UNIVERSIDAD SIMÓN BOLÍVAR
DECANATO DE ESTUDIOS PROFESIONALES
COORDINACIÓN DE INGENIERÍA DE LA COMPUTACIÓN

DESARROLLO DEL MÓDULO PRINCIPAL Y ESTADÍSTICAS DE LA LIBRERÍA AUDITORÍAS TURPIAL

Por:
Stefani Carolina Castellanos Torres

INFORME DE PASANTÍA
Presentado ante la Ilustre Universidad Simón Bolívar
como requisito parcial para optar al título de
Ingeniero de la Computación

Sartenejas, Enero 2018

UNIVERSIDAD SIMÓN BOLÍVAR
DECANATO DE ESTUDIOS PROFESIONALES
COORDINACIÓN DE INGENIERÍA DE LA COMPUTACIÓN

DESARROLLO DEL MÓDULO PRINCIPAL Y ESTADÍSTICAS DE LA LIBRERÍA AUDITORÍAS TURPIAL

Por:
Stefani Carolina Castellanos Torres

Realizado con la asesoría de:
Tutor Académico: Prof. Angela Di Serio
Tutor Industrial: Ing. Pedro Romero

INFORME DE PASANTÍA
Presentado ante la Ilustre Universidad Simón Bolívar
como requisito parcial para optar al título de
Ingeniero de la Computación

Sartenejas, Enero 2018

RESUMEN

DEDICATORIA

"Es mucho más difícil juzgarse a sí mismo,
que juzgar a los otros.
Si consigues juzgarte rectamente
es que eres un verdadero sabio."
- Antoine de Saint-Exupéry. El Principito

"No vivas imaginando problemas
que no han ocurrido ni van a suceder;
disfruta del presente,
la vida hay que aprovecharla"
— Enrique Barrios. Ami, el niño de las estrellas.

AGRADECIMIENTOS

ÍNDICE

RESUMEN . . . iii
ÍNDICE . . . vi
LISTA DE TABLAS . . . ix
LISTA DE FIGURAS . . . x
LISTA DE SÍMBOLOS . . . xii
LISTA DE ABREVIACIONES . . . xii
GLOSARIO . . . xiii
INTRODUCCIÓN . . . 1
1 **ENTORNO EMPRESARIAL** . . . 2
1.1 Descripción . . . 2
1.2 Misión . . . 2
1.3 Visión . . . 2
1.4 Estructura . . . 2
2 **DEFINICIÓN DEL PROBLEMA** . . . 5
2.1 Antecedentes . . . 5
2.2 Planteamiento del problema . . . 6
2.3 Justificación . . . 6
2.4 Objetivo general . . . 7
2.5 Objetivos específicos . . . 7
3 **MARCO TEÓRICO** . . . 8
3.1 Auditoría . . . 8
3.2 Acciones auditables . . . 8
3.3 Microservicio . . . 8
3.4 Integración Contínua . . . 9
3.5 Pruebas automatizadas . . . 9
3.6 Patrón Modelo-Vista-Controlador . . . 9
3.7 Patrón Modelo-Vista-Plantilla . . . 10
3.8 Señales . . . 10

3.9 Mixins . . . 11
3.10 Serializadores . . . 11

4 **MARCO TECNOLÓGICO** . . . 12
4.1 Python . . . 12
4.2 Django . . . 12
4.3 Pytest . . . 13
4.4 JavaScript Object Notation (JSON) . . . 13
4.5 PostgreSQL . . . 13
4.6 MySQL . . . 13
4.7 SQLite . . . 13
4.8 Jsonfield . . . 14
4.9 Git . . . 14
4.10 Jenkins . . . 14
4.11 Lenguaje de marcado de hipertexto . . . 14
4.12 Javascript . . . 15
4.13 jQuery . . . 15
4.14 Amcharts . . . 15
4.15 Datatables . . . 15
4.16 Bootstrap . . . 15

5 **MARCO METODOLÓGICO** . . . 16
5.1 Turpial Agile Unified Process (TAUP) . . . 16
5.1.1 Fase de Concepción . . . 16
5.1.2 Fase de Construcción . . . 17
5.1.3 Fase de Transición . . . 17

6 **DESARROLLO** . . . 18
6.1 Fase de concepción . . . 18
6.1.1 Análisis de requerimientos . . . 18
6.1.2 Adaptación de la metodología a la pasantía . . . 20
6.1.3 Arquitectura propuesta del sistema . . . 21
6.1.4 Diseño del módulo *Core* . . . 22
6.1.5 Diseño del módulo de estadísticas . . . 23

6.1.6 Propuesta para la integración continua . . . 25
6.1.7 Plan de pruebas . . . 26
6.1.8 Planificación del desarrollo del proyecto . . . 27
6.2 Fase de construcción . . . 27
6.2.1 Construcción del módulo *Core* . . . 27
6.2.1.1 Preparación del ambiente de desarrollo . . . 27
6.2.1.2 Estructura de la tabla de auditorías . . . 27
6.2.1.3 Selección de un modelo auditable . . . 28
6.2.1.4 Registro de una acción auditable . . . 28
6.2.1.5 Instalación de la librería utilizando PIP . . . 31
6.2.1.6 Listados de auditoría . . . 32
6.2.1.7 Reestructuración de la arquitectura . . . 32
6.2.1.8 Integración continua . . . 34
6.2.1.9 Pruebas del módulo *Core* . . . 36
6.2.1.10 Permisología . . . 37
6.2.1.11 Análisis de desempeño . . . 38
6.2.1.12 Prototipo de tabla de auditorías descentralizada . . . 38
6.2.2 Construcción del módulo Estadísticas . . . 38
6.2.2.1 Creación del formulario de filtros . . . 38
6.2.2.2 Cálculos estadísticos . . . 40
6.2.2.3 Construcción de gráficos . . . 41
6.2.3 Plantillas del módulo Estadísticas . . . 42
6.2.3.1 Pruebas del módulo Estadísticas . . . 45
6.3 Fase de transición . . . 46
6.3.1 Integración de la librería con Turpial Calendar . . . 46
6.3.2 Puesta en producción . . . 47

CONCLUSIONES Y RECOMENDACIONES . . . 49
Conclusiones . . . 49
Recomendaciones . . . 49

APÉNDICES . . . 51
A MANIFIESTO ÁGIL . . . 52
B MANUAL DE METODOLOGÍA DE TURPIAL (RESUMEN) . . . 54

C BACKLOG DEL PROYECTO 59

LISTA DE TABLAS

6.1 Modelo de datos de la tabla "AuditableAction" 24
6.2 Comparación de Jenkins vs Fabric. 25
6.3 Modelo de datos de la tabla "AuditableAction" 28

LISTA DE FIGURAS

1.1 Diagrama de la estructura de Turpial Development 3

3.1 Diagrama del patrón MVC (creación propia). 10
3.2 Diagrama del patrón MVT (creación propia). 10

6.1 Arquitectura de la librería Auditorías Turpial 21
6.2 Arquitectura de tecnologías a utilizar 22
6.3 Modelo Entidad-Relación del ejemplo. 30
6.4 Gráfico de flujo del orden en el que se activan las señales para guardar un cambio en los campos "ManyToManyField". 30
6.5 Arquitectura final de la librería. 33
6.6 Estructura del módulo *Core* en el patrón MVT (creación propia). 34
6.7 Modelo Entidad-Relación de la base de datos de prueba (creación propia). . 36
6.8 Ejemplo del formato JSON para cargar un *Fixture* (creación propia). 37
6.9 Estructura del formulario de filtros (creación propia). 40
6.10 Ejemplo de "stats" (creación propia). 40
6.11 Definición de "StatisticsView" incluyendo el campo "charts_theme" (creación propia). 44
6.12 Estructura del módulo de Estadísticas (creación propia). 44
6.13 Diagrama Entidad-Relación de los nuevos modelos en las pruebas (creación propia). 47

LISTA DE ABREVIACIONES

AJAX	*Asynchronous JavaScript And XML*. JavaScript asíncrono y XML.
API	*Application Programming Interface*. Interfaz de programación de aplicaciones.
CSV	*Comma Separated Values*. Archivo de valores separados por comas.
HTML	*HiperText Markup Language*. Lenguaje de marcado de Hipertexto
HTTP	*Hipertext Transfer Protocol*. Protocolo de Transferencia de Hipertexto
INVEST	*Independent Negotiable Valuable Estimable Small Testable*. Independiente Negociable Estimable Small Testable.
MVC	*Model-View-Controller*. Modelo-Vista-Controlador.
MTV	*Model-Template-View*. Modelo-Plantilla-Vista.
PIP	*Pip Installs Packages*. Pip instala paquetes.
PDF	*Portable Document Formats*. Formato de documento portátil.
SVG	*Scalable Vector Graphics*. Gráficos Vectoriales Escalables.
URL	*Uniform Resource Locator*. Localizador Uniforme de Recursos

GLOSARIO

AJAX: técnica de desarrollo web para crear aplicaciones interactivas. Estas aplicaciones se ejecutan en el cliente y mantiene la comunicación asíncrona con el servidor en segundo plano. De esta forma es posible realizar cambios sobre las páginas sin necesidad de recargarlas, mejorando la interactividad, velocidad y usabilidad en las aplicaciones.

Backend: parte de las aplicaciones que procesa la entrada desde la interfaz de usuario.

Callback: cualquier código ejecutable que es suministrado como argumento a otro código. Este es ejecutado en cualquier momento.

Candlestick: tipo de gráfico financiero que es utilizado para describir la fluctuación de los precios.

Fixture: colección de datos que pueden ser utilizadas para poblar la base de datos.

Framework: conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar.

Frontend: parte del software que interactúa con los usuarios.

In-Process: programa ejecutable que funciona como servicio para otro en lugar de la versión instalable. Ejemplo: DLL.

Look-and-feel: aspecto del sitio para el usuario y lo que siente cuando él al interactuar con el mismo.

PIP: repositorio de software (paquetes) para el lenguaje de programación Python.

Plugin: aplicación o programa que se relaciona con otro para agregarle una función nueva específica.

Portable: aplicación informática que puede ser utilizada, sin instalación previa, en un ordenador que posea el sistema operativo para el que fue programada.

Query: consulta realizada contra una base de datos. Se usa para obtener datos, modificarlos o bien borrarlos.

Queryset: en Django, representa una colección de objetos de la base de datos. Puede tener uno o muchos filtros. En términos de SQL, es equivalente a una sentencia SELECT y los filtros equivalen a la cláusula WHERE.

Responsive: es una filosofía de diseño y desarrollo cuyo objetivo es adaptar la apariencia de las páginas web al dispositivo que se esté utilizando para visitarlas.

Sass: un metalenguaje de Hojas de Estilo en Cascada (CSS).

Scatter: diagrama matemático que utiliza coordenadas cartesianas para mostrar valores, típicamente de un conjunto de datos con dos variables.

Sprints: bloques temporales cortos y fijos (iteraciones). Cada iteración tiene que proporcionar un resultado completo, un incremento de producto que sea potencialmente entregable.

Triggers: acciones (funciones) que se ejecutan cuando sucede algún evento sobre las tablas de una base de datos, a las que se encuentra asociado.

INTRODUCCIÓN

CAPÍTULO 1

Entorno empresarial

En este capítulo se describe en ambiente de la empresa Turpial Development, en la cual se llevó a cabo la pasantía. Se presenta la misión, visión y estructura de la empresa, así como el cargo desempeñado por el pasante durante este período.

1.1 Descripción

Turpial Development es una empresa mediana, con cuatro años en el mercado que está enfocada en el desarrollo de sistemas y aplicaciones web y móviles. Fundada e integrada por jóvenes venezolanos, ofrece soluciones que cumplen con altos estándares de usabilidad, diseño y funcionalidad [27].

1.2 Misión

La empresa tiene como misión "prestar servicios y consultoría en diseño y desarrollo de soluciones web, a la medida del cliente, caracterizadas por una alta calidad, excelente soporte y experiencia de usuario" [27] .

1.3 Visión

Su visión es "servir de plataforma para el desarrollo y éxito de nuevos emprendimientos en el área web" [27].

1.4 Estructura

En la figura 1.1 se muestra la estructura organizacional de Turpial Development, la cual está conformada por cuatro departamentos:

Dirección de Operaciones

Ente encargado del "funcionamiento de todos los procesos de soporte de la empresa, como administración, recursos humanos, contabilidad y compras con el fin de garantizar el correcto desarrollo de los procesos principales de la empresa" [27].

Dirección de Proyectos

Ente encargado de "atender a las necesidades y requerimientos de los clientes, sirviendo de enlace con las direcciones de desarrollo y diseño para garantizar la calidad del producto o entrega" [27].

Dirección de Desarrollo

Ente encargado de "la conceptualización y desarrollo de las necesidades del cliente, implementando e integrando el diseño acordado y las funcionalidades requeridas por dicho cliente. Se divide en tres departamentos: Conceptualización, *Backend* y *Frontend*" [27].

Dirección de Diseño

Ente encargado de "la conceptualización y diseño de la interfaz gráfica de los proyectos en función de las necesidades del cliente y de la mano con las decisiones tomadas por la Dirección de Desarrollo" [27].

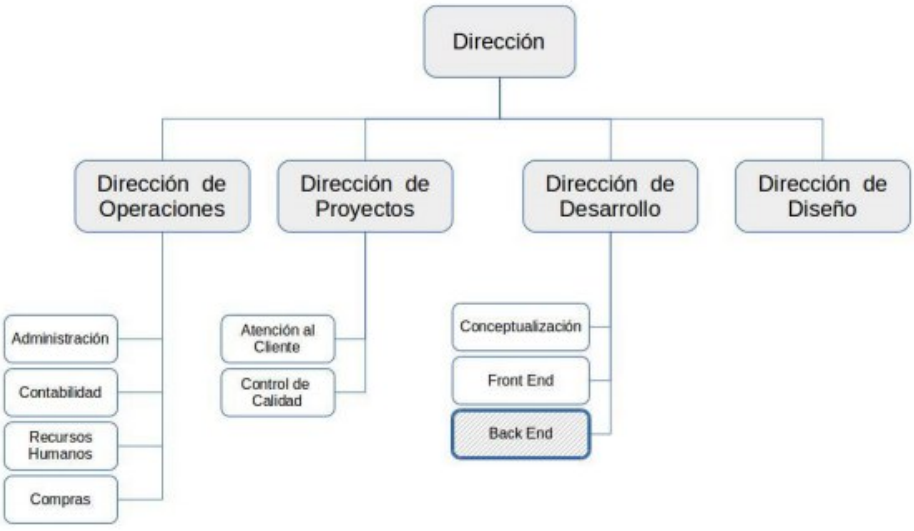

Figura 1.1: Diagrama de la estructura de Turpial Development

Ubicación del pasante

El desarrollo de la pasantía fue llevado a cabo en la Dirección de Desarrollo de la empresa. El pasante fue asignado al departamento de *Backend* con el cargo de Pasante, bajo la supervisión del Tutor Industrial, el Ingeniero Pedro Romero. El pasante debe cumplir tareas tales como: levantar requerimientos, diseñar y desarrollar las funcionalidades de la solución propuesta para el sistema.

CAPÍTULO 2

Definición del problema

En este capítulo se explica la motivación de la empresa para desarrollar el proyecto y la justificación del mismo. Además, se establecen los objetivos que se esperan alcanzar durante la pasantía.

2.1 Antecedentes

En ocasiones, las aplicaciones necesitan un sistema que controle y supervise el uso de la información, bien sea para cumplir con regulaciones o para observar el comportamiento de la información considerada sensible para el negocio. Turpial Development desea contar con una librería confiable y de fácil instalación que provea estas funcionalidades, de manera que puedan ser ofrecidas como un servicio a sus clientes. Esta librería debe poder instalarse en el *framework* escrito en Python, Django, puesto que la empresa estableció como estándar de desarrollo para sus aplicaciones, estas tecnologías

Existen algunas librería disponibles que son capaces de realizar estos procedimientos. Una de las más populares es django-reversions, una extensión de Django que provee control de versiones para los modelos y la posibilidad de restaurar la base de datos a una versión anterior. La empresa utilizó esta librería en algunos proyectos, sin embargo, su integración delega demasiada responsabilidad al programador por lo que es susceptible a errores humanos y agrega una complejidad, que en muchos casos, es innecesaria para el comportamiento que se desea lograr.

Además, de esta extensión, hay otras que no han sido utilizadas por Turpial Development (Apéndice D) que ofrecen estas características, utilizando otras funciones que provee el *framework*, sin embargo los problemas explicados anteriormente se mantienen.

2.2 Planteamiento del problema

Motivados por la cantidad de clientes que requieren un sistema de auditorías y porque la extensiones de Django no son fáciles de implementar, Turpial Development, se vio en la necesidad de crear su propia librería. Esta permitirá llevar la trazabilidad de la información sensible del negocio a través de auditorías de manera sencilla.

La empresa requiere que la librería sea fácil de integrar en otro proyecto, disponga de facilidades para identificar cuáles acciones del sistema deben ser auditables y almacenar las auditorías en una base de datos. Asimismo, debe crear listados y personalizar su apariencia sin necesidad de que el desarrollador cree las plantillas desde cero.

Adicionalmente, ninguna de las librerías disponibles cuentan con un módulo que permita visualizar los datos obtenidos a través de estadísticas o reportes. El desarrollador debe realizar esta labor por él mismo; analizando la información recaudada y ajustando la estructura de datos para hacer uso de otras librerías que dispongan de la capacidad de realizar gráficas o generar PDF. En este sentido, se desea que el producto a desarrollar proporcione un módulo capaz de realizar los cálculos necesarios para mostrar estadísticas referentes a la información recaudada por las auditorías y sus respectivas gráficas; y un módulo que genere reportes CVS o PDF de los diferentes listados y que sean personalizables.

2.3 Justificación

La empresa Turpial Development decidió invertir en el desarrollo de una librería para la gestión de auditorías que se ajuste tanto a las necesidades de sus programadores, como a la de sus clientes; que agilice el desarrollo de cualquier proyecto que requiera un historial de las operaciones y posea una base de datos relacional. La arquitectura planteada por la empresa consiste en realizar dicha librería bajo la estructura de microservicio. De esta manera, podrá incluirse fácil y rápido dentro de las aplicaciones, a través de configuraciones y ajustes menores que no interfieran con su correcto funcionamiento. Se desarrollará como una aplicación en Django que se agregará como un módulo más al sistema que lo instale.

Adicionalmente, se desea utilizar alguna herramienta automatizada para integrar el trabajo de manera continua, empaquetar, probar y desplegar la librería de manera que se actualice con rapidez y esté disponible en todo momento. Asimismo, la empresa considera que es de vital importancia realizar pruebas automatizadas que aseguren la calidad de sistema que se desea entregar y que las pruebas finales se realicen sobre un proyecto de uso interno denominado Turpial Team.

2.4 Objetivo general

Implementar, probar y presentar las funcionalidades de selección, gestión y listados de auditorías y todas las funcionalidades del módulo Estadísticas de la librería de Auditorías Turpial e implantar un sistema de integración continua con el repositorio.

2.5 Objetivos específicos

- Diseñar el módulo *Core* de la librería Auditorías Turpial y construir parcialmente de dicho módulo, específicamente las funcionalidades de selección de modelos a auditar y gestión de las auditorías, bajo una arquitectura de microservicios.
- Diseñar y construir el módulo Estadísticas de la librería Auditorías Turpial, bajo una arquitectura de microservicios.
- Diseñar e implantar un sistema de integración continua con el repositorio donde se dispondrá la librería Auditorías Turpial.

CAPÍTULO 3

Marco teórico

En este capítulo se presentan los aspectos teóricos que sustentan y respaldan el desarrollo del proyecto de pasantía. A continuación se describen los conceptos que permiten explicar el problema planteado.

3.1 Auditoría

Es una revisión sistemática para determinar si la calidad de las actividades cumplen con los acuerdos planificados y si estos están implementados efectivamente y son adecuados para alcanzar sus objetivos. Ofrecen una oportunidad de mejora al sistema y pueden ser llevadas a cabo para cumplir con normas regulatorias. Las auditorías se pueden aplicar a sistemas, procesos, programas o servicios y pueden ser internas (realizadas por la misma empresa) o externas (realizadas por proveedores) [28].

3.2 Acciones auditables

Se considera una acción auditable todo flujo del sistema que cree, edite o borre alguna información sensible para el mundo de negocio. Son registradas incluyendo información sobre quién realizó la acción, qué acción se intentó realizar y cuándo ocurrió la acción.

3.3 Microservicio

Es un tipo de arquitectura que consiste en desarrollar una aplicación como un conjunto de pequeños servicios. Dichos servicios son procesos autónomos, cohesivos e independientes que suelen interactuar con otros componentes a través de mensajes [8]. Se enfocan en resolver un único problema y funcionan de manera aislada; si se presentan una falla no se propaga y puede ser atendido más rápidamente [18].

Estos servicios se manejan de manera descentralizada y hacen uso de un despliegue completamente automatizado. Cada uno de ellos pueden ser escritos en diferentes lenguajes

y tecnologías para almacenar información y aún así, interactuar y compartir información [18].

3.4 Integración Contínua

Es una práctica de desarrollo de *software* en la que los miembros del equipo combinan su trabajo de forma periódica, usualmente cada día. Cada integración está verificada por una herramienta automatizada que empaquete y pruebe el código para detectar errores lo más rápido posible [11], de esta manera se puede mejorar la calidad del *software* y reducir el tiempo en validar y publicar actualizaciones.

3.5 Pruebas automatizadas

Las pruebas tienen como objetivo ejercitar el código para detectar errores y verificar que el *software* satisface los requerimientos especificados para asegurar su calidad. Estas son realizadas desde el punto de vista del usuario y las funcionalidades son probadas ingresando información y examinando la salida [9].

Esta labor puede ser larga y repetitiva, por lo que es conveniente contar con herramientas que provean métodos que faciliten el proceso de escribir y ejecutar casos de prueba y así, reducir significativamente el esfuerzo y tiempo invertido por los desarrolladores [9]. A este conjunto de casos de pruebas se le conoce como prueba automatizada.

3.6 Patrón Modelo-Vista-Controlador

Este patrón de diseño asigna objetos en una aplicación a uno de tres roles: modelo, vista o controlador y define cómo se comunican entre ellos. La colección de objetos de cada tipo puede ser referido como una "capa" [2].

- Modelo: controla el comportamiento y la data de la aplicación, responde las solicitudes de información (usualmente desde la vista) y las instrucciones de cambio de estado (usualmente desde el controlador).
- Vista: maneja la presentación de la información.
- Controlador: interpreta las entradas del usuario y le informa al modelo y/o vista para cambiar lo que sea apropiado [16].

Es importante notar que la vista y el controlador dependen del modelo, sin embargo el modelo es independiente. Esta separación permite probar el modelo aparte de la presentación visual.

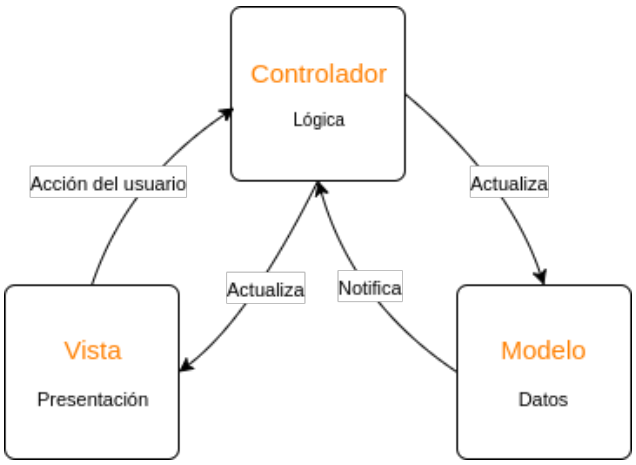

Figura 3.1: Diagrama del patrón MVC (creación propia).

3.7 Patrón Modelo-Vista-Plantilla

Es una adaptación del patrón MVC, en el cual la "vista" define cuál dato es presentado y su comportamiento, no como se muestra. El dato es obtenido a través de la función de *callback* para pedir una URL en particular. Por otro lado, la "vista" delega a la "plantilla" la presentación de la información.

En el caso de Django, el "controlador" es el *framework* en sí: "la maquinaria que envía una petición a la vista apropiada, de acuerdo a la configuración de URL de Django" [6].

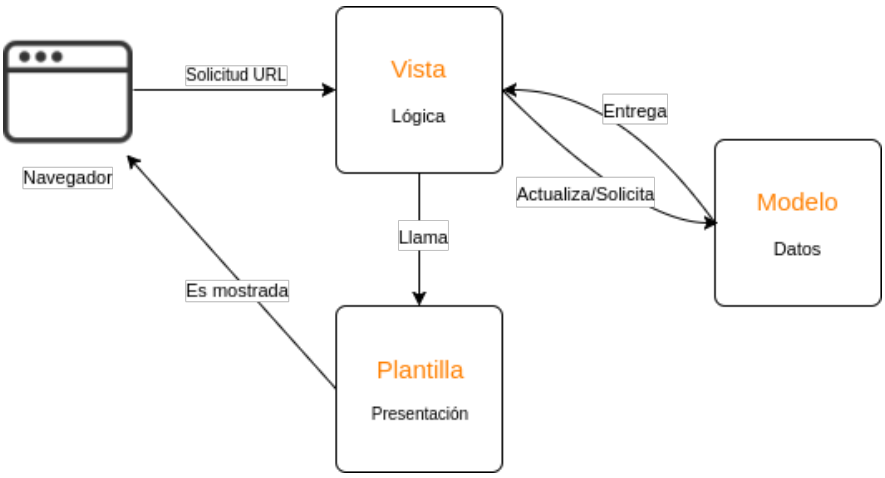

Figura 3.2: Diagrama del patrón MVT (creación propia).

3.8 Señales

Permiten a las aplicaciones ser notificadas cuando ocurra una acción en algún otro lugar, es decir, las señales permiten a ciertos emisores notificar a un conjunto de receptores que alguna

acción está siendo ejecutada [7]. En términos de *software*, son el análogo a interrupciones de *hardware* [25].

3.9 Mixins

Es un estilo de programación en el cual las unidades de funcionalidad son creadas en una clase y se incorporan en otras [10]. Pueden entenderse como "un subclase abstracta que puede ser usada para especializar el comportamiento de una variedad de padres" [4].

Usualmente, los *Mixins*, definen nuevos métodos que realizan alguna acción y luego llaman a los métodos del padre correspondiente; pueden utilizarse en varias clases de la jerarquía y sin importar en qué clases sean usadas.

Hay varias razones para usar *Mixins*: extienden las clases existentes sin tener que editar, mantener o combinar el código fuente; mantienen el proyecto en componentes separados; facilitan la creación de nuevas clases con funcionalidades pre-fabricadas y que se puede combinar según sea la necesidad [10].

3.10 Serializadores

La *Serialization* es un proceso que consiste en convertir un objeto en un formato que sea fácilmente transmitido a través de la red o que pueda alojarse en un lugar de almacenamiento consistente (i.e base de datos). Los *serializers* pueden ser utilizados, simplemente, para que un objeto sea legible por humanos.

CAPÍTULO 4

Marco tecnológico

En este capítulo se describen los aspectos tecnológicos relevantes para la comprensión del proyecto, así como las herramientas utilizadas durante la pasantía.

4.1 Python

Lenguaje de programación interpretado y fácil de entender. Posee estructuras de datos de alto nivel y una aproximación sencilla al paradigma de programación orientado a objetos. Cuenta con una amplia variedad de librerías que fomentan la reutilización de código; también posee tipos de datos dinámicos y una sintaxis simple para facilitar el rápido desarrollo de aplicaciones [22]

4.2 Django

Framework de código abierto, escrito en Python y está basado en el patrón Modelo-Plantilla-Vista. Proporciona diversas funcionalidades reutilizables para desarrollar, rápidamente, aplicaciones web escalables [6].

Django está equipado con un conjunto de Clases y *Mixins* que pueden ser heredados y proveen el comportamiento estándar de alguna acción particular (i.e crear un elemento). Adicionalmente, el programador puede crear otros nuevos para extender estas funcionalidades.

También incluye un despachador de señales que ayuda a las aplicaciones desacopladas a recibir notificaciones cuando alguna acción ocurra en algún otro lugar en el *framework*. Algunos de los eventos sobre los que notifica Django son la inserción, actualización y eliminación de elementos de la base de datos y migración de la misma.

4.3 Pytest

Framework escrito en Python, que facilita la escritura de complejas pruebas de funcionalidad para aplicaciones y librerías y así, asegurar la calidad del *software* que será entregado [21]. Pytest, permite controlar la ejecución de pruebas automatizadas y comparar los resultados obtenidos con los esperados [21].

4.4 JavaScript Object Notation (JSON)

Formato para intercambiar datos, ligero, independiente del lenguaje y está basado en textos. JSON, define un pequeño conjunto de reglas para la representación de datos estructurados que sean portables [5]. También, puede representar cuatro tipos primitivos (cadenas de caracteres, números, booleanos y null) y dos tipos estructurados (objetos y arreglos) [5].

4.5 PostgreSQL

Sistema de base de datos relacional de código abierto, con una arquitectura que goza de buena reputación gracias a su confiabilidad e integridad de los datos [26]. Puede ser implantada en la mayoría de los sistemas operativos y soporta claves foráneas, conjunciones, vistas, *triggers*, procedimientos y la mayoría de los tipos de datos [26].

Este manejador de base de datos posee un tamaño ilimitado (depende del *hardware*), con un máximo de 32 TB por tabla, 1 GB por campo, entre 250 y 1600 columnas por tabla y sin límites en la cantidad de índices por tablas [26].

4.6 MySQL

Sistema de base de datos relacional de código abierto que soporta múltiples plataformas y las todas la operaciones de SQL [19]. Soporta grandes volúmenes de datos, más de 50 millones de registros; puede manejar 200.000 tablas y hasta 64 índices por tablas[19].

4.7 SQLite

Librería *in-process* que implementa un motor de base de datos SQL transaccional, de código abierto, que es autocontenido, no necesita configuración y no utiliza servidores puesto que escribe directamente a los archivos de disco [24]. Posee todas las funcionalidades de una base de datos SQL completa: múltiples tablas, índices, *triggers* y vistas [24]. Es

multi-plataformas, se puede copiar libremente los archivos entre sistemas con diferentes arquitecturas [24].

El tamaño máximo de base de datos es de 140 TB y 1 GB por fila, máximo 32767 columnas por tabla dependiendo del tipo de columna [24]. La cantidad máxima de índices y tablas está estrechamente relacionada con la cantidad máxima de páginas (un poco más de 2 billones) ya que estos ocupan al menos una página del archivo de la base de datos [24].

4.8 Jsonfield

Plugin que crea un campo en Django que permite almacenar JSON en los modelos. Este, se ocupa de la serialización y la validación del campo en cuestión [13]. Aunque PostgreSQL tiene soporte nativo de los campos tipo JSON, Jsonfield, utiliza una abstracción para asegurar la compatibilidad con el resto de los manejadores de base de datos relacionales, con los que Django tiene integración [13]. Esto es de suma importancia si se desea crear una librería. El campo se traduce a uno de tipo texto.

4.9 Git

Sistema de control de versiones de código abierto, diseñado para administrar cualquier tipo de proyecto con rapidez y eficiencia [12]. Git, dispone de facilidades para llevar el seguimiento de los cambios realizados, soportar el desarrollo no-lineal, cambiar fácilmente de contexto y realizar experimentos sin afectar las versiones entregables del proyecto [12].

4.10 Jenkins

Servidor de automatización de código abierto y de fácil instalación que puede ser utilizado para automatizar tareas relacionadas con el empaquetado, pruebas y despliegue de un *software* [14]. Puede ser utilizado como un servidor de integración continua en donde la versión más reciente del proyecto sea descargada, se ejecuten las tareas descritas anteriormente y si ocurre algún fallo se le notifique a los programadores [14]. Adicionalmente, Jenkins dispone de un gran número de *plugins*, por lo que se adapta a casi cualquier proyecto sin importar qué tecnología se estén usando [14].

4.11 Lenguaje de marcado de hipertexto

Por sus siglas en inglés, *HiperText Markup Language* (HTML), es un formato de datos simple, usado para crear documentos portables de una plataforma a otra y ha sido utilizado

ampliamente en la World Wide Web desde 1990 [3]. HTML, describe la manera en la que la información es presentada en las plantillas, usualmente una página web.

4.12 Javascript

Lenguaje interpretado, multi-paradigma y dinámico que soporta estilos de programación funcional, orientado a objetos e imperativa, así como funciones de primera clase [17]. Es comúnmente utilizado como el lenguaje de *script* para páginas web.

4.13 jQuery

Librería de JavaScript que simplifica la manipulación de documentos HTML, manejo de eventos, animaciones y AJAX [15]. Cuenta con una API fácil de usar que funciona en una gran cantidad de navegadores [15].

4.14 Amcharts

Librería de JavaScript que permite añadir fácilmente gráficos interactivos a los sitios web y aplicaciones [1]. Es compatible con todos los navegadores modernos y la mayoría de los antiguos. Posee facilidades para crear gráficos de torta, barras, línea, *candlestick*, *scatter* entre otros, y cualquier combinación de ellos, exportar e importar los datos [1]. También cuenta con *plugins* para extender su funcionalidad, es *responsive*, fácil de personalizar y soporta múltiples lenguajes [1].

4.15 Datatables

Plugin para la librería jQuery de JavaScript. Añade interacción a cualquier tabla en HTML, dispone de funciones de búsqueda, paginación, ordenamiento, filtros, entre otras [23]. Además, provee facilidades para exportar diferentes tipos de archivo como CSV, PDF, XLS e incluso imprimir el contenido de la tabla [23].

4.16 Bootstrap

Framework de código abierto para desarrollar rápidamente aplicaciones, tanto web como móviles, utilizando HTML, CSS y JavaScript [20]. También usa variables y *mixins* de Sass, tiene un sistema de cuadrícula (*grid*) *responsive*, gran cantidad de componentes pre-fabricados y poderosos plugins construidos con jQuery [20].

CAPÍTULO 5

Marco metodológico

5.1 Turpial Agile Unified Process (TAUP)

TAUP es una metodología ágil creada por la empresa Turpial Development basada en los principios del "manifiesto ágil" que brinda a su equipo de trabajo una manera eficaz de llevar a cabo el desarrollo de *software*. En el apéndice A, se encuentra mayor detalle sobre el manifiesto ágil.

TAUP considera que la prioridad es la satisfacción del cliente mediante entregas continuas, las cuales deben realizarse en lapsos entre una semana y un mes. Está conformada por tres fases: concepción , construcción y transición. En el apéndice B se especifican los diferentes aspectos relacionados a la metodología.

5.1.1 Fase de Concepción

El objetivo de esta etapa es que el equipo de desarrollo profundice la comprensión de los requerimientos del sistema. Para ello, se escriben las Historias de Usuario (HU) con su respectiva prioridad y nivel de dificultad. Estas son establecidas por el cliente y el equipo, respectivamente, haciendo uso de la siguiente escala: alta, media alta, media, media baja y baja. También, el equipo, asigna *Story Points* a cada HU que ponderan el esfuerzo que se requiere para culminarla.

Las HU cuentan con criterios de aceptación bien definidos que brindan más detalle respecto a los aspectos que validan su completitud y poseen pruebas de aceptación para asegurar que el producto cumpla con los estándares establecidos. Adicionalmente, deben cumplir con ciertos requisitos para que se comience a desarrollar (*Definition of Ready* o DoR) y otros para que sea considerada como culminada (*Definition of Done* o DoD).

Por último, se construye la lista de todas las Historias de Usuarios (*Backlog*) en el orden que se van a ejecutar tomando en cuenta la prioridad y el riesgo de la tarea. Adicionalmente,

se elabora un calendario en el cual se agregarán los *Sprints* que sean necesarios para que todas las Historias de Usuarios sean cumplidas a cabalidad, llamado *Release Plan* (Apéndice D).

5.1.2 Fase de Construcción

El enfoque de esta fase es desarrollar todo lo planteado en la Concepción, validando así, la arquitectura planteada. La metodología se basa en *Sprints* o iteraciones que son bloques de tiempo de duración corta y fija, en los que al final se puede obtener un producto potencialmente entregable.

El primer día de cada *Sprint* se lleva a cabo la Reunión de Planificación, en la que todos los miembros del equipo revisan lo que se tiene planteado en el *Release Plan*. Además, se verifica que cada una de esas Historias de Usuario estén en estado DoR para que sean divididas en tareas y se aclare cualquier duda acerca de lo que se va a realizar.

Durante el *Sprint*, diariamente, se lleva a cabo una reunión corta llamada *Daily Stand Ups Meeting*, en la cual se discute el estado de las tareas para el momento, que se va a hacer y cuáles obstáculos se han presentado en el desarrollo de alguna HU.

El último día de cada *Sprint*, los miembros del equipo deben reunirse para realizar la Revisión de Iteración, en la cual estará presente el cliente para mostrarle todos los avances. Luego, se realiza la Reunión de Retrospectiva, en donde se discute que hizo bien, que se puede mejorar y a que se compromete.

5.1.3 Fase de Transición

En esta fase se probará todo lo desarrollado en la fase de Construcción. Estas pruebas funcionales y no funcionales, verifican que el proyecto pueda ser utilizado en un ambiente de producción, además se creará un manual para el usuario, el cual le dará la capacitación necesaria al cliente para poder utilizar sin problemas el sistema. También se validará la documentación, comprobando que todo el código esté correctamente explicado para que pueda ser entendido con facilidad en caso de que sea necesario realizar cambios.

Luego de que el sistema esté completo, se tendrá que desplegar en el ambiente de producción. Al terminar este proceso se otorgará un tiempo prudencial al cliente para efectuar pruebas y se llevan a cabo las correcciones pertinentes.

CAPÍTULO 6

Desarrollo

En el presente capítulo se detalla la planificación siguiendo la metodología Turpial Agile Unified Process (TAUP). Adicionalmente, se describe la evolución del proyecto y sus dificultades, así como las actividades realizadas que llevaron a cumplir los objetivos planteados y logros adicionales.

6.1 Fase de concepción

En esta sección se detallan las funcionalidades de los módulos Principal (*Core*) y Estadísticas de la librería Auditorías Turpial según los requerimientos del cliente; y se muestra el diseño de la solución y planteamiento de la arquitectura. También, se elaboran los documentos según TAUP y se definen las tecnologías necesarias para el desarrollo del proyecto. Este proceso abarcó las primeras cuatro semanas de la pasantía.

6.1.1 Análisis de requerimientos

Antes de tomar alguna decisión de implementación, fue necesario establecer cuál es la tecnología a la que va dirigida el producto final y cuáles son las funcionalidades mínimas que debe poseer. En primer lugar, se decidió que se desarrollaría una librería en Django, para Django, ya que la empresa suele utilizar este herramienta en sus aplicaciones; y utilizaría una base de datos relacional, en particular PostgreSQL, MySQL o SQLite porque se integran fácilmente al *framework* .

En segundo lugar, se determinaron las Historias de Usuario (HU). La librería consta de tres módulos: *Core*, Estadísticas y Reportes. El líder del proyecto se encargó de las HU correspondientes al módulo Principal y el pasante realizó el levantamiento de requerimientos del módulo de Estadísticas (ver apéndice C) según las necesidades del cliente. Adicionalmente, elaboró los documentos correspondientes, Documento de Requerimientos y *Release Plan* siguiendo las plantillas de la empresa.

En este caso en particular, el rol del cliente lo interpretó la empresa misma, puesto que el producto será ofrecido como un servicio a clientes externos. El rol de *product owner* lo desempeñó el tutor industrial para gestionar el desarrollo de la pasantía.

Para escribir las HU, es indispensable contar con el actor que ejecuta alguna acción específica, por lo que se distinguieron dos tipos de usuarios:

- El programador, quien descargará la librería y la incluirá en la aplicación de Django que está desarrollando.
- Los "usuarios" finales, quienes utilizarán la aplicación en donde se instale la librería y la interfaz gráfica provista.

En líneas generales, el módulo *Core* debe contar con las siguientes características:

- Seleccionar cuál modelo (tabla) es auditable.
- Registrar el autor, acción, fecha, estado anterior y estado actual de una instancia particular en formato JSON. Las acciones auditables son: Crear, Actualizar y Eliminar.
- Listar todas las operaciones, filtrarlas y ordenarlas.
- Restringir el acceso del personal no autorizado a los listados.
- Proveer etiquetas personalizadas para las plantillas de los listados que faciliten la inclusión de los mismos.

El módulo de Estadísticas debe proveer el cálculo del total de auditorías, cantidad de modelos auditables, porcentaje de cobertura, porcentaje de crecimiento de los datos, promedio por día, mínimo y máximo. Dichos resultados pueden estar filtrados por un rango de tiempo, por acción, por autor y por modelo. Asimismo, debe contar con facilidades para incluir los gráficos que representen los cálculos mencionados anteriormente.

El módulo de Reportes ofrece la posibilidad de generar archivos sobre los listados en diversos formatos (CSV y PDF) y personalizar su apariencia con opciones como modificar los márgenes, espacios, incluir el nombre y logo del sistema, entre otros. Adicionalmente, la librería debe ser mantenible, eficiente, simple, confiable, escalable y fácil de integrar y configurar.

Por otro lado, es indispensable que se instalen automáticamente las dependencias de la librería en el sistema que la utilice para facilitar su uso y evitar errores. También, se requiere

que la librería se actualice mediante el uso de una herramienta de integración continua.

En esta pasantía se abarcarán las funcionalidades correspondientes a la selección del modelo auditable, registro de traza de auditoría y listados (sin filtros ni ordenamiento) del módulo *Core* y completamente el módulo de Estadísticas con sus respectivas pruebas automatizadas. También se incluye la instalación y configuración del sistema de integración continua. El módulo de Reportes está fuera del alcance.

6.1.2 Adaptación de la metodología a la pasantía

En el capítulo anterior se explicó la metodología TAUP, sin embargo, dependiendo del proyecto que se desea desarrollar, se pueden realizar algunas modificaciones que mejoren la dinámica y la velocidad del equipo o porque el cliente así lo requiera.

Se ideó un código compuesto por una letra y un número que facilita referenciar las HU. La letra representa el módulo a la que pertenece, *Core* o Estadísticas, y el número denota el orden en que fue concebida. Adicionalmente, se utilizó una modificación para las escalas de prioridad y riesgo, que está conformada por tres valores: alta, media y baja. Para más información sobre las HU desarrolladas, leer el Apéndice C.

Por otro lado, se agregaron nuevos criterios a la *Definition of Ready*, con lo que se tiene lo siguiente:

- Debe ubicarse dentro de uno de los módulos del proyecto.
- Debe de tener asignado una prioridad por el cliente.
- Debe de tener asignado un riesgo por el equipo de desarrollo.
- Debe de tener asignado su respectivo puntaje.
- (Opcional pero deseado) Debe de tener una breve descripción, aclaratoria o criterio adicional según sea el caso.

Asimismo, se amplió la *Definition of Done* para agregar las pruebas automatizadas de cada HU. Posee los siguientes estatutos:

- Debe realizarse la codificación respectiva
- El código generado debe estar debidamente documentado para facilidad de programador

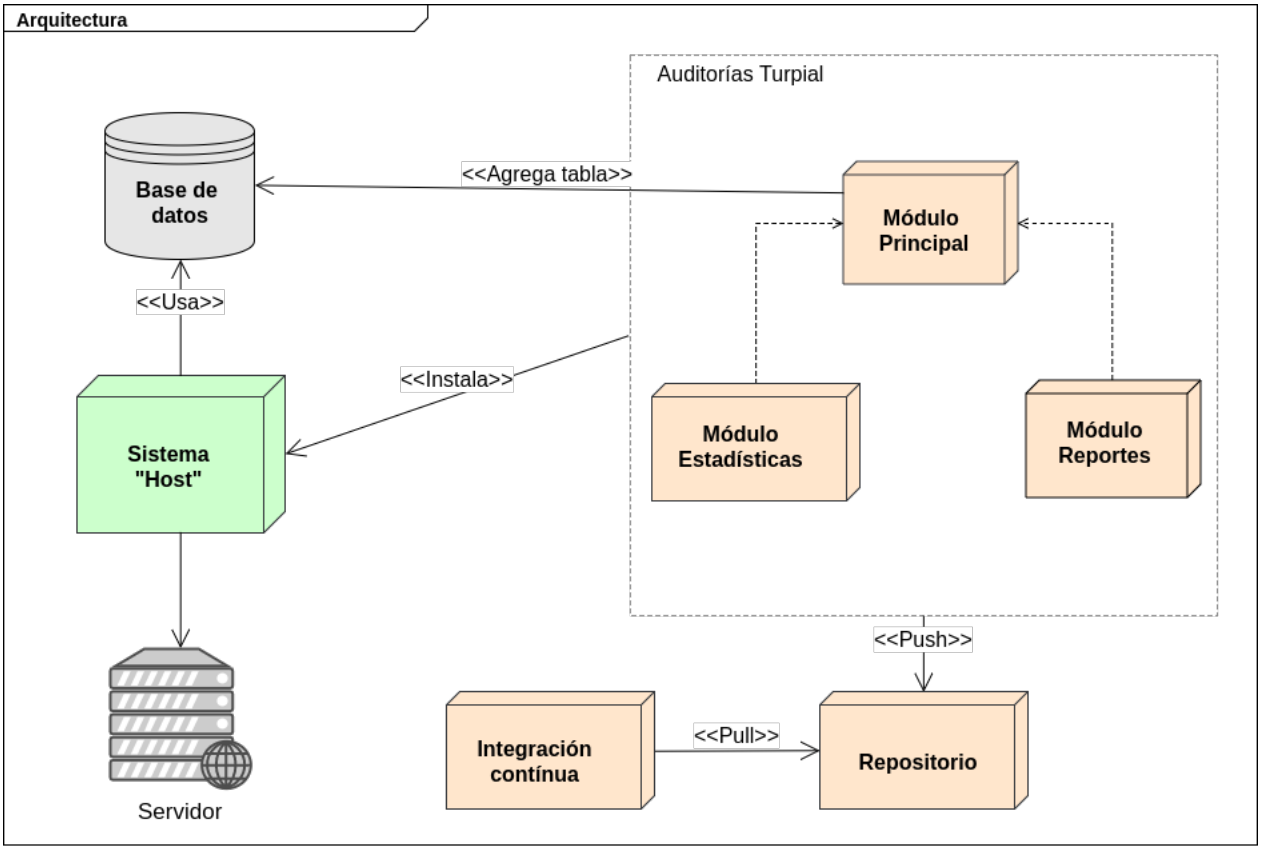

Figura 6.1: Arquitectura de la librería Auditorías Turpial

- Debe de realizarse la documentación respectiva (de ser necesaria) de todos los aspectos de configuración asociados al desarrollo y buen funcionamiento de la historia de usuario.
- Deben realizarse pruebas automatizadas a la codificación generada.
- Debe presentarse la nueva funcionalidad al cliente.
- Debe estar disponible en el repositorio.

6.1.3 Arquitectura propuesta del sistema

El planteamiento inicial (Figura 6.1) consistía en desarrollar cada módulo de la librería en una aplicación de Django distinta, las cuales se instalarían por separado en el sistema, el cual será referido como "Host" para simplificar la notación. Los módulos de Estadística y Reportes serían ofrecidos como microservicios dependientes del módulo principal pero independientes entre ellos, de manera que si alguno falla, el otro no sea afectado.

Se decidió que el módulo Principal agregaría una nueva tabla en la base de datos ya existente del "Host" en la que mantendrá la información acerca de las auditorías y los otros

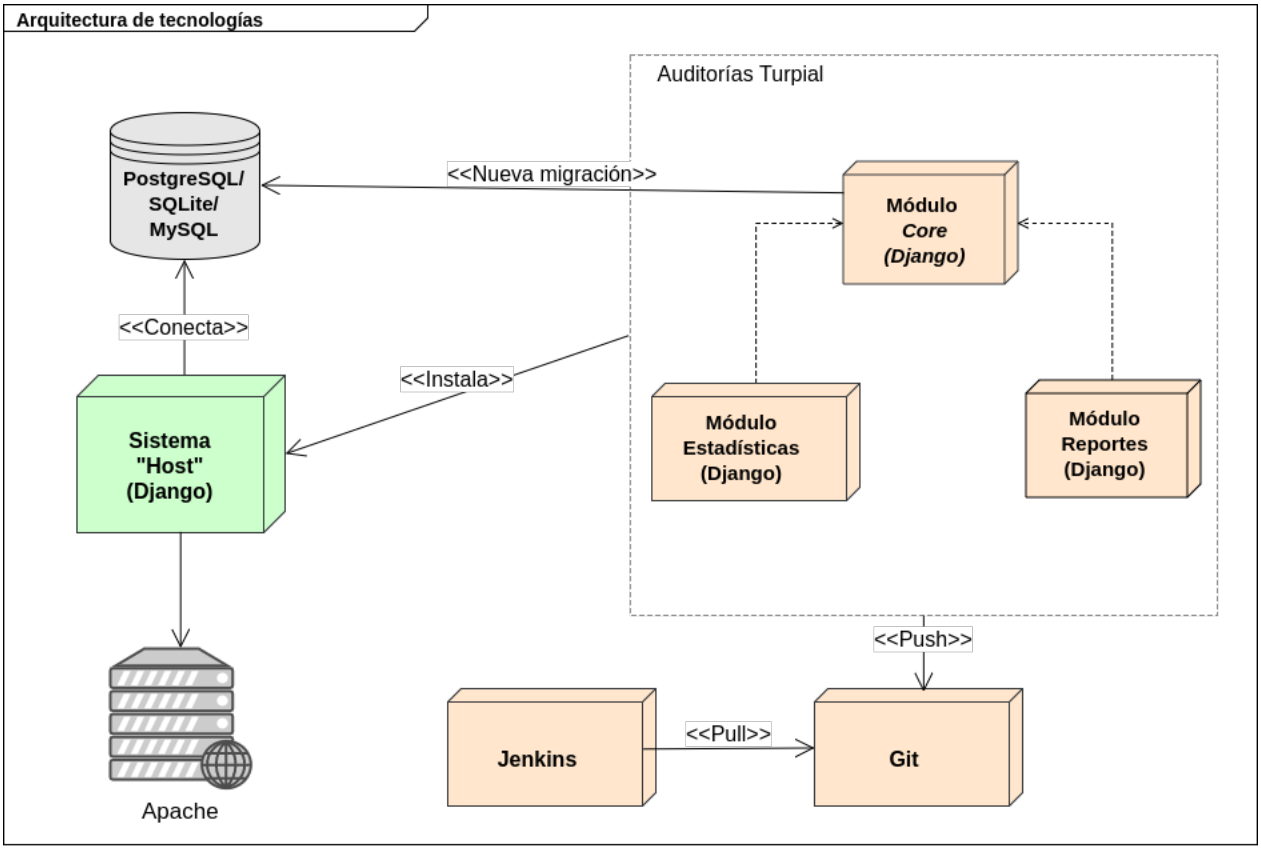

Figura 6.2: Arquitectura de tecnologías a utilizar

módulos podrían leerla para procesarla y mostrarla como sea pertinente.

No obstante, la arquitectura mostrada en la figura 6.1 sufrió modificaciones durante el desarrollo el proyecto para simplificarla. En lugar de crear tres aplicaciones, una para cada módulo, se separó Auditorías Turpial en dos: *backend* y *frontend*. En la sección que explica la fase de construcción se ofrecerán mayores detalles de estos cambios y sus razones.

En la figura 6.2 se muestran las tecnologías a utilizar en el proyecto. Como se mencionó anteriormente, se utilizará Django para el desarrollo. Como herramienta de control de versiones se utilizará Git y para automatizar la integración continua se usará Jenkins. El sistema "Host" será un proyecto de la empresa que utilizará como servidor Web, Apache y una base de datos relacional.

6.1.4 Diseño del módulo *Core*

Uno de los problemas más significativos que la empresa encuentra en otras extensiones de Django disponibles, es el hecho de que el registro de auditoría se guarda a nivel de la vista y es responsabilidad del programador colocar el código para esta funcionalidad. Esto

da cabida a que se olvide colocar en alguna de las vistas correspondientes a algún modelo, por lo que podría no guardarse todos los tipos de operaciones.

La librería Auditorías Turpial pretende evitar este problema guardando el registro de auditoría cada vez que ocurre alguna operación sobre el modelo a través del uso de las *signals* que ofrece el *framework* . Éstas, necesariamente deben distinguir entre un modelo auditable y uno que no lo es. Para esto, se contempló el uso de un *mixin* que pueda ser heredado por cualquier modelo y así proveer todas las funcionalidades mencionadas.

Por otro lado, se desea incluir en las auditorías el usuario que realizó la acción. La solución mencionada anteriormente no es suficiente para lograr esto, puesto que a nivel de modelos no se posee información sobre el *request* y no se puede saber qué usuario está en sesión. Por esta razón, se planteó agregar otro *mixin* a nivel de vistas que completara la información antes de guardarla en la base de datos. Sería responsabilidad del programador heredarlo en todas las vistas cuyos modelos mantienen un historial de transacciones.

Django posee mecanismos para traducir los modelos a otro formato con una estructura bien definida, más conocidos como *serializers*. En particular, posee maneras de convertirlos a JSON, por lo que se decidió utilizar dicha funcionalidad para cumplir con el requisito de mostrar los cambios en los valores de los campos del modelo auditable en este formato.

En cuanto a los listados, se acordó utilizar Datatables para mostrarlos como una tabla que se pueda ordenar y filtrar fácilmente. Este *plugin* puede manejar aproximadamente 10000 filas en sus tablas procesándolas del lado del cliente. Sin embargo, las auditorías serán potencialmente millones de registros, es por esta razón se debe realizar el procesamiento del lado del servidor.

Para la autenticación se consideró utilizar una tabla de usuarios propia con su respectiva permisología, con la intención de que no interfiriera con la del sistema "Host", sin embargo, esto no fue posible. Se profundizará esta explicación de esta decisión en la fase de desarrollo del presente capítulo.

6.1.5 Diseño del módulo de estadísticas

Uno de los requisitos mínimos con el que debe contar el módulo de Estadísticas es exponer un conjunto de gráficas que permitan visualizar e interpretar fácilmente los resultados de los cálculos de las auditorías. Para esto, se investigaron dos librerías de JavaScript: Amcharts y Charts.js. En la siguiente tabla se muestran las características principales:

Tabla 6.1: Modelo de datos de la tabla "AuditableAction"

Nombre del campo	**Tipo de dato**	**Descripción**
ID	INT (Entero)	Identificador del registro auditable.
author	VARCHAR	Referencia al usuario que realizó la acción.
action	ENUM. Estas acciones son: CREATED, UPDATED, DELETED.	El tipo de acción efectuada sobre un modelo auditable.
model_json_old	VARCHAR	Antiguo valor. Un JSON que representa el valor que tenía el registro que se ve afectado por la operación. En caso de que la acción sea "creado", el JSON representará el objeto vacío.
model_json_new	VARCHAR	Nuevo valor. Un JSON que representa el valor que tiene el registro que se ve afectado por la operación. En caso de que la acción sea "eliminado", el JSON representará el objeto vacío

Como los desarrolladores de Turpial Development son los principales usuarios de este proyecto, el pasante investigó si existía alguna librería que utilicen usualmente. En la empresa han utilizado varias, incluyendo las dos presentadas anteriormente, por lo que no tienen una preferencia en este sentido.

Debido a que la cantidad de registros de auditorías pueden crecer rápidamente, es necesario escoger una librería de gráficos que maneje adecuadamente grandes volúmenes de datos, por lo que se decidió utilizar Amcharts. Otra característica que inclina la balanza a favor de esta solución, es el hecho de que posee un *plugin* que permite exportar datos a un archivo CSV o PDF, una funcionalidad que se compenetra bien con el módulo de Reportes.

Aunque Charts.js posee una extensión de Django para facilitar la creación de los gráficos a través de las vistas, este proyecto requiere una lógica muy específica para entregar el conjunto de datos que se utilizará para mostrarlos. Por esta razón, es mejor tener total control sobre su implementación.

Los cálculos estadísticos serán entregados a la plantilla a través de una vista que incluirá el formulario de los filtros y la manipulación de los datos para crear los gráficos pertinentes y según los requiera la librería escogida.

6.1.6 Propuesta para la integración continua

Dado que en la empresa no existe precedente sobre el uso de una herramienta automatizada para integrar el código de manera continua, se le otorgó al pasante la libertad de escoger entre dos herramientas sugeridas por el líder del proyecto: Jenkins o Fabric. Para decidir cuál herramienta se adecuaba más al proyecto y a las necesidades de la empresa, fue indispensable que el pasante investigara las opciones en profundidad, sus fortalezas, limitaciones y recomendaciones de la comunidad. A continuación se muestra una comparación entre ellas:

Tabla 6.2: Comparación de Jenkins vs Fabric.

	Jenkins	**Fabric**
Descripción	Un servidor de automatización que puede ser utilizado para automatizar cualquier clase de tarea cómo construir, probar y desplegar software	Es una librería y una herramienta de línea de comandos para coordinar el uso de SSH para despliegues de aplicaciones o tareas de sistemas de administración.
Código abierto	Sí.	Sí.
Lenguaje	Escrito en Java	Escrito en Python
Extensible	Si. Cuenta con una gran cantidad de *plugins* para ampliar sus funcionalidades básicas y una tienda en donde pueden adquirirse	No.
Interfaz gráfica	Sí.	No.
Integración con Gitlab	Sí.	No
Ejecutar un script	Sí.	Sí.
Empaquetamiento y despliegue programado	Si. Se puede configurar un horario para ejecutar algún script que contenga todas las instrucciones para empaquetar, probar y desplegar el sistema.	No. Se debe ejecutar el archivo de configuración a través de la consola.
Observaciones	Altamente recomendado por la comunidad.Puede generar reportes de las pruebas ejecutadas.	Fácil de aprender, de configurar y de instalar.

Considerando las características que posee cada herramienta, se decidió utilizar Jenkins puesto que se pueden ejecutar *scripts* de manera programada, generar reportes, observar el estado del despliegue de varios proyectos e incluso integrarlo con Gitlab, un servicio Web de

control de versiones y desarrollo de *software* colaborativo basado en Git que es utilizado por la empresa.

Si bien Fabric es más simple, no es realmente un servidor para integración continua, se asemeja más a *scripts* que permiten automatizar tareas y se requiere de una herramienta suficientemente general que pueda ser usado tanto en la pasantía como en otros proyectos de la empresa.

6.1.7 Plan de pruebas

Para verificar que cada funcionalidad posea el comportamiento esperado, se realizaron pruebas de manera automatizada utilizando Pytest. Más específicamente, se llevaron a cabo pruebas unitarias, de regresión y de integración.

Adicionalmente, se convino que uno de los criterios de aceptación de las HU era validar el producto con el cliente para asegurar el cumplimiento de las reglas de negocio y que el producto desarrollado cumple los estándares. A esto se le conoce como pruebas de aceptación y fueron realizadas en cada cierre de *Sprint*.

Como el proyecto en cuestión es una librería, no puede funcionar por sí sola, sino que necesita instalarse en otro. Para esto, se creó una aplicación sencilla en Django y se instaló la librería en él. En cada *Sprint*, se constató que el programador pueda usar cada funcionalidad desarrollada sin ningún inconveniente. Este mismo proyecto, se utilizó para mostrar los avances al cliente.

En la fase de transición, se planificó que se realizaran pruebas en una aplicación desarrollada para uso interno de la empresa, llamado Turpial Team. No obstante, debido a que aún se encuentra en desarrollo no está disponible actualmente. Se optó por utilizar otro proyecto de uso interno de Turpial Development, llamado Turpial Calendar. Esta aplicación sirve para planificar eventos de la empresa y enviar notificaciones. Esta decisión no afecta de ninguna manera la pasantía ya que la librería debe poder ser instalada en cualquier proyecto con las características mencionadas en la sección de requerimientos.

En el apéndice E se encuentra el documento de Plan de Pruebas creado para la empresa, para ofrecer más detalle de los explicado en esta sección.

6.1.8 Planificación del desarrollo del proyecto

Luego de finalizar el proceso de investigación, aclarar los requerimientos y determinar las HU, se procedió a planificar la fase de construcción del proyecto. Para ello, se tomó en cuenta la prioridad, precedencia y puntaje de cada HU para determinar el orden. Se decidió iniciar con las HU relacionadas con el *Core* y la instalación de la librería.

Se planificaron ocho *Sprints* con una duración de dos semanas laborales (diez días) cada uno. Estos *Sprints* abarcan la fase de construcción del módulo *Core* y Estadísticas de Auditorías Turpial, así como la fase de transición.

6.2 Fase de construcción

Una vez culminada la fase de concepción, se procedió con la instalación del ambiente de desarrollo e implementación de los módulos que abarca la pasantía. También, se incluyen las pruebas pertinentes para cada funcionalidad desarrollada según indica el Plan de Pruebas.

6.2.1 Construcción del módulo *Core*

En esta sección se describe el proceso para desarrollar cada HU relacionada con el módulo *Core*, los problemas encontrados y sus soluciones. También se explica el nuevo diseño de la arquitectura y las pruebas efectuadas.

6.2.1.1 Preparación del ambiente de desarrollo

Antes de iniciar con la implementación de la librería, se instalaron y configuraron todas las herramientas necesarias para esto. En primer lugar, se instaló Python 2.7 y luego PIP. Se instaló el *plugin* de Python, Virtualenv, el cual permitió configurar el ambiente virtual. Este, se utilizó para instalar los requerimientos de la librería, en particular, Django 1.10 y Pytest 3.2. Estos, se registraron en el archivo "requirements.txt" para facilitar futuras instalaciones y determinar las dependencias de la librería. Luego, se procedió a crear la aplicación de Django y la estructura del módulo *Core*.

6.2.1.2 Estructura de la tabla de auditorías

Para registrar apropiadamente la información de la auditoría como fue establecida en la fase anterior, se creó un modelo en Django cuyo nombre es "AuditableAction". Este se agregará como una tabla adicional en la base de datos del sistema "Host" y posee los siguientes campos:

Tabla 6.3: Modelo de datos de la tabla "AuditableAction"

Nombre del campo	**Tipo de dato**	**Descripción**
ID	INT (Entero)	Identificador del registro auditable.
author	VARCHAR	Referencia al usuario que realizó la acción.
action	ENUM. Estas acciones son: CREATED, UPDATED, DELETED.	El tipo de acción efectuada sobre un modelo auditable.
model_json_old	VARCHAR	Antiguo valor. Un JSON que representa el valor que tenía el registro que se ve afectado por la operación. En caso de que la acción sea "creado", el JSON representará el objeto vacío.
model_json_new	VARCHAR	Nuevo valor. Un JSON que representa el valor que tiene el registro que se ve afectado por la operación. En caso de que la acción sea "eliminado", el JSON representará el objeto vacío

En cuanto a los campos tipo JSON, se utilizó la *plugin* de Django, Jsonfield, que provee todos los validadores necesarios para que el campo de sea considerado un JSON, y así, mantener la integridad de la base de datos. Adicionalmente, este *plugin* puede hacer uso del campo JSON que posee nativamente PostgreSQL; en el resto de los manejadores, se representa como un campo de texto.

6.2.1.3 Selección de un modelo auditable

Para que un programador pueda escoger qué modelo es auditable y cuál no, se creó un *mixin* a nivel de modelos, llamado "AuditableMixin". Este permite heredar el comportamiento necesario para registrar las auditorías en la base de datos.

6.2.1.4 Registro de una acción auditable

Como bien se ha dicho antes, Django posee *signals* que son detonadas por diferentes eventos; dos de ellos están relacionados con la creación y actualización de objetos, "pre_save" y "post_save". Estas señales se activan antes de guardar un objeto en la base de datos y después, respectivamente; bien sea para crearlo o actualizarlo. Cada una de ellas posee información sobre la instancia que será guardada en la base de datos y el modelo que envía la señal (*sender*). El *sender* es de gran utilidad puesto que permite determinar cuál es su clase, y así auditar solo los modelos que sean una subclase del "AuditableMixin" definido

anteriormente.

En principio, se optó por hacer un manejador para la señal “pre_save” debido a que, antes de guardar en base de datos, se puede obtener tanto el valor actual de la instancia como el que tendrá. Desafortunadamente, esta solución no asegura que se almacene la instancia a auditar en la base de datos, pero sí la traza de auditoría. Podría ocurrir algún error justo en medio de estas dos operaciones lo que generaría problemas de consistencia.

Utilizar la señal “post_save” tampoco fue una opción, ya que no se podía obtener el valor del registro anterior, lo cual es de suma importancia para comparar los cambios ocurridos.

Motivado por esto, se decidió sobreescribir el método “save()”, con lo que se puede realizar el procesamiento para capturar el valor anterior, guardar el objeto auditable y generar la auditoría. Se distingue el tipo de acción si la instancia posee o no una clave primaria; si no la posee se está creando.

El primer inconveniente encontrado con esta solución es el hecho de que, al cargar la base de datos con un *Fixture*, el *framework* no utiliza “save()”. Asume que los datos dentro del archivo son íntegros y los inserta, activando las señales correspondiente. El segundo, es que este comportamiento se replica con las inserciones y actualizaciones masivas, “bulk_create” “bulk_update”. Los mencionados métodos se traducen directamente a sentencias de SQL, por lo que tampoco activan *signals*. En la documentación de la librería se explican estos problemas y esperan ser corregidos en una próxima versión.

Con respecto a la acción DELETE, no hubo mayores contratiempos, se utilizó la señal “post_save”, la cual se activa luego de que se elimina una instancia, dado que no se requiere registrar el valor posterior.

Relación muchos a muchos Django cuenta con la facilidad de, a nivel de modelos, crear relaciones muchos-a -muchos sin necesidad de que el programador explícitamente genere la tabla intermedia que los conecta; a través del campo ”ManyToManyField”. Los cambios ocurridos en ellos, activan una señal llamada “m2m_changed”, la cual se utilizó para llevar control de los cambios de apuntadores (llaves primaria) hechos. No obstante, fue descartado debido a que el comportamiento natural del *framework* cuando se crea por primera vez el registro, es guardar la instancia sin los cambios en estos campos y luego guardarlos nuevamente.

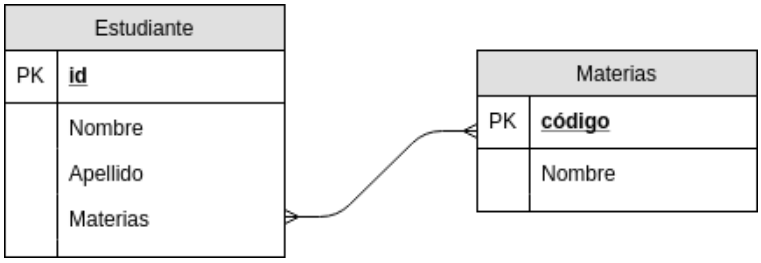

Figura 6.3: Modelo Entidad-Relación del ejemplo.

Por ejemplo, si se tiene un tabla Estudiante, que tiene una relación muchos-a-muchos con Materias como muestra la figura 6.3, al intentar guardar una instancia de Estudiante con algunas Materias, se sigue el siguiente flujo:

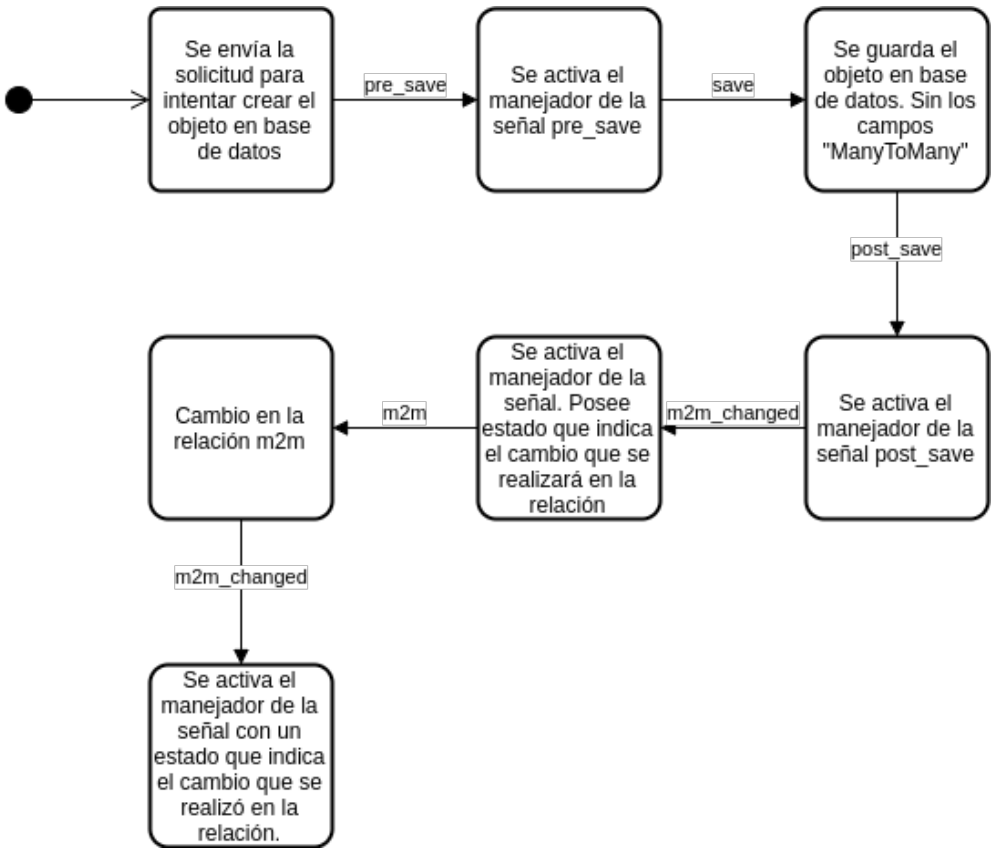

Figura 6.4: Gráfico de flujo del orden en el que se activan las señales para guardar un cambio en los campos “ManyToManyField”.

Debido al orden en que se activan la señales (figura 6.4), se almacena el objeto en base de datos y luego, cuando cambia el campo ”materias”, se guarda nuevamente, por lo que era imposible obtener en un sólo registro de auditoría todos los cambios.

El pasante propuso al cliente dos opciones: la primera es permitir que se generen dos

registros de auditoría, el primero con acción CREATE y el segundo con UPDATE, puesto que en realidad es una actualización. La segunda, es excluir esta relación de la auditoría, es decir, no utilizar "m2m_changed".

Esta última, fue la escogida por el cliente y se documentó apropiadamente. Si el programador desea llevar auditorías de los cambios ocurridos en casos como estos, puede escribir su propia tabla intermedia para relacionar ambos modelos y utilizar la opción "through" provista por "ManyToManyField" como indica la documentación de Django. Si, esta tabla, hereda el "AuditableMixin" se replica el comportamiento buscado inicialmente al incluir los campos muchos-a-muchos

Serializers Como se diseñó en la fase de concepción, se utilizaron los *serializers* provistos por el *framework* para almacenar la estructura completa de la instancia en cuestión, con el formato JSON. Los *serializers* incluyen la clave primaria del objeto, el nombre del modelo y los campos que este contiene.

Automáticamente, Django incluye la representación de los campos "ManyToManyField" y al decidir no rastrear los cambios en estos, fue necesario excluirlos explícitamente para evitar inconsistencias y confusiones.

Inclusión del usuario en la traza de auditoría Lo explicado anteriormente funciona perfectamente para registrar la auditoría y verificar los cambios, sin embargo no es suficiente para obtener el usuario, puesto que a nivel de modelos no se cuenta con esta información. Para solucionar esto, se creó otro *mixin* llamado "AuditableMixinView", que inyecta la información del usuario en un campo oculto incluido en el "AuditableMixin". De esta manera, en las *signals* se puede poseer la información del usuario en sesión y todo lo relacionado con él.

6.2.1.5 Instalación de la librería utilizando PIP

Una vez se obtenida una versión suficientemente sólida de la librería, se prosiguió a satisfacer uno de los requerimientos más importantes: que la librería sea capaz de instalarse utilizando PIP. El pasante debió documentarse al respecto, debido a que existía gran desconocimiento en todo el equipo. En la página web oficial de Django se encuentra un tutorial bastante detallado para construir extensiones el cual sirvió de guía para el proceso.

Lo primero que se hizo fue construir un archivo llamado "setup.py", que contiene la información de la librería, el nombre, la versión, los autores, la licencia y los paquetes que requiere, entre otros. En el caso de Auditorías Turpial, el paquete requerido fue Jsonfield.

Luego, se creó el archivo "MANIFEST.in" para incluir aquellos archivos que no son detectados automáticamente por el "setup.py" como las plantillas de Django, los estáticos (javascript, css), la licencia, entre otros.

Por último, en la cónsola, posicionados en la carpeta en la que se encuentra el achivo "setup.py", se procede a empaquetar la librería. Una vez culminado este proceso, se tiene un archivo comprimido que se puede utilizar para instalar la librería con PIP.

El pasante documentó con gran detalle este proceso para que el resto de los miembros del equipo pudiesen seguir los pasos. Esta documentación está disponible para la empresa, en caso de que deseen crear una nueva librería o actualizar Auditorías Turpial.

6.2.1.6 Listados de auditoría

Culminada toda la construcción de los *mixins*, *signals* e instaladores de la librería, se procedió a implementar la funcionalidad de listados de todas las auditorías registradas en base de datos. Para esta funcionalidad se decidió utilizar el *plugin* de JavaScript, Datatables, que permite integrar de forma fácil y sencilla una tabla a la plantilla. Los listados son procesados en el servidor y tienen paginación, puesto que las auditorías pueden poseer millones de registros. Con esta decisión se evitan dos aspectos relevantes, sobrecargar el navegador de altos volúmenes de contenidos e incrementar el tiempo de espera del usuario para el cargado de los listados.

Al usuario presionar alguno de los botones en la plantilla del listado, se realiza una solicitud GET de dicha página utilizando AJAX. De esta manera, se refresca ese componente sin necesidad de recargar completamente la página, lo que mejora la experiencia de usuario.

6.2.1.7 Reestructuración de la arquitectura

Antes de iniciar con la elaboración de los otros módulos de la librería, se notó que la funcionalidad de generar reportes está íntimamente relacionada con los listados de auditoría. Los filtros y el ordenamiento aplicados a los listados, deben aparecer en los CSV y PDF generados. Es por esta razón que, al finalizar el desarrollo de esta HU, el equipo se replanteó la estructura establecida en la fase de concepción.

En lugar de crear tres aplicaciones en Django, una para cada módulo, se decidió crear sólo dos, como muestra la figura 6.5. La primera de ellas sería el backend de Auditorías Turpial,

que mantiene toda la lógica para registrar las auditorías, explicada en los puntos anteriores; mientras que la segunda tendría los módulos de Reportes y Estadísticas. A esta última la llamaremos Auditorías Turpial UI (User Interface) y contendrá todo el frontend del proyecto. Las funcionalidades de los listados fueron colocadas en el módulo de Reportes para facilitar la generación de PDF y CSV.

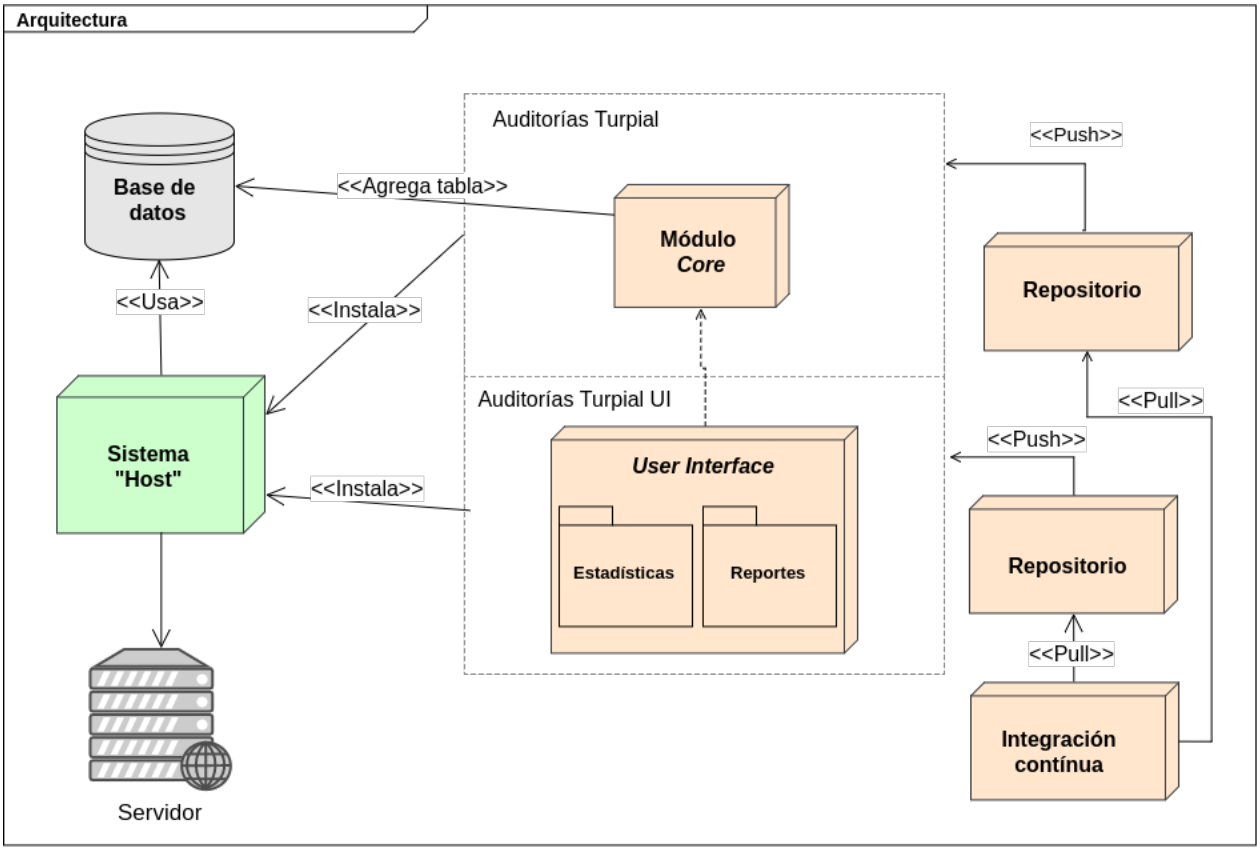

Figura 6.5: Arquitectura final de la librería.

Esta reestructuración permite separar la captura de información de la manera en la que se muestra, sin que se vean afectados los beneficios que ofrecen los microservicios. Aunque los módulos de Estadísticas y Reportes se encuentren dentro de la misma aplicación, el programador puede elegir no instalar alguno de ellos en su sistema si lo excluye de las "INSTALLED_APPS" (aplicaciones instaladas) de Django.

En este punto del desarrollo, se dispone de un módulo Core que cumple todas las funcionalidades planteadas según la nueva arquitectura. En la figura 6.10 se muestra la estructura de este módulo dentro del patrón MVT.

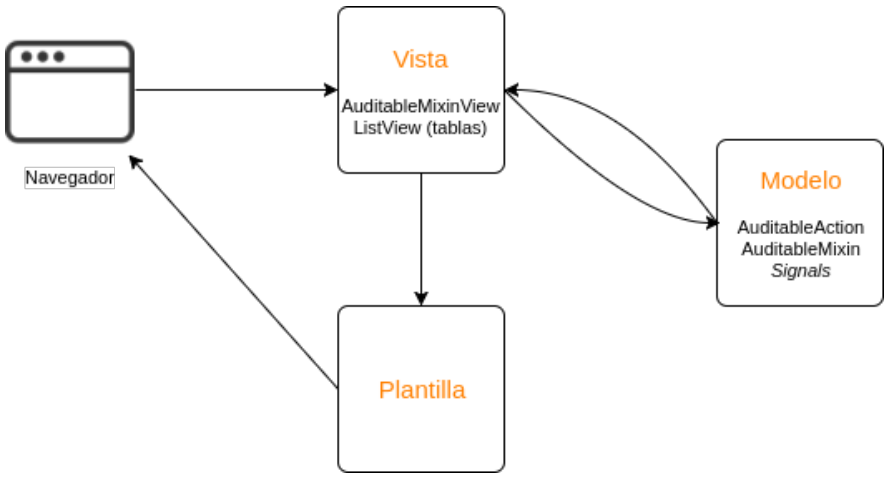

Figura 6.6: Estructura del módulo *Core* en el patrón MVT (creación propia).

6.2.1.8 Integración continua

Luego de haber terminado gran parte del módulo *Core*, se prosiguió a instalar Jenkins. Para esto, se ingresó al terminal de la computadora, se agregó el repositorio de la herramienta y se instaló.

Una vez realizado esto, el pasante ingresó, mediante la interfaz gráfica, al sistema de integración continua para configurar el repositorio del proyecto, ejecutando los siguientes pasos:

1. Instalar el *plugin* de Git en Jenkins.
2. Agregar las credenciales de Git del pasante (usuario y contraseña) en el sitio administrador. De esta manera quedan almacenadas dentro de Jenkins para futuras acciones que las requieran.
3. Incluir una “nueva tarea” para crear un nuevo proyecto.
4. Clonar el repositorio, ingresando a la sección de “Configuración” del proyecto creado y se proporcionó el enlace HTTPS. La herramienta permite seleccionar una rama específica de la cual hacer *pull*; se escogió la rama ”develop”.
5. En la configuración del proyecto, agregar la ejecución de un *script* de shell luego de hacer *pull* del repositorio para desplegar el proyecto.

El pasante creó un *script* que permite instalar el ambiente virtual y los *plugins* necesarios para que el proyecto pueda ejecutarse, migrar la base de datos y ejecutar las pruebas. Esto es lo mínimo requerido por la HU. Adicionalmente, construyó otro *script* que automatiza el

empaquetamiento de la librería, el cual podría incluirse en el despliegue de la librería.

Por último, se probó la configuración de Jenkins mediante la interfaz. El pasante utilizó el botón de "build" y comprobó en la consola de la herramienta que se había descargado la información más reciente de la rama "develop" y que se había ejecutado el *script* exitosamente.

Con el uso de Jenkins, se puede llevar el control de los despliegues de la librería y, en caso de que alguno falle, se puede detectar rápidamente verificando el estado del "build". De esta manera, gran parte del tiempo, la librería está disponible para la empresa y pueden verificar que sea estable sin necesidad de descargarla y probarla.

El comportamiento logrado con el *script* mencionado anteriormente, no evita que en el repositorio exista una versión inestable, puesto que el "build" se efectúa luego de que los cambios han sido incluidos en el repositorio. Es por esta razón que el pasante decidió investigar sobre posibilidad de evitar que se incluya código que no pase las pruebas.

Dado que se utilizó Gitlab, se investigó sobre la configuración de Jenkins con este y se encontró que existe un *plugin* que permite la integración de ambos sistemas. El pasante procedió a instalarlo, realizando las siguientes configuraciones sobre ambos sistemas:

- En Gitlab, fue necesario generar un identificador único llamado "Gitlab API Token" creado para cada usuario. Adicionalmente, se debe crear un "Webhook" con el URL en el que está instalado Jenkins que incluye el proyecto en cuestión.
- En Jenkins, se utiliza el "Gitlab API Token" como credencial para conectarse con Gitlab. Luego, se procede a configurar el proyecto. En la sección de "Configuración", el *plugin*, agrega una nueva opción para accionar el empaquetamiento cuando ocurra un evento en Gitlab. En particular, el evento de *push*.

Para crear el "Webhook", era indispensable instalar Jenkins en un servidor de la empresa, para que el URL mencionado pueda accederse desde Gitlab. No obstante, el líder técnico consideró que no era necesario el comportamiento adicional descrito, así que se descartó.

El pasante dejó una documentación detallada acerca del proceso llevado a cabo para referencias futuras de algún otro miembro de la empresa, así como de la integración con Gitlab. Con esto, se pueden replicar fácilmente los pasos e instalar Jenkins en un servidor de la compañía e incluir otros proyectos, además de esta pasantía.

6.2.1.9 Pruebas del módulo *Core*

Para el módulo Core, se generaron veintidós (22) casos de prueba automatizadas utilizando Pytest (ver Apéndice F). Dentro de estas pruebas, Once (11) consistían en validar el funcionamiento del "AuditableMixin", verificando que se guardara apropiadamente todos los campos la traza de auditoría.

Dentro del módulo de pruebas de la librería se agregaron cuatro (4) modelos con las relaciones mostradas en la figura 6.6, que sólo existen cuando se crean las pruebas. Al inicializar la ejecución de los casos, se efectúa una migración de la base de datos, que permite a estos modelos ser incluidos en el esquema. Estos, poseen relaciones suficientemente complejas para poder realizar casos de pruebas con uno-a-uno, uno-a-muchos y muchos-a-muchos.

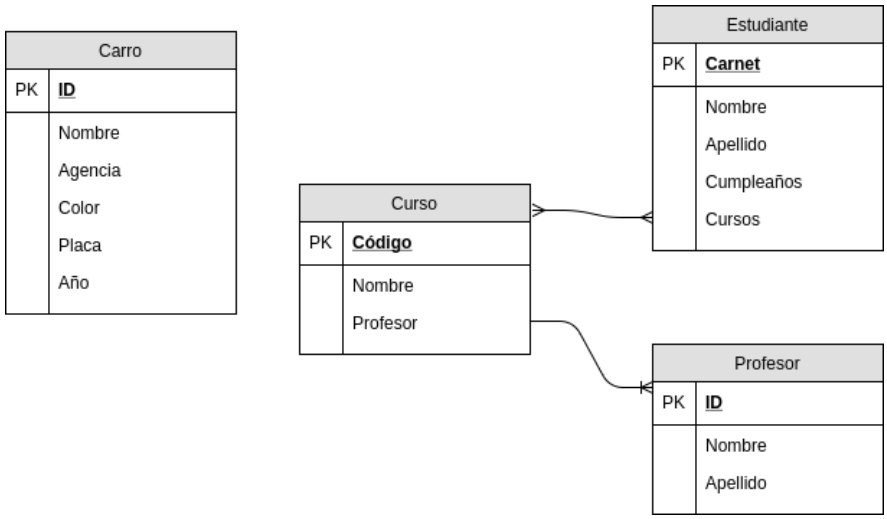

Figura 6.7: Modelo Entidad-Relación de la base de datos de prueba (creación propia).

Antes de que comiencen a ejecutarse las pruebas, se precargan los registros con un *Fixture* que utiliza el formato JSON. Estos están escritos siguiendo la estructura de los *serializers*; contienen la clave primaria, el modelo al que pertenece y el valor de cada campo como muestra la figura 6.9. Luego, se configuró Pytest para que utilizara el comando de Django "loaddata" y así, cargar el *Fixture*. La base de datos es desechada al finalizar la ejecución.

De estos casos de prueba, se encontró que fallaron dos, debido a que el *serializer* no excluía los campos "ManyToManyField". Se corrigió el código según lo explicado en la sección sobre *serializers*, y se obtuvo que todas las pruebas fueron exitosas.

Once de los casos de pruebas están dedicados a probar el "AuditableMixinView" para verificar que el usuario se agregue correctamente. Se configuró "RequestFactory" para simular un *request* que solicitara un URL particular. De esta manera, se pueden probar las vista como se haría con cualquier otra función; con una entrada fija y una salida esperada,

Figura 6.8: Ejemplo del formato JSON para cargar un *Fixture* (creación propia).

sin inspeccionar el contexto de la petición.

Todas estas pruebas resultaron exitosas exceptuando el caso C04-P06 (ver Apéndice F) puesto que al eliminar un modelo no-auditable con una referencia a otro que sí lo es, no se realiza la llamada a la función de la vista del modelo auditable. Por esta razón, no se almacena el usuario en estos registros. Esto fue documentado y se aceptó este pequeño error puesto que se conserva la integridad del sistema de auditorías.

Una vez terminada las pruebas unitarias y de integración, el módulo *Core* cumple todos los requisitos del cliente, incluyendo el servidor de Integración Continua. De esta manera, se alcanzan los objetivos planteados.

6.2.1.10 Permisología

Con relación a la permisología de las auditorías, la HU C11 (ver Apéndice B) especifica que se desea contar con un sistema de autenticación (login) y que cada usuario autenticado posea un conjunto de permisos asociados. Estos, permitirán restringir ciertas acciones dentro de las auditorías, en particular visualizar listados.

Para crear el sistema de autenticación, se quería crear un modelo de usuarios para la librería. No obstante, al pasante investigar sobre esto, encontró que la documentación de Django indica que los *plugins* no deben crear su propio modelo de usuarios. Debido a que podría generar conflictos con la aplicación que lo instale. Por consiguiente, se descartó la HU relacionada con la autenticación (C11.1).

Tomando en cuenta esto, se sugirió realizar un módulo de Permisología que incluyera,

tanto las restricciones de los listados, como los de generación de reportes (PDF y CSV).

Dado que la implementación de este módulo no se encontraba en el objetivo de la pasantía, se decidió no incluirlo para esta versión de la librería. Sin embargo, el pasante decidió realizar el diseño del mismo con la intención de que sirva de referencia.

Este módulo también sería un microservicio, como Estadísticas y Reportes, que incluya permisos básicos y le brinde facilidades para generar nuevos permisos. Para restringir una funcionalidad específica, se utilizaría el mismo patrón de desarrollo del resto de la librería; ofrecer un *mixin* que encapsule el comportamiento. Para obtener mayor detalle sobre el módulo de Permisología.

6.2.1.11 Análisis de desempeño

6.2.1.12 Prototipo de tabla de auditorías descentralizada

6.2.2 Construcción del módulo Estadísticas

En esta sección, se explica el desarrollo de módulo de Estadísticas; cómo se generaron los filtros, cálculos y gráficos necesarios, para presentar al usuario final la información recopilada por el módulo *Core*.

6.2.2.1 Creación del formulario de filtros

Para cumplir con las HU que se refieren al filtrado de los registros de auditoría, se procedió a crear un formulario. Inicialmente, se incluyeron aquellos campos relacionados con el rango de tiempo. De esta manera se puede construir el flujo de datos, básico, sin preocuparse por el resto de los filtros. Es importante destacar que ninguno de los campos es requerido y si no se selecciona, se obtiene toda la información.

Para crear los campos fecha de inicio, hora de inicio, fecha de fin y hora de fin; se aprovecharon los tipos “DateField” y “TimeField” de Django. Estos manejan la validación del formato de los valores ingresados y, en caso de ser correctos, procede a convertirlos en un objeto ”datetime” de Python para facilitar su manipulación. Aunque Django posee un campo ”DateTimeField”, en el que se puede obtener tanto la fecha como la hora, se decidió mantener estos campos separados, para tener completo control sobre los validadores y rellenar la información no requerida.

El método “clean()” del formulario fue sobreescrito para manejar la lógica de los

validadores y evitar el envío de un formulario incorrecto que pueda generar una excepción. Este método se encarga de:

- Validar que la fecha de inicio no sea mayor que la de fin, tomando en cuenta la hora.
- Validar que la fecha de fin no sea mayor a la actual, tomando en cuenta la hora.
- Arrojar un error en el formulario cuando se coloca la hora (de fin o inicio) pero no la fecha relacionada. Este caso es muy importante puesto que no se puede filtrar únicamente con las horas porque generaría un error en el *query*.
- En caso de que no sea ingresado, rellenar el campo de la hora de inicio con el valor 00:00.
- En caso de que no sea ingresado, rellenar el campo de la hora de inicio con el valor 23:59.

Una vez estructurado el formulario con el rango de tiempo, se prosiguió a armar el *query* asociado. Para esto, se utilizó la vista prefabricada del *framework* "ListView", que maneja toda la lógica de listar objetos utilizando un *queryset* particular. La vista resultante se denominó "StatisticsView". Este *queryset* se configuró para que obtuviese todos los resultados de la tabla de auditorías, no obstante, el programador puede cambiar este atributo.

El formulario se incluye en una vista adicional llamada "FilterView" que hereda "StatisticsView". De esta manera, en el código se distingue fácilmente la lógica relacionada a los filtros y la relacionada con los gráficos. En "FilterView", se colocó el formulario en el contexto y si el mismo es válido, se procede a llamar a la función "build_filters". Esta función, añade nuevos filtros al *queryset* inicial, según los valores enviados en el formulario a través del método POST.

Es importante destacar que el *framework* maneja los *queries* de manera "lazy", lo que quiere decir que no serán evaluadas hasta que realmente sean necesarias. Este comportamiento permite filtrar el *queryset* inicial tantas veces como indique el formulario, sin necesidad de cargar todos los objetos en memoria ni aumentar el tiempo de respuesta del usuario.

También se agregaron tres botones que permiten agilizar el envío del formulario cuando se desea filtrar por la fecha, mes o año actual. Estos botones rellenan los campos de fecha según sea el caso, haciendo uso de funciones de JavaScript.

```
class FilterForm(forms.Form):
    """Filter form for statistics."""

    start_date = forms.DateField()
    start_time = forms.TimeField()
    end_date = forms.DateField()
    end_time = forms.TimeField()
    author = forms.ChoiceField()
    action = forms.ChoiceField()
    model = forms.ChoiceField()
```

Figura 6.9: Estructura del formulario de filtros (creación propia).

Una vez creada la estructura que permite filtrar por rango de tiempo, se procedió a agregar al formulario, los campos “action”, “model” y “author” para filtrar por tipo de acción, modelo y autor respectivamente. Con cada uno de ellos, se repitió el proceso descrito anteriormente.

6.2.2.2 Cálculos estadísticos

Luego de obtener el *queryset* filtrado, se procedió a construir los cálculos estadísticos indicados por los requerimientos: el promedio, máximo, mínimo y total de elementos. Para esto, se procesa el *queryset* creando un diccionario, llamado ”stats”, cuya clave es el día en que fue realizada la operación y su valor es la cantidad de transacciones en ese día.

```
{
    "all": {
        "count": 4,
        "max": 2,
        "min" : 1,
        "avg" : 0.010,
        "data": {
            "2017-10-16": 1,
            "2017-10-19": 1,
            "2017-10-25": 2
        }
    }
}
```

Figura 6.10: Ejemplo de “stats” (creación propia).

En el formulario, se creó la función ”days_elapsed” para obtener la cantidad de días transcurridos entre la fecha de inicio y la de fin. Este valor se utiliza para calcular el promedio apropiadamente, dividiendo el total de operaciones entre la cantidad de días

transcurridos. Si no se provee fecha de inicio, se toma la fecha del primer objeto en la tabla de auditorías para calcular la diferencia de fechas.

Para calcular el porcentaje de crecimiento, en "build_filters" se determina la cantidad de registros que ocurrieron antes de la fecha de inicio ingresada en el formulario. Este valor, denominado "offset", se utiliza para conocer cuántas auditorías se registraron en el periodo anterior. El porcentaje de crecimiento se calcula restando el valor de periodo anterior con el actual, y dividiéndolos entre la cantidad total de modelos.

En cuanto a la cantidad de modelos auditables, se utilizó la función "get_models" de Django que retorna una lista de todos los modelos de la aplicación. Cada modelo es representado por un objeto que incluye su clase, con el cual se puede contar cuántos heredan el "AuditableMixin". Siguiendo este mismo procedimiento se puede determinar el porcentaje de cobertura de la librería, dividiendo la cantidad de modelos auditables entre el total de modelos.

Todos los cálculos de estadísticas se colocan en el contexto de la vista "StatisticsView" para que puedan ser utilizados en las plantillas.

6.2.2.3 Construcción de gráficos

Antes de procesar los datos necesarios para crear un gráfico, se investigó en profundidad las funcionalidades que provee Amcharts. Este *plugin*, requiere crear un elemento en HTML con la etiqueta "¡div¿", que posea un identificador único (atributo id) con el que se pueda referenciar para construir el gráfico con JavaScript. Para esto, se utiliza la función "Amcharts.makeChart" pasando como parámetros el identificador y un objeto con las opciones para crearlo.

Los gráficos de Amcharts son altamente configurables, pues cuentan con una gran cantidad de opciones que permiten crear comportamientos complejos. Entre estas opciones se destacan:

- dataProvider: datos para mostrar el gráfico en formato JSON. Puede contener cualquier cantidad de campos y se puede utilizar cualquier.
- type: tipo de gráfico. Los posibles valores de esta opción son: “serial”, “pie”, “xy”, “radar”, “funnel”, “gauge”, “map”, “gantt” y “stock”.
- theme: tema o estilo del gráfico. Amcharts posee unos temas propios creados en JavaScript, algunos de estos son "chalk" "black" "light".

- balloon: crea los mensajes que aparece al posicionar el cursor sobre los ítems del gráfico. Amcharts crea automáticamente la información que aparece, sólo se necesita ajustar la apariencia.
- graphs: crea la visualización de los datos en los siguientes subtipos de gráfico: “line”, “column”, “step line”, “smoothed line”, “olhc” y “candlestick”. Cuenta con varias opciones de personalización, incluyendo la posibilidad de modificar la información mostrada en el “balloon”
- legend: leyenda del gráfico en cuestión.
- export: habilita un menú que permite exportar los datos en diferentes formatos.
- responsive: ajusta el tamaño del gráfico dependiendo de la pantalla.

Para construir los gráficos con Amcharts, se creó una clase llamada ”ChartData” que encapsula toda la lógica necesaria para obtener la información relacionada con él. La clase cuenta con atributos como el identificador, tipo, tema, y los datos. Este último, denominado ”data”, se inicializa con el resultado de procesar ”stats”, y convertirlo en un objeto JSON. De esta manera, puede ser utilizado como ”dataProvider”. La mencionada clase puede ser heredada por el programador y ampliada para ajustarlo a sus necesidades.

La clase ”ChartData” es usada para crear una variable en el contexto de la vista ”StatisticsView” que posea toda la información, y así, pueda ser utilizada en la plantilla. En la vista se incluyen seis gráficos:

- Gráfico de línea con la evolución de las auditoría.
- Gráficos de torta con la comparación entre cantidad de operaciones por autor, por modelo y por acción.
- Gráfico de columnas apiladas con la comparación de cada operación realizada por cada usuario.
- Gráfico de columnas apiladas con la comparación de cada operación realizada sobre cada modelo.

6.2.3 Plantillas del módulo Estadísticas

Dado que el sistema en cuestión es una librería, es sumamente importante que las plantillas generadas sean fáciles de utilizar y personalizar por el programador que las utilice. Por esta razón, se crearon varias plantillas que pueden agregarse en el sistema ”Host”

haciendo uso de la etiqueta "include". Esta etiqueta permite a Django, cargar y mostrar un plantilla utilizando el contexto e incluso se puede pasar parámetros para incluir nuevos valores de variable en el contexto.

Entre las plantillas creadas, la principal es "turpial_statistics.html" que incluye varios componentes:

- El formulario del filtro.
- Todos los cálculos estadísticos.
- Gráfico de línea con la evolución de las auditoría.
- Gráficos de torta con la comparación entre cantidad de operaciones por autor, por modelo y por acción.
- Gráfico de columnas apiladas con la comparación de cada operación realizada por cada usuario.
- Gráfico de columnas apiladas con la comparación de cada operación realizada sobre cada modelo.

Si el programador desea una plantilla con toda la información relacionada con las estadísticas puede incluir esta plantilla en las propias.

En cuanto a los gráficos, se observó que la creación de cada uno era similar; por lo que se decidió fabricar una plantilla reutilizable que recibe como parámetro el objeto de la clase "ChartData". Con esto, se le ofrece al programador una manera sencilla de crear gráficos de Amcharts. Además, es utilizado en "turpial_statistics.html" para generar los gráficos que ofrece.

Esta idea fue aplicada a los cálculos estadísticos con las plantillas "turpial_cards.html" y "turpia_card_stats.html". La primera presenta la información general relacionada con la librería, como la cantidad de modelos auditables, porcentaje de cobertura y de crecimiento. La segunda, muestra las estadísticas relacionada con la cantidad de registros auditables (promedio, máximo, mínimo, total). El programador puede utilizar cada plantilla para presentar la información como mejor le convenga en caso "turpial_statistics.html" no se ajusta a sus necesidades.

Todas las plantillas utilizan Bootstrap para mostrar la información, dado que la empresa posee como estándar de desarrollo, este *framework*.

Adicionalmente, se creó un tema propio para los gráficos ("turpial.js"), aprovechando las facilidades que ofrece Amcharts. En este tema, se configuraron los colores de los gráficos, el estilo de las leyendas, el tamaño de las letras entre otros.

El programador también puede crear su propio tema, utilizando el atributo del "StatisticsView" llamado "chart_theme". Este requiere un diccionario con el nombre del tema y su ubicación en el proyecto (dirección). La dirección fue incluida dado que el tema debe cargarse después que el JavaScript principal de Amcharts pero antes de generar el gráfico o no estará disponible. El valor de la variable "chart_theme" se utiliza en el atributo del tema en la clase "ChartData".

```
class StatisticsView(ListView):
    """Handles variables related to statistics."""

    queryset = AuditableAction.objects.all()
    charts_theme = {"name": "turpial", "path": "amcharts/themes/turpial.js"}
```

Figura 6.11: Definición de "StatisticsView" incluyendo el campo "charts_theme" (creación propia).

Para el resto de los estilos, el programador tiene completa libertad de sobreescribirlo al cambiar el estilo de Bootstrap como lo indican.

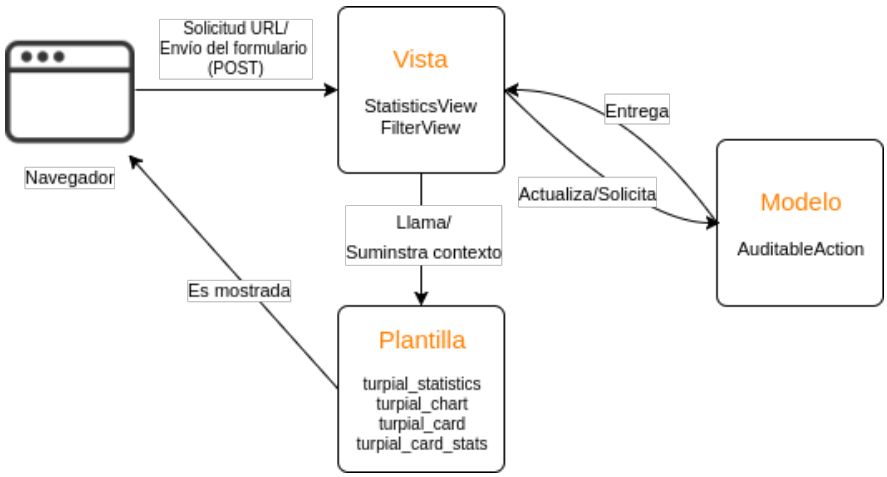

Figura 6.12: Estructura del módulo de Estadísticas (creación propia).

Luego de culminar las plantillas, se posee un módulo de Estadísticas que cumple con todos

los requerimientos establecidos. En la figura 6.11 se muestra la estructura final del módulo de Estadísticas y en dónde se ubica cada función del mismo.

6.2.3.1 Pruebas del módulo Estadísticas

Para el módulo de estadísticas se escribieron 106 casos de prueba, utilizando los mismos modelos que en el módulo *Core* y el mismo *Fixture*. De estos casos, 73 corresponden al formulario de filtros y 33 a los cálculos estadísticos en la vista "StatisticsView".

Las pruebas del formulario se enfocan en determinar si es válido, y si lo es, proceder a filtrar el queryset inicial según los valores provistos. Para comparar que los resultados obtenidos en el *queryset* son los esperados, se utilizó la función "filter" de Python sobre la lista de todos los elementos de la tabla. También, se incluyeron los casos de pruebas en el que el formulario es inválido, en ellos se verifica que ocurra una excepción que incluya un mensaje de error que notifique al usuario. Esto permite asegurar que los filtros funcionan correctamente. Al ejecutar estas pruebas, se obtuvo que todas fueron exitosas.

Para los cálculos estadísticos se deben probar las funciones que entregan estos resultados, haciendo una solicitud POST al URL que llama a la vista "FilterView". Para esto, se utilizó "Client()" una clase de Django que permite enviar la petición, simulando el navegador. De esta manera se puede inspeccionar el contexto e incluso el HTML de la respuesta, y así constatar que los valores son los esperados.

Al ejecutar estas pruebas fallaban todas, excepto las que obtenían los resultados sin filtrar. Esto sucedió debido a que no se enviaban correctamente los datos del formulario que están relacionados con el tiempo. Para corregir el error, todos los valores fueron convertidos en *string*.

Luego, fallaron las relacionadas con el filtro de "acción". No se estaba calculando el "offset" en este caso, por lo que el porcentaje de crecimiento era incorrecto. Se corrigió el código y las pruebas fueron exitosas.

Otro problema que surgió fue que, como el promedio cambia a medida que pasan los días, cuando no se envía la fecha de fin las pruebas comenzaron a fallar a medida que avanzaron los días. Para lidiar con esto, se creó una función, en lugar de colocar un valor fijo con el cual comparar el promedio como se estaba haciendo con el resto de las estadísticas. Esta función divide la cantidad de registros esperados entre días transcurridos para calcular el promedio. Los días transcurridos fueron obtenidos con la función "days_elapsed". Luego se

obtuvo que las pruebas fueron exitosas.

Finalmente, se crearon casos de pruebas en los que se genera un modelo al momento de ejecutar el caso en particular. Con la intención de validar que se calculan correctamente la cantidad de modelos auditables y el porcentaje de cobertura. Estas pruebas resultaron exitosas.

6.3 Fase de transición

En esta sección se explica el proceso de integración de la librería en el proyecto Turpial Calendar, los errores encontrados y sus correcciones. Así como, la puesta en producción de ese sistema. Este proceso duró una semana.

6.3.1 Integración de la librería con Turpial Calendar

Antes de iniciar el proceso de integración, se descargó el proyecto Turpial Calendar y se instalaron los paquetes de los cual depende. Este proyecto utiliza Django 1.11 y MySQL.

A continuación, se instaló la librería Auditorías Turpial (turpial_auditor); utilizando la facilidad que ofrece PIP para esto, a través del enlace de un repositorio. Este proceso se repitió para Auditorías Turpial UI (turpial_auditor_ui).

Con respecto a la configuración de las librerías, se siguió paso a paso la documentación correspondiente. En primer lugar, se incluyeron en "INSTALLED_APPS" los módulos necesarios para registrar las auditorías y para visualizar las estadísticas. Luego, agregaron modelos auditables utilizando el *mixin* para esto. Los modelos que presentan auditorías son: Eventos, Proyectos, Empleado y Categoría (de un calendario).

De estos modelos, se ubicaron las vistas que permiten crear y eliminarlos y se incluyó el *mixin* para agregar el usuario en la traza de auditorías, "AuditableMixinView".

Después, se incluyó un URL en Turpial Calendar que llamase al "FilterView" y que permitiera ver la plantilla de estadísticas. Se repitió este proceso para crear otra plantilla que utilice los estilos de Turpial Calendar y se reorganice la información de otra manera para probar esta funcionalidad.

Errores encontrados Se ejecutó el proyecto y se intentó crear un evento en el calendario para verificar que las auditorías se estaban realizando. Sin embargo, al tratar de enviar el

formulario ocurrió una excepción relacionada con el campo “ForeignKey”. Django incluye en los modelos, un campo oculto para realizar una referencia “hacia atrás” para obtener el modelo que guarda una relación con este. Al excluir los campos muchos-a-muchos en el serializer se utiliza una función que permite determinar cuales son y se accede al nombre de dicho atributo. Este campo oculto, no posee la función que accede al nombre y esto ocasionaba la falla.

Una vez detectado el error, se corrigió y se constató que el *Core* funcionaba como se esperaba. Luego, se escribieron tres nuevas pruebas para validar que este error fue solventado. Para esto, se agregaron dos nuevas tablas en la base de datos de prueba: Compañía y Persona, como se muestra en la figura 6.13. Estas pruebas fueron exitosas.

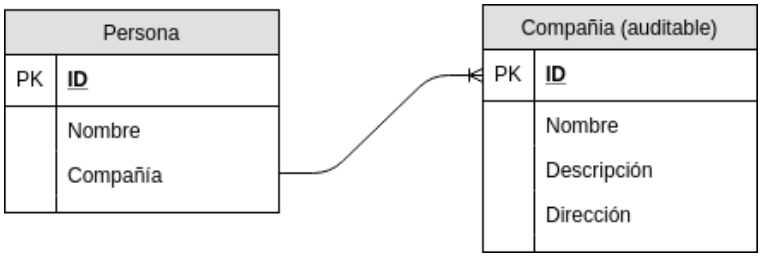

Figura 6.13: Diagrama Entidad-Relación de los nuevos modelos en las pruebas (creación propia).

En el módulo de Estadísticas se encontró un pequeño error en el cálculo de la cantidad de días transcurridos entre la fecha de inicio y la de fin. Cuando no se envía la fecha de inicio en el formulario, se escogía la fecha del primero objeto auditable. Sin embargo, si este es borrado de la base de datos, no se obtenía ninguna fecha para realizar el cálculo. Esto se solucionó utilizando el mismo *queryset* ya filtrado y obteniendo el primer registro que posee.

6.3.2 Puesta en producción

Finalizado el proceso de integración, se prosiguió a poner el proyecto Turpial Calendar en el servidor de Turpial Development, Webfaction. Para esto, se ingresó a este sistema y se creó un sitio web, agregando el dominio en el que estaría disponible el proyecto. Después, se configuró el tipo de aplicación (Django), se creó la base de datos con su respectivo usuario y contraseña y se migró.

Para clonar el repositorio de Turpial Calendar, se ingresó mediante el protocolo SSH al servidor y luego se creó el ambiente virtual del proyecto. Se instalaron las dependencias y las librerías turpial_auditor y turpial_auditor_ui desde su respectivos repositorios.

A continuación, se configuró el servidor web, Apache. Se agregó un nombre de servidor, un alias y una tarea para activar el servidor (*daemon*).

Finalmente, se comprobó que el sistema funcionara correctamente, incluyendo las librerías y todas las funciones relacionadas con ellas.

CONCLUSIONES Y RECOMENDACIONES

Conclusiones

Recomendaciones

REFERENCIAS

[1] amCharts. Amcharts. https://www.amcharts.com/, 2017. Consultado el 28/10/2017.

[2] Apple Inc. Model-view-controller. https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html, 2015. Consultado el 27/08/2017.

[3] Berners, L. Hypertext markup language - 2.0. https://tools.ietf.org/html/rfc1866, 1995. Consultado el 15/07/2017.

[4] Bracha, G.Cook, W. Mixin-based inheritance. http://www.bracha.org/oopsla90.pdf, 1990. Consultado el 29/09/2017.

[5] Bray, Ed. Json. https://tools.ietf.org/html/rfc7159, 2014. Consultado el 17/07/2017.

[6] Django Software Foundation. Faq: General. https://docs.djangoproject.com/es/1.11/faq/general/, 2017. Consultado el 27/08/2017.

[7] Django Software Foundation. Faq: General. https://docs.djangoproject.com/en/1.11/topics/signals/, 2017. Consultado el 28/08/2017.

[8] Dragoni, N.Giallorenzo, S. Microservices: yesterday, today, and tomorrow. https://arxiv.org/pdf/1606.04036.pdf, 2017. Consultado 28/07/2017.

[9] Esmite, I. Farías, M. Automatización y gestión de las pruebas funcionales usando herramientas open source. http://sedici.unlp.edu.ar/bitstream/handle/10915/21787/Documento_completo.pdf?sequence=1. Consultado el 25/07/2017.

[10] Esterbrook, C. Using mix-ins with python. http://www.linuxjournal.com/article/4540, 2001. Consultado el 29/09/2017.

[11] Fowler, M. Continuous integration. https://www.martinfowler.com/articles/continuousIntegration.html, 2006. Consultado el 28/07/2017.

[12] Git. Git. https://git-scm.com/, 2017. Consultado el 17/07/2017.

[13] Jasper, B. Django-jsonfield. https://pypi.python.org/pypi/jsonfield, 2017. Consultado el 04/12/2017.

[14] Jenkins CI. Jenkins. https://jenkins.io/doc/, 2017. Consultado el 18/10/2017.

[15] jQuery. Jquery. https://jquery.com/, 2017. Consultado el 10/11/2017.

[16] Microsoft. Model-view-controller. https://msdn.microsoft.com/en-us/library/ff649643.aspx. Consultado el 27/08/2017.

[17] Mozilla. Javascript. https://developer.mozilla.org/es/docs/Web/JavaScript, 1995. Consultado el 15/07/2017.

[18] Sam Newman. *Building Microservices*. O'Reilly Media, Inc., 1st edition, 2015.

[19] Oracle. The main features of mysql. https://dev.mysql.com/doc/refman/5.7/en/features.html, 2017. Consultado el 04/11/2017.

[20] Otto, M. Bootstrap. https://getbootstrap.com/, 2017. Consultado el 10/11/2017.

[21] Pytest. Pytest. https://docs.pytest.org/en/latest/, 2017. Consultado el 17/07/2017.

[22] Python Software Foundation. The python tutorial. https://docs.python.org/3/tutorial/index.html, 2017. Consultado el 15/07/2017.

[23] SpryMedia Ltd. Django-jsonfield. https://datatables.net/, 2017. Consultado el 10/11/2017.

[24] SQLite. About sqlite. https://www.amcharts.com/, 2017. Consultado el 04/11/2017.

[25] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 4th edition, 2014.

[26] The PostgreSQL Global Development Group. About postgresql. https://www.postgresql.org/about/, 2017. Consultado el 28/10/2017.

[27] Turpial Development. Manual informativo de turpial development, 2016. Documento Confidencial.

[28] M.B. Weinstein. *Total Quality Safety Management and Auditing*. Taylor & Francis, 1997.

APÉNDICE A

Manifiesto Ágil

Es un estatuto que promueve modelos organizacionales basados en la colaboración entre personas sin documentar de manera excesiva el proceso. Posee sistema que se adapta a las necesidades del cliente y lo incluyen de manera activa en el proceso de desarrollo. Se rige por los siguientes 12 principios.

1. La mayor prioridad es satisfacer al cliente mediante la entrega temprana y continua de software con valor.
2. La mayor prioridad es satisfacer al cliente mediante la entrega temprana y continua de software con valor.
3. Aceptar que los requisitos cambien, incluso en etapas tardías del desarrollo. Los procesos Ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente.
4. Entregar software funcional frecuentemente, entre dos semanas y dos meses, con preferencia al periodo de tiempo más corto posible.
5. Los responsables de negocio y los desarrolladores trabajan juntos de forma cotidiana durante todo el proyecto.
6. Los proyectos se desarrollan en torno a individuos motivados. Hay que darles el entorno y el apoyo que necesitan, y confiarles la ejecución del trabajo.
7. El método más eficiente y efectivo de comunicar información al equipo de desarrollo y entre sus miembros es la conversación cara a cara.
8. El software funcionando es la medida principal de progreso.
9. Los procesos Ágiles promueven el desarrollo sostenible. Los promotores, desarrolladores y usuarios debemos ser capaces de mantener un ritmo constante de forma indefinida.
10. La atención continua a la excelencia técnica y al buen diseño mejora la Agilidad.
11. La simplicidad, o el arte de maximizar la cantidad de trabajo no realizado, es esencial.

12. Las mejores arquitecturas, requisitos y diseños emergen de equipos auto-organizados.

13. A intervalos regulares el equipo reflexiona sobre cómo ser más efectivo para a continuación ajustar y perfeccionar su comportamiento en consecuencia.

APÉNDICE B

Manual de metodología de Turpial (resumen)

Descripción de la metodología TAUP

TAUP o Turpial Agile Unified Process es una metodología ágil creada por la empresa Turpial Development basada en los principios del "manifiesto ágil" que brinda a su equipo de trabajo una manera eficaz de llevar a cabo el desarrollo de software.

Fase de Concepción

El objetivo de esta etapa es que el equipo de desarrollo profundice la comprensión de los requisitos del sistema y validación de la arquitectura propuesta.

Historias de Usuario (HU)

Es una técnica para expresar requerimientos que se caracteriza por ser sencilla de escribir, leer y evaluar. El usuario final o cliente le dará valor a esta HU ya que describe una funcionalidad que él necesita.

Las historias de usuarios se escriben en oraciones que poseen tres componentes: el rol del usuario que quiere realizar la acción, que desea hacer y por qué. Una buena HU está caracterizada por ser independiente, negociable, valiosa, estimable, pequeña y comprobable (INVEST).

Adicionalmente, deben contar con criterios de aceptación bien definidos que brinden más detalle respecto a los aspectos de la Historia de Usuario que validan su completitud. Deben ser escritos en lenguaje natural y cada oración debe decir una funcionalidad que tendrá que ser ejecutada por la HU.

Por último, una HU, debe contar con pruebas de aceptación ya que van a permitir que el producto cumpla con los estándares y que se satisfagan los requerimientos establecidos.

Corresponden a una condición que se pueda verificar de los criterios de aceptación y una vez que han sido validadas, se puede dar por culminada la Historia de Usuario.

Priorizar Historias de Usuario

El cliente establecerá una prioridad para las Historias de usuario, la cual es uno de los factores que influyen en el orden de ejecución de las mismas. Se puede utilizar la siguiente escala de cinco valores: Alta, Media alta, Media, Media Baja y Baja.

Análisis de riesgo de Historias de Usuario

El equipo de desarrollo debe establecer un nivel de dificultad para cada HU, para esto, los desarrolladores deben tomar en cuenta diversos factores como: aprender nuevas tecnologías, manejar herramientas desconocidas o investigar un tema en particular. La escala cuenta con cinco valores: Alto, Medio Alto, Medio, Medio Bajo, Bajo.

Para este proyecto se utilizó, al igual en la sección anterior, una escala reducida para puntuar el riesgo: Alto, Medio y Bajo.

Análisis de Esfuerzo de Historias de Usuario

Representa una estimación de cuánto esfuerzo se requiere para culminar una Historia de Usuario, a esto se le denominará Story Points. Al momento de asignar los Story Points se deben considerar diversos factores como el riesgo, complejidad y desconocimiento de la tarea a realizar, entre otros. La escala de ponderación viene dada por la secuencia de Fibonacci.

Cuando una Historia de Usuario consigue un Story Point muy alto (usualmente un valor mayor a 8), deberá ser revisada nuevamente ya que podría dividirse en varias Historias de Usuario que puedan más manejables. Cada Story Point corresponde a lo que se le denomina un Login Point que posee el valor de uno y corresponde a un módulo de autenticación de usuarios de un
sistema.

El equipo de desarrollo debe estimar de manera consensuada los Story Point a través de una dinámica denominada Planning Poker; cada miembro puntuará las Historias de Usuario, se eliminarán el valor más alto y el más bajo, el resto se promedian.

Backlog

A la lista de todas las Historias de Historia de Usuarios construidas a partir de toda la información suministrada en los puntos anteriores se le denominará Backlog, al construirla se determina el orden se van a ejecutar tomando en cuenta la prioridad y el riesgo de la tarea, sin embargo, puede cambiar.

Release Plan

Es un calendario en el cual se agregarán los Sprint que sean necesarios para que todas las Historias de Usuarios sean cumplidas a cabalidad.

Definición de Listo "DoR"

DoR (Definition Of Ready) definido así por sus siglas en inglés, es un acuerdo que se establece en el equipo de desarrollo para cada Historia de Usuario que responde a la pregunta '¿Qué tiene que estar listo antes de empezar a trabajar en una Historia de Usuario y la misma pueda ser ejecutada en el próximo Sprint?'.

Se debe tener en cuenta los siguientes aspectos:

- Que sea independiente o de lo contrario que la(s) Historia de Usuario que lo preceden ya están implementadas o van a desarrollarse en el mismo Sprint.
- Que la Historia de Usuario tiene que ser desarrollada en un solo Sprint.
- Que los criterios de aceptación para la Historia de Usuario sean detallados
- Que las pruebas de aceptación puedan realizarse en el Sprint.
- Que la Historia de Usuario tenga asignado su Story Point.
- Tomar en cuenta si existe alguna dependencia externa para la Historia de Usuario.

Definición de Terminado "DoD"

DoD (Definition of Done) definido así por sus siglas en inglés, responde a la pregunta "¿Qué tiene que cumplir una Historia de Usuario para que se considere finalizada?", también puede incluir otros aspectos que sean considerados por el equipo de desarrollo como primordiales para que una HU sea considerada terminada. Este acuerdo explica con detalle los criterios para que toda HU sea considerada terminada.

- La Historia de Usuario ya ha sido analizada.
- El código tiene que estar escrito.
- El código tiene que estar documentado o comentado.
- El código se ha integrado a todo lo desarrollado con anterioridad.
- La Historia de Usuario ha pasado por todas sus pruebas de aceptación con éxito.
- Haberle mostrado la Historia de Usuario al cliente y que este la acepte.

Fase de Construcción

El enfoque de está fase es desarrollar todo lo planteado en la Concepción, validando así, la arquitectura planteada. La metodología se basa en Sprints o iteraciones.

Sprints

El desarrollo del software se tiene que ejecutar por bloques de duración corta y fija, de 5 días (semana laboral) hasta 1 mes. Cada Sprint tiene que proporcionar un resultado concreto, un incremento de producto que sea potencialmente entregable, de tal manera que pueda estar disponible para ser utilizado por el cliente y así, proporcionar observaciones de lo que se le está mostrando.

El primer día de cada Sprint se lleva a cabo la Reunión de Planificación, en la que todos los miembros del equipo revisan lo que se tiene planteado en el Release Plan para ese Sprint, verifican que cada una de esas Historias de Usuario estén en estado DoR para que sean divididas en tareas y se aclare cualquier duda acerca de lo que se va a realizar. Luego de que se conozcan cuáles son las tareas a desarrollar, se distribuyen a cada miembro del equipo y se colocan en un Kanban Board para saber cuáles están por hacer (To Do), cuales se están haciendo (Doing) y cuales están finalizadas (Done).

Durante el Sprint se debe realizar todos los días el Daily Stand Ups Meeting cuyos integrantes son el equipo desarrollador y su líder de proyecto. Tienen un horario (no debería de durar más quince minutos) y lugar fijo y los participantes tendrán tiempo para hablar y responder tres preguntas: qué ha realizado desde el último Daily, qué se está haciendo y qué bloquea su progreso.

El último día de cada Sprint, los miembros del equipo se deben reunir para realizar una Revisión de Iteración, en la cual estará presente el cliente para mostrarle todos los avances

y que dé su opinión para que el equipo realice las correcciones pertinentes lo más pronto posible. Luego, se realiza la Reunión de Retrospectiva, en donde se discute que hizo bien, que se puede mejorar y a que se compromete

Para cada Sprint es necesario ver qué cantidades de Story Points se van a realizar y cuáles se están trabajando y cuales están ya en DoD. Para esto se crea una gráfica que sirve para saber si se puede o no terminar lo planteado para el Sprint, llamada Burndown Chart, en la cual se coloca en el eje X el tiempo que dura el Sprint y en el eje Y la cantidad de Story Points. Esta práctica también sirve para referirse al avance de todo el proyecto, haciendo el mismo procedimiento antes explicado, pero cambiando los ejes de manera que en ellos se encuentre el total de Story Point del desarrollo del proyecto y todos los Sprint del mismo.

Fase de Transición

En esta fase se probará todo lo desarrollado en la fase de construcción. Estas pruebas funcionales y no funcionales verifican que el proyecto pueda ser utilizado en un ambiente de producción, además se creará un manual para el usuario, el cual le dará la capacitación necesaria al cliente para poder utilizar sin problemas el sistema.

Pruebas

Se realiza la validación del sistema a través de pruebas de integración, despliegue y comportamiento y que el mismo pueda ser desplegado en un ambiente de preproducción. También se validará la documentación, en donde se verificará que todo el código esté correctamente documentado para que pueda ser entendido con facilidad en caso de que sea necesario realizar cambios. Se crea el Manual de Usuario, en el que se especifica todas las funcionalidades del sistema para que el cliente pueda consultar si tiene alguna duda con respecto a la funcionalidad.

Por último, se verifica que todo lo anterior se haya cumplido, de no ser así se tendrá que hacer pruebas de regresión y modificaciones a la validación de la documentación hasta que el cliente esté satisfecho con el trabajo realizado.

Puesta en Producción

Luego de que el sistema esté completo, se tendrá que desplegar en el ambiente de producción, al terminar este proceso se otorgará un tiempo prudencial al cliente para efectuar pruebas. El reporte de estos defectos será más formal ya que llevarán un registro, junto con sus detalles para que los desarrolladores puedan corregirlos.

APÉNDICE C

Backlog del proyecto

Núcleo

Código	Historia de usuario	Prioridad	Riesgo	Story Points
C01	Como programador, quiero indicar cuáles elementos de mi base de datos quiero auditar para poder indicar a la librería cuales modelos debe llevarle seguimiento.	Alta	Alta	5
C02	Como programador, quiero poder almacenar cuando un elemento auditable fue creado así podré saber cuándo, quién y qué fue ingresado en el sistema.	Alta	Alta	5
C03	Como programador, quiero poder almacenar cuando un elemento auditable fue editado así podré saber cuándo, quién y cómo fue alterada la información en el sistema.	Alta	Alta	5
C04	Como programador, quiero poder almacenar cuando un elemento auditable fue eliminado así podré saber cuándo, quién y cuál la información fue borrada en el sistema.	Alta	Alta	5
C05	Como usuario, quiero disponer de listas de cada elemento auditable para poder observar el comportamiento de mi información.	Alta	Baja	2
C06	Como usuario, quiero que las listas de mis auditorías puedan ordenarse según autor, fecha de creación, acción efectuada sobre la data	Alta	Baja	1

C07	Como usuario, quiero que las listas de mis auditorías puedan filtrarse según autor, fecha de creación, acción efectuada sobre la data; para así facilitar la búsqueda de auditorías en específico.	Alta	Baja	1
C08	Como programador, quiero poder configurar el orden en que se desplegarán las columnas de mis listados, así mantener los patrones de experiencia de usuario de mi sistema.	Media	Media	2
C09	Como programador, quiero contar con etiquetas personalizadas que me permitan incluir cualquier listado de las auditorías en mi sistema	Alta	Media	3
C10	Como programador, quiero contar con un medio para personalizar el Look and Feel de mis listados de auditorías para ajustarlos al estilo de mi sistema.	Media	Alta	5
C11	Como usuario, quiero contar con un control de acceso a los listados de las auditorías y así solo brindar a un conjunto de usuarios las auditorías(Dividida en C11.1, C11.2 y C11.3)	Media	Baja	8
C11.1	Como usuario, quiero contar con un login para restringir el acceso a usuarios no autorizados a mis auditorías.	Media	Baja	2
C11.2	Como usuario, quiero poder autorizar/desautorizar usuarios al acceso de auditorías.	Media	Baja	5
C11.3	Como usuario, quiero poder listar a todos los usuarios autorizados a acceder a las auditorías.	Media	Baja	1
C12	Como usuario, quiero que el sistema se actualice con la última versión de la librería por medio de integración continua para garantizar el soporte al sistema en donde se instale la librería.	Alta	Alta	5

C13	Como desarrollador quiero que se instalen automáticamente las dependencias necesarias en el sistema que se instale la librería.	Alta	Media	3

Estadísticas

Código	Historia de usuario	Prioridad	Riesgo	Story Points
E01	Como usuario quiero ver las estadísticas globales entre todas las acciones auditables a lo largo de un rango de tiempo.	Alta	Media	2
E02	Como usuario quiero disponer de gráficas con las estadísticas globales.	Media	Media	2
E03	Como usuario quiero ver las estadísticas por acción auditable a lo largo de un rango de tiempo.	Alta	Media	2
E04	Como usuario quiero disponer de gráficas con las estadísticas por acción auditable.	Media	Media	2
E05	Como usuario quiero ver las estadísticas por autor a lo largo de un rango de tiempo.	Alta	Media	2
E06	Como usuario quiero disponer de gráficas con las estadísticas por autor.	Media	Media	1
E07	Como usuario quiero ver las estadísticas de una tabla de auditoría específica de la base de datos a lo largo de un rango de tiempo.	Alta	Media	3
E08	Como usuario quiero disponer de gráficas con las estadísticas de una tabla de auditoría específica.	Media	Media	1