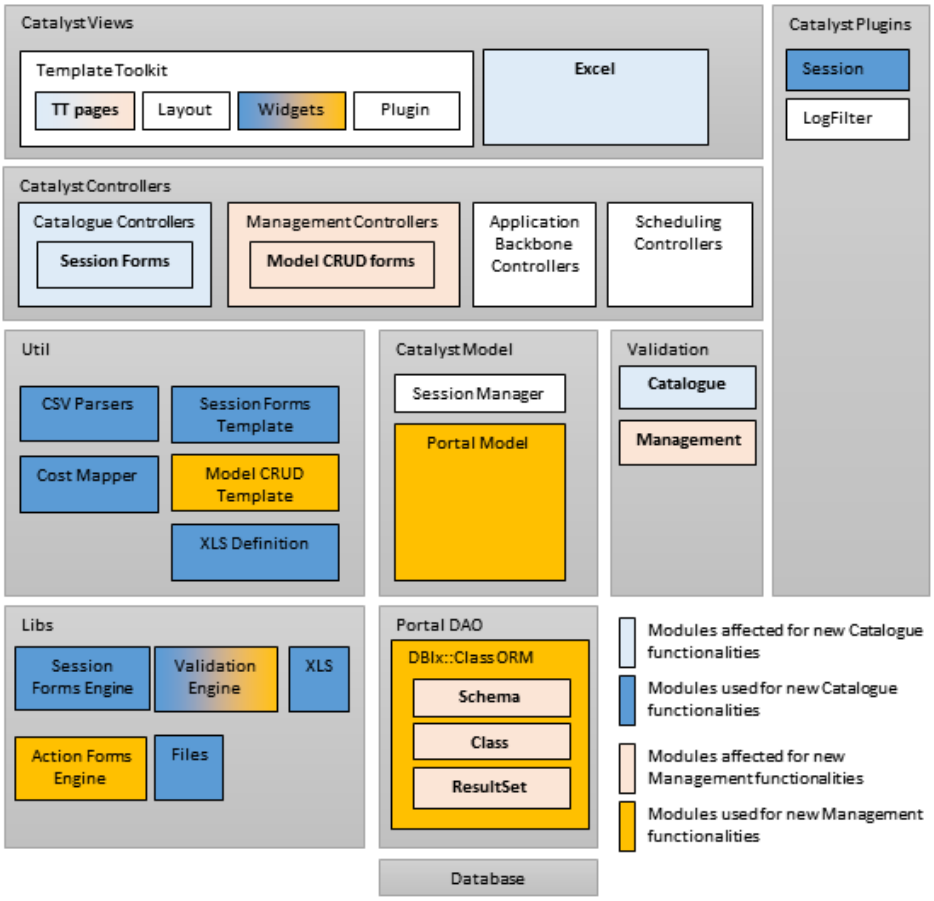

Figure 2.10: Modules responsible by the catalogue and management functional groups

In regards to the modifiability and extensibility scenarios defined in table 2.2, the scenario `ModExt-1` is guaranteed by the `Proxy` component which is capable of redirecting the traffic onto multiple web servers serving the same content. The rest of the scenarios are directly related on how flexible is to implement the functional requirements represented in figure 2.9. By examining the mentioned functional decomposition, it becomes clear that the developer most common task is to create or change features which fit within the catalogue and management functionalities. In figure 2.10, we have an example of on how is the presented functional decomposition mapped into our previously defined modules and layers. Thus, the scenarios `ModExt-2` and `ModExt-5` are granted by the underlying qualities of both `Session Forms Engine` and `Action Forms Engine`. The scenarios `ModExt-3` and `ModExt-4` are implemented by the flexibility created within the `Template Toolkit` with the creation of the `Widgets` package.

The deployability quality attribute scenarios defined within table 2.3, are achieved through a mix of the underlying qualities of our `Code Repository` and `Portal FE` components. The flexible set of functionalities provided by chosen technology for the `Code Repository` allow for the organization of the code base in such a way that respects the defined interactions within the given scenarios. Thus, these code repository interaction guidelines become a well enforced protocol which has to be followed by the developers. On the other hand, by taking advantage of the virtual hosting capabilities of the web server, and by taking advantage of the flexibility of the web application FCGI interface, it is possible to easily create the required environments and deploy new releases seamlessly.

For the supportability scenarios defined within table 2.4, we have the `Portal FE` component supporting the `Support-1`, `Support-2`, `Support-3` scenarios. The underlying web server and Catalyst MVC framework engine are capable of intercepting and logging wrong-doings as they happen in real-time. The last two scenarios, `Support-4` and `Support-5`, are not directly delivered by any of the components defined within our system, but rather by the qualities of the infrastructure where these components are deployed in. Our system was implemented following the very same processes which the data center portal is trying to computerize. This means that both backups and monitoring thresholds were configured when the infrastructure was delivered. Of course, these backups and monitoring are performed by another systems, but they are outside the scope of our project, thus we left them out when describing the components.

Finally, the `Usability-1` scenario from the usability quality attribute scenarios defined in table 2.5, was achieved by the `Layout` functionality implemented within the `Template Toolkit` views. The `Usability-2` quality scenario was possible by taking advantage off the provided behaviours within the `Session Forms Engine`. The `Usability-3` quality scenario was made possible by introducing the concept of session within our web application, thus turning our application away from its HTTP stateless form. The `Usability-4` and `Usability-5` scenarios were backed by both the `Session Forms Engine` and `Widgets` packages.

3

Authentication, Authorization, Auditing

Contents

Until this very specific moment, the system had been serving its users without any type of control. It had been functioning in an anonymous way where, whoever had access to the intranet, had access to consume the system functionalities with no trace. In terms of security, it was impossible to tell who did what and when. In terms of usability, it was impossible to keep track of a given user requests, since such concept did not exist. In the following sections, we shall detailed on how this problem was tackled.

3.1 Rationale

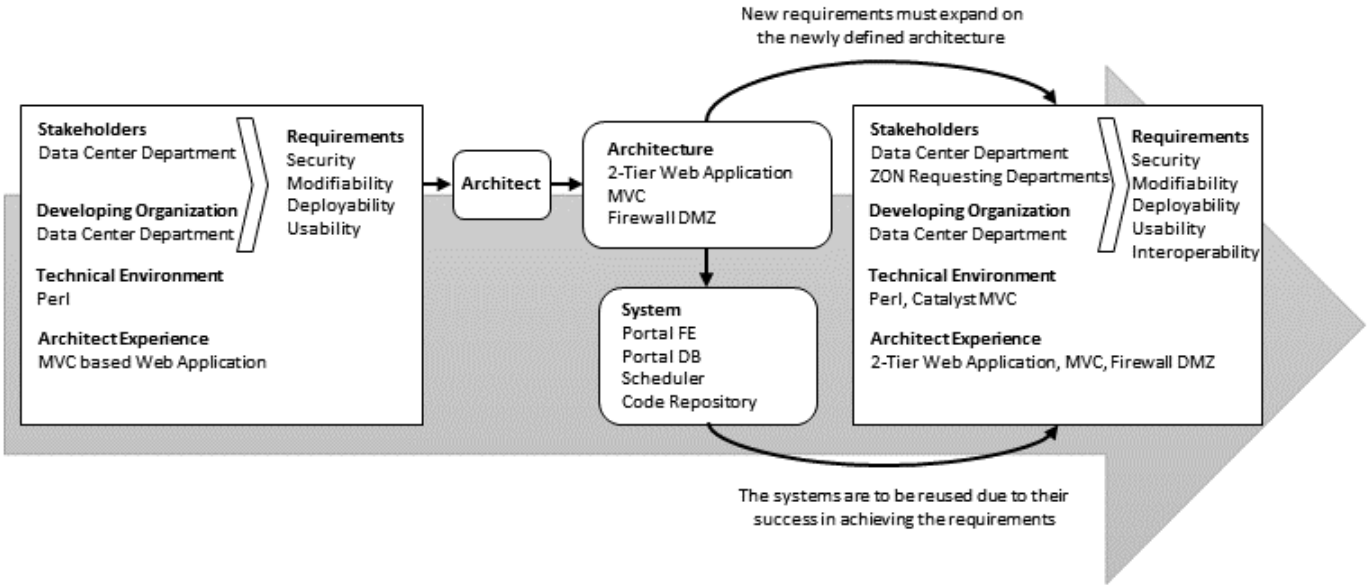

Figure 3.1: Architecture Business Cycle for the Authentication, authorization, auditing phase

The designing of an architecture is the result of many influences. The Architecture Business Cycle (ABC)[15] defines that the decision of a given architecture is influenced by the stakeholders, the developing organization, the technical environment and the architect experience. The stakeholders and the developing organization of a given organization influence the architecture in terms of its underlying requirements, which the architect has to map them into quality attributes. The technical environment introduces what one may see as, limitations or synergies, depending on the perspective, which the final architecture must take into account. For example, if the technical expertise of an organization is heavily leaned towards a given technology, it makes no sense for the architect designing a new architecture, to chose another technology, assuming that the required qualities can still be achieved with the previous one. Likewise, when presented with similar challenges, the architect will most likely follow successful approaches which he experienced before. Consequently, the choice of a given architecture is a conjugation of all these influences, and it impacts the way on which the new systems will be organized. One may think that the cycle ends there, but that is not the reality. Since, the newly designed architectures and systems, will influence the future iterations of the business and architecture. After all, the architecture becomes embedded in the organization, possible having its employees expertised towards the technologies behind it, and thus, mutating yet again all

the possible influences upon the architect.

In figure 3.1, we have applied the described Architecture Business Cycle into our reality. In the very beginning, there was only on stakeholder sponsoring this project, the data center department of ZON. This department sought to standardize its internal processes by improving the way on which it interfaced with the other areas of the organization. Thus, in short words, this stakeholders was looking for an attractive and user friendly interface to provide to its clients, flexible enough to iterate over new functionalities rather quickly, and capable of deploying new releases rather frequently. These general requirements easily map onto the usability, modifiability and deployability qualities, respectively. The security quality requirement did not appeared from any stakeholder requirement. Instead, it was related to the need of the system being in compliance with the underlying security architecture of the developing organization. Considering that the developing organization was also the data center department, and considering that part of the employees within the department had perl know-how, the perl programming language became one of the main influences irradiating from the underlying technical environment. Additionally, the architect of this system had only experience with MVC based web applications, in order to implement similar systems.

Thus, the architecture, which the architect designed having all these influences in mind, resulted in a 2-Tier MVC Web Application contained within a DMZ, in general lines. Architecture which resulted in four distinct systems as we saw before: the Portal FE, Portal DB, Scheduler and Code Repository. As we shall witness, this previously defined architecture and these previously defined systems will now influence, our next iteration upon the architecture. The first remarkable difference worth mentioned, is that the data center department is no longer the only stakeholder. The other departments of ZON which now became the main users of the portal also are. Obviously, their main concerns are related to usability requirements. They want to be able to track their own requests and the requests of their own departments. On the other hand, the data center department wants to be capable of controlling who has access to the portal and manage what they can consume within the portal. Thus, they want to be capable of authenticating, authorizing and auditing the users of the portal. Simultaneously, they want these processes to be easily manageable through the portal interface. All of these requirements have to be achieved within the limitations introduced by the influence of previous architecture and systems. Therefore, the architecture which will be presented in the following sections, will be identical to the one which was previously introduced, simply augmented on and from the necessary systems, in order to achieve the desired qualities.

3.2 Quality Attribute Scenarios

In table 3.1, we have defined the concrete quality attribute scenarios introduced for the authentication, authorization and auditing phase of the architecture. In the security department, they are mostly aimed at introducing the mechanisms which allow the system to be capable of authenticating users and authorizing them while keeping a trail of that. In the supportability department, we have one scenario that is related to the capability of easily auditing the system in real-time. In terms of deployability,

Scenario	Source	Stimulus	Artifact	Environment	Response	Measure
Security Sec-12	Unauthenticated portal user	attempts to access the authenticated only services	Portal	DMZ	The access denied until he authenticates himself	The information about the attempt is logged for auditing
Security Sec-13	Unknown source	Attempts to look inside the content of the established connections	Portal	DMZ	The exchanged information is encrypted	Breaking the encryption is not possible with the current technology
Security Sec-14	Authenticated Portal user	Attempts to access to a service which his group is not allowed to access	Portal	DMZ	The system denies the access and logs the attempt.	The block is correctly performed and logged for later auditing
Supportability Support-6	Developer Operator	Wishes to audit the history of users authentications and authorizations	Portal	Any	Information about these accesses is maintained within a privileged view in the system, only accessible by them	The audit information is updated in real-time
Deployability Deploy-6	System Operator	Wishes to deploy a new release which introduces new controllers	Portal	Any Environment	Authorization entries for the new controllers are automatically populated within the domain	No time is spent on exposing to the web application the new functionalities
Usability Usability-6	Operator	Needs to be able to efficiently manage groups of users and their authorizations	Portal	Runtime	The portal allows to perform aggregated operations upon the users and the authorizations	The time taken to manage a group of one person or a group of many is approximately the same
Interoperability Interoperability-1	Portal	Wishes to authenticate its users	Active Directory	Any	This authentication is done through integration with existing system of the organizations rather than creating a new mushroom to provide such	The system is capable of adapting in order to support interoperability with such type of systems

Table 3.1: Quality attribute scenarios - Authentication, authorization and auditing phase

we have a scenario related with automatically generating the ACL objects to control the accesses of controllers which appear in new releases, avoiding having to manually create them. When it comes to usability, the main concern of our architecture must be to provide the means to easily manage the access control lists of a given group of users. Finally, the interoperability category of quality attributes was introduced, which is related with the ease of integrating our system with existing systems. In this case, our system is not meant to create a new authentication mushroom, but instead, to take advantage of the existing central authentication system within the organization.

3.3 Architectural Components

In figure 3.2, we have representation of the new architectural view of our system components. In this phase, the architecture did not change much compared to the previously one, since the most important components were already introduced in our system. Nevertheless, one of most noticeable differences is the introduction of the `AD` component. The `AD` component exists within a different DMZ since its ownership does not belong to our system. This component is a pool of Active Directory servers maintained by another department within our organization, which provides the user repository and authentication for the whole company. The architects whom implemented that system inside our organization, probably had in mind availability scenario, thus, the fact of it being a pool of active directories. This component was adopted into our architecture in order to comply with the following scenarios: `Sec-12` and `Interoperability-1`. The strategy implemented is the one where every time the `Portal FE` needs to authenticate a given user, it passes the received credentials on to the `AD`

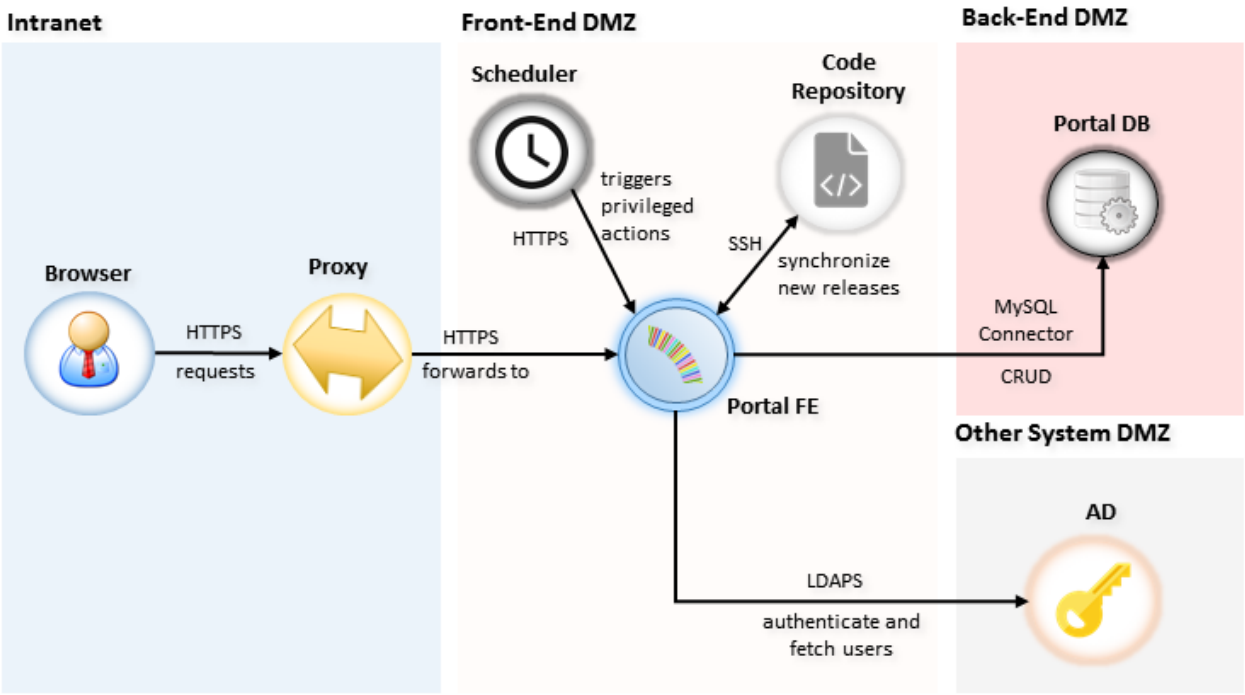

Figure 3.2: Architectural components for the authentication, authorization and auditing phase

through the `LDAPS` protocol. If the `AD` returns a positive response towards the authentication, then the `Portal FE` proceeds to fetch from it information about the user such as its username, name, email and deparment, updating this one a local copy maintained within the `Portal DB`. Another difference that can be spotted is the fact that all the HTTP connections changed into HTTPS connections. Security scenario `Sec-13` documents this change since sensitive information such as user passwords may now travel through the channels. The rest of the quality attribute scenarios were achieved by changes changes within the `Portal FE` and `Portal DB`, as we shall describe inside the following sections.

3.4 Module View

In figure 3.3, we have updated the previously presented layered module view of our architecture, with the necessary additions to empower our quality attribute scenarios for this phase. In order to implement the `Sec-12` and `Sec-14` scenario, we took advantage of the `Catalyst Plugins` by consuming an `ACL Engine` and an `Authentication` module which allow for us to introduce means to intercept those activities with behaviours implemented within our business logic. Thus, supporting the newly loaded `Authentication` plugin, we created our own `Auth` layer which deals with the hassle of interacting with the `AD` component through `LDAP` connections. This specific logic implements the operation of authenticating a given user against a pre-defined pool of active directories, and the operation of fetching the user information to update within our data access objects. It is also within this layer, where successful and failed authentication attempts are registered in the corresponding auditing table. The `Access Control` layer was created in order to provide the specific validation behaviour onto the corresponding `Catalyst Plugin`. The behaviour defined within this layer is responsible for both

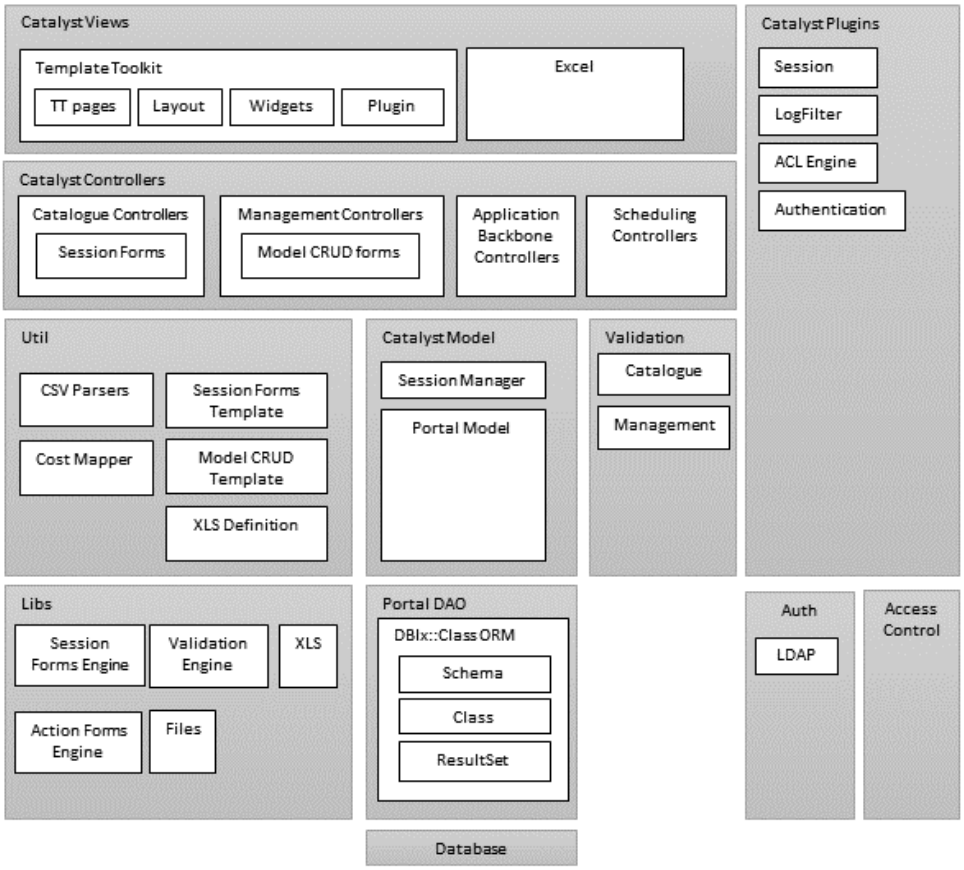

Figure 3.3: Layered module view for the authentication, authorization and auditing phase

populating our ACL domain, as well as, enforcing the defined access rules, by reading and writing the corresponding newly created classes within the `Portal DAO`.

In figure 3.4, we have the architectural specific tables that had to be added into our system in order to support the specified quality attribute scenarios. We have defined that the ACL rules must be specified on the granularity level of a `Catalyst Controller`. For instance, if we consider a controller that implements a five step service form, the ACL rules which may be defined can only go to the granularity of, either the user has access to all the steps or to none of them. This level of granularity is specific enough to have a strong control on the accesses within our portal, while being coarse enough to be easily manageable by the operators. At bootstrap time, the `ACL Engine` transverses the tree of controllers, looking for new ones which have not yet been mapped into the database. It maps the controllers within this situation into the table `Action`. It is through this method that the `Deploy-6` deployability scenario is collimated. Controllers mapped within this table will automatically appear in the corresponding access control interfaces which the operators will use to provide accesses to groups.

Then, the engine has the notion of groups. These groups are mapped into the table `Role`, and they are basically an agglomerate of users. Obviously, a given user may belong to distinct groups, being this mapping under the responsibility of the operators. The access control is done on the level of the groups. This means, that a given user can access whatever it is permitted for its given groups.

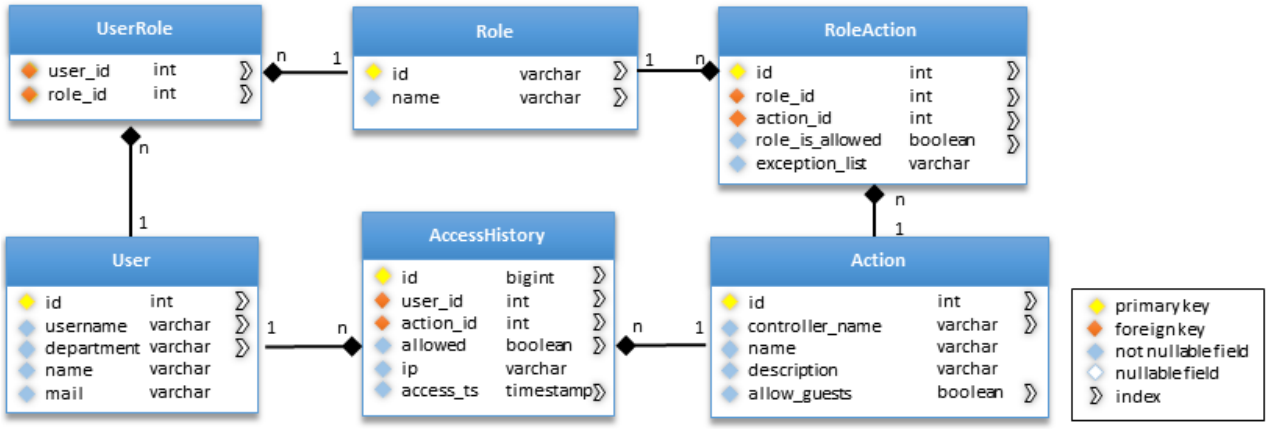

Figure 3.4: Database schema for the ACL engine

Thus, it is possible to create relationships, where we define that, a given group can access or cannot access, a given set of actions. Additionally, it is also possible to define an exception list which contains the users to whom the defined rule applies by the negative. Supporting the `Sec-14` scenario, we have the table `AccessHistory`, which records all these resolutions. Whenever an ACL rule is applied, its details and its final resolution is registered within this table, along side with a time stamp. This way, it becomes possible to audit the system and understand if someone is having access to resources it should not have.

Now that we mentioned the main additions into our layered module view, it is important to also mention that the already existing layers and modules were also affected in this phase. For instance, the `Portal DAO` had to be augmented to include the changes to the new database schema. The `Application Backbone Controllers` had to be added of the controllers which implement the login and logout functions. The `Management Controllers` had to be added of the controllers which implement the managing of the ACL system. Consequently, within the `Catalyst Views`, the corresponding `TT Pages` had to be implemented. All these additions do not affect the underlying architecture, they assent themselves on the qualities which we introduced on the previous architecture iteration. As a matter of fact, they demonstrate on how easy it is to modify and extend our system in order to quickly iterate over new functionalities.

3.5 Closing Comments

In the previous sections, we went through the necessary architectural changes which directly tackled the defined quality attribute scenarios for this phase. In figure 3.5, we have a more practical representation of all the functionalities available within the portal by the end of this phase. Needless is to say that these functionalities were implemented within the boundaries set by our architecture. We can also see that comparing to the previous phase, most of the functionalities added on this phase had more to do with the new components added to our architecture. It is important to mention that, with the introduction of the `AD` component within our architecture, our architecture has become heavily depend on it. Although, this component is redundant enough, should all the instances of its pool fail

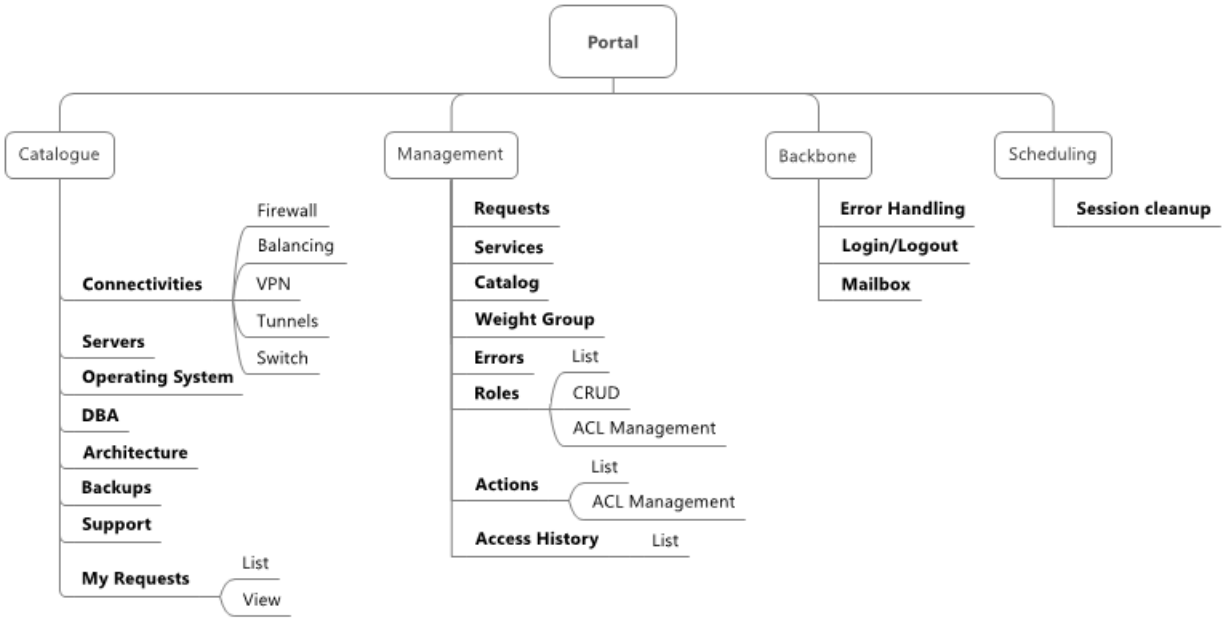

Figure 3.5: Functional decomposition view for the authentication, authorization and auditing phase

or malfunction at the same time, and our system entry will be effectively blocked, even though the service is up. A failure in the `AD` component implies that, users attempting to log in our system on that given moment, are unable to do such.

Also, on the granularity level chosen for the management of ACL rules, we have implications on the way which developers must implement new controllers. Since the defined ACL behaviour, only allows the specification of rules on the controller level, it becomes impossible to block the access of a given method within a given controller. Unless, of course, the developer implements the controller in such a way that a given block is hard coded in the controller, instead of being controlled by the architecture of the system. Relatively to the scenarios which we barely mentioned, the `Sec-13` was easily achieved due to the previously chosen web server capability of switching from HTTP to HTTPS easily. In regards to the `Usability-6` scenario, it was easily achieved through: the extension of the data access objects methods in order to support aggregated operations, implementation of form behaviours which support such, implementation of interactive widgets which facilitate these manipulations.

4

Workflow & Integration Phase

Contents

Having the mechanisms of authentication, user, group, ownership, access control and auditing in the previous iteration of the architecture, the focus of our next efforts turned into elimination of the manual processes forced upon our users, which enforced complying with the ITIL framework at the cost of duplicate efforts. Until this very moment, our clients had to create a request within our portal application, and than manually open a service request within the official ITIL tool, HPSM. In the following sections, we shall detail on the implementation of such workflow and integrations were achieved.

4.1 Rationale

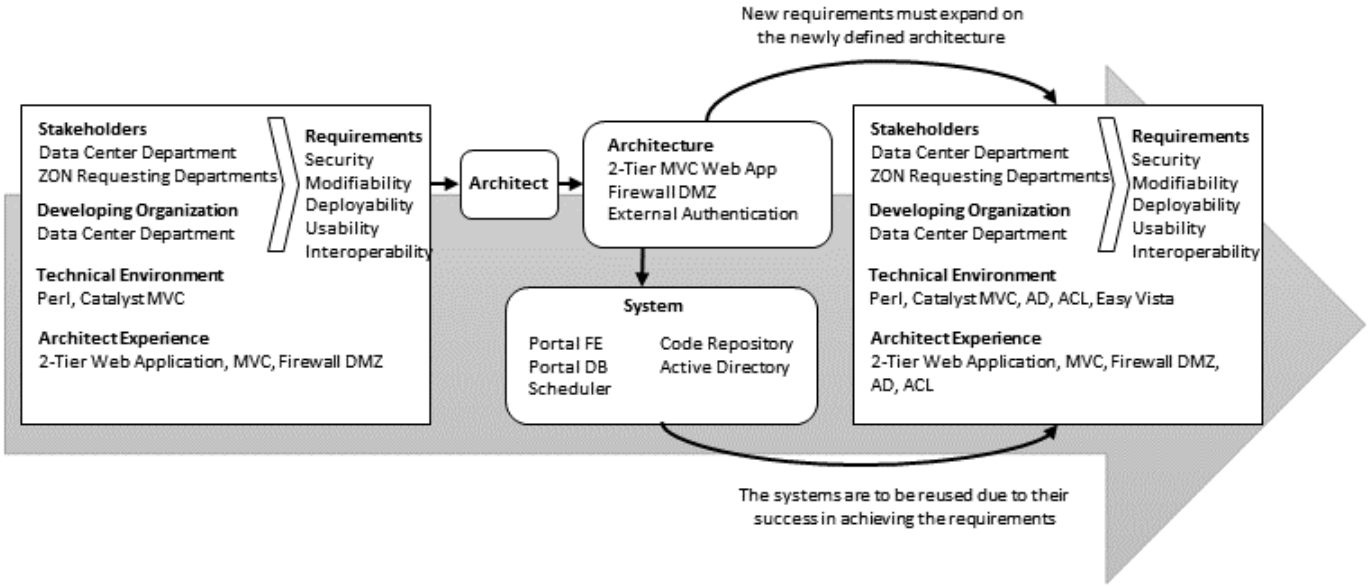

Figure 4.1: Architecture Business Cycle for the Workflow and integration phase

The stakeholders of our system were looking at three major requirements for the next iteration: introduction of observable life cycles within the portal entities such as requests, automatically map portal requests into HPSM service requests in order to eliminate duplicated efforts, and replacing HPSM with Easy Vista as the chosen internal task manager tool. In figure 4.1, we have represented our current situation, in terms of ABC. The technical environment of our system changed, since now we have the means to authenticate and authorize our users. The requirements demanded by our stakeholders implicate iterating yet again the current architecture, in order to expand on the security, modifiability, usability and interoperability scenarios. If implemented correctly, this phase will greatly influence the way our organization functions. Taking away from the users, the responsibility of mapping requests in both tools, not only facilitates their daily life, it also introduces great efficiency gains and errors reduction. We shall now detail in the following sections, what was sough in terms of architectural changes at this stage and detail on how it was achieved.

4.2 Quality Attribute Scenarios

Scenario	Source	Stimulus	Artifact	Environment	Response	Measure
Security Sec-15	User	Attempts to perform an unauthorized action within a defined workflow of a given domain entity (eg. requests, services, ...)	Portal	DMZ	The attempt is rejected	The rejecting conditions are correctly evaluated by the system
Supportability Support-7	Developer Operator	Wishes to audit the history of recent invocations performed by the system, in order to verify its normal functioning	Portal	Any	Information about these accesses is maintained within a privileged view in the system which contains all the relevant information of previous invocations	The history of invocations is generated in real-time
Modifiability ModExt-7	Developer	Wishes to add/change/remove already defined states, actions, conditions from a given workflow skeleton	Portal	Design Time	Correctly identifies and alters the corresponding definition without affecting other workflows definitions	Changes to the workflow skeleton have a small complexity
Modifiability ModExt-8	Developer	Wishes to create a new type of entities for the domain of the application	Portal	Design Time	He must implement the corresponding workflow for that entity	The newly defined workflow is isolated from the others
Modifiability ModExt-9	Developer	Wishes to add/change/remove new integration services	Portal	Design Time	Correctly identifies and alters or creates the corresponding modules without affect existing integration services	These integration services can be easily reused by any Perl application
Usability Usability-7	User	Needs to be able to efficiently perform the different actions of entities workflows	Portal	Runtime	The portal provides intuitive widgets which expose with clarity the workflow state and available actions	Interactions with portal entities successfully became standardized in the user interface point of view
Usability Usability-8	User	Wishes to receive important feedback through email notifications	Portal	Runtime	Notifications email are sent to the user when important actions affect entities which are from the user interest	The regularity on which the user needs to consult the portal is reduced
Interoperability Interoperability-2	Portal	Wishes to mirror its ITIL manipulations	HPSM	Any	This integration is performed through the provided HPSM Application Program Interface (API)	Relevant entites are mapped in order to enforce the ITIL compliance
Interoperability Interoperability-3	Portal	Wishes to notify about its requests manipulations, and vice versa	Easy Vista	Any	This integration is performed through the provided Easy Vista and Portal FE API	Both Portal FE and Easy Vista have the same view of the state of the requests
Interoperability Interoperability-4	Portal	Wishes to notify the users about important events	SMTP	Any	This integration is performed through the SMTP protocol	Email messages are correctly delivered to its users
Interoperability Interoperability-5	Portal	Wishes to invoke web services	Systems	Any	This invocation is done in an asynchronous and fault tolerant manner	Invocations are only completed when the target system positively acknowledges

Table 4.1: Quality attribute scenarios - Workflow and integrations phase

In table 4.1, we have defined the concrete quality attribute scenarios introduced for the workflow and integrations phase. In the security department, they are aimed at introducing mechanisms which extend the access control to the level of actions through the conditions of workflows. In the supportability department, we have a scenario related to the capability of auditing the system remote invocations in real-time. In terms of modifiability, we have scenarios related with the ease of modification and extension of both workflows and integration services. When it comes to usability, we are concerned with providing intuitive and ease to learn interfaces for the whole workflow process, and introducing ways of reducing the time a given user has to spend within the portal. Finally, related to the interoperability department, we have scenarios related to the introduction of integrations with the HSPM, Easy Vista and Simple Mail Transfer Protocol (SMTP) systems.

4.3 Architectural Components

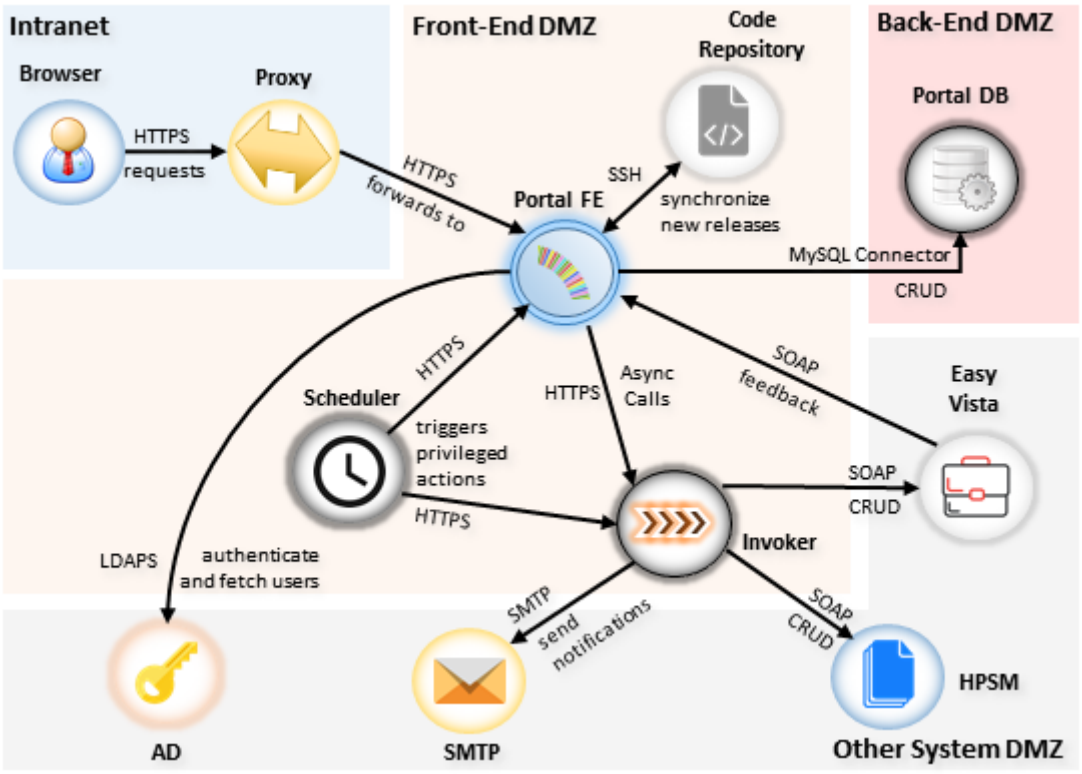

Figure 4.2: Architectural components for the workflow and integrations phase

In figure 4.2, we have the representation of the resulting architectural components view of this iteration. While most of the requirements related to the introduction of a workflow engine were done by changing the existing `Portal FE` and `Portal DB` components, the same does not apply to the integration requirements. In order to comply with the `Interoperability-5` scenario, which specifies that invocations must be done in a tolerant and asynchronous manner, the `Invoker` component was introduced. This component acts on our architecture as a message queue capable of maintaining the messages order, if required. Its goal is to receive the serialized services to be invoked from the `Portal FE`, and perform them asynchronously, freeing the `Portal FE` to serve other requests. When it finally obtains the result to an invocation, it executes the callback which was defined within the serialized service. If the invocation fails due to a fault, this component is responsible to persist the invocation and keep re-invoking it until a valid response is returned. The pace on which this component retries its invocations is defined by the `Scheduler` component. Following the `Portal FE` success, the `Invoker` was also implemented as a Catalyst MVC web application. Thus, the `Scheduler` triggers the retrying of invocations by accessing a privileged Uniform Resource Locator (URL) within the `Invoker` application, in a timely manner.

Outside of our jurisdiction, but now part of our architecture, we have the `SMTP`, the `HPSM` and the `Easy Vista` components. These components are the target systems, to where the `Invoker` may dispatch its invocations. The `SMTP` component is responsible to answer to part of our `Usability-8` and `Interoperability-4` scenarios. It provides a way to the portal send mail notifications to its

clients. The HPSM component is responsible to answer to part of our Interoperability-2 scenario, which requires the portal to be capable of mapping the state of its entities within the tool HPSM through web-services. Finally, the Easy Vista component is responsible to answer to part of our Interoperability-3 scenario, which specifies that the portal must be capable of notifying our newly adopted task manager, that new requests arrived, and therefore, tasks must be dispatched to the responsible teams. Additionally, the Portal FE needs to provide a web service interface for the Easy Vista component to consume, so that it can notify the portal back, when the underlying tasks of a given request have been completed. This communication was implemented through Simple Object Access Protocol (SOAP)[33], due to constraints introduced by the Easy Vista which is now part of our technical environment.

4.4 Module View

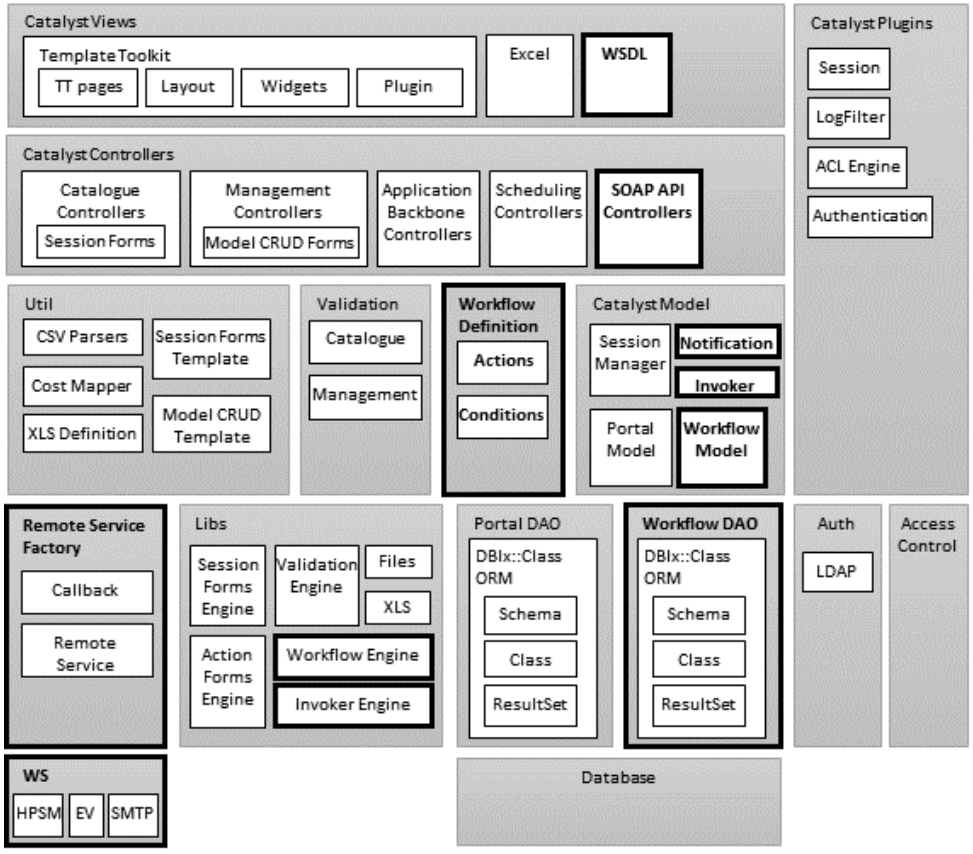

Figure 4.3: Layered module view for the workflow and integrations phase

In figure 4.3, we have highlighted the changes which affected our layered module view on this phase. They are mainly related with the introduction of both an workflow and an invoker engine, and exposing these newly introduced engines to the underlying MVC framework.

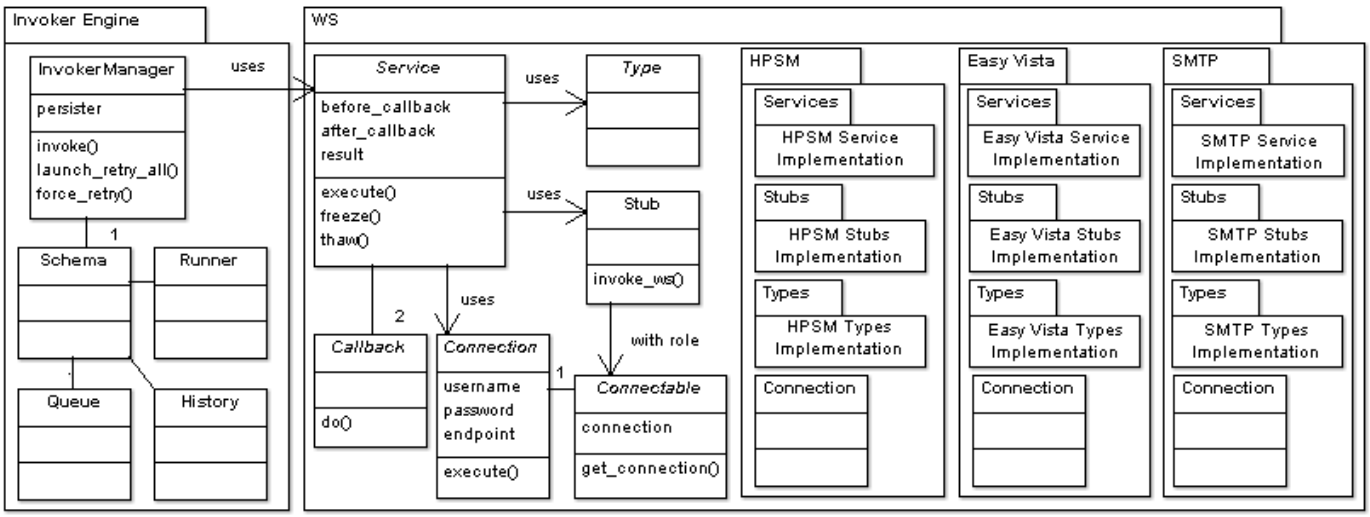

Figure 4.4: Package class diagram of the Invoker engine

4.4.1 WS

This newly introduced layer within our architecture, provides us with the necessary tactics in order to achieve the `ModExt-9`, `Interoperability-2`, `Interoperability-3`, `Interoperability-4` and `Interoperability-4` scenarios defined in table 4.1. In terms of modifiability, the encapsulation of these web services are segregated per target system to be invoked. Thus, creating or changing the behaviour of the services which integrate with a given system, can be done in a very straight forward and modular manner, without affecting the other existing integration packages. In figure 4.4, we can see that our `WS` module defines five main abstract classes which developers implementing new integrations service must extend: `Service`, `Connection`, `Stub` and `Type`.

In terms of modifiability and interoperability, these classes are then implemented by system specific packages such as the represented `HPSM`, `Easy Vista` and `SMTP` packages. Within each one of them, we have specific implementations for the `Service`, `Stub`, `Type` and `Connection` classes. The `Connection` class is responsible to define how the target system web services are invoked. For instance, in the HPSM and Easy Vista systems, they are invoked through a SOAP engine, while for the SMTP system, they are invoked through the SMTP protocol. The `Connection` class encapsulates this communication behaviour. The `Service` classes encapsulate specific stubs calls, along-side with before and after invocations callbacks. These are the classes which will be serialized and travel from the `Portal FE` into the `Invoker` component, in order to request a remote web service invocation. The before and after callbacks, are responsible to define what has to be performed before actually invoking something, and after receiving a response. The `Stub` classes are responsible to expose the available remote API and implement their invocation and response handling. Finally, within the `Type` classes, we have the encapsulation of the objects which can be exchanged in the invocations. These objects are responsible to deal with their internal data marshalling.

4.4.2 Invoker Engine

This engine runs within the Invoker component rather than within the Portal FE. Since the Invoker component is also a Catalyst MVC web application, we will describe the Invoker Engine within this section, instead of presenting yet another layered module view which will be very identical to the one presented in figure 4.3. Thus, the Invoker Engine is a key module in the tactics necessary to achieve our interoperability scenarios. It deals with all the logic behind the reception Service objects and invokes them in the respective systems. In figure 4.4, it is represented by the Invoker Manager module. When it receives a new service for invocation, it queues it within the defined schema Queue. After the Scheduler component triggers the retrying of all the invocations, the Invoker Manager launches a new Runner which keeps tracks of how many invocations it had pending and how many it dispatched on a given run, and invokes every queued Service. For every queued Service, it rebuilds the service object, calls the defined before callback, invokes the external system, calls the defined after callback and registers the invocation status inside the History schema. Thus, not only all the interoperability scenarios are partly implemented by this component, but also the Support-7 supportability scenario is based on it.

4.4.3 Remote Service Factory

This layer contains packages which define concrete implementations of callbacks and remote services which are application specific. The main idea behind it, is that the Callback implementations are domain specific and thus should be independent of the respective external systems where they will be used. Thus, the Remote Service Factory is responsible to instanciate the general services defined within the WS layer with concrete values and callbacks that implement the integrations sought by the Portal FE component. An example is, create a service which tells HPSM to update a given service request with information provided by the portal itself. Thus, the Remote Service Factory layer also plays a major role in the implementation of the ModExt-9 scenario within the architecture. In the appendix A, section A.2, code example 11, we have the example of a concrete implementation for a Remote Service Factory, while on code example 12, we have the same but for a Callback.

4.4.4 Integration related modifications

To bring a closure to the modifications done, in order to achieve the desired integration requirements, we have additional changes performed inside the Catalyst Model, Catalyst Controllers and Catalyst Views layers. Two new catalyst components were defined within the Catalyst Model layer. The Invoker catalyst component is responsible for establishing the HTTPS interface with our architectural Invoker component. Whenever our logic required to send a remote service created through Remote Service Factory, it will have to be sent through this Invoker catalyst component. On the other hand, the Notification catalyst component is responsible to create the dynamic content of the notification messages necessary to implement the Usability-8 scenario. Finally, the SOAP API Controllers within the Catalyst Controllers layer, and the WSDL within the

Catalyst Views layer, were added to our architecture, in order to provide the required integration from the Easy Vista component towards our Portal FE component.

4.4.5 Workflow Engine

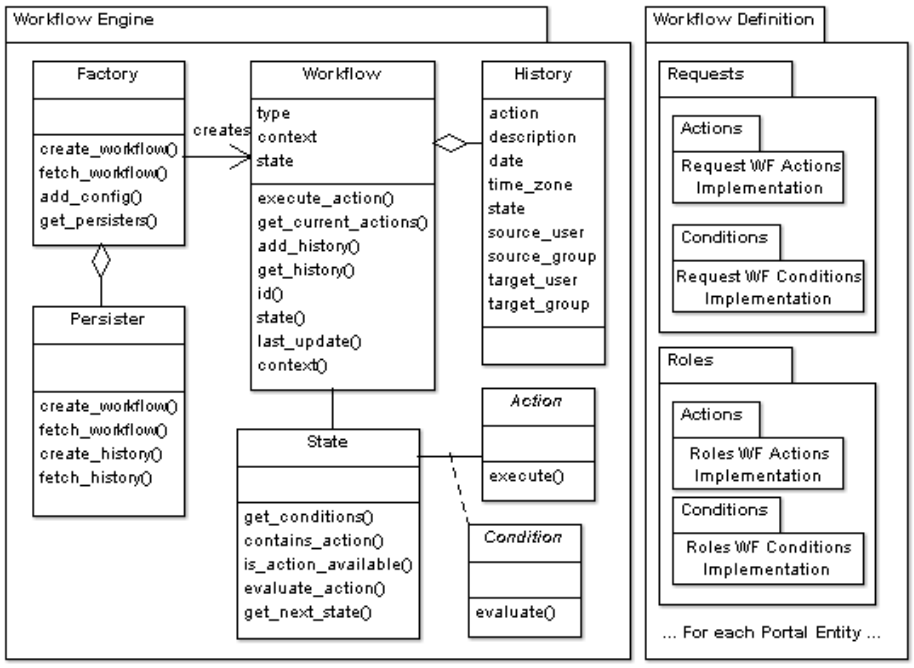

Figure 4.5: Package class diagram of the Workflow engine

The workflow engine solution which was implemented within our Portal FE was not developed from scratch. Instead, an open source perl library named Workflow[34] was used in order to buff our system with workflow capabilities. This engine was simple and yet flexible enough to introduce the desired qualities within our system. The quality attribute scenarios which are directly affected by this module are the Sec-15, ModExt-7 and ModExt-8 scenarios presented in table 4.1. In figure 4.5, we have a representation of the most important classes within this module. The Factory class is responsible to load all the Extensible Markup Language (XML) configuration files which define the workflows skeletons, states, conditions and actions, and create on demand workflows which represent those definitions. In the appendix A, section A.2, code examples 6, 7 and 8, we have concrete examples of how are these configuration files defined. The Persister is responsible for implementing the behaviour of persisting those workflows within a persistent layer. In our case, we extended this class in order to be able to persist workflows through our ORM layer, represented in the Workflow DAO layer. The Workflow class encapsulates the logic of one workflow in an object oriented way. It contains the workflow current state and a workflow context which will be used to maintain information and passed them on to actions and conditions. The State class encapsulates the workflow state itself. It contains a set of actions which may become available depending on whether or not the corresponding conditions are met. The behaviour of this specific conditions is implemented by extending the Condition class. The behaviour of the available actions can also be implemented by extending the Action class.

4.4.6 Workflow Definition

In figure 4.5, we have also a package that represents the `Workflow Definition` layer within our system. The packages inside this layer are responsible to implement specific behaviour for both the actions and conditions of the workflow, by extending the `Action` and `Condition` classes, respectively. The `Sec-15` scenario is directly related to the implementation of these `Conditions` extensions. The `ModExt-8` is directly related to the ease of creation of these `Actions` and `Conditions` extensions for a given portal domain entity. The `ModExt-7` is achieved by the ease of creation and modification of the corresponding workflow configuration files which create the workflow skeleton for a given portal domain entity. These XML configurations files define, for each workflow type, what are the available states, what are the available actions within each state, what are the conditions which must be met for each action, and what are the classes which implement these actions and conditions. In the appendix A, section A.2, code example 9, we have an example of a concrete action implementation, while on code example 10, we have the example of a concrete condition implementation.

4.4.7 Workflow related modifications

Additionally to what we have mentioned before, a new `Workflow Model` was added into the `Catalyst Model` layer in order to expose the newly created `Workflow DAO` layer. And, within the `Catalyst Views` layer, the list of available `Widgets` was extended to include widgets which empower our `Usability-7` scenario, by providing easy to learn, intuitive and reusable workflow interactions widgets. Since all the portal domain specific entities started being supported by this workflow engine, reusing the newly created widgets made it easier for the user to learn how to manipulate our domain specific entities, since the process became standardized and shared among all of them.

4.5 Closing Comments

In the previous sections, we went through the necessary architectural changes which directly tackled the defined quality attribute scenarios for this phase. In figure 4.6, we have a more practical representation of all the functionalities available within the portal by the end of this phase. Although, our architecture became dependent of third party controlled components, such as `HPSM` and `Easy Vista`, the way we have implemented these integrations guarantees that our system will continue to function normally in the event of one of them failing. This is mainly because our integrations are done in an asynchronous and fault tolerant manner by the `Invoker` component, which provides guarantee that our invocations will be answered somewhere in the near future when the systems come back online. On the other hand, the workflow engine introduced an elegant manner of defining and controlling behaviour within our domain entities, behaviour which is dependent on the entities current state, opening a whole new world of possible business logics.

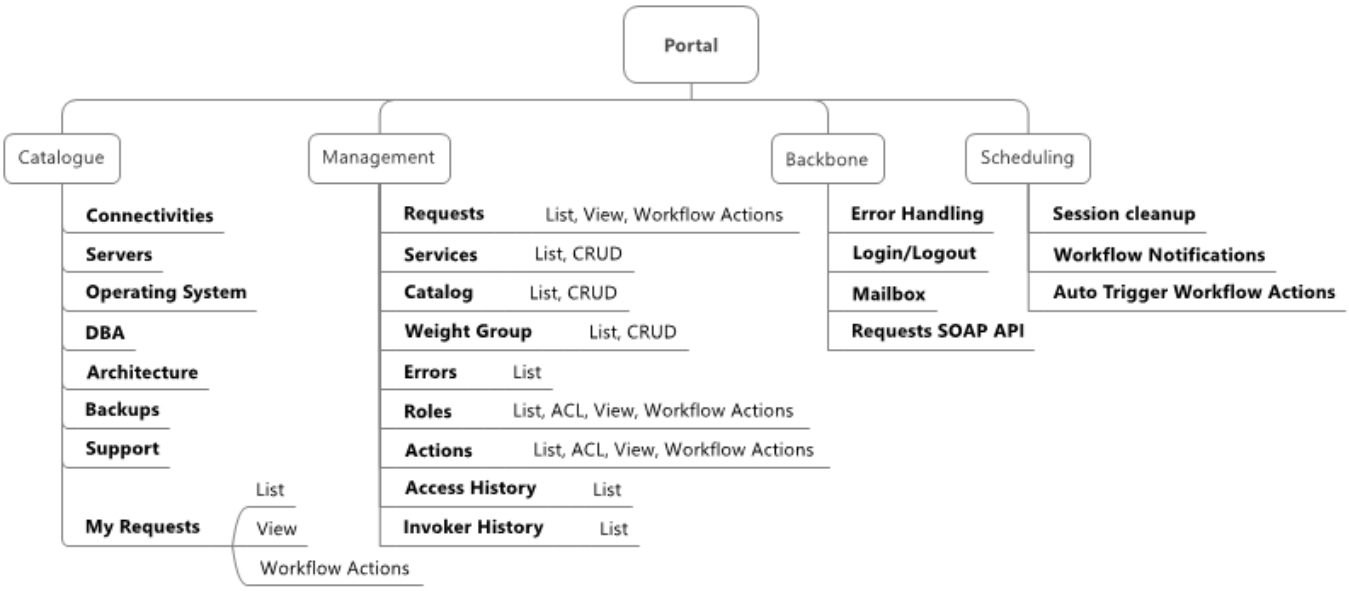

Figure 4.6: Functional decomposition view for the workflow and integrations phase

5

CMDB & Task Management Phase

Contents

Having ZON merged with Optimus under the name of NOS, and having its internal data center teams still functioning in distinct ways, it became necessary to merge most of what was possible from both worlds processes inside the data center portal. At the architectural level of our system, that meant adopting an ex-Optimus CMDB and replacing the HPSM CMDB as the main reference for configuration items, while keeping the HPSM CMDB synchronized at the same time. It also meant to remove the Easy Vista, out of our architecture since, when weighting its functionalities and licensing cost, against the cost of developing those very same functionalities in-house inside the data center portal, the cost reduction and benefits of bringing its functionalities in-house was just too great to ignore. Thus, in the following sections, we shall detail the final iteration upon our architecture.

5.1 Rationale

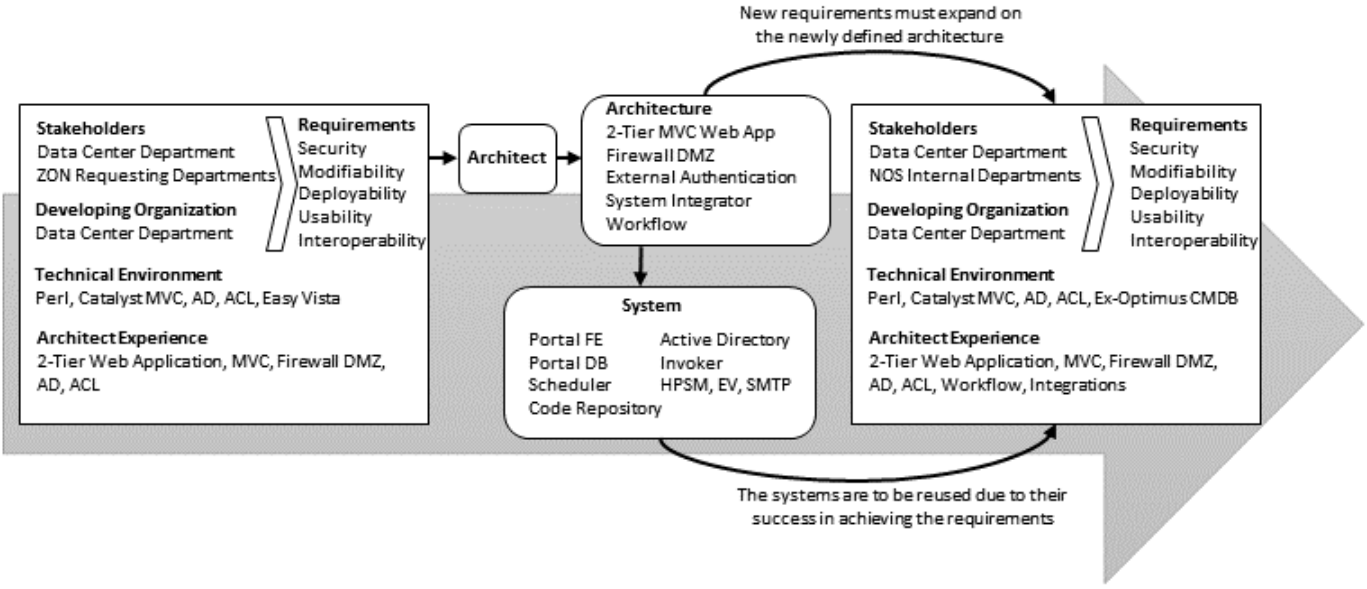

Figure 5.1: Architecture Business Cycle for the CMDB and task management phase

In figure 5.1, we have represented this phase Architecture Business Cycle. In the previous iteration, we have introduced a couple of new systems to our architecture such as the Invoker, the HPSM, the Easy Vista, the Mail server known as SMTP. We also embedded our Portal component with a workflow engine which we have used to empower our domain specific entities with states and workflow capabilities depending on those states. These previously introduced capabilities in our architecture, shall now be the catalyst to achieve our new set of requirements imposed by our stakeholders. Stakeholders which are no longer only associated to ZON, being now associated with NOS instead: the merging of all ex-Zon and ex-Optimus requesting departments into a organization. Thus, our technical environment changes for this phase, since we will be adopting new systems and dropping others. The existing ex-Optimus CMDB which we will simply call by CMDB, will have to be part of our system, while the existing task management tool named Easy Vista, will have to be discontinued having its functionalities implemented within the portal itself. We shall now detail in the following sections, what was sough in terms of architectural qualities and detail on how it was achieved.

5.2 Quality Attribute Scenarios

Scenario	Source	Stimulus	Artifact	Environment	Response	Measure
Usability Usability-9	User	Wants to have access to the same type quick navigation interfaces which existed within Easy Vista	Portal	Runtime	The portal provides familiar information	User can navigate to his current most important entities with a single click
Usability Usability-10	User	Wishes to access and filter listings which may contain information that crosses two database domains	Portal	Runtime	The portal reuses the familiar listing and filters interface	User does not notice that the type of listing he is using contains internally more complexity than the other
Interoperability Interoperability-3	Portal	Needs to create, read, update, delete configuration items	CMDB	Any	This integration is performed through the an Oracle DB connector while respecting the imposed rules by the responsible third party.	Portal correctly ensures and implements the imposed manipulation rules
Interoperability Interoperability-6	Portal	Needs to mirror the CMDB creation, reading, updating, deleting of configuration items	HPSM	Any	This integration is performed through the provided HPSM API with the values manipulated within the Portal CMDB	Changes to the Portal CMDB are correctly mirrored into the HPSM

Table 5.1: Quality attribute scenarios - CMDB and task management phase

In table 5.1, we have defined the concrete quality attribute scenarios for the CMDB and task management phase. In terms of security, we have equipped our architecture with the necessary security qualities. So, on this phase, we will simply take advantage of whatever is in place for that category. The same is also true when it comes to the modifiability, supportability and deployability quality attributes. Therefore, we only have new scenarios when it comes to our usability and interoperability quality attributes. In the usability department, we are looking into empower our users with the capability of making smart decisions by providing them with the most relevant information for his specific case. In terms of interoperability, we are replacing the Easy Vista scenario which was introduced on the previous phase, with scenarios that are related to the CMDB integration.

5.3 Architectural Components

In figure 5.2, we have the representation of the resulting architectural components view of this iteration. The first noticeable change is that the `Easy Vista` component no longer exists. Instead, similar business logic was implemented within the `Portal FE` and `Portal DB` by taking advantage of the modifiability and usability scenarios introduced in the previous phase. Thus, providing implementing the `Easy Vista` functionalities within the `Portal FE` was simply, an exercise of designing and implemented the underlying web application domain and business logic, supported on the well defined architectural skeleton. On the other hand, the `Interoperability-3` scenario was implemented with the introduction of the new `CMDB DB` component and underlying connection with the `Portal FE`. This database resides inside an Oracle Real Application Cluster, which is basically a cluster of database servers. Even though this cluster is operated by the data center department, it is managed by a different deparment entity. Thus, it resides outside our Back-End DMZ, although it is going to act as a database back-end to our `Portal FE` component.

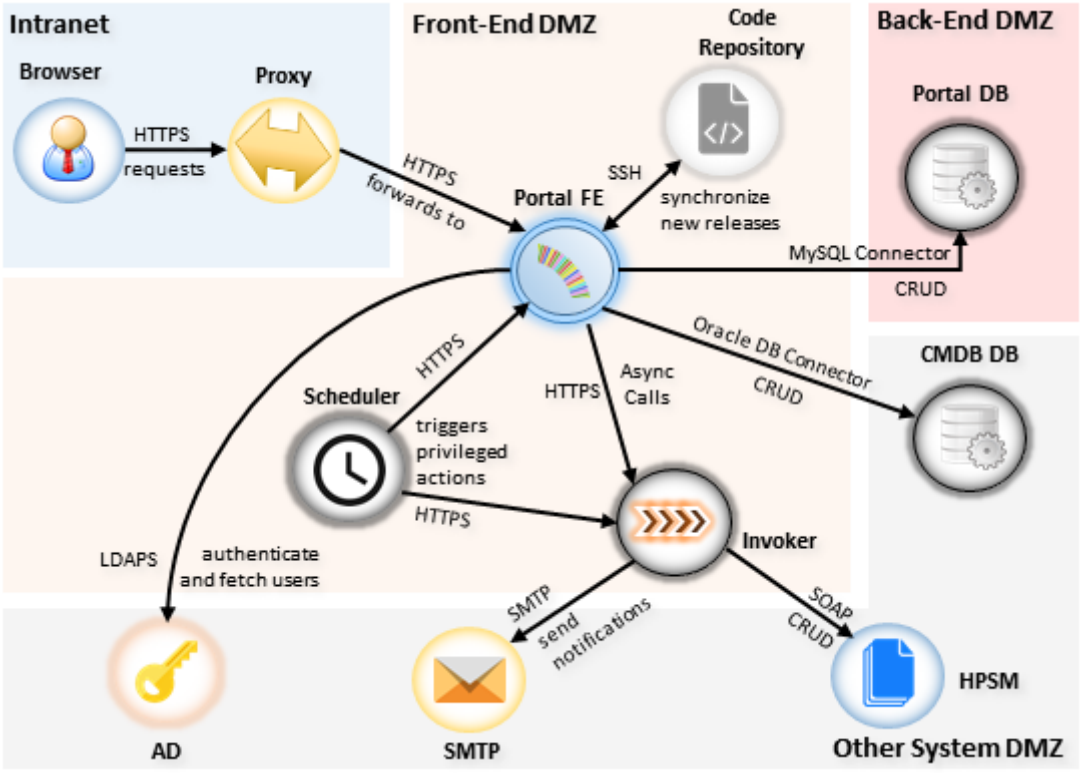

Figure 5.2: Architectural components for the CMDB and task management phase

The `Interoperability-6` scenario is partly implemented by the existing `HPSM` component and partly implemented by the logic implemented within the `Portal FE`. The tactic implemented in order to achieve this scenario is that whenever, within the workflow of a given configuration item, a modification is performed, that modification is not only persisted within the `Portal DB` and `CMDB DB` components, but it is also mirrored into the `HPSM` component through its web services. This ensures that the scenario is implemented and thus, that our system is not undercutting its requirement to be compliant with the ITIL policies, which based themselves upon the adopted ITIL tool `HPSM`. Another point worth mentioning is that, with the introduction of this `CMDB DB`, we introduced inter-database dependencies and interactions. The `Usability-10` scenario describes them in the user perspective. Basically, the configuration items now live outside the `Portal DB`. This means that whenever we want to relate a request with a configuration item, we are relating entities living within the `Portal DB`, with entities living within the `CMDB DB`. The same happens with the workflow state of a given configuration item, whenever we want find all the configuration items which are currently in a given state, we are performing queries which cross distinct databases. In the following section, we will go into detail on how was this issue tackled from the architectural point of view.

5.4 Module View

In figure 5.3, we have highlighted the changes which occurred within our layered module view during this phase. These changes are for most of the part, either related with the introduction of the a new data access object provider tweaked towards the CMDB database, or related with the

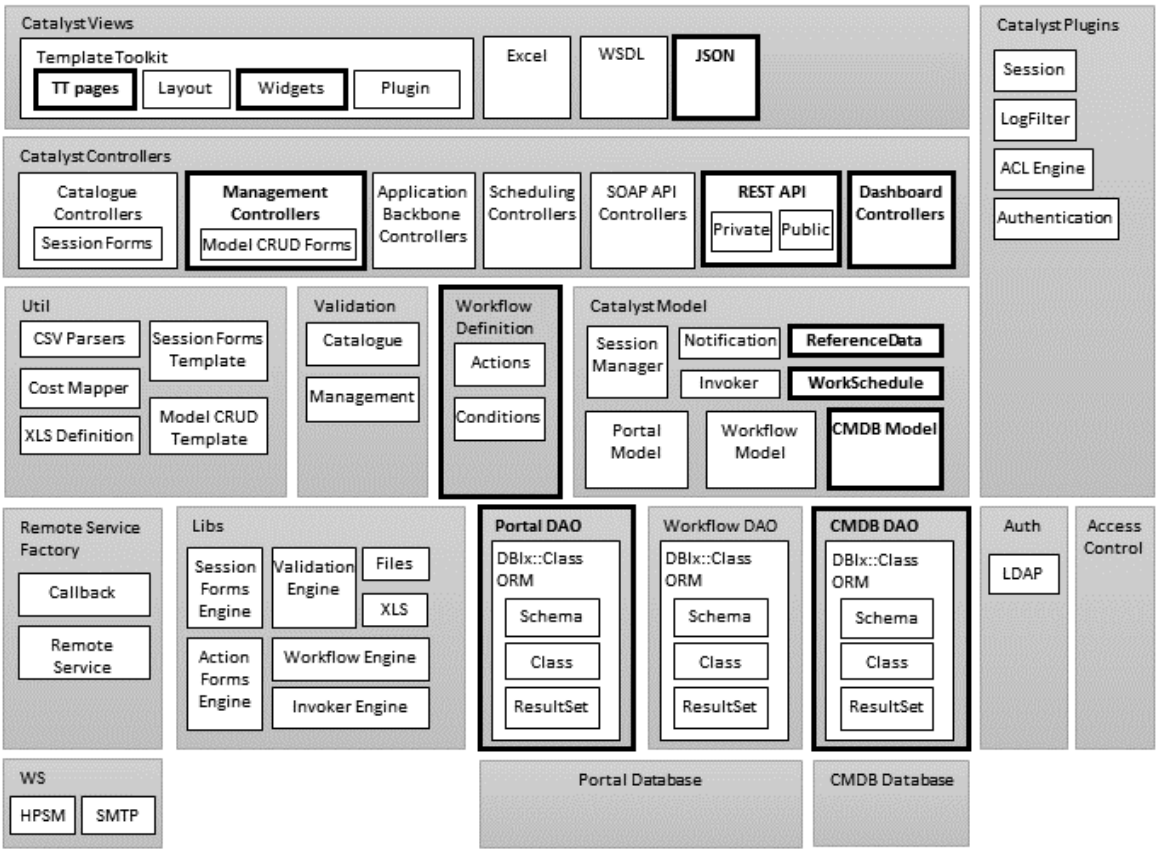

Figure 5.3: Layered module view for the CMDB and task management phase

implementation of new functionalities within the `Portal FE` component.

5.4.1 CMDB DAO

The `CMDB DAO` layer follows the same concept behind the `Portal DAO` layer, which was described at section 2.5 of chapter 2. In very general terms, it implements the CMDB domain specific classes and behaviour, by extending the classes provided by the `DBIx::Class ORM`, which is an implementation of an Active Record. It is within this layer that the `Interoperability-3` scenario is tackled. The scenario mentions that on its response, the responsible third party rules have to be respected. Since this database originated in a different team, there was a specific set of rules that were being manually followed when changing or deleting rows. These set of rules was related with keeping an history of the previous rows. Changing a given entry, meant to set valid from and valid to timestamps on the previous entry and cloning it into a new row with the changed values. This was achieved by the usage of Perl reflection, where the behaviour of update or delete methods within the Active Record classes was intercepted, and added of these extra steps. Part of the `Usability-10` scenario was also tackled here. The strategy was to replicate the searchable columns from the `Portal DB` workflow entity inside the structure of the `CMDB DB` configuration items entity. Thus, every time the workflow suffered from a state change or action, the resulting values where replicated and propagated into the corresponding entity within the `CMDB DB`. Thus, inter-database domain queries became virtually possible, since the essential information lived locally in both databases.

5.4.2 CMDB related modifications

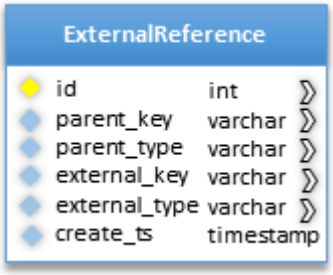

Figure 5.4: External references table for cross domain entities

In order to achieve the requirements of the `Interoperability-3` and `Interoperability-6` scenarios, further changes were performed inside the `Catalyst Model`, `Workflow Definition`, `Catalyst Controllers` and `Catalyst Views` layer. In the `Catalyst Views` layer, a `CMDB Model` exposer to the `CMDB DAO` was introduced, so that the `Catalyst Controllers` may access to the data access object which implement the `CMDB` component domain. A `ReferenceData` module was also added to this layer. This very simple module defines an interface of methods responsible to fetch from the `Portal Model`, the reference data related to the underlying name signature of its methods (eg. reference data for the operating systems, for the suppliers brands, etc). New workflow definitions were created for the `CMDB DB` domain specific entities inside the `Workflow Definition` layer. The corresponding `Management Controllers` which expose the workflows actions to the users were also created within the `Catalyst Controllers` layer. On the presentation side of it, the corresponding workflow action views were implemented by a new set of `TT pages` inside the `Catalyst Views`. Finally, to close the last opened end of the `Usability-10` scenario, where we want to be capable of telling which configuration items are related to a given request, we have created the table `ExternalReference` within the `Portal Database` represented in figure 5.4. This table keeps a track of all the relationships between entities which do not coexist within the same domain.

5.4.3 Task Management related modifications

Since modifications required in order to implement the task management requirement are mostly on the functionality implementation side, and less on the architectural side of the system, we only have one related scenario which is the `Usability-9`. So, instead of only giving focus to this scenario, we are also going to give emphasis on how our architecture contained all the required bits necessary to implement the business logic that once belonged to the `Easy Vista` component. Thus, relatively to the `Usability-9` scenario, a `REST API` was introduced within the `Catalyst Controllers` in order to empower our UI of widgets capable of making Asynchronous JavaScript and XML (AJAX)[35] calls that returned the relevant points of interest to a given user in a timely manner. Along side with that, still inside the `Catalyst Controllers`, a new type of controllers was introduced, the `Dashboard Controllers`, which are responsible to fetch information that is evaluated as important to the user by implemented business logic. Obviously, the corresponding `Widgets` and `TT Pages` were created on the `Catalyst Views`

layer.

Relatively to the implementation of the `Easy Vista` business logic within our portal, the affected layers were: the `Portal DAO`, the `Catalyst Model`, the `Workflow Definition`, the `Catalyst Controllers` and the `Catalyst Views`. Inside the `Portal DAO` layer, the corresponding task management domain specific entities were created and business logic implemented. Inside the `Catalyst Model` layer, a new `WorkSchedule` catalyst component was created with the goal of providing support to the calculations which involved dates, times and work schedules. Inside the `Workflow Definition` layer, new workflow definitions, states, actions and conditions were defined for the newly created types of data access objects inside the `Portal DAO`. Consequently, inside the `Catalyst Controllers` layer, the corresponding workflows actions were exposed through the `Management Controllers`. Finally, within the `Catalyst Views`, the views which correspond to those workflow actions were implemented in the `TT pages`, and `Widgets` related to managing tasks were also created.

5.5 Closing Comments

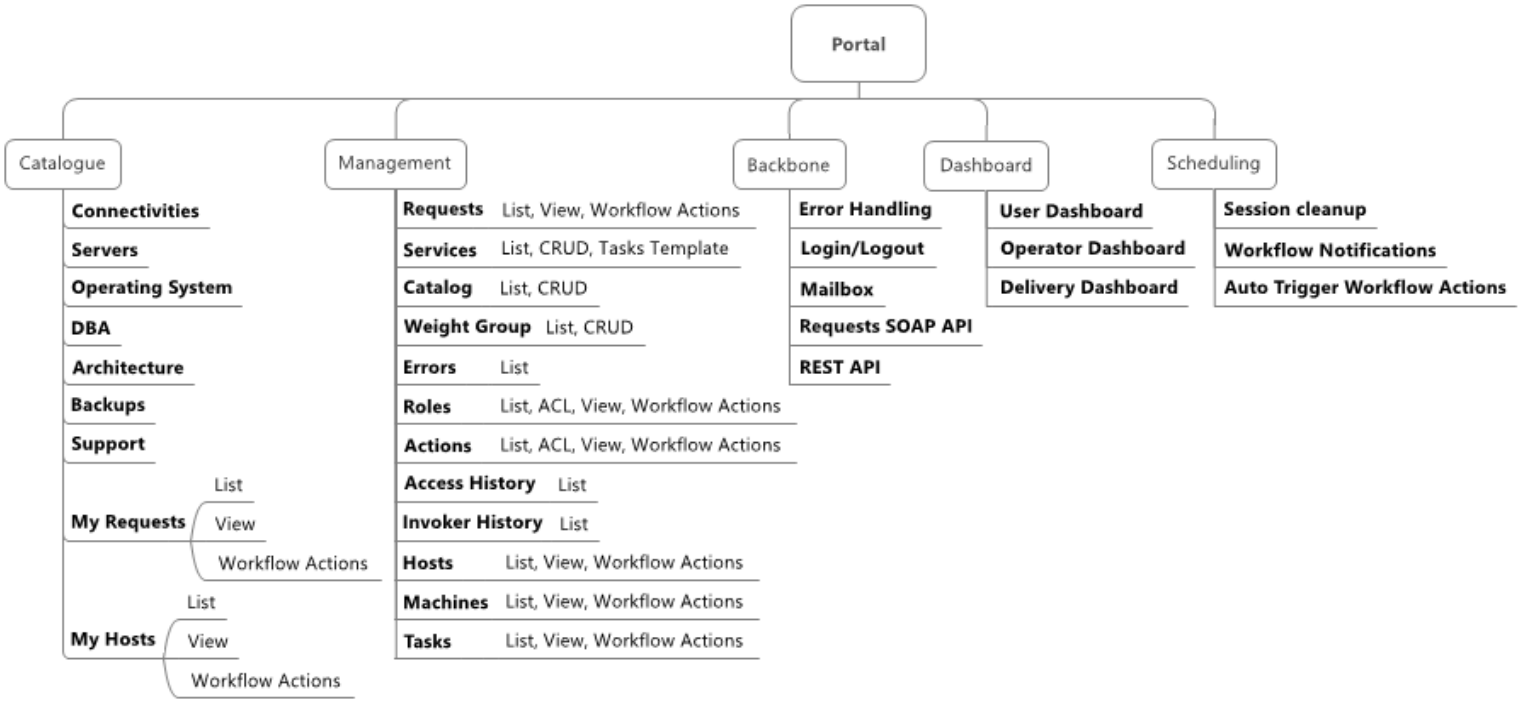

Figure 5.5: Functional decomposition view for the CMDB and task management phase

In figure 5.5, we have the representation of all the functionalities available within the portal by the end of the CMDB and task management phase. Throughout the chapter, we saw how the resulting architecture, refined throughout many iterations, matured to the point where it became capable of easily acting as an implementation recipe for the sort of requirements drafted by the data center department. We saw how the forces of the stakeholders and the technical environment can directly influence the resulting architecture with the introduction of a third party managed `CMDB DB` in our system. From the architectural point of view, if the choice was given of implementing a CMDB domain

within the existing `Portal DB`, or attempting to adopt a database that belongs to another area, we would have chosen the first. After all, the first scenario implementation would be easier since the existing database was just a schema of tables with data inserted, which of course, could be easily migrated into the `Portal DB` schema. But, since the second scenario was imposed by our stakeholders upon us for internal reasons, we had the chance to face the challenges of having domain entities spread-out through distinct database engines, and describe a possible solution for the issue.

6

Conclusions and Future Work

Contents

We shall now meditate upon the path which brought the architecture to where it is at the moment, and draw some lessons about the mistakes that were made and possible improvements. We shall also present concrete data with the statistics of the usage of many different modules within the data center portal. To conclude this thesis, we shall present the roadmap of future releases in drafting.

6.1 Lessons

When the project first started, we did not put too much emphasis on its architecture. Its early stages were all about speed, and none about software quality. The immediate result of that was a double edged sword: lots of functionality appeared from night to day, in terms of presentation and usability, it was very user friendly and attractive, but, on the other edge, code was just being copied and repeated all over. Thus, the system grown at the fast pace on which the stakeholders came up with new ideas but, as soon as the stakeholders started wanting to change functionality instead of adding, the house of cards fell. Since code was duplicated all over, a simple change, a simple bug correction, would have to be repeated throughout many modules. This led us to stop, sit and think on how to bring qualities that had been overlooked in the conception of the system. Consequently, the introduction of the described architecture introduced in chapter 2 happened by refactoring the prototype to become much more than that. We learned that even a prototype should have a well defined underlying architecture, especially the sooner, the better. That because, sooner or later, if the prototype is successful, it will no longer be seen as a prototype, but rather as system in production. And, introducing an architecture into a chaotic system, is more complex than simply put some effort in drafting the architecture for its early stages.

When we finally started drafting the architecture for the system, we felt that not only it was very important consider the stakeholders requirements, but as well as prepare for the future by embedding the architecture with modifiability and extensibility qualities. If the stakeholders asked for a certain functionality, there is a high chance that they will alter it in the future, and then go back to what it was, over and over. When we are talking about in-house development of a tool that supports processes, it has to be flexible enough to empower the changing of the processes. The processes should not need to adapt to the system, instead, the system should adapt to the processes. These intra-department processes are in a constant quest for the self-improvement, and only by having flexible system which can evolve easily with them, can they be implemented efficiently. Another topic worth mentioning is that throughout our architecture business cycle, we were influenced into introducing external systems maintained by third parties into our architecture. Downtimes on these third party systems do happen, and when they do, most likely, we will not be informed of them. Thus, it is important to architect these integrations with the assumption that the attempts to integrate will fail quite often. Only with a fault tolerant integration components, we will have our system safeguarded.

As a final remark on the conception of this system: we believe that introducing continuous integration to our architecture is something that we need to consider in the near future. The system so far has been designed and developed by just one person, but having the logbook completely full with new

stories to develop, more and more the need of expanding the team becomes noticeable. Thus, we believe that introducing testability and continuous integration into our architecture is going to become very relevant in the near future. Once again, we can only hope we are not too late for that.

6.2 Usage Data

In table 6.1, we have a representation of the volume of requests, made within the data center portal, since the time it was first put into production. The first release dates back to 12nd March 2012. We can see that throughout the first month, only a dozen of requests were made through the data center portal. This mainly because the data center clients were still being informed and educated of the new process. Consequently, in the following months, we saw a steady increment of the requests performed via the data center portal. By the end of its first year, the mark of 1894 requests had been achieved, which amounts to an average of 189 requests per month, 9 requests a day. In 2013, the number of requests performed through the data center portal doubled, with an average of 324 requests per month, 16 requests per day. This is explained by the fact that some of our clients did not adopt the system right away when it was released. And only by the end of the 4rd quarter of 2012, we had everyone requesting through the portal. In 2014, the year where ZON and Optimus became the new brand NOS, we had a mark of 4300 requests, with an average of 358 requests per month and 17 requests per day. The increment is not as many as expected for a fusion of two companies, mainly because internally, both ex-Zon and ex-Optimus still existed. Finally, by the end of the 3rd quarter of 2015, we have already achieved the mark of requests of the previous year. This is mainly because 2015, is the year where the processes within the data center department started to get standardized for both areas.

Year	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	Total
2012	-	-	17	111	162	136	197	169	255	334	293	220	1894
2013	322	270	265	286	333	245	398	250	414	453	396	266	3898
2014	324	358	361	350	375	328	426	275	363	364	414	362	4300
2015	440	470	589	522	410	420	521	399	505	-	-	-	4276

Table 6.1: Volume of portal requests since the beginning of times

In table 6.2, we have the volume of requests divided in classes, and we have the number of users which fit within each class. The important to retain from this data is that until this date, we have had 375 distinct employees or collaborators within NOS, performing requests inside the data center portal. Also, we can see that we have a couple of users which fit inside the classes with a large amount of requests. This is justified with the fact that, just like the data center department has its own delivery team, there are other departments in the organization which have their corresponding delivery teams, too. Thus, instead of giving the access to all the employees within these departments, these departments opted to pipe those requests through their delivery teams.

No of requests	No of users
$]0-10]$	259
$]10,30]$	27
$]30,50]$	39
$]50,80]$	20
$]80,120]$	14
$]120,180]$	9
$]180,250]$	5
$]250,+\infty[$	2
Total	375

Table 6.2: Classes of portal request volumes mapped to the number of users who fit in those classes

In table 6.3, we have the volume of workflow actions performed by the workflow engine since its introduction. If we consider that after the end of December of 2013, the workflow engine was being used widely within all the entities of the data center portal, we can easily justify the huge spike on its numbers, since creating a requests, updating a requests, rejecting a requests, and every other action became a workflow action. Also, at September of 2015, we can notice a huge spike within the average monthly values. That is because the task management module was put into production, on the first of that month. With that, all the data center technicians migrated from Easy Vista into the Portal, in order to manage their tasks.

Year	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	Total
2013	-	-	-	-	-	45	1075	687	1134	1205	1083	748	5977
2014	2155	2818	2803	2614	2929	2142	2926	1959	2327	2582	2750	2336	30341
2015	2689	2902	3520	13530	3545	2940	3750	3180	10290	-	-	-	46346

Table 6.3: Volume of workflow actions performed in portal since the workflow engine introduction

In table 6.4, we have the volume of the number of successful integrations done by our system. We divided the data into the separate systems with which we integrate with. The numbers speak by themselves. If we consider that before the HPSM integration, the data center clients and the data center delivery teams had to manually map into the HPSM what was later automatized with these integrations, we can clearly see the impact it had in the efficiency of the department. After all, for every request, the client had to create it within the data center portal, and then create a service request inside the HPSM with the portal id in the description. Then, the data center delivery team had to pick the service request in HPSM, change it to work in progress, create the corresponding change, associate the service request with the change, associate the configuration items with change, if the number of configuration items was small the association would be easy, if the number of configuration items was large it would require huge amounts of times to associate, then the change also had to

be assigned and manipulated, etc. Basically, these integrations took out of our stakeholders a huge amount of non-consequent work.

Year	System	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	Total
2013	HPSM	-	-	-	-	-	-	-	-	-	-	-	256	256
	EV	-	-	-	-	-	-	-	-	-	-	-	-	-
	SMTP	-	-	-	-	-	-	-	-	-	-	-	55	55
2014	HPSM	3513	4220	4315	4307	4798	3795	5297	3425	3926	4711	4736	4094	51137
	EV	-	-	-	-	295	746	947	631	789	790	814	795	5807
	SMTP	1265	1451	1702	1728	1823	1900	1942	1262	1654	1777	1923	1657	20084
2015	HPSM	4628	5452	6432	5988	4411	6439	6164	5288	9030	-	-	-	53832
	EV	871	1096	1380	1155	930	914	1182	914	-	-	-	-	8442
	SMTP	1828	2027	2439	2094	2068	1675	2043	1872	2463	-	-	-	18509

Table 6.4: Volume of the number of integration services invoked by the portal

And finally, to bring closure to our usage data walk-through which puts into numbers the impact of the system upon our organization, we have the volume of the tasks submitted to the data center technicians, within the table 6.5. As mentioned before, the task management was released on the first of september, and therefore, we only have data until the half of October. Nevertheless, we can see that a huge amount of tasks has already been dispatched to the implementing teams through the data center portal. We are speaking of almost a thousand tasks just in the first month of service.

Team	September, 2015	1st half of October, 2015	Total
Delivery	63	21	84
Unix & Storage	211	117	328
Backups	22	9	31
Database	7	5	12
Field Support	60	38	98
Microsoft & Virtualization	99	38	137
Monitoring	144	155	299
Network	373	175	548
Total	979	558	1537

Table 6.5: Volume of tasks submitted to technicians since the implementation of the task management module

6.3 Roadmap

As future work, a series of well defined requisites have already been identified. These requisites span a wide range of subjects, and they are mainly focused on the expansion of both functionalities

and catalogue. A catalogue, tweaked towards supporting the areas which operate the systems belonging to NOS external clients, is being drafted and implemented. These development will not impact the architecture since all the required qualities are already set in place. In other words, it is the simple introduction of a new family of services, just like happened in the very first iteration of the architecture.

One of the most urgent requirements is the enrichment of the `CMDB DB` component with new classes of configurations items. Yet again, the architecture is equipped with the means to empower this development without any change on it. Similar to what was done in the `CMDB` phase after the architecture was defined, new data access objects classes will be defined for the new classes of configuration items, new workflows will be also defined for them, new controllers which expose the workflow actions for each new entity will be created, and the corresponding views will be implemented.

With some initial trials already under belt, there is the integration with automation orchestrators, in order to empower uses cases which allow for the automated provisioning of virtual servers. This development will require a new interoperability scenarios to be introduced, which means that new web services integration packages will have to be developed while adhering to the already defined architecture.

Somewhere in the future, the system will have to support the managing of available network segments. It will also have to be capable of managing the knowledge base of the existing data center client platforms. In order to achieve that, it is predictable that no additional changes will have to be performed inside the defined architecture. Instead, in the very image of the Task Management phase, the current portal domain will have to be extended in order to support these new business logics.

There is also a very important integration, pending at the moment, which is the migration of all the underlying integrations, from the HPSM, to the new ITIL tool, which is slowly being implemented within the organization. This integration will imply deprecating the integrations introduced for the HPSM, by eliminating this component from the architecture, and at the same time, it will imply introducing the corresponding integrations with the final tool, which will replace HPSM as new component inside the architecture. Thus, one may say that this development will be very similar to the HPSM part of the integration phase described before.

Finally, at the bottom of the queue, with no estimated date of release, there is the budget management module, which will be responsible for creating the notion of charge back within the requests performed by the data center users. No architectural changes are predicted on this phase, since the implemented systems already provide the necessary qualities in order to implement the underlying domain logic of this budget management module.

Bibliography

[1] itSMF UK, *ITIL Foundation Handbook*. The Stationery Office; 3rd ed. (2012) edition (January 31, 2012), 2012.

[2] G. Donnell, *The CMDB imperative how to realize the dream and avoid the nightmares*. Upper Saddle River, N.J: Prentice Hall, 2009.

[3] D. Scott, J. E. Pultz, E. Holub, T. Bittman, and P. McGuckin, "Introducing the gartner it infrastructure and operations maturity model," *Gartner. Donna Scott, Jay E. Pultz, Ed Holub, Thomas J. Bittman, Paul McGuckin*, vol. 1, 2007.

[4] M. Thomatis, *Network design cookbook : architecting Cisco networks*. Raleigh, N.C: Lulu, 2012.

[5] R. Allen and L. Hunter, *Active directory cookbook*. " O'Reilly Media, Inc.", 2006.

[6] K. Zeilenga, "Lightweight directory access protocol (ldap): Technical specification road map," 2006.

[7] R. S. Sandhu and P. Samarati, "Access control: principle and practice," *Communications Magazine, IEEE*, vol. 32, no. 9, pp. 40–48, 1994.

[8] P. Adamczyk, "The anthology of the finite state machine design patterns," in *The 10th Conference on Pattern Languages of Programs*, 2003.

[9] P. Avgeriou and U. Zdun, "Architectural patterns revisited–a pattern," 2005.

[10] "EasyVista — ITSM, ITAM, IT Consumerization," http://www.easyvista.com/.

[11] "Remedy - IT Service Management Suite - BMC," www.bmc.com/it-solutions/remedy-itsm.html.

[12] R. T. Monroe, A. Kompanek, R. Melton, and D. B. Garlan, "Architectural styles, design patterns, and objects," *IEEE software*, p. 43, 1996.

[13] A. Leff and J. T. Rayfield, "Web-application development using the model/view/controller design pattern," in *Enterprise Distributed Object Computing Conference, 2001. EDOC'01. Proceedings. Fifth IEEE International*. IEEE, 2001, pp. 118–127.

[14] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[15] L. Bass, *Software architecture in practice*. Boston: Addison-Wesley, 2003.

[16] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, “Hypertext transfer protocol–http/1.1,” Tech. Rep., 1999.

[17] L. Wall, T. Christiansen, and J. Orwant, *Programming perl*. ” O’Reilly Media, Inc.”, 2004.

[18] J. Rockway, *Catalyst accelerating Perl web application development*. Birmingham, UK: Packt Pub, 2007.

[19] “Git - SCM,” https://git-scm.com.

[20] “Apache Subversion,” https://subversion.apache.org.

[21] J. Martin and S. Institute, *Managing the data-base environment*. Prentice-Hall Englewood Cliffs (NJ), 1983.

[22] K. Amirtahmasebi, S. R. Jalalinia, and S. Khadem, “A survey of sql injection defense mechanisms,” in *Internet Technology and Secured Transactions, 2009. ICITST 2009. International Conference for*. IEEE, 2009, pp. 1–8.

[23] E. Rescorla, “Http over tls,” 2000.

[24] S. Gundavaram, *CGI programming on the World Wide Web*. O’Reilly, 1996.

[25] B. Adida, “It all starts at the server [world wide web and fastcgi],” *Internet Computing, IEEE*, vol. 1, no. 1, pp. 75–77, 1997.

[26] J. Postel, “Transmission control protocol,” 1981.

[27] “DBIx::Class - An ORM for Perl that is extensible and flexible,” www.dbix-class.org.

[28] M. Fowler, “Inversion of control containers and the dependency injection pattern,” 2004.

[29] E. Freeman, E. Robson, B. Bates, and K. Sierra, *Head first design patterns*. ” O’Reilly Media, Inc.”, 2004.

[30] G. Wilson, D. Aruliah, C. T. Brown, N. P. C. Hong, M. Davis, R. T. Guy, S. H. Haddock, K. Huff, I. M. Mitchell, M. D. Plumbley *et al.*, “Best practices for scientific computing,” *PLoS biology*, vol. 12, no. 1, p. e1001745, 2014.

[31] M. Fowler, *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.

[32] “Bootstrap - The world’s most popular mobile-first and responsive front-end framework,” getbootstrap.com.

[33] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, “Simple object access protocol (soap) 1.1,” 2000.

[34] “Workflow - search.cpan.org,” http://search.cpan.org/dist/Workflow/lib/Workflow.pm.

[35] J. J. Garrett *et al.*, “Ajax: A new approach to web applications,” 2005.

A

Code Examples

A.1 Data Center Catalogue

```
my $step1 = Zon::Portal::Forms::Step->with_traits(qw/
  Zon::Portal::Forms::Behaviour::Step::Uploadable
/)->new(
  view_name => $view_name,
  view_path => $view_path,
  view_file => 'step1.tt',
  uid       => 'step1',
  next      => 'resume',
  form      => $form,

  base_selector  => 'base',

  bases      => [
    Zon::Portal::Forms::Base->new(
      uid         => 'base',
      parameters  => {
        title       => Zon::Portal::Forms::Type::Text->new( session_name  => 'title', parameter_name => 'title' ),
        description => Zon::Portal::Forms::Type::Text->new( session_name  => 'description', parameter_name => 'description' ),
      },

      submit_validation  => Zon::Portal::Forms::Validation->new(
        validator  => 'Portal::Validation::Catalogue::InformationRequestOther',

        values   => {
          title       => { type => 'param', value => 'title'       },
          description => { type => 'param', value => 'description' },
        },

        map     => {
          title       => 'step1:base:title'       ,
          description => 'step1:base:description' ,
        },

        errors   => {
          title       => [ 'step1:base:title'       ],
          description => [ 'step1:base:description' ],
        },
      ),
    ),

    Zon::Portal::Forms::Base->with_traits(qw/Zon::Portal::Forms::Behaviour::Base::Uploadable/)->new(
      uid         => 'attachments',
      parameters  => {},
      upload      => 'attachment',

      upload_list => 'files',

      upload_validation  => Zon::Portal::Forms::Validation->new(
        validator  => 'Portal::Validation::Catalogue::InformationRequestOther',

        values     => { upload => { type => 'upload', value => 'attachment' } },
        map        => { upload => 'step1:attachments:upload'                  },
        errors     => { upload => [ 'step1:attachments:upload' ]              }
      )
    ),
  ]
);

sub step1          : Action { my ( $self, $c ) = @_; $step1->show($c);     }
sub submit_step1   : Action { my ( $self, $c ) = @_; $step1->submit($c);   }
sub goto_step1     : Action { my ( $self, $c ) = @_; $step1->goto($c);     }
sub upload_step1   : Action { my ( $self, $c ) = @_; $step1->upload($c);   }
sub deupload_step1 : Action { my ( $self, $c ) = @_; $step1->deupload($c); }
```

Code 1: Example of a Catalogue Controller creating a step through the Session Forms Engine

```perl
package Portal::Forms::Template::Base::Catalogue::Schedule;

use utf8;

use Zon::Portal::Forms::Step;
use Zon::Portal::Forms::Base;
use Zon::Portal::Forms::Form;
use Zon::Portal::Forms::Validation;
use Zon::Portal::Forms::Type::Text;
use Zon::Portal::Forms::Type::ArrayText;
use Zon::Portal::Forms::Type::TimeRange;

use Portal::Validation::Template::Catalogue::Schedule;

sub new {
  my ( $class, %args ) = @_;

  die "step_uid is required!" if !$args{step_uid};

  return Zon::Portal::Forms::Base->new(
    uid         => 'schedule',
    parameters     => {
      date    => Zon::Portal::Forms::Type::Text->new( session_name => 'date', parameter_name => 'schedule_date' ),
      time    => Zon::Portal::Forms::Type::TimeRange->new( session_name => 'time', parameter_name => 'schedule_time' ),
    },

    show_hook    => sub {
      my ( $self, $c, $at ) = @_;

      $c->stash->{step} = $args{step_uid};
    },

    submit_validation  => Zon::Portal::Forms::Validation->new(
      validator  => 'Portal::Validation::Template::Catalogue::Schedule',

      values   => {
                date => { type => 'param', value => 'schedule_date'   },
                hi  => { type => 'param', value => 'schedule_time_hi'  },
                mi  => { type => 'param', value => 'schedule_time_mi'  },
                hf  => { type => 'param', value => 'schedule_time_hf'  },
                mf  => { type => 'param', value => 'schedule_time_mf'  },
              },

      map     => {
                date => 'step:date',
                hi  => 'step:hi',
                mi  => 'step:mi',
                hf  => 'step:hf',
                mf  => 'step:mf',
              },

      errors   => {
                date => [ 'step:date' ],
                time => [ 'step:hi', 'step:mi', 'step:hf', 'step:mf' ],
              },
    ),
  );
}

1;
```

Code 2: Example of a Session Forms Template class which will be consumed by Controllers

```
package Portal::Validation::Template::Catalogue::Schedule;

use Moose;

use DateTime;
use Zon::Portal::Validation::Util;
use Zon::Portal::Validation::Class;

use utf8;


field 'step:date' => {
    required    =>  1,
    validation  =>
        sub  {
            my ( $self, $this, $params ) = @_;

            my $date = $params->{'step:date'};
            $self->error($this, 'msg') if(is_date_lt_now($date, '/', 1));
        }
};

field 'step:hi' => {
    required    => 1,
    regex       => regex_hour_digit
};

field 'step:mi' => {
    required    => 1,
    regex       => regex_min_digit
};

field 'step:hf' => {
    required    => 1,
    regex       => regex_hour_digit
};

field 'step:mf' => {
    required    => 1,
    regex       => regex_min_digit
};
```

Code 3: Example of the Validation class which is used on the previous exampled

```
[% step = 'step1' %]

<div class="container">
        <form method="POST" enctype="multipart/form-data">
    <input name="base" type="hidden" value=""/>

    <!-- title -->
    <div>
    [% PROCESS widget/inputtext_default input = {
        label    = 'Subject'
        name     = 'title'
        required = 1
        error    = c.stash.errors.base.title
        value    = c.session.wizard.$step.base.title
        size     = 40
        tooltip  = tooltip.title
    }%]
    </div>

    <!-- description -->
    <div>
    [% PROCESS widget/textarea_default input = {
        label    = 'Description'
        name     = 'description'
        required = 1
        error    = c.stash.errors.base.description
        value    = c.session.wizard.$step.base.description
        cols     = 40
        rows     = 10
        tooltip  = tooltip.description
    }%]
    </div>

    [% IF c.session.wizard.FM.get('attachments:files').getList().size() %]
    <!-- attachments -->
    <div>
    [% PROCESS widget/table_upload input = {
        wizard   = 1
        selector = { name = 'base', base = 'attachments' }
        list     = c.session.wizard.FM.get('attachments:files').getList()
        action   = "deupload_$step"
    }%]
    </div>
    [% END %]

    <!-- attachment -->
    <div>
    [% PROCESS widget/upload_default input = {
        label    = 'Attachment'
        name     = 'attachment'
        required = 0
        error    = c.stash.errors.attachments.upload
        action   = "upload_$step"
        selector = { name = 'base', base = 'attachments' }
    }%]
    </div>


    <!-- Buttons -->
        <div>
    [% PROCESS widget/buttons_default input = {
        next     = 'submit_step1'
    }%]
        </div>
  </form>
</div>
```

Code 4: Example of the implementation of the TT page from a step form using the widgets

```
package Portal::Controller::Management::Task;

use Moose;
use utf8;

BEGIN { extends 'Catalyst::Controller' }

use Try::Tiny;

my $view_name        = 'TT';
my $view_path        = 'shared/lib/workflow-task/';

use Portal::Action::Class::Shared::Task::List;
use Portal::Action::Class::Shared::Task::View;
use Portal::Action::Class::Shared::Task::Submit;
use Portal::Action::Class::Shared::Task::Assign;
use Portal::Action::Class::Shared::Task::Reassign;
use Portal::Action::Class::Shared::Task::Halt;
use Portal::Action::Class::Shared::Task::Resume;
use Portal::Action::Class::Shared::Task::Cancel;
use Portal::Action::Class::Shared::Task::Update;
use Portal::Action::Class::Shared::Task::Repick;
use Portal::Action::Class::Shared::Task::Complete;
use Portal::Action::Class::Shared::Task::Resubmit;
use Portal::Action::Class::Shared::Task::Pick;
use Portal::Action::Class::Shared::Task::Edit;
use Portal::Action::Class::Shared::Task::Adjust;

use Portal::Validation::Shared::Task;

my $list = Portal::Action::Class::Shared::Task::List->new(
    uid                 => 'list',

    view_name           => $view_name,
    view_path           => $view_path,
    view_file           => 'list.tt',
);

...

sub list            : Action { my ( $self, $c ) = @_; $list->list($c);       }
sub reset_list      : Action { my ( $self, $c ) = @_; $list->reset($c);      }
sub pool_list       : Action { my ( $self, $c ) = @_; $list->pool($c, 'tasks'); }
sub view            : Action { my ( $self, $c ) = @_; $view->show($c);       }
sub submit_view     : Action { my ( $self, $c ) = @_; $view->submit($c);     }
sub back_view       : Action { my ( $self, $c ) = @_; $view->back($c);       }
sub edit            : Action { my ( $self, $c ) = @_; $edit->show($c);       }
sub submit_edit     : Action { my ( $self, $c ) = @_; $edit->submit($c);     }
sub adjust          : Action { my ( $self, $c ) = @_; $adjust->show($c);     }
sub submit_adjust   : Action { my ( $self, $c ) = @_; $adjust->submit($c);   }
sub submit          : Action { my ( $self, $c ) = @_; $submit->show($c);     }
sub submit_submit   : Action { my ( $self, $c ) = @_; $submit->submit($c);   }
sub assign          : Action { my ( $self, $c ) = @_; $assign->show($c);     }
sub submit_assign   : Action { my ( $self, $c ) = @_; $assign->submit($c);   }
sub reassign        : Action { my ( $self, $c ) = @_; $reassign->show($c);   }
sub submit_reassign : Action { my ( $self, $c ) = @_; $reassign->submit($c); }
sub halt            : Action { my ( $self, $c ) = @_; $halt->show($c);       }
sub submit_halt     : Action { my ( $self, $c ) = @_; $halt->submit($c);     }
sub resume          : Action { my ( $self, $c ) = @_; $resume->show($c);     }
sub submit_resume   : Action { my ( $self, $c ) = @_; $resume->submit($c);   }
sub cancel          : Action { my ( $self, $c ) = @_; $cancel->show($c);     }
sub submit_cancel   : Action { my ( $self, $c ) = @_; $cancel->submit($c);   }
sub update          : Action { my ( $self, $c ) = @_; $update->show($c);     }
sub submit_update   : Action { my ( $self, $c ) = @_; $update->submit($c);   }
sub repick          : Action { my ( $self, $c ) = @_; $repick->show($c);     }
sub submit_repick   : Action { my ( $self, $c ) = @_; $repick->submit($c);   }
sub complete        : Action { my ( $self, $c ) = @_; $complete->show($c);   }
sub submit_complete : Action { my ( $self, $c ) = @_; $complete->submit($c); }
sub resubmit        : Action { my ( $self, $c ) = @_; $resubmit->show($c);   }
sub submit_resubmit : Action { my ( $self, $c ) = @_; $resubmit->submit($c); }
sub pick            : Action { my ( $self, $c ) = @_; $pick->show($c);       }
sub submit_pick     : Action { my ( $self, $c ) = @_; $pick->submit($c);     }
```

Code 5: Example of a management controller being implemented with Action Forms Engine classes

A.2 Workflow Engine

```
<workflow>
  <type>task</type>
  <time_zone>local</time_zone>
  <description>The life cycle of a task</description>
  <initial_state>Created</initial_state>
  <persister>Database</persister>

  <state name="Created">
    <action name="init in task" resulting_state="Created"><condition name="needs_init"/></action>
    <action name="edit in task" resulting_state="Created"><condition name="is_delivery in task"/></action>
    <action name="submit in task" resulting_state="Unassigned"><condition name="is_delivery in task"/></action>
    <action name="cancel in task" resulting_state="Cancelled"><condition name="is_delivery in task"/></action>
  </state>

  <state name="Unassigned">
    <action name="adjust in task" resulting_state="Unassigned"><condition name="is_delivery in task"/></action>
    <action name="pick in task" resulting_state="Assigned"><condition name="is_target_group in task"/></action>
    <action name="assign in task" resulting_state="Assigned"><condition name="can_assign in task"/></action>
    <action name="halt in task" resulting_state="Halted"><condition name="is_delivery in task"/></action>
    <action name="cancel in task" resulting_state="Cancelled"><condition name="is_delivery in task"/></action>
  </state>

  <state name="Assigned">
    <action name="adjust in task" resulting_state="Assigned"><condition name="is_delivery in task"/></action>
    <action name="update in task" resulting_state="Assigned"><condition name="is_target_user in task"/></action>
    <action name="repick in task" resulting_state="Assigned"><condition name="can_repick in task"/></action>
    <action name="reassign in task" resulting_state="Assigned"><condition name="can_reassign in task"/></action>
    <action name="halt in task" resulting_state="Halted"><condition name="can_halt in task"/></action>
    <action name="cancel in task" resulting_state="Cancelled"><condition name="is_delivery in task"/></action>
    <action name="complete in task" resulting_state="Completed"><condition name="is_target_user in task"/></action>
  </state>

  <state name="Halted">
    <action name="adjust in task" resulting_state="Halted"><condition name="is_delivery in task"/></action>
    <action name="resume in task" resulting_state="Assigned"><condition name="can_resume in task"/></action>
    <action name="resubmit in task" resulting_state="Unassigned"><condition name="can_resubmit in task"/></action>
    <action name="cancel in task" resulting_state="Cancelled"><condition name="is_delivery in task"/></action>
    <action name="reassign in task" resulting_state="Halted"><condition name="can_reassign in task"/></action>
  </state>

  <state name="Cancelled"></state>

  <state name="Completed"></state>
</workflow>
```

Code 6: Example of the XML which creates the workflow skeleton

```
<actions>
  <action name="halt in task"     class="Portal::Workflow::Action::Task::Halt"/>
  <action name="cancel in task"   class="Portal::Workflow::Action::Task::Cancel"/>
  <action name="adjust in task"   class="Portal::Workflow::Action::Task::Adjust"/>
  <action name="init in task"     class="Portal::Workflow::Action::Task::Init"/>
  <action name="edit in task"     class="Portal::Workflow::Action::Task::Edit"/>
  <action name="submit in task"   class="Portal::Workflow::Action::Task::Submit"/>
  <action name="pick in task"     class="Portal::Workflow::Action::Task::Pick"/>
  <action name="assign in task"   class="Portal::Workflow::Action::Task::Assign"/>
  <action name="update in task"   class="Portal::Workflow::Action::Task::Update"/>
  <action name="repick in task"   class="Portal::Workflow::Action::Task::Repick"/>
  <action name="reassign in task" class="Portal::Workflow::Action::Task::Reassign"/>
  <action name="complete in task" class="Portal::Workflow::Action::Task::Complete"/>
  <action name="resume in task"   class="Portal::Workflow::Action::Task::Resume"/>
  <action name="resubmit in task" class="Portal::Workflow::Action::Task::Resubmit"/>
</actions>
```

Code 7: Example of the declarations of workflow actions within XML configuration files

```xml
<conditions>
  <condition name="is_target_group in task" class="Portal::Workflow::Condition::Task::IsTargetGroup"/>
  <condition name="is_target_user in task" class="Portal::Workflow::Condition::Task::IsTargetUser"/>
  <condition name="is_team_leader in task" class="Portal::Workflow::Condition::Task::IsTeamLeader"/>
  <condition name="has_target_user in task" class="Portal::Workflow::Condition::Task::HasTargetUser"/>
  <condition name="is_delivery in task" class="Portal::Workflow::Condition::Task::IsDeliveryGroup"/>

  <condition name="can_repick in task" class="Workflow::Condition::LazyAND">
    <param name="condition1" value="is_target_group in task" />
    <param name="condition2" value="!is_target_user in task" />
  </condition>

  <condition name="can_assign in task" class="Workflow::Condition::LazyOR">
    <param name="condition1" value="is_delivery in task" />
    <param name="condition2" value="is_team_leader in task" />
  </condition>

  <condition name="can_reassign in task" class="Workflow::Condition::LazyOR">
    <param name="condition1" value="is_delivery in task" />
    <param name="condition2" value="is_team_leader in task" />
  </condition>

  <condition name="can_resume as delivery in task" class="Workflow::Condition::LazyAND">
    <param name="condition1" value="is_delivery in task" />
    <param name="condition2" value="has_target_user in task" />
  </condition>

  <condition name="can_resume in task" class="Workflow::Condition::LazyOR">
    <param name="condition1" value="can_resume as delivery in task" />
    <param name="condition2" value="is_target_user in task" />
  </condition>

  <condition name="can_resubmit in task" class="Workflow::Condition::LazyAND">
    <param name="condition1" value="is_delivery in task" />
    <param name="condition2" value="!has_target_user in task" />
  </condition>

  <condition name="can_halt in task" class="Workflow::Condition::LazyOR">
    <param name="condition1" value="is_delivery in task" />
  </condition>
</conditions>
```

Code 8: Example of the declarations of workflow conditions within XML configuration files

```perl
package Portal::Workflow::Action::Task::Edit;

use Moose;
use Workflow::Exception qw( workflow_error );
use Workflow::History::Extended;

BEGIN { extends 'Workflow::Action' }

sub execute {
  my ( $self, $wf ) = @_;

  my $wf_context   = $wf->context;
  my $app_context  = $wf_context->param('app_context');
  my $user         = $wf_context->param('user');
  my $entity       = $wf_context->param('entity');
  my $stash        = $wf_context->param('volatile_stash');
  my $source_group = $wf_context->param('assigned_to');
  my $target_group = $wf_context->param('assigned_to');

  keys %{$stash};
  while( my ( $key, $value ) = each %{$stash} ) {
    $entity->$key($value);
  }

  $entity->update();

  $wf_context->param( assigned_to => $target_group );

  $wf->add_history(
    Workflow::History::Extended->new({
       action       => 'edit in task'
      , description  => ''
      , user         => $user->username
      , user_group   => $source_group
      , target_group => $target_group
      , time_zone    => $wf->time_zone
    })
  );
}
```

Code 9: Example of a class implementing a concrete workflow action

```perl
package Portal::Workflow::Condition::Task::IsDeliveryGroup;

use Moose;
use Workflow::Exception qw( condition_error );

BEGIN { extends 'Workflow::Condition' }

sub evaluate {
  my ( $self, $wf ) = @_;

  my $app_context    = $wf->context->param('app_context');
  my $system_user    = $app_context->model('GIFRDB::User')->system_user($app_context);
  my $current_user   = $wf->context->param('user');
  my $entity         = $wf->context->param('entity');
  my $delivery_group = $entity->delivery_group->name;

  # This system user can do anything
  return if $current_user->username eq $system_user->username;

  my $does_role  = $current_user->does_role($delivery_group);

  condition_error "The user role not allowed to this action!" unless $does_role;

  return 1;
}
```

Code 10: Example of a class implementing a concrete workflow condition

A.3 Integration

```
package Portal::RemoteServiceFactory::HPSM::UpdateSR;

use utf8;

use WS::HPSM::Service::UpdateSR;
use WS::HPSM::Type::ServiceRequest;
use Portal::Callback::HPSM::RefillSR;

sub new {
  my ( $class, %params ) = @_;

  my $service = WS::HPSM::Service::UpdateSR->new({
    request_data    =>  $data,

    before_callback =>  Portal::Callback::HPSM::RefillSR->new({
        request_id  => $params{request_id}
      , state       => $params{state}
      , group_name  => $params{assigned_to}
      , operator    => $params{operator}
      , cause       => $params{cause}
      , resolution  => $params{resolution}
      , solution    => $params{solution}
    })
  });

  return $service;
};
```

Code 11: Example of a remote service factory of a given HPSM service

```
package Portal::Callback::HPSM::CreateSR;

use Moose;

BEGIN { extends 'WS::Callback' }

has 'request_id' => (
      is        => 'ro'
    , isa       => 'Str'
    , reader    => 'get_request_id'
    , writer    => 'set_request_id'

    , required  => 1
);

override 'do' => sub {
    my ( $self, $context, $service, $value ) = @_;

    $context->{external_references}->create({
          parent_key    => $self->get_request_id
        , parent_type   => 'Request'
        , external_key  => $value->number
        , external_type => 'SR'
    });
};
```

Code 12: Example of a Portal callback for an HPSM service