UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

DISEÑO E IMPLEMENTACIÓN DE FRAMEWORK PARA SMART MUSEUMS

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

MICHEL ANDRÉS LLORENS ACUÑA

PROFESOR GUÍA:
NANCY HITSCHFELD KAHLER

MIEMBROS DE LA COMISIÓN:
NELSON BALOIAN TATARYAN
PATRICIO POBLETE OLIVARES

SANTIAGO DE CHILE
2017

Resumen

Dentro del marco del programa de colaboración entre la Universidad de Chile y la Universidad de Duisburg-Essen (PRASEDEC), se gestó la idea de la realización de museos virtuales para poder acercar a la gente piezas históricas y culturales no sólo del lugar en sí donde se encuentran, sino también de lugares remotos como lo es Armenia.

Hasta el momento, el enfoque que se le había dado al problema era la realización de software de escritorio y montar exhibiciones físicas para que la gente pudiera acercarse a ver estos museos virtuales. Ante ello, se buscó la forma de abstraer aún más esta digitalización y poder llevar estos museos a Internet, lugar donde se podrían ver las exhibiciones independientemente del lugar físico.

La presente memoria consistió en el diseño e implementación de un framework web para montar toda la estructura que podría necesitar un museo virtual (sin contar hardware) y mostrarlo a público. En ese esquema, se generó tanto un servidor web modular que permite el trabajo colaborativo de múltiples equipos de forma simultanea, así como plantillas de front-end de visualización de exhibiciones y manejo de recursos, para poder fácilmente extender el framework a cualquier tipo de recurso de una exhibición[1]. Sin embargo, ya que este proyecto está incluido en el marco de PRASEDEC, el framework por si sólo es funcional y está técnicamente listo para ejecutarse (salvo por temas de credenciales), permitiendo así que la estructura por defecto de un Exhibit sea un museo en tecnología Unity.

El trabajo fue validado realizando pruebas de extensibilidad para otros formatos de Exhibit, cambio de interfaces visuales (para los visitantes), deployment en un servidor de acceso público y presentado a potenciales usuarios, entre ellos al Smithsonian Institution.

Se concluyó que el trabajo realizado no sólo entrega un medio para facilitar la exposición de museos virtuales al público general, sino que a su vez reduce el tiempo necesario en implementación para ir desde la idea a la entrega al público, permitiendo así que tanto grandes como pequeñas entidades (museos por ejemplo) puedan utilizar esta herramienta para dar apoyo digital a sus visitantes, pues el Framework quedó abierto a público de manera Open Source.

[1]Debido a que en español no existe una única palabra para definir este tipo de recurso, se empleará el inglés utilizando el término Exhibit, que es parte de una Exhibition o Exhibición en español.

A todos aquellos que encontraron su profesión en su hobby y su trabajo en su diversión.

Agradecimientos

Inicialmente quisiera agradecer a mi núcleo familiar. A mis padres, a mi abuela y a mi hermana por todo su apoyo y paciencia en estos largos años que fueron y siempre serán los más eternos de mi existencia, y que pese a todas las caídas, quejas y malas caras nunca me dejaron de animar ni apoyar en esta decisión.

Así mismo quisiera agradecer a mis amigos Alex y Oscar por estar allí, pese a no entender nada ni saber de mi existencia en semanas o meses. Y a mi amigo Damián por molestar perseverantemente y finalmente hacer que entrase y terminase computación, y no dejarme de recordar que cuando tenga tiempo libre comeremos una hamburguesa.

A Catalina, por toda tu paciencia, apoyo incondicional, comida y amor en lo que fue mi periodo más complejo de la universidad y de mi vida. Y bueno, también gracias de antemano por todo lo que nos deparará el futuro ahora que esto culminó.

También quisiera agradecer a los tres profesores más importantes en mi carrera en orden de aparición. Al profesor Baloian por enseñarme computación y demostrarme que tenía madera para dedicarme a esto. Al profesor Patricio Poblete por acogerme, enseñarme y acercarme a lo bonito que es la computación y hacer que así, esto se convirtiera en mi futuro. Y finalmente a mi profesora guía Nancy por guiarme y orientarme no sólo en esta memoria, sino en mis últimos dos o tres años de carrera para ser el profesional que hoy soy.

Finalmente, aparte de agradecerle a toda la gente que no fue nombrada pero sabe que de una u otra forma es parte de mi y de este mérito, quisiera firmemente darme las gracias a mi mismo. Por todos esos eternos momentos en que pude desertar y no lo hice, por todo lo complejo físicamente y psicológicamente que fue llegar a este punto. Costó, pero se pudo y se logró. Gracias.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Índice de Códigos

Capítulo 1

Introducción

En el presente capítulo, se explica en términos generales el contenido de este documento. Para cumplir con lo anterior, primero se presenta el contexto bajo el cual se generó la presente memoria y se abordó el problema; luego se presenta el foco principal del proyecto en torno al cual se desarrolla la solución; y finalmente la presentación concreta de los objetivos a cumplir. En adición a lo anterior, se incluye una sección a modo de resumen general de los capítulos siguientes.

1.1. Contexto

Con el fin de entender el tema central de esta memoria y el porqué de ella, es importante presentar el contexto en el cual se identificó el problema, se buscó cómo solucionarlo y se contrastaron soluciones ya existentes. En este caso particular, la raíz del tema yace en el programa de colaboración entre la Universidad de Chile y la Universität Duisburg-Essen, y el cómo distribuir más cómodamente cultura al mundo.

1.1.1. PRASEDEC

El programa de colaboración entre la Universidad de Chile y la Universität Duisburg-Essen, denominado PRASEDEC por sus siglas en inglés[1], ha estado directamente involucrado en un área llamada *Knowledge communication and mobile social infrastructure*, o bien, en español, *Comunicación del conocimiento e infraestructura social móvil* y ligado a esto corresponde todo lo relacionado a las *'ciudades inteligentes'*. Por ello mismo, un tema a abordar son los **Smart Museum** y la idea de virtualizar museos, o piezas históricas, tanto para su preservación como también para su exhibición.

Dicho lo anterior, el proyecto de Smart Museum se centró en la digitalización de cemente-

[1] **Pr**actice-driven **A**dvance of **S**tudies and **E**xchange between the Universities of **D**uisburg-**E**ssen and **C**hile.

rios en Armenia, con el fin de preservar los denominados **Khachkar**, o piedras conmemorativas grabadas, las cuales desde el año 2012 forman parte del listado de *patrimonio cultural inmaterial de la Humanidad* de la UNESCO. Para esto, el equipo de PRASEDEC utilizó diversas tecnologías basadas fundamentalmente en el lenguaje de programación JAVA y en software propietario para generar pequeñas exhibiciones digitales, las cuales se presentaban físicamente utilizando computadoras.

Durante la reunión anual del año 2016 se presentó la intención de llevar estos museos digitales a una forma más accesible al público utilizando Internet. Sin embargo, también se buscaba la forma de no ligarse de forma permanente a una única tecnología para que, de dicho modo, los museos y sus exhibiciones pudieran irse actualizando sin caer en lo obsoleto.

Otros problemas que se detectaron en las versiones del software de escritorio ya utilizadas radicaban en la centralización de las tareas en los programadores, pues ellos debían generar las exposiciones, construir modelos 3D, conseguir recursos como música o texturas, entre otras cosas. Así mismo, también estaban a cargo de la tarea de recibir y procesar feedback por parte de los visitantes de las exhibiciones con respecto elementos en particular de éstas.

Bajo todo ese marco, se generó la presente memoria que tiene como finalidad entregar una plataforma sobre la cual se puedan presentar y manejar museos virtuales utilizando tecnologías web. En consecuencia, y para lograr esto, ésta debe poder adaptarse a diversas tecnologías de visualización a fin de brindarle más y mejores recursos a los visitantes; y en lo referente a lo administrativo del museo, la posibilidad de permitir división de trabajo y responsabilidades para repartir de mejor manera los recursos humanos que conlleva el mantener una exhibición funcionando a toda hora (por estar en Internet), utilizando funcionales tales como agendas de exhibiciones, capacidad de almacenar recursos digitales, capacidad de almacenar elementos particulares a ser exhibidos (como vídeos, animaciones digitales, presentaciones, etc), almacenaje y capacidad de interactuar frente a opiniones y comentarios entregados por visitantes, y la creación de exhibiciones como tal. Todo lo anterior a través de utilización de grupos de trabajo y limitaciones de visibilidad.

1.1.2. Museos virtuales en Internet

Del mismo modo que PRASEDEC llegó a la conclusión de que con las tecnologías actuales era necesario, de una u otra forma, migrar hacia Internet, en el mundo de los museos físicos, estos también llevan bastante tiempo intentando generar directrices de cómo se debería hacer la transición pasando desde páginas estáticas con fotografías hasta vistas de 360°. A continuación se presentan ejemplos de virtualización de museos en conjunto a sus características principales.

- **Staatliche Kunstsammlungen Dresden :**[2] Utilización de fotografías panorámicas (ver Figura 1.1) para reconstruir un espacio 3D utilizando una tecnología análoga al servicio Street Map de Google.
- **Salvador Dali Foundation :**[3] Utilización de Adobe Flash Player para construir un

[2]https://ruestkammer.skd.museum/en/ausstellungen/riesensaal/
[3]https://www.salvador-dali.org/en/

espacio 3D con base en fotografías panorámicas, análogo al servicio Street Map.

- **Vatican Museums:** [4] Galería web dedicada que contiene imágenes, textos, vídeos, audios.
- **National Gallery of Art (Estados Unidos):** [5] Composición entre recursos estáticos (fotografías, textos y audios) y paneles de navegación para cambiar entre páginas que representan cuartos y/o orientaciones espaciales.
- **British Museum:** [6] Colaboración con Google Cultural Institute para tener una galería de imágenes con información respectiva, navegación 3D utilizando Google Street View y líneas de tiempo interactivas utilizando WebGL.
- **Smithsonian National Museum of Natural History:** [7] Vista dedicada para navegación 3D utilizando una tecnología análoga al servicio Street Map de Google, pero además incluyendo un mapa para navegar directamente entre cuartos y pisos.
- **Louvre:** [8] Utilización de fotos panorámicas de forma directa con vínculos para cambiar de habitación (también con un mapa de navegación). Todo lo anterior utilizando Macromedia Flash Player

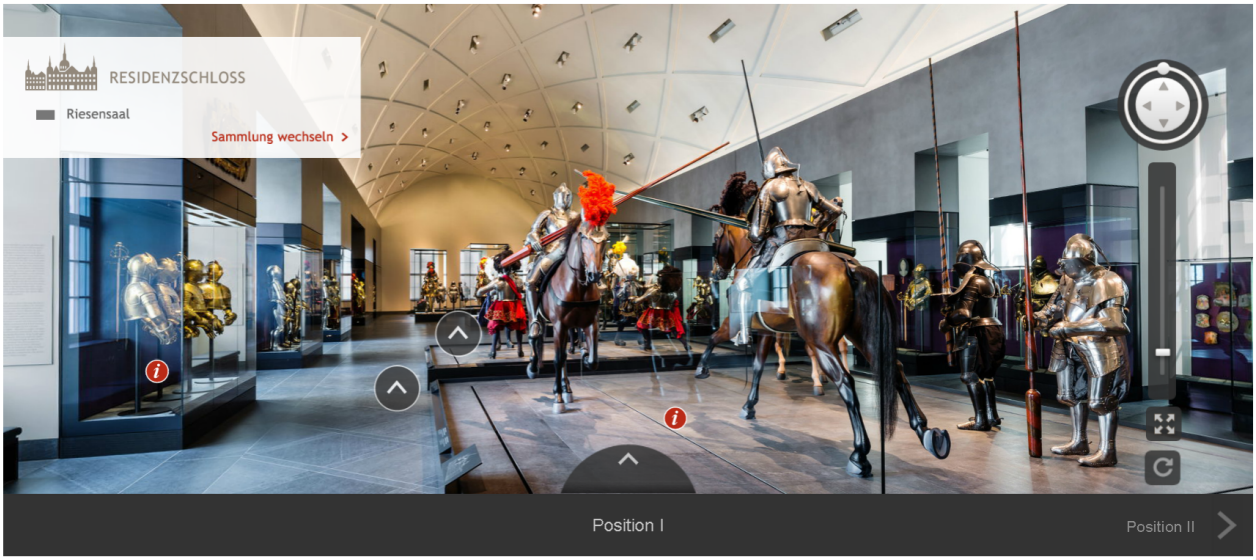

Figura 1.1: Museo virtual a base de fotografías panorámicas.

Sin embargo, estas soluciones están empotradas en las páginas web de cada institución, delegando así el trabajo en el equipo de programación para hacer que cada nueva exposición calce bien en las páginas ya existentes. Además, muchas de estas exhibiciones están ligadas directamente con una única tecnología, de las cuales algunas serán eliminadas de los navegadores web en los próximos años (como lo es Flash). En consecuencia, una solución así no cumple con lo esperado en el esquema de delegación de trabajo, subdivisión de éste y man-

[4] http://www.museivaticani.va/content/museivaticani/en.html
[5] https://www.nga.gov/content/ngaweb.html
[6] http://www.britishmuseum.org/with_google.aspx
[7] http://naturalhistory.si.edu/VT3/
[8] http://www.louvre.fr/en

tenibilidad futura. Se entrará en más detalles con respecto a éstas tecnologías en el **Marco Teórico**.

1.2. Framework

Como bien se indicó anteriormente, la idea tras la presente memoria era generar una plataforma adaptable a diversas tecnologías y que fuera independiente de éstas. Por ello se llegó a que la mejor solución era el diseño e implementación de un framework. Un framework corresponde a una estructura, tanto conceptual como de implementación, para asistir determinados procesos, utilizando módulos concretos de software y permitiendo así la extensión de éste hacia los propósitos deseados. Los principios fundamentales que debe seguir son:

1. **No es una biblioteca**. El framework dictamina el flujo de control del programa o, en este caso, de la plataforma web.
2. Posee un comportamiento predeterminado, el cual debe ser funcional, claramente identificable y acotado.
3. Debe ser extensible, ya que el framework funciona como estructura básica para objetivos más específicos que se le deseen dar.

De dicho modo, un framework se ajusta con precisión a lo buscado como solución al problema planteado y pasa así a ser el objetivo principal de la presente memoria.

1.3. Objetivos

En esta sección se presentarán los objetos que guían el proyecto, comenzando de manera general por los objetivos principales para luego desglosarlos en objetivos específicos.

1.3.1. Principales

Los objetivos principales corresponden a diseñar, implementar y desarrollar un framework web destinado a museos virtuales. Para esto, el Framework[9] deberá contemplar las siguientes funcionalidades o características:

- Debe ser extensible, mantenible, seguro y tener un comportamiento funcional mínimo.
- Debe contemplar dos grupos de usuarios: visitantes que navegarán por el sitio viendo exhibiciones del museo; y curadores[10] los cuales deben poder crear, organizar y presentar

[9]Se hará diferenciación entre *Framework* y *framework*, siendo el primero el proyecto de la memoria y por tanto un nombre propio mientras el segundo corresponde al término en sí.

[10]Se denomina Curadores a los profesionales con un extenso conocimiento, experiencia y capacidad para desarrollar y estructurar museos. Dándole así semánticas a las exhibiciones que son captadas por su público. En el presente trabajo también se le dará énfasis al **Equipo de Curadores** o simplemente **Curadores** como

exhibiciones desde una interfaz administrativa.

- Debe contemplar exhibiciones de distintos tipos en su diseño, pues deberá ser extensible a tecnologías actuales y futuras, pero teniendo como mínimo funcional para su entrega, elementos de **Unity**[11] que deberán poder visualizarse.

1.3.2. Específicos

Con el fin de cumplir los objetivos generales, se plantea el siguiente desglose de objetivos específicos, los cuales a su vez en la etapa de desarrollo se subdividen siguiendo el patrón de metodologías ágiles y *features*[12].

- Creación de un servidor back-end robusto y modular para facilitar la creación de nuevo contenido e interfaces.
- Diseño e implementación de base de datos para el manejo de recursos, opiniones, usuarios y exhibits del museo.
- Diseño e implementación de interfaz administrativa para el equipo de curadores del museo, teniendo en cuenta la extensibilidad de esta.
- Diseño e implementación de interfaz para la navegación y visualización de las exhibiciones del museo, teniendo en cuenta los distintos exhibits a mostrar y organización jerárquica de estos.
- Chequeo de compatibilidad del Framework con distintos navegadores web.
- Pruebas de extensibilidad del Framework.
- Pruebas de deployment del Framework.
- Dejar la implementación abierta a la comunidad como aporte al desarrollo científico y cultural.

1.4. Contenidos de la Memoria

La presente memoria está organizada de la siguiente forma. El Capítulo 2 presentará y explicará los conceptos claves, además de componentes técnicos necesarios para poder entender el proyecto. El Capítulo 3 abarcará las especificaciones del problema a resolver y su respectivo desglose. El Capítulo 4 describirá en detalle el diseño de la solución y la implementación de esta. Por otro lado, el Capítulo 5 utilizará la solución desarrollada para buscar una validación de esta a través de distintas pruebas. Finalmente, el Capítulo 6 contrastará los objetivos de la memoria con las validaciones realizadas, para concluir sobre ello y dar lugar al trabajo futuro.

---

el equipo humano tras la administración de un museo, ya sea virtual o físico.

[11] Motor de videojuegos multiplataforma (computadores, consolas de videojuegos, celulares y navegadores web): https://unity3d.com/es/

[12] Palabra anglosajona referente a *característica*. Se utiliza para mencionar características específicas y mínimas de un software.

Capítulo 2

Marco Teórico

En el presente capítulo se incluyen todos los conceptos necesarios para entender cómo se desarrolló la solución y bajo qué contextos se tomaron decisiones de diseño. En consecuencia, se presentarán soluciones actuales al problema como entrada a la toma de decisiones, las cuales contemplan: metodologías de desarrollo, arquitecturas de software, definiciones y conceptos relevantes a la utilización de un framework; qué tecnologías se utilizarán; finalmente, retomando conceptos más abstractos, se hablará sobre seguridad informática y la usabilidad de interfaces.

2.1. Soluciones actuales

Al inicio del presente documento, y como se indicó por **Biella et al.**[2], existen pocas implementaciones para Smart Museums de tipo web que aprovechen los beneficios de las tecnologías 3D y las posibilidades que la misma plataforma web entrega. Reduciéndose así, en forma general, a galerías de objetos 3D del estilo catálogo, o bien a fotografías panorámicas como se indicó en la **Introducción**.

Dicho lo anterior, se propuso repasar los museos mencionados en la **Introducción** a fin de analizar qué características son útiles, cuales no y qué cosas se podrían utilizar para la presente memoria. De lo cual se desprende el siguiente listado de características a contemplar en el Framework.

- No vincular estructuralmente una visualización con el resto del sitio web, a fin de permitir su cambio sin comprometer el resto de la plataforma.
- Aislar la visualización a su propia página en el sitio, a fin de que el visitante solo observe lo que él quiere visualizar.
- Permitir que cada recurso se visualice de forma independiente y, así, permitir que éste utilice la forma óptima para hacerlo.
- Debido a que las tecnologías de visualización van evolucionando a través del tiempo, permitir la integración de éstas a través de extensibilidad. A modo de permitir así la

inclusión de tecnologías como Street View, pero también de tecnologías que pronto dejarán de funcionar como lo son Flash. De este modo, se puede reutilizar material de exhibiciones en caso de migraciones.

Sin embargo, entre las soluciones existentes, pero no publicada directamente en algún museo, destacaba como buena aproximación, en términos de visualización y navegación, el trabajo de **Kiourt et al.**[8], quienes propusieron la utilización de motores de videojuegos para la generación de museos virtuales, en su caso la utilización de Unity. No obstante, dicha opción se encontraba acotada por dos situaciones: No existía posibilidad para que de un curador sin experiencia en programación pudiese generar museos desde cero; además, Unity permitía visualización web a través de NPAPI[1] pero, debido a problemas de seguridad detectados en este, los navegadores web más actuales tienen deshabilitada la opción de ejecutarlos (de forma análoga a los Applets de JAVA).

En vista del anterior problema relacionado a la visualización, se planteó la utilización de WebGL como un estándar para las vistas, pero, debido a que es muy lento y complejo en términos de programación el escribir código directamente, dado que es una librería de bajo nivel para aplicaciones web (que suelen ser de alto nivel), se decidió investigar el uso de librerías wrapper[2] para incorporar un módulo de generación de museos como parte del Framework; se considera la utilización de Three.js[3], WhiteStormJS[4] o BabylonJS[5] bajo lo comparado en términos teóricos y prácticos en la Tabla 2.1.

Durante el presente año, se detectó que Unity reemplazó su modelo web por la traducción directa de su código y recursos a WebGL de forma eficiente; esto, sumando a un proyecto de estudiantes asociados a PRASEDEC, que diseñaron e implementaron un esquema de trabajo utilizando Unity, que permite a gente sin experiencia en computación generar museos en Unity y por tanto, se logró combinar ambos sucesos para llevar a cabo museos web 3D de alta calidad sin tener que utilizar directamente WebGL (o wrappers) para dicha tarea. Por tanto, en vez de enfocar parte del proyecto en crear un módulo para generar museos, se cambió a la implementación de un módulo genérico y extensible para la visualización de distintos tipos de Exhibit, teniendo como requisito basal la visualización de Unity utilizando WebGL.

Tabla 2.1: Tabla Comparativa entre Three.js, WhiteStormJS y BabylonJS.

	Three.js	WhiteStormJS	BabylonJS
Complejidad de uso	Mediano	Mediano	Fácil
Complejidad de integración	Fácil	Difícil	Fácil
Documentación	Buena documentación y extensos tutoriales/ejemplos	Incompleta	Buena documentación
Facilidad de Actualización	Es retro compatible	Puede romper compatibilidad	Es retro compatible
Trabajo con cuerpos 3D	Soporta OBJ y STL	Soporta OBJ y STL	Soporta OBJ y STL
Aplicación de texturas	Fácil	Fácil y soporta cuerpos irregulares	Fácil

[1]Netscape Plugin Application Programming Interface
[2]Librerías que 'envuelven' (wrap en inglés) a otras para facilitar su uso.
[3]https://threejs.org/
[4]https://whsjs.io/
[5]https://www.babylonjs.com/

2.2. Metodologías de desarrollo

Para el desarrollo del presente proyecto se planteó la utilización de *metodologías ágiles* por sobre ingeniería de software clásica. La razón de esto corresponde a que, pese ser el trabajo de una única persona, es mucho más cómodo para el versionamiento de software el trabajar sobre features en específico, y proceder de forma incremental en lugar de ir trabajando sobre *issues*[6], dejando así estos últimos sólo para la notación de errores o conflictos por arreglar.

Así mismo, esta idea se incorporó a la metodología de desarrollo incremental clásica, para ir trabajando en base a etapas, y dentro de cada etapa trabajar sobre features, siguiendo el siguiente esquema:

1. Identificación del feature a implementar.
2. Investigar si las tecnologías o conocimientos actuales son suficientes para la resolución de este. De no ser suficientes, investigar entorno al feature.
3. Diseñar solución.
4. Implementar solución.
5. Realizar comprobaciones de comportamiento para ver correctitud de la implementación.
6. De existir problemas, intentar solucionarlos con la solución propuesta o bien, volver a realizar todo el procedimiento desde el punto 1.

De esta forma, se busca tener en todo instante una solución funcional y, sobre esta, ir construyendo una nueva versión funcional, lo que a corto y largo plazo ayuda a la gestión de tiempo para el proyecto y a que se pueda identificar directamente las features implementadas y las pendientes.

2.3. Arquitecturas

Teniendo en mente los objetivos y el cómo se trabajará, lo siguiente es identificar qué tipo de Arquitectura de Software se empleará, a fin de enfocar todo el diseño e implementación hacia esta.

Para esto, es necesario indicar cuál es el uso esperado de la aplicación (o servicio) que se creará. En este caso, al tratarse de una aplicación web, se espera poder tener un servidor físico que ejecute la aplicación, puntos de acceso remoto que visiten la aplicación, y una capacidad de almacenamiento para los datos persistentes de la aplicación. Además, se espera que la aplicación sea extensible, mantenible y escalable como buenos principios básicos para un framework, el cual se puede utilizar para una aplicación que reciba desde un visitante a miles de estos casi indistintamente.

Sin embargo, y difiriendo de la aplicación académica, el acotarse a una única arquitectura limita las posibilidades de una aplicación, pues, en la práctica, se buscan las mejores

[6]Un issue corresponde a un problema en particular de un software y está comúnmente relacionado uno a uno con los requisitos de software o de usuario

características de cada uno. Por ello se planteó la utilización de dos arquitecturas complementarias entre si, como lo son **Arquitectura de tres capas** y **Modelo-Vista-Controlador (MVC)**, las cuales se explicarán a continuación en conjunto a sus respectivas justificaciones.

2.3.1. Arquitectura de tres capas

La arquitectura de tres capas (ver Figura 2.1) corresponde a un patrón de arquitectura de software enfocado en dividir las tareas en tres capas bien identificadas, las cuales son:

- **Capa de presentación**: Corresponde a la interfaz visual de los usuarios que tiene como función traducir la información proveniente de los servidores a una forma comprensible por los usuarios. Un ejemplo de esto corresponde a la visualización de los datos de una base de datos, las cuales se podrían mostrar como una tabla, como un gráfico o como un número, según el fin práctico que se busque.
- **Capa lógica**: Corresponde a la coordinación, procesamiento y decisiones que debe realizar el servidor para poder llevar la información al usuario.
- **Capa de datos**: Corresponde a las bases de datos o sistemas de almacenamiento de la información, las cuales sólo responden a las peticiones explícitas de la Capa lógica.

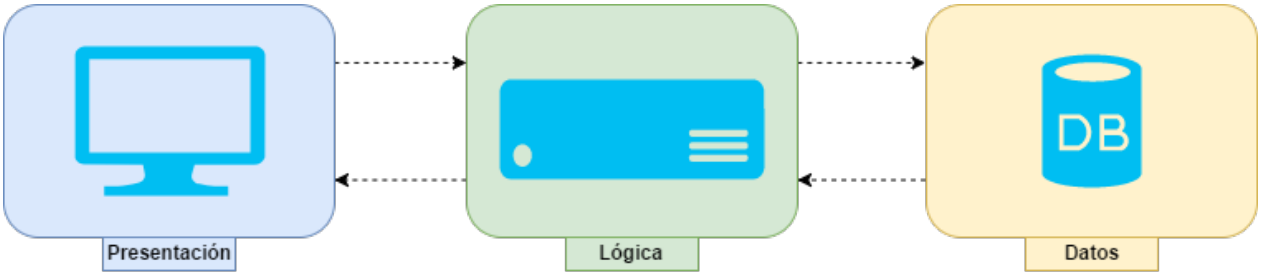

Figura 2.1: Arquitectura de tres capas.

En mira de lo anterior, es claro ver el por qué se planteó esta arquitectura, ya que permite la división entre responsabilidades, así como también la escalabilidad de la solución propuesta. La razón de esto último es que cada capa funciona de forma autónoma y, por tanto, es divisible. No obstante, dicha división no siempre es literal, pues la mayor parte de las veces la presentación la entrega el mismo servidor, por tanto no es factible separarlas físicamente. Por otro lado, la capa lógica se conecta a la capa de datos, la cual puede tener, por ejemplo, una o más bases de datos. Así mismo, se puede tener uno o más servidores que simulen ser uno utilizando **distribución de carga**; este tipo de escalabilidad se denomina **Escalabilidad Horizontal**(ver Figura 2.2), pues a fin de permitir un mayor flujo de usuarios, en vez de optimizar un único servidor, se emplea más de uno. La razón de esto es que un computador físicamente tiene limitantes, entonces, independiente de sus especificaciones técnicas, eventualmente puede conllevar un cuello de botella al atender usuarios. Por otra parte, el aumentar la capacidad de las máquinas se denomina **Escalabilidad Vertical**.

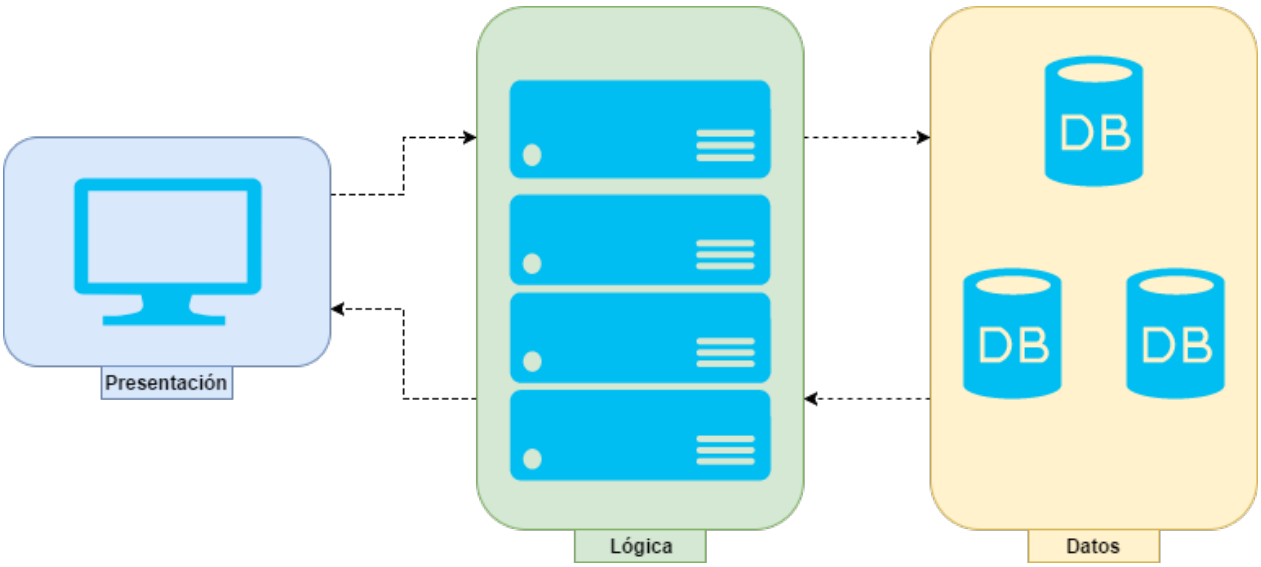

Figura 2.2: Arquitectura de tres capas escalada horizontalmente.

2.3.2. Modelo-Vista-Controlador

La arquitectura MVC se enfoca en dividir las tareas de un software en tres grupos (que se describirán a continuación), con el fin de modularizar este mismo y permitir la extensibilidad de sus partes de forma independiente. De dicha forma, es posible realizar modificaciones en metodologías para resolver problemas sin necesitar modificar los datos ni cómo se verán. Permitiendo, de ser necesario, trabajo colaborativo y en paralelo para la mantención, extensión y documentación del software.

Los tres grupos de MVC son los siguientes, y derivan directamente del nombre del modelo.

- **Modelo**: Corresponde a la parte de los datos de una aplicación, siendo este el que se modifica, y de forma indirecta actualiza la vista.
- **Vista**: Corresponde al output o salida percibido por un usuario de la aplicación. No siempre lo recibido por un usuario corresponde a una única Vista, pues el resultado puede corresponder a la visualización de múltiples vistas.
- **Controlador**: Corresponde a la lógica que acepta input o entrada por parte del usuario, le manda órdenes a los modelos y gestiona las vistas.

Dicho esto, se puede establecer una relación uno es a uno con la Arquitectura de tres capas, relacionándose Vista con Presentación, Lógica con Controlador y Modelo con Datos. Volviendo así estas dos arquitecturas compatibles.

Sin embargo, la razón fundamental de elegir ambas arquitecturas radica en que la Arquitectura de tres capas es útil en términos físicos, por ejemplo, en cómo montar una aplicación y a su vez distribuirla a los usuarios objetivo; mientras que la Arquitectura MVC está enfocada a los desarrolladores y en cómo estos van a hacerse cargo del código de la aplicación durante su vida útil. Por dichas justificaciones se decidió trabajar bajo ambos paradigmas.

2.4. Framework

Tal y como se indicó en la **Introducción**, los frameworks siguen tres principios fundamentales, no obstante, es importante especificar la diferencia entre los distintos tipos de frameworks existentes, que corresponden a **black-box**, **white-box** y **gray-box**; ya que, tal y como especifican **Riehle et al.**[12] y **Fayad et al.**[5], sus diferencias son muy importantes al momento de decidir qué tipo, y, en consecuencia, qué metodología es más correcta aplicar para cada proyecto.

2.4.1. Black-box

Se utiliza la denominación black-box, o caja negra, para cualquier framework donde los usuarios, o en este caso desarrolladores, pueden hacer uso directo de este sin tener que crear nuevas subclases o tener que integrar código dentro del núcleo de este.

Generalmente este tipo es compilado y de tipo propietario, y, por tanto, cualquier cambio distinto al paradigma con que se diseñó necesita inevitablemente la asistencia de los desarrolladores del framework. Pues, reiterando, se trata de una caja negra para el usuario, donde se ingresa un input y se obtiene un output, sin saber cómo se hizo realmente.

Parte de las características que conlleva utilizar black-box son:

- No se necesita conocimiento del framework.
- Puede utilizarse componiendo objetos entregados por el framework.
- Fácil de utilizar y extender, pero complejo en términos de desarrollo.

2.4.2. White-box

La denominación de white-box, o caja blanca, viene de la transparencia del código de un framework al estar totalmente disponible al momento de desarrollar, lo que, a su vez, se muestra al tener que utilizar herencia sobre estructuras provistas por este, o bien la utilización de plantillas o similares, y en consecuencia está muy ligado al paradigma orientado a objetos.

Normalmente está asociado a código abierto, pues para su funcionamiento el código debe poder ser visto y analizado, ya sea con fines de extender correctamente la aplicación o con el fin de poder optimizar tiempos de ejecución, a manera de ejemplo.

Los beneficios y costos que conlleva utilizar white-box son:

- Entrega el código fuente para ser usado como núcleo directo de la nueva aplicación.
- Requiere conocimiento de este para poder utilizarlo.
- Suele basarse en herencia de clases.
- Fácil de desarrollar, complejo de usar y extender.

2.4.3. Gray-box

Tal y como se observó en **Johnson et al.**[7], **Aoyama et al.**[1] y **Markiewicz et al.**[9], acotarse a elegir sólo blanco o negro lo único que hace es cortar posibilidades y cuando se quiere llevar un proyecto a un entorno de producción más que una ventaja es una desventaja para el futuro del proyecto. Por tanto se planteó el uso mixto de ambos paradigmas, en lo denominado gray-box, o caja gris.

La ventaja de esta metodología es que la mayor parte del framework es visible por el usuario en lo que respecta al funcionamiento particular que desea desarrollar o extender, mientras que los procesos más complejos, o que son parte del núcleo actúan como caja negra, con el fin de evitar que procesos críticos fallen o desvíen su flujo de control.

Un ejemplo de esto es la visibilidad de todas las clases necesarias para generar herencia y extensión, a fin de poder ver cómo funcionan, entenderlo y aplicarlo. Mientras que partes más críticas, como el manejo de requests (o consultas) por parte del protocolo HTTP hacia una clase que puede ser trabajada, se hace en oscuro para evitar que cualquier intervención provoque una potencial falla del principio funcional que debe tener el framework.

En consecuencia, se decidió que la utilización de un gray-box es lo más óptimo para el proyecto, ya que parte de sus principios es que debe ser extensible, mantenible y open-source. Pero, por temas de alcance, debe construirse sobre otro framework ya existente, el cual, a su vez, actuará como caja negra llegando así a lo buscado.

2.5. Tecnologías utilizadas

En base a lo anterior, se buscó un framework base para realizar aplicaciones web y sobre el cuál se construirá el propio.

Entre las opciones encontradas se postuló la utilización de:

- Java Servlets[7].
- Ruby on Rails[8].
- Python Django[9].
- Node.js[10].

Además, se buscó que el framework base cumpliera un listado de características que puede verse a continuación:

- Nivel de complejidad al momento de programar sobre el framework y extenderlo.
- El código debe ser legible, extensible y mantenible.

[7] http://www.oracle.com/technetwork/java/index-jsp-135475.html
[8] http://rubyonrails.org/
[9] https://www.djangoproject.com/
[10] https://nodejs.org/es/

- El servidor debe ser robusto y modular.
- Debe poder utilizarse cualquier base de datos, o al menos cubrir un amplio espectro. Además para evitar problemas relacionados al lenguaje SQL debe poder abstraerse de éste.
- Debe poder hacerse un deployment[11] de forma intuitiva e idealmente automatizada.
- En caso de necesitar utilizarse librerías externas, estas deben ser fácilmente incorporables y no debe afectar a todo el sistema donde se vaya a ejecutar.

Dicho lo anterior, se realizó una comprobación empírica de las alternativas que resultó en la Tabla 2.2. Lo cual decantó en la utilización de **Python Django**, pues además, Python permite que las dependencias se puedan trabajar a través de ambientes virtuales, mediante el uso de la librería nativa **pip**.

Tabla 2.2: Tabla comparativa entre frameworks basales.

	Java Servlets	Ruby on Rails	Python Django	Node.js
Complejidad al programar	Medio	Alto	Medio	Alto
Código legible, extensible y mantenible	Si	Problemas de mantenibilidad	Si	Problemas de legibilidad
Servidor robusto y modular	Si	Si	Si	Modular
Independencia de SQL	Media	Media	Alta	Alta (librerías externas)
Deployment intuitivo	No	No	Si	Si
Encapsulamiento librerias	Si	Si	Si	Si

Ahora bien, para la parte de usuarios del framework es necesario utilizar librerías de Javascript, pues es el lenguaje que los navegadores web entienden. Pero a su vez, las librerías de Javascript son muy específicas y, por tanto, suelen usarse varias, e incluso unas usan otras como base (poniendo como ejemplo jQuery y sus extensiones). Dicho esto, se decidió utilizar la aplicación de npm **Bower**[12], la cual permite realizar un listado de dependencias de Javascript en un archivo de texto (con o sin versión específica) para bajarlas automáticamente desde sus repositorios.

Las librerías de Javascript que fueron usadas son:

- Bootstrap
- Font Awesome
- Roboto Fontface
- jQuery y sus complementos Backstretch, UI y Bar-rating

2.6. Seguridad informática

Uno de los mayores problemas que hoy debe enfrentarse la computación radica en la seguridad informática. Durante años la defensa primordial radicaba en los computadores mismos y la utilización de software del estilo antivirus para prevenir virus, troyanos y gusanos provenientes de archivos infectados, ya sea mediante descarga o con acciones simples, como

[11] Proceso de puesta en marcha de un servicio. Será explicado en detalle en su respectiva sección.
[12] https://bower.io/

utilizar un disco o diskette infectado. Hoy en día la historia es aún más compleja con la proliferación del internet y las aplicaciones web, pudiendo vulnerar a los servidores mismos con sólo realizar request simples y, con ello, obtener muchísima información valiosa.

A continuación se hablará de dos principales problemas de las aplicaciones web y cómo prevenirlos, o bien sanitizarlos, tal y como indica **Stuttard et al.**[13] en su libro, los cuales corresponden a **ataques de acceso a control** e **inyección de SQL**.

2.6.1. Ataque de acceso a control

Tal y como hace mención su nombre, corresponde a ataques que buscan tener acceso a poderes administrativos con el fin de poder atacar el servicio y/o obtener información sensible de este. Comúnmente se relaciona con la exposición directa de URLs administrativas a público y nulo manejo de permisos de usuarios para determinadas acciones. A modo de ejemplificar, asumamos que tenemos el método para obtener a todos los usuarios del sistema, que se utiliza internamente para mostrarlos al administrador. Este perfectamente puede estar asociado a la URL `http://my-app.com/admin/utilities/getUsers`, por tanto, cualquier persona que llame a esa página obtendrá un JSON como el del Código 2.1 sin necesitar ser administrador.

```
[{
    "firstName": "John"
    "lastName" : "Smith",
    "age" : 25
},{
    "firstName": "Juan"
    "lastName" : "Smith",
    "age" : 21
},,{
    "firstName": "Pedro"
    "lastName" : "Smith",
    "age" : 28
}]
```

Código 2.1: Ejemplo JSON usuarios.

De forma similar, algunas páginas utilizan una flag para comprobar si se trata o no de un administrador y la envían al sistema, ya sea por POST o GET, como en la siguiente URL `http://my-app.com/admin/utilities/getUsers?admin=true`.

Para solucionar esto, se recomienda la utilización de tokens CSRF[13], una suerte de llave de Hash suficientemente larga para no poder ser generada aleatoriamente y aceptada de forma simultánea. Por tanto, al utilizar estas llaves, que generalmente los mismos servidores generan, y enviarlas a través de formularios tipo POST se evita que entidades ajenas tomen

[13]Cross-Site Request Forgery

control de funcionalidades administrativas. Así mismo, esto se ve apoyado por la utilización de sesiones y grupos de usuarios, que deben ser utilizados y chequeados en todos los pasos que involucren restricción de permisos.

Otra forma de toma de control se ve reflejada en la utilización de nombres por defecto de archivos subidos al servidor. La razón de esto es que, técnicamente, un usuario puede subir cualquier tipo de archivo a un servidor, que pueden ser tanto benigno como maligno. Un ejemplo de vulnerabilidad que suele darse en estos casos, y principalmente utilizando el lenguaje PHP (razón por la cual no se consideró desde un inicio), corresponde a que el ingreso a cualquier página terminada en `.php` ejecuta inmediatamente su código, lo cual obviamente es detenido si se usan bien los chequeos de privacidad. Otro es el caso con archivos externos, no propios del servidor; si se permite el ingreso sin restricciones, perfectamente se pueden subir archivos del estilo `commandConsole.php` que, si alguien externo adivina su ruta, puede literalmente tomar control de todo el servidor, su información y su forma de operar, ya que no existen mecanismos para restringir el acceso a utilidades de un archivo `.php`, independientemente si está en las carpetas principales del servidor, o en una carpeta de basura.

La forma de solucionar esto consta de dos pasos: el primero es chequear el tipo de archivos que se suben al servidor desde este mismo, tanto comprobando su nombre como su *metadata*, a fin de evitar archivos que a propósito cambiaron su tipo para poder ser subidos; el segundo paso corresponde a nunca guardar un archivo con su nombre original, sino que aplicarle alguna función de Hash para que en caso de ser malicioso, dificultar su acceso y evitar así ceder el control del servidor.

2.6.2. Inyección de SQL

Corresponde a uno de los problemas más comunes de las aplicaciones web y, que pese a su conocimiento, sigue siendo el que más ocurre. Consiste en aprovecharse de aplicaciones que cargan dinámicamente contenido, como en la URL de ejemplo `http://my-app.com/view?id=5`, que en verdad utilizan esa información de forma **directa** para realizar consultas SQL. En el ejemplo anterior, el `id` puede ser entregado a una función SQL como la del Código 2.2.

```
1 SELECT news FROM NewsTable WHERE new_id=id;
```

Código 2.2: Ejemplo consulta SQL.

Ahora bien, nada impide que un usuario malicioso se cuelgue de esto para intentar inyectar consultas SQL. Siguiendo el ejemplo, bastaría cambiar la URL por `http://my-app.com/view?id='5;SELECT * FROM *'` (asumiendo que la consulta esté correcta) para que al inyectarse se pase a tener el Código 2.3 y la página web muestre información que no debería.

```
1 SELECT news FROM NewsTable WHERE new_id=id; SELECT * FROM *;
```

Código 2.3: Ejemplo inyección consulta SQL.

Es importante denotar que esta es sólo una de las tantas formas de inyectar SQL, tal como mencionan **Huang et al.**[6], pues hay formas de determinar qué tipo de base de datos es y, de esa forma, extraer aún más información utilizando consultas del tipo boolean; dicha información está fuera del alcance de esta memoria.

Las formas de prevenir este problema son dos principalmente: la primera es utilizar librerías que limpien los datos que se utilizarán dentro de una consulta SQL, matando caracteres especiales como `';'` y haciendo que la consulta falle antes de mostrar más información. La segunda corresponde a no utilizar el lenguaje SQL de forma directa y utilizar métodos internos que limpien información incorrecta en caso de que se intente obtener. Por ejemplo `News.objects.get(new_id=id)`, que utiliza objetos para manejar las bases de datos y por tanto no permite la inyección de consultas SQL.

2.7. Usabilidad de interfaces

Un tema ampliamente abordado en aplicaciones web, así como en software de escritorio con interfaces, corresponde a la usabilidad. El término viene de la palabra anglosajona *usability* que literalmente es *facilidad de uso*. Por tanto, se refiere a la facilidad que puede tener un usuario inexperto para utilizar el sistema, lo cual además depende de la frecuencia con que un usuario lo utilizará.

Por ejemplo, un software de escritorio no siempre necesita una excelente usabilidad, porque este podría tener muchas herramientas y formar parte de un estándar (software CAD por ejemplo), entonces, en dicho caso, se evalúa la usabilidad a largo plazo. Por otro lado, cuando se trata de una página web que siempre es visitada por nuevos usuarios, se debe maximizar la usabilidad, pues es parte del núcleo que ayuda a retener usuarios.

Nielsen[10] definió la usabilidad bajo cinco componentes de calidad:

- **Learnability** o habilidad de aprender. Se define como qué tan fácil es para un usuario realizar tareas básicas la primera vez que interactúan con el diseño.
- **Efficiency** o eficiencia. Una vez los usuarios han aprendido el diseño, qué tan rápido pueden realizar tareas.
- **Memorability** o capacidad de memorizar. Si un usuario deja de utilizar la plataforma por un periodo de tiempo y luego vuelve a utilizarla, qué tan difícil es restablecer su habilidad con esta.
- **Errors** o errores. Cuántos errores un usuario comete, qué tan severos son y qué tan rápida es la respuesta ante ellos.
- **Satisfaction** o satisfacción. Qué tan placentero es usar el diseño.

Si bien estos puntos no son absolutos, en general los estudios convergen en cualidades similares. En el presente proyecto se prefirió seguir el enfoque de **Palmer**[11], quien realizó un estudio estadístico acerca de la usabilidad en las aplicaciones web y llegó a los siguientes cinco factores que determinar el éxito de un sitio:

- Tiempo de descarga.
- Navegación y Organización.
- Interactividad.
- Capacidad de respuesta.
- Información y Contenido.

Si bien a simple vista ambos punteos se parecen en cuanto a enfoque, el último es más abordable desde el punto de vista de un desarrollador, mientras el acercamiento de **Nielsen** va más enfocado a un diseñador. En consecuencia, los principios esperados en el proyecto para generar atracción de visitantes y usuarios recurrentes de la aplicación como lo son Curadores, son los siguientes:

- La aplicación debe ser de rápido acceso y liviana para reducir el tiempo de descarga, salvo se desee explícitamente ver contenido que agregue tiempo a la descarga.
- La navegación debe ser clara y explícita, para esto utilizar menús auto explicativos y, en caso de existir jerarquías, que sean de clara transición.
- La interacción humano-computador debe ser clara, y a cada acción le debe corresponder una reacción. Es decir, si se realiza un click, este debe realizar algo visible y no debe trabajar en silencio.
- Debe existir un canal de comunicación entre usuarios y encargados de la aplicación, a fin de generar feedback y retro alimentar la aplicación.
- De mostrarse información, debe existir precaución en torno a la cantidad de información mostrada y la variedad de ésta. Por tanto, es ideal la creación de filtros y/o categorías para evitar grandes flujos de información y disminuir su atractivo visual al usuario.

Capítulo 3

Especificaciones del Problema

Considerando lo presentado en el capítulo previo, es necesario adentrarse más en cuál es el problema y en qué partes se puede dividir a fin de abordarlo con mayor precisión. Por ello, en el presente capítulo se explicará en mayor detalle el problema y su relevancia, para luego aplicar ingeniería de software y dividirlo en requisitos específicos, los cuales llevarán a lograr definir con claridad los criterios de aceptación del proyecto.

3.1. Problema a resolver

A través de la existencia de PRASEDEC se han formulado múltiples proyectos en torno a la generación de museos virtuales, y la administración de estos por gente especializada. En un principio, la presente memoria tenía como fin brindar una plataforma web para permitir tanto la gestión, creación y visualización de museos virtuales, tal y como postuló **Biella et al.**[3] en 2010. Sin embargo, reconociendo el avance de las tecnologías web durante los últimos años y la capacidad de personalización que cada institución busca, la idea apuntaba a la generación de un framework más que a una simple plataforma web.

Dicho lo anterior, la presente memoria tenía como metas principales la creación de un conjunto de herramientas para la generación de museos virtuales, la confección de herramientas administrativas para manejar exposiciones a visitantes, y, finalmente, la visualización de estas. No obstante, y siguiendo la misma idea de avance de tecnologías web, durante el presente año los motores para crear videojuegos dieron el salto definitivo hacia permitir la creación de aplicaciones de alta calidad tanto para escritorio como para web y, con esto, PRASEDEC encargó a un grupo distinto la confección de museos utilizando la tecnología **Unity** y, por tanto, retirándola de prioridad de la presente memoria. Lo que reformuló el problema de fondo a ser resuelto.

En consecuencia, la presente memoria tiene como finalidad resolver los siguientes problemas: La creación de una plataforma web reutilizable y mantenible que permita mostrar museos virtuales de distintas índoles, es decir, no acotarla a un única visualización, como lo sería utilizar Javascript, applets, Unity, etc. Así mismo permitir la administración de dichos

museos, o exposiciones, ayudar en el manejo de la interacción museo-visitante, y facilitar el trabajo colaborativo entre los miembros de una misma organización; todo esto sin perder en el camino la seguridad de la plataforma, al tener que almacenar material que podría ser considerado sensible, como resulta ser la organización de exhibiciones y todo su material antes de generar exhibiciones en sí, o bien, previo a mostrarlas abiertamente al público. Finalmente, también se busca ayudar a la preservación de estas exhibiciones, y que no se pierdan o filtren al tener que cambiar equipos de forma física.

3.2. Relevancia

Tal y como se indicó en la **Introducción**, el actual proyecto de museos inteligentes está ligado directamente con la digitalización y difusión de elementos culturales de Armenia, particularmente los **Khachkar**, que son considerados *patrimonio cultural inmaterial de la Humanidad*. Por ello, se está relacionando con el gobierno, universidades de dicho país y el Instituto Smithsoniano con el fin de ayudar a preservar este patrimonio.

En dicho contexto, la relevancia de la presente memoria está en poder aportar a la causa, generando una plataforma que se pueda utilizar para almacenar, distribuir y visualizar las piezas patrimoniales independientemente del país físico en que uno se encuentre y, de paso, facilitando la recopilación de archivos digitales utilizados en este proyecto, los cuales son producidos en Armenia pero utilizados en otros países como lo son Alemania y Chile.

Sin ánimos de acotar el uso del Framework, también se busca que este sea de código abierto, a fin de que cualquier institución pueda utilizarlo y, de dicha forma, aportar a la generación y preservación de cultura sin incurrir en grandes gastos en el área de tecnologías de la información, permitiendo así que tanto grandes como pequeñas instituciones puedan realizar exhibiciones en internet a la población y se pueda difundir el conocimiento, la historia y el patrimonio.

3.3. Requisitos

Si bien se decidió abordar el proyecto utilizando *metodologías ágiles*, aún es relevante analizar de forma directa los requisitos del pŕoyecto, para usarlos de base en el proceso de identificar features a implementar, y dictaminar prioridades en estas.

En consecuencia, a continuación se dividirá el problema en tablas correspondientes a **requisitos de usuario** y **requisitos de software**.

Los **requisitos de usuario** corresponden a declaraciones en lenguaje natural de características que el usuario, cliente o contra parte desea tener en la aplicación, teniendo así un **título**, una **descripción** de lo que se entiende por dicho requerimiento, un **tipo** que puede ser *funcional*, *restricción* y *calidad* para poder clasificar los requisitos (recordar que la contra parte no tiene por qué ser un experto en ciencias de la computación), una **estabilidad** que

corresponde a la flexibilidad con respecto a dicha necesidad, pudiendo esta ir cambiando en el tiempo. Finalmente, tiene un **tipo de usuario asociado**, los cuales se describirán en breve.

Por otro lado, los **requisitos de software** corresponden a la parte técnica y están dados por el diseñador del proyecto, el cual en este caso corresponde al escritor de la presente memoria. Dichos requisitos son más específicos que los de usuario pero mantienen el lenguaje natural, pues su idea es que alguien sin conocimientos en ciencias de la computación logre entender qué se hará. A diferencia de los requisitos de usuario, los tipos de requisitos son más variados, ya que tienen como fin el poder establecer características específicas a implementar y ayudar en la documentación. Un ejemplo de esto es un requisito que se titule **'Utilizar script en C'** que exista con los tipos **mantenibilidad** e **interoperabilidad**. En el primer caso la idea tras dicho script es que el programa pueda ser mantenido en el tiempo (en el lugar destino del software puede que se trabaje únicamente con C y ese script sea crucial), mientras que con el segundo tipo se busca que el programa funcione en distintos entornos computacionales (puede ser una librería que, al estar escrita en C, funcione con características específicas dependiendo de cada sistema operativo o bien ser independiente a este). De estos últimos requisitos se desprenderán los feature a utilizar en las metodologías ágiles.

Para asegurarnos de que cada requisito de usuario sea abordado, se incluirá la **matriz de trazado**, la cual muestra de forma explícita el cruce entre ambos tipos de requisitos con el fin de asegurar una buena práctica de ingeniería de software, tales como evitar que un único requisito de usuario posea más de tres requisitos de software.

Para los requisitos se definieron los siguientes **tipos de usuario asociado** de la aplicación:

- **TU01** Visitante.
- **TU02** Curador - Opiniones.
- **TU03** Curador - Recursos.
- **TU04** Curador - Exhibits.
- **TU05** Curador - Agenda.
- **TU06** Personal TI.

Los usuarios de estilo **curador** están divididos en categorías dependiendo de las funcionalidades que desempeñarán, para evitar tener un único y general tipo de usuario.

3.3.1. Requisitos de usuario

Tabla 3.1: **RU001: Extensibilidad**

Descripción	El software a ser desarrollado debe permitir modificaciones tanto de apariencia como funcionales, además de ser mantenible
Tipo	Calidad
Estabilidad	Transable
Tipo de usuario asociado	TU06

Tabla 3.2: **RU002: Funcionalidad basal**

Descripción	El software debe tener un mínimo funcional y no ser una maqueta, además debe funcionar sobre computadores de escritorio
Tipo	Restricción
Estabilidad	Intransable
Tipo de usuario asociado	TU06

Tabla 3.3: **RU003: Compatibilidad**

Descripción	El sitio web que genere debe poder ser visualizado en distintos navegadores web
Tipo	Funcional
Estabilidad	Transable
Tipo de usuario asociado	TU01, TU2, TU03, TU04, TU05, TU06

Tabla 3.4: **RU004: Tipos contenido**

Descripción	Debe permitir que se utilicen distintos tipos de exhibits
Tipo	Funcional
Estabilidad	Intransable
Tipo de usuario asociado	TU01, TU04

Tabla 3.5: **RU005: Grupos usuarios**

Descripción	Deben existir distintos grupos de usuario con acciones limitadas
Tipo	Funcional
Estabilidad	Intransable
Tipo de usuario asociado	TU06

Tabla 3.6: **RU006: Datos usuarios**

Descripción	Los usuarios registrados deben poder modificar su información de contacto
Tipo	Funcional
Estabilidad	Transable
Tipo de usuario asociado	TU02, TU03, TU04, TU05

Tabla 3.7: **RU007: Información exhibiciones**

Descripción	Todo el personal administrativo debe poder ver la información de las exhibiciones disponibles
Tipo	Funcional
Estabilidad	Intransable
Tipo de usuario asociado	TU02, TU03, TU04, TU05

Tabla 3.8: **RU008: Sencillez**

Descripción	Las exhibiciones deben ser de fácil acceso para los visitantes
Tipo	Funcional
Estabilidad	Intransable
Tipo de usuario asociado	TU01

Tabla 3.9: **RU009: Visualizar exhibiciones**

Descripción	Un visitante debe poder ver cada elemento de la exhibición desde el mismo sitio
Tipo	Restricción
Estabilidad	Transable
Tipo de usuario asociado	TU01

Tabla 3.10: **RU010: Retroalimentación visitantes**

Descripción	Un visitante debe poder calificar una exhibición y opinar sobre esta
Tipo	Funcional
Estabilidad	Intransable
Tipo de usuario asociado	TU01

Tabla 3.11: **RU011: Manejar opiniones**

Descripción	Se deben poder administrar y filtrar las opiniones de los visitantes, además de poder contactar a un visitante de ser necesario
Tipo	Funcional
Estabilidad	Intransable
Tipo de usuario asociado	TU02

Tabla 3.12: **RU012: Administrar recursos**

Descripción	Se debe poder subir, eliminar y bajar recursos
Tipo	Funcional
Estabilidad	Transable
Tipo de usuario asociado	TU03

Tabla 3.13: **RU013: Facilidad de ver recursos**

Descripción	Un curador debe poder revisar el contenido de los recursos y además poder filtrarlos
Tipo	Funcional
Estabilidad	Transable
Tipo de usuario asociado	TU03

Tabla 3.14: **RU014: Administrar exhibits**

Descripción	Se debe poder subir. eliminar y previsualizar los exhibits subidos al sistema
Tipo	Funcional
Estabilidad	Intransable
Tipo de usuario asociado	TU04

Tabla 3.15: **RU015: Información de uso de exhibits**

Descripción	Se debe poder ver la cantidad de visitas de un elemento en exhibición en particular, además de su popularidad y si se encuentra en uso
Tipo	Funcional
Estabilidad	Intransable
Tipo de usuario asociado	TU04

Tabla 3.16: **RU016: Manejo agenda**

Descripción	Deben poder crearse, modificarse y eliminarse exhibiciones, las cuales deben tener un tiempo de inicio y término
Tipo	Funcional
Estabilidad	Intransable
Tipo de usuario asociado	TU05

Tabla 3.17: **RU017: Modificación agenda**

Descripción	Cada exhibición debe poder editarse, cambiando los elementos a mostrar, su nombre o las fechas de este
Tipo	Funcional
Estabilidad	Transable
Tipo de usuario asociado	TU05

Tabla 3.18: **RU018: Accesibilidad exhibiciones**

Descripción	Una exhibición debe poder ocultarse de los visitantes independientemente de las fechas de esta
Tipo	Funcional
Estabilidad	Intransable
Tipo de usuario asociado	TU05

Tabla 3.19: **RU019: Facilidad de ejecución**

Descripción	El software debe poder ejecutarse fácilmente
Tipo	Restricción
Estabilidad	Transable
Tipo de usuario asociado	TU06

Tabla 3.20: **RU020: Independencia de sistema operativo**

Descripción	El software debe poder ejecutarse independientemente del sistema operativo de la máquina
Tipo	Restricción
Estabilidad	Transable
Tipo de usuario asociado	TU06

3.3.2. Requisitos de software

Tabla 3.21: **RS001: Buenas prácticas**

Descripción	El software seguirá patrones de programación bien descritos, explícitos y siguiendo la arquitectura designada
Tipo	Mantenibilidad
Estabilidad	Intransable
RU asociado	RU001
Tipo de usuario asociado	TU06

Tabla 3.22: **RS002: Documentación**

Descripción	Se documentará toda toma de decisiones en diseño, implementación y validación
Tipo	Documentación
Estabilidad	Transable
RU asociado	RU001
Tipo de usuario asociado	TU06

Tabla 3.23: **RS003: Vistas con datos reales**

Descripción	Salvo vistas propiamente estáticas, cada vista estará asociada a un modelo de base de datos
Tipo	Funcional
Estabilidad	Intransable
RU asociado	RU002
Tipo de usuario asociado	TU06

Tabla 3.24: **RS004: Vistas optimizadas para computadores de escritorio**

Descripción	Se debe dar prioridad a una mejor visualización de los contenidos de la aplicación desde dimensiones de pantallas de computador de escritorio
Tipo	Usabilidad
Estabilidad	Transable
RU asociado	RU002
Tipo de usuario asociado	TU01, TU02, TU03, TU04, TU05, TU06

Tabla 3.25: **RS005: Independencia de navegador**

Descripción	Las vistas mostradas deben funcionar independientemente del navegador web que se utilice
Tipo	Interoperabilidad
Estabilidad	Transable
RU asociado	RU003
Tipo de usuario asociado	TU01, TU02, TU03, TU04, TU05, TU06

Tabla 3.26: **RS006: Soporte múltiples exhibits**

Descripción	Se debe poder tener distintos tipos de exhibits con los cuales trabajar y/o visualizar
Tipo	Funcional
Estabilidad	Intransable
RU asociado	RU004
Tipo de usuario asociado	TU01, TU04

Tabla 3.27: **RS007: Extensibilidad exhibits**

Descripción	Cada modelo de exhibit debe ser fácilmente implementable e incorporable a la base de datos
Tipo	Mantenibilidad
Estabilidad	Intransable
RU asociado	RU004
Tipo de usuario asociado	TU06

Tabla 3.28: **RS008: Sistema de usuarios**

Descripción	El sistema debe permitir la existencia de usuarios y un login o inicio de sesión respectivo
Tipo	Funcional
Estabilidad	Intransable
RU asociado	RU005
Tipo de usuario asociado	TU02, TU03, TU04, TU05, TU06

Tabla 3.29: **RS009: Grupos de usuarios**

Descripción	El sistema debe permitir tener permisos de usuarios para el acceso a determinadas zonas
Tipo	Interacción
Estabilidad	Intransable
RU asociado	RU005
Tipo de usuario asociado	TU06

Tabla 3.30: **RS010: Modificación datos personales**

Descripción	Cada usuario debe tener acceso a una vista específica donde cambiar sus datos personales
Tipo	Funcional
Estabilidad	Transable
RU asociado	RU006
Tipo de usuario asociado	TU02, TU03, TU04, TU05, TU06

Tabla 3.31: **RS011: Visualización de agenda**

Descripción	Todo personal administrativo del framework debe poder visualizar la agenda como página de inicio
Tipo	Funcional
Estabilidad	Intransable
RU asociado	RU007
Tipo de usuario asociado	TU02, TU03, TU04, TU05, TU06

Tabla 3.32: **RS012: Usabilidad en vista visitante**

Descripción	El visitante debe poder acceder de forma fácil e intuitiva a las exhibiciones
Tipo	Usabilidad
Estabilidad	Intransable
RU asociado	RU008
Tipo de usuario asociado	TU01

Tabla 3.33: **RS013: Aplicación maneja las visualizaciones**

Descripción	La aplicación es la encargada de servir y visualizar el contenido de cada exhibición sin necesidad de descargar elementos extras y/o tener que el visitante ejecutarlos
Tipo	Usabilidad
Estabilidad	Transable
RU asociado	RU009
Tipo de usuario asociado	TU01

Tabla 3.34: **RS014: Sistema de opiniones**

Descripción	Debe existir tanto un modelo como una vista para que cada visitante pueda dejar su opinión y calificación sobre las exhibiciones
Tipo	Funcional
Estabilidad	Intransable
RU asociado	RU010, RU011
Tipo de usuario asociado	TU01, TU02

Tabla 3.35: **RS015: Validación de opiniones**

Descripción	Con el fin de evitar ataques de spam de parte de bots, debe implementarse un sistema de validación de opiniones que requiera utilizar un correo válido y acceder a una página de validación
Tipo	Funcional
Estabilidad	Intransable
RU asociado	RU010
Tipo de usuario asociado	TU01, TU02, TU06

Tabla 3.36: **RS016: Manejo de opiniones**

Descripción	Debe existir una vista que permita manipular los modelos de opiniones, permitiendo así cambiar su estado entre válido e inválido, eliminarlas y finalmente filtrarlas de acuerdo a su estado
Tipo	Funcional
Estabilidad	Intransable
RU asociado	RU011
Tipo de usuario asociado	TU02

Tabla 3.37: **RS017: Manejo y extensibilidad recursos**

Descripción	Las entidades recurso deben poder ser extensibles para agregar nuevos tipos de contenido de forma fácil y rápida
Tipo	Mantenibilidad
Estabilidad	Transable
RU asociado	RU012
Tipo de usuario asociado	TU06

Tabla 3.38: **RS018: Vista administración recursos**

Descripción	Deben existir vistas que permitan subir, eliminar y bajar recursos
Tipo	Funcional
Estabilidad	Transable
RU asociado	RU012
Tipo de usuario asociado	TU03

Tabla 3.39: **RS019: Validación recursos y exhibits**

Descripción	Los recursos subidos al servidor deben ser validados para evitar archivos maliciosos, además de ser sanitizados
Tipo	Operacional
Estabilidad	Intransable
RU asociado	RU012, RU014
Tipo de usuario asociado	TU03, TU04, TU06

Tabla 3.40: **RS020: Filtrado recursos**

Descripción	La vista de recursos debe permitir filtro por tipo y poder revisar cada uno de estos
Tipo	Funcional
Estabilidad	Intransable
RU asociado	RU013
Tipo de usuario asociado	TU03

Tabla 3.41: **RS021: Vista administración exhibits**

Descripción	Deben existir vistas que permitan subir, eliminar y previsualizar los exhibits
Tipo	Funcional
Estabilidad	Intransable
RU asociado	RU014
Tipo de usuario asociado	TU04

Tabla 3.42: **RS022: Información de uso de exhibits**

Descripción	Debe existir una vista donde aparezcan todos los exhibits, y además la información entregada por los visitantes, tales como la cantidad de visitas, el rating de este y si se encuentra en uso
Tipo	Funcional
Estabilidad	Intransable
RU asociado	RU015
Tipo de usuario asociado	TU04

Tabla 3.43: **RS023: Administración agenda**

Descripción	Sólo los usuarios con permisos deben poder crear, modificar o eliminar modelos del tipo exhibición
Tipo	Funcional
Estabilidad	Intransable
RU asociado	RU016
Tipo de usuario asociado	TU05

Tabla 3.44: **RS024: Múltiples exhibit por exhibición**

Descripción	Cada exhibición debe poder alojar múltiples exhibits de distintos tipos
Tipo	Operacional
Estabilidad	Intransable
RU asociado	RU016
Tipo de usuario asociado	TU05, TU06

Tabla 3.45: **RS025: Edición total de exhibiciones**

Descripción	Una exhibición debe permitir modificar toda su información
Tipo	Funcional
Estabilidad	Transable
RU asociado	RU017
Tipo de usuario asociado	TU05

Tabla 3.46: **RS026: Visibilidad exhibiciones**

Descripción	Como parte de las características de las exhibiciones, estas deben tener un estado de visibilidad con respecto a los visitantes
Tipo	Funcional
Estabilidad	Intransable
RU asociado	RU018
Tipo de usuario asociado	TU05

Tabla 3.47: **RS027: Scripts de ejecución del framework**

Descripción	El Framework debe incluir scripts que permitan facilitar el proceso de deployment de la aplicación
Tipo	Portabilidad
Estabilidad	Transable
RU asociado	RU019
Tipo de usuario asociado	TU06

Tabla 3.48: **RS028: Independencia de sistema operativo**

Descripción	El Framework debe ser independiente del sistema operativo y por tanto debe poder hacer deployment en distintas plataformas
Tipo	Portabilidad
Estabilidad	Transable
RU asociado	RU019, RU020
Tipo de usuario asociado	TU06

3.3.3. Matriz de trazado

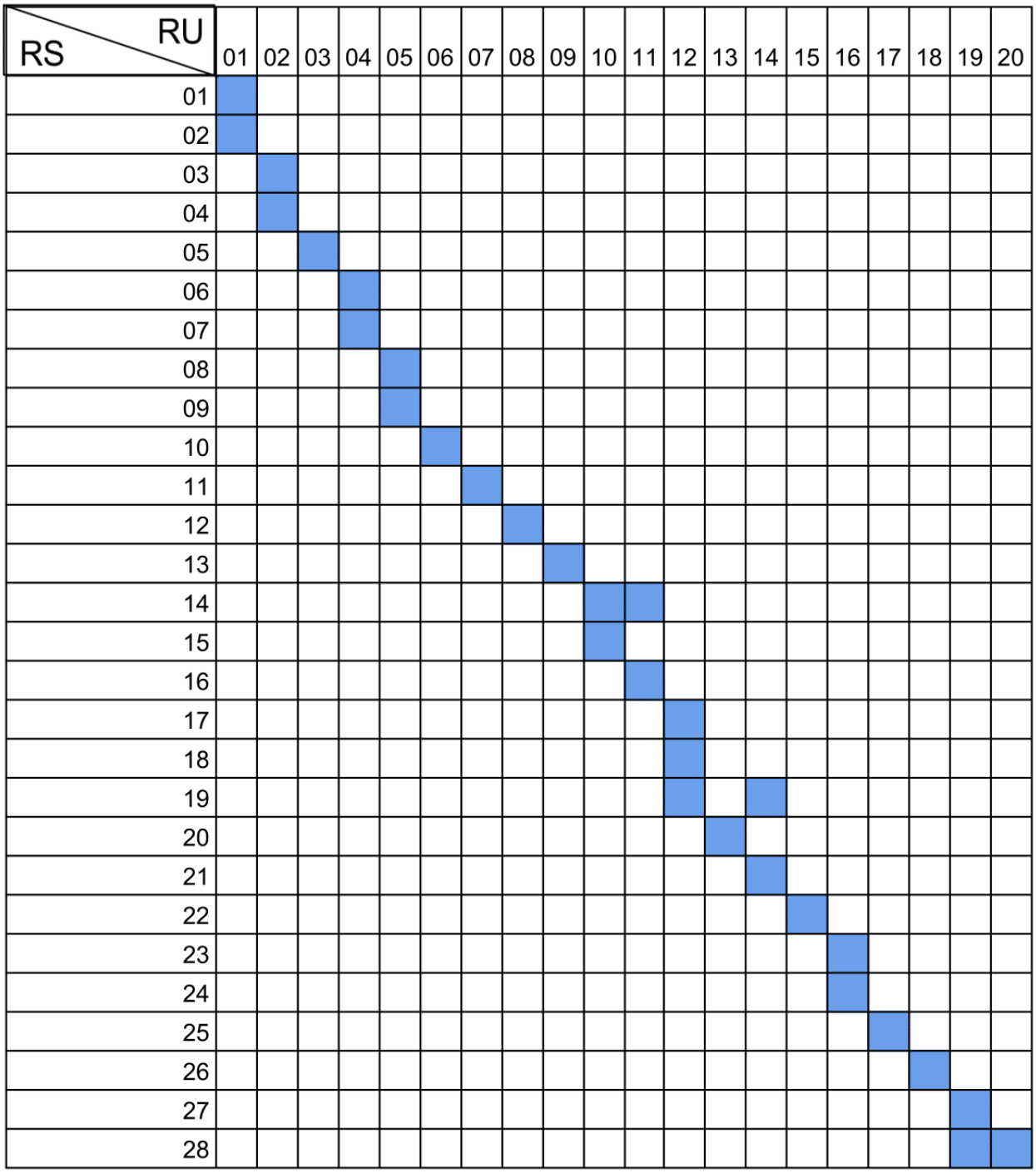

Figura 3.1: Matriz de trazado de requisitos de usuario vs requisitos de software.

3.4. Criterios de aceptación

La aceptación de la presente memoria recae fundamentalmente en tres puntos que se explicarán a continuación.

El Framework realizado debe ser funcional y fácilmente extensible, por tanto, se deberá realizar una prueba de extensibilidad cambiando así su apariencia visual así como el tipo de contenido que podrá visualizar, teniendo como mínimo la visualización de Unity y agregando otros, como PDF o vídeo.

Retomando la funcionalidad, ésta debe ser probada en distintos sistemas operativos y poder mostrarse fuera del área local de un computador, por tanto, es necesario realizar pruebas de deployment y tener acceso remoto a la aplicación utilizando un dispositivo conectado a una red distinta de Internet.

El trabajo debe ser de código abierto para que pueda fomentarse tanto su uso como extensión; de forma simultánea, debe incluir guidelines para poder ayudar a la creación de nuevos museos virtuales y, en consecuencia, lograr un aporte al proyecto de Virtual Museums de PRASEDEC.

Dicho todo esto, tanto el diseño de la solución como la validación de ésta se centrarán tanto en resolver el problema, guiándose por los requisitos planteados, así como lograr los criterios de aceptación utilizándolos como base misma de las pruebas a realizar en la validación.

Capítulo 4

Descripción de la Solución

Utilizando los conocimientos del **Marco Teórico** y el desglose de requisitos de **Especificaciones del Problema**, en el presente capítulo se describe con detalle todo el proceso de toma de decisiones que conllevó el diseño de la solución, incluyendo: la arquitectura lógica del software, el diseño completo de la base de datos, la lógica tras los procesos que realizará el framework, cómo funcionan las interfaces visuales y lo relativo a la seguridad informática de la plataforma.

4.1. Arquitectura lógica

Tal y como se anunció dentro del **Marco Teórico**, el presente proyecto contemplaría la utilización de la **Arquitectura de tres capas** así como también la **Arquitectura MVC**. Para poder visualizar esto, es importante primero explicitar cómo funciona un proyecto de Django.

Un proyecto está compuesto por tres carpetas principales llamadas **static**, **templates** y una con el nombre del proyecto, en este caso **VirtualMuseumsFramework**. Las dos primeras corresponden a visualizaciones y material complementario o estático (principalmente CSS, Javascript y recursos como imágenes). Por otro lado, la carpeta con el nombre del proyecto corresponde al núcleo de la aplicación, donde se encuentran todas las configuraciones (entre ellas de la base de datos) y el **URL Dispatcher**, que corresponde a la entidad encargada de realizar pareos entre URL solicitada y vista cargada.

Una versión sencilla de una aplicación corresponde a que el URL Dispatcher indique páginas estáticas de los template y los muestre al usuario. Sin embargo, eso carece de dinamismo, y no es útil para lo que se busca, lo cual es solucionado creando **Aplicaciones de Django**. Estas aplicaciones son una o más nuevas carpetas (el nombre de cada una responde a la aplicación a ser añadida) que en su interior contienen los archivos `models.py`, `views.py` y `urls.py`, los cuales, como sus nombres hacen mención, corresponden a los **Modelos** con que trabajará la aplicación (ver sección **Diseño base de datos**), las **Vistas** a ser mostradas (y la lógica que llevan por detrás), y finalmente otro URL Dispatcher, con fin de establecer

jerarquías.

Ahora, en lo referente a las arquitecturas, en la Figura 4.1 se puede ver la distribución física del presente proyecto, donde los directorios **static** y **templates** corresponden a los elementos que un usuario podrá visualizar, y por tanto son los comprendidos en la **capa de presentación**; como la referencia a estos elementos se hace mediante *paths* relativos y/o absolutos, es posible separarlas de forma física y servirlas desde otro servidor, sin embargo, por temas de completitud del software se trabajará sin separaciones. Los directorios **Apps** corresponden a las aplicaciones del Framework, las cuales fueron separadas estratégicamente dependiendo del grupo de usuarios al cual está enfocado, para facilitar la modularidad, y justamente debido a que son la parte central de la aplicación corresponden a la **capa de negocio** en conjunto con el directorio *VirtualMuseumsFramework*. La **capa de datos** siempre se considera externa y el único vínculo directo se encuentra en `VirtualMuseumsFramework/settings.py`, pudiendo así ser totalmente independiente del Framework. En el proyecto, por configuración inicial, se utiliza la base de datos **SQLite**, que corresponde a un archivo físico que se encuentra en el mismo directorio del proyecto.

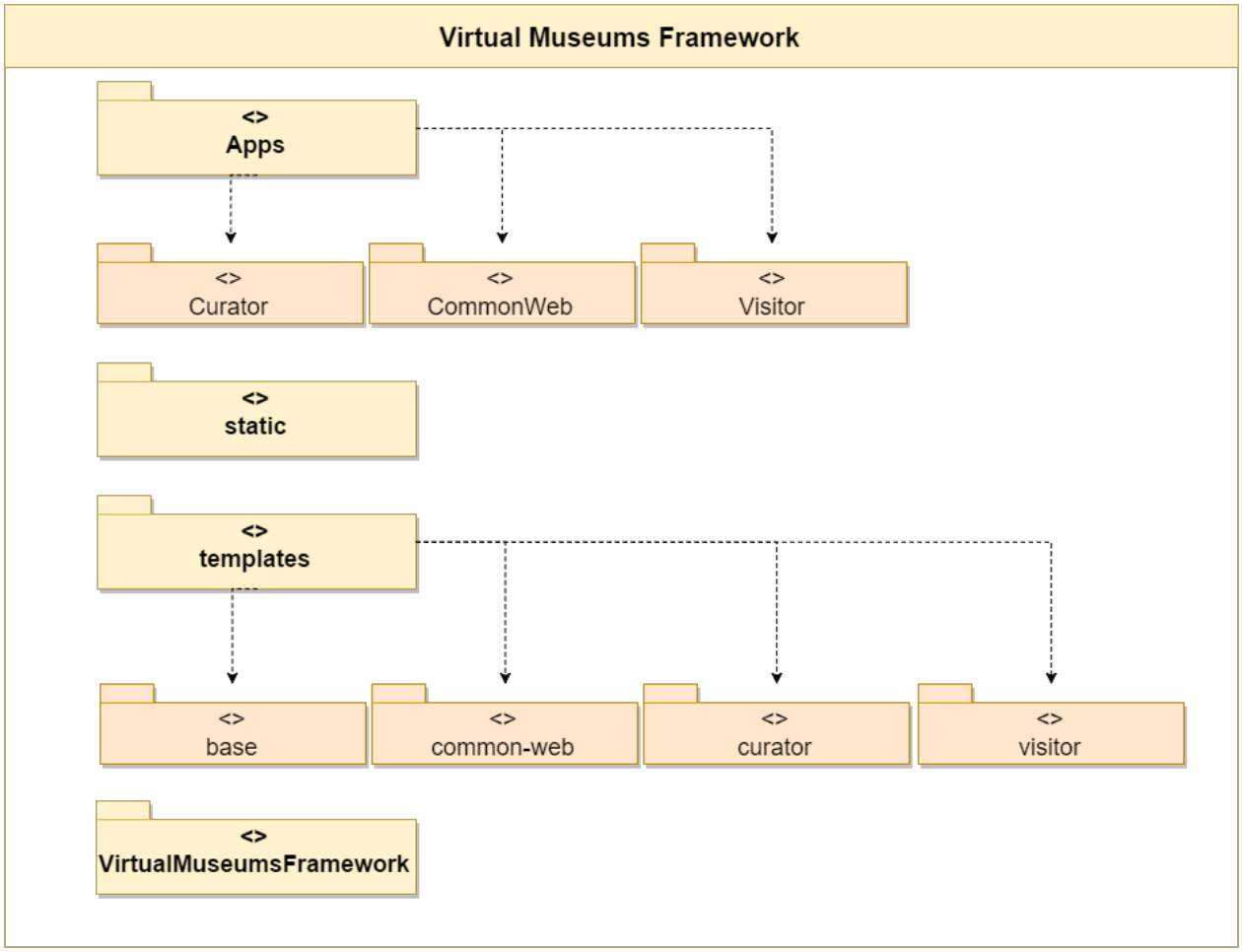

Figura 4.1: Estructura física del proyecto.

La arquitectura **MVC**, por otro lado, se emplea fundamentalmente dentro de cada aplicación de **Apps**, utilizando `models.py`, `views.py` y los `templates` (que fueron separados en

directorios análogos a las aplicaciones). Sin embargo, es importante denotar que en Django el MVC funciona en una estructura llamada **MTV** según sus desarrolladores, que corresponde a **Model-Template-View**. Esta variación del MVC nace a raíz de que el controlador de la aplicación en sí está distribuido entre la lógica de Django para manejar request y vincular modelos con la base de datos y en cada clase del archivo `views.py`; esto pues, tal y como se explicará en las sub-sección y sección **Procesamiento de requests** y **Diseño interfaces**, cada vista está asociada a una clase, la cual procesa lógica antes de entregar una vista mediante un template. Un ejemplo de esto se puede ver en el Código 4.1, donde la clase `SchedulingView` entrega la vista `curator/scheduling.html`, pero además recopila la información de las exhibiciones mediante un método interno, y las incorpora al renderizado de la vista.

```
1 class SchedulingView(TemplateView):
2     template_name = 'curator/scheduling.html'
3 
4     @method_decorator(login_required(login_url='/auth/login'))
5     def get(self, request, *a, **ka):
6         exhibitions = get_exhibitions()
7         return render(request, self.template_name, exhibitions)
```

Código 4.1: Ejemplo de una clase vista.

4.2. Diseño base de datos

Siguiendo la premisa principal del proyecto, la base de datos debe ser extensible, mantenible y transparente, además de no acotarse a un engine en particular de bases de datos, ya que, al tratarse de un framework, la idea es que cada desarrollador que desee emplear el proyecto pueda utilizar el engine que le resulte más cómodo o bien, para el cual disponga una licencia.

Dicho esto, se decidió no utilizar directamente consultas SQL para evitar problemas de estructuras, sino que usar el sistema provisto por Django para manejar entidades.

4.2.1. Modelos Django

En el framework Django, con el fin de poder utilizar distintas bases de datos, el manejo intrínseco de estas las hace de mismo de manera interna y sólo entregándole al desarrollador la clase `django.db.models.Model` para que trabaje.

Lo interesante de dicha modalidad es que cada clase, que extiende la clase Model utilizando herencia, es un mapeo uno a uno de una tabla de base de datos, donde cada atributo de la clase corresponde a un campo de la tabla. Un ejemplo de esto es el Código 4.2 que se mapea directamente a la entidad de la Figura 4.2, teniendo especial cuidado de que, al momento de instanciar cada atributo, se le debe especificar a qué tipo corresponderá en la base de datos,

además de otros parámetros opcionales, como el largo máximo, la posibilidad de ser un dato en blanco o, en casos, como la fecha, de que se agregue automáticamente.

```python
from django.db import models

class ExternalFile(models.Model):
    title = models.CharField(max_length=30, blank=False)
    description = models.CharField(max_length=255, blank=False)
    uploaded_at = models.DateTimeField(auto_now_add=True)
```

Código 4.2: Ejemplo implementación de Django Model.

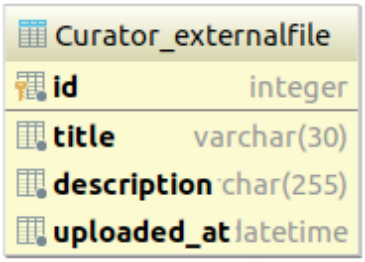

Figura 4.2: Ejemplo entidad base de datos.

Teniendo ya la base de cómo se representan las bases de datos utilizando los modelos de Django, es importante mostrar la metodología para realizar consultas a la base de datos. En el Código 4.3 se puede observar que para realizar una query, o consulta, sobre una determinada tabla, se realiza una invocación a la clase equivalente y, mediante métodos estáticos, se le solicitan los objetos deseados, pudiendo ser uno en particular o un conjunto de estos. Se debe tener especial cuidado entre las llamadas `get()`, `all()` y `filter()`, pues las dos últimas pueden entregar conjuntos vacíos si es que no hay datos que satisfagan las consultas, sin embargo `get()` **puede producir errores y lanzar excepciones**, pues se realiza una consulta específica. Esto se puede atrapar utilizando la metodología de `try - except`, o bien utilizando un filter y chequeando si la lista de retorno posee algún elemento.

```python
music = ExternalMusic.objects.get(id=_id) # Obtener un recurso del id específico.
exhibits = Exhibit.objects.all() # Obtener todas las Exhibits.
opinions = Opinion.objects.filter(validated=True) # Obtener todas las opiniones validadas.
opinions = opinions.exclude(status=False) # Excluir elementos de un conjunto de resultados.
```

Código 4.3: Ejemplo consultas Django.

De manera intuitiva, el modificar una fila de una tabla corresponde a obtener el elemento, por ejemplo utilizando `model = Model.objects.get(id=specific_id)`, y luego modificar el atributo deseado del objeto de la forma `model.attribute = new_attribute`, para finalmente guardar el cambio utilizando `model.save()` (de forma análoga se puede eliminar un elemento en particular utilizando `model.delete()`). De esta forma, todo el proceso de trabajar con la base de datos es ajeno a la implementación misma de esta y de sus consultas.

4.2.2. Herencia

Como toda tabla de una base de datos es una clase de Python, de forma directa heredando la clase `django.db.models.Model`, surge la pregunta de cómo funciona la herencia de una clase que hereda `Model`.

Siguiendo el pensamiento de herencia clásica, se pensaría que, en consecuencia, la nueva clase al heredar `Model` se crearía una tabla de nombre equivalente con los atributos propios y los del padre. Sin embargo, a través a investigación empírica se llegó a una solución más razonable desde el punto de vista de bases de datos.

Lo que realmente ocurre, y que es posible observar en la Figura 4.3 es que se intenta seguir la idea de las **Formas normales** de evitar atributos duplicados. Por tanto, la nueva tabla `Curator_externalimage` en verdad sólo agrega los atributos que no tiene su padre `Curator_externalfile`, y crea la llave foránea `externalfile_ptr_id` de la tabla padre que almacena los atributos correspondientes a este.

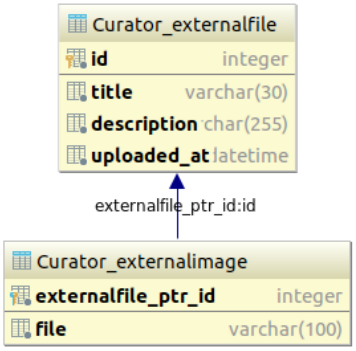

Figura 4.3: Ejemplo de herencia doble en la base de datos.

4.2.3. Recursos

Parte fundamental del proyecto corresponde a almacenar archivos subidos por el equipo de Curadores al servidor mediante el Framework. No obstante, es necesario siempre hacer hincapié en la **seguridad informática**, tanto de la información presente en las bases de datos como de los servidores mismos. Por ello, cada archivo que se sube a un servidor debe almacenarse con un nombre distinto al original, e idealmente tras un hash (por ejemplo el archivo `'ejemplo.txt'` será almacenado como `'a6c05bd5d579650bdb5f2088a3b8ea444529a7.txt'`. La razón de esto es por vulnerabilidades tales como permitir subir archivos *backdoor*, que una vez identificado su ruta de acceso se puede utilizar y vulnerar la seguridad, tal y como se indicó en el **Marco Teórico**.

Sin embargo, para el usuario que subió dicho archivo, debe poder seguir siendo visible bajo alguna etiqueta o título que permita su fácil manipulación pero que no revele el nombre hash del archivo. Por tanto cada entidad de la base de datos que represente un archivo alojado

debe mantener un título y un path al archivo original, sumado a otras variables útiles como la descripción y la fecha de subida.

Sumado a lo anterior, es deseable tener distintos tipos de recursos y realizar la validación correspondiente a cada uno (la cual será mencionada en **Diseño de procesos**). Por ello, tomando base en la **Herencia** de clases, se decidió crear el modelo entidad-relación de la Figura 4.4, teniendo como tipos base para los recursos: Vídeo, archivos de audio, imágenes, modelos 3D y un modelo Template para facilitar la creación de nuevos recursos.

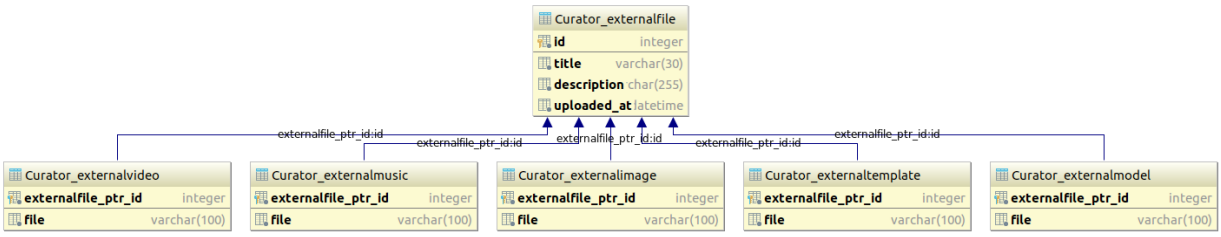

Figura 4.4: Diagrama relacional recursos.

4.2.4. Exhibits

Teniendo como definición de un *Exhibit* la unidad mínima de una exposición, que debe ser auto contenida, a fin de poder visualizarla, es entendible así mismo que no tiene un tipo fijo de archivos, tal y como ocurrió previamente con los **Recursos**. De esto se desprende que sigue el mismo modelo de una clase madre y utilizar herencia para los distintos tipos. No obstante, y adelantando materias propias de la extensibilidad de las vistas que se verá dentro del presente capitulo, cada exhibit sabe su propia ID, pero no puede saber qué tipo de exhibit es realmente para cargar sus contenidos.

Esto último conlleva a un problema si pensamos en extensibilidad de vistas. Si se desea simplemente hacer click en un exhibit y visualizarlo apropiadamente, necesitamos saber a qué tipo corresponde para cargar la vista correcta; la solución simple a este problema corresponde a chequear contra cada tipo de exhibit si el ID corresponde, y luego obtener las referencias a los archivos útiles. Pero si en vez de uno o dos tipos de exhibit se tienen cientos, la carga a la base de datos sería exagerada para la operación mínima que se requiere. Por ello, se prefirió tomar la decisión de utilizar un poco de memoria extra en la base de datos añadiendo la columna `exhibit_type_id` y la tabla `Curator_exibittype`, siguiendo el principio de las **Formas normales**.

Con dicha decisión, cada exhibit sabe qué tipo es de forma inmediata sin tener que realizar N consultas sobre la base de datos, con N igual a la cantidad de tipos de exhibit disponibles; de dicha forma, se obtiene de manera expedita la vista correspondiente.

El diagrama entidad-relación de esta estructuración puede observarse en la Figura 4.5, teniendo como ejemplo de implementación el exhibit para Unity.

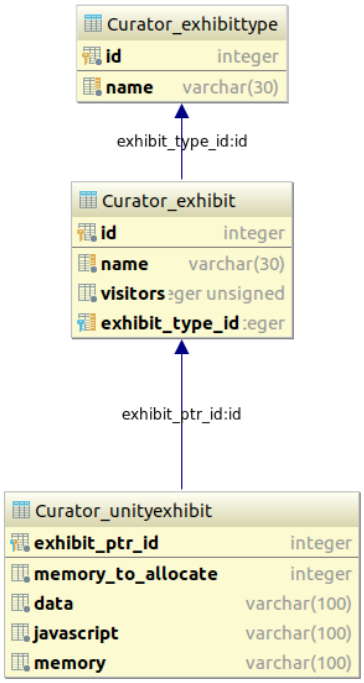

Figura 4.5: Diagrama relacional exhibits.

4.2.5. Opiniones

Como parte de los requisitos de usuario se encontraba que se debía poder incluir opiniones emitidas por parte de los visitantes del museo. En base a un refinamiento de este requisito se llegó a que la mejor opción correspondía a que se pudiera realizar opiniones con respecto a los exhibits en sí, permitiendo así que se pudieran utilizar filtros y evaluar contenidos específicos.

Como se puede ver en la Figura 4.6, una opinión consta de un nombre, el texto propiamente tal, un rating, un correo electrónico y una llave foránea a un exhibit como parte principal de lo que debe ver un visitante. Mientras que internamente también maneja un estado, una llave de hash, una validación y un timeout, todo esto último corresponde a medidas de seguridad para evitar **spam** por parte de bots, uno de los principales problemas que acompleja hoy en día los sitios web con retroalimentación de visitantes.

La idea tras estas medidas es que, una vez emitida una opinión, se genera una llave de hash y se envía un correo de validación al visitante. A partir de dicho momento tiene un determinado tiempo (por defecto 24 horas) para validar la opinión, o esta pasará a considerarse spam, siendo imposible validarla y será eliminada a través de procesos automáticos. La variable `status` es manejada por el equipo de curadores para separar opiniones aceptadas de las rechazadas, con el fin de poder realizar una limpieza manual, o bien, poder contactar a la persona que emitió determinada opinión.

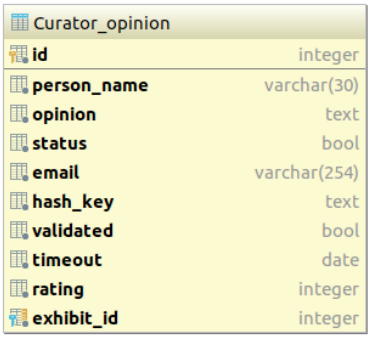

Figura 4.6: Tabla de opiniones.

4.2.6. Exhibitions

Como bien se ha mencionado a través del documento, una de las metas del presente proyecto es poder permitir la visualizaciones de **Exhibits**, que corresponden a piezas específicas de una exhibición o **Exhibition**. La razón de esta diferenciación corresponde a la generación de jerarquías al momento de crear exposiciones. A modo de ejemplificar, se podría querer mostrar tres exhibiciones de forma simultánea y cada cual con sus propios contenidos específicos (que incluso podrían repetirse), como se puede ver en la Figura 4.7.

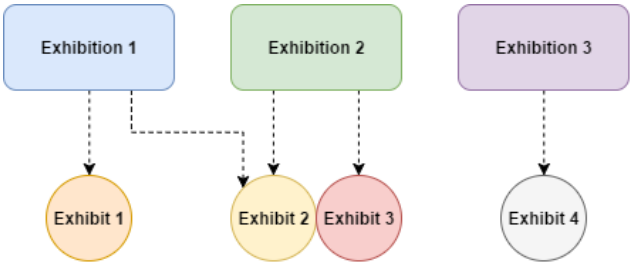

Figura 4.7: Jerarquía de exhibits.

Lo anterior puede verse como una relación One-to-Many entre Exhibition y Exhibit; el primero tiene la información de cuándo debe estar disponible para los visitantes el acceso a los exhibits, además de un nombre y un estado (publicado o no). Lo anterior, sumado a lo expuesto en **Exhibit** y **Opiniones**, conlleva a lo ilustrado en el diagrama de la Figura 4.8 que corresponde al diagrama entidad-relación completo de estas entidades.

4.3. Diseño de procesos

Con el fin de entender de mejor manera cómo funcionará el Framework, hay que hacer hincapié en cómo interactuarán los usuarios, y en cómo reaccionará el sistema frente a dichas interacciones. Para esto, se procederá a utilizar **diagramas de flujo** y **diagramas de secuencia**.

Los **diagramas de flujo** corresponden a las posibles interacciones que realizará un usuario

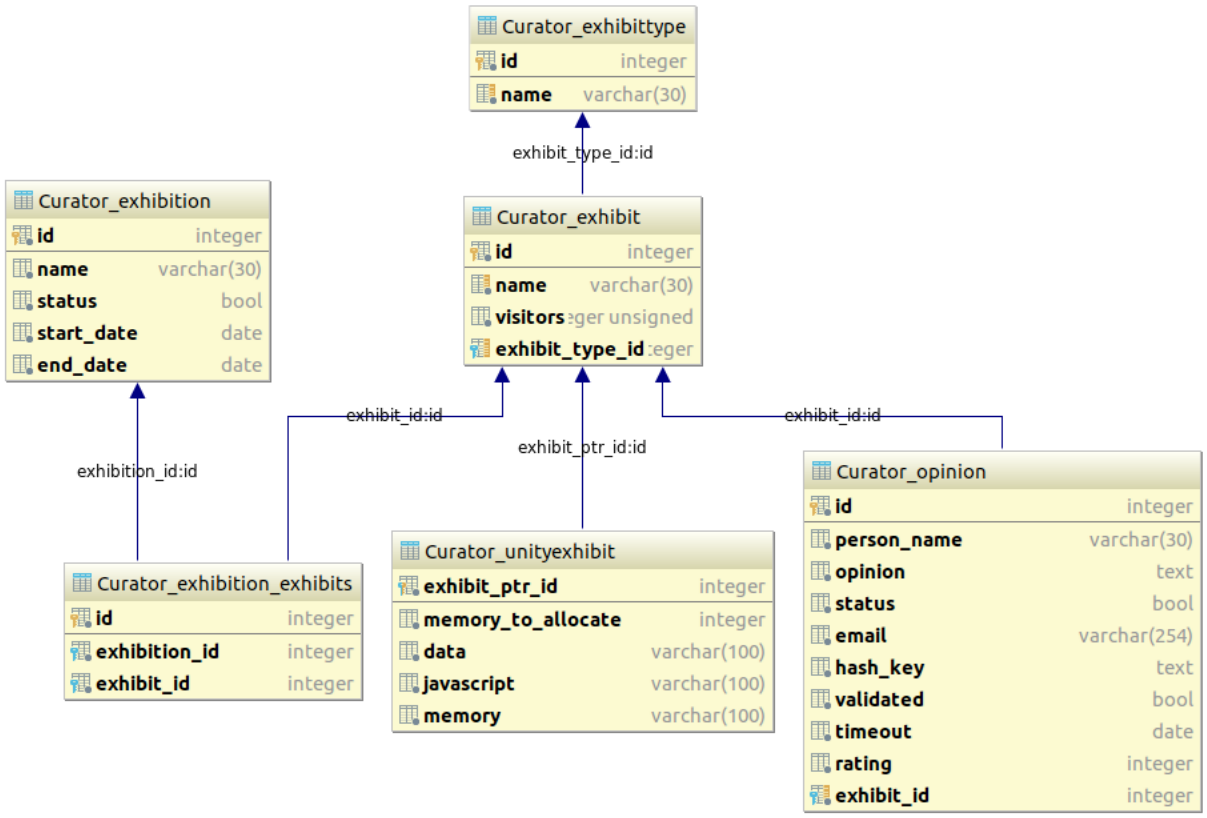

Figura 4.8: Diagrama relacional Exhibitions.

frente a interfaces visuales y, de forma más particular en nuestro caso, cómo navegará por un sitio web. Para esto se toma en cuenta que existen distintos tipos y niveles de permisos, los cuales pueden generar uno o más diagramas de flujo, o bien producir puntos de disyunción para omitir lugares restringidos.

Por otro lado, los **diagramas de secuencia** reflejan cómo funciona un procedimiento computacional, y cómo las partes de este interactúan entre sí a través del tiempo; se leen de arriba hacia abajo y se va saltando entre las líneas de cada componente, tal y como el sistema va llamando a sus elementos.

4.3.1. Diagrama de flujo de usuarios

En las primeras etapas del proyecto se diagramó el Framework como un conjunto de operaciones excluyentes dependiendo de si el usuario pertenecía o no a la categoría de Curador, generando así dos grupos, el primero denominado Visitante era toda persona no identificada y que accedía de forma inmediata a una navegación en 3D de los elementos del museo. De forma antagónica, el Curador disponía de un set de funcionalidades, tales como manejar opiniones, ver recursos y agenda y por último realizar un manejo de los denominados **cuartos**.

La razón de esto correspondía a la necesidad de que el Framework generara museos 3D, y los curadores pudieran manejar el contenido de cada cuarto del museo. Todo lo anterior puede verse en la Figura 4.9.

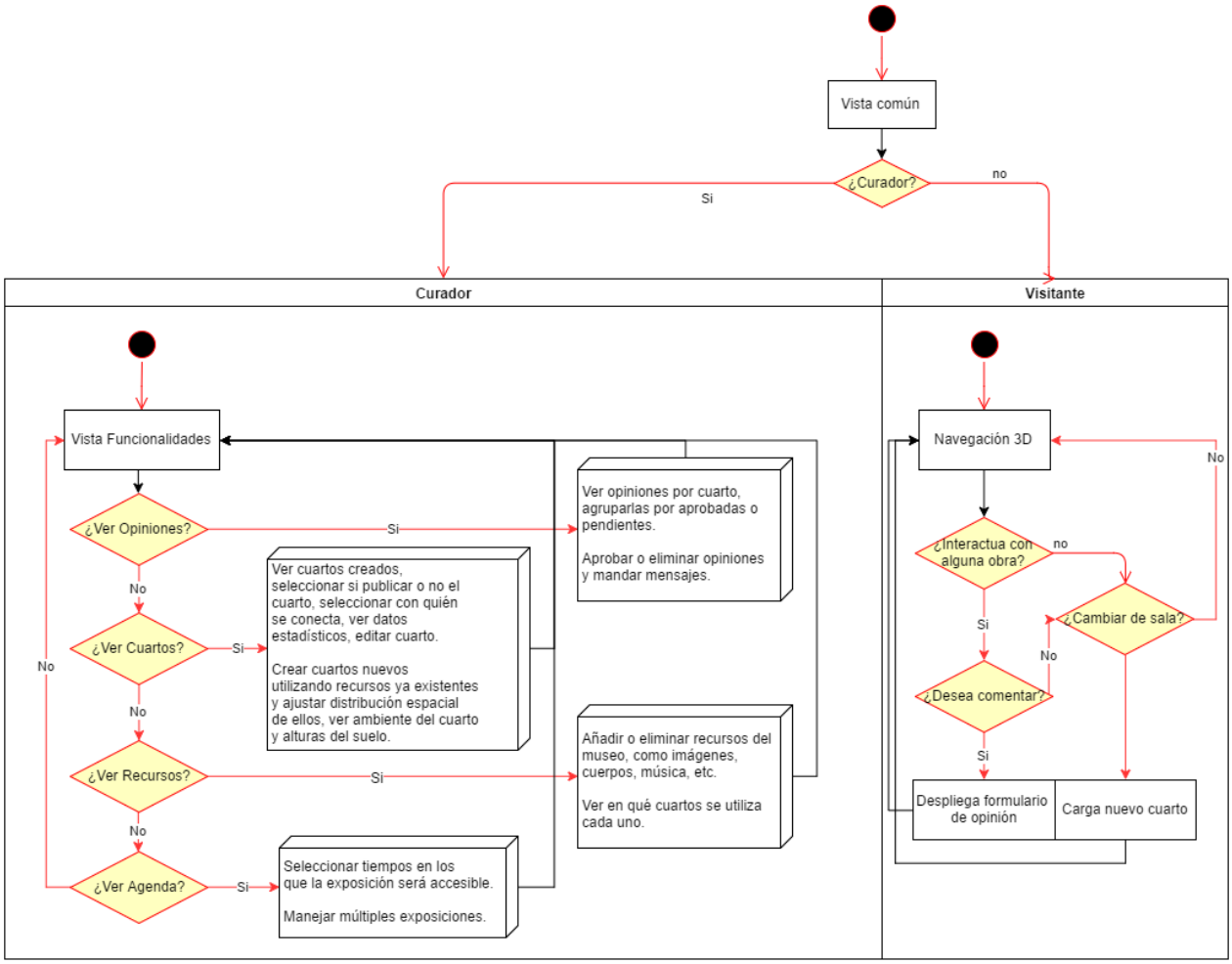

Figura 4.9: Primer boceto de diagrama de flujo del Framework.

Sin embargo, al no continuar con la idea de la generación 3D se debió dar especial énfasis en usabilidad y seguridad informática, lo que conllevó al rediseño de procesos. En la Figura 4.10 se puede observar lo denominado **Vista Común**, que corresponde a las interfaces visuales compartidas entre usuarios identificados y visitantes. En ella se dispone de un menú principal el cual permite acceder a las exhibiciones, información de contacto y la interfaz del curador (aquí se detiene el diagrama de flujo, pues pasa a uno específico). En caso de decidir ver las exhibiciones, se presenta un listado de exhibiciones disponibles. En caso de que no existan exhibiciones activas en dicho instante, se mostrará la fecha del más próximo en caso de existir (esta parte no se incluye de forma explícita en el diagrama). Cuando se tienen las exhibiciones, que sirven como una colección de exhibits, se permite el acceso a ver las exhibits en particular de cada uno y la posibilidad de visualizarlas y en consecuencia realizar una opinión sobre estas.

En lo referente a la vista de los curadores, al tratarse de una vista administrativa la

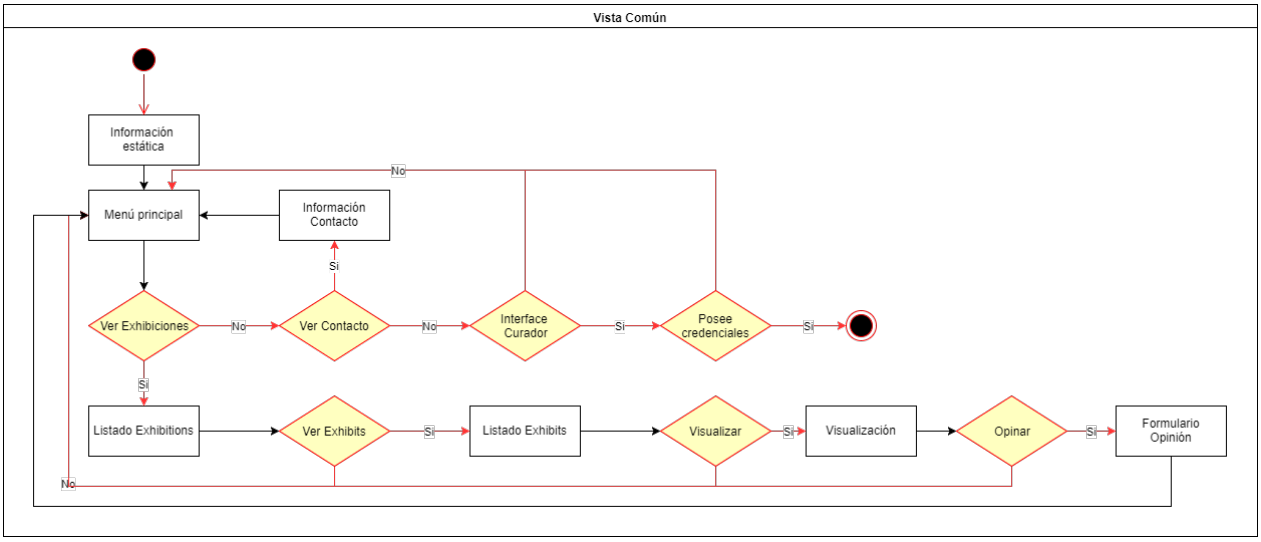

Figura 4.10: Diagrama de flujo de vistas comunes.

complejidad escala sobretodo al incluir grupos de usuarios y permisos para cada uno. En la Figura 4.11 es posible ver que las acciones iniciales fueron divididas en torno a cuatro grupos de usuarios: *Scheduling_team*, *Opinion_team*, *Museum_team* y *Resources_team*. Estos usuarios tienen vistas exclusivas, con salvedad de la Agenda, la cual es una vista común para todos, excepto por los botones de acciones, que dependen del *Scheduling_team*. La razón de esto corresponde a que todo integrante del equipo de curadores debe saber qué exhibiciones están actualmente activas, y otras informaciones relevantes. Otro cambio incluido, pero menos significativo, fue la inclusión de la posibilidad de editar datos personales de la cuenta, tales como información de contacto o contraseña. La razón de esto último fue permitir en trabajo futuro la intercomunicación entre distintos equipos utilizando dicha información, sin embargo para el presente proyecto se encontraba fuera del alcance.

4.3.2. Procesamiento de requests

Teniendo en mente ya la arquitectura lógica, cómo funcionará la base de datos y las interacciones de los usuarios, lo siguiente es abordar el cómo internamente se resolverá cada *request* realizada por los usuarios del Framework.

Es importante denotar que, al estar trabajando sobre Django, hay parte de esta lógica que ya está cubierta como corresponde a los llamados a la base de datos y así mismo la tenencia de un **URL Dispatcher**. Este último corresponde a un archivo de Python como el del Código 4.4, que es llamado cada vez que se realiza un request al servidor y, utilizando el arreglo `urlpatterns`, realiza una búsqueda de patrones para identificar cuál es la URL correcta (notar que utiliza expresiones regulares, por tanto se podrían utilizar para redirigir correctamente direcciones difíciles en términos humanos), y luego llamar a la vista correcta o, en casos específicos, llamar a otros URL dispatcher para modularizar la aplicación.

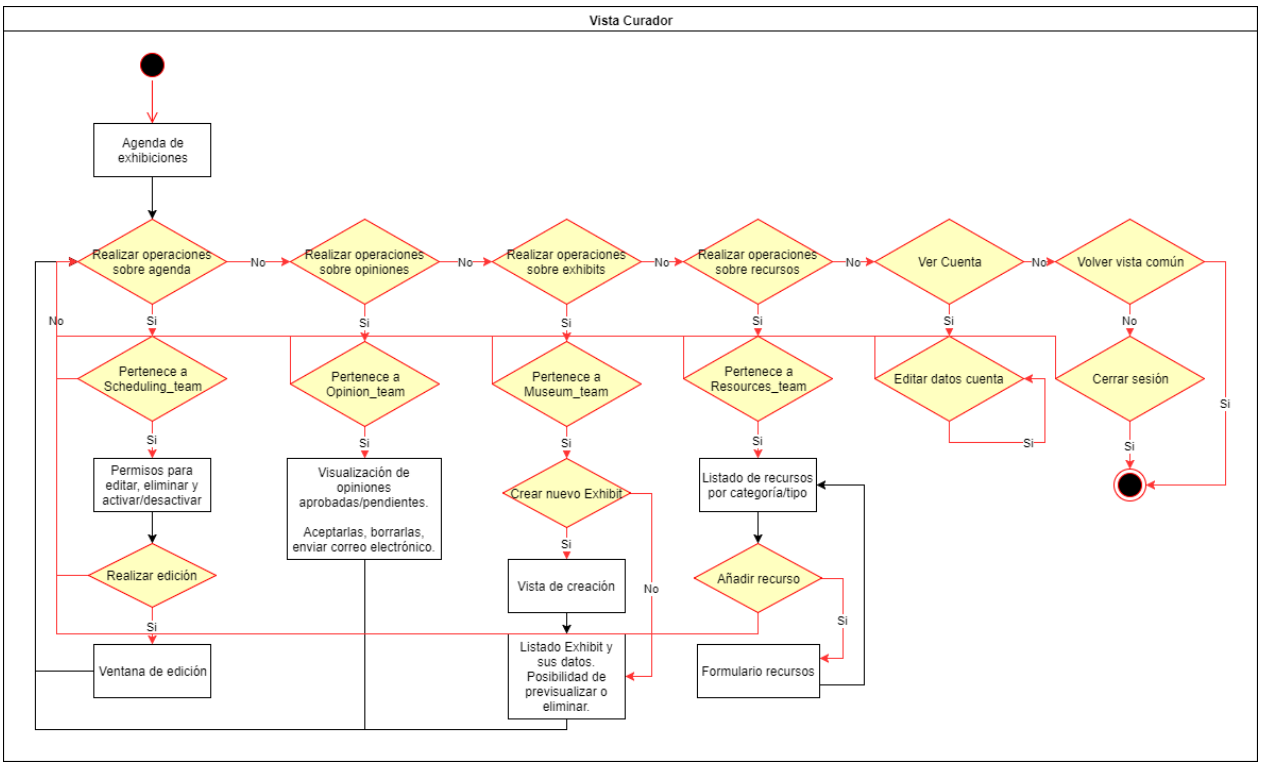

Figura 4.11: Diagrama de flujo de vistas curador.

```
urlpatterns = [
    url(r'^$', IndexView.as_view(), name='index'),
    url(r'^contact/', ContactView.as_view(), name='contact'),
    url(r'^auth/login', LoginView.as_view(), name='login'),
    url(r'^visitor/', include('Apps.Visitor.urls')),
    url(r'^favicon\.ico$', RedirectView.as_view(url='/static/images/museum.ico',
                                                permanent=True))]
```

Código 4.4: Ejemplo URL Dispatcher.

En el caso del presente proyecto, y como se pudo observar en lo mencionado anteriormente, en vez de entregarle a un usuario vistas de forma directa (utilizando HTML por ejemplo), se le pide a una respectiva **clase asociada** la labor de generar la vista. En el Código 4.5 se tiene una única URL, que corresponde a una página de contacto, para lo cual se entrega como primer parámetro la URL desde la cual debe ser accedida (en este caso `/contact/`) y se indica que la clase encargada de entregar la vista es `ContactView` mediante su función `as_view()`.

```
url(r'^contact/', ContactView.as_view(), name='contact')
```

Código 4.5: URL página contacto.

En consecuencia, y tal como se puede ver en la Figura 4.12, la secuencia de un request corresponde a un llamado al **URL Dispatcher**, quien busca a la respectiva **clase asociada**

y le entrega la consulta mediante un llamado GET o POST. No obstante, por temas de seguridad sería vulnerable el hecho de que con sólo realizar una consulta se entregue un resultado. Por ello, utilizando la request misma del usuario, y los datos que Django automáticamente va guardando en estas y en las cookies, se realiza un chequeo de permisos con el **gestor de permisos**, el cual corresponde a una serie de métodos que permiten comprobar contra la **base de datos** si la request proviene de un usuario autentificado o no, y si pertenece a un grupo determinado de usuarios en caso de ser una vista restringida por permisos. Si los permisos son correctos, la clase asociada procede a realizar operaciones internas y, de requerirlo, realizar consultas a la base de datos por información específica para luego generar una cadena de retornos. Es importante indicar que si el gestor de permisos no valida al usuario, lo que en verdad se retornará corresponderá típicamente a una página de código `404`.

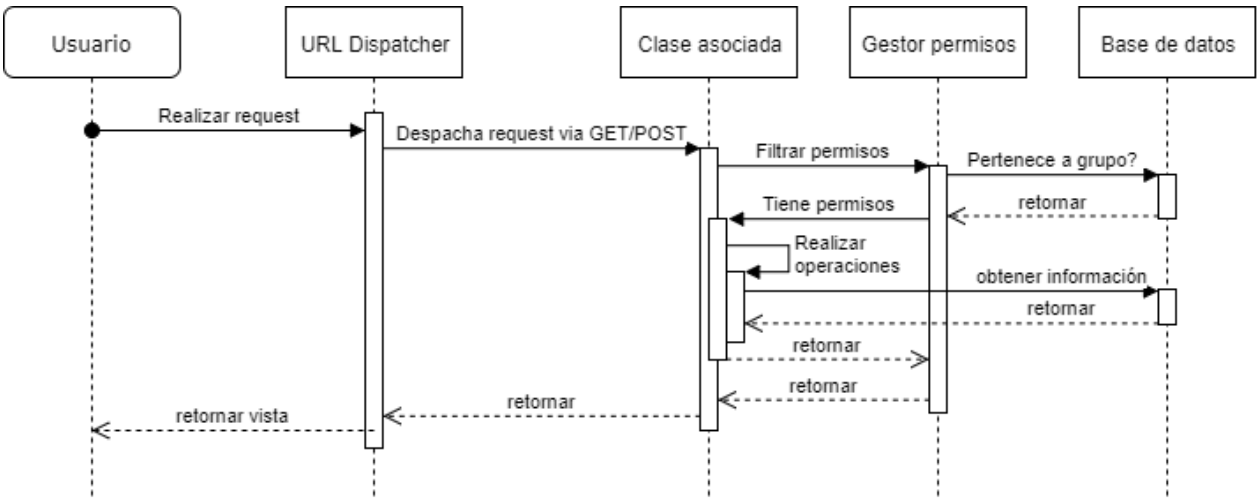

Figura 4.12: Diagrama de secuencia de un request.

Se puede observar así mismo en la Figura 4.12 el cómo se van atravesando las distintas capas de la *Arquitectura de tres capas*. Partiendo en el usuario, que se encuentra en la *capa de presentación* dónde se tiene la visualización que le entrega la *capa de negocio*, que corresponde a: el URL Dispatcher, la clase asociada a la vista y el gestor de permisos. Finalmente, estos últimos acceden a la *capa de datos*.

4.4. Diseño interfaces

Si bien las interfaces visuales están ligadas intrínsecamente con la usabilidad por parte de un usuario, es importante darle especial énfasis a las mecánicas tras estas, ya que al tratarse de un framework lo más relevante es entender el cómo funciona, cómo utilizarlo y extenderlo. Por ello, a continuación se explicará la relación de las vistas con Django, y cómo además se relacionan con el lenguaje Python para lograr el objetivo buscado.

No obstante, se explicará también la usabilidad básica del sistema para seguir el principio de tener un comportamiento predeterminado funcional. La razón principal de que este contenido esté poco abordado radica en que generalmente cada entidad, como una empresa o

institución, que utiliza un sitio web tiene asociada un patrón o esquema de colores, figuras, etc, las cuales definen la **identidad** de un sitio web o de una compañía, y que sobreescribirán la estética del Framework y, en consecuencia, su usabilidad. Hoy en día todo ello es bastante más sencillo al abstraerse de los archivos HTML y concretarse en los archivos CSS, además de la utilización de frameworks visuales como lo son **Bootstrap**. En el capítulo **Validación de la solución** se ilustrará el cambio de usabilidad dependiendo de la identidad del sitio.

4.4.1. Django templates y extensibilidad

Una de las grandes ventajas que posee Django corresponde a la existencia y utilización de **templates**. Una template corresponde a un archivo estático de tipo `HTML` que funciona como plantilla (tal y como indica su nombre), la gran diferencia con el uso estándar de HTML radica en que el servidor debe **renderizar** cada template antes de llegar a un usuario, una suerte de compilación. Y por tanto, le permite agregar cierta lógica a los archivos estáticos que no llega directamente al usuario, como lo es utilizar funciones y directrices de Python dentro del archivo HTML, o bien definir plantillas reutilizables y extensibles mediante tags especiales.

La idea tras el funcionamiento de los templates cae en los tags especiales que permiten ejecutar código Python. El primer tag importante es `{%%}` , el cual permite realizar operaciones lógicas tales como crear bloques de posible extensión mediante la instrucción `{% block name_block%}{% endblock%}`, extender plantillas con `{% extends ‘‘base_file.html’’%}`, o bien operaciones más estándares como lo son `{% if variable%} {% endif%}`. El segundo tag importante es `{{ variable }}` que funciona como una 'impresión' de variables sobre el archivo HTML; como ejemplo de esto último podemos imaginar un perfil de usuario donde su nombre en el template se ve con la operación `Usuario: {{ user.username }}`.

Dicho lo anterior, podemos ver el Código 4.6 donde se tiene el **head** de un documento HTML, el cual se utilizará como base para los archivos del proyecto, con el fin de siempre utilizar las mismas librerías base de CSS y Javascript, pero, simultáneamente, permitir la utilización de otros archivos según se requiera al incorporar los bloques `{% block css%}` y `{% block js%}`. En lo referente al cuerpo de la página, este se completará utilizando el bloque `{% block body%}`.

En el Código 4.7 se extiende la template base con el fin de darle un cuerpo a la página a ser visualizada. Para esto se utiliza el tag `{% extends ‘‘base/base.html’’%}`, y a continuación deben declararse todos los bloques anunciados por su template padre, tal y como si se tratasen de clases abstractas. Una particularidad de este ejemplo es la inclusión del tag `{% include ‘‘curator/navbar.html’’%}`, el cual corresponde a importar un archivo HTML y 'pegarlo' donde se anuncia el tag. Esto es muy útil cuando se tienen piezas de código HTML estático que se utilizan varias veces, pero no siempre, como en el ejemplo donde se incluye una barra de navegación que sólo le pertenece a las interfaces de los curadores, permitiendo así que los visitantes tengan una particular y no deba existir código duplicado.

Siguiendo este mismo principio para evitar código duplicado, es posible generar estructuras genéricas que pueden ir utilizándose en vistas similares pero que, a su vez, mantengan su

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="author" content="Michel Llorens">
    <title>Virtual Museums: Curator</title>

    <link href="/static/.../css/bootstrap.min.css" rel="stylesheet">
    <link href="/static/.../css/roboto-fontface.css" rel="stylesheet">
    {% block css %}{% endblock %}

    <script src="/static/.../jquery.min.js"></script>
    <script src="/static/.../bootstrap.min.js"></script>
    {% block js %}{% endblock %}
  </head>
  {% block body %}{% endblock %}
</html>
```

Código 4.6: Template base del proyecto.

respectivo contenido. Un ejemplo de esto son las Figuras 4.13, 4.14 y 4.15 que corresponden a las vistas de administración de recursos, las cuales no sólo reutilizan la barra de navegación, sino también la estructura comprendida entre el título, el selector y el botón de añadir nuevo recurso (ver Figura 4.16), que van respondiendo acorde al recurso seleccionado.

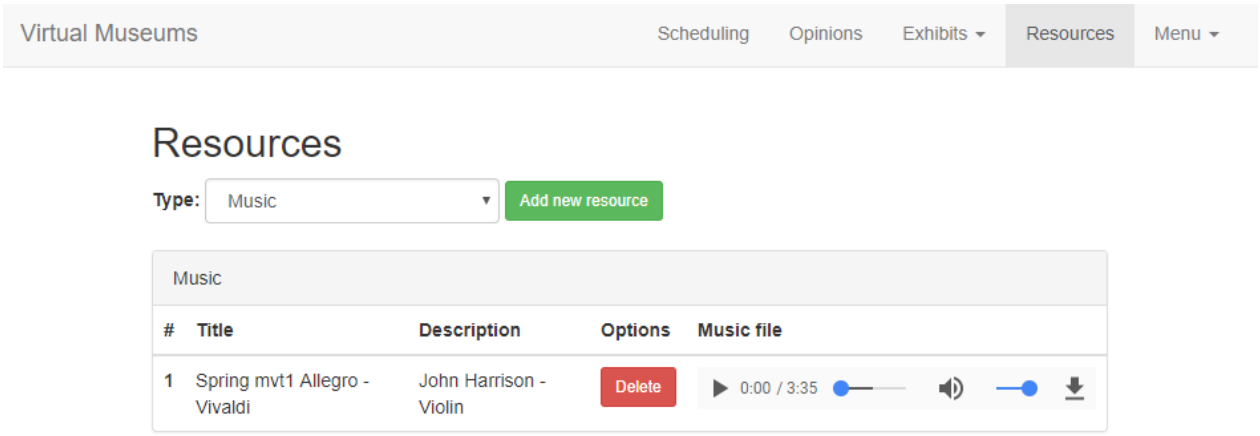

Figura 4.13: Vista de recursos de audio.

```
{% extends "base/base.html" %}

{% block css %}{% endblock css %}
{% block js %}{% endblock js %}

{% block body %}
    <body>
    {% include "curator/navbar.html" %}
    <div class="container">
        <h2>Title</h2>
    </div>
    </body>
{% endblock body %}
```

Código 4.7: Extensión de template.

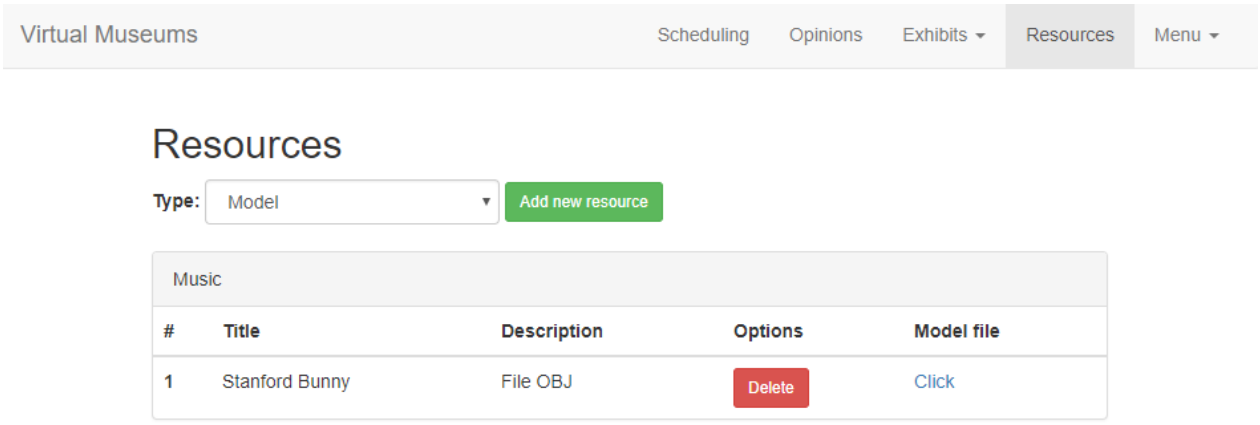

Figura 4.14: Vista de recursos de modelos.

4.4.2. Programación orientada a objetos

Tal y como se ha mencionado a través del presente capítulo, las vistas son una asociación entre clases y templates, donde estas últimas de cierta forma siguen ya un paradigma orientado a objetos a fin de reutilizar código. En consecuencia, es intuitivo pensar que las clases asociadas también deben seguirlo.

En el Código 4.1 se pudo observar que cada clase vista hereda de una clase de Django llamada **TemplateView**, análogo a los modelos que heredaban de `Model`. Dicha herencia nos permite utilizar y sobreescribir los métodos `get(self, request, *a, **ka)` y `post(self,`

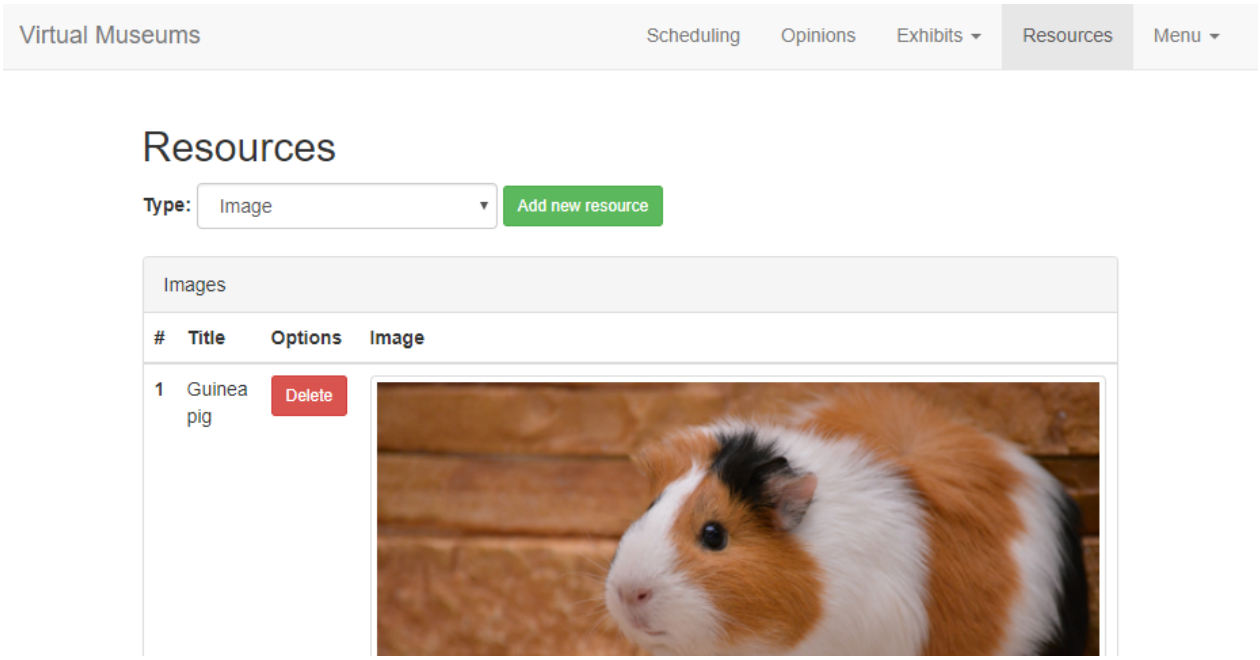

Figura 4.15: Vista de recursos de imágenes.

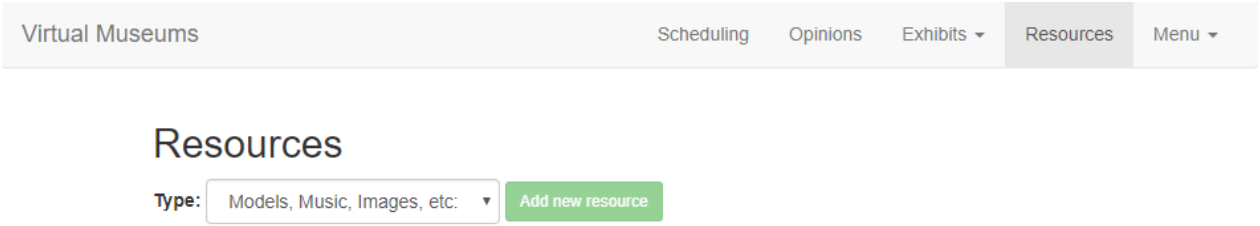

Figura 4.16: Templates reutilizados utilizando {% include %} para ver recursos..

`request, *a, **ka)`, los cuales son utilizados como caja negra por Django para procesar las requests. Entonces, esta herencia resume el implementar una vista (sin entrar en la lógica de cada una) en extender `TemplateView` e implementar el método GET o POST (según se requiera), y finalmente **renderizar** una template, pasando al método `render` la request, el path al template y, como argumento opcional, un conjunto de variables que el template podrá utilizar mediante sus funciones `{%%}` y `{{ }}`.

Toda esta estructura de herencia permite aplicar un principio fundamental de la programación orientada a objetos, que es que si a una clase le piden un método que sólo el padre tiene, entonces se llama dicho método. Además, teniendo en consideración los permisos de cada método y cada variable, y al hecho de que, a través de investigación y práctica, se pudo corroborar que las visibilidades al menos corresponden al nivel `protected`, lo que tiene como consecuencia, que se pueda reutilizar código como en el Código 4.8; donde se creo una clase de vista genérica llamada `AddExhibitView` que permite agregar al servidor un nuevo **exhibit**, pero, como bien se sabe, estos no poseen un único tipo ni almacenan la misma cantidad de archivos, y de hecho es la pieza fundamental que se busca poder extender al utilizar el Fra-

mework. Entonces, a fin de reutilizar código y facilitar la extensión a nuevos tipos de exhibit, la clase `AddExhibitView` utiliza formularios y tipos de exhibits de acuerdo a sus variables internas `form` y `exhibit_type`, que están instanciadas con código de ejemplo, pero nunca serán llamados realmente, pues la clase `AddUnityView` (que está asociada a una URL en el URL Dispatcher) sobre escribió las variables internas con información correcta, como lo son el formulario `UnityExhibitForm`, el cual será pasado como argumento al template para luego visualizarlo correctamente, y, de dicha forma, abstraer el cómo agregar un nuevo exhibit sin tener que utilizar metodologías del tipo `if-else if-else` o `switch-case-default`, que en general son mal vistas e implican una mayor inmersión en el código del Framework para lograr incorporar nuevas features.

```
class AddExhibitView(TemplateView):
    template_name = 'curator/add-exhibit.html'
    form = NewExhibitForm
    exhibit_type = ''

    @method_decorator(group_required('Museum_team'))
    @method_decorator(login_required(login_url='/auth/login'))
    def get(self, request, *a, **ka):
        form = self.form()
        args = {'form': form, 'exhibit_type': self.exhibit_type}

        return render(request, self.template_name, args)

    @method_decorator(group_required('Museum_team'))
    @method_decorator(login_required(login_url='/auth/login'))
    def post(self, request, *a, **ka):
        request_form = self.form(request.POST, request.FILES)
        args = {'exhibit_type': self.exhibit_type}

        if request_form.is_valid():
            request_form.save()
            args['success'] = 'success'
            request_form = self.form()

        args['form'] = request_form

        return render(request, self.template_name, args)


class AddUnityView(AddExhibitView):
    form = UnityExhibitForm
    exhibit_type = 'Unity'
```

Código 4.8: Implementación de herencia en clases vista.

4.4.3. Utilización de diccionarios

Pese a la existencia de clases y herencia en Python, hay situaciones en las cuales se necesita tener una única página capaz de manejar distintos elementos y tipos sin tener que crear URLs adicionales y, en consecuencia, nuevas clases. Un ejemplo de esto es la vista de manejo de recursos que maneja distintos tipos desde una única vista basal (ver Figura 4.16), donde, si bien implementamos un esquema de templates para poder tener la visualización adecuada en torno a cada recurso, no necesariamente cada recurso tiene la misma estructura, o peor aún, no todos los recursos se buscan o eliminan con las mismas instrucciones. Basta recordar el diagrama de la Figura 4.4 para darse cuenta que si se necesita, ejemplificando, un recurso de audio no se puede llegar y realizar una consulta al modelo padre `externalfile`, pues este no incluye un 'tipo' en su información.

Por tanto, es claro que necesitamos una estructura más flexible, y que sea fácil extender, para alojar la información y métodos que corresponden a cada tipo. Para esto, recordaremos que Python es un lenguaje bastante flexible, capaz de funcionar con programación orientada a objetos pero también utilizando programación imperativa como lo es el lenguaje de programación C y por tanto funcionando de forma similar en varios aspectos (tal y como indica **van Rossum**[14]).

Dicho esto, lo ideal sería tener un listado de **struct** que sigan la misma estructura e ir llamándolos acorde a una etiqueta. Para esto se postuló la utilización de *clases abstractas*, sin embargo estas necesitarían una mayor inmersión de un usuario para entenderlas bien, ya que en Python, al no ser un lenguaje compilado sino interpretado, es mucho más sencillo generar una excepción por no implementación de un método abstracto, y botar el servidor, que seguir mal un guideline y producir errores menores que Django es capaz de atajar y seguir funcionando sin mayores inconvenientes.

La solución a este problema viene en que Python permite **funciones de orden superior** (las funciones o métodos pueden ser utilizados como parámetros o asignados a variables) y en la utilización de **diccionarios**. Dicha combinación permite tener un diccionario de funciones y, así mismo, composición o anidación de diccionarios. Un ejemplo de esto es el Código 4.9.

```
1 def add(x,y): return x + y
2 def hello(): print "hello world"
3 function_dictionary = {
4     'levelOne': {
5                 'sum': add,
6                 'levelTwo': {
7                             'hello': hello,
8                       },},}
```

Código 4.9: Anidación de diccionarios y funciones como variables.

Con estas herramientas se definió la variable global para las vistas de recursos llamada `POSSIBLE_RESOURCE`, la cual es un diccionario que utiliza el nombre de los tipos de recursos como llave, aprovechando así que esa variable ya se manejaba en las requests de los usuarios. Luego cada 'valor' del diccionario corresponde a un segundo diccionario que funciona similar

a un `struct`, teniendo cinco llaves: `name`, `form`, `template`, `elements` y `delete`, las cuales corresponden al nombre del recurso (notar que un recurso puede tener un nombre complejo y se acepta la utilización de diminutivos), el formulario para subir recursos, el template asociado a dicho recurso y las funciones para obtener los elementos y eliminar los elementos de dicho tipo. Todo esto se puede ver en el Código 4.10

```
POSSIBLE_RESOURCE = {
    'Music': {
        'name': 'Music', 'form': MusicForm, 'template': '.../resources-music.html',
        'elements': query_music, 'delete': delete_music
    },
    'Image': {
        'name': 'Image', 'form': ImageForm, 'template': '.../resources-images.html',
        'elements': query_image, 'delete': delete_image
    },
    'Model': {
        'name': 'Model', 'form': ModelForm, 'template': '.../resources-models.html',
        'elements': query_model, 'delete': delete_model
    },
    'Video': {
        'name': 'Video', 'form': VideoForm, 'template': '.../resources-video.html',
        'elements': query_video, 'delete': delete_video
    },
}
```

Código 4.10: Diccionario que almacena la información de cada recurso.

4.4.4. Usabilidad en las interfaces

En el carácter de componente mínimo funcional del Framework, las interfaces visuales circularon entorno al principio de ser explícitas, lo que significa que deben denotar explícitamente qué acciones realizan para aportar **Learnability** y **Efficiency**, además de mantener un patrón regular en la forma de desempeñar acciones para generar **Memorability** y **Satisfaction**.

En la Figura 4.17 se puede ver el contraste entre dos vistas correspondientes a las opiniones y a los recursos, las cuales, si bien no presentan un mismo enfoque funcional, comparten el mismo principio de usabilidad correspondiente a la agrupación de contenido y la factibilidad de realizar acciones dependiendo de la selección, ambas en un mismo bloque visual (ver Figura 4.18) que continua existiendo independientemente de las acciones a tomar (salvo incluya cambiar de visualización).

Así mismo, este fenómeno se repite en las interfaces que representan un listado de componentes como scheduling con exhibits, donde, además, se puede apreciar el factor de aprendizaje y eficiencia al estar explícito qué acciones se pueden realizar sobre sus elementos; en

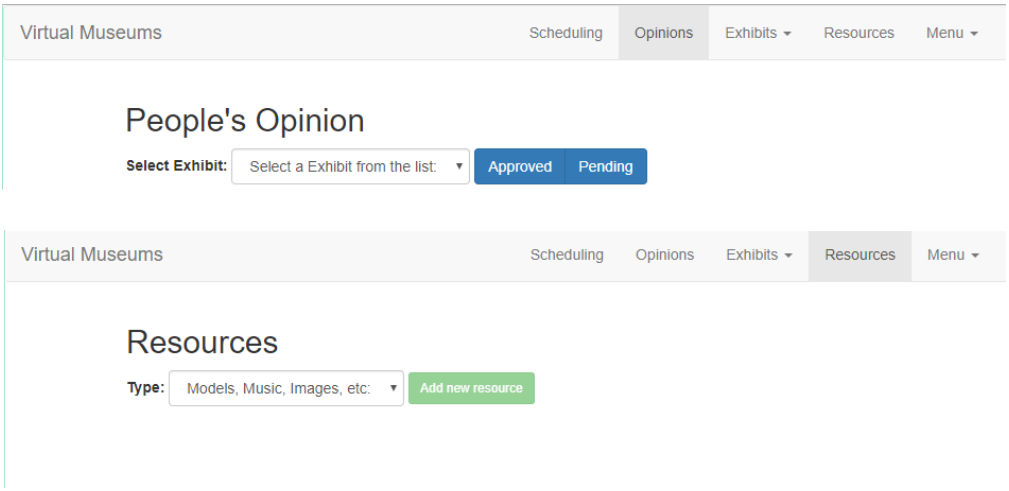

Figura 4.17: Contraste de Memorability entre vistas.

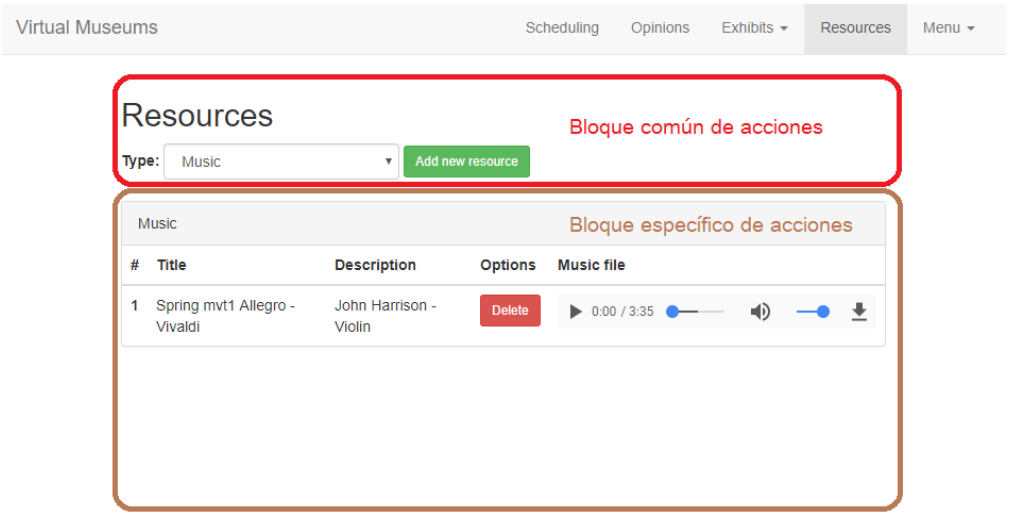

Figura 4.18: Bloques en una vista.

particular como se puede ver en la Figura 4.19, lo que además conllevó a decidir no incluir acciones por grupos (como eliminar todos), pues se prestaban para confusión.

Ahora bien en lo referente a los factores claves de un sitio web, se buscó que cada vista destacara todos los elementos con los cuales puede tener interacción utilizando un color distinto al blanco/gris generales, con excepción de la barra de navegación, la cual, además de ser auto explicativa, agrupa los elementos a modo de mejorar la navegación por parte de los curadores y generar organización. Todo esto sumado a que, dependiendo de los permisos que tenga cada usuario, este verá más o menos menús como se puede ver en la Figura 4.20.

En lo referente al tiempo de descarga, todas las páginas intentan cargar el menor contenido posible salvo cuando explícitamente se necesita, como lo son visualizar recursos o cargar un exhibit.

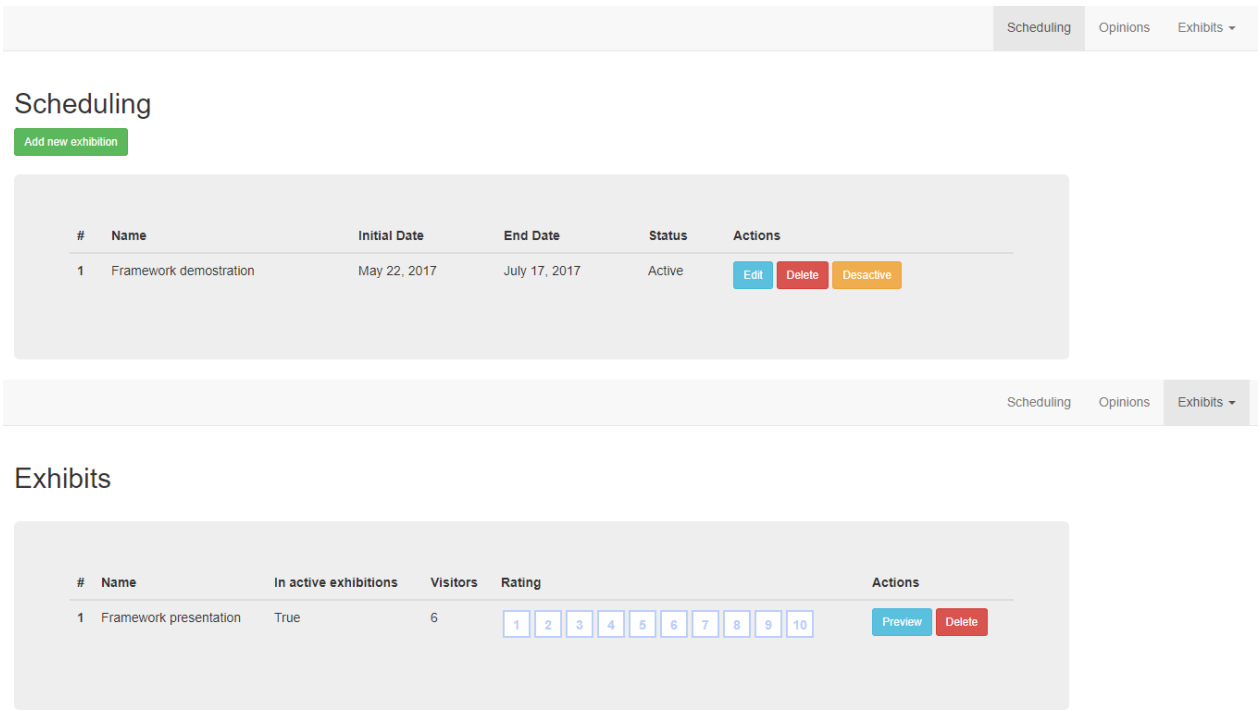

Figura 4.19: Vistas con acciones específicas por elemento.

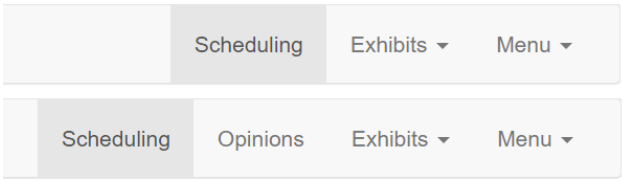

Figura 4.20: Barras de navegación de usuarios con distintos permisos.

La visualización de la parte de curadores es la que debió tener más cuidado, debido a que corresponde a una zona de uso intenso y recurrente por parte de equipos consolidados, mientras que, por otro lado la vista del visitante se enfocó en facilitar la navegación, los tiempos de carga y la interactividad. Como se puede ver en la Figura 4.21, la navegación entre las exhibiciones y exhibits contempla una jerarquía explícita, tal y como se diseñó en los modelos, mientras que una vez visualizado un exhibit, el acceso a la vista de emisión de opinión queda con un acceso directo y descriptivo en la barra de navegación de visitantes(ver Figura 4.22). El único cambio de paradigma corresponde a que la acción de emitir opiniones abre una pestaña nueva del navegador, con el fin de no colisionar con la visualización en sí.

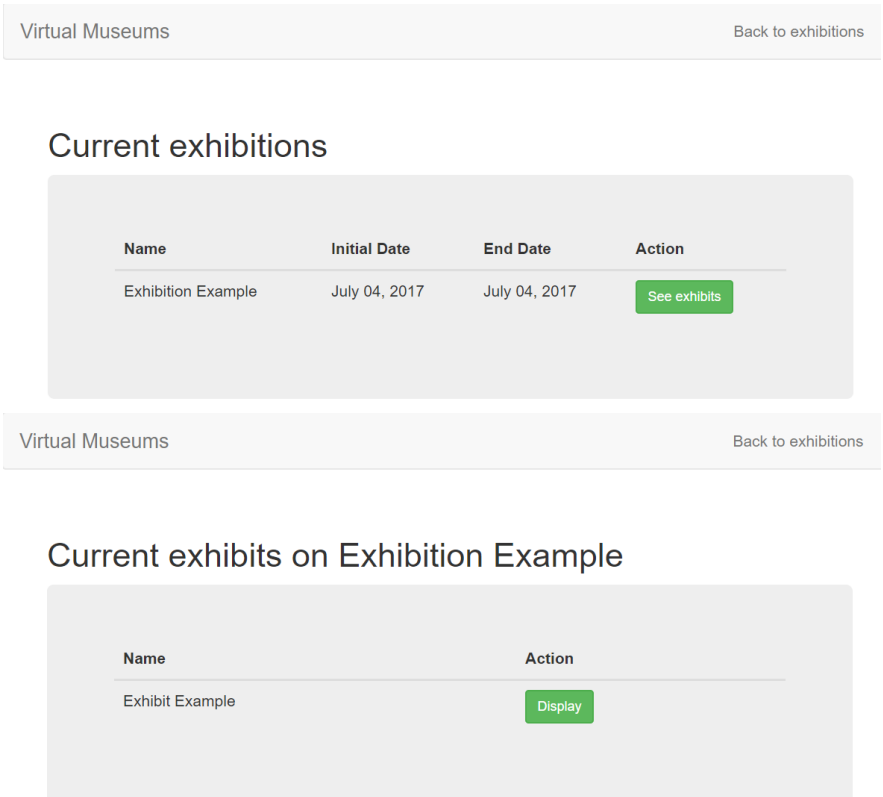

Figura 4.21: Vistas jerárquicas para ver exhibits.

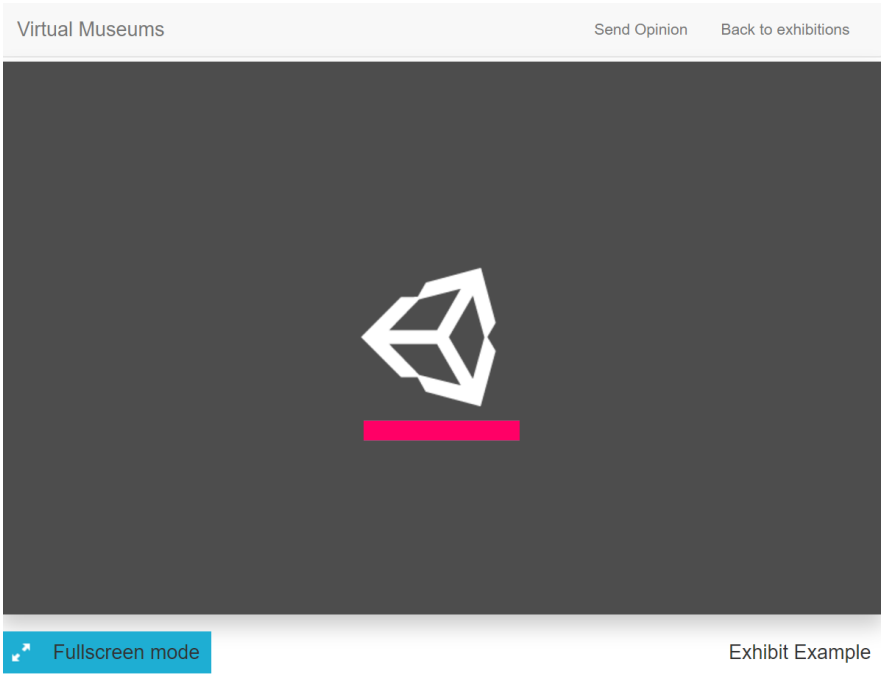

Figura 4.22: Vista de visualización de exhibit.

4.5. Seguridad en Django

Tal y como se especificó en el **Marco Teórico**, la seguridad informática es un punto crucial en un sitio web, sobretodo si este dispone de vistas restringidas a tipos de usuario, o, más generalmente, cuando existe manejo de cuentas.

Ante ello, es necesario implementar medidas de seguridad como lo son: comprobación de formularios, usos de GET/POST y decoradores, que se explicarán a continuación.

4.5.1. Formularios y validadores

Generalmente en sitios web los formularios de inicio de sesión están escritos en HTML y Javascript para comprobar la información de estos desde la capa de presentación. Esto cambió con la aparición de HTML5 que, asignando tipos específicos para cada campo, permitió evitar realizar dobles comprobaciones en formularios, como el hecho de que un correo electrónico tenga el formato correcto.

Sin embargo, aún así los formularios no poseen una seguridad basal por defecto. Django incluye entre sus funciones el tag `{% csrf_token %}`, que genera automáticamente en un formulario un token CSRF asociado al usuario que ingresó y, por tanto, permite internamente comprobar dicho token y saber si lo emitió el usuario correcto.

Esta medida ayuda a evitar suplantación de identidad mediante el robo de cookies, o derechamente del token CSRF. Sin embargo, este token **sólo puede ser enviado mediante POST**. Lo que nos lleva al siguiente método de seguridad que ofrece Django, la definición explícita de qué métodos permite una determinada URL, tal y como se puede ver en el Código 4.11, que permite get (que redirige a una página de error específica) y post (que renderiza la vista). En caso de que un método no esté definido el sistema arrojará una vista de error `404`.

```
1 class ExampleView(TemplateView):
2
3     def get(self, request, *a, **ka):
4         return redirect('/visitor/error')
5
6     def post(self, request, *a, **ka):
7         return render(request, template, arguments)
```

Código 4.11: Clase vista con métodos GET y POST.

Django también nos ofrece la oportunidad de convertir los formularios en clases que después serán visualizadas. Todo esto mediante la herencia de la clase `forms.ModelForm` y la utilización de clases internas llamadas `Meta`. En el Código 4.12 puede verse la clase `TemplateForm`, que es un formulario para completar la información de un modelo (en este caso `ExternalTemplate`), el cual perfectamente puede ser un recurso. En este ejemplo además se utilizó herencia de la clase `CommonMeta` a fin de poder reutilizar esa información común entre formularios, como corresponde a pedir los campos `title`, `description` y `file`,

además de que el campo `description` debe tener características específicas al momento de convertirse en HTML.

```
class CommonMeta:
    fields = ('title', 'description', 'file',)
    widgets = {
        'description': Textarea(attrs={'cols': 30, 'rows': 5}),
    }

class TemplateForm(forms.ModelForm):
    class Meta(CommonMeta):
        model = ExternalTemplate
```

Código 4.12: Clase que representa un formulario.

Ahora retomando el tema de seguridad, un punto importante recae en la subida de archivos a un servidor, pues estos pueden ser archivos maliciosos y la única forma de tener total certeza es sanitizarlos. Django, por tanto, nos permite crear métodos para validar nuestros formularios y evitar estos problemas. Siguiendo el caso del Código 4.12, podemos tener el Código 4.13, que corresponde a la clase modelo del formulario, la cual solicita como parámetros de su campo `file` las variables `upload_to` y `validators`, que corresponden a acciones que se harán antes de guardar un archivo.

En el caso de `validators`, corresponde a una lista de validadores que el archivo debe pasar antes de poder guardarse en el servidor. En este caso puntual, se valida que su extensión en verdad sea `.txt` y no un archivo del estilo `file.php.txt`. Si todos los validadores son aprobados, se pasa a `upload_to`, quien renombra el archivo utilizando funciones de hash para luego reposicionarlo en algún directorio y, de dicha manera, aumentar la seguridad del servidor.

```
def validator_template(external_file): file_extension_validation(external_file, ['.txt'])
def rename_template(instance, filename):
    return file_rename(filename, '/static/external-content/template')

class ExternalTemplate(ExternalFile):
    file = models.FileField(upload_to=rename_template, validators=[validator_template])
```

Código 4.13: Validación de un formulario para la clase `ExternalTemplate`

4.5.2. Decoradores

Finalmente, la última gran herramienta que tenemos para aumentar la seguridad de la aplicación son los denominados **decoradores**. Ellos corresponden a un simple tag que se antepone a una función o método y que realizan operaciones **antes de invocar el método**. En consecuencia, si uno pasa correctamente los decoradores, se puede usar el método y si no, se generará un error del estilo `404` o una redirección. Un ejemplo de esto se puede ver en

el Código 4.14, donde el método `get` requiere que el usuario se haya autentificado antes de utilizarlo y, en caso de que no lo haya hecho, redirigirlo hacia el login; por otro lado, el método `post` además requiere que el usuario pertenezca específicamente al grupo `example_group`, lo que resulta en un error `404` en caso de que intente accederse sin los permisos adecuados pero estando con una sesión iniciada.

```
class ExampleView(TemplateView):

    @method_decorator(login_required(login_url='/auth/login'))
    def get(self, request, *a, **ka):
        return render(request, template)

    @method_decorator(group_required('example_group'))
    @method_decorator(login_required(login_url='/auth/login'))
    def post(self, request, *a, **ka):
        return render(request, template)
```

Código 4.14: Ejemplos de decoradores.

El decorador `login_required` está provisto por Django como parte de su framework y su uso es tal y como se explicó. Por otro lado, el decorador `group_required` fue creado como parte del presente Framework, siendo una facilidad de Django el poder crear uno mismo decoradores; en este caso el decorador puede verse en el Código 4.15, donde se aprecia que se comprueba que el usuario pase la prueba de pertenecer a un grupo específico contra la base de datos, siempre y cuando este esté autentificado.

```
def group_required(*group_names):
    def in_groups(u):
        if u.is_authenticated():
            if bool(u.groups.filter(name__in=group_names)) | u.is_superuser:
                return True
            return False
    return user_passes_test(in_groups)
```

Código 4.15: Decorador de comprobación de grupos.

Capítulo 5

Validación de la solución

El presente capítulo tiene como finalidad retomar los criterios de aceptación planteados en **Especificaciones del Problema** y, además de contrastarlos con la solución desarrollada, utilizar esta misma para validar todos los puntos referentes a utilización por terceros del Framework, como lo son: la extensibilidad, el deployment, guías de uso y retroalimentación por parte de usuarios, para comprobar temas de usabilidad.

5.1. Extensión Framework

Con el fin de cumplir los **criterios de aceptación**, una parte fundamental de la validación corresponde a la extensión del Framework a una versión operativa y específica. Por esto, y con el fin de generar una vista agradable para visitantes de un museo, se decidió dar énfasis en modificar la vista para el visitante y mantener las interfaces de los curadores, no obstante, se incorporaron nuevos tipos de exhibits que agregaron nuevos formularios y previsualizaciones a los curadores.

5.1.1. Nuevos exhibits: Vídeo y PDF

El Framework tiene como base exhibits confeccionados en Unity, los cuales poseen un formulario que puede verse en la Figura 5.1, el cual tiene como parámetros el nombre del exhibit, la cantidad de memoria RAM que utilizará (al crear una aplicación web de Unity se debe especificar cuánta RAM tiene asociada), y los tres archivos que entrega el proyecto de Unity, que contienen en sí toda la información. En términos de modelos, los exhibits de Unity estaban asociados al Código 5.1, en el cual se puede apreciar que posee una clase foránea llamada `ExhibitType` que, mediante la sobre escritura del método `save(self, *args, **kwargs)`, sirve para definir el tipo de cada exhibit, a fin de que cada uno sepa qué tipo le corresponde.

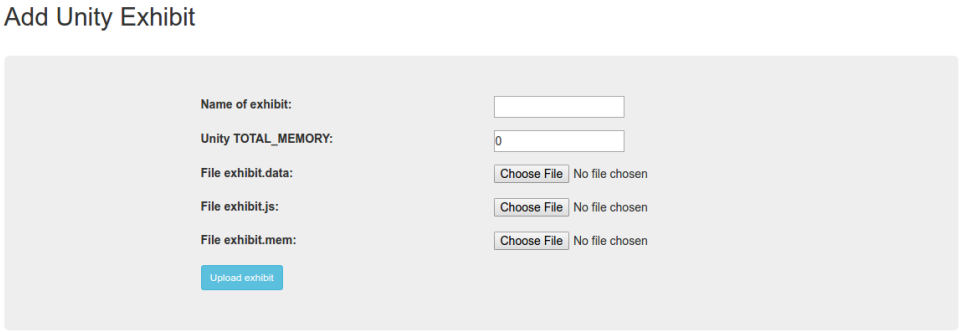

Figura 5.1: Formulario de subida de exhibit tipo Unity.

```python
class ExhibitType(models.Model):
    name = models.CharField(max_length=30, blank=False, unique=True)

class Exhibit(models.Model):
    name = models.CharField(max_length=30, blank=False, unique=True)
    visitors = models.PositiveIntegerField(default=0)
    exhibit_type = models.ForeignKey(ExhibitType, null=True)

class UnityExhibit(Exhibit):
    memory_to_allocate = models.IntegerField(default=0)
    data = models.FileField(upload_to=rename_unity_files,
                            validators=[validator_data])
    javascript = models.FileField(upload_to=rename_unity_files,
                            validators=[validator_javascript])
    memory = models.FileField(upload_to=rename_unity_files,
                            validators=[validator_memory])

    def save(self, *args, **kwargs):
        self.exhibit_type = ExhibitType.objects.get(name='Unity')
        super(UnityExhibit, self).save(*args, **kwargs)
```

Código 5.1: Clases asociadas al exhibit Unity.

Siguiendo el mismo patrón, se implementaron los exhibits **video** y **pdf**, que tienen como finalidad exponer vídeos, y mostrar documentos o presentaciones en formato PDF, respectivamente. Para esto, se siguió el estándar que se puede apreciar en el Código 5.2, donde se extienden las clases padre y se realizan vínculos a nuevas funciones de validación y cambio de nombre de archivos, recordando así la seguridad.

Se debe hacer hincapié a que esto únicamente agrega estos nuevos exhibits a la base de datos, y no aún a la vista de curadores. Para esto último falta agregar las clases asociadas a los formularios, para luego crear las clases asociadas a las previsualizaciones por parte los curadores, además de incorporar los formularios a los posibles elementos a añadir.

```python
class VideoExhibit(Exhibit):
    video = models.FileField(upload_to=rename_video_files, validators=[validator_video])

    def save(self, *args, **kwargs):
        self.exhibit_type = ExhibitType.objects.get(name='Video')
        super(VideoExhibit, self).save(*args, **kwargs)

class PDFExhibit(Exhibit):
    pdf = models.FileField(upload_to=rename_pdf_files, validators=[validator_pdf])

    def save(self, *args, **kwargs):
        self.exhibit_type = ExhibitType.objects.get(name='Pdf')
        super(PDFExhibit, self).save(*args, **kwargs)
```

Código 5.2: Clases VideoExhibit y PDFExhibit.

En lo referente a los formularios, y tal como se mencionó en **diseño de la solución**, corresponde a una clase que hereda de `forms.Modelforms`, que está encargada de asociar un formulario a un modelo en particular y además se considera la inclusión de una clase `Meta`, que se relaciona con qué campos tendrá el respectivo formulario. Para Unity se puede ver el Código 5.3, indicando el orden de los campos y cuál será la etiqueta respectiva en su parte visual (Figura 5.1). Finalmente, para los exhibits de tipo video y pdf el procedimiento es similar, teniendo como resultado lo que puede verse en el Código 5.4, donde se encuentra el nuevo formulario para el tipo pdf.

Ahora bien, teniendo los modelos y los formularios es necesario agregarlos a la vista del curador, tanto como opción de exhibit a añadir, así como también su respectiva previsualización. Para lo primero, seguimos el patrón planteado en el Código 4.8, sólo teniendo que heredar la clase `AddExhibitView` e indicando el tipo del exhibit y el formulario asociado (esto se puede ver en el Código 5.5), sin olvidar el agregar las nuevas vistas al URL Dispatcher como se ve en el Código 5.6 e incorporar dichas URLs en la template del menú de navegación. Con todo ello, tenemos como resultado las Figuras 5.2 y 5.3.

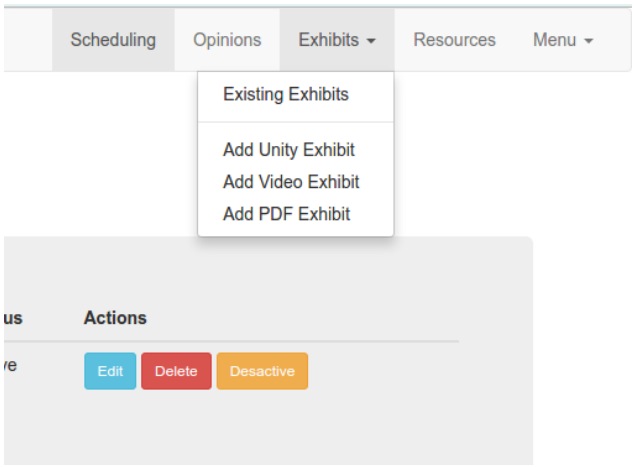

Figura 5.2: Menú de navegación incluyendo nuevos exhibits.

```python
class UnityExhibitMeta:
    fields = {'name', 'memory_to_allocate', 'data', 'javascript', 'memory'}
    labels = {'name': 'Name of exhibit',
              'memory_to_allocate': 'Unity TOTAL_MEMORY',
              'data': 'File exhibit.data',
              'javascript': 'File exhibit.js',
              'memory': 'File exhibit.mem'}

class UnityExhibitForm(forms.ModelForm):
    field_order = ['name', 'memory_to_allocate', 'data', 'javascript', 'memory']

    class Meta(UnityExhibitMeta):
        model = UnityExhibit
```

Código 5.3: Clases asociadas al formulario de UnityExhibit.

```python
class PDFExhibitMeta:
    fields = {'name', 'pdf'}
    labels = {'name': 'Name of exhibit',
              'pdf': 'PDF File'}

class PDFExhibitForm(forms.ModelForm):
    field_order = ['name', 'pdf']

    class Meta(PDFExhibitMeta):
        model = PDFExhibit
```

Código 5.4: Clases asociadas al formulario de PDFExhibit.

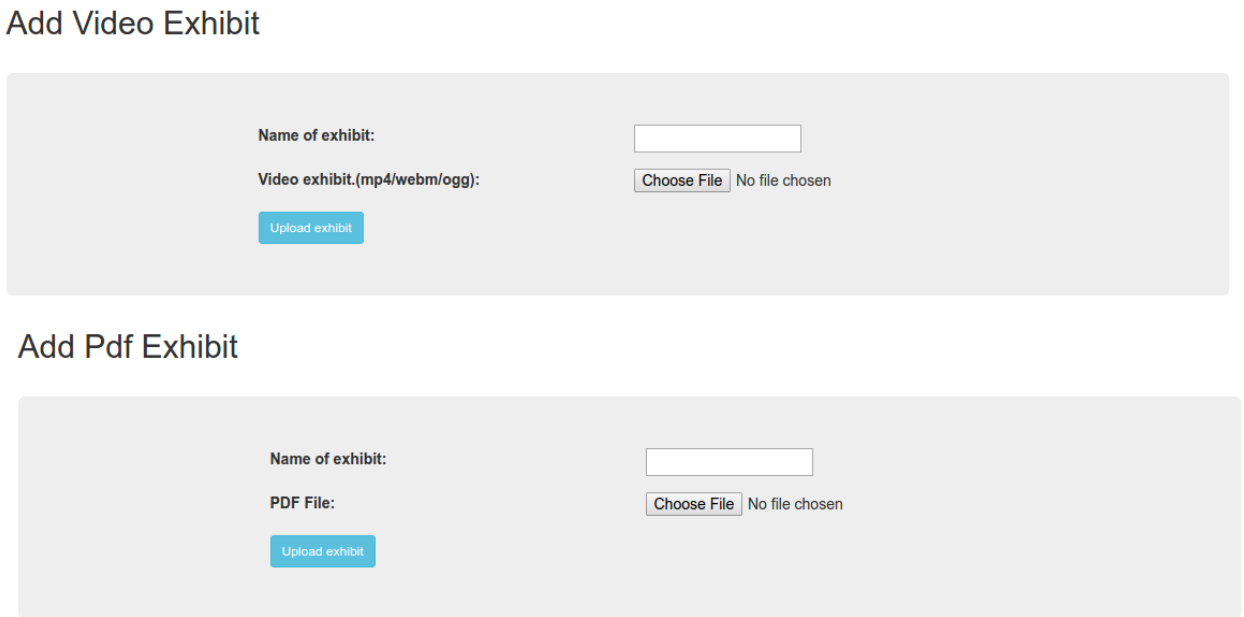

Figura 5.3: Formularios de subida de exhibits tipo Video y PDF.

```
class AddVideoView(AddExhibitView):
    form = VideoExhibitForm
    exhibit_type = 'Video'

class AddPDFView(AddExhibitView):
    form = PDFExhibitForm
    exhibit_type = 'Pdf'
```

Código 5.5: Nuevas clases de vista para los formularios de video y pdf.

```
urlpatterns = [...
    url(r'^add-unity-exhibit', AddUnityView.as_view(), name='new-unity'),
    url(r'^add-video-exhibit', AddVideoView.as_view(), name='new-video'),
    url(r'^add-pdf-exhibit', AddPDFView.as_view(), name='new-pdf'),
    ...]
```

Código 5.6: Asociación de vistas de añadir video y pdf en el URL Dispatcher.

Finalmente, queda agregar la previsualización de los nuevos exhibits. Para esto se tiene incluida una clase llamada `PreviewExhibitView`, la cual utiliza un diccionario llamado `MUSEUMS_TYPES` que contiene los métodos `get`, `delete` y una variable asociada a un `template` de cada tipo de exhibit. La idea tras esto es utilizar llamados del estilo `template = MUSEUM_TYPES[exhibit.exhibit_type.name]['template']` para obtener toda la información de un exhibit únicamente utilizando el `id` de este (recordar que cada exhibit sabe qué tipo es). En este caso, el diccionario `MUSEUMS_TYPES` se puede ver en el Código 5.7, lo cual resulta en una previsualización como la vista de la Figura 5.4.

```
MUSEUM_TYPES = {
    'Unity': {'delete': delete_unity_files, 'get': get_unity_data,
              'template': 'curator/preview_exhibits/preview_unity.html'},
    'Video': {'delete': delete_video_files, 'get': get_video_data,
              'template': 'curator/preview_exhibits/preview_video.html'},
    'Pdf': {'delete': delete_pdf_files, 'get': get_pdf_data,
            'template': 'curator/preview_exhibits/preview_pdf.html'}}
```

Código 5.7: Diccionario `MUSEUMS_TYPES` y los métodos para el tipo pdf.

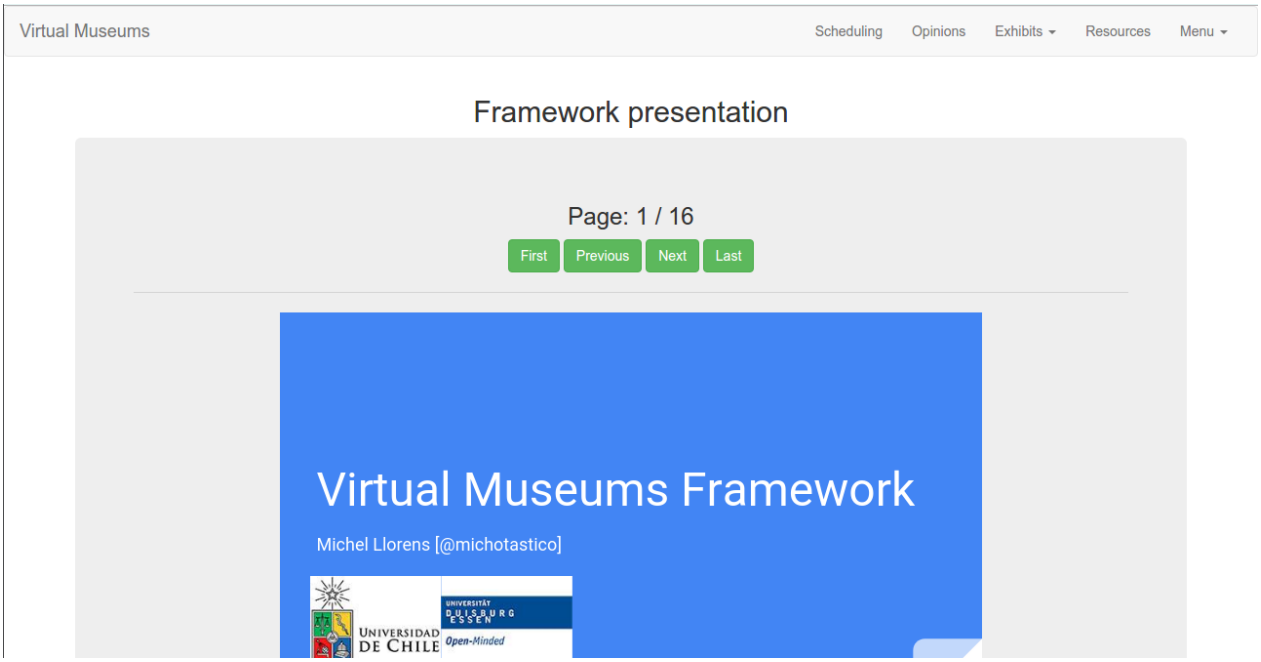

Figura 5.4: Previsualización de un PDFExhibit desde la vista de curadores.

5.1.2. Visualización visitante

Para la visualización por parte del visitante, se buscó un cambio visual más agradable para estos, intentando seguir la idea tras el sitio de PRASEDEC[1]. Para lograr aquello, lo primero que se realizó fue modificar visualmente la página de inicio, pasando de la Figura 5.5 a la Figura 5.6, donde se incluyó información acerca de PRASEDEC, se agregaron imágenes y se pasó a usar colores más llamativos. También se modificó la barra de navegación para que fuera siguiendo el scroll del visitante (en la misma figura se puede apreciar el icono del menú desplazado), el cual, al apretarlo, pasa a verse como en la Figura 5.7, donde además se incluyó un hipervínculo a la página de PRASEDEC. Todo este cambio estético se realizó cambiando la configuración de los archivos `CSS`, además de agregar contenido de forma estática a las `template`, pues al ya estar dividido en cuerpo, barra de navegación y pie de página, sólo es necesario ir cambiando el contenido específico de cada una.

[1] prasedec.dcc.uchile.cl

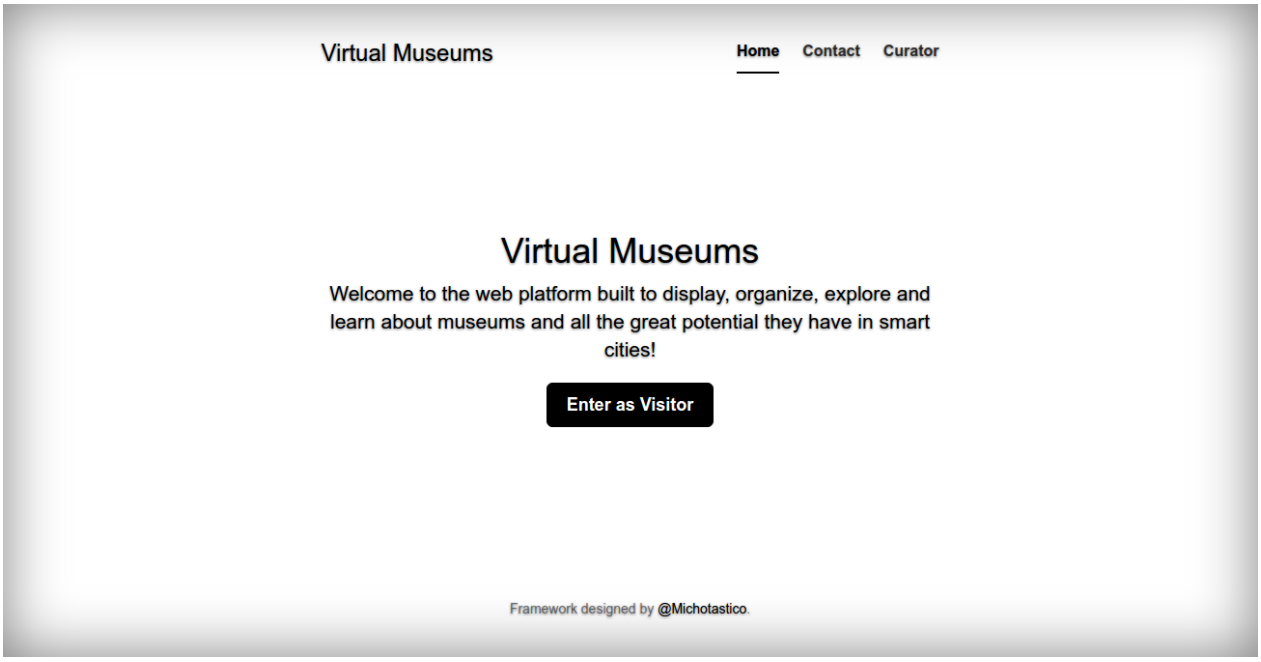

Figura 5.5: Página inicial Framework.

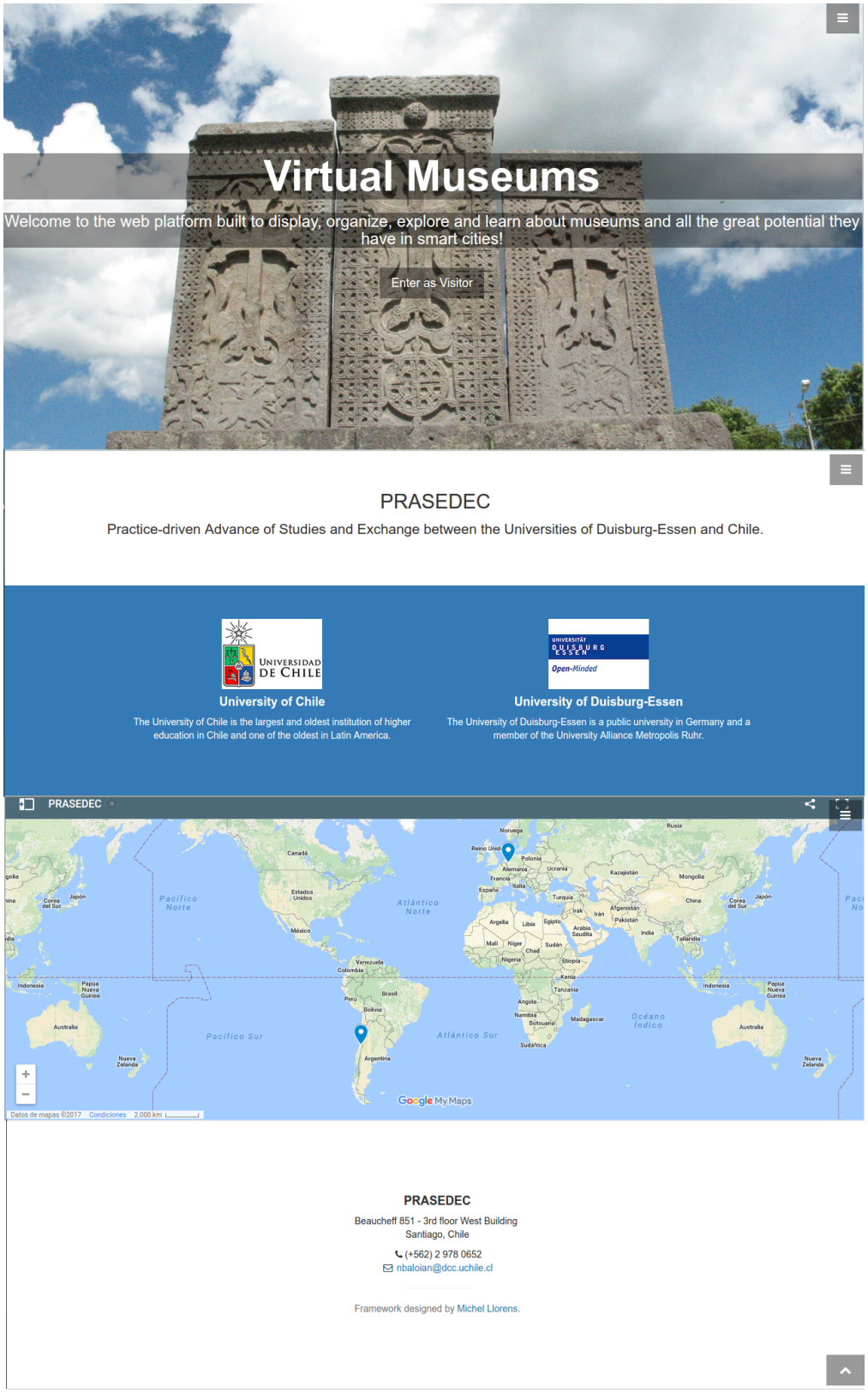

Figura 5.6: Página inicial Framework extendido.

Figura 5.7: Nueva barra de navegación visitante.

De dicha manera, esas mismas nuevas características pudieron agregarse a la vista de exhibiciones, donde observando la Figura 5.8, se puede ver que se comparte tanto la barra de navegación como el pie de página con la página principal. En el caso de estar en una visualización como tal, la barra de navegación puede detectar esta información y añadir la opción de enviar una opinión; es importante indicar que este análisis por parte de la vista se realiza durante el renderizado de la template utilizando el Código 5.8, que chequea la existencia de la variable `id` y, en relación a ello, permite un hipervínculo a la vista de enviar opinión, añadiendo la información de `id` como parámetro de tipo GET.

```
1 {% if id %}
2     <li>
3         <a href="/visitor/send-opinion?id={{ id }}" target="_blank">Send Opinion</a>
4     </li>
5 {% endif %}
```

Código 5.8: Utilización de ID para agregar el vínculo a *enviar opinión*

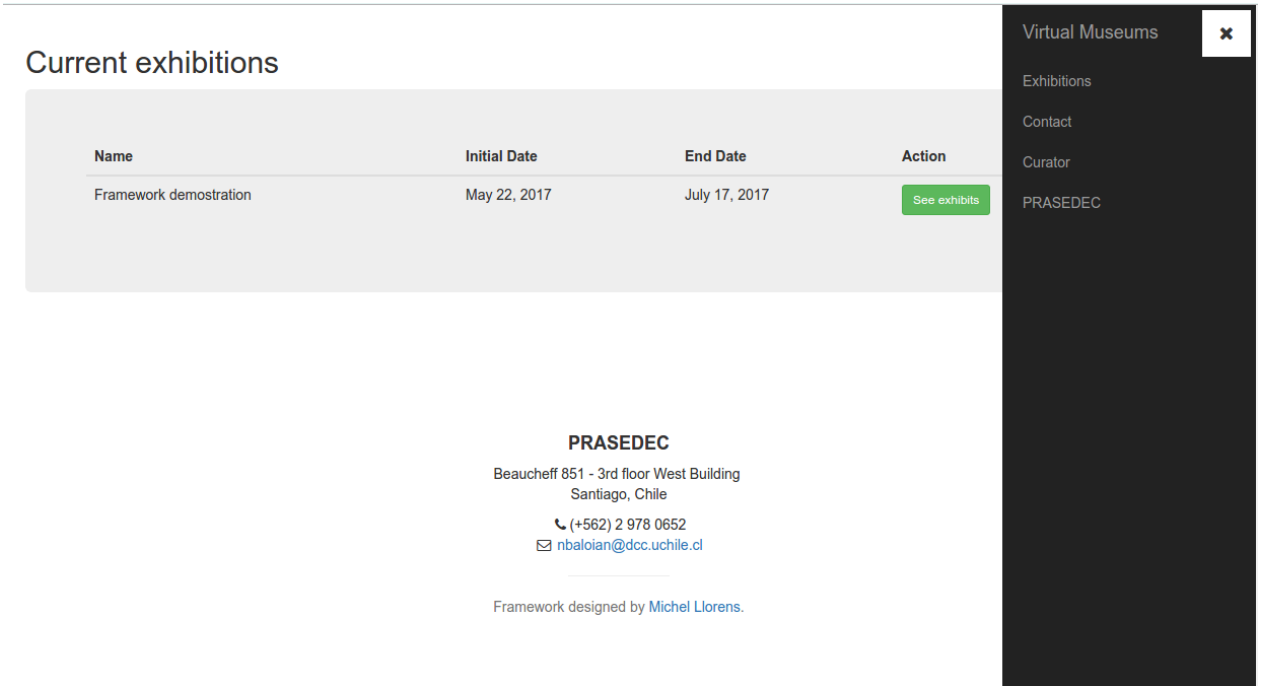

Figura 5.8: Nueva vista de exhibiciones.

Finalmente, la nueva ventana/pestaña abierta por el vínculo de nueva opinión nos lleva a la Figura 5.9, la cual posee el mismo menú de todas las demás vistas del visitante, a fin de mantener la misma usabilidad y no forzar a un nuevo aprendizaje.

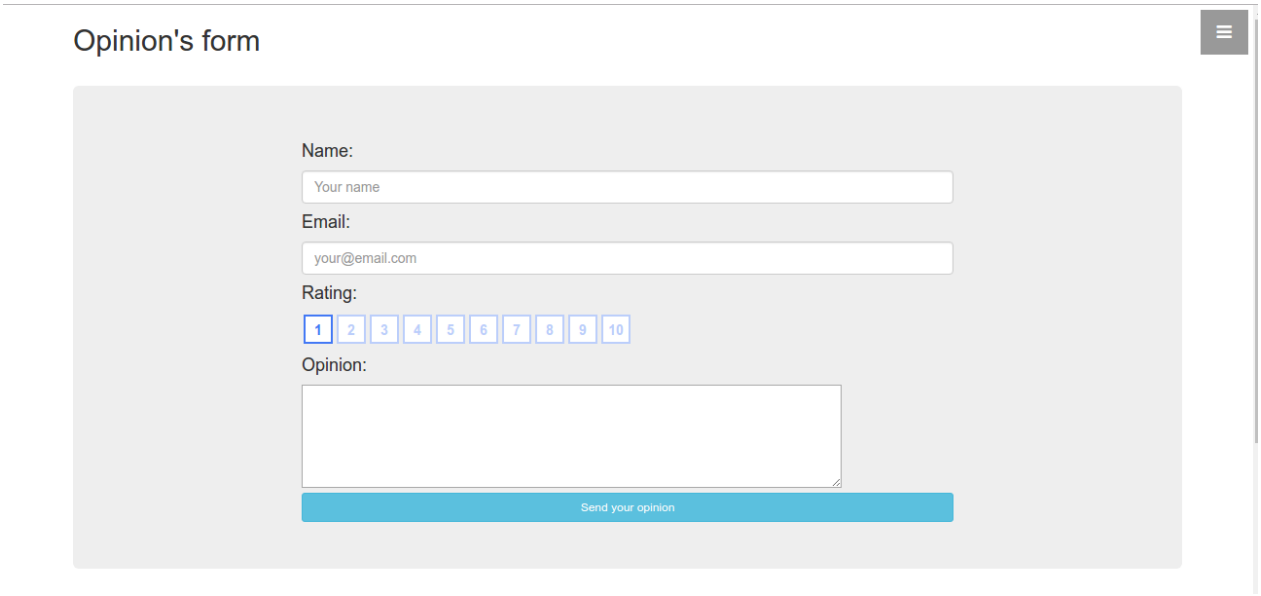

Figura 5.9: Formulario de opinión de visitantes.

Para la visualización por parte del visitante, se sigue exactamente la misma idea de la

previsualización de los curadores. Se tiene la clase `VisualizationView`, la cual obtiene los argumentos de cada visualización mediante el llamado `MUSEUM_TYPES[exhibit_type]['get'](exhibit)`, y el nombre de la respectiva template mediante `MUSEUM_TYPES[exhibit_type]['template']`. En consecuencia, el diccionario `MUSEUM_TYPES` del visitante sólo necesita el método `get` y la variable `template`. Es necesario indicar que perfectamente el método `get` del visitante y del curador pueden ser iguales; no obstante, el llamado a este método se realiza desde dos aplicaciones de Django distintas (`visitor` y `curator`), por tanto no sería correcto realizar llamados cruzados pues se rompería el encapsulamiento de las aplicaciones.

5.2. Deployment

El deployment, o despliegue, corresponde a la etapa en la cual un servicio (generalmente web) que está listo para su uso es puesto en marcha en un servidor. Por tanto, y de forma consecuente, en la presente sección se abordará el procedimiento de deployment del Framework, el cual fue probado tanto en Windows como en Linux; sin embargo, el procedimiento de dejar el servicio funcionando utilizando procesos en segundo plano o background sólo fue probado en Linux, específicamente en un servidor corriendo `Ubuntu 16.04.5 LTS x86_64`, debido a limitantes de recursos (no fue posible conseguir ni montar un servidor del tipo `Windows server`).

5.2.1. Configuración inicial

El Framework como requerimientos mínimos necesita que el servidor disponga de **Python** en versión `2.7.x`, la librería de Python **virtualenv** (que se puede instalar utilizando la instrucción de terminal `pip install virtualenv`) y la aplicación de **Node.js** llamada **Bower**. Una vez el servidor posee esos programas mínimos es necesario tener descargada una versión del Framework, la cual puede ser obtenida mediante la plataforma GitHub en la URL `https://github.com/Michotastico/Virtual-Museums-Framework`.

5.2.2. Configuración base de datos

Actualmente Django es compatible con la gran mayoría de bases de datos disponibles, particularmente las más conocidas **PostgreSQL**, **MySQL**, **SQLite** y **Oracle**. Por lo tanto, teniendo una base de datos y creando un espacio en este para la aplicación, podemos proceder a editar el archivo `settings.py` que se encuentra dentro del directorio `VirtualMuseumsFramework` del directorio raíz del Framework.

En dicho archivo se encuentra la variable `DATABASES`, la cual es un diccionario de bases de datos. Allí, por defecto, se encuentra una base de datos de **SQLite** que se crea inmediatamente al ejecutar la aplicación con todas las dependencias correctas, pero esta puede ser reemplazada comentando y/o eliminando la entrada `default` del diccionario, para luego crear una nueva entrada `default` con la base de datos correcta. Un ejemplo de esto se encuentra en el Código

5.9, donde se utiliza una base de datos **PostgreSQL**, que se conecta utilizando el `HOST` y su respectivo `PORT` con las credenciales `USER` y `PASSWORD`, y, por tanto, la base de datos puede encontrarse en un servidor remoto.

```
DATABASES = {
    # 'default': {
    #     'ENGINE': 'django.db.backends.sqlite3',
    #     'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    # }
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'virtual_museums_framework',
        'HOST': '127.0.0.1',
        'PORT': '5432',
        'USER': 'vm_db',
        'PASSWORD': '*****',
    }
}
```

Código 5.9: Configuración base de datos en Django

5.2.3. Dependencias

Teniendo ya los programas iniciales y la base de datos, faltan los elementos más importantes: **Django** y las **librerías estáticas** de Javascript. Para esto se asumirá que se tiene `virtualenv`, en caso contrario Django puede instalarse directamente, y se deberá seguir los pasos indicados en el Código 5.10 para crear una instancia de `virtualenv` y ejecutarla.

```
$ cd FrameworkLocation/..
$ virtualenv framework_venv
$ source framework_venv/bin/activate
```

Código 5.10: Configuración `virtualenv`

Una vez activa, el terminal pasará a verse de la forma `(framework_venv) user@server:$`, lo que significa que ya se está usando un ambiente virtual y, por tanto, pueden instalarse las dependencias (como Django), utilizando el comando `cd FrameworkLocation`, y luego `pip install requirements.txt`, que instalará las dependencias de Python.

Para las otras dependencias, se encuentran en el mismo directorio los archivos `.bowerrc` y `bower.json`, que corresponden a dónde se bajarán las dependencias y cuáles son estas, respectivamente. Para instalarlas sólo debe ejecutarse desde ese mismo directorio el comando `bower install`.

El último cambio a realizar debe ser en el archivo `settings.py` sobre las variables del tipo `WEBSITE_`, las cuales corresponden a las dependencias del servidor para enviar correos

electrónicos a fin de validar opiniones y los datos de contacto en caso de cualquier problema.

5.2.4. Ejecución

Teniendo ya las dependencias, es posible terminar las últimas configuraciones para la ejecución, que corresponden a la configuración de la base de datos desde el Framework y la carga de los primeros contenidos. Si bien el Framework incluye los archivos `firstRun.sh` y `firstRun.cmd`, que incluyen la instalación de las dependencias de Javascript y las configuraciones antes nombradas, se explicará paso a paso para una mayor comprensión.

Lo primero es ejecutar `python manage.py makemigrations`, lo cual traduce los modelos a entidades que, luego, serán pasadas a la base de datos mediante el comando `python manage.py migrate`, el cual cargará todos los modelos en la base de datos configurada y creará las respectivas tablas con sus atributos. Luego, se deberán cargar los datos mínimos, como lo son los tipos de exhibits y grupos de usuarios, lo cual se hace a través de los comandos `python manage.py loaddata fixtures/exhibit_types.json` y `python manage.py loaddata fixtures/user_groups.json`, respectivamente. El último paso corresponde a la creación de superusuario o administrador mediante el comando `python manage.py createsuperuser`, el cual tendrá acceso a la sección de administración de usuarios y grupos mediante una URL del estilo `http://framework.montado/admin`, reemplazando `framework.montado` con la dirección real donde está alojado el servicio.

Finalmente, para ejecutar el servidor se llama al comando `python manage.py runserver PORT`, donde PORT es el puerto por el cual saldrá el servicio. A continuación, se debe usar la combinación de teclas `control + z` que ejecuta la señal de sistema `SIGTSTP` que produce una suspención de la ejecución del comando llamado anteriormente, para luego utilizar el comando `bg` que manda a background el comando pausado y lo resume; de esta manera ,el servidor queda funcionando sin necesidad de tener abierto un terminal que ejecute directamente el servidor. Existen otras opciones como la utilización de `CRON` u otros procesos que se ejecutan automáticamente, pero la forma de lanzar el programa depende directamente el administrador de sistemas.

5.3. Ejemplo y guidelines

La extensión realizada como ejemplo de validación para el Framework actualmente se encuentra corriendo en las dependencias del Departamento de Ciencias de la Computación y es accesible mediante la URL `https://vm.dcc.uchile.cl/`. Ahora bien, queda ver las **guidelines** o directrices para crear contenido de calidad y aceptable a mostrar en el Framework.

En la extensión se incluyeron archivos de tipo vídeo y PDF, donde estos se encuentran acotados netamente por la cantidad de contenidos de estos; más específicamente, un vídeo puede perfectamente ser una película, no obstante, debe valorarse la calidad de las redes y las velocidades de descargas del público objetivo, por tanto, podría ser mucho mejor la utilización

de vídeos más cortos, o fraccionar contenido más grande en capítulos, representados por varios exhibits dentro de una misma exhibición, mejorando así el bienestar el visitante.

Respecto a los PDFs, con el fin de no depender de terceros en términos de herramientas, se utilizó una librería de Javascript para la visualización y manejo de estos documentos, renderizándolos en un `canvas`, y, en consecuencia, los documentos son dibujados más que simplemente incluidos. Esto conlleva a que no es posible realizar una copia del material expuesto de forma directa, lo que mejora la protección de recursos, pero, simultáneamente, se desconoce el tamaño máximo que la librería permite en archivos, desconociendo así la máxima calidad de renderizado. Por tanto, y nuevamente, se recomienda mesura al momento de elegir qué documentos se van a presentar y, dependiendo del caso, dividirlo en archivos más pequeños al igual que los vídeos.

Finalmente, los exhibits de tipo Unity son los más delicados, pues estos, a diferencia del resto, se basan en la utilización de **WebGL** y, por tanto, la posibilidad de conectarse a la **GPU** de cada computador y utilizar la memoria **RAM** del navegador web. Lo que directamente refleja el problema de que no se puede saber cuánta RAM tiene disponible un navegador web; aún más, no existe un estándar sino que depende directamente de las ambiciones de cada navegador, no obstante, cada uno pide una cantidad **máxima**. Es decir, si una aplicación de Unity ejecutándose de forma nativa en un computador utiliza 4Gb de RAM, es virtualmente imposible que un navegador web pueda ejecutarlo, pues, generalmente y con cierta sobrecarga, estos llegan a los 2Gb de utilización de RAM. Es cierto que los navegadores tienen un modo de alto consumo de RAM, pero esto requiere flags especiales como `-renderer-process-limit` para el navegador Chrome, que deben incluirse al momento de ejecutar el navegador y nada obliga a un usuario a usarla. Finalmente y en conclusión, lo más recomendable sería dividir una aplicación de Unity dependiendo de cuánta RAM utilice la aplicación.

En la Figura 5.10 se puede apreciar que en la aplicación de ejemplo se requiere aproximadamente 1Gb de RAM, lo cual, si bien es alto, se encuentra dentro de lo aceptable aunque puede ser demasiado para algunos navegadores. Unity en sus últimas versiones está incluyendo versiones comprimidas de sus archivos con el formato `gz`, que permiten reducir el problema de carga y manipulación de los componentes de Unity en web; sin embargo, no se probó su correcta utilización con Django debido a temas de compatibilidades de tecnologías.

En consecuencia, y volviendo a la idea tras PRASEDEC, en caso que se tenga una gran exhibición en Unity, lo ideal sería separar el programa en partes más pequeñas para evitar tener problemas con RAM, como, por ejemplo, separar un museo en tres exhibits correspondientes a primer piso, segundo piso y jardín, o en caso de que se requiera modelos más pesados o exigentes en uso de memoria, ir separando por pieza. De hecho, parte de las decisiones de diseño de permitir que una exhibición permitiera múltiples exhibits viene de poder tener múltiples cuartos y reducir el uso de memoria individual de cada uno.

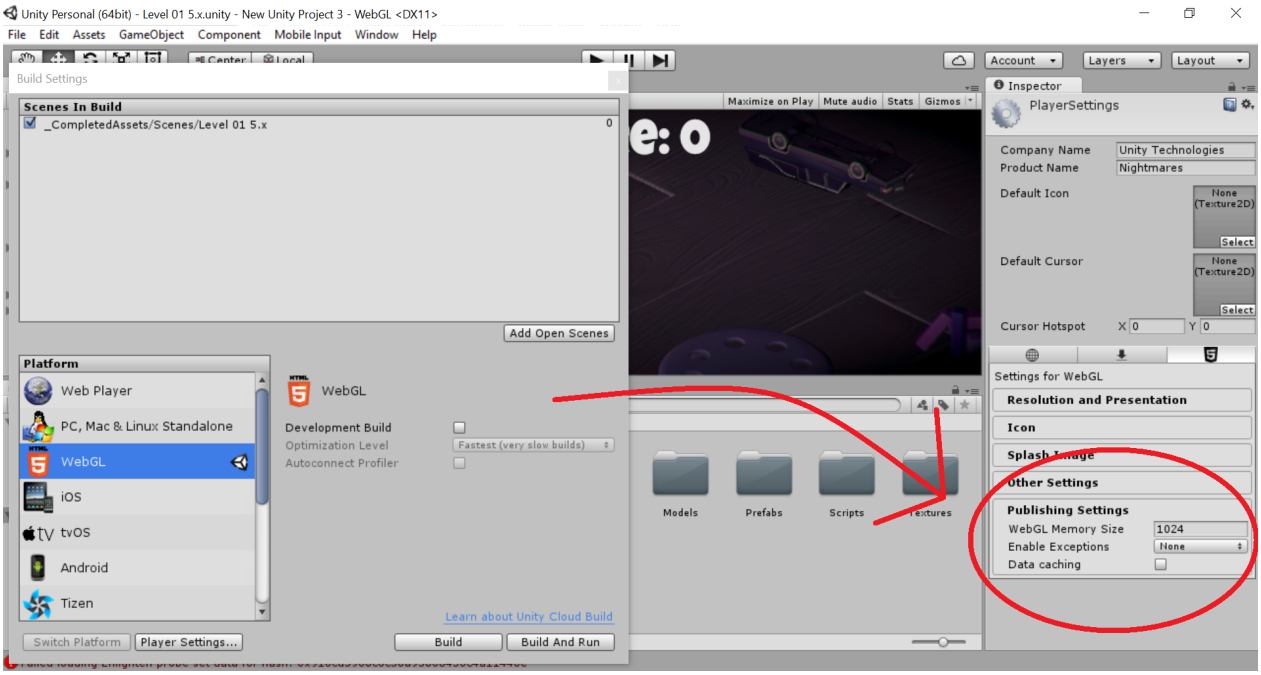

Figura 5.10: Opciones de compilación de Unity.

5.4. Feedback humano y usabilidad

Para esta última sección se decidió realizar consultas tanto a personas capacitados en ciencias de la computación como otros que no, a fin de comprobar la usabilidad de la versión extendida del Framework, en el área de curadores y en el área de visitantes, y con ello comprobar si se cumplieron los objetivos o no.

Los resultados de las consultas resultaron favorables, demostrando que las interfaces son claras y llamativas, además de ser fáciles de utilizar, que es justamente lo buscado. En aspectos más técnicos se dio razón en la facilidad de extensión, no sin antes realizar comentarios tales como reutilizar la idea de diccionario para otras funcionalidades, o que en vez de diccionarios se utilicen las clases abstractas, tal y como se postuló en diseño, y además obligar a cumplir con las firmas para que el sistema sea aún más robusto.

Un último feedback humano correspondió al realizado en una exposición a la sociedad Smithsoniana por parte del profesor Nelson Baloian acerca del presente proyecto, teniendo una buena aceptación, además de ser considerada más como una herramienta a usar que una rareza, lo que es justamente lo esperado al buscar que el Framework pueda ser usado por distintas entidades y principalmente por museos o encargados de brindar y preservar la cultura.

Capítulo 6

Conclusión

En este capítulo se realizará una recapitulación de los objetivos del Trabajo de Título con respecto a la validación de este, pudiendo así comprobar qué se obtuvo, qué objetivos se cumplieron, qué pudo realizarse de mejor manera y qué trabajo queda propuesto para extender la vida útil de la solución.

6.1. Resultados obtenidos y objetivos cumplidos

Como resultados se logró efectivamente diseñar e implementar un framework web basado en Django que permitiera la gestión y visualización de museos virtuales; de dicha manera, se logran cumplir los criterios de aceptación del producto final y además resolver el problema planteado, logrando también así cumplir con todos los requisitos de usuario y los objetivos, tanto principales como específicos. Sobretodo lo relacionado a extensibilidad y mantenibilidad, como se pudo observar en **Validación de la solución** al poder confeccionar un sitio web específico con base en el Framework, el cuál además heredaba la seguridad informática y la compatibilidad con navegadores web de escritorio[1]. Lo que permitió que se pudiera montar con acceso a través de Internet, a modo de ejemplo público de la aplicación, demostrando así que el Framework realmente puede ser utilizado y ser así una herramienta que permita difundir cultura y ser ajustado a las necesidades particulares de las organizaciones que planeen utilizarlo.

6.2. Análisis crítico de los resultados

Ahora bien, pese a que se cumplieron todos los objetivos, se cree que estos se pudieron subdividir más, con el fin de probar cosas más específicas y exigirle un poco más al software. Particularmente, se cree que se debieron incluir pruebas de carga o esfuerzo para ver la

[1]Se comprobó con Google Chrome v60, Firefox v47, Microsoft Edge v38, Internet Explorer v11 y Opera v46.

resiliencia del Framework y, de dicha manera, poder analizar aún más profundamente las posibilidad de tener crecimiento vertical, y no sólo horizontal, al momento de aumentar la cantidad de usuarios de la aplicación. Así también, se cree que no basta con pruebas específicas en un conjunto acotado de computadores para realizar ejecuciones de la plataforma, pues, por ejemplo se probó con computadores con sistema operativo Linux, pero perfectamente se pudo haber incorporado el resto de los sistemas Unix, como lo son los productos de Apple; no obstante, aquello se encontraba fuera del alcance logístico de la memoria, lo que ocurre de manera similar con la idea de realizar pruebas con Windows server.

6.3. Trabajo futuro

Como trabajo futuro se plantean nuevas features a incluir dentro del Framework base, como lo es la incorporación de un canal de comunicación directo entre usuarios de tipo curador, ya sea directamente o entre grupos de trabajo; como base se planteó que el equipo de curadores se reunía físicamente, o bien tenía canales de comunicación como correo electrónico, pero agregaría considerable valor el hecho de no necesitar nada más que la misma plataforma para dejar mensajes y, de dicha manera, aumentar la productividad.

También queda propuesto volver a realizar una refactorización de código en lo referente a la utilización de diccionarios, para trabajar sobre distintos tipos de Exhibits, pues actualmente se utilizan los de Python de forma directa y como variables globales, pero perfectamente éstos podrían ser clases individuales de carácter Singleton y de dicha manera hacer el código aún más claro y extensible. De igual manera se propone incluir pruebas de carga o esfuerzo para tener datos concretos de la capacidad máxima que aguanta el Framework como base sin necesidad de crecer en recursos físicos.

Una sugerencia planteada para el trabajo futuro es la consideración de migrar la aplicación desde Python `2.7.x` a Python `3.x.x`, ya que se anunció que, a partir del año 2020, ya no se le dará soporte oficial a la versión `2.7` de Python y se terminará la migración hacia Python `3`, por tanto es un gran punto a considerar para mantener estabilidad y soporte a lo largo de los años.

En términos de investigación, queda propuesto un estudio de viabilidad para incorporar realidad aumentada a las exhibiciones, a fin de brindarles una mayor inmersión. Así mismo, se podría ver el impacto y alcance que tendría la utilización de este mismo Framework en las exhibiciones físicas como material de apoyo, para brindar información extra y/o complementaria.

Finalmente, se espera que este proyecto siga vivo a través de los años independientemente de la colaboración de su autor original, y que siga mejorándose y facilitando cada vez más la posibilidad de que cualquier entidad pueda utilizar el Framework; se espera que sea útil para exponer información y cultura sin necesitar tantos conocimientos computacionales, o bien sin tener que invertir grandes montos de dinero que no siempre se poseen.

Bibliografía

[1] Mikio Aoyama et al. New age of software development: How component-based software engineering changes the way of software development. In *1998 International Workshop on CBSE*, pages 1–5, 1998.

[2] D. Biella, W. Luther, and D. Sacher. Schema migration into a web-based framework for generating virtual museums and laboratories. In *Virtual Systems and Multimedia (VSMM), 2012 18th International Conference on*, pages 307–314, Sept 2012.

[3] Daniel Biella, Wolfram Luther, and Nelson Baloian. Virtual museum exhibition designer using enhanced arco standard. In *Chilean Computer Science Society (SCCC), 2010 XXIX International Conference of the*, pages 226–235. IEEE, 2010.

[4] Ivan Blekanov, Anggai Sajarwo, and Sergey Sergeev. Web-based framework and search engine analytics for thematic virtual museums. IEEE Proceedings of the 2015 4th International Conference on Interactive Digital Media (ICIDM), IEEE, 2015.

[5] Mohamed Fayad and Douglas C Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, 1997.

[6] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung Tsai. Web application security assessment by fault injection and behavior monitoring. In *Proceedings of the 12th international conference on World Wide Web*, pages 148–159. ACM, 2003.

[7] Ralph E Johnson and Brian Foote. Designing reusable classes. *Journal of object-oriented programming*, 1(2):22–35, 1988.

[8] Chairi Kiourt, Anestis Koutsoudis, and George Pavlidis. Dynamus: A fully dynamic 3d virtual museum framework. *Journal of Cultural Heritage*, 2016.

[9] Marcus Eduardo Markiewicz and Carlos JP de Lucena. Object oriented framework development. *Crossroads*, 7(4):3–9, 2001.

[10] Jakob Nielsen. Usability 101: Introduction to usability, 2003.

[11] Jonathan W Palmer. Web site usability, design, and performance metrics. *Information systems research*, 13(2):151–167, 2002.

[12] Dirk Riehle and Thomas Gross. Role model based framework design and integration. In

ACM SIGPLAN Notices, volume 33, pages 117–133. ACM, 1998.

[13] Dafydd Stuttard and Marcus Pinto. *The web application hacker's handbook: finding and exploiting security flaws*. John Wiley & Sons, 2011.

[14] Guido Van Rossum. An introduction to python for unix/c programmers. *Proc. of the NLUUG najaarsconferentie. Dutch UNIX users group*, 1993.