Universidad Simón Bolívar
CI-3715
Abril-Julio 2023

Diseño de Volunteer Work Manager

Equipo:
Team 2 - Chigüirongos
Integrantes:
Pedro Rodríguez 15-11264
Gabriela Panqueva 18-10761
Kenny Rojas 18-10595
Simón Puyosa 18-10717

Introducción

El siguiente informe describe la situación actual de las actividades del proyecto Calendar Manager for Volunteer Work, con el objetivo de comunicar la información relevante del mismo como el progreso acumulado, la metodología usada de trabajo, el problema planteado así como su análisis y modelo de solución.

Para lograr la mejor comprensión sobre el contenido del siguiente informe se debe tener conocimiento en el área de análisis, modelado y desarrollo de software, más específicamente se necesita noción en el área de metodología ágil de desarrollo aplicada al marco de trabajo SCRUM y su terminología, así como también el entendimiento de los diagramas UML, arquitecturas de diseño de software, herramientas de diseño como Figma y por último, se necesita conocimiento de los frameworks utilizados y su terminología, los cuales son React JS, Spring Boot y el ORM Hibernate y la plataforma Docker.

El marco de trabajo seleccionado fue SCRUM, el cual se basa en el aprendizaje continuo y la adaptación frente a escenarios fluctuantes. Por otro lado, se siguió la metodología ágil que se centra en un conjunto de valores y técnicas aplicadas en ciclos de trabajos cortos, con el objetivo de lograr la mayor eficiencia en los procesos de entrega de proyectos. En este sentido, se ha trabajo sobre los siguientes elementos:

- Sprints: Iteraciones en periodos de tiempo cortos en los que se trabaja para completar una cantidad de trabajo establecida. En cada Sprint realizado se ha completado un 20% del total del proyecto en relación a la entrega anterior en un tiempo de 2 semanas, con excepción del primero que fue un 10% en un plazo de una semana.

- Ceremonias: Reuniones en las cuales el equipo se comunica para aportar información relevante del proyecto. Se clasifica en 4:

    - Sprint Planning: Se realiza al inicio del Sprint, se discute la importancia del Sprint, los objetivos a alcanzar en mismo y el plan para lograrlo.

    - Daily Scrum: Se realizan los días lunes, miércoles, viernes y sábado, son breves reuniones frecuentes en las cuales se asignan tareas a cada integrante y también se informa el progreso de las mismas.

    - Sprint Review: Reuniones realizadas al final de cada Sprint, en las cuales se inspecciona el estado del producto por parte del cliente.

    - Retrospectiva: Reuniones bisemanales en las cuales se discuten los aprendizajes y cosas por mejorar, se realizan los domingos a las 3:00 PM luego de la entrega.

A cada ceremonia se le realiza su respectiva minuta, las cuales se pueden encontrar en la siguiente carpeta dentro de la carpeta del Sprint correspondiente.

- Product backlog: Es una lista ordenada por prioridad de las tareas pendientes por desarrollar y contiene historias de usuario que explican de manera breve las funcionalidades a desarrollar en cada tarea. Para el manejo de este se creó un Team Space en la plataforma Notion, el cual puede accederse desde el siguiente enlace.

Tomando en cuenta lo anteriormente expuesto se puede abordar el problema a resolver: Se desea una aplicación de gestión de trabajos de voluntariado que permita a los usuarios tener un rol con sus funcionalidades correspondientes. En este sentido, los roles permitidos son:

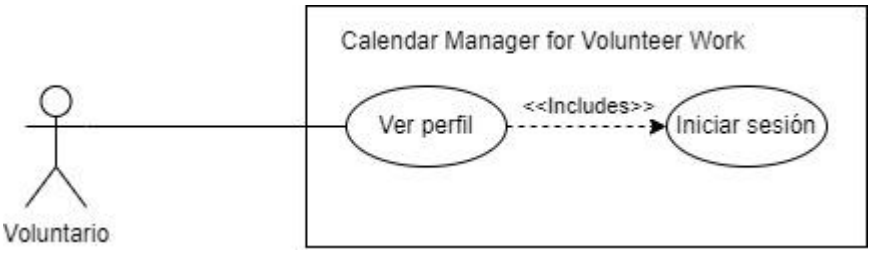
- Voluntario: Tiene la posibilidad de encontrar y postularse a los trabajos que más se adecuen a sus preferencias de horarios y áreas de trabajo para realizarlos.

- Proveedor: Tiene la posibilidad de proveer trabajos de voluntariado, así como modificarlos, borrarlos y aceptar voluntarios previamente postulados para que los lleven a cabo.

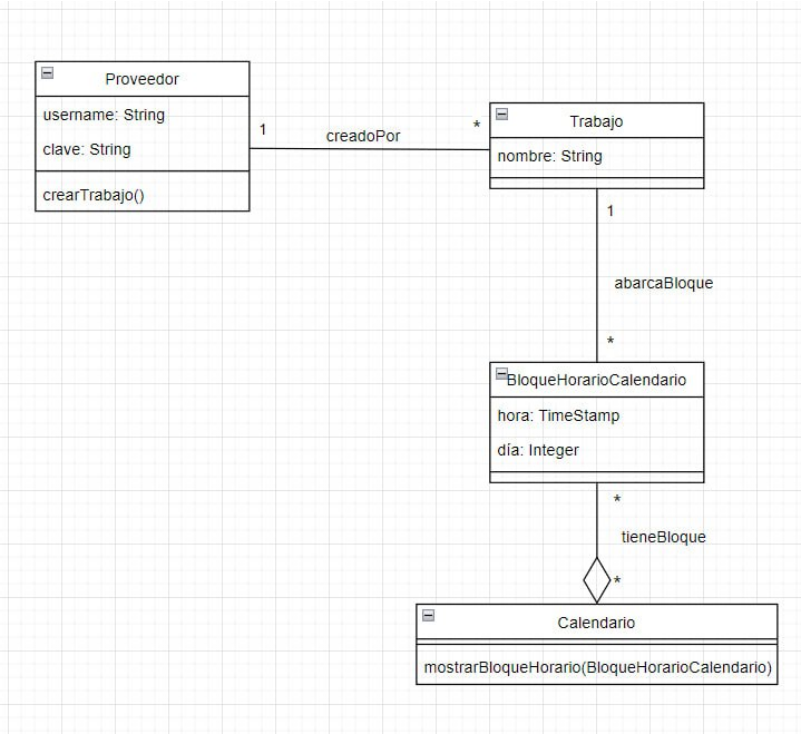
- Administrador: Es un tipo especial de usuario que tiene permitido gestionar las cuentas de los usuarios.

Los requerimientos detallados se encuentran en el siguiente documento.

Para la resolución del problema propuesto, se realizó un modelo de análisis con el lenguaje UML, el cual contiene los casos de uso de las funcionalidades de la aplicación, por ejemplo, un caso de uso de la funcionalidad ver perfil para voluntarios fue modelado de la siguiente manera:



Los casos de uso también se describen de manera textual, en el cual se explican con más detalle. Por otro lado, también se realizaron diagramas de clase con el lenguaje UML para describir las clases que participan en cada caso de uso del modelo análisis de la solución, por ejemplo, un diagrama de las clases que participan en el caso de uso crear trabajo para proveedores:



Los diagramas expuestos anteriormente se pueden encontrar en el siguiente enlace (únicamente con permisos de lector) junto a las ceremonias de SCRUM realizadas.

El informe se divide en dos grandes partes, BackEnd y FrontEnd, que a su vez se dividen en múltiples subpartes. Cada una de estás secciones contendrán una explicación de las responsabilidades resultantes de esta división, y adicionalmente una descripción de todos los objetos pertenecientes a estas divisiones. Para finalizar, se tomará como ejemplo una petición HTTP que pueda realizarse en el sistema, y se explorará secuencialmente su ciclo de vida, empezando desde el FrontEnd.

El FrontEnd se encarga de la visualización de interfaces de usuario y la interacción de estas con el BackEnd. El diseño visual así como posibles interacciones con el usuario, se realiza en la herramienta Figma, la cual permite al equipo realizar prototipos que son analizados e implementados, los diseños se pueden visualizar aquí. Luego, como tecnología principal en este, se utiliza JavaScript junto a la librería React, siendo esta última útil para la construcción de interfaces reactivas e interactivas de usuario. Adicionalmente nos apoyamos en los paquetes de React Material UI, el cual nos provee de elementos visuales e interactivos ya hechos, y React Router, el cual se encarga del enrutamiento y renderizado de distintas vistas según la ruta actual de un usuario.

El BackEnd se encarga de la lógica de negocio. Como tecnología principal en este, se utiliza Java junto al framework Spring Boot, y PostgreSQL para la base de datos utilizada en el sistema. Adicionalmente nos apoyamos en múltiples paquetes y tecnologías de Spring Boot, que son: Spring Security para la autenticación y autorización de usuarios, e Hibernate, un ORM para el manejo del modelo entidad-relación implementado en el proyecto.

En el diseño del sistema se deben considerar las condiciones de borde, que son situaciones excepcionales o límites que pueden ocurrir durante la ejecución. Algunas condiciones de borde que se pueden considerar son: usuarios que intentan acceder a rutas no autorizadas, usuarios que ingresan datos inválidos en formularios, errores de conexión con la base de datos, entre otros. Estas condiciones deben ser manejadas adecuadamente en el diseño del sistema para proporcionar una experiencia de usuario robusta y segura.

Por otro lado, también se pueden utilizar diferentes patrones de diseño para abordar problemas comunes y mejorar la modularidad, escalabilidad y mantenibilidad del código. Algunos patrones que se pueden aplicar en este contexto son:

- Patrón MVC (Modelo-Vista-Controlador): Se utiliza para separar las responsabilidades del modelo de datos, la lógica de negocio y la interfaz de usuario.

- Patrón Singleton: Se puede utilizar para asegurar que solo exista una instancia de ciertos objetos en el sistema.

- Patrón DAO (Data Access Object): Se puede utilizar para abstraer el acceso a la capa de persistencia de datos y proporcionar métodos de alto nivel para interactuar con la base de datos.

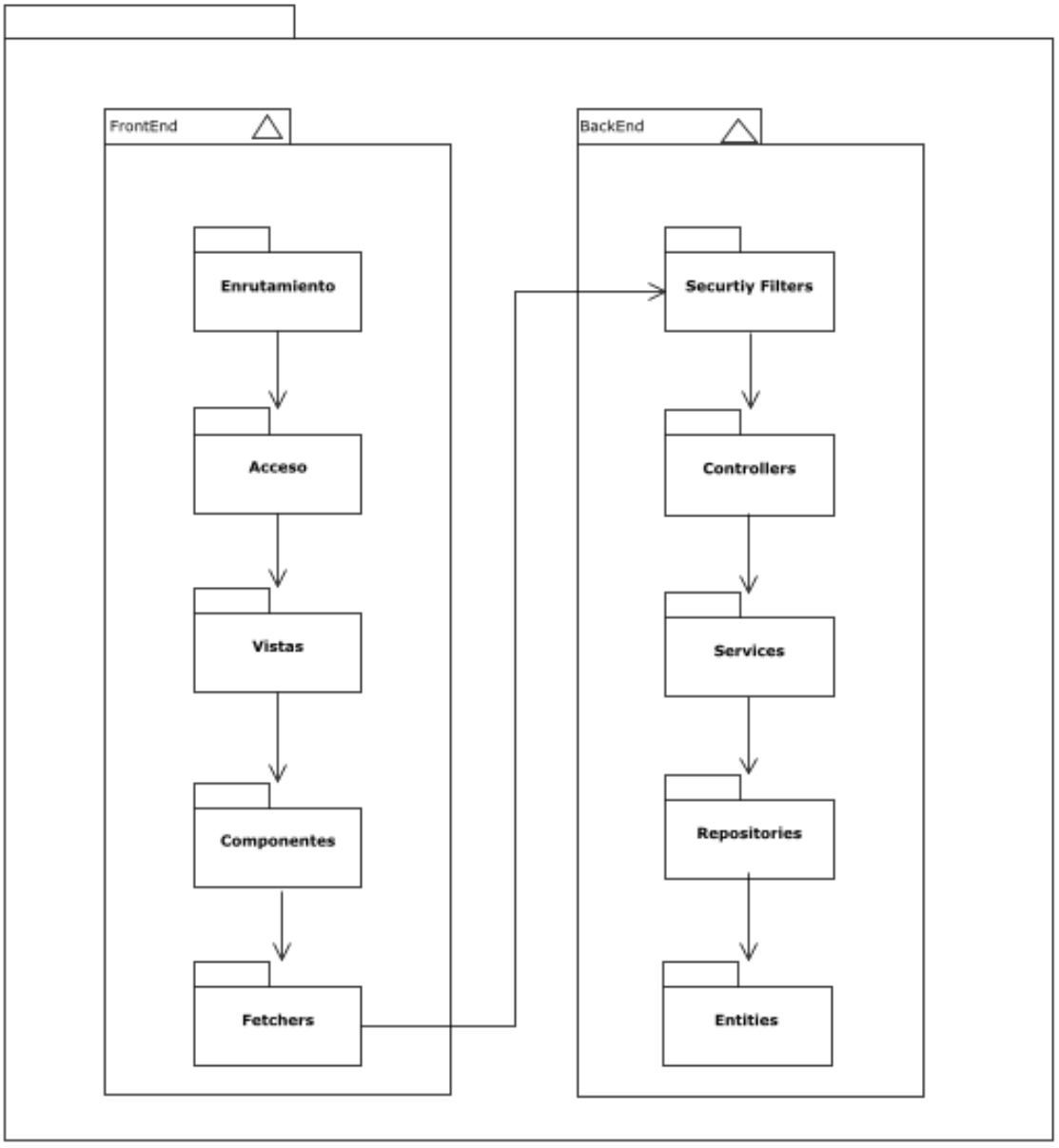
- Patrón Façade: Permite disminuir la complejidad del sistema al dividir el mismo en subsistemas, y limitar la comunicación y dependencia entre ellos.

La implementación de todo lo expuesto anteriormente se empaqueta en la plataforma Docker, la cual facilita el despliegue de la aplicación sin necesidad de tener que instalar dependencias del proyecto en las máquinas que ejecuten la misma. Docker despliega las aplicaciones con el uso de contenedores de imágenes, en las cuales se encuentran los frameworks a utilizar. En nuestro caso, se tienen tres imágenes, una para el FrontEnd con ReactJS, otra para la base de datos de Postgres 14 y por último, una para el BackEnd con Spring Boot.

Índice

Diseño del sistema

Para abordar el diseño del sistema de voluntariado, se tomó la decisión de dividir este en capas, específicamente dos: BackEnd y FrontEnd, cada una con sus propias subcapas. A continuación, se presenta un diagrama UML de paquetes donde se pueden apreciar cada una de las capas en el proyecto:



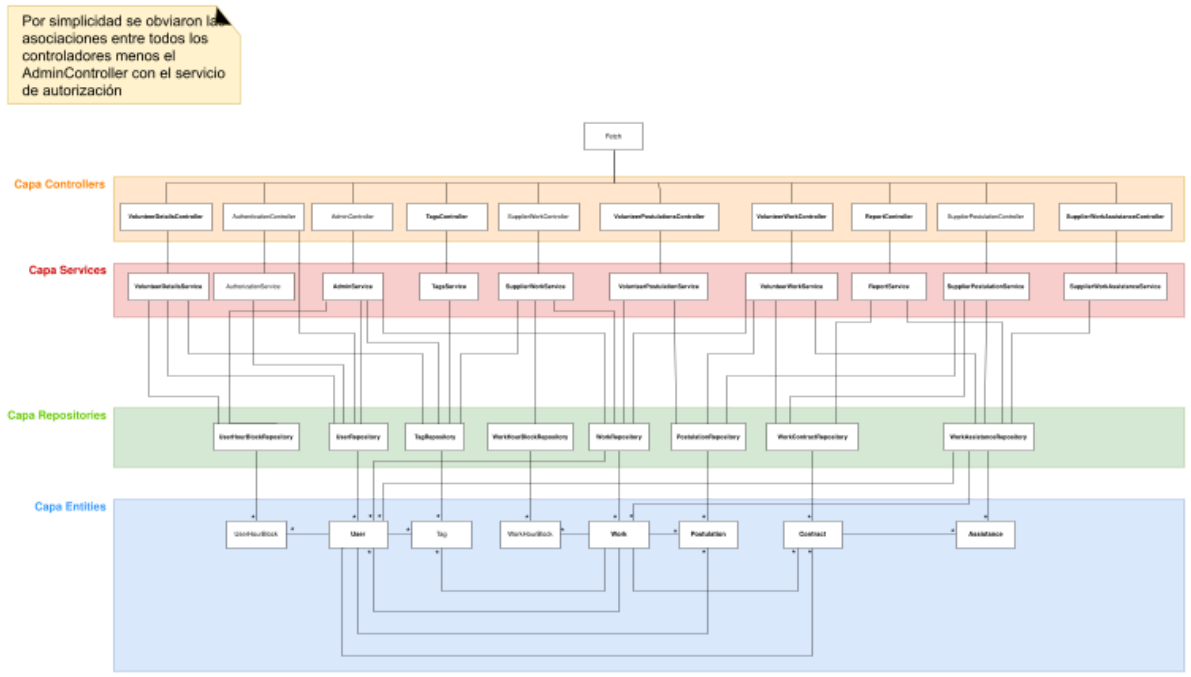
Esta separación se realiza al tomar en cuenta el concepto de segregación de responsabilidades, el cual consiste en la separación de un sistema en distintos componentes o partes, cada uno con responsabilidades de las cuales están encargados. La principal razón que

nos llevó a realizar esto es que la separación en capas nos permite el desarrollo en partes independientes del proyecto, lo cual implica que se puede implementar progresivamente, siguiendo un orden específico, marcado por el flujo de ejecución de la aplicación.

Diseño del modelo solución

Se diseñó un modelo de solución del problema planteado con el uso de diagramas de clases, el cual refleja los tipos de clases existentes en el sistema, así como las relaciones entre los mismos. Este modelo se divide en capas, las cuales contienen un tipo de clase y se muestra su relación con otro tipo de clase que corresponde a una capa inferior. A continuación se presentará el modelo, desde el nivel más alto hasta el más bajo.

Primeramente, se tiene el siguiente diagrama general, el cual muestra todas las capas existentes así como las clases del sistema y las relaciones entre ellas:

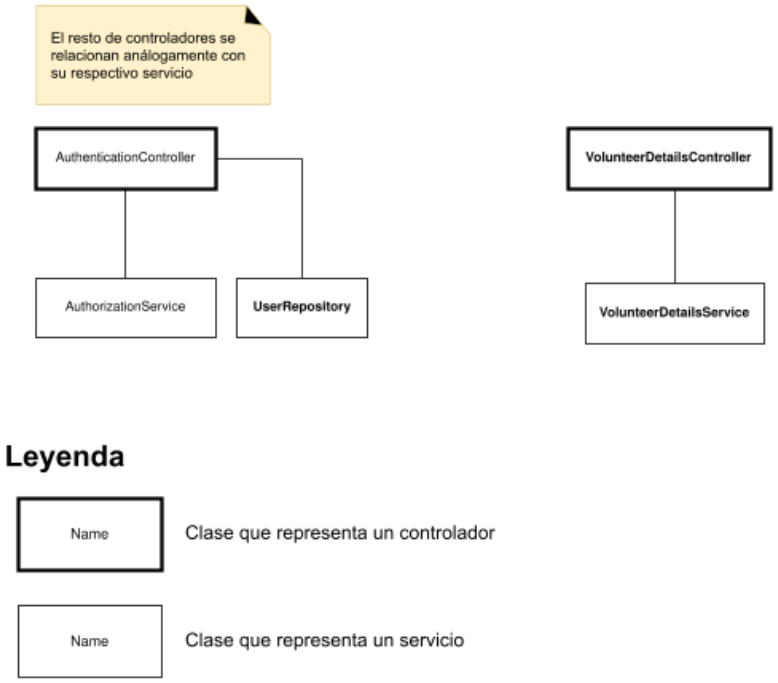


Para una mejor visualización, visite el siguiente enlace.

Luego de exponer el diagrama general, se mostrarán las capas en las que se divide el diagrama de solución.

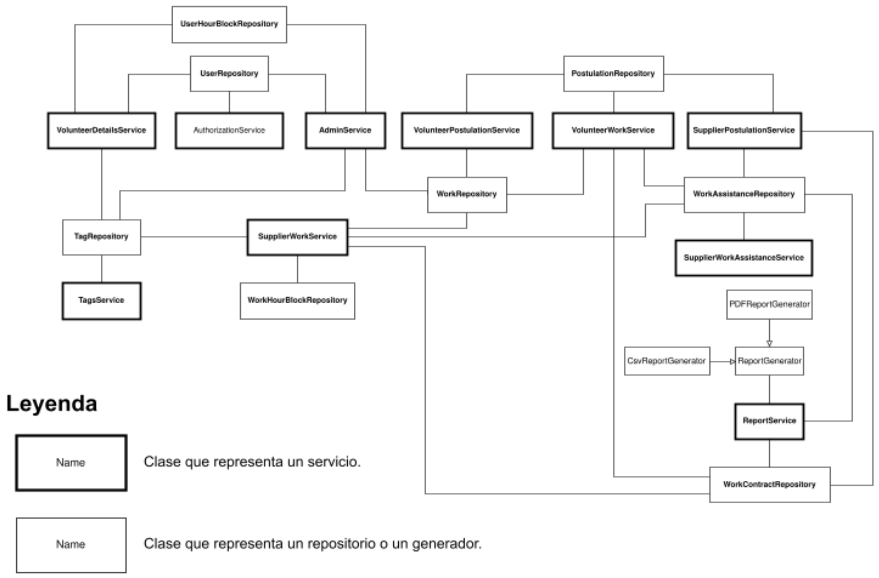
En el nivel más alto se encuentra la capa de los controladores. Estas clases son las que reciben las peticiones directamente del FrontEnd mediante los “fetchs” y se comunican con la clase de servicio correspondiente para procesar la solicitud y devolver la respuesta obtenida. Al comienzo de la siguiente página se encuentra el diagrama que representa las clases de controladores y sus interacciones con los servicios correspondientes. En el mismo

se muestra como a cada controlador le corresponde un único servicio, a excepción de AuthenticationController al que le corresponde el servicio AuthorizationService pero tiene comunicación directa con la clase de repositorio UserRepository.



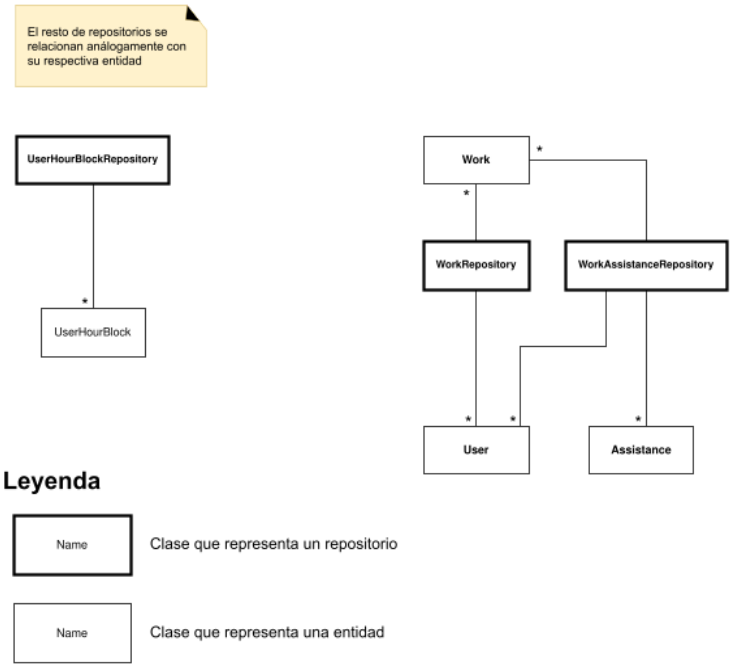
Para mejor detalle de la clase de controlador, vea el anexo 1 figura 1.

Por debajo de esta capa, se encuentra la capa de los servicios, quienes hacen de fachada al recibir las peticiones exteriores y manejar internamente la lógica para responder a estas solicitudes. Se muestran las interacciones de los servicios con las clases de repositorios en el siguiente diagrama:



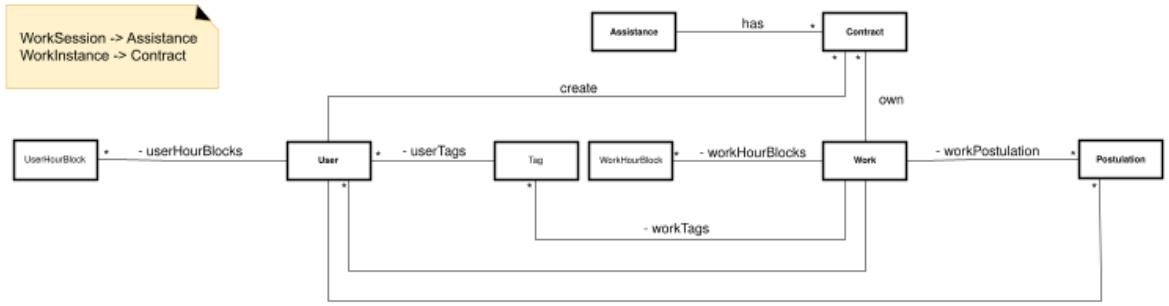
Para mejor detalle de la clase de servicio, vea el anexo 1 figura 2 y 6.

El siguiente nivel corresponde a la capa de repositorios y muestra cómo se relacionan estas clases con las clases de entidades correspondientes. Los repositorios se encargan de efectuar las operaciones y métodos sobre sus respectivas entidades, las relaciones entre repositorios y entidades se muestran en el siguiente diagrama:



En el diagrama anterior se muestra la relación de cada repositorio con su respectiva entidad, sin embargo, existen dos excepciones a esta norma, los repositorios WorkRepository y WorkAssistanceRepository que no solo se relacionan con las entidades Work y Assistance respectivamente, sino que también tienen relación con la entidad User. Para mejor detalle de la clase de repositorio, vea el anexo 1 figura 3.

Por último, en el nivel más bajo se encuentra la capa que representa las entidades de negocio y las relaciones existentes entre las mismas.



Para mejor detalle de las clases de entidades de negocio, vea el anexo 1 figura 4.

Las siguientes secciones del informe estarán dedicadas a explorar las capas de BackEnd y FrontEnd junto con cada una de sus subcapas.

Capa BackEnd

Esta es una de las dos capas más generales del proyecto, se encarga de controlar todas las reglas del negocio de nuestra aplicación y también de almacenar la información necesaria para el funcionamiento del sistema, así como de brindar un subsistema de autenticación a los usuarios. Utiliza un sistema de peticiones para comunicarse con la capa del FrontEnd, esto quiere decir que si esta última necesita interactuar con la capa del BackEnd, esta le envía una petición al BackEnd para realizar dicha interacción, el cual le responde con la información solicitada y un status exitoso, o fallido en caso de haber ocurrido algún problema con la petición.

Se decidió subdividir esta capa en otras subcapas con responsabilidades más específicas, que son: Security Filters, Controllers, Services, Repositories y Entities.

Subcapa Security Filters

En el sistema de voluntariado, los usuarios tienen acceso a múltiples funcionalidades relacionadas a su rol, por ejemplo, los voluntarios pueden postularse para un trabajo, los proveedores ofrecer un trabajo, y los administradores manejar usuarios. Sin embargo, los roles no tienen acceso a las funcionalidades de otros roles, por lo cual existe una necesidad de controlar quién está autorizado a hacer qué.

Esta subcapa se encarga de controlar el acceso a las distintas peticiones HTTP que puede realizar un usuario, con el fin de permitir que únicamente aquellos con la autorización necesaria para hacer uso de una funcionalidad específica del sistema, puedan acceder a esta. Para esto se hace uso del framework Spring Security, el cual provee una serie de filtros por los cuales debe pasar cualquier petición HTTP antes de ser atendida por las demás subcapas del BackEnd. Para una mayor información sobre su funcionamiento, se recomienda realizar una lectura sobre la arquitectura de Spring Security, cuya documentación oficial se incluye en la bibliografía.

Spring Security provee por defecto de múltiples filtros por los cuales una petición HTTP debe pasar antes ser atendida por un controller. De estos filtros interactuamos (o configuramos) únicamente el filtro CORS. Este último es un mecanismo basado en cabeceras HTTP que permite indicarle al servidor que puede recibir peticiones de dominios distinto al suyo, esto permite la comunicación entre el FrontEnd y el BackEnd al hacer uso de distintos dominios. Adicionalmente se añadieron dos filtros personalizados encargados de verificar la identidad del usuario que realiza las peticiones. Ambos filtros utilizan cookies, pequeños ficheros de información que se almacenan en el navegador y pueden ser usadas por las

páginas webs, las cuales son utilizadas en el sistema para hacer seguimiento del tipo de usuario que está haciendo uso de este.

Antes de comenzar a desarrollar el funcionamiento de los filtros personalizados, es necesario explicar que es un token JWT (JSON Web Token), debido a que estos son utilizados en ambos filtros. JWT es un estándar abierto (RFC 7519.) que define un medio para transmitir información mediante el formato JSON (JavaScript Object Notation), de forma segura y verificable. Un token JWT consta de una cabecera con información de un algoritmo de firmado a usar (como HS256), un payload o carga que contiene distintos campos o claims a conveniencia del proveedor del token, y por último una firma generada mediante la codificación en Base64 de las dos primeras partes del token y una palabra secreta, los cuales son firmados con el algoritmo especificado en la cabecera. La firma es el componente más importante de este, debido a que permite verificar que el mismo no ha sido alterado por ningún actor malicioso y la información que posee es verídica.

Filtro de usuarios JWT

Este filtro es el encargado de autenticar a los usuarios con los roles de voluntario o proveedor en el sistema, y realiza la aprobación o rechazo de las distintas peticiones HTTP que se puedan realizar según el usuario y su rol. Este filtro se encarga de examinar un token JWT contenido en una cookie la cual le es entregada a un usuario al momento de iniciar sesión en el sistema, y contiene los siguientes claims:

- Subject: El username del usuario.
- Role: Rol del usuario.
- Expiration Time: Tiempo de expiración del token.

Cada vez que el BackEnd recibe una petición HTTP a una ruta que requiera de autorización mediante roles (a excepción de las rutas de administración), el filtro intenta extraer el token JWT de la cookie para examinarlo. Si el token existe, se verifica su validez y su tiempo de expiración, para así obtener el username de quien realiza la petición, autenticarlo y autorizarlo en el sistema. En caso de que el token no exista o sea inválido, no se autentica ni autoriza al que creó la petición.

Filtro de autorización de administradores

El objetivo de este filtro es autenticar y autorizar a un usuario con rol de administrador en el sistema, de este modo se le permite realizar peticiones HTTP a las distintas rutas de administración. Este filtro es similar al de usuarios JWT, dado que un administrador al iniciar sesión recibe una cookie con un token JWT que es utilizado para autenticarse y autorizarse en el sistema, sin embargo se diferencia en el contenido del token y el tipo de la cookie en el cual se encuentra.

Primero, a diferencia de la cookie que se le entrega a un usuario del tipo voluntario o proveedor al iniciar sesión, la entregada al administrador es del tipo HTTP-Only. Una cookie de este tipo no puede ser accedida mediante scripts de cliente en un navegador, y únicamente pueden accederse desde BackEnd, el cual tiene acceso a su contenido. Esto se hizo principalmente para prevenir la lectura del token por posibles scripts maliciosos, además de que no es necesario su lectura fuera del BackEnd.

Para finalizar, el token JWT contiene en la cookie, un campo de nombre *secret* que a su vez contiene como valor una palabra. Esta palabra es extraída por el filtro y se compara con una palabra secreta que únicamente conoce el BackEnd. Si ambas palabras coinciden, entonces el administrador es autenticado y autorizado, en caso contrario, no se le otorga acceso a la ruta solicitada.

Subcapa Controllers

El sistema de voluntariado desarrollado es un sistema interactivo, dado que ofrece múltiples funcionalidades con el objetivo de que los usuarios hagan uso del mismo. Para esto introducimos el concepto de controllers, clases encargadas de procesar las peticiones entrantes, preparar un modelo de datos y devolver una respuesta. En Spring Boot, los controllers desempeñan un papel fundamental al definir la API y las rutas de las peticiones HTTP de una aplicación web, y estos se definen mediante las etiquetas de anotación @Controller o @RestController.

Además de su función principal, los controllers tienen la capacidad de acceder a otros componentes de la aplicación, como servicios y repositorios, con el fin de ejecutar la lógica de negocio requerida y devolver los resultados apropiados en la respuesta HTTP. Esto permite una separación de responsabilidades y una estructura modular en la aplicación, en el cual los controllers coordinan las interacciones entre las diferentes capas y componentes del sistema.

Esta subcapa contiene todos los controllers de la API que manejan las peticiones entrantes y generan las respuestas correspondientes. La misma puede interactuar tanto con la subcapa Service, así como Models directamente en casos específicos. Finalmente, todos los controllers fueron anotados con la etiqueta @RestController, la cual indica que las respuestas de los controllers se encuentran en el cuerpo de la respuesta HTTP.

Las siguientes secciones del informe están dedicadas a explicar muy brevemente de qué clase de peticiones se encarga cada controller implementado.

AuthenticationController

Este controller se encarga de manejar las peticiones relacionadas con la autenticación y autorización de los usuarios en el sistema. Entre las peticiones más representativas de este controlador se encuentran:

Inicio de sesión

Recibe una petición del tipo POST para realizar el inicio de sesión de un usuario mediante la ruta *"/auth/login"*. Requiere un DTO en el cuerpo con las credenciales de inicio de sesión. En caso de que las credenciales sean correctas, responde con una cookie con el token JWT de autenticación y autorización de usuario.

Registro de usuario

Recibe una petición POST para registrar un nuevo usuario mediante la ruta *"/auth/register"*. Requiere un DTO en el cuerpo con los distintos datos del usuario a registrar. Verifica los datos ingresados del usuario a registrar, y en caso de ser correctos, registra al usuario.

AdminController

Se encarga de manejar las peticiones relacionadas con las peticiones de administrador, es decir, permite administrar los usuarios y toda la información correspondiente a estos. Entre las peticiones más representativas de este controlador se encuentran:

Edición de un usuario

Recibe una petición POST para actualizar un nuevo usuario mediante la ruta *"/admin/edit-user"*. Requiere un DTO en el cuerpo que contiene los nuevos datos del usuario a editar. Verifica los datos ingresados del usuario a editar, y en caso de ser correctos, realiza los cambios correspondientes.

Eliminación de usuarios

Recibe una petición GET para eliminar un usuario mediante la ruta *"/admin/delete-user"*. Requiere un parámetro URL con el nombre del usuario (codificado) a eliminar. Después de haber comprobado la existencia del usuario, se elimina el mismo.

SupplierPostulationController

Se encarga de manejar todas las peticiones relacionadas con el manejo de las postulaciones por parte del proveedor, es decir, aceptar una postulación, rechazarla y ver todas las postulaciones relacionadas con los trabajos creados por el proveedor. Entre las peticiones más representativas de este controlador se encuentran:

Aceptar postulación

Recibe una petición de POST para aceptar una postulación mediante la ruta *“/api/supplier/accept-postulation”*. Requiere un parámetro URL con la id de la postulación a aceptar, y la cookie JWT que recibe un proveedor al iniciar sesión necesario para obtener su username.

Rechazar postulación

Recibe una petición de POST para rechazar una postulación mediante la ruta *“/api/supplier/reject-postulation”*. Requiere un parámetro URL con la id de la postulación a

rechazar, y la cookie JWT que recibe un proveedor al iniciar sesión el cual se usará para obtener su username.

SupplierWorkController

Este se encarga de manejar las peticiones relacionadas con la gestión de trabajos de voluntariado por parte de un proveedor. Entre las peticiones más representativas de este controlador se encuentran:

Creación de trabajo

Recibe una petición POST para crear un nuevo trabajo de voluntariado mediante la ruta *"/api/supplier/work-create"*. Requiere un DTO en el cuerpo que contiene todos los datos del trabajo a crear, y la cookie JWT que recibe un proveedor al iniciar sesión para obtener el username de este.

Obtención de trabajos de un proveedor

Recibe una petición GET para obtener los datos de un trabajo de voluntariado mediante la ruta *"/api/supplier/works"*. Requiere la cookie JWT que recibe un proveedor al iniciar sesión, para así obtener el username del que vamos a buscar todos sus trabajos creados. Retorna la lista de trabajos del proveedor junto a sus datos en el cuerpo de la respuesta.

SupplierWorkSessionController

Se encarga de manejar todas las peticiones para el manejo de las sesiones de los trabajos. Entre las peticiones más representativas de este controlador se encuentran:

Cambiar el estado de una sesión

Recibe una petición POST para cambiar el estado de una sesión mediante la ruta *"/api/supplier/session-status"*. Requiere la cookie JWT que recibe el proveedor al iniciar sesión para obtener su username, también es necesario dos parámetros URL los cuales son el id de la postulación a la cual se le quiere cambiar el estado, y el nuevo estado que se le asignará.

Obtener sesiones de un trabajo

Recibe una petición POST para obtener las sesiones de un trabajo en una fecha y bloque horario particular mediante la ruta *"/api/supplier/work-sessions"*. Requiere la cookie JWT que recibe el proveedor al iniciar sesión para obtener su username, además es necesario un DTO el cual contiene los detalles del trabajo, la fecha y el bloque horario del cual se quiere las sesiones. Retorna todas las sesiones existentes para ese trabajo en la fecha y bloque horario requerido.

TagsController

Se encarga de manejar la petición de obtener todas las tags existentes en el sistema, esto mediante la ruta *“/api/all/tags”*. Recibe una peticiones GET y no requiere de ningún parámetro.

VolunteerDetailsController

Se encarga de manejar las peticiones relacionadas con la edición de las preferencias de los voluntarios y la obtención de detalles de los mismos. Entre las peticiones más representativas de este controlador se encuentran:

Edición de las preferencias de un voluntario

Recibe una petición POST para actualizar las preferencias de un voluntario mediante la ruta *"/api/user/user-edit"*. Requiere un DTO en el cuerpo que contiene las preferencias del usuario ya existente. Verifica los datos ingresados del usuario a editar, y en caso de ser correctos, realiza los cambios correspondientes.

Obtención de los datos de un voluntario

Recibe una petición GET para obtener la información de un voluntario mediante la ruta *"/api/user/details"*. No requiere de parámetros y obtiene el username, a través del JWT (cookie de autenticación), que es el usuario del cual se obtendrán sus datos. Retorna toda la información relacionada con el usuario, sus tags y los bloques de horario.

VolunteerPostulationsController

Se encarga de manejar las peticiones relacionadas a la creación, edición y eliminación de postulaciones por parte de los voluntarios. Entre las peticiones más representativas de este controlador se encuentran:

Crear postulación

Recibe una petición de tipo POST para crear una nueva postulación mediante la ruta *“/api/user/postulate”*. Requiere un DTO que contiene todos los datos necesarios para crear la postulación, como lo son el username del voluntario, el nombre del trabajo, la fecha de inicio y la fecha de fin de la postulación, y también requiere la cookie JWT recibida al iniciar sesion para verificar la identidad del voluntario.

Cancelar postulación

Recibe una petición de tipo GET para cancelar una postulación mediante la ruta *“/api/user/cancel-postulation”*. Requiere un parámetro URL el cual es la id de la postulación la cual el voluntario quiere cancelar, además requiere la cookie JWT que recibe al iniciar sesión para verificar la identidad del voluntario.

VolunteerWorkController

Se encarga de manejar las peticiones para obtener los trabajos existentes en el sistema y las sesiones relacionadas al voluntario. Entre las peticiones más representativas de este controlador se encuentran:

Obtener trabajos

Recibe una petición de tipo POST para obtener los trabajos del sistema de un mes y un año particular mediante la ruta *"/api/user/works-by"*. Requiere dos parámetros URL los cuales son dos enteros que representan el mes y el año en los cuales se buscará los trabajos, también requiere una lista de DTO que contienen la información de los bloques horarios de preferencia del usuario para en caso de existir trabajos que coincidan con estos bloques horarios se retornen solo estos y no todos los trabajos existentes en ese mes y año, por último requiere la cookie JWT recibida al iniciar sesion para poder comprobar la identidad del voluntario.

Obtener sesiones

Recibe una petición de tipo GET para obtener las sesiones relacionadas al voluntario mediante la ruta *"/api/user/work-sessions"*. Requiere la cookie JWT que recibe el voluntario al iniciar para obtener su username, también requiere dos parámetros URL los cuales son dos enteros que representan el mes y el año en los cuales se buscará las sesiones del voluntario.

ReportController

Se encarga de manejar todas las peticiones relacionadas a la generación de reportes, ya sea como un voluntario, un proveedor o el administrador del sistema. Entre las peticiones más representativas de este controlador se encuentran:

Generar reportes para proveedores

Recibe una petición GET para generar un reporte para un proveedor, mediante la ruta *"/api/supplier/generate-report"*. Requiere cuatro parámetros URL los cuales son: Dos fechas que representan el espacio de tiempo en el cual se va a buscar los datos para generar el reporte, el formato del reporte que puede ser formato csv o formato pdf, y el tipo de reporte solicitado que para el proveedor puede ser reporte de seguimiento o reporte de postulaciones. Además requiere la cookie JWT que recibe el proveedor al iniciar sesión para poder validar así su identidad.

Generar reportes para voluntarios

Petición similar a la de generación de reportes para proveedor, con la diferencia de que solo se puede generar reportes de seguimiento.

Subcapa Services

Los servicios son componentes que se utilizan para implementar la lógica de una aplicación. Estos son los responsables de realizar operaciones más complejas, como la

manipulación y el procesamiento de datos, la coordinación entre varios repositorios y la implementación de reglas específicas.

Los servicios en Spring Boot se definen como clases anotadas con @Service, lo que indica que son componentes de servicio. Estos servicios proporcionan métodos que pueden ser invocados desde los controllers u otros componentes de la aplicación para realizar todo tipo de operaciones, como la recuperación y manipulación de datos desde los repositorios y la integración con otros servicios externos. También pueden contener la lógica para transformar y procesar los datos antes de enviarlos a los controllers para su presentación.

Esta subcapa se encarga de ejecutar toda la lógica relacionada a las distintas funcionalidades ofrecidas a los usuarios en la capa Controllers, comunicarse con otros servicios, realizar consultas a la capa de repositories y dar una respuesta/recursos de vuelta al controller que lo invocó en caso de ameritarlo.

AuthorizationService

Este se encarga de toda la lógica relacionada a la autenticación/autorización y registro de usuarios. Ofrece las siguientes funcionalidades:

- Inicio de sesión de un usuario.
- Registro de un usuario.
- Obtención de la entidad de un usuario.

SupplierWorkService

Este se encarga de toda la lógica relacionada y funcionalidades del manejo de trabajos de voluntariado por parte de un proveedor en el sistema. Ofrece las siguientes funcionalidades a los proveedores:

- Crear un trabajo de voluntariado.
- Editar un trabajo de voluntariado creado previamente por el proveedor.
- Eliminar un trabajo de voluntariado creado previamente por el proveedor.
- Mostrar todos los trabajos de voluntariado creados por el proveedor en un mes y año específicos.

SupplierPostulationService

Este se encarga de toda la lógica y funcionalidades relacionadas con el manejo de postulaciones de voluntarios a trabajos, por parte de un proveedor en el sistema. Ofrece las siguientes funcionalidades a los proveedores:

- Aceptar o rechazar postulaciones de voluntarios a trabajos creados por un proveedor.
- Obtener las postulaciones pendientes por manejar de un trabajo creado por un proveedor.
- Obtener los trabajos creados por el proveedor, que tengan postulaciones pendientes por manejar.

SupplierWorkSessionService

Este se encarga de toda la lógica y funcionalidades relacionadas con el manejo de las sesiones de trabajo de un voluntario, por parte de un proveedor en el sistema. Ofrece las siguientes funcionalidades a los proveedores:

- Cambiar el estado de una sesión de trabajo (relacionada al proveedor) de un voluntario, por: aceptada, pendiente o rechazada.
- Obtener las sesiones de trabajo en una fecha y bloque horario, de todos los voluntarios en un trabajo creado por un proveedor.

AdminService

Este se encarga de toda la lógica y funcionalidades relacionadas con el administrador del sistema. Ofrece las siguientes funcionalidades a los administradores:

- Edita los detalles de un usuario existente en el sistema.
- Cambia el estado de suspensión de un usuario.
- Elimina un usuario del sistema creado previamente.
- Establece una nueva contraseña para un usuario.

VolunteerDetailsService

Este se encarga de toda la lógica y funcionalidades relacionadas con el gestionar los detalles y preferencias de los voluntarios en el sistema. Proporciona las siguientes funcionalidades:

- Edita los detalles de un voluntario existente en el sistema.
- Obtiene todos los detalles de un voluntario existente en el sistema.

VolunteerWorkService

Este se encarga de toda la lógica y funcionalidades relacionadas con gestionar los trabajos disponibles para los voluntarios en el sistema. Actualmente permite obtener todos los trabajos disponibles para los voluntarios y obtener todas las sesiones de un voluntario así como también los trabajos asociados a un sesión de trabajo.

VolunteerPostulationService

Este se encarga de toda la lógica y funcionalidades relacionadas con el manejo de postulaciones por parte de un voluntario en el sistema. Proporciona las siguientes funcionalidades:

- Postular un voluntario a un trabajo.
- Cancelar una postulación a un trabajo.
- Editar una postulación a un trabajo.
- Obtener la postulación de un trabajo.
- Obtener el trabajo de una postulación.

TagsService

Este se encarga de toda la lógica y funcionalidades relacionadas con el gestionar las etiquetas en el sistema. Actualmente solo permite obtener todas las etiquetas de usuarios y trabajos disponibles en el sistema.

ReportService

Se encarga de toda la lógica y funcionalidades relacionadas a la generación de reportes ya sea para un voluntario, un proveedor o el administrador del sistema. Proporciona una única funcionalidad la cual permite generar un reporte, esta funcionalidad haciendo uso de la segregación de responsabilidades se encarga de obtener los datos para el reporte y usarlos para crear dicho documento.

Subcapa Repositories

Esta subcapa se encarga del acceso y persistencia de recursos en la aplicación. Para este sistema se utiliza una única base de datos compartida por todos los servicios y controllers que necesiten persistir o acceder a recursos. En la subcapa, es utilizado principalmente el

ORM Hibernate para realizar todas las operaciones y consultas, y para la persistencia se utiliza PostgreSQL en la base de datos.

Hibernate es un ORM (Object–relational mapping) que implementa JPA (Java Persistence API), siendo este el encargado de manejar el mapeo objeto-relacional y la persistencia de datos, además de traducir las operaciones de la capa de aplicación en consultas SQL y gestionar la comunicación con la base de datos de manera transparente. Por su parte, JPA es una especificación que define un modelo de persistencia de objetos de Java para ser usado por las ORMs.

La principal forma de interacción con estos recursos en Spring Boot es mediante repositorios, una abstracción proveída por el framework, la cual consiste en una interfaz que se encarga de realizar todas las consultas necesarias a la base de datos. Cabe destacar que esta interacción es realizada mediante entidades, clases que representan los datos o recursos almacenados en la base de datos. En la siguiente subcapa se describe cómo se definen, dado a que estas pueden verse como una capa.

Ahondando un poco más en la definición de consultas en un repositorio, actualmente en el proyecto se utilizan dos métodos distintos para definir consultas en un repositorio:

- JPA Query Methods: El repositorio deriva una consulta a la base de datos basada en el nombre de una función, sus parámetros y tipo de retorno. Esto permite crear consultas sencillas rápidamente, ya que el método es derivado en su totalidad, sin embargo, para consultas complicadas, los nombres de los métodos pueden llegar a ser extremadamente largos y engorrosos de usar.

- Query Annotation: Spring Boot provee la anotación @Query que permite escribir una consulta de SQL y asociarla con un método del repositorio, tomando en cuenta también sus argumentos y tipo de retorno. Cabe destacar que también se permite la definición de consultas en JPQL y HQL, ambos lenguajes de consultas, definidos por JPA e Hibernate respectivamente. En el proyecto se utiliza HQL.

Para más información sobre la definición de consultas, revisar las referencias 20 y 21.

Actualmente en el proyecto se cuentan con los siguientes repositorios:

- UserRepository: Consultas relacionadas a usuarios del sistema.

- WorkRepository: Consultas relacionadas a trabajos de voluntariado en el sistema.

- PostulationRepository: Consultas relacionadas a postulaciones de usuarios en el sistema.

- WorkInstanceRepository: Consultas relacionadas a instancias de trabajo en el sistema.

- WorkSessionRepository: Consultas relacionadas a sesiones de trabajo de un usuario en el sistema.

- WorkHourBlockRepository y UserHourBlockRepository: Consultas relacionadas a bloques horarios de un trabajo y preferidos por un usuario respectivamente.

- TagRepository: Consultas relacionadas a las etiquetas que puede tener un trabajo o preferir un usuario.

Subcapa Entities

Esta subcapa se encarga simplemente de la representación de datos persistentes en la base de datos utilizada por el proyecto. Como se mencionó anteriormente, Hibernate es un ORM, por lo que la principal forma de representación de datos en la base de datos es mediante POJOs (Plain old Java objects), los cuales llevan anotaciones especiales. Esta representación la realiza Hibernate mediante entidades, las cuales representan datos persistentes almacenados, y estas se definen al utilizar la etiqueta @Entity en una clase. Actualmente en el proyecto se encuentran definidas las siguientes entidades:

- UserEntity: Representa a los usuarios de la aplicación.

- WorkEntity: Representa la plantilla inicial de los trabajos de voluntariado creados por los proveedores.

- TagEntity: Representa las distintas etiquetas que pueden tener los trabajos.

- UserHourBlock: Representa un bloque horario y el día de la semana en el cual se ubica la participación de un usuario en un trabajo.

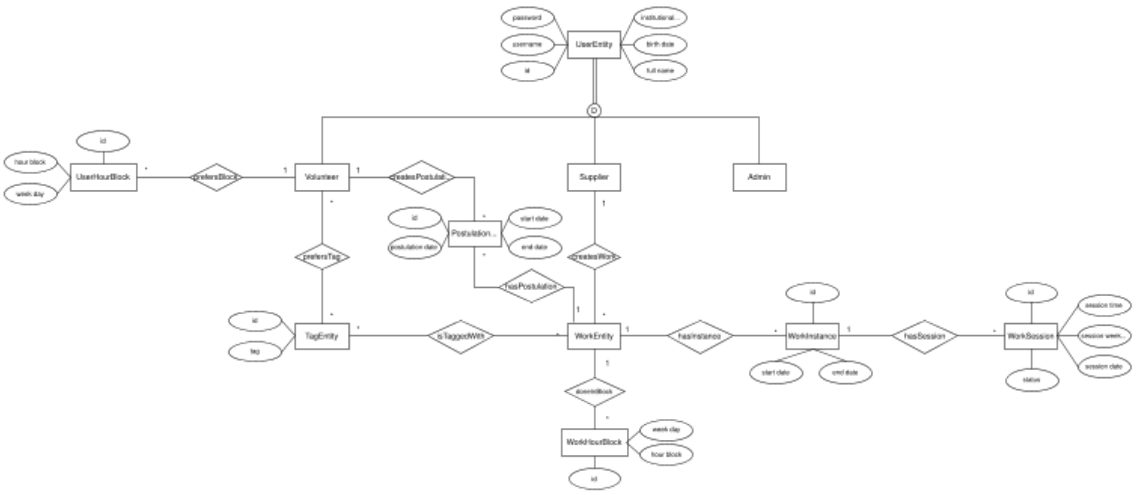
- WorkHourBlock: Representa un bloque horario y el día de la semana en el cual se ubica un trabajo.

- PostulationEntity: Representa una postulación a un trabajo, realizada por un voluntario.

- WorkInstanceEntity: Representa un trabajo instanciado. Puede verse como un contrato entre un voluntario y un proveedor cuando este último acepta una postulación.

- WorkSessionEntity: Representa una sesión de un trabajo de voluntariado en la cual un voluntario debe de realizar tras postularse y ser aceptado por el proveedor.

A continuación se presenta un diagrama ERE (Entity-Relationship Extended) con las distintas entidades y relaciones entre ellas:



El mismo puede ser accedido mediante el siguiente enlace para una mejor visualización.

Capa FrontEnd

El FrontEnd es una capa que consiste en la presentación visual del sistema a un usuario mediante una interfaz, la cual tiene como propósito permitirles a los usuarios acceder a las funcionalidades en el sistema accesibles para estos, a través de elementos interactivos como botones y formularios. Como se mencionó anteriormente, la capa de FrontEnd interactúa con la capa de BackEnd mediante peticiones, esto tanto para obtener información de la subcapa de modelos a la cual un usuario quiera acceder, o para realizar una acción que el usuario tenga permitido realizar en el sistema.

Esta capa se implementa con el lenguaje de programación JavaScript mediante la biblioteca React, que permite la creación de interfaces de usuario interactivas y reactivas. La principal característica de React es su enfoque en la construcción de componentes reutilizables y su manejo eficiente de las actualizaciones del DOM (Document Object Model). Debido a que son un concepto vital y el principal bloque de construcción de React, nos vemos en la necesidad de definir brevemente qué es un componente en el contexto de la librería: Un componente es un bloque de código que define un elemento de una interfaz, encapsula su lógica y es reutilizable, por lo que puede ser utilizarse en más de una sección del código. Utiliza un modelo de programación basado en componentes, en el cual cada parte de la interfaz de usuario se representa como un componente independiente. React utiliza una técnica de nombre"Virtual DOM" para mejorar el rendimiento, la cual consiste en la creación de una representación virtual de los elementos del DOM, para luego compararla con el DOM y determinar qué cambios deben aplicarse en este para minimizar la cantidad de

actualizaciones. Para obtener una mayor comprensión de React y React DOM, se recomienda consultar las referencias bibliográficas 13 y 15, las cuales proporcionan información adicional sobre estos temas.

Se decidió subdividir esta capa en otras subcapas con responsabilidades más específicas, que son: Enrutamiento, Acceso, Vistas, Componentes, Fetchers.

Subcapa de enrutamiento

El enrutamiento (en inglés routing) es un concepto fundamental en el desarrollo de aplicaciones web que se refiere al proceso de dirigir y manejar las peticiones de los usuarios a diferentes páginas o vistas en una aplicación. La capa de enrutamiento es responsable de determinar qué componente o vista se debe mostrar al usuario en función de la URL solicitada. Además, el enrutamiento también permite la navegación entre diferentes páginas de manera fluida y sin tener que recargar toda la aplicación. Esto mejora la experiencia del usuario al proporcionar una navegación más rápida y eficiente. En resumen, el enrutamiento juega un papel crucial en la arquitectura y funcionalidad de una aplicación web al permitir la organización y la gestión de las diferentes vistas y su interacción con el usuario.

Antes de desarrollar más sobre la implementación de la capa de enrutamiento, debemos definir qué son los términos Client-Side Rendering y Single Page Application (SPA de ahora en adelante). Empezando por el primero, Client-Side Rendering es una técnica en el cual el trabajo de renderizar una página web es responsabilidad del navegador del cliente, tal que cada vez que el usuario acceda a una nueva dirección de la página web, no sea necesario hacer una petición a un servidor que retorne un documento con la nueva vista, sino que en base a un documento HTML inicial el navegador realiza modificaciones a este para generar la nueva interfaz. En el caso de SPA, simplemente nos referimos a aplicaciones web que utilicen Client-Side Rendering y por consiguiente consisten únicamente en una sola vista manejada por el navegador.

Luego de definir SPA y Client-Side Rendering, podemos señalar que en el contexto de una SPA desarrollada con React, el enrutamiento se encarga de gestionar las transiciones entre diferentes componentes o vistas sin necesidad de recarga. Esto permite una experiencia de usuario fluida, donde los cambios de contenido se realizan de forma dinámica y rápida.

Para la implementación de esta capa se utiliza un paquete de JavaScript de nombre React Router, que consiste en una librería de enrutamiento específica para aplicaciones basadas en React, que proporciona un conjunto de componentes y utilidades que permiten definir y gestionar el enrutamiento dentro de una aplicación de React. Con React Router, se pueden definir rutas y establecer reglas para que cuando un usuario navegue a una determinada URL, se cargue el componente correspondiente a una vista. También ofrece características como enrutamiento anidado, enrutamiento basado en parámetros, redirecciones, entre otros.

Subcapa de acceso

Similar a como la capa de BackEnd tiene la necesidad de proteger el acceso a las distintas peticiones disponibles en el sistema, en la capa de FrontEnd existe una necesidad de controlar quienes pueden acceder a las distintas rutas disponibles en la subcapa de enrutamiento, ya que . Se definieron los siguientes tipos de control de acceso en el sistema:

- Acceso según rol: Solo se permite el acceso a los usuarios con el rol (excluyendo el de administrador) necesario para acceder a la ruta.

- Acceso administrativo: Solo se permite el acceso a un usuario con el rol de administrador.

- Acceso público: Se permite el acceso a cualquier tipo de usuario.

Esta subcapa se apoya de las subcapas de enrutamiento y de componentes en su implementación. En el caso de la subcapa de enrutamiento, esta apoya la subcapa de acceso mediante la carga de componentes según la URL a la que quiera acceder un usuario, ya que esto nos permite establecer la ruta adecuada para procesar la solicitud y proporcionar los recursos solicitados. Respecto a la subcapa de componentes, fueron definidos los siguientes componentes:

- RequireAuth: Este componente se encarga de controlar el acceso a las rutas de acceso según rol. Para esto, el componente recibe como parámetro el rol con acceso a la ruta, y cuando un usuario intenta acceder a esta, decodifica el token JWT de la cookie que se recibe al iniciar sesión en el sistema y verifica que el rol contenido en este coincida con el recibido por el componente. En caso contrario se niega el acceso a la ruta.

- RequireAdminAuth: Este componente se encarga de controlar el acceso a las rutas de acceso administrativo. Para esto, realiza una petición HTTP al BackEnd que envía la cookie de administración que recibe un administrador al iniciar sesión, y si se encuentra presente es autorizado mediante una respuesta positiva. En caso contrario, se niega el acceso a la ruta.

En ambos componentes, si el usuario que intenta acceder a la ruta cuenta con la autorización necesaria, se procede a cargar el componente asociado a la vista de la ruta, y en el caso contrario, se carga una vista que le indica al usuario que no cuenta con autorización. Para las rutas de acceso público, no se utiliza ningún componente adicional ya que se le indica a React Router que directamente cargue el componente de la vista sin ningún paso intermedio.

Subcapa de vistas

Las vistas en una aplicación web son las distintas representaciones visuales que se muestran al usuario. Cada página o vista puede estar compuesta por uno o más componentes. Cada página tiene su propia estructura y contenido único.

La diferencia principal entre componentes y páginas/vistas radica en el nivel de granularidad. Los componentes son elementos más pequeños y reutilizables que componen las páginas o vistas, que son las representaciones completas y visibles para el usuario. A pesar de las diferencias entre vistas y componentes, se debe acotar que una vista es un componente de React que contiene a su vez más componentes.

Vista Login

Esta vista se encarga de permitirle a un usuario iniciar una sesión, se compone de dos elementos:

- Un formulario de inicio de sesión con los siguientes campos:
    - Nombre de usuario.
    - Contraseña.
- Un enlace a la vista de registro.

Los campos del formulario se validan de manera dinámica, esto es, la vista se encarga de verificar el correcto llenado de los campos y en caso de algún campo inváido, se muestra un mensaje con la razón de la invalidez del mismo. Al enviarse el formulario se realiza la autenticación para determinar la correctitud de los datos, que en caso de ser correctos, se autoriza al usuario a iniciar una sesión y se redirige a su perfil, y en caso contrario, se le proporciona un mensaje de error.

Vista registro

Esta vista permite a un usuario registrarse, y está compuesta por dos elementos que se visualizan

- Una selección del rol con el cual el usuario quiere registrarse.

- Un formulario de registro con los siguientes campos:
    - Nombre de usuario.
    - Nombre completo.
    - Clave.
    - Fecha de nacimiento.
    - ID institucional.

Cada uno de los campos del registro se validan dinámicamente, y en caso de no ser válidos, el título de los campos cambia para mostrar la razón de su invalidez.

Adicionalmente, si el registro es fallido, el usuario recibe una alerta indicando que el proceso falló, mientras que en caso contrario indica que el registro fue exitoso.

Vista panel proveedor

Esta vista se encarga de mostrar a un usuario proveedor un panel compuesto por:

- Calendario mensual de navegación.
- Calendario semanal con bloques horarios.
- Lista de trabajos.
- Botón de generación de reportes.

Las funcionalidades que presenta este panel son:

- Visualización de trabajos.
- Manejo de las postulaciones de un trabajo.
- Manejo de una sesión de trabajo ya concluida.
- Creación de trabajos.
- Edición de trabajos.
- Eliminación de trabajos.
- Cierre de sesión de un usuario.
- Generación de reportes de un proveedor.

A excepción de la visualización de trabajos y el cierre de sesión, todas estas funcionalidades cuentan con modales con los cuales el usuario puede interactuar y realizar todas las funciones relacionadas a la funcionalidad deseada. Adicionalmente, cada una de estas funcionalidades cuenta con alertas que indican si se pudieron realizar de manera exitosa o fallida.

Vista panel voluntario

Esta vista se encarga de mostrar a un usuario voluntario un panel compuesto por:

- Calendario mensual de navegación.

- Calendario semanal con bloques horarios.
- Lista de trabajos disponibles ordenador por preferencias de etiquetas.

Las funcionalidades que presenta este panel son:

- Visualización de trabajos disponibles.
- Manejo de las postulaciones de un trabajo.

Vista postulaciones voluntario

Esta vista se encarga de mostrar a un usuario voluntario una tabla que contiene todas las postulaciones hechas por el usuario, la misma se muestra por páginas y el usuario elige la cantidad de postulaciones que quiere ver por página.

Las funcionalidades que presenta esta vista son:

- Visualización de postulaciones realizadas.
- Filtrar postulaciones.
- Manejo de las postulaciones.

Vista sesiones voluntario

Esta vista se encarga de mostrar a un usuario voluntario un panel compuesto por:

- Calendario mensual de navegación.
- Calendario semanal con bloques horarios.
- Botón de generación de reportes.

Las funcionalidades que presenta este panel son:

- Visualización del estado de la sesión.
- Generación de reportes de sesiones de un voluntario.

Vista panel administrador

Esta vista se encarga de mostrar a un usuario administrador un panel compuesto por:

- Lista de usuarios en el sistema.
- Botón de creación de usuario.
- Botón de generación de reportes.

Las funcionalidades que presenta este panel son:

- Manejo de usuarios del sistema: Creación, edición, suspensión y restauración de contraseñas.
- Generación de reportes de proveedores.

Subcapa de componentes

Como se mencionó anteriormente en la introducción de la capa, los componentes son elementos reutilizables y autónomos que representan una parte específica de la interfaz de usuario. Estos componentes desempeñan un papel crucial en el desarrollo de aplicaciones, ya que permiten la modularidad y la reutilización del código.

En el contexto de React, una biblioteca muy popular para la construcción de interfaces de usuario, los componentes son la base fundamental. React sigue una estructura jerárquica en la que los componentes se pueden anidar unos dentro de otros. Esto permite la creación de componentes más pequeños y especializados, que se combinan para formar la interfaz de usuario completa de una aplicación.

Cada componente en React puede tener su propio estado interno, lo que significa que puede almacenar y gestionar datos específicos de ese componente. Además, los componentes pueden recibir propiedades (props) que les proporcionan datos externos y los utilizan para personalizar su comportamiento y apariencia. Las propiedades se pasan a los componentes como argumentos y se utilizan para configurar su estado interno, determinar qué datos mostrar y cómo interactuar con el usuario.

Los componentes en React también pueden incluir lógica y manipulación de eventos. Pueden responder a acciones del usuario, como hacer clic en un botón o introducir datos en un formulario, y realizar acciones específicas en consecuencia. La lógica y la manipulación de eventos se implementan utilizando métodos y funciones que se ejecutan en respuesta a ciertos eventos.

Además de la lógica y la manipulación de eventos, los componentes en React también se encargan del renderizado de elementos HTML en la interfaz de usuario. Utilizando una sintaxis llamada JSX, los componentes pueden definir la estructura y el contenido de los elementos HTML que se mostrarán en pantalla. Esta combinación de lógica, manipulación de

eventos y renderizado de elementos HTML permite a los componentes de React proporcionar una interfaz de usuario dinámica e interactiva.

Subcapa de Fetchers

Como se ha venido mencionando en las secciones anteriores del informe, cuando el FrontEnd necesita obtener información persistente o hacer uso de una funcionalidad del sistema, este realiza peticiones al BackEnd. y para esto utilizamos “Fetchers”. Los Fetchers no son más que una serie de métodos (agrupados según la funcionalidad a la que acceden) que se encarga de realizar estas peticiones HTTP a la subcapa de Controllers del BackEnd, recibir una respuesta y transmitirla al componente que invocó el método

Esta subcapa fue implementada usando la Fetch API de JavaScript, que provee al lenguaje de un método de nombre *“fetch”* mediante el cual se pueden realizar peticiones HTTP a una URL. La razón de elegir esta API nativa y no una librería externa como Axios, es que debido a la sencillez de las peticiones y el sistema, no nos vimos en la necesidad de usar las funcionalidades ofrecidas por estas librerías externas como interceptores de peticiones y respuestas HTTP.

El proyecto actualmente consta de las siguientes agrupaciones de Fetchers:

- ApiFetches: Se encarga de realizar todos los fetches relacionados a las funcionalidades de registro, cierre de sesion y obtención de tags de voluntariado y proveedor

- ApiFetchesAdmin: Se encarga de realizar todos los fetches relacionados a las funcionalidades del administrador, como el manejo de usuario y trabajos.

- ApiFetchesVolunteer: Se encarga de realizar todos los fetches relacionados a las funcionalidades de obtención de trabajos, edición de información y preferencias y agregar, editar y eliminar postulaciones propias.

- ApiFetchesSupplier: Se encarga de realizar todos los fetches relacionados a las funcionalidades de creación, edición y eliminación de trabajo y manejo de postulaciones de su trabajo.

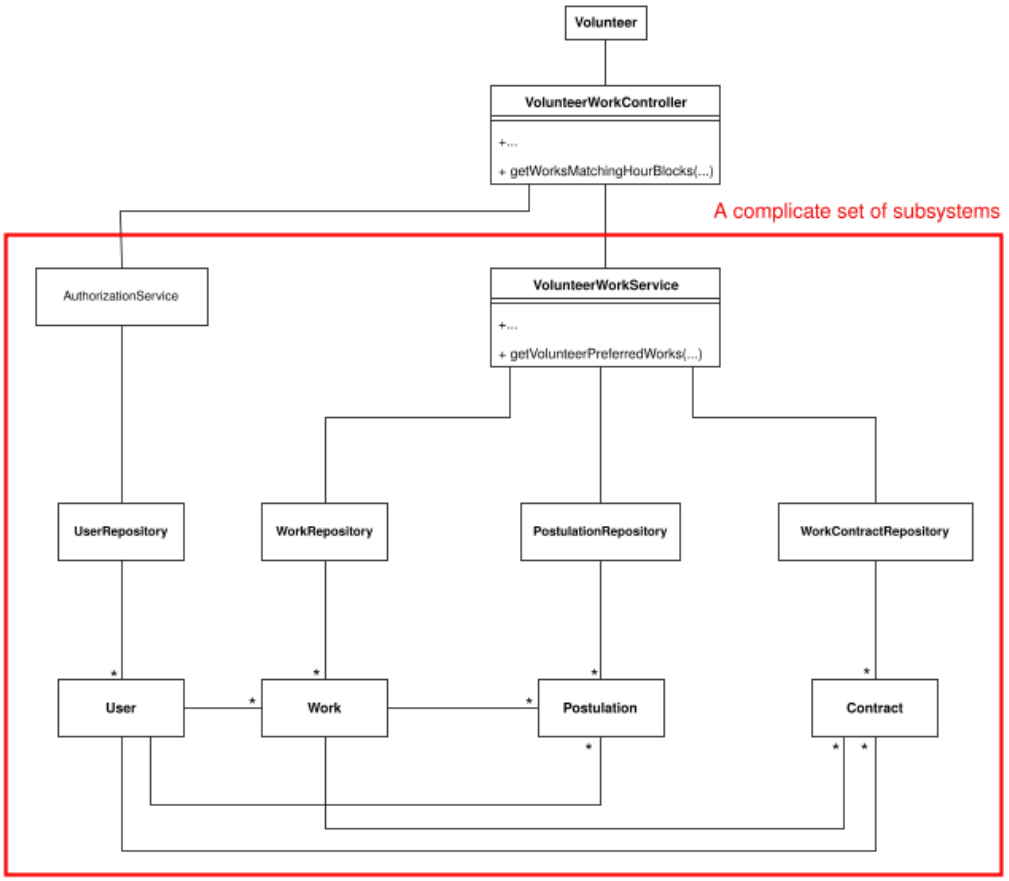
Patrones de diseños utilizados

Patrón Fachada (Façade)

El patrón de diseño estructural llamado Facade ofrece una manera sencilla de interactuar con una biblioteca, un framework o cualquier grupo de clases complejas, proporcionando una interfaz simplificada con el fin de ocultar la complejidad, haciendo que sea más fácil para cualquier usuario hacer uso de dicha biblioteca, framework, o grupo de clases.

Para el proyecto Volunteer Work Manager este patrón se utilizó para resolver el problema de permitir una comunicación sencilla entre el FrontEnd y el BackEnd con un bajo nivel de acoplamiento. Lo utilizamos para esconder del FrontEnd toda la lógica del negocio que se realiza en el BackEnd, dándole como única forma de comunicación los controladores y servicios los cuales son unas clases que reciben la petición del FrontEnd y la procesan respectivamente, para posteriormente darle respuesta, todo esto sin que el FrontEnd sepa cómo se llegó a la respuesta recibida.

Como ejemplo podemos observar el siguiente diagrama el cual refleja las interacciones entre las distintas clases para procesar la petición de obtener todos los trabajos como voluntario:

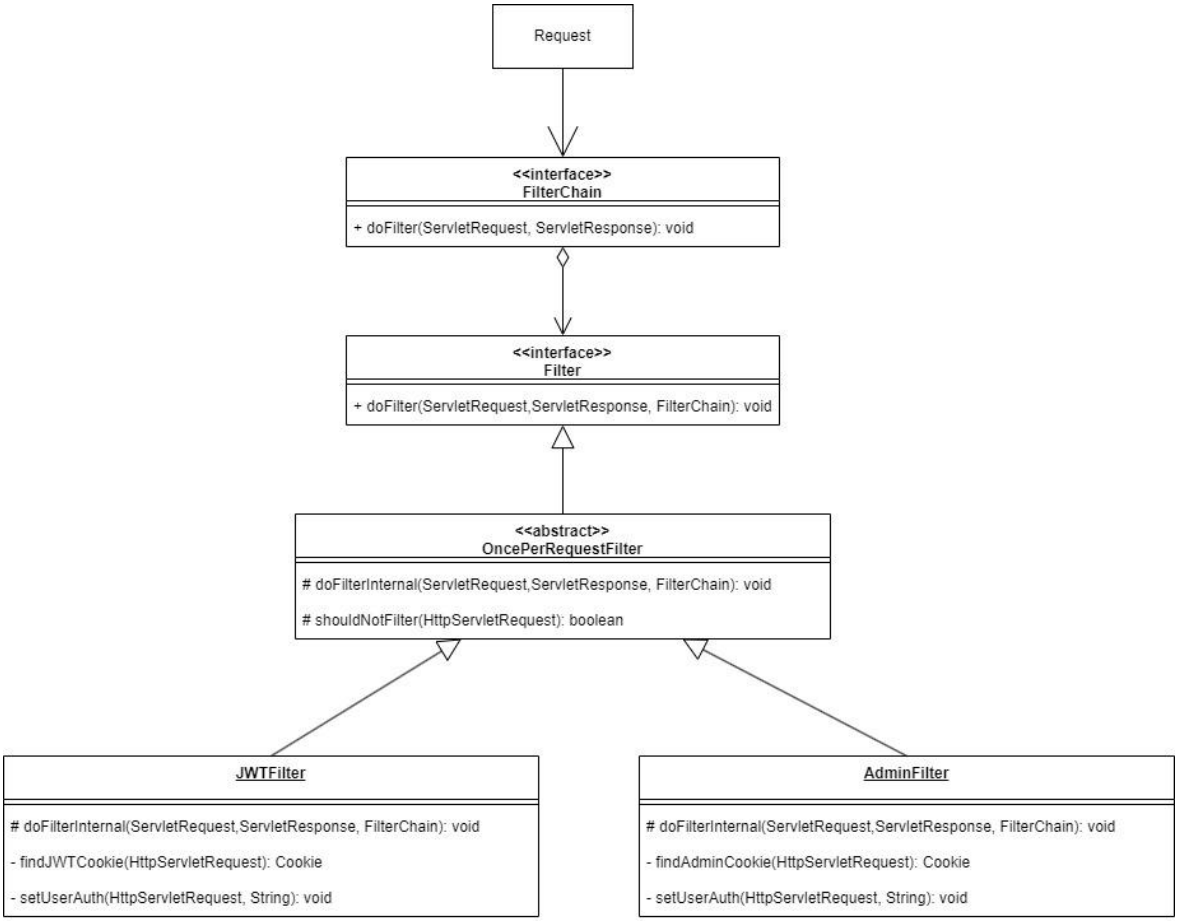


Patrón Cadena de responsabilidad (Chain of responsibility)

Chain of Responsibility es un patrón de diseño de comportamiento que permite la transferencia de solicitudes a lo largo de una cadena de manejadores. Cada manejador en la cadena, al recibir una solicitud, tiene la opción de procesarla o transferirla al siguiente manejador. De esta manera, se puede crear una cadena de responsabilidad para manejar solicitudes de manera flexible y eficiente.

Este patrón es utilizado para establecer una serie de condiciones que debe cumplir una petición HTTP y quien la envía, para determinar si está autenticado y autorizado para acceder a los recursos o funcionalidades a las que intenta acceder. Para esto, se utiliza en el proyecto el framework Spring Security, cuya arquitectura se basa enteramente en este patrón, ya que provee al desarrollador de una cadena de filtros de seguridad por los cuales debe pasar una petición antes de ser aprobada y procesada. Cada uno de estos filtros determina sus reglas y decide si rechazar, aprobar o redirigir la petición al siguiente en la cadena.

A continuación, se muestra un diagrama de clases instanciado en uno de los filtros de seguridad personalizados implementados por el equipo en el proyecto.



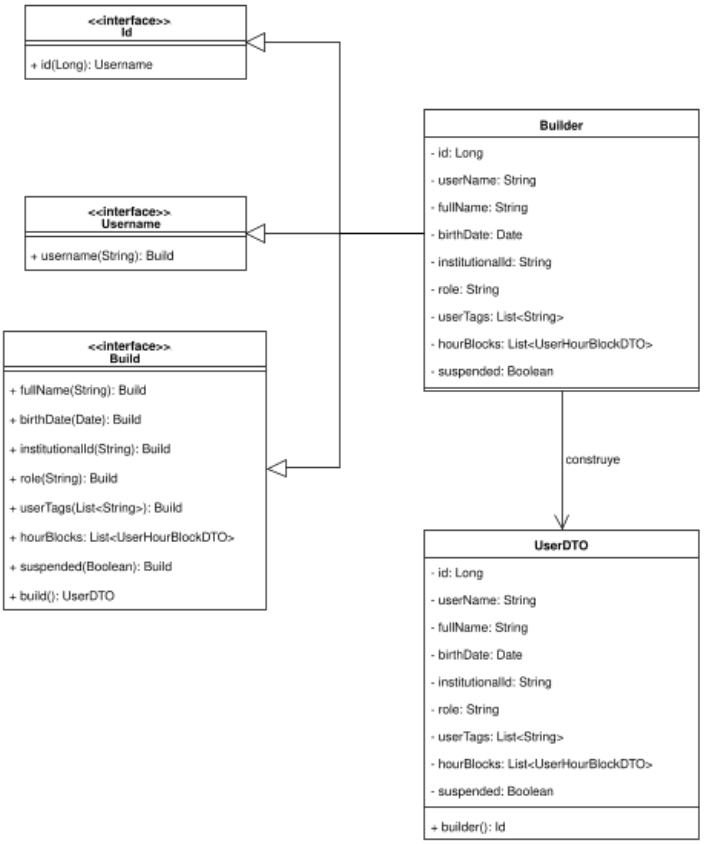
Por último, no se conocen todos los detalles internos de las implementaciones del framework, por lo que no todos los detalles de este se van a apreciar en el diagrama anterior.

Patrón Constructor (Builder)

Builder es un patrón de diseño que se utiliza en la creación de objetos complejos de manera gradual. Este patrón nos proporciona la capacidad de generar diferentes tipos y representaciones de un objeto mediante el uso de un mismo código de construcción. En otras palabras, con Builder podemos crear diferentes versiones de un mismo objeto de forma más sencilla y organizada.

Este patrón es utilizado para reducir la cantidad de clases necesarias para la transmisión de información mediante DTOs (Data Transfer Objects) entre los objetos límites en el FrontEnd, los controladores y servicios del BackEnd, dado a que mucho de estos objetos contenían la misma información pero con un atributos adicionales. Adicionalmente nos permite mejorar la legibilidad del código al no tener que utilizar constructores con grandes cantidades de parámetros, y permitirnos crear los atributos de este objeto de una manera más explícita.

A continuación, se muestra un diagrama de clases instanciado de la clase UserDTO, cuyas instancias pueden construirse mediante este patrón. Cabe destacar que las interfaces y clases relacionadas al patrón, son implementadas como objetos anidados en la clase UserDTO.



Ciclo de vida de una petición

En esta sección se describe el ciclo de vida de una petición en el sistema, desde su generación en el FrontEnd, su paso por las subcapas del BackEnd, y su regreso al FrontEnd. Para esto se examinarán las siguientes dos peticiones: Postularse a un trabajo y aceptar una postulación. Adicionalmente se presentarán los diagramas de secuencia de los casos de uso asociados a los requerimientos de estas peticiones, por lo que no solo se mostrará la secuencia de la petición a explorar, sino distintas peticiones asociadas a los casos de uso.

Postularse a un trabajo

1. FrontEnd:

El usuario ya previamente registrado y con una sesión iniciada como un usuario de tipo voluntario, accede a su panel, donde observa todos los trabajos disponibles, navega hasta encontrar el trabajo de su preferencia y al hallar este trabajo, presiona el botón de postularse. Si el trabajo es de tipo sesión, el sistema le pide confirmación al usuario para realizar la postulación, si el trabajo es de tipo recurrente el sistema pide la fecha de inicio y de fin de la postulación para posteriormente, realizar la postulación.

2. BackEnd:

2.1. Security Filters:

Se verifica si la petición cumple con todo lo requerido por los filtros CORS de Spring Boot. Luego pasa por el filtro personalizado para el administrador, pero como la petición no se hace a ninguna ruta exclusiva para este, es ignorado el filtro. Finalmente atraviesa el filtro de usuarios JWT en el cual se usa la cookie que recibe un usuario al iniciar sesión, y de este modo se permite que la petición sea procesada por los controllers.

2.2. Controllers:

Se verifica si existe algún controller en la dirección requerida por la petición y del mismo tipo. Como existe, se procede a hacer uso del servicio necesario para procesar la petición.

2.3. Services:

Se utilizan los datos proporcionados en el cuerpo de la petición y el username almacenado en la cookie, y luego se procesa toda esta información para ser almacenada en la base de datos.

2.4. Repositories y Entities:

Luego de procesar y generar los datos necesarios, se crea una nueva entidad postulación, la cual asocia al voluntario con un trabajo y una fecha de inicio y fin. Además la postulación tendrá una fecha de creación y el estado de

la misma, que inicialmente es “PENDIENTE”. Posteriormente usando esta entidad y el repositorio pertinente se almacena la nueva postulación en la base de datos.

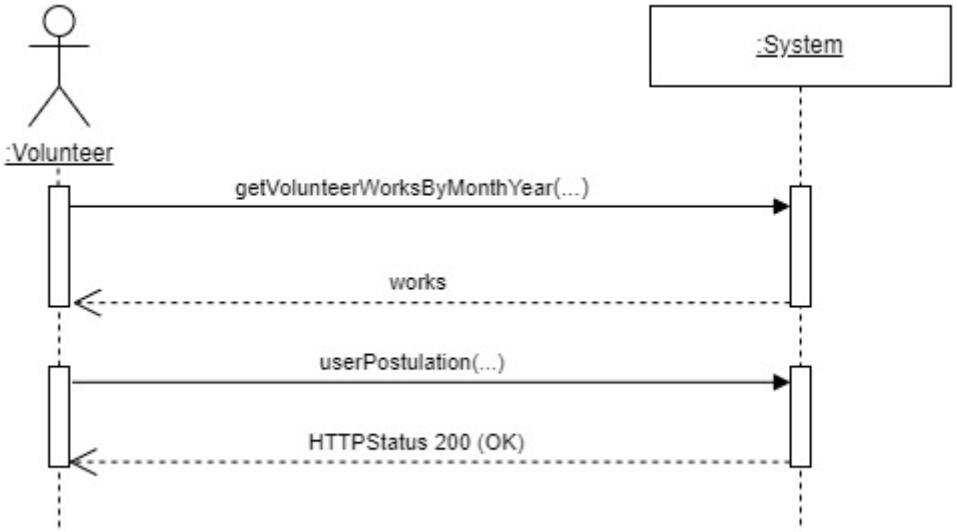
2.5. Controllers:

Al terminar de almacenar la postulación se le retorna una respuesta HTTP al FrontEnd indicando que se creó la postulación de manera exitosa.

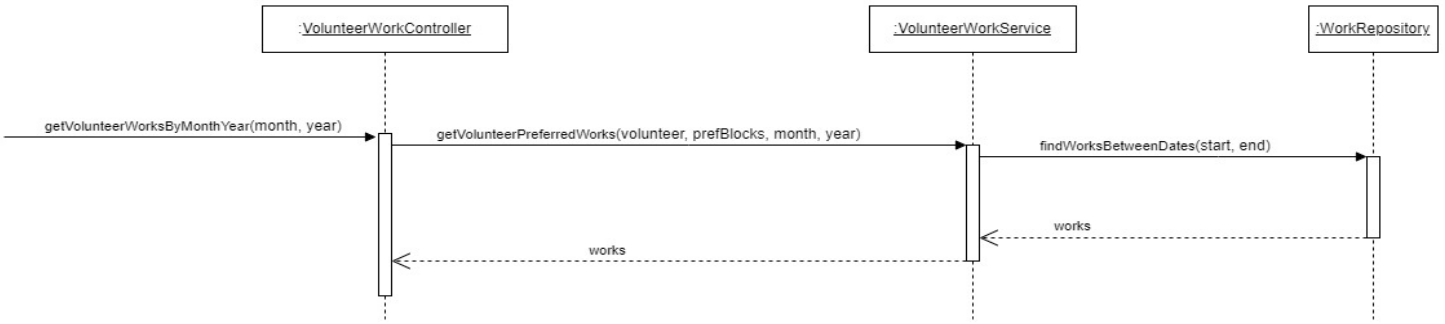
1.1. FrontEnd:

El usuario recibe un mensaje en forma de popup indicando que se creó la postulación satisfactoriamente.

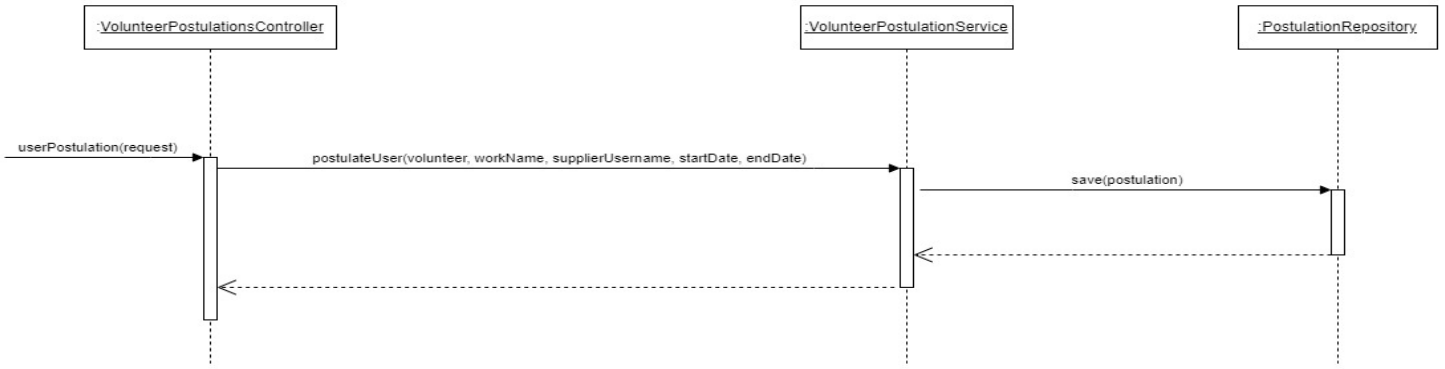
Para un mejor entendimiento de este ciclo de vida mostraremos su diagrama de secuencia basado en el caso de :



El diagrama anterior muestra de manera general como es la interacción entre el sistema y el voluntario, los siguientes diagramas ejemplifican los detalles de estas interacciones. Primero se presenta el diagrama de secuencia para obtener los trabajos disponibles.



Por último se tiene la secuencia de procesar la nueva postulación.



Aceptar postulación

1. FrontEnd:

   El usuario ya previamente registrado y logueado como un usuario de tipo proveedor, accede a su panel, donde observa todos los trabajos que ha creado, navega hasta encontrar el trabajo de su preferencia y al hallarlo, presiona el botón para visualizar la lista de todas las postulaciones realizadas a ese trabajo, navega por la lista hasta encontrar una postulación que le interese y presiona el botón para aceptarla.

2. BackEnd:

   2.1. Security Filters:

   Análogo a la petición de postularse a un trabajo.

   2.2. Controllers:

   Se verifica si existe algún controller en la dirección requerida por la petición y del mismo tipo. Como existe, se procede a hacer uso del servicio necesario para procesar la petición.

   2.3. Services:

   Se usan los datos proporcionados en el cuerpo de la petición y el username almacenado en la cookie, y se procesa toda esta información para crear la instancia del trabajo y las sesiones necesarias para que la postulación pueda ser marcada como aceptada.

   2.4. Repositories y Entities:

   Con la información procesada por el servicio, se crea una instancia de trabajo la cual servirá de certificado para indicar que el proveedor aceptó la postulación del voluntario, posteriormente se crean todas las sesiones de trabajo que existan entre la fecha en la que se aceptó la postulación y la fecha de fin de la postulación, finalmente se almacenan en la base de datos todas las nuevas entidades que se crearon usando los repositorios pertinentes.

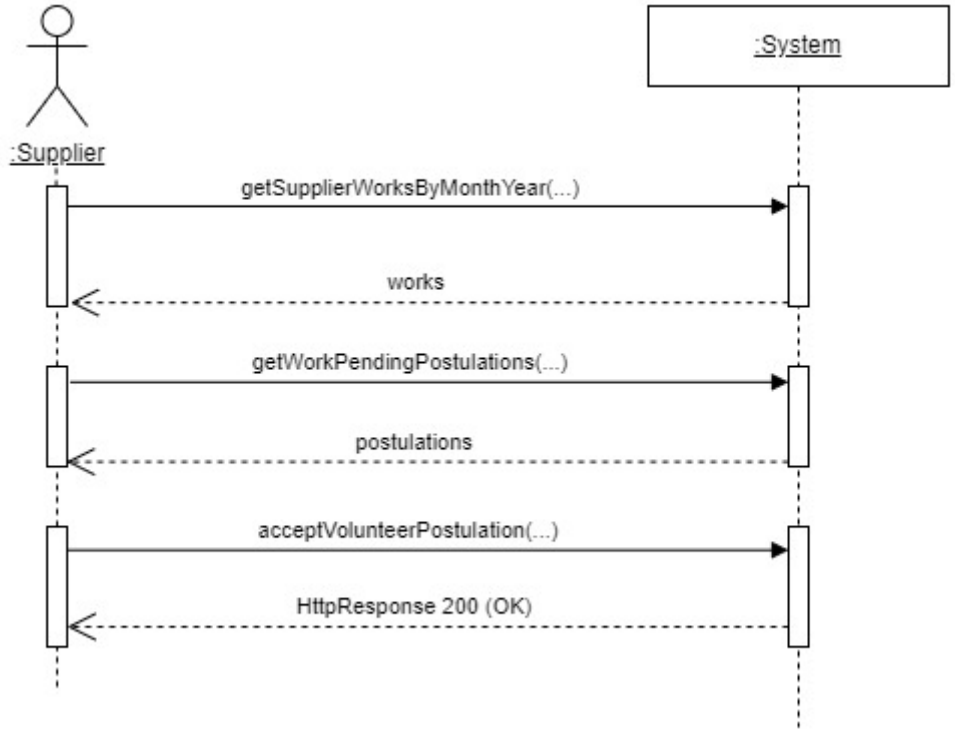
2.5. Controllers:

Al terminar de crear la instancia de trabajo y todas las sesiones necesarias, se retorna una respuesta HTTP al FrontEnd indicando que se aceptó la postulación de manera exitosa.

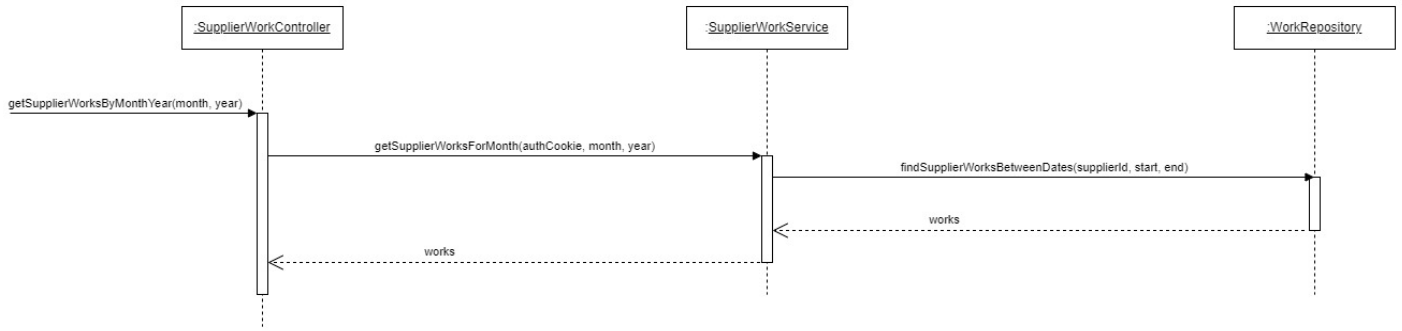
1.1. FrontEnd:

El usuario recibe un mensaje en forma de popup indicando que se aceptó la postulación satisfactoriamente.

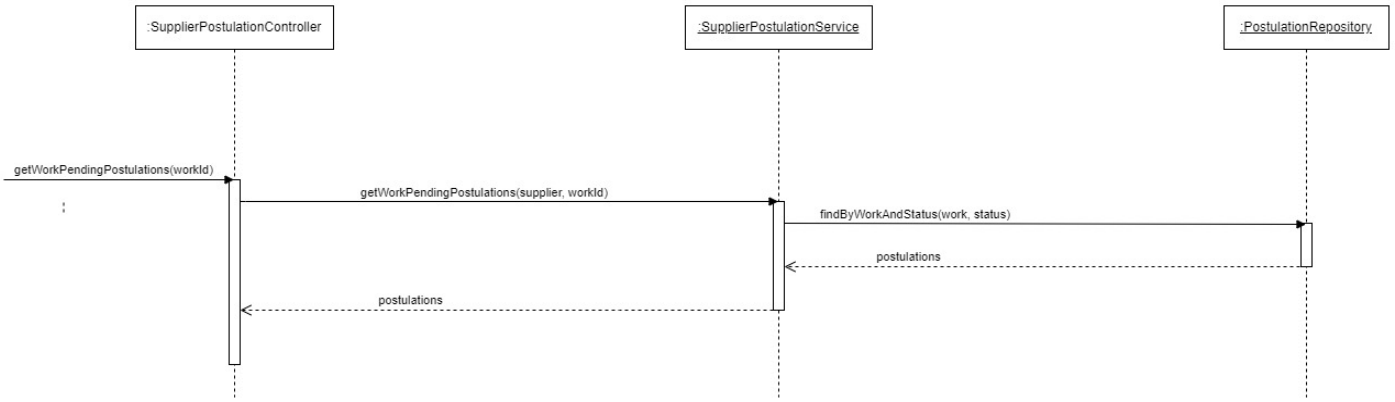
Al igual que en la petición anterior, en esta última parte veremos los diagramas de secuencia para procesar esta petición.



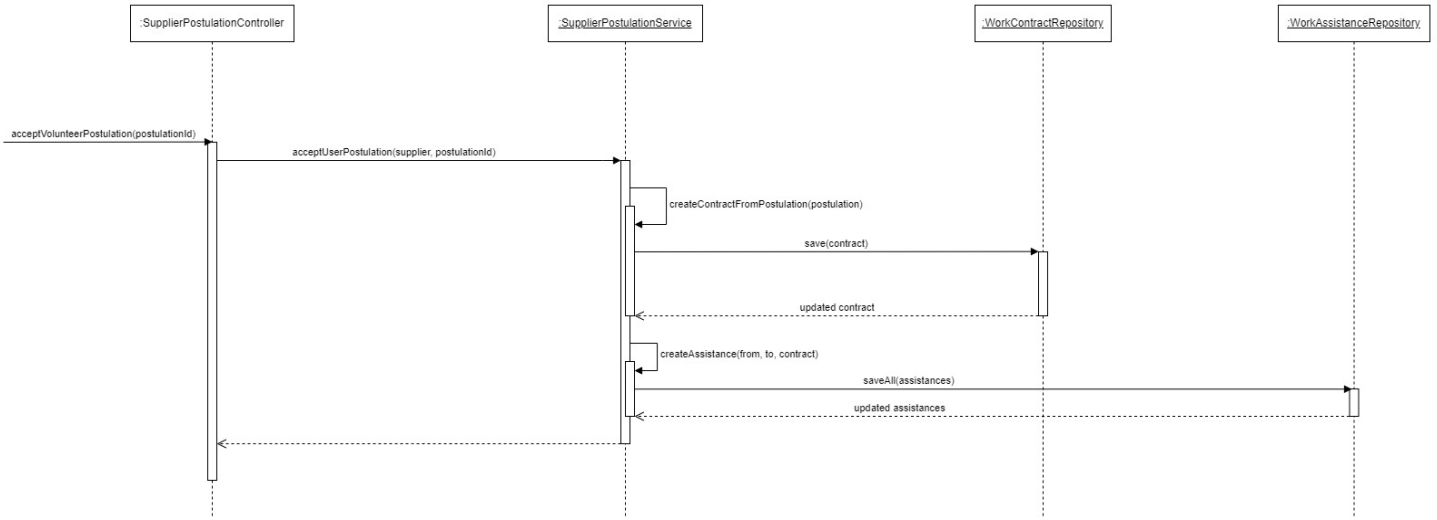
En este diagrama se muestra de manera general como es la interacción entre el sistema y el proveedor. Los siguientes diagramas presentarán los detalles de estas interacciones. El primer diagrama de secuencia mostrará cómo se obtienen los trabajos creados.



Luego se tiene el diagrama de secuencia para obtener las postulaciones realizadas para un trabajo en particular.



Finalmente, se muestra el diagrama de secuencia de aceptar una postulación.



Conclusiones

Tras realizar el proyecto y analizar los objetivos, el desarrollo y resultados, se ha demostrado que es posible implementar el 90% de los requerimientos planteados en el tiempo establecido de once semanas con el uso del modelo de análisis y de solución, la arquitectura por capas: Frontend, BackEnd y sus subcapas, así como el marco de trabajo SCRUM con metodología ágil que permitió avanzar notablemente en cortos periodos de tiempo.

Lograr los objetivos de cada Sprint fue laborioso para todo el grupo, especialmente el Sprint inicial ya que ninguno de los integrantes había desarrollado software usando el marco SCRUM y la metodología ágil. Para la mayoría hubo un nivel alto de dificultad al adaptarse a este proceso de desarrollo. En este sentido, la poca o nula experiencia laboral de algunos retrasó el trabajo en ocasiones.

Esta falta de experiencia laboral también trajo como consecuencia inconvenientes a la hora de organizar un equipo en el que cada integrante tiene habilidades y conocimientos diferentes, de manera que cada uno tenga tareas las cuales sean capaces de realizar en una cantidad de tiempo aceptable, sin generar bloqueos, y que además sea capaz de integrarse con los resultados de otras tareas que tengan relación.

Asimismo, existieron inconvenientes en la estructuración del proyecto, ya que se hizo desde cero. Las dificultades surgieron ya que el sistema desarrollado consta de múltiples capas, sistemas comunicándose entre sí y múltiples lenguajes, característica que no poseen los proyectos que se realizan a lo largo de nuestra carrera universitaria debido a que principalmente se desarrollan sistemas de mucha menor escala, en los cuales la complejidad se encuentra en el área teórica y no en el área de diseño. Adicionalmente, al estar limitados de tiempo (Sprints de 2 semanas) nos vimos en la necesidad de diseñar el sistema de manera muy pronta (principalmente la segregación de responsabilidades), lo cual resultó en decisiones de diseño erradas como: Implementar servicios adicionales de autorización en la capa de lógica de negocio a pesar de que ya se cuenta con una capa dedicada a esto.

En cuanto al stack seleccionado, para el equipo fue un reto aprender y realizar búsquedas eficientes de información acerca del stack al mismo tiempo que se usaba, ya que el tiempo es un recurso bastante limitado y se tenían que realizar tareas de implementación de funcionalidades al mismo tiempo que se aprendía a usar las herramientas. En ocasiones, la poca información obtenida y poca documentación buscada trajo como consecuencia realizar una misma tarea más de una vez ya que se debía repetir hasta que quedara un producto de calidad aceptable.

Siguiendo con los obstáculos generados por la poca experiencia utilizando el stack, la alta personalización que ofrece Spring Boot en su configuración nos generó dificultades para usar el framework, principalmente porque es sencillo encontrar todas las características ofrecidas de fábrica por este, pero lo contrario ocurre al momento de buscar información

sobre cómo realizar una implementación propia de esta. Un ejemplo de esto es la siguiente situación vivida por el equipo: Al momento de implementar un sistema de autenticación y autorización en la capa de BackEnd con el framework Spring Security, se realizó una lectura de la documentación oficial de este para adaptarlo a las necesidades del sistema, sin embargo, fue necesario recurrir a fuentes alternas de información para entender cómo realizar esta personalización, dado a que en la documentación oficial no se ofrece un nivel aceptable de detalle en este punto.

Por último referente al stack seleccionado, al equipo le resultó una dificultad el uso de la plataforma Docker en la cual se desplegó el software. La dificultad se encontró principalmente en la instalación del programa en las máquinas de los integrantes como también la actualización de imágenes de los contenedores cada vez que se realizaba un cambio en el BackEnd, esto debido a los escasos recursos de conexión a internet y de las máquinas usadas, lo cual ralentizó el flujo de trabajo considerablemente para algunos de los integrantes.

Luego de exponer las dificultades y retos del equipo, es momento de mostrar los objetivos que fueron logrados y una estimación cuantitativa.

Sobre la completitud del producto, se alcanzó el 90% del mismo, ya que se lograron implementar todos los puntos del documento de requerimientos, además de correcciones y recomendaciones del facilitador para mejorar la implementación y el diseño visual. El 10% faltante se debe a que el sistema aún requiere mejoras, por ejemplo, el componente encargado de listar trabajos fue implementado en los primeros Sprints del proyecto y su diseño no es el más adecuado debido a una mala segregación de responsabilidades (no solo se encarga de listar, sino mostrar, editar, crear y eliminar trabajos), lo cual generó problemas en los dos últimos Sprints dado a que debíamos recibir eventos de componentes anidados a la lista, los cuales pasaban por múltiples componentes antes de llegar al padre, el cual debía definir múltiples callbacks, resultando en un código complejo de seguir y desordenado.

Según las pruebas manuales realizadas al sistema, se puede señalar que no hay errores encontrados, aunque, no se puede afirmar con 100% de exactitud ya que el equipo no diseñó pruebas unitarias que se encargaran de encontrar posibles errores en el uso del producto, por lo cual no hay certeza de que no ocurra una falla. Sin embargo, se reitera que por lo probado exhaustivamente por el equipo, el software parece funcionar sin problemas.

Por otro lado, se tuvo que reconsiderar parte del diseño inicial ya que a medida que el proyecto crecía, el equipo observaba que algunas de las decisiones de diseño implican realizar malas prácticas como tener antipatrones, por ejemplo, las clases de entidades poseían conocimiento de la lógica de negocios, lo cual no es un antipatrón y mala segregación de responsabilidades al ignorar el objetivo de la capa de servicios. La implementación final está acorde a un 90% del diseño inicial debido a que se eliminaron parte de los antipatrones, lo cual trajo consigo un cambio al diseño planteado. En cuanto al diseño visual, se mapeó lo diseñado en Figma de tal manera que se conserva un 80%, esto se debió a que el equipo no

pudo plasmar todo lo que se quería en el diseño por falta de conocimiento, tales como el uso de iconografía en Figma, lo cual ocasionó que ningún ícono de las vistas (indicativos, íconos de botones, entre otros) fuera previamente ilustrado en la herramienta. En general, el producto completo corresponde en un 85% al diseño inicial tanto de lo visual así como de la implementación de funcionalidades.

En cuanto al aprendizaje, el equipo se enfrentó a varios retos y obstáculos mencionados anteriormente, sin embargo, se tuvo la capacidad de buscar información acerca del stack para mejorar las implementaciones, por ejemplo, se mejoró la estructura de los reportes ya que en el código de implementación inicial se repetía muchas instrucciones debido a que dependiendo del tipo de usuario, se tenían datos distintos, pero luego de investigar acerca de la interfaz Stream en Java e incluirlo en la implementación, se simplificó mucho el código y se aumentó la escalabilidad del mismo. También se tuvo la competencia para mejorar la organización del grupo y cumplir retos como trabajar mejor en equipo, por ejemplo, al principio del desarrollo del producto había descontento ya que no se obtuvo un buen rendimiento por falta de experiencia en temas de desarrollo por parte de algunos integrantes, lo cual se refleja en los dailies ya que los avances eran minúsculos y se generaban bloqueos por las mismas razones. Esto mejoró al punto de que el líder del grupo pudo confiar en el trabajo del mismo, ya que a lo largo de los Sprints aumentó el rendimiento con respecto al Sprint inicial.

Anexo 1

Detalles de los servicios del sistema

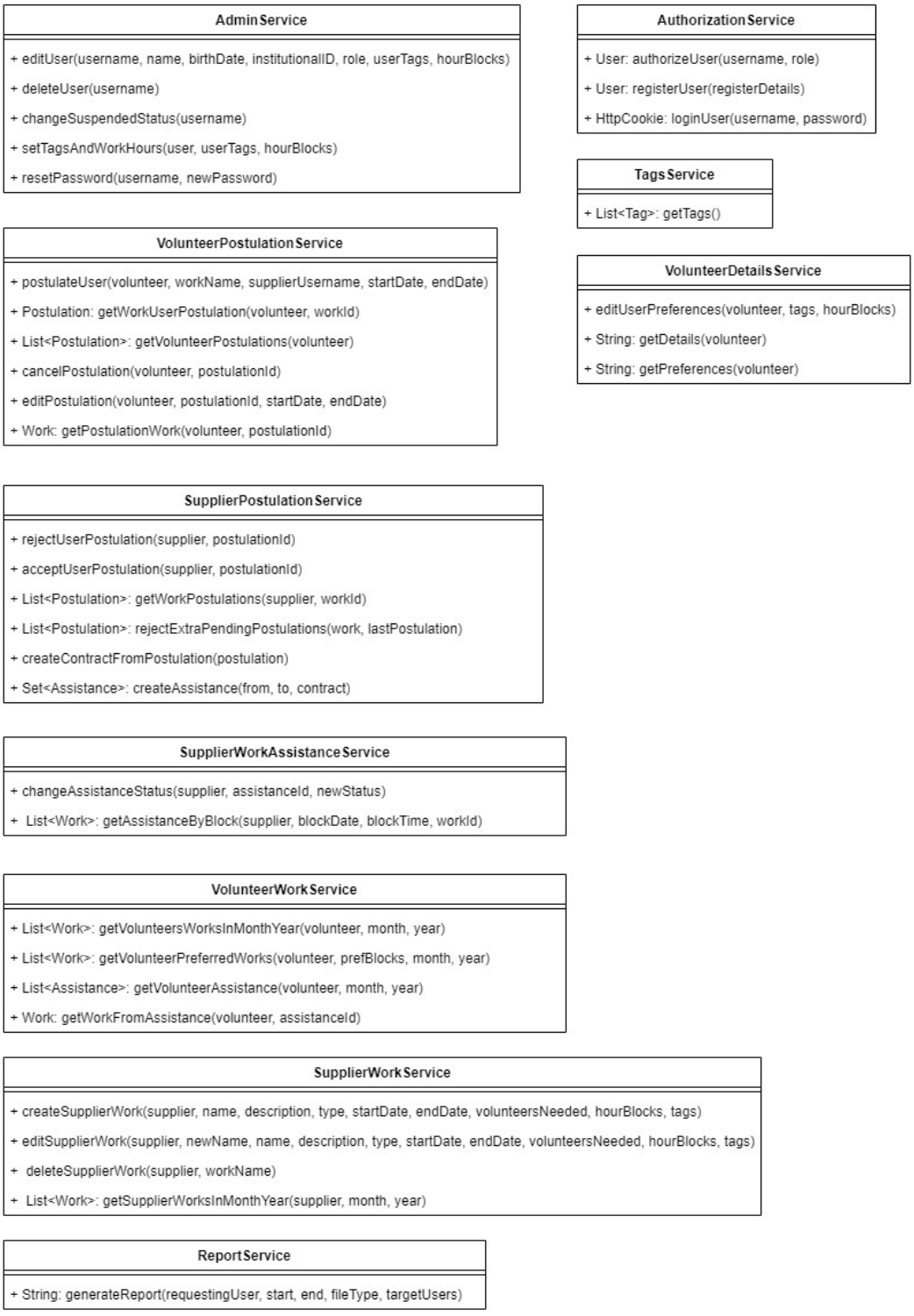


Figura 1

Detalles de los controladores del sistema.

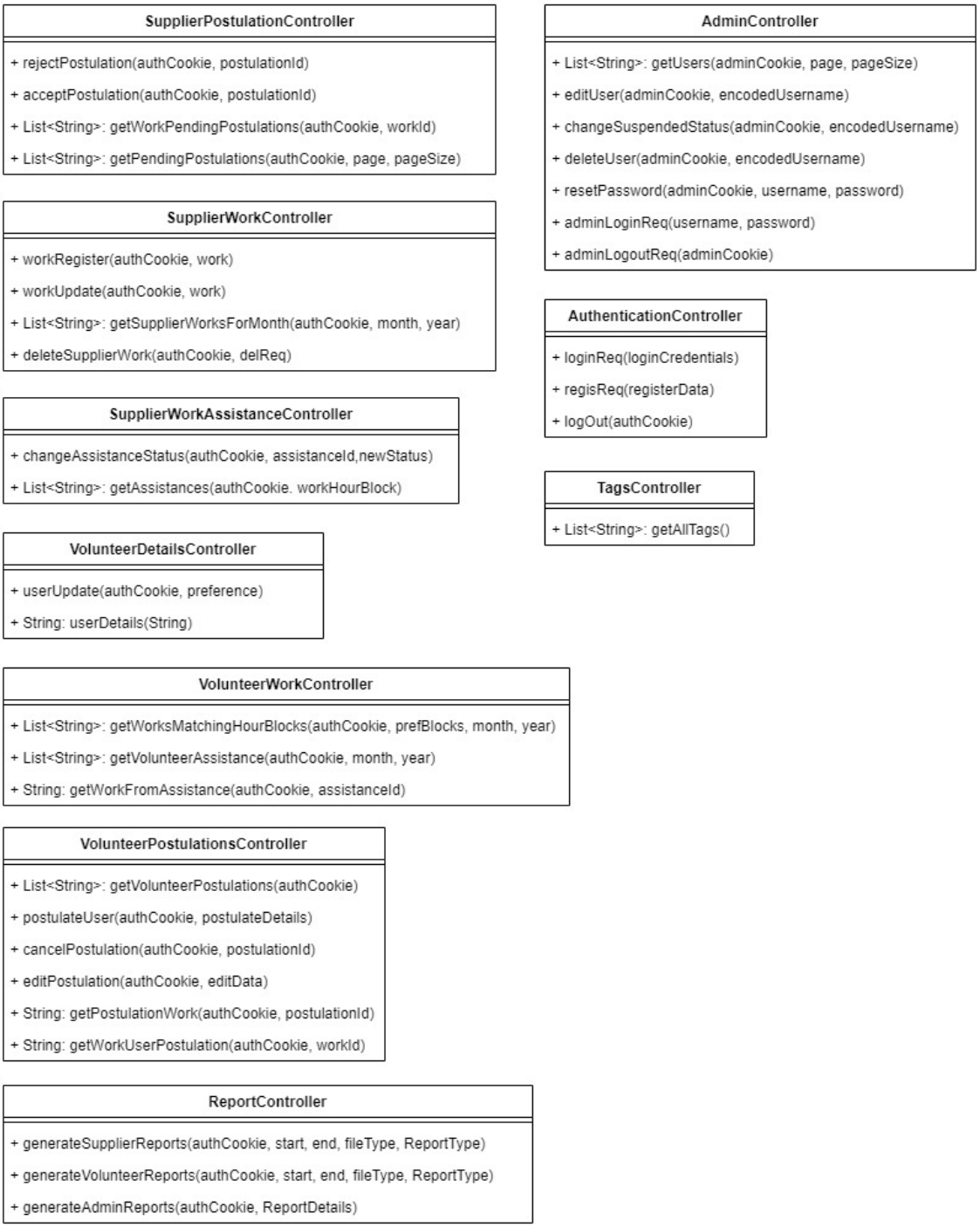


Figura 2

Detalles de los repositorios del sistema.

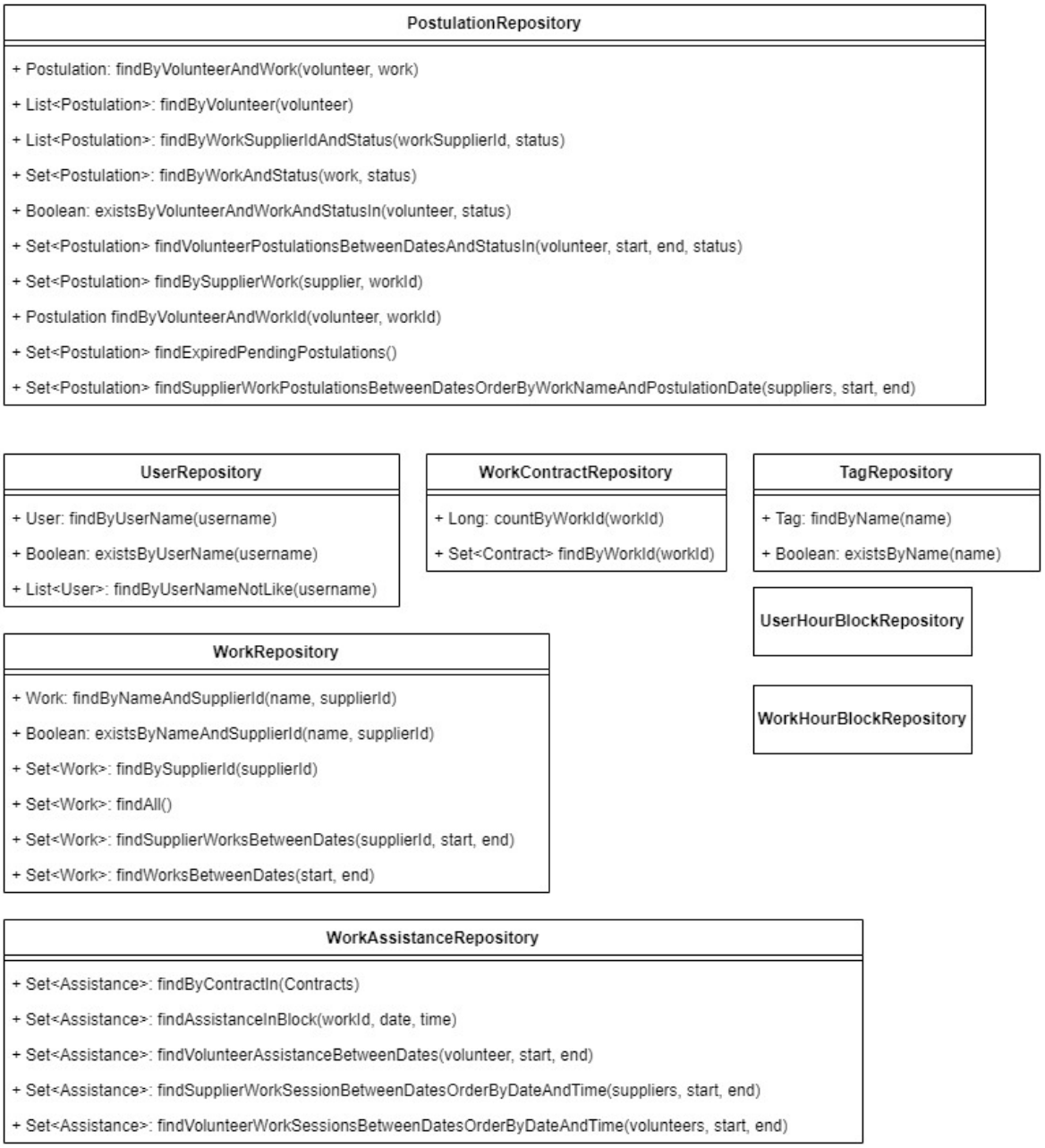


Figura 3

Detalles de las entidades de negocio del sistema.

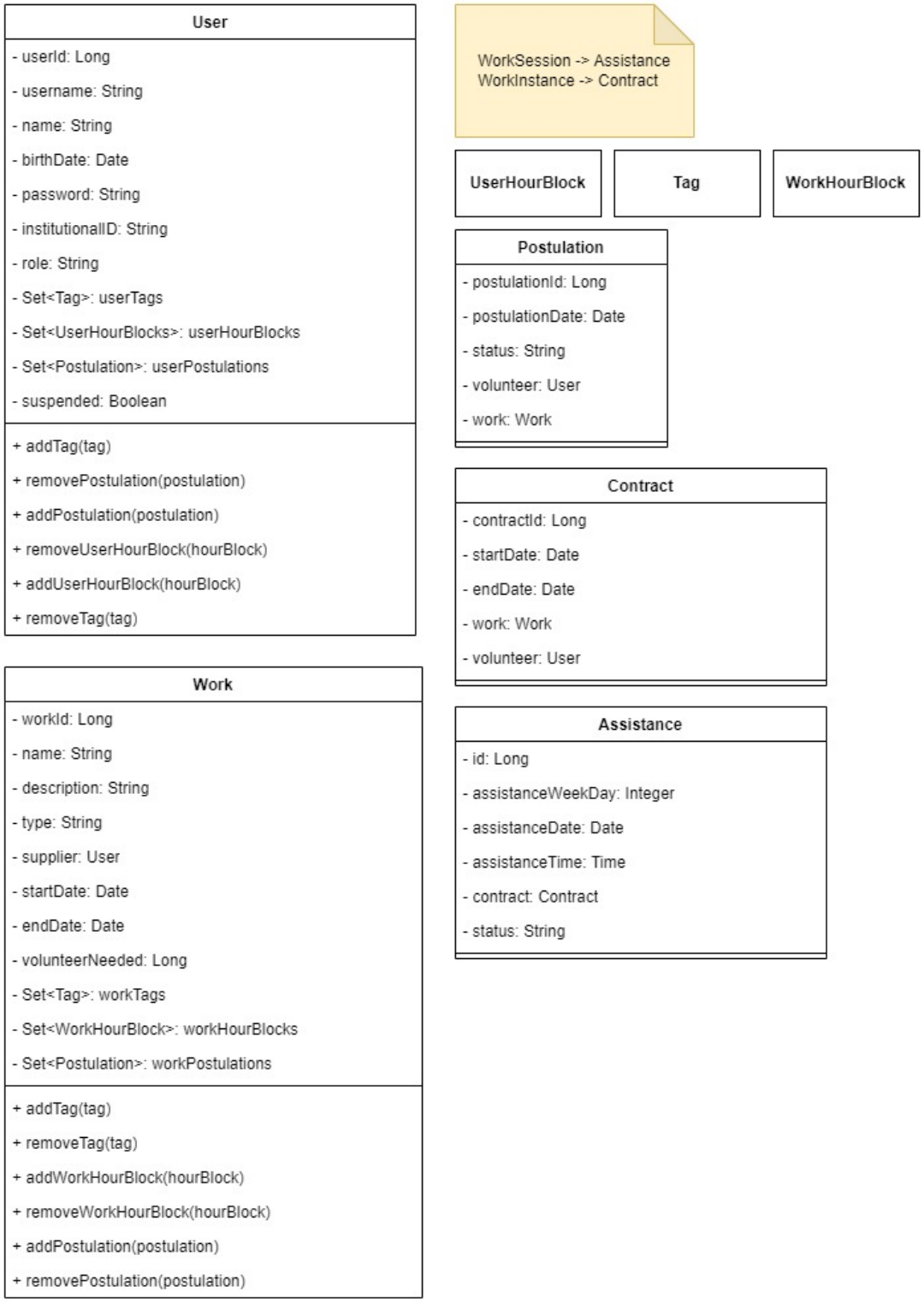


Figura 4

Funciones del FrontEnd para poder hacerle peticiones al BackEnd

Figura 5

Detalles de implementación del Report Generator

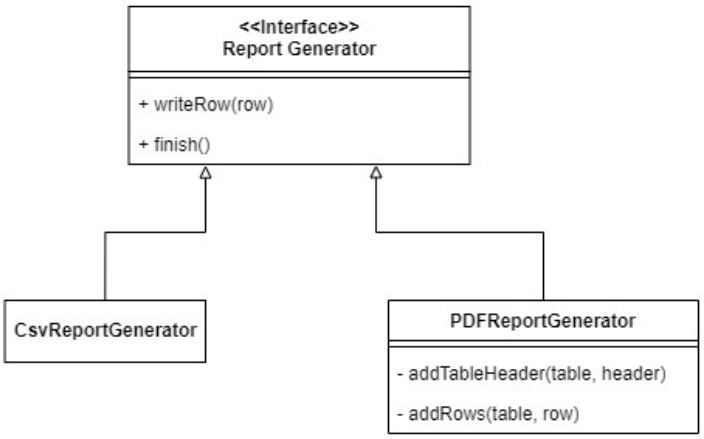


Figura 6

Bibliografía

1. The 2020 scrum GUIDETM. Scrum Guide | Scrum Guides. (n.d.). Recuperado de https://scrumguides.org/scrum-guide.html

2. Fakhroutdinov, K. (n.d.). The Unified Modeling Language. UML Diagrams - overview, reference, and examples. Recuperado de https://www.uml-diagrams.org/

3. Spring Framework: Introduction to Spring Framework | Spring Boot. Recuperado de https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/overview.html

4. Spring Security: Spring Security | Authentication, authorization, and protection against common attacks. Recuperado de https://docs.spring.io/spring-security

5. Architecture. Architecture - Spring Security. (n.d.). Recuperado de https://docs.spring.io/spring-security/reference/servlet/architecture.html

6. MozDevNet. (n.d.). Cross-origin resource sharing (CORS) - HTTP: MDN. HTTP | MDN. Recuperado de https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

7. JWT Token. Recuperado de https://jwt.io

8. Hibernate ORM. Recuperado de https://hibernate.org/orm/

9. ORM. Recuperado de https://hibernate.org/orm/what-is-an-orm/

10. JPA. Recuperado de https://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html

11. Elmasri R. & Navathe S. (2007). Fundamentals of database systems (5th ed.). Pearson/Addison Wesley.

12. Controllers y Rest Controller. Recuperado de https://stackabuse.com/controller-and-restcontroller-annotations-in-spring-boot/

13. ReactJS. Recuperado de https://react.dev

14. React Router. Recuperado de https://reactrouter.com/en/main

15. React DOM. Recuperado de https://www.javatpoint.com/what-is-dom-in-react.

16. MozDevNet. (n.d.-b). Fetch API - web apis: MDN. Web APIs | MDN. Recuperado de https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

17. Shvets, A. (n.d.). Patrón Facade. Refactoring Guru. Recuperado de https://refactoring.guru/es/design-patterns/facade

18. Shvets, A. (n.d.). Patrón Chain of Responsibility. Refactoring Guru. Recuperado de https://refactoring.guru/es/design-patterns/chain-of-responsibility

19. Shvets, A. (n.d.). Builder. Refactoring Guru. Recuperado de https://refactoring.guru/es/design-patterns/builder

20. Oliver Gierke, T. D. (n.d.). Spring Data JPA - Reference Documentation. https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods

21. Chapter 15. HQL: The hibernate query language. (n.d.). https://docs.jboss.org/hibernate/orm/3.5/reference/en/html/queryhql.html

22. Elmasri, Ramez; Navathe, Shamkant B. (2015). Fundamentals of database systems (Seventh ed.). Pearson.

23. Docker Overview. Docker Documentation. (2023, July 13). https://docs.docker.com/get-started/overview/