# Diseño

## Arquitectura

### Diagrama de paquetes

#### Un conjunto de letras negras en un fondo blanco Descripción generada automáticamente con confianza baja

Figura <TBD>: Diagrama de paquetes del backend

Imagen que contiene Interfaz de usuario gráfica

Descripción generada automáticamente

Figura <TBD>: Diagrama de paquetes del frontend

### Diagrama de despliegue

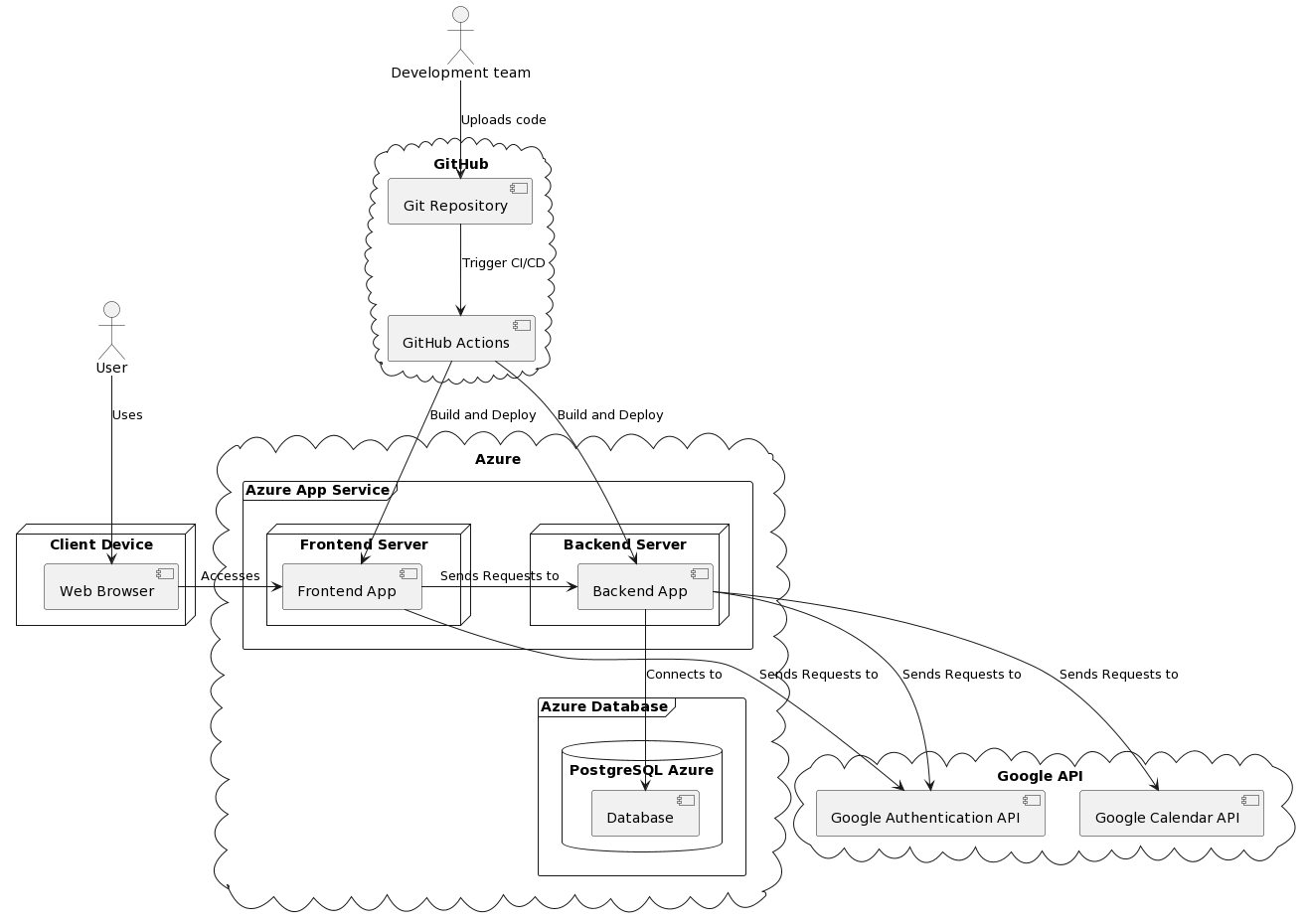


Figura <TBD>: Diagrama de despliegue de la aplicación

## Patrones de diseño utilizados

Los patrones de diseño son, de acuerdo con la definición de Erich Gamma en su libro Patrones de Diseño: “Descripciones de clases y objetos relacionados que están particularizados para resolver un problema de diseño general en un determinado contexto”. En el desarrollo del proyecto se han utilizado diversos patrones, en concreto han sido utilizados: Repository, Factory Method, Singleton, Data Transfer Object (DTO), Builder, Adapter, y Strategy.

Además, se ha utilizado la Inyección de Dependencias, un principio fundamental en el diseño de aplicaciones modernas, y que supona la base de los frameworks de desarrollo moderno como Spring o Micronaut.

A continuación, se proporciona una descripción detallada de cada uno de estos patrones, incluyendo cómo y por qué se han aplicado en el proyecto.

### Inyección de Dependencias

La inyección de dependencias es un principio fundamental en el desarrollo de software moderno y es esencial para los programas basados en el framework de Micronaut. La idea principal de la inyección de dependencias es promover la encapsulación del código eliminando referencias directas a las clases que utilizan, permitiendo por ejemplo definir campos que serán inyectados en base a interfaces, pudiendo decidir que implementación concreta se usará en base a diversas configuraciones. En lugar de que las clases creen las instancias de las clases de las que dependen, esas instancias se "inyectan" en las mediante un "contenedor" o "motor" de inyección de dependencias.

Por ejemplo, las instancias de los servicios (como UserServiceImp) y los repositorios (como UserRepository) son inyectadas automáticamente por Micronaut. Este enfoque reduce el acoplamiento entre las clases, hace que el código sea más modular y flexible, y facilita las pruebas unitarias al permitir que las dependencias reales se sustituyan fácilmente por objetos simulados (mocks) en las pruebas.

### Patrón Repositorio

El patrón Repositorio proporciona una abstracción de la persistencia de datos, permitiendo que la aplicación acceda a los datos sin tener que conocer los detalles de cómo se almacenan. En el proyecto se utiliza tanto en *GoogleCredentialRepository* como en *UserRespository*. El primero de ellos es ligeramente más complejo por definir métodos nuevos y ya que una vez explicado uno la implementación del otro se entiende por analogía.

GoogleCredentialRepository, al igual que UserRepository, es solo una interfaz que extiende de *CrudRepository* sin una clase de implementación declarada ya que Micronaut Data genera automáticamente la implementación de la interfaz en tiempo de compilación. Esta implementación generada se encarga de las operaciones de base de datos, como consultas, inserciones, actualizaciones y eliminaciones.

Micronaut Data utiliza el análisis de nombres de métodos para inferir las operaciones de la base de datos. Por ejemplo, el método findByKey se traduciría a una consulta SQL que busca una entrada en la base de datos donde la clave coincide con el parámetro proporcionado.

### Patrón Factory Method

El patrón Factory Method se utiliza para crear objetos sin exponer la lógica de creación al cliente. En este proyecto, JPADataStoreFactory es un ejemplo de este patrón. Esta clase proporciona un método para obtener una instancia de DataStore, ocultando los detalles de cómo se crea la instancia​. Esto permite la creación de diferentes tipos de DataStore si se añaden más implementaciones en el futuro sin cambiar el código que utiliza la factoría.

Se trata de una factory method ya que en el patrón factory method, una clase tiene un método que se utiliza para crear un objeto, y ese método se suele sobrescribir en las subclases para crear diferentes tipos de objetos. En este caso, el método de fábrica es getDataStore(), y se puede implementar de diferentes maneras en diferentes clases que implementan la interfaz DataStoreFactory.

Hay más ejemplos similares como el AuthorizationCodeFlowFactory que devuelve un objeto de tipo AuthorizationCodeFlow configurado para el proveedor necesario, en este caso solo existe el método getGoogleAuthorizationCodeFlow().

### Patrón Singleton

El patrón Singleton garantiza que una clase solo tenga una instancia y proporciona un punto de acceso global a ella, compartiendo por tanto estado. En este proyecto, al usar el framework Micronaut basado en inyección de dependencias basado en beans, la mayoría de dependencias inyectadas son en sí mismas un singleton. Tomemos como ejemplo cualquier implementación de un servicio, UserServiceImp por ejemplo, en este caso la clase esta anotada con @Singleton, por lo que cuando esta clase se inyecte mediante el motor de inyección de dependencias de Micronaut, este se encargará de crear una instancia si no existiese ya, y en caso contrario devolver la instancia existente.

Dado que en el proyecto no se instancian clases directamente cuando estas están anotadas con @Singleton, @Repository o @Factory ya que esto sería una mala práctica en el contexto de una aplicación basada en la inyección de dependencias, siempre se usará la misma instancia de todas estas clases.

### Patrón de objetos de transferencia de datos (DTO)

Este patrón se utiliza ampliamente en el proyecto, y su finalidad es abstraer las diferentes capas de la aplicación entre sí, así como ocultar detalles de implementación. De esta manera podremos cambiar la implementación de los servicios que los controladores seguirán manejando DTOs y el usuario de la API no será consciente de estos cambios. Lo mismo sucede si se cambian las entidades, los servicios entienden de objetos de modelo por lo que la implementación especifica de las entidades afectara a los repositorios, pero no a los servicios. Nos permite por tanto asegurar la encapsulación y reducir el acoplamiento.

Todos los resultados que la aplicación comparte mediante sus controladores son DTOs generados a partir de las clases del modelo mediante converters. Pero como he explicado también se usa para comunicaciones entre diferentes capas de la aplicación, un ejemplo podría ser la clase VerificationResult.

La clase VerificationResult se utiliza para transferir los resultados de la verificación de autenticación. La clase encapsula un valor booleano para indicar si la verificación fue exitosa y una cadena que proporciona una razón para cualquier fracaso. Esto simplifica la transmisión de datos y permite una mayor flexibilidad si los detalles de la implementación de la verificación cambian en el futuro​.

### Patrón builder

Se usa ampliamente en la construcción de DTOs en el proyecto. Su uso permite mejorar la legibilidad del código, simplificar la construcción de objetos, promover que los objetos sean inmutables y ayuda a cumplir con el principio de responsabilidad única.

La implementación concreta en el proyecto es la proporcionada por Lombok mediante la anotación @Builder con el parámetro de toBuilder puesto a true. Esta implementación nos permite crear objetos a partir de un builder invocado mediante un método estático de la clase anotada. El toBuilder nos permite que de un objeto existente podamos obtener un builder con los fields que tenia el objeto original, esto facilita mucho la modificación de objetos y lo hace de una forma legible, se usa por ejemplo en la actualización de los datos del usuario en UserServiceImp.

### Patrón adapter

Este patrón se utiliza para convertir la interfaz de una clase en otra interfaz que los clientes esperan. Es decir, ayuda a que las clases trabajen juntas que no podrían hacerlo de otro modo debido a las interfaces incompatibles. Se usa ampliamente para convertir objetos entre diferentes capas de la aplicación, permitiendo mantener un bajo acoplamiento. Un ejemplo podría ser la interfaz Event de la librería de Google Calendar que se convierte a una interfaz Meeting propia de NoMoreMeetings, permitiendo que en el futuro se puedan conectar diferentes proveedores, ya que solo será necesario un nuevo converter (implementación del patrón Adapter). También se ayuda a cumplir con el principio de responsabilidad única.

La clase GCalendarEventToMeeting implementa TypeConverter<Event, Meeting>, que convierte el objeto Event en un objeto Meeting. Esto se realiza en el método convert, que toma un objeto Event, extrae la información necesaria y la utiliza para construir un objeto Meeting mediante el patrón Builder.

### Patrón strategy

El patrón strategy es un patrón de diseño que permite seleccionar un algoritmo o estrategia en tiempo de ejecución. En lugar de implementar un único algoritmo directamente en la lógica de una clase, el código se refiere a una interfaz abstracta, lo que permite cambiar los algoritmos o estrategias libremente.

Se puede observar su uso en el proyecto en la clase GoogleAuthenticationProvider, que implementa la interfaz AuthenticationProvider. Respecto del patron strategy:

* AuthenticationProvider es la estrategia base o interfaz común que define cómo se deben implementar las estrategias concretas. Define un método authenticate() que todas las estrategias concretas deben implementar.
* GoogleAuthenticationProvider es una estrategia concreta que implementa una versión específica del método authenticate(). En este caso, implementa la autenticación a través de Google.

Se podría tener muchas otras clases, como MicrosoftAuthenticationProvider o YahooAuthenticationProvider que implementen AuthenticationProvider con su propia lógica de autenticación.

El código que llama a authenticate() no necesita saber qué tipo de proveedor de autenticación se está utilizando ya que simplemente llama al método en la interfaz AuthenticationProvider, y el patrón strategy se encarga del resto. Este desacoplamiento hace que el código sea más flexible y fácil de extender con nuevas estrategias de autenticación.

### Patrón de composición (React)

## La composición en React se refiere a cómo los componentes de React se pueden combinar y anidar para crear componentes más complejos. En lugar de heredar la funcionalidad de un componente "padre", en React, los componentes "hijos" se pueden pasar a los componentes "padre" como props y pueden ser renderizados por este mismo componente padre.

## Este patrón de composición se utiliza en todo React y se usa ampliamente en el frontend del proyecto. Por ejemplo, en el componente App​, se utiliza el componente Routes para renderizar diferentes componentes dependiendo de la ruta de la URL. Y este mismo componente contiene a su vez referencias a diferentes componentes como NavBar o UserProfile, permitiendo la navegación y haciendo uso de esta composición característica de React y que, aunque no es considerado un patrón de diseño en los libros escritos por Erich Gamma, Richard Helm, Ralph Johnson, o John Vlissides (también conocidos como el "Gang of Four") sí que puede considerarse comotal, siempre considerando que es característico de las aplicaciones escritas en React.

## Además, los hooks de React también facilitan la composición, permitiendo que la lógica del estado y los efectos secundarios se encapsulen en funciones reutilizables, que pueden ser utilizadas en múltiples componentes.

## Realización de casos de uso en diseño

En esta sección se describe cómo los componentes de software interactúan entre sí y con las interfaces externas para llevar a cabo los casos de uso identificados en el análisis. A continuación, se listan los casos de uso identificados en el análisis y se explica su implementación final.

### REGISTRARSE CON LA CUENTA DE GOOGLE

Para el caso de uso "Registrarse usando la cuenta de Google", el diseño se implementa principalmente a través de llamadas entre el frontend y Google por un lado y entre el backend y Google así como entre el backend y el frontend.

#### Flujo en el frontend

En primer lugar, el usuario accede a la interfaz del frontend y hace clic en el botón para iniciar sesión con Google. Este evento es manejado por la librería de OAuth para React de Google dentro del componente LoginComponent, que procede a realizar una llamada a la API de Google para obtener su idToken, con el que llamara a la ruta ‘/auth/login’ del backend. En este caso el backend se limitará a buscar el usuario en la base de datos, y al no encontrarlo devolverá un 401 Unauthorized con un mensaje de error especificando que el usuario no ha sido encontrado, por lo que el frontend entenderá que el usuario no está registrado.

Si el usuario no está registrado, se realiza una segunda llamada a la API de Google para solicitar los permisos necesarios para acceder al calendario del usuario. Después de obtener el token de identificación que incluye estos permisos, LoginComponent envía una solicitud de registro al backend en la ruta '/auth/signup'. Este proceso se realiza a través de las funciones de callback de la librería de Google antes mencionada y Axios para realizar llamadas al backend.

Interfaz de usuario gráfica, Aplicación, Tabla

Descripción generada automáticamente

Figura <TBD>: Diagrama de flujo del LoginComponent del frontend

#### Flujo en el backend

El backend recibe la solicitud a través de la ruta especificada en AuthController, ruta que no necesita de usuario : POST ‘/auth/signup’.

El controlador entonces llama al método verify(idToken) de GoogleTokenVerifier para verificar que es un token valido. Si el resultado de la validación es correcto entonces se llama al método signupUser(payload, codeToken) de AuthenticationService.

Dentro de dicho método se utiliza el método newTokenRequest del AuthorizationCodeFlow para obtener, llamando a Google, las credenciales necesarias en base al token recibido por parte del frontend. Una vez obtenidas son guardadas en base de datos usando el método createAndStoreCredential(tokenResponse, email) del mismo AuthorizationCodeFlow.

Una vez guardadas las credenciales de Google se guarda en base de datos el usuario correspondiente con su email como id, usando para ello el método save de UserRepository y el controlador devuelve un código 201 con el contenido del usuario en el body codificado como json, que el frontend usará para cargar la página de perfil del usuario.

Interfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente

Figura <TBD>: Flujo de registro de usuario en el backend

### INICIAR SESIÓN CON LA CUENTA DE GOOGLE

Para este caso el proceso es similar en el frontend para el caso de registro de usuario, ya que se lleva a cabo el mismo flujo descrito en el caso anterior, pero en la primera llamada al backend a la ruta POST ‘/auth/login’ esta sí devolverá un 200 OK con los datos del usuario, al estar ya registrado.

#### Flujo en el frontend

Se produce el mismo flujo que en el caso anterior, pero al encontrar el backend al usuario tras llamar al endpoint de login, se carga el usuario y se redirige a la página de perfil del usuario directamente.

#### Flujo en el backend

En este caso en la primera llamada al endpoint de login ‘/auth/login’ se valida el token pasado por el frontend en la clase GoogleAuthenticationProvider, al tratarse de un endpoint que requiere de autorización. Esta clase valida que el token sea validado mediante el método verify(token) de GoogleTokenVerifier, y una vez verificado usa el método loadCredential(email) de AuthorizationCodeFlow para cargar el los datos de Google del usuario correspondiente de base de datos. También, acto seguido, se procede a verificar que el token de acceso sigue siendo valido y se solicita uno nuevo si no es asi mediante los métodos getAccessToken(), que devolverá un nulo si este ha dejado de ser valido, y el método refreshToken() de la instancia de credenciales del usuario obtenida anteriormente.

Una vez validado tanto la validez del idToken recibido del frontend, la existencia del usuario en base de datos y la no caducidad de su token de acceso, se llama al AuthController, donde el método correspondiente a la ruta ‘/auth/login’ se encarga de llamar al método getUser(email) de UserService, el cual a su vez llama a findById(email) de UserRepository. El controller devuelve un código 200 OK y la información del usuario en formato json como parte del body.

Interfaz de usuario gráfica, Word

Descripción generada automáticamente con confianza media

Figura <TBD>: Flujo de login del usuario en el backend

### Configurar el tiempo de trabajo

En este caso en el frontend, una vez situado en la página del perfil del usuario permite al mismo modificar su horario de trabajo mediante un modal. Una vez el usuario guarda los cambios en el modal se llama al backend, concretamente a la ruta PATCH ‘/user ‘, donde se guardan los cambios en base de datos.

Para ello en el backend se realizan las tareas ya descritas correspondientes a cualquier endpoint securizado, y en el controller se llama al método updateUser(email, updateRequest) de UserService, el cual a su vez llama a los métodos findById(email) y update(updatedUser) de UserRepository respectivamente para poder obtener el usuario actual, modificar los datos y guardar los datos de nuevo en base de datos.

Diagrama

Descripción generada automáticamente con confianza media

Figura <TBD>: Actualización de los datos de un usuario

### Seleccionar calendarios para los cálculos

En este caso en el frontend, una vez situado en la página del perfil del usuario permite al mismo modificar su calendario seleccionado de un desplegable. El frontend llama desde UserProfile al backend para obtener la lista de calendarios del usuario, hace una petición GET a /user/calendars que retornara una lista de los calendarios del usuario en formato json.

Para ello en el backend se realizan las tareas ya descritas correspondientes a cualquier endpoint securizado, y en el controller se llama al método getUserCalendars(email) de UserService, el cual:

1. Inicializa el servicio de la librería de Google Calendar que realizara las peticiones necesarias para realizar llamadas a la API de Google Calendar con las credenciales del usuario y de la app.
2. Realiza una llamada a la API de Google Calendar para obtener los calendarios del usuario y los filtra eliminando los ocultos y eliminados.

Una vez el frontend tiene los calendarios se los muestra al usuario en un desplegable, el cual podrá cambiar y cuando en el frontend pulse en el botón de guardado el componente UserProfile del frontend enviara una petición PATCH a ‘/user’ donde indicara el id del calendario seleccionado por el usuario.

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media

Figura <TBD>: Flujo de modificación de calendario para realizar los cálculos

### Seleccionar periodo temporal para los cálculos

En este caso en el frontend, una vez situado en la página del perfil del usuario, el componente UserProfile muestra un campo de entrada que permite al usuario introducir el número de días que quiere analizar.

Una vez el usuario modifica el número de días a analizar, el estado local del componente (daysToAnalyze) se actualiza con el nuevo valor. Cuando el usuario esté satisfecho con los cambios, puede pulsar en el botón de guardado. Al hacer clic en el botón de guardado, el frontend enviará una petición PATCH a '/user' en el backend con la nueva configuración de días a analizar.

En el backend, se recibirá la petición en el endpoint correspondiente y se procederá a actualizar la información del usuario en la base de datos con los nuevos días a analizar tal. Para hacer esto, el backend ejecutará el método updateUser(email, updateUserDTO) del UserService, que se encargará de realizar la actualización correspondiente en la base de datos. Una vez que la actualización se ha realizado correctamente, el backend devolverá una respuesta con el estado 200 OK y los datos actualizados del usuario en formato JSON.

Finalmente, el frontend recibirá la respuesta del backend, actualizará el estado local del componente y el sessionStorage con los nuevos datos del usuario, y mostrará una alerta de éxito para confirmar que los cambios se han guardado correctamente.

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media

Figura <TBD>: Flujo de actualización del periodo de cálculo

### Configurar el tiempo entre reuniones que cuenta como tiempo de reunión

Este caso es muy similar al anterior. En este caso en el frontend, una vez situado en la página del perfil del usuario, en el componente UserProfile existe un campo de entrada que permite al usuario introducir el tiempo deseado (en minutos) entre reuniones.

Cuando el usuario modifica este valor, el estado local del componente (timeBetweenMeetings) se actualiza con el nuevo valor. Una vez que el usuario está satisfecho con los cambios, puede pulsar en el botón de guardado. Al hacer clic en el botón de guardado, el frontend envía una petición PATCH a '/user' en el backend con la nueva configuración de tiempo entre reuniones.

En el backend, esta petición es recibida en el endpoint correspondiente y se procede a actualizar la información del usuario en la base de datos con el nuevo tiempo entre reuniones. Para hacer esto, el backend ejecuta el método updateUser(email, updateUserDTO) del UserService, que se encargará de realizar la actualización correspondiente en la base de datos. Una vez que la actualización se ha realizado correctamente, el backend devuelve una respuesta con el estado 200 OK y los datos actualizados del usuario en formato JSON.

Finalmente, el frontend recibe la respuesta del backend, actualiza el estado local del componente y el sessionStorage con los nuevos datos del usuario, y muestra una alerta para confirmar que los cambios se han guardado correctamente.

Imagen que contiene Tabla

Descripción generada automáticamente

Figura <TBD>: Flujo de actualización del tiempo entre reuniones que se utiliza para hacer los cálculos

### Ver el porcentaje de tiempo gastado en reuniones y tiempo libre

Este caso de uso es más complejo, debido al procesamiento que se hace en el backend.

#### Flujo en el frontend

En este caso, en el frontend, una vez situado en la página "Time analysis", el componente WorkHoursComponent extrae la información del usuario de sessionStorage. Inicialmente, intenta cargar los datos de análisis de tiempo del usuario, si estos no están disponibles, realiza una petición GET a '/user/timeAnalysis' en el backend para obtenerlos.

Cuando el frontend recibe los datos, los almacena en el estado local del componente (timeAnalysisData). Luego, se calculan las horas ocupadas, tentativas, fuera de la oficina y libres, y se preparan para mostrarlas en una tabla y un gráfico de tarta. El cálculo de las horas libres se realiza restando las horas ocupadas, tentativas y fuera de la oficina del total de horas trabajadas. Todos estos datos se muestran en una tabla con sus respectivos porcentajes. El gráfico de tarta se genera a partir de estos datos, excluyendo cualquier elemento con valor 0. Finalmente, tanto la tabla como el gráfico de tarta se presentan al usuario en la página de análisis de tiempo.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Figura <TBD>: Flujo en el frontend de obtención del análisis de los datos de tiempo de trabajo.

#### Flujo en el backend

El método principal es getTimeSpentInMeetings(String userEmail). Este método es llamado desde el controller al recibir la petición GET del frontend. Y realiza lo siguiente:

1. Recupera la información del usuario a través del método getUser(String userEmail) del servicio UserService.
2. Calcula el intervalo de tiempo a analizar utilizando el método daysToAnalyze() del objeto user, y lo usa para obtener las fechas de inicio y fin.
3. A continuación, llama al método getMeetings(UserDTO user, LocalDate startDate, LocalDate endDate), el cual utiliza getCalendarMeetings(user, startDate, endDate) del servicio CalendarService para obtener una lista de todas las reuniones del usuario en el intervalo de tiempo especificado.
4. Esta lista de reuniones se pasa al método getMeetingsMap(List<Meeting> meetings), que crea un mapa de objetos MeetingDay (uno por cada día que tiene al menos una reunión) indexado por la fecha de las reuniones.
5. Después, se llama a calculateBusyTentativeOooHours(Map<LocalDate, MeetingDay> meetingsMap, UserDTO user, LocalDate startDate, LocalDate endDate), donde está la principal lógica de cálculo y que se encarga de calcular el total de horas ocupadas, tentativas y fuera de la oficina (OOO) para cada día en el intervalo de tiempo.
6. Después, el método calculateWorkedDays(LocalDate startDate, LocalDate endDate) calcula la cantidad de días laborales en el intervalo de tiempo, excluyendo sábados y domingos.
7. Finalmente, getTimeSpentInMeetings(String userEmail) devuelve una nueva instancia de TimeAnalysisResults, que contiene la cantidad total de horas ocupadas, tentativas y OOO, así como las horas trabajadas totales, calculadas multiplicando los días trabajados por las horas de trabajo del usuario obtenidas con getWorkingHours(UserDTO user).

Además, la clase contiene métodos privados que dividen la lógica para hacerla más legible, son los siguientes:

Los métodos calculateBusyTentativeOooMinutesForDay(Map<LocalDate, MeetingDay> meetings, LocalDate day, int resolutionMinutes, int workingHours, LocalTime startHour, LocalTime endHour) y calculateBusyTentativeOooMinutesForInterval(Meeting[] busyMeetings, Meeting[] tentativeMeetings, Meeting[] oooMeetings, LocalTime previousInterval, LocalTime interval, int resolutionMinutes) trabajan juntos para calcular el total de minutos ocupados, tentativos y OOO para cada día e intervalo de tiempo respectivamente.

El método adjustDayMinutesToWorkingMinutes(int busyDay, int tentativeDay, int oooDay, int workingMinutes, LocalDate day) se encarga de asegurar que la cantidad total de minutos (ocupados, tentativos y OOO) no exceda la cantidad de minutos de trabajo en un día. En caso de que sí lo haga, llama a subtractMinutesFromMeetings(int busyDay, int tentativeDay, int oooDay, int difference) para ajustar la cantidad de minutos.

Finalmente, hasMeetingInInterval(Meeting[] meetings, LocalTime start, LocalTime end) es un método auxiliar que verifica si existe alguna reunión dentro de un intervalo de tiempo específico.

Escala de tiempo

Descripción generada automáticamente con confianza baja

Figura <TBD>: Flujo en backend de cálculo de tiempo gastado en reuniones

## Diagrama de clases de diseño

### Backend

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza bajaFigura <TBD>: Diagrama de clases de los paquetes controller y authentication, y clases relacionadas de los paquetes domain y conversion

Figura <TBD>: Diagrama de clases de los paquetes service y controller, y clases relacionadas de los paquetes domain, conversion Interfaz de usuario gráfica

Descripción generada automáticamentey exceptions

Imagen que contiene Diagrama

Descripción generada automáticamente

Figura <TBD>: Diagrama de clases del paquete repository y clases relaciondas de service y authentication

### Frontend

Diagrama

Descripción generada automáticamente

Figura <TBD>: Diagrama de clases del frontend basado en la navegación y el uso de composición de componentes de React

## Descripción de las clases de diseño

### Clases de diseño del backend

El backend se estructura en varios paquetes que encapsulan diferentes áreas funcionales del software. Cada uno de estos paquetes contiene una serie de clases diseñadas para cumplir con los requisitos específicos de su dominio.

En este apartado, se proporciona una descripción detallada de las clases diferenciando por paquetes. Esta descripción abarcará el propósito de cada clase, sus responsabilidades principales y cómo interactúan con otras clases en el sistema.

#### Authentication

##### com.dfernandezaller.authentication

* **VerificationResult**: Esta es una clase abstracta que encapsula el resultado de un proceso de verificación. Contiene dos campos, un booleano isValid para indicar si la verificación fue exitosa y una cadena failureReason para almacenar el motivo del fallo en caso de que la verificación no haya tenido éxito.

##### com.dfernandezaller.authentication.google

* **AuthorizationCodeFlowFactory**: Esta clase es un singleton y una fábrica que se encarga de construir y proporcionar un objeto de tipo AuthorizationCodeFlow configurado para Google. Utiliza una DataStoreFactory y una GoogleConfiguration para configurar el flujo de autorización.
* **GoogleAuthenticationProvider**: Esta clase es un proveedor de autenticación que implementa la interfaz AuthenticationProvider. Su principal función es proporcionar un método para autenticar una solicitud HTTP y una solicitud de autenticación. Verifica el token de autenticación y, si es válido, carga las credenciales del usuario y verifica el token de acceso. En caso de que el token de acceso esté desactualizado, intenta actualizarlo. Este proveedor de autenticación utiliza un GoogleTokenVerifier y un AuthorizationCodeFlowFactory. Todas las peticiones a endpoints que requieran de autenticación son procesadas por esta clase.

##### com.dfernandezaller.authentication.google.dto

* **GoogleVerificationResult**: Esta clase hereda de VerificationResult y agrega un campo adicional, payload, que es un objeto de tipo GoogleIdToken.Payload. Esta clase encapsula el resultado de la verificación de un token de Google. Al igual que su clase padre, puede indicar si la verificación fue exitosa y proporcionar el motivo del fallo en caso contrario. Además, también proporciona la información útil contenida en el payload del token.

#### Controller

##### com.dfernandezaller.controller

* **AuthController**: Esta clase es un controlador responsable de manejar las solicitudes de autenticación y registro. El controlador AuthController utiliza los servicios AuthenticationService, UserService y GoogleTokenVerifier para autenticar a los usuarios y registrar nuevos usuarios en el sistema.
  + *loginGoogle(Authentication authentication)*: Este método maneja las solicitudes de autenticación y devuelve una respuesta HTTP que contiene los detalles del usuario si la autenticación es exitosa. Si el usuario no se encuentra, se devuelve un error HTTP 401 - No autorizado.
  + *signupGoogle(GoogleSignupData signupData)*: Este método maneja las solicitudes de registro de usuarios. Verifica el token de identificación proporcionado y si es válido, registra al usuario en el sistema. Si el token no es válido, se devuelve un error HTTP 401 - No autorizado.

##### com.dfernandezaller.controller.dto

* **CalendarDTO**: Este objeto de transferencia de datos se utiliza para enviar y recibir detalles del calendario del usuario. Los campos de este DTO incluyen el nombre, el ID, el color de fondo, la descripción, la ubicación y si es el calendario principal del usuario.
* **GoogleSignupData**: Este DTO se utiliza en el proceso de registro de un usuario. Contiene un codeToken y un idToken que se utilizan para autenticar y registrar al usuario en el sistema.
* **UpdateUserDTO**: Este DTO se utiliza para actualizar los detalles de un usuario existente. Los campos de este DTO incluyen las horas de inicio y fin de trabajo y almuerzo, el tiempo entre reuniones, el ID del calendario y los días a analizar.
* **UserDTO**: Este objeto de transferencia de datos se utiliza para enviar y recibir detalles del usuario. Los campos de este DTO incluyen el correo electrónico, el nombre, la URL de la foto del usuario, las horas de inicio y fin de trabajo y almuerzo, el tiempo entre reuniones, el ID del calendario y los días a analizar.

#### Service

Describiré directamente las implementaciones de las interfaces ya que cada interfaz tiene una implementación.

##### com.dfernandezaller.service.imp

* **AnalyzeTimeServiceImp:** es la implementación de la interfaz AnalyzeTimeService. Esta clase se encarga de analizar el tiempo dedicado por el usuario a las reuniones, y proporciona un desglose de dicho tiempo en diferentes categorías.
  + *getTimeSpentInMeetings(String userEmail)*: Este método recibe un correo electrónico de usuario, busca el usuario correspondiente y calcula el tiempo total que el usuario ha pasado en las reuniones. Esto se hace al obtener las reuniones del usuario en un período específico, clasificar estas reuniones por fecha, y luego calcular el tiempo ocupado, tentativo y fuera de la oficina (ooo) para cada día.
  + *getMeetings(UserDTO user, LocalDate startDate, LocalDate endDate)*: Este método recibe un usuario, una fecha de inicio y una fecha de fin, y luego llama al servicio CalendarService para obtener todas las reuniones del usuario dentro de ese rango de fechas.
  + *calculateWorkedDays(LocalDate startDate, LocalDate endDate)*: Este método calcula la cantidad total de días trabajados entre dos fechas dadas, excluyendo los sábados y domingos.
  + *getWorkingHours(UserDTO user)*: Este método devuelve la cantidad de horas de trabajo por día para un usuario específico. Esto se calcula restando el tiempo de almuerzo del tiempo total de trabajo.
  + *getMeetingsMap(List<Meeting> meetings)*: Este método clasifica una lista de reuniones por fecha, guardándolas en un Map donde la clave es la fecha y el valor es un objeto MeetingDay que contiene todas las reuniones de ese día.
  + *calculateBusyTentativeOooHours* *Map<LocalDate, MeetingDay> meetingsMap, UserDTO user, LocalDate startDate, LocalDate endDate)*: Este método calcula el tiempo total ocupado, tentativo y fuera de la oficina para cada día dentro de un rango de fechas dado.
  + *calculateBusyTentativeOooMinutesForDay* *(Meeting[] busyMeetings, Meeting[] tentativeMeetings, Meeting[] oooMeetings, LocalTime previousInterval, LocalTime interval, int resolutionMinutes)*: Este método calcula el tiempo ocupado, tentativo y fuera de la oficina en minutos para un día específico.
* **UserServiceImp**: Esta es la implementación de la interfaz UserService. Se encarga todas las operaciones relacionadas con los usuarios. Esta anotada tanto con @Singleton como con @Transactional, esta última anotación para garantizar que todas las operaciones de base de datos se manejen como una única transacción.
  + *getUser(String email)*: Se comunica con el repositorio para obtener los datos del usuario y los convierte en un objeto UserDTO.
  + *updateUser((String email, UpdateUserDTO requestDTO):* Obtiene de base de datos los datos actuales del usuario, actualiza los campos proporcionados en el objeto UpdateUserDTO, guarda el usuario actualizado en la base de datos y devuelve el UserDTO actualizado.
  + *getUserCalendars(String name):* Inicializa el servicio de calendario de Google, recupera los calendarios del usuario, filtra las entradas no válidas y las convierte en objetos CalendarDTO.
* **GoogleTokenVerifier**: Esta es una implementación de la interfaz TokenVerifier. Se encarga de verificar la validez de los token que se les pasa a su único método mediante el uso de la API de Google.
  + *GoogleTokenVerifier(GoogleConfiguration googleConfiguration):* Este constructor inicializa el GoogleIdTokenVerifier con un NetHttpTransport y una instancia de GsonFactory. Se configura el verificador para que use el clientId de la configuración de Google.
  + *verify(String token)*: Este método intenta verificar el token proporcionado utilizando el GoogleIdTokenVerifier. Si el token es válido, se crea un nuevo objeto GoogleVerificationResult con campo isValid a true, una cadena vacía como mensaje y la carga útil del token. En caso de que el token no sea válido, se crea un GoogleVerificationResult con campo isValid a false, un mensaje indicando que el token es inválido y una carga útil nula. Durante la verificación, si ocurre una GeneralSecurityException, se lanza una nueva SecurityException con un mensaje explicativo. Del mismo modo, si se produce una IOException, se lanza una BusinessException.
* **GoogleAuthenticationService**: Esta es la implementación de la interfaz AuthenticationService. Se encarga del proceso de signup de los usuarios de Google. Está anotada con @Singleton y @Transactional, esta última implica que todas las operaciones de la base de datos se manejarán como una única transacción.
  + *signupUser(GoogleIdToken.Payload payload, String token)*: Este método es el encargado de registrar a un nuevo usuario. Primero obtiene una respuesta de token a través del método getTokenResponse(token), luego intenta guardar las credenciales de Google con el método privado *saveGoogleCredentials(payload.getEmail(), tokenResponse)*. A continuación, se construye un nuevo objeto User con la información de payload, se guarda en la base de datos y se convierte en un UserDTO para ser retornado.
  + *saveGoogleCredentials(String email, TokenResponse tokenResponse)*: Este método privado verifica si ya existen credenciales para el correo electrónico dado, si es así, se lanza una BusinessException indicando que el usuario ya existe. En caso contrario, crea y guarda las nuevas credenciales y registra un mensaje de seguimiento.
  + *getTokenResponse(String token)*: Este método privado intenta crear token que contenga token de acceso, de refresco y la información del usuario a partir del idToken proporcionado. Si se produce una IOException, se lanza una BusinessException con un mensaje explicativo.
* **GoogleCalendarService**: Esta es la implementación de la interfaz CalendarService. Se encarga de aquellas operaciones relacionadas con el calendario de Google como el obtener los calendarios del usuario. Está anotada con @Singleton.
  + *getCalendarMeetings(UserDTO user, LocalDate startDate, LocalDate endDate)*: Este método recupera las reuniones de un calendario dentro de un rango de fechas especificado. Intenta inicializar el servicio de calendario de Google, realiza una petición para obtener los eventos en el rango de fechas establecido y los ordena por la hora de inicio. Luego, convierte cada evento en un objeto Meeting y los devuelve como una lista. En caso de error durante la comunicación con el servicio de Google, se lanza una BusinessException.
  + *getCalendarService(String userEmail)*: Este método privado intenta inicializar el servicio de calendario de Google. Para ello, crea una nueva conexión segura HTTP con Google y utiliza las credenciales del usuario para construir el servicio de calendario. En caso de un error durante la creación de la conexión segura, se lanza una BusinessException.

#### Exceptions

##### com.dfernandezaller.exceptions

* **BusinessException**: Esta es una clase que extiende RuntimeException, lo que significa que es una excepción no comprobada que puede lanzarse durante la ejecución normal del programa. Está anotada con @Value y @EqualsAndHashCode(callSuper = true), que son anotaciones de Lombok para generar automáticamente métodos toString(), equals(), hashCode() y getters para todos los campos, y un constructor que inicializa todos los campos. También permite que equals() y hashCode() consideren las propiedades de la clase de la que extiende (RuntimeException).
  + *BusinessException(String message)*: Este constructor toma un mensaje como argumento y lo pasa al constructor de la superclase (RuntimeException), lo que resulta en una excepción con un mensaje determinado.
  + *BusinessException(String message, Throwable cause)*: Este constructor toma un mensaje y una causa (otra excepción) como argumentos y los pasa al constructor de la superclase. Esto resulta en una excepción con un mensaje determinado y una causa subyacente.

#### Configuration

##### com.dfernandezaller.configuration

* **GoogleConfiguration**: Esta clase es responsable de mapear y alojar las propiedades de configuración específicas de Google en la aplicación. Las propiedades se obtienen del archivo de configuración de la aplicación (por ejemplo, un archivo application.yaml o application.properties) bajo el prefijo google. Esta clase es anotada con varias anotaciones:
  + @Data: Una anotación de Lombok que genera automáticamente los métodos getters, setters, equals(), hashCode() y toString() para los campos de la clase.
  + @Serdeable: Una anotación de Micronaut que indica que la clase puede ser serializada y deserializada automáticamente por Micronaut.
  + @ConfigurationProperties("google"): Una anotación de Micronaut que indica que esta clase aloja propiedades de configuración con el prefijo especificado ("google" en este caso).

Los campos de la clase incluyen:

* + clientId: El ID de cliente del proyecto de Google Cloud asociado.
  + clientSecret: El secreto de cliente del proyecto de Google Cloud asociado. Este es un valor confidencial que no debe ser expuesto o compartido públicamente.
  + applicationName: El nombre de la aplicación tal y como se registra en Google Cloud.

#### Model

##### com.dfernandezaller.model

* **Meeting**: Este es un record que representa una reunión. Esta clase tiene cuatro campos:
  + *type*: Un objeto de tipo MeetingType que representa el estado de la reunión.
  + *startTime* y *endTime*: Estos campos son objetos LocalDateTime que representan el inicio y el fin de la reunión. Están anotados con @JsonFormat(pattern = "yyyy-MM-dd HH:mm:ss") para garantizar que se serializan y deserializan correctamente con el formato de fecha y hora especificado.
  + *isAllDay*: Un valor booleano que indica si la reunión dura todo el día.
* **MeetingDay**: Esta clase representa un día que contiene varias reuniones. Contiene una lista de objetos Meeting. Esta clase tiene varios métodos para filtrar y obtener reuniones de diferentes tipos (aceptadas, tentativas, fuera de la oficina).
* **MeetingType**: Este es un tipo de enumeración que representa el estado de una reunión. Los posibles estados son: OOO (fuera de la oficina), TENTATIVE (tentativo), ACCEPTED (aceptado), CANCELLED (cancelado), OTHER (otro).
* **TimeAnalysisResults**: Este es un record que contiene los resultados de un análisis de tiempo. Contiene cuatro campos de tipo double que representan el número total de horas ocupadas, horas tentativas, horas fuera de la oficina y el tiempo total de trabajo.
* **TimeDistribution**: Este es un record que representa la distribución del tiempo entre las horas ocupadas, tentativas y fuera de la oficina. Antes de asignar los valores a los campos, estos se redondean a dos decimales.

#### Conversion

* **GCalendarEventToMeeting**: Este es un convertidor de tipo que se utiliza para transformar un objeto Event (una clase que representa un evento de Google Calendar) a un objeto Meeting (parte del paquete Model). En este convertidor, se extrae y se ajusta la información relevante de Event para cumplir con la estructura de Meeting.
* **GoogleCalendarListEntryToCalendarDTO**: Este convertidor de tipo transforma un objeto CalendarListEntry (una clase de la API de Google Calendar que representa una entrada en la lista de calendarios de un usuario) a un objeto CalendarDTO (un objeto de transferencia de datos utilizado y declarado en el paquete Controller).
* **UserDTOToUser** y **UserToUserDTO**: Estos convertidores de tipo se utilizan para transformar entre UserDTO (un objeto de transferencia de datos declarado y usado en el paquete controller) y User (la entidad que representa al usuario en base de datos).

### Clases de diseño del frontend

El frontend de se organiza en torno a varios componentes. Como es habitual en las aplicaciones escritas con React, cada uno de estos componentes tiene un papel específico y contiene la lógica y la presentación relacionadas con una pieza específica de la interfaz de usuario si bien muchas veces se componene unos de otros. A continuación, se proporciona una descripción detallada de los componentes clave.

Existen algunos componentes más de los aquí descritos cuyo papel es secundario, como ficheros css o clases que describen objetos que ya se han descrito en el backend como Meeting o User. Se omitirán ya que son virtualmente idénticos a los DTOs del backend.

* **App**: Este es el componente principal que encapsula toda la aplicación y define las rutas de esta. Se encarga de definir las diferentes rutas de la aplicación utilizando el componente BrowserRouter de React Router, que permite el enrutamiento en aplicaciones single page. Define rutas para la página de inicio, perfil de usuario, análisis de tiempo y una página de error
* **Layout**: Este componente actúa como una envoltura para las rutas del frontend. Renderiza la barra de navegación en todas las páginas y utiliza el componente Outlet de React Router para renderizar la ruta actual. El Outlet simplemente renderiza la ruta que se haya seleccionado dentro del componente Routes de la aplicación.
* **Landing**: Este componente define la vista de la página de inicio de la aplicación. Utiliza el sistema de grids de Material UI para organizar el contenido en una disposición de tres columnas. Cada columna tiene un ícono, un título y una descripción. Este componente se centra en mostrar información sobre la aplicación.
* **ErrorPage**: Este es un componente simple que muestra un mensaje de "Página no encontrada". Este componente se muestra cuando el usuario navegue a una ruta que no existe en la aplicación, proporcionando un manejo básico de errores de ruta.
* **WorkHoursComponent**: Este componente proporciona una vista detallada del análisis de tiempo del usuario. Utiliza una llamada al backend para obtener datos de análisis de tiempo y luego presenta la información al usuario de dos formas: una tabla y un pie chart.  
  Además, este componente implementa la lógica necesaria para gestionar la carga de datos y para manejar situaciones en las que los datos no están disponibles o el usuario no está autenticado. Funcionalidades clave del componente:
* Realizar una llamada a la API del backend para obtener los datos de análisis de tiempo del usuario.
  + Almacenar y gestionar los datos obtenidos del backend en el estado local del componente.
  + Transformar estos datos en un formato adecuado para ser presentados en una tabla y en un gráfico circular.
  + Presentar los datos al usuario tanto en forma de tabla como en un gráfico circular.
  + Manejar casos en los que los datos no están disponibles o el usuario no está autenticado, redirigiendo al usuario a la página de inicio en caso de que no esté autenticado.
  + Implementar un indicador de carga para informar al usuario de que los datos están siendo obtenidos.
* **Navbar**: Este componente es la barra de navegación superior que está presente siempre en la interfaz. En ella se muestra el logo de la aplicación, un enlace a cada vista y un botón para poder iniciar sesión con Google, registrarse con Google o hacer logout en función de la situación actual.
* **LoginComponent**: Este componente forma parte de la NavBar y contiene la lógica necesaria para llevar a cabo el login, el logout, el registro del usuario y los cambios visuales del botón correspondiente de la navbar. Funcionalidades clave del componente:
* Realizar una llamada a la API de Google para hacer el login del usuario.
* Realizar una llamada a la API del backend con los datos de la llamada a Google para proceder al login del usuario.
* Realizar una llamada a la API de Google para pedir los permisos necesarios en caso de que el backend responda que no está registrado al intentar hacer el login.
* Realizar una llamada a la API del backend para proceder al registro del usuario con los datos de la llamada a Google para obtener el token con los permisos de lectura de calendario.
  + Almacenar y gestionar los datos obtenidos del backend y de Google en el estado local del componente.
  + Manejar el estado del botón que ha de mostrarse en cada momento, así como mostrar una alerta si es necesario proceder al registro del usuario.
* **UserProfile**: Este componente es el componente que representa la vista del perfil del usuario. Permite ver la información del usuario como su horario laboral actual, para cuantos días ha de realizarse el análisis de las reuniones del calendario, el calendario a analizar seleccionado, foto y datos personales del usuario y el horario dedicado a almorzar. Además de mostrar esta información contiene la lógica necesaria para modificarla. Funcionalidades clave del componente:
* Realizar una llamada a la API del backend para poder cargar los calendarios del usuario, y mostrar un spinner de carga mientras se obtienen en el sitio donde se ubica el desplegable para seleccionar calendario.
* Realizar una llamada a la API del backend para obtener los datos del usuario.
* Realizar una llamada a la API del backend para modificar los datos del usuario.
* Mostrar la información relativa al horario laboral y el horario de dedicado a almorzar en un modal y realizar una llamada al backend si los datos son modificados.
  + Mostrar una alerta que informa al usuario si se han guardado los cambios con éxito.

## Modelo de base de datos

La base de datos es una base de datos relacional, en concreto se usa una instancia de PostgreSQL v.15 desplegada en la nube de Azure.

Una cosa a tener en cuenta en el diseño es que, aunque es un requisito conocido por el backend el que los IDs del usuario de la aplicación y de las credenciales sea el email y este deba coincidir, no se ha establecido esa relación a la hora de modelar la base de datos debido a que se ha procurado en todo momento hacer un diseño que pueda ser ampliado de forma sencilla a otros sistemas de calendario como Microsoft Outlook, y establecer esa restricción en base de datos puede ser contraproducente para ese objetivo. No obstante, esta restricción para el caso de Google Calendar es controlada en la lógica de la aplicación, las operaciones de guardado de usuarios son transaccionales para evitar tener estados inconsistentes en la base de datos.

A continuación, se puede ver una figura que describe el modelo E-R.

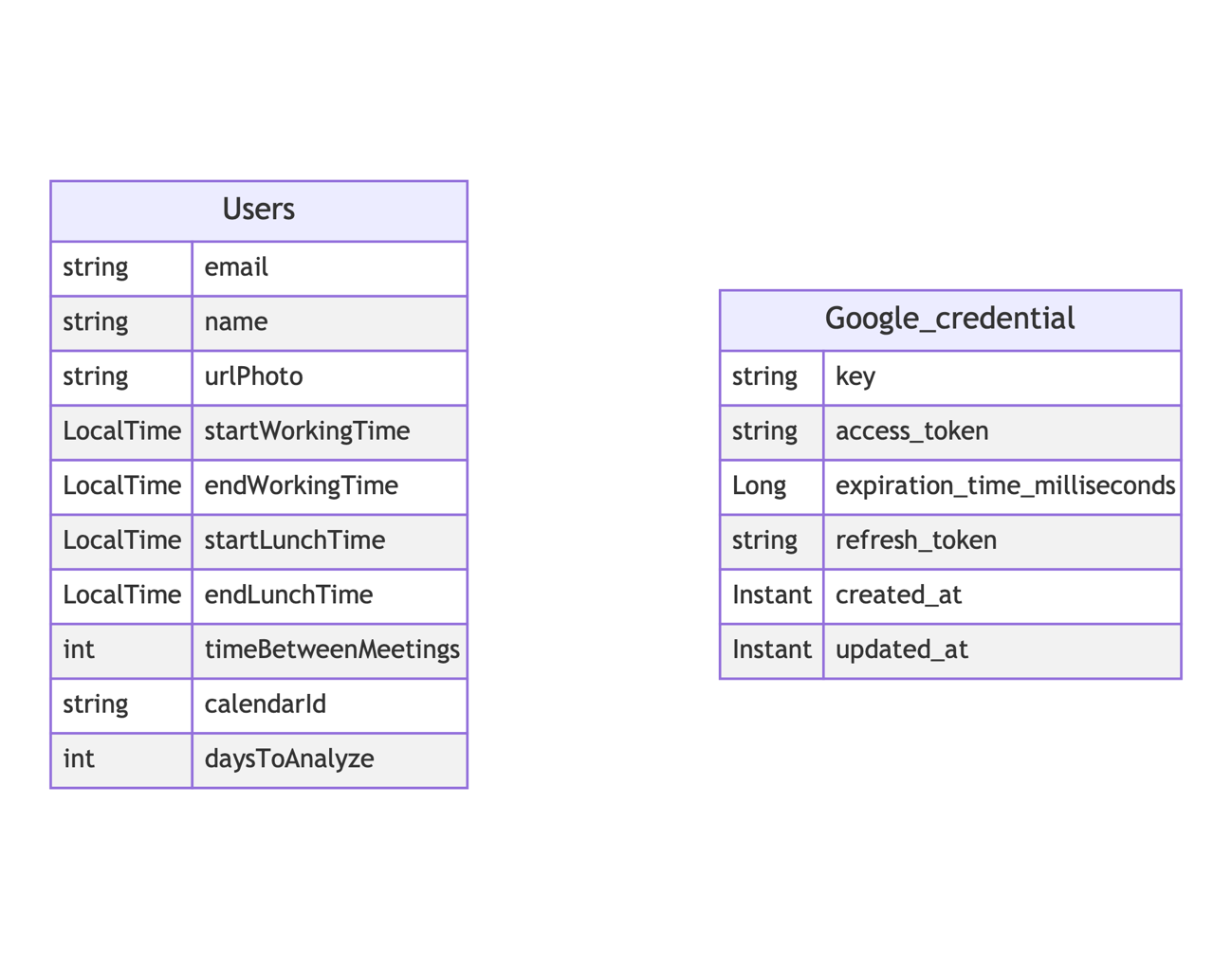


Figura <TBD>: Modelo E-R de la base de datos de la aplicación

### Descripción de las tablas

#### Users

Esta tabla almacena la información de los usuarios del sistema. Cada registro en esta tabla representa un usuario individual con sus respectivos detalles y configuraciones.

Atributos:

* email (clave primaria): Representa el correo electrónico único asociado al usuario.
* name: Almacena el nombre del usuario.
* url\_photo: Contiene la URL de la foto del usuario.
* startWorkingTime: Indica la hora de inicio de trabajo del usuario.
* endWorkingTime: Indica la hora de finalización de trabajo del usuario.
* startLunchTime: Representa la hora de inicio del tiempo de almuerzo del usuario.
* endLunchTime: Representa la hora de finalización del tiempo de almuerzo del usuario.
* timeBetweenMeetings: Especifica la duración en minutos entre las reuniones del usuario que ha de tomarse como tiempo de reunión durante el análisis.
* calendarId: Almacena el ID del calendario asociado al usuario que se usará para el análisis.
* daysToAnalyze: Indica la cantidad de días a analizar para programar reuniones y eventos en el calendario del usuario.

#### Google\_credentials

Esta tabla almacena las credenciales de Google utilizadas para autenticación y autorización en el sistema. Las credenciales están asociadas a un usuario específico y se utilizan para acceder a los servicios de Google.

Atributos:

* id (clave primaria): Representa la clave única que identifica las credenciales de Google.
* access\_token: Almacena el token de acceso utilizado para autenticación en los servicios de Google.
* expiration\_time\_milliseconds: Indica el tiempo de expiración en milisegundos para el token de acceso.
* refresh\_token: Contiene el token de actualización utilizado para obtener un nuevo token de acceso cuando el actual expira.
* created\_at: Registra la fecha y hora de creación de las credenciales.
* updated\_at: Registra la fecha y hora de la última actualización de las credenciales.

## Diseño de la interfaz

A continuación, se mostrará el diseño final de las vistas ya planificadas en la sección mockups del capítulo de análisis. La interfaz es responsive, por lo que se adapta al tamaño del dispositivo utilizado.

### Dispositivos de escritorio

#### Landing page

Esta vista pretende informar al usuario de las funcionalidades de la aplicación y servir como página inicial a la espera de que el usuario inicie sesión o se registre, según el caso.

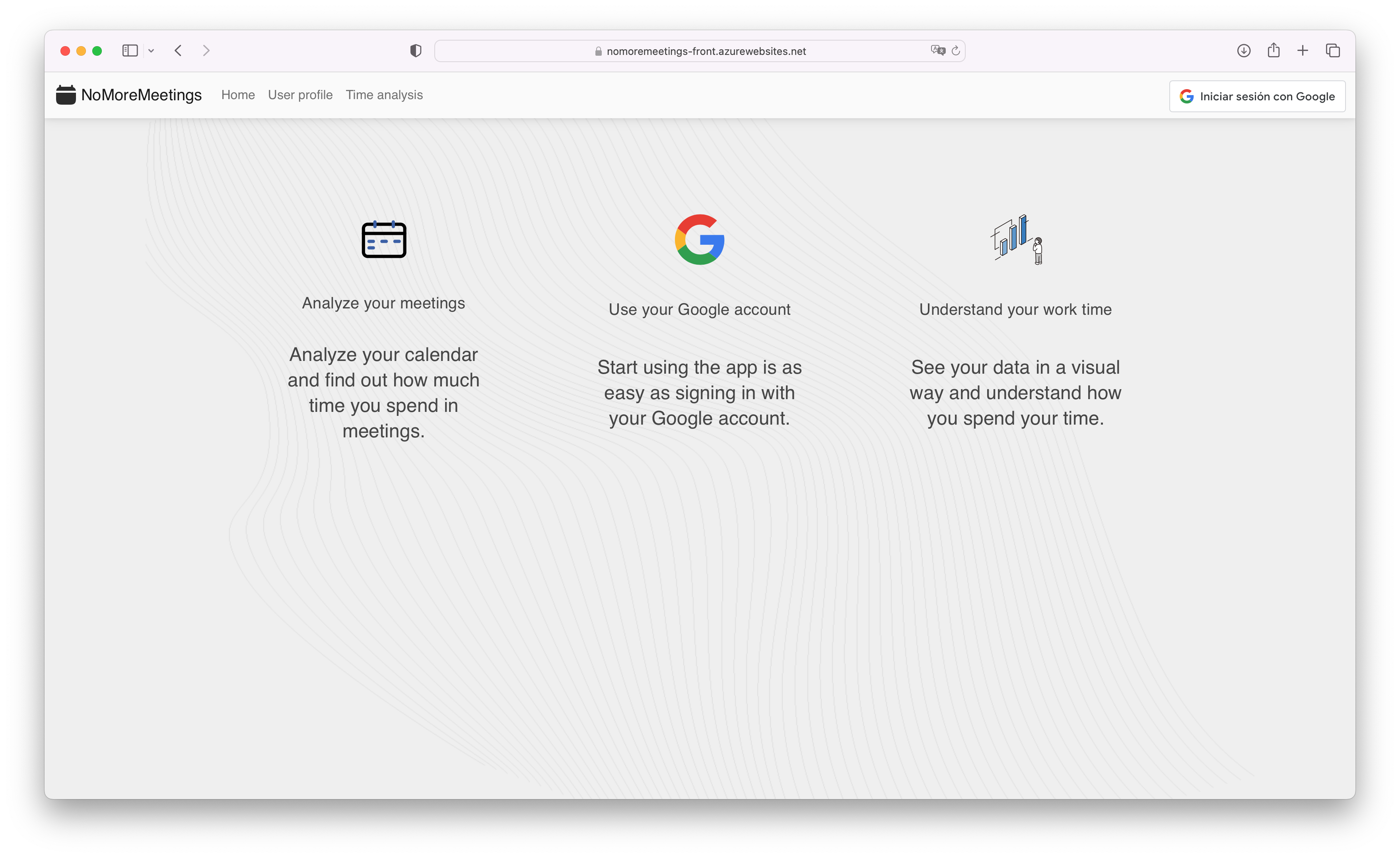


Figura <TBD>: Landing page de NoMoreMeetings vista en un PC

#### User profile

Esta vista muestra la información del usuario como nombre, apellidos, horario de trabajo y foto de perfil. Además, muestra configuraciones como el calendario que se usará para realizar los cálculos, el número de días a analizar y el tiempo entre meetings que se tomara como parte de la última meeting al realizar el análisis.

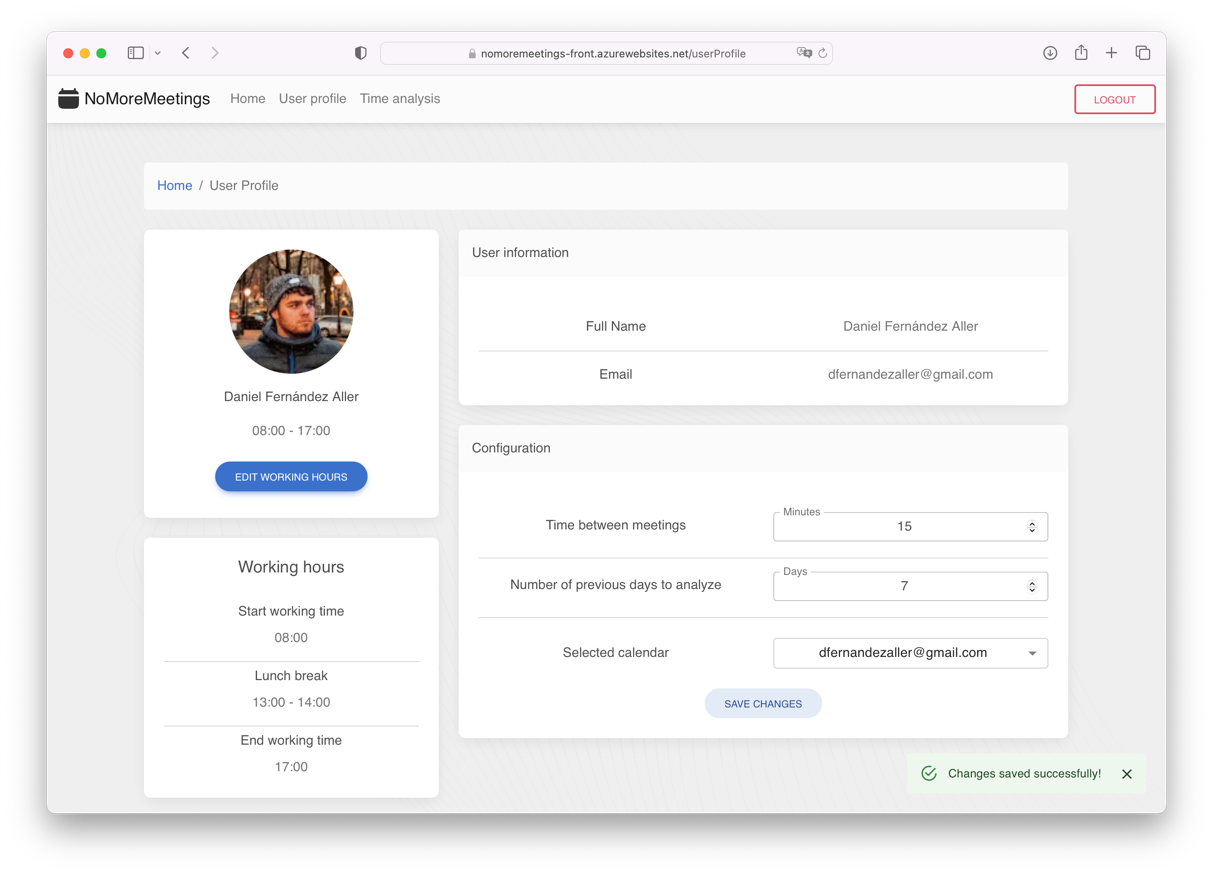


Figura <TBD>: Página del perfil del usuario vista en un PC

#### Time analysis

Esta vista muestra el resultado del análisis de los datos del calendario seleccionado del usuario en base a su configuración.

#### Interfaz de usuario gráfica, Time Analysis, Sitio web de NoMoreMeetings

Figura <TBD>: Página de análisis de tiempo del calendario del usuario

### Dispositivos móviles

Veamos ahora la interfaz de usuario en dispositivos móviles.

### Interfaz de usuario gráfica, Texto, Aplicación Descripción generada automáticamente

Figura <TBD>: User profile en iPhone

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente  
Figura <TBD>: Modal en iPhone

Interfaz de usuario gráfica

Descripción generada automáticamente  
Figura <TBD>: User profile en iPhone

Tabla

Descripción generada automáticamente  
Figura <TBD>: Time analysis en iPhone

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamente  
Figura <TBD>: Landing page en iPhone con barra de navegación desplegada