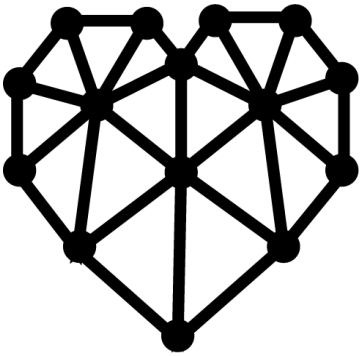
Memoria TP6

30/05/2022

Marcos Garralaga Blasco 795936
César Borja Moreno 800675
Clara Cerdán Torrubias 800676
Jesús Roche Soguero 798893
Juliana Zordan Cestari 776390

Arquitectura Software



MeetMe

Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Índice

Introducción

En este documento se describe el desarrollo de una red social orientada a citas desde el punto de vista arquitectural. La aplicación recibe el nombre de "MeetMe" y está enfocada en conceder una primera cita a dos personas desconocidas. Esta se ha desarrollado por Clara Cerdan (800676), Cesar Borja (800675), Juliana Zordan (776390), Jesús Roche (798893) y Marcos Garralaga (795936).

Toda la documentación aquí expuesta se puede encontrar en el repositorio Documentación de la organización de Github ArqSoft21-UNIZAR .

Requisitos del sistema

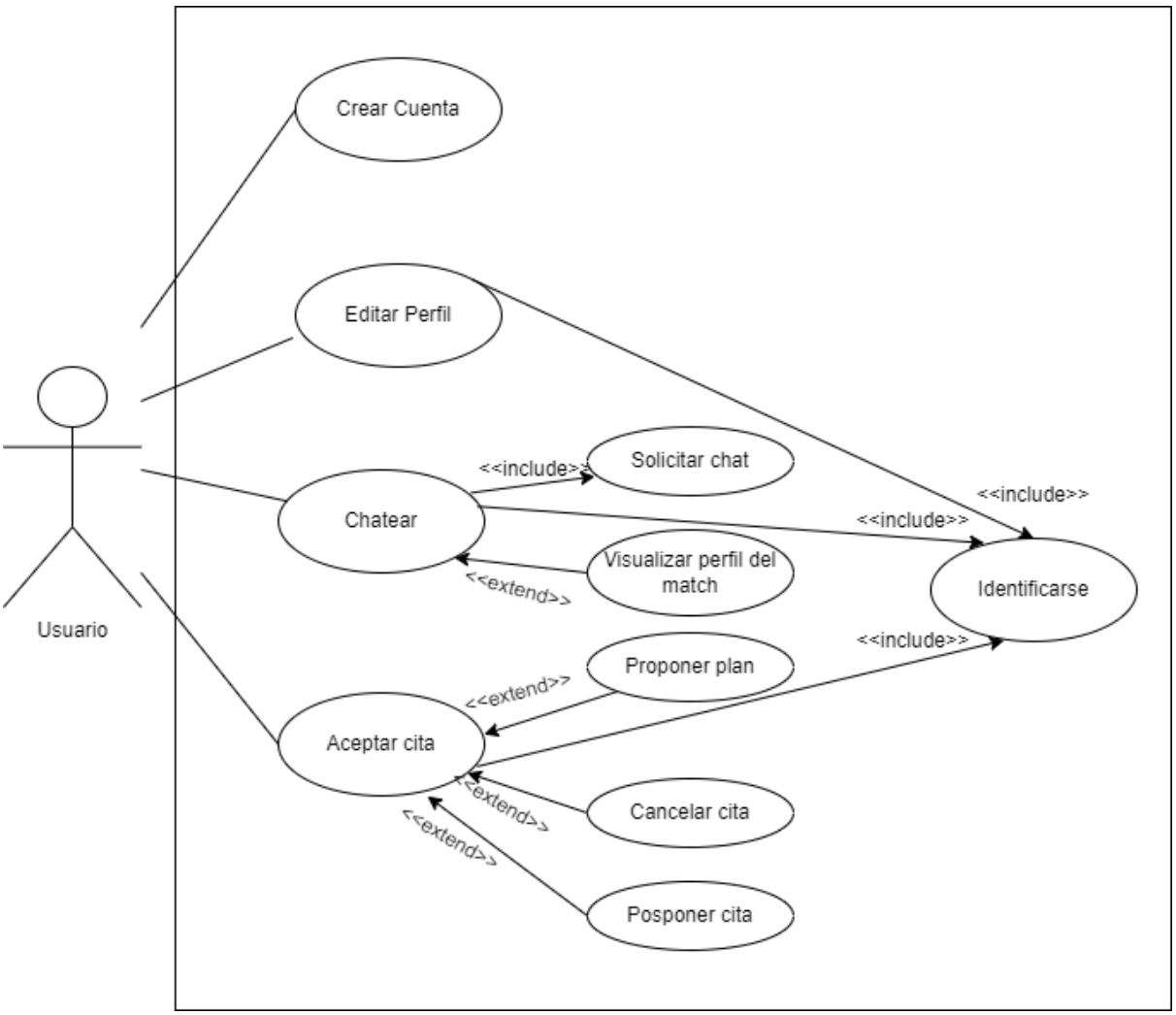
RF	Descripción
RF-1	El sistema permitirá al usuario crear una cuenta.
RF-2	El sistema permitirá al usuario gestionar su propio perfil.
RF-3	El sistema permitirá al usuario solicitar un chat cada día, pudiendo tener varios chats simultáneamente.
RF-4	El sistema permitirá al usuario especificar las etiquetas con las que se identifica.
RF-5	El sistema creará chats emparejando usuarios atendiendo a las etiquetas y la información que se dispone de estos.
RF-6	El sistema permitirá al usuario enviar mensajes a otro usuario en un chat.
RF-7	El sistema permitirá al usuario leer los mensajes de un chat.
RF-8	El sistema permitirá a los usuarios ver el perfil de las personas con las que tiene un match.
RF-9	El sistema mostrará al usuario cuántos mensajes le quedan por escribir en el chat.
RF-10	El sistema mostrará al usuario cuántos caracteres le quedan por escribir en un mensaje.
RF-11	El sistema permitirá a los usuarios conceder una cita.
RF-12	El sistema propondrá un plan a los usuarios para su cita, pudiendo estos aceptarlo o proponer una alternativa propia (hasta concertar una cita).
RF-13	El sistema permitirá al usuario cancelar o posponer una cita.

RNF	Descripción
RNF-1	El idioma de la aplicación es el español.
RNF-2	La aplicación funcionará en Google Chrome.
RNF-3	El sistema necesita conexión a internet de mínimo 100 kB/s de bajada para el correcto funcionamiento del sistema.
RNF-4	Se respetará la Ley Orgánica de Protección de Datos de Carácter Personal.
RNF-5	El sistema debe tener un tiempo de respuesta inferior a 500 ms desde que se envía un mensaje hasta que aparece en la pantalla.
RNF-6	El sistema debe soportar a 20 usuarios concurrentes.
RNF-7	El sistema limita los mensajes enviados por cada usuario en cada chat a 50 mensajes de hasta 280 caracteres cada uno.

Diccionario

- **Cuenta:** correo, contraseña y perfil de un usuario.
- **Perfil:** información pública asociada a un usuario (cualquier usuario que comparta un chat con este usuario puede acceder) que incluye el nombre, apellido(s), la edad, el género, la foto y localidad (ciudad/pueblo).
- **Chat:** Un chat es un servicio de mensajería instantánea que permite comunicarse a dos personas de forma inmediata y mantener una conversación por escrito en tiempo real.
- **Chatear:** comunicación basada en mensajes que se realiza entre dos usuarios de la aplicación, este se establece cuando dos usuarios “compatibles” lo solicitan en el mismo día. Simplificando, chatear consiste en el envío y la lectura de mensajes entre dos usuarios que estén en un mismo chat.
- **Mensaje:** cadena de 280 caracteres que tiene un receptor y un emisor.
- **Cita:** acuerdo entre dos usuarios que especifica cómo se encontrarán para conocerse. Esta consta de plan, lugar, fecha y hora.
- **Plan:** idea principal de lo que se va a hacer en una cita. Ir al cine, de picnic…
- **Etiqueta:** concepto que define gustos, hobbies, características o preferencias de un usuario. Un usuario puede tener varias etiquetas asociadas.
- **Match:** acción realizada por el sistema de emparejar dos usuarios en un mismo chat.

Diagrama de casos de uso



Descripción de casos de uso

Caso de uso: Crear cuenta

Pre: el usuario está interesado en crear una cuenta en la aplicación.
Post: el usuario ha creado una cuenta en la aplicación.

- Flujo de eventos principal
    1. El caso de uso comienza cuando el usuario accede a la aplicación web.
    2. El usuario ingresa sus datos personales identificatorios tales como su correo electrónico y su contraseña. Luego añade los datos públicos del perfil como nombre, género, foto de perfil y etiquetas.
    3. El caso de uso termina cuando el usuario pulsa “aceptar”.

- Flujo de eventos alternativo:
  El usuario ingresa sus datos personales identificatorios tales como su correo electrónico y su contraseña. Luego añade los datos públicos del perfil como nombre, género, foto de perfil y etiquetas.. Decide no añadir género porque el usuario no se siente cómodo etiquetandose como uno de los dos.

- Flujo de eventos alternativo:
  El correo electrónico introducido tiene un formato no válido y se notifica al usuario del error tras haber pulsado el botón “aceptar”.

Caso de uso: Editar perfil

Pre: El usuario tiene ya una cuenta creada y se ha identificado en la aplicación.
Post: El usuario ha modificado su perfil.

- Flujo de eventos principal
  1. El usuario accede a su perfil.
  2. A continuación, decide revisar sus etiquetas y últimamente le está gustando mucho salir a correr, así que decide añadir una etiqueta con ese concepto.
  3. El usuario está satisfecho con las modificaciones que ha realizado a su perfil, pulsa *aceptar* y el caso de uso acaba.

Caso de uso: Identificarse

Pre: el usuario tiene cuenta creada.
Post: el usuario ha accedido a su “home” de la aplicación.

- Flujo de eventos principal:
  1. El usuario ingresa su correo electrónico y contraseña.
  2. El usuario pulsa en “aceptar”.
  3. Como el usuario ha ingresado sus datos correctamente, se acaba el caso de uso.

- Flujo de eventos alternativo:
  Si el correo electrónico que introduce el usuario no tiene cuenta en la aplicación o la contraseña no es correcta, se notifica al usuario que hay un error en los datos y se reinicia el caso de uso.

Caso de uso: Solicitar chat

Pre: El usuario abre la aplicación y está identificado.
Post: El usuario ha solicitado un chat.

- Flujo de eventos principal:.
  1. El usuario se encuentra en su “home”, como aún no tiene ninguna conversación activa, solicita un nuevo chat.
  2. El sistema le empareja con otro usuario según sus etiquetas y de la información que dispone, aparecerá un nuevo chat en su ventana de chats, terminando así el caso de uso.

- Flujo de eventos alternativo:

Si el usuario ya ha solicitado el chat diario, e intenta solicitar otro, el sistema no le emparejará con nadie y aquí se acaba el caso de uso.

Caso de uso: Chatear

Pre: el usuario ha abierto la aplicación, se ha identificado y ha solicitado un chat.
Post: el usuario

- Flujo de eventos principal
    1. El usuario decide enviarle un mensaje a la persona con la que le han emparejado..
    2. La persona con la que está hablando le responde al mensaje. El usuario recibe lo que ha dicho por escrito, lo lee y responde.
    3. Mientras mantienen una conversación, el usuario visualiza el perfil de la persona con la que está hablando.
    4. Extend *'Visualizar perfil del match'*.
    5. A continuación el usuario mira los caracteres restantes y nota que le queda solo para envíar un mensaje más.
    6. Una vez enviado el mensaje y acabados los caracteres de ambos usuarios, se acaba el caso de uso.

Caso de uso: Aceptar cita

Pre: el usuario se ha identificado en la aplicación y ya ha chateado con su match. A ambos se les han acabado los caracteres.
Post: el usuario ha aceptado la cita tras haber hablado con su match.

- Flujo de eventos principal:
    1. El caso de uso comienza cuando ambos usuarios han agotado los caracteres disponibles.
    2. Al usuario le ha gustado la persona con la que le han emparejado y decide que quiere tener una cita mediante la opción *'quiero tener una cita'*. El otro usuario también piensa lo mismo así que selecciona la misma opción.
    3. El paso siguiente es escoger un plan, ambos usuarios escogen de las dos opciones existentes, la de elegir plan predeterminado.
    4. Luego de ambos haber estado de acuerdo con el mismo plan, se concreta la cita en una fecha, lugar y hora y se acaba el caso de uso.

- Flujo de eventos alternativo:
  Si el usuario decide no tener una cita con la persona con la que está chateando, se acaba el caso de uso.

- Flujo de eventos alternativo:
  Si ambos usuarios escogen la opción de proponer plan, tendrán que intercambiar mensajes para escoger un plan y se continúa el caso de uso.

- Flujo de eventos alternativo:
  Después de haber concretado la cita, el usuario cambia de opinión y decide cancelar la cita (Extend *'Cancelar cita'*) y se reinicia el caso de uso.

- Flujo de eventos alternativo:
  Después de haber concretado la cita, al usuario le surge un percance y decide posponer la cita (Extend *'Posponer cita'*) y se acaba el caso de uso.

Caso de uso: Proponer plan

Pre: el usuario tiene cuenta, se ha identificado y ha aceptado tener una cita con su match la cual también ha aceptado.
Post: el usuario ha acabado con una cita y un plan concretado con su match.

- Flujo de eventos principal

  1. El usuario se encuentra en el chat del match al que acaba de aceptar la cita.
  2. El usuario selecciona la opción de "Proponer plan".
  3. Aparece una nueva ventana en la que se le pregunta al usuario cuál sería un buen plan que tener con su match.
  4. El usuario escribe en el cuadro de texto con caracteres limitados que a él o ella le gustaría ir al parque "Tío Jorge" para tener un picnic.
  5. Le da al botón de "enviar" y espera la respuesta de su match.
  6. Recibe la aprobación del plan de su emparejamiento y aquí acaba el caso de uso.

- Flujo de eventos alternativo:
  Si el emparejamiento decide rechazar el plan que ha propuesto, será su match al que le toque proponer otro plan. En vez de que el usuario reciba la aprobación, recibirá el plan propuesto por su pareja. Este decidirá si aceptar o rechazar. Si rechaza se le volverá a pedir otro plan hasta que ambos se pongan de acuerdo.

Caso de uso: Posponer cita

Pre: el usuario tiene cuenta, se ha identificado y se han puesto de acuerdo en tener una cita y ya han propuesto el plan.
Post: la cita del usuario ha sido pospuesta.

- Flujo de eventos principal:
  1. El usuario estando en el "home", accede al chat del match con el que va a tener una cita (ya planeada).
  2. El usuario está demasiado ocupado con la universidad y prefiere tener la cita en otro momento, por lo que pulsa el botón de *posponer cita*.
  3. A continuación se abre el chat y el usuario propone una nueva fecha y hora para tener la cita, y envía la sugerencia.
  4. El otro usuario no acepta la nueva sugerencia, así que propone otra fecha y hora y la envía.
  5. El usuario recibe la propuesta y la acepta, quedando así fijada una nueva cita y acabándose el caso de uso.

- Flujo de eventos alternativo:

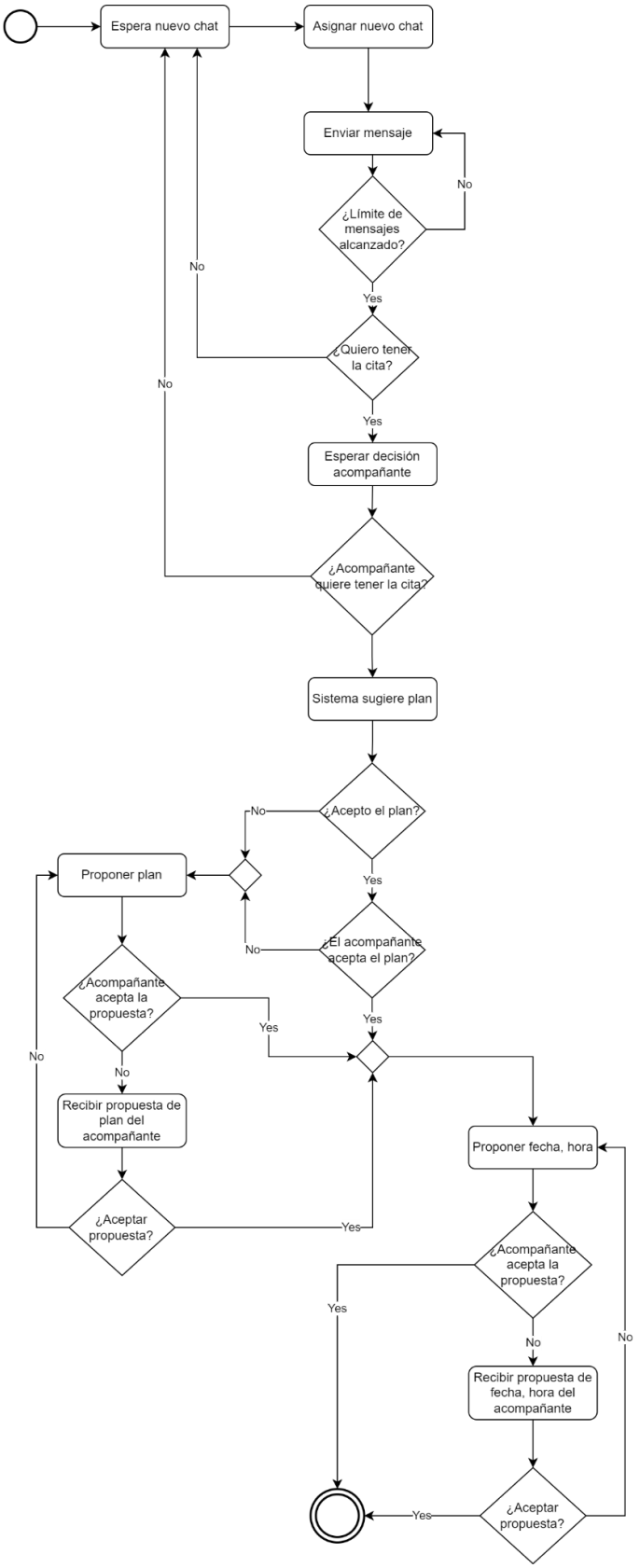
Si la pareja del usuario decide que esa fecha y esa hora no está bien, se volverá a hacer una nueva propuesta y así hasta que se pongan de acuerdo.

Caso de uso: Cancelar plan

Pre: el usuario tiene cuenta, se ha identificado y se han puesto de acuerdo en tener una cita y ya han propuesto el plan.
Post: la cita ha sido cancelada.

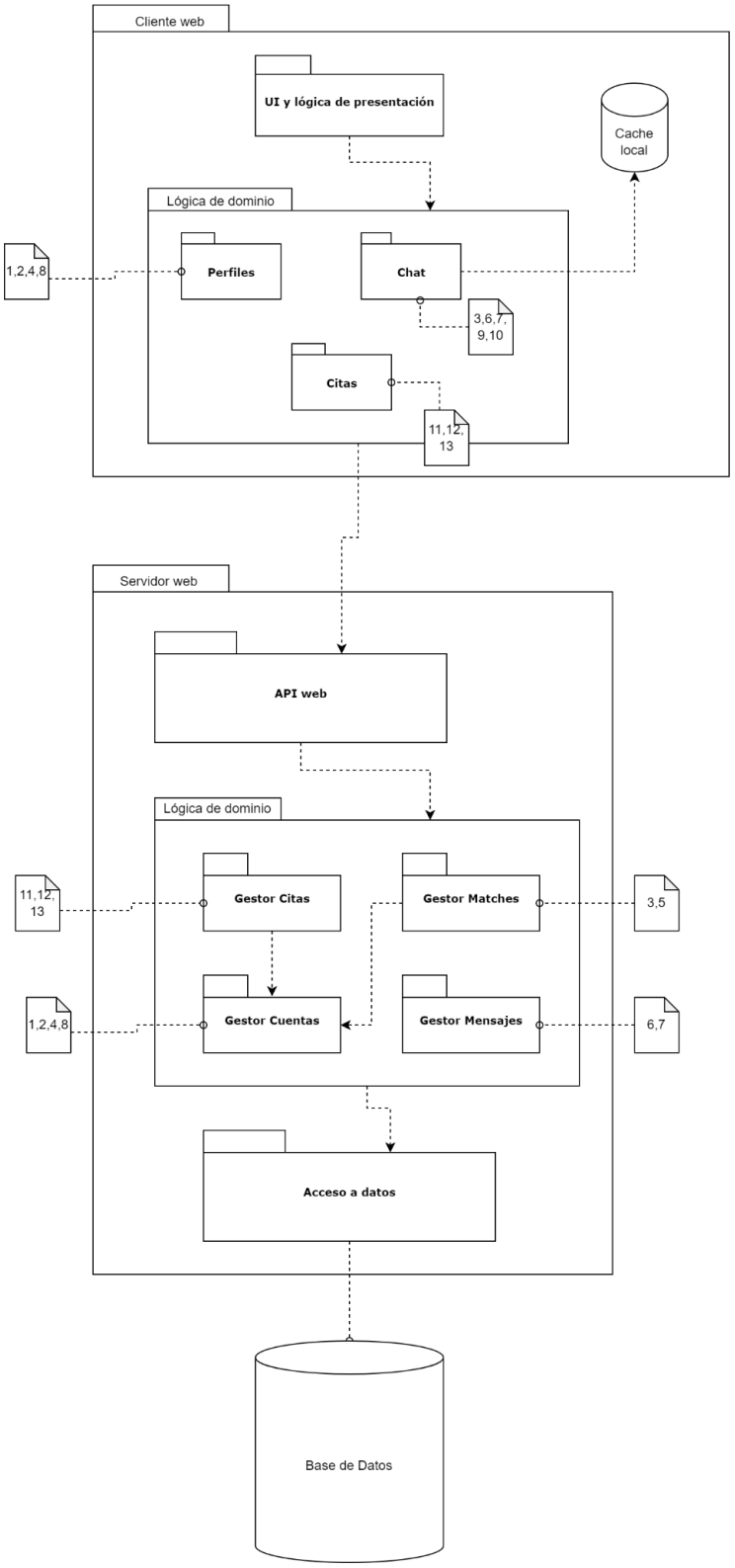
- Flujo de eventos principal:
    1. El usuario estando en el “home”, accede al chat del match con el que va a tener una cita (ya planeada).
    2. Al usuario le ha surgido un nuevo trabajo en otra ciudad y su incorporación es inminente. Por ello, decide cancelar su cita y pulsar “cancelar cita”.
    3. El caso de uso acaba cuando el sistema le pregunta que para asegurarse que si de verdad quiere cancelar la cita y el sistema se lo notifica a su match.

Diagrama de actividad de concertar una cita

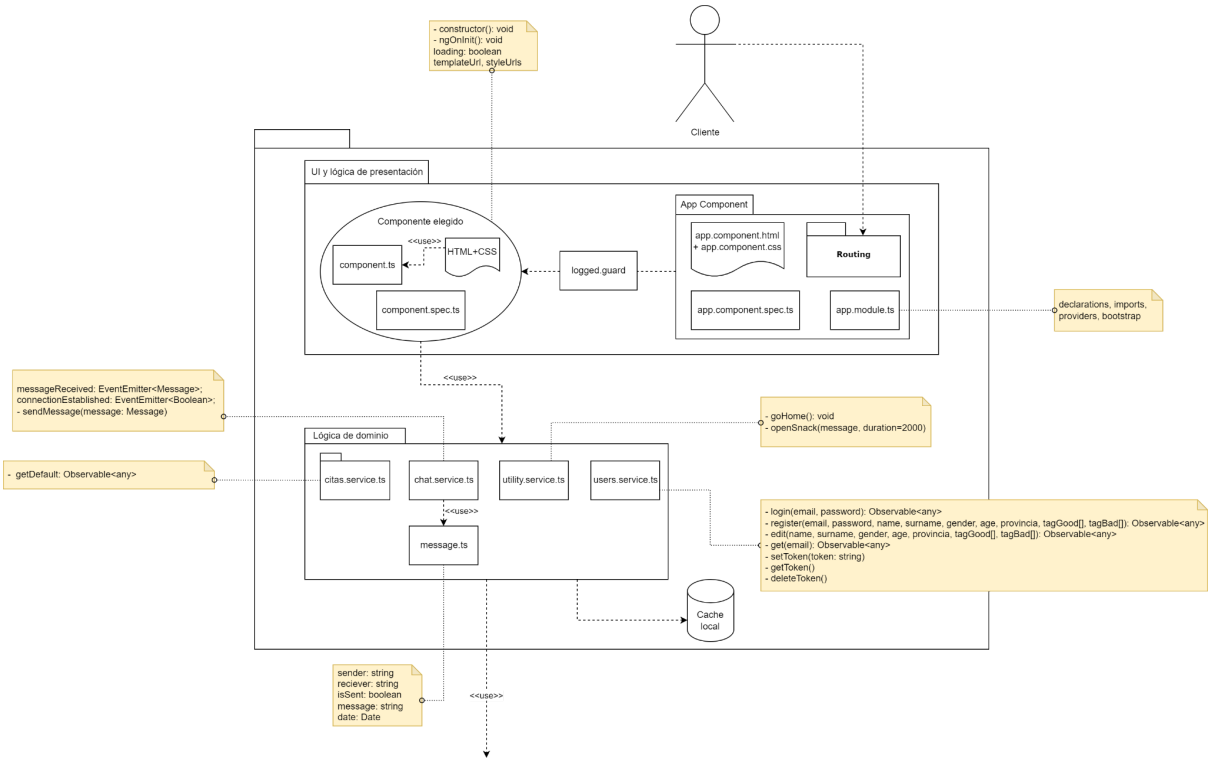


Vista de módulos

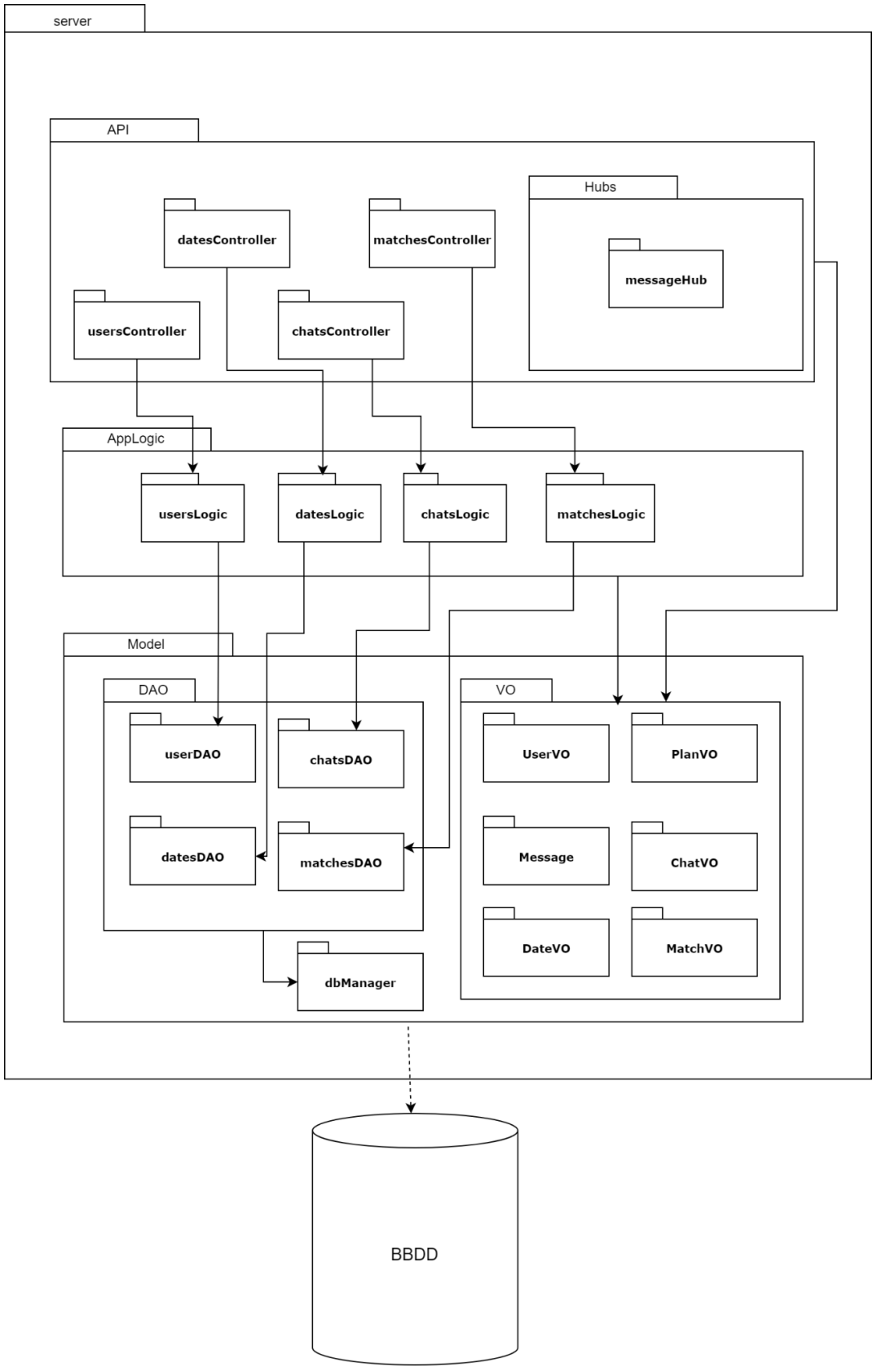
General



Frontend


- constructor(): void
- ngOnInit(): void
loading: boolean
templateUrl, styleUrls
Cliente
UI y lógica de presentación
Componente elegido
<<use>>
component.ts
HTML+CSS
component.spec.ts
logged.guard
App Component
app.component.html
+ app.component.css
Routing
app.component.spec.ts
app.module.ts
declarations, imports,
providers, bootstrap
<<use>>
messageReceived: EventEmitter<Message>;
connectionEstablished: EventEmitter<Boolean>;
- sendMessage(message: Message)
- goHome(): void
- openSnack(message, duration=2000)
Lógica de dominio
- getDefault: Observable<any>
citas.service.ts
chat.service.ts
utility.service.ts
users.service.ts
<<use>>
message.ts
- login(email, password): Observable<any>
- register(email, password, name, surname, gender, age, provincia, tagGood[], tagBad[]): Observable<any>
- edit(name, surname, gender, age, provincia, tagGood[], tagBad[]): Observable<any>
- get(email): Observable<any>
- setToken(token: string)
- getToken()
- deleteToken()
Cache
local
sender: string
reciever: string
isSent: boolean
message: string
date: Date
<<use>>

Backend


server
API
Hubs
datesController
matchesController
messageHub
usersController
chatsController
AppLogic
usersLogic
datesLogic
chatsLogic
matchesLogic
Model
DAO
VO
userDAO
chatsDAO
UserVO
PlanVO
datesDAO
matchesDAO
Message
ChatVO
DateVO
MatchVO
dbManager
BBDD

Interfaz vista de módulos Backend

Módulo API:
El módulo API es utilizado por el Frontend.

Interfaz de módulo **chatsController**:
- ChatVO **getChat**(ChatVO chat):
  Devuelve un chat.

Interfaz de módulo **datesController**: (No implementado)

Interfaz de módulo **matchesController**:
- IActionResult **requestMatch**(UserVO user)
  Solicita un match. Devuelve OK si ha ido bien, BadRequest si ha ido mal.
- IEnumerable<UserVO> **getMatches**(UserVO user)
  Devuelve todos los usuarios con los que "user" tiene un match.
- IActionResult **deleteMatch**(MatchVO match)
  Elimina un match. Devuelve OK si ha ido bien, BadRequest si ha ido mal.

Interfaz de módulo **usersController**:
- IActionResult **login**(UserVO user)
  Autentica al usuario "user". Devuelve OK si ha ido bien, BadRequest si ha ido mal.
- IActionResult **register**(UserVO user)
  Registra al usuario "user". Devuelve OK si ha ido bien, BadRequest si ha ido mal.
- IActionResult **delete**(UserVO user)
  Elimina al usuario "user". Devuelve OK si ha ido bien, NotFound si no existe el usuario en la base de datos.
- IActionResult **edit**([FromBody] UserVO user)
  Edita el usuario "user". Devuelve OK si ha ido bien, BadRequest si ha ido mal.
- IActionResult **get**([FromBody] UserVO user)
  Devuelve la información del usuario "user" si ha ido bien, NotFound si no existe el usuario en la base de datos.

Módulo AppLogic:
Interfaz módulo **chatsLogic**:
- ChatVO **getChat**(String emisor, String receptor)
  Devuelve el chat entre "emisor" y "receptor"
- bool **saveMessage**(Message m)
  Devuelve true si se ha podido guardar el mensaje, false en caso contrario.

Interfaz módulo **datesLogic**: (Los demás métodos de datesLogic no están implementados)
- PlanVO **getDefaultPlan**()
  Devuelve un plan por defecto.

Interfaz módulo **matchesLogic**:
- List<UserVO> **getMatches**(String email)
  Devuelve la lista de usuarios con los que está matcheado el usuario de email "email".
- bool **deleteMatch**(String email1, String email2)

Devuelve true si se ha podido eliminar el match entre los usuarios con emails "email1" y "email2", false en caso contrario.

- bool **requestMatch**(String email)
  Devuelve true si se ha podido solicitar el match por el usuario con email "email", false en caso contrario.

Interfaz módulo **usersLogic**:

- int **login**(String email, String password)
  Devuelve 0 si se autenticado al usuario con éxito, 1 si la contraseña no es válida y 2 si no existe un usuario con email "email" en la base de datos.
- int **register**(UserVO user)
  Devuelve 0 si se ha registrado al usuario "user" con éxito, 1 si ya existe el usuario y 2 si no se pudo crear el usuario.
- bool **delete**(UserVO user)
  Devuelve true si se el usuario "user" ha sido eliminado con éxito, false en caso contrario.
- bool **edit**(String email, String nombre, String apellidos, String sexo, DateTime fNacimiento, String localidad, String meGusta1, String meGusta2, String meGusta3, String noMeGusta1, String noMeGusta2, String noMeGusta3, String orientación, int capacidad)
  Devuelve true si el usuario con email "email" ha sido editado satisfactoriamente con los campos pasados por parámetro, false en caso contrario.
- UserVO **get**(String email)
  Devuelve el usuario con email "email".

Módulo Model:

Módulo VO:

Interfaz de módulo **Message:**

- <<constructor>> **Message**(String sender, String reciever, String message, DateTime date):
  Crear un mensaje con un emisor "sender" y un receptor "reciever", con el contenido "message" y una fecha "date".

Interfaz de módulo **ChatVO**:

- <<constructor>> **ChatVO**(String emisor, String receptor):
  Crea un chat con un emisor y un receptor.
- bool **addMsg**(Message m):
  Añade el mensaje "m" a la lista de mensajes del chat.

Interfaz de módulo **PlanVO**:

- <<constructor>> **PlanVO**(String lugar, String descripcion, double lat, double lon):
  Crea un plan en un lugar con una descripción y unas coordenadas.

Interfaz de módulo **DateVO**:

- <<constructor>> **DateVO**(String email1, String email2, DateTime momento, PlanVO plan):

Crea una cita entre dos usuarios identificados con sus emails con un plan y un momento de tiempo

Interfaz de módulo **MatchVO**:

- <<constructor>> **MatchVO**(String email1, String email2):
  Crea un match entre el usuario con email "email1" y el usuario con email "email2".

Interfaz de módulo **UserVO**:

- <<constructor>> **UserVO(**string email, string fNacimiento, string password, string nombre, string apellidos, string sexo, string localidad, string meGusta1, string meGusta2, string meGusta3, string noMeGusta1, string noMeGusta2, string noMeGusta3, string orientacion, int capacidad)
  Crea un objeto usuario con la información relevante al mismo.

Módulo DAO:

Interfaz de módulo **chatsDAO:**

- bool **saveMessage**(String sender, String reciever, String message, DateTime date):
  Guarda un mensaje en la base de datos. Devuelve true si se guarda correctamente.
- List<Message> **GetMessages**(String sender, String reciever):
  Devuelve la lista de mensajes entre el emisor "sender" y el receptor "reciever".

Interfaz de módulo **datesDAO:**

- PlanVO **getDefaultPlan**():
  Devuelve un plan aleatorio del conjunto de planes por defecto de la base de datos.

Interfaz de módulo **matchesDAO:**

- bool **addMatch**(String email1, String email2):
  Añade un match entre los usuarios con emails "email1" y "emai2". Devuelve true si se ha añadido correctamente, false si los emails son iguales.
- bool **deleteMatch**(String email1, String email2):
  Elimina el match entre los usuarios con emails "email1" y "emai2". Devuelve true si se ha eliminado correctamente, false en caso contrario.
- bool **findMatch**(String email1, String email2):
  Devuelve true si existe un match entre los usuarios con emails "email1" y "email2", false en caso contrario.
- List<string> **getMatches**(String email1, String email2):
  Devuelve una lista de usuarios con sus datos.

Interfaz de módulo **usersDAO:**

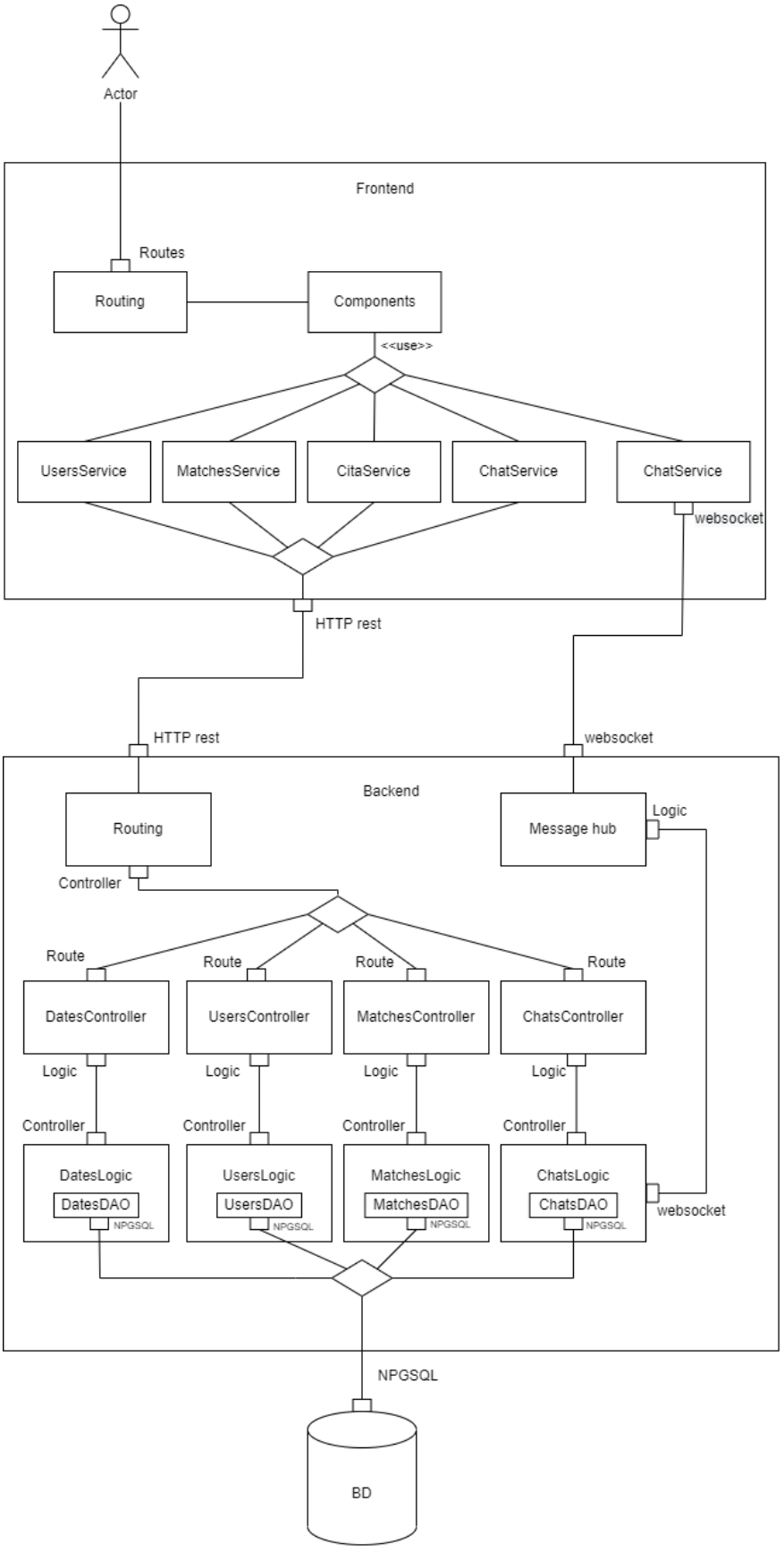
- UserVO **getUser**(String email):
  Devuelve el usuario con email "email".
- bool **createUser**(UserVO user):

Introduce el usuario "user" en la base de datos. Devuelve true si ha sido creado con éxito.

- bool **deleteUser**(String email, String password):
  Elimina el usuario con email "email" y contraseña "password" de la base de datos. Devuelve true si ha sido eliminado con éxito, false en caso contrario.

- bool **editUser**(UserVO):
  Devuelve true si el usuario "user" ha sido editado satisfactoriamente con los campos del objeto UserVO, false en caso contrario.

Vista de componente y conector



Documentación puertos

NPGSQL:

- **ExecuteNonQueryAsync:** executes SQL which doesn't return any results, typically INSERT, UPDATE or DELETE statements. Returns the number of rows affected.
- **ExecuteScalarAsync:** executes SQL which returns a single, scalar value.
- **ExecuteReaderAsync:** execute SQL which returns a full resultset. Returns an NpgsqlDataReader which can be used to access the resultset (as in the above example).

Websocket

- **startConnection:** Empezar una conexión
- **createConnection:** Crear un socket
- **send:** Enviar un mensaje por el socket

HTTP rest

- **get:** Realiza petición a una URL sin "cuerpo" del mensaje.
- **post:** Realiza petición a una URL con "cuerpo" del mensaje.

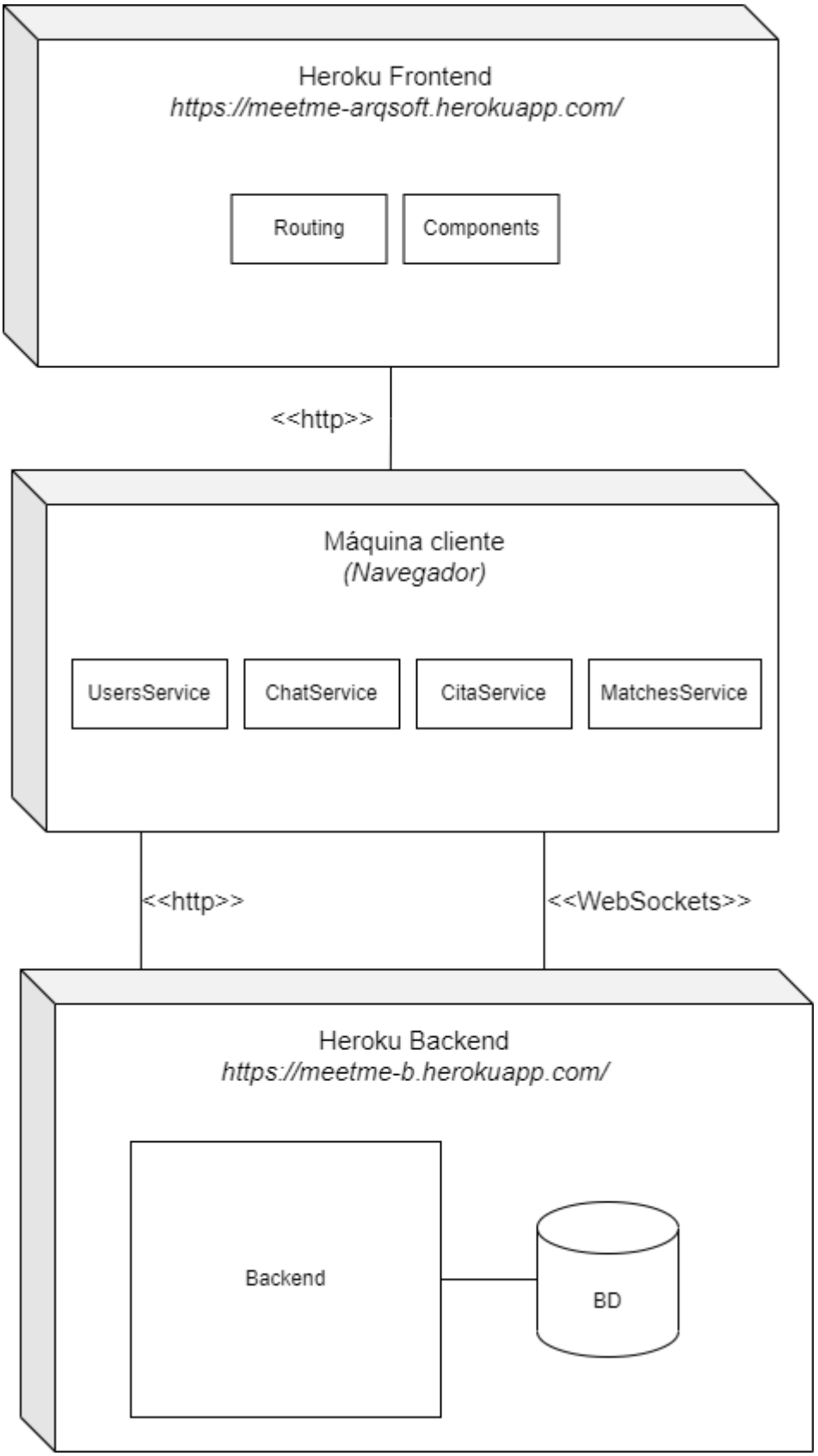
Routes

Lista de [Path, component, guards] desde la que se hace **find**

Logic, Controller

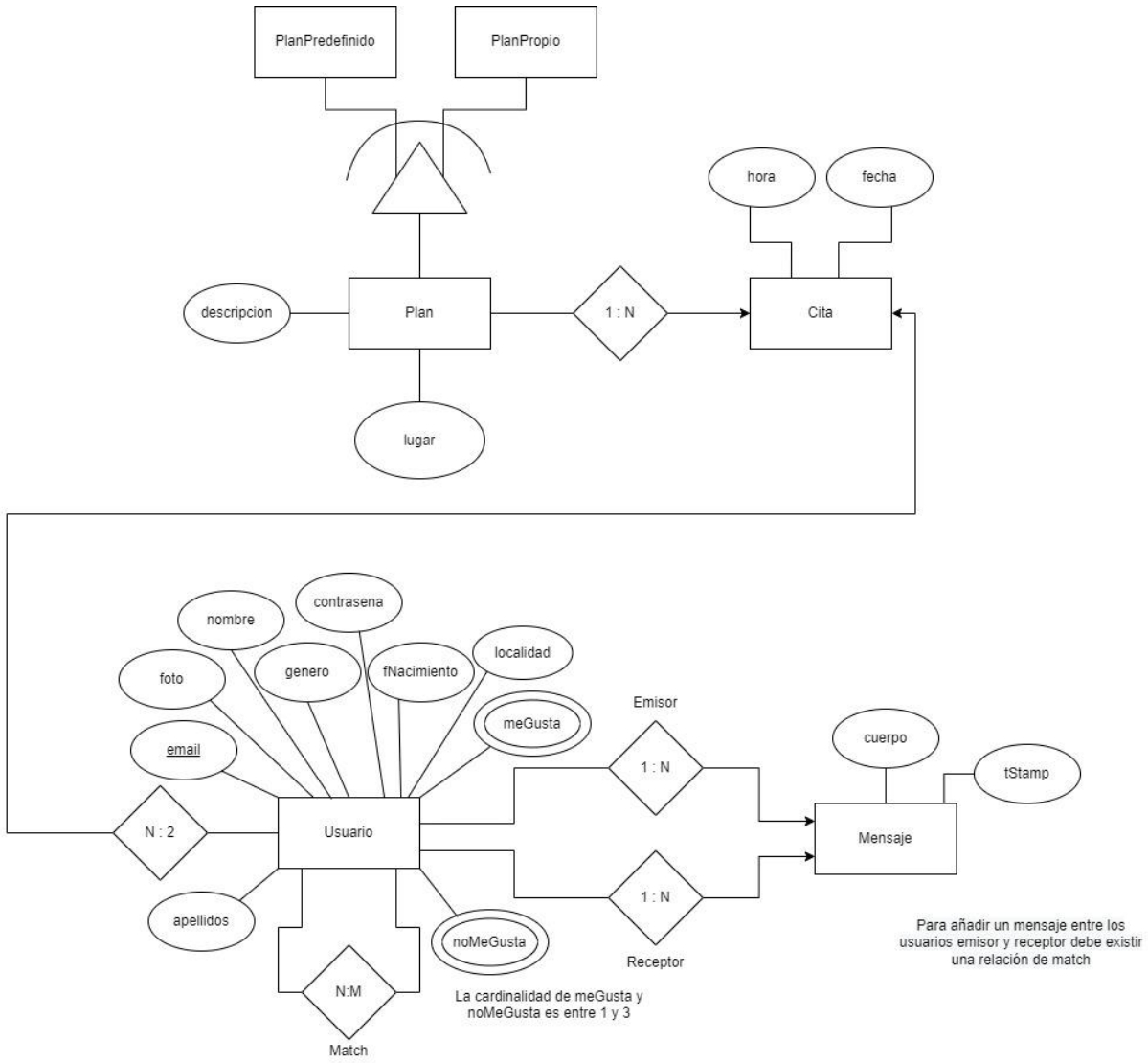
Llamada a método C#

Vista de despliegue

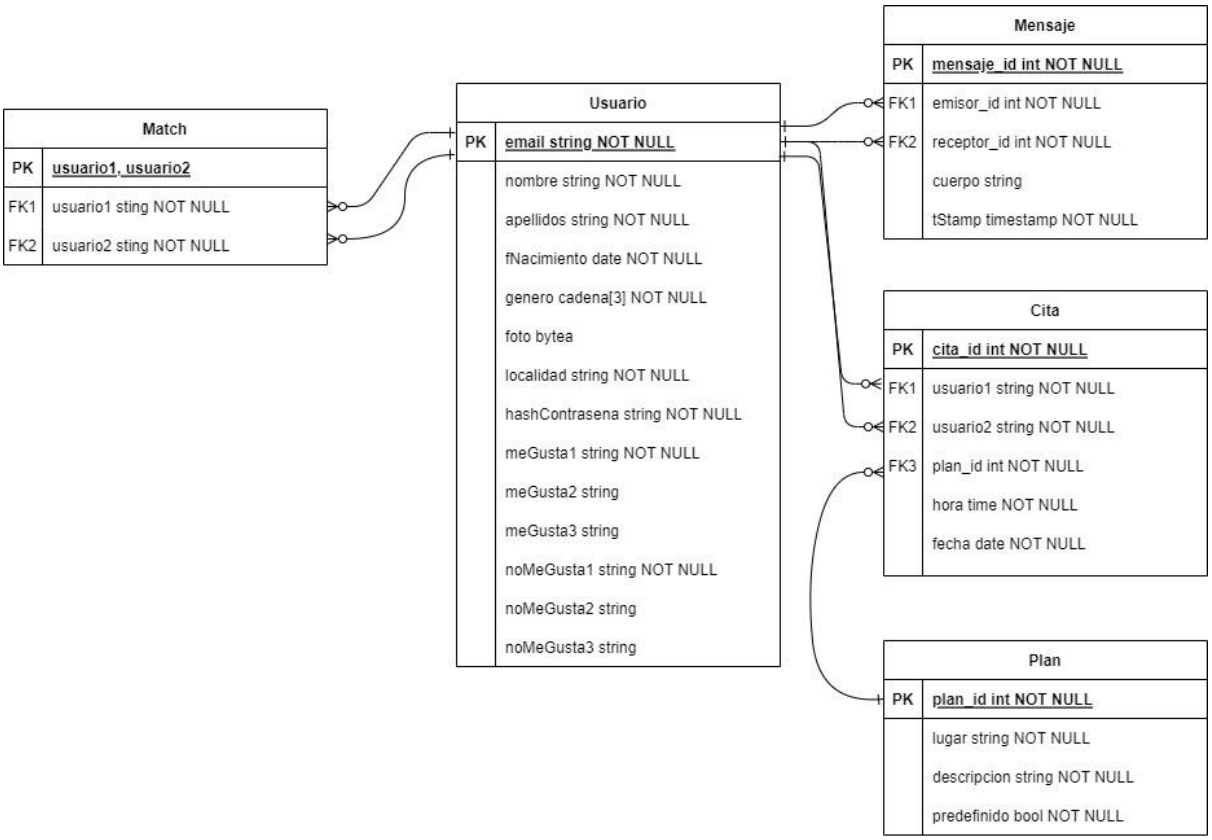


Base de datos

Modelo entidad/relación



Modelo relacional



Mapa de navegación

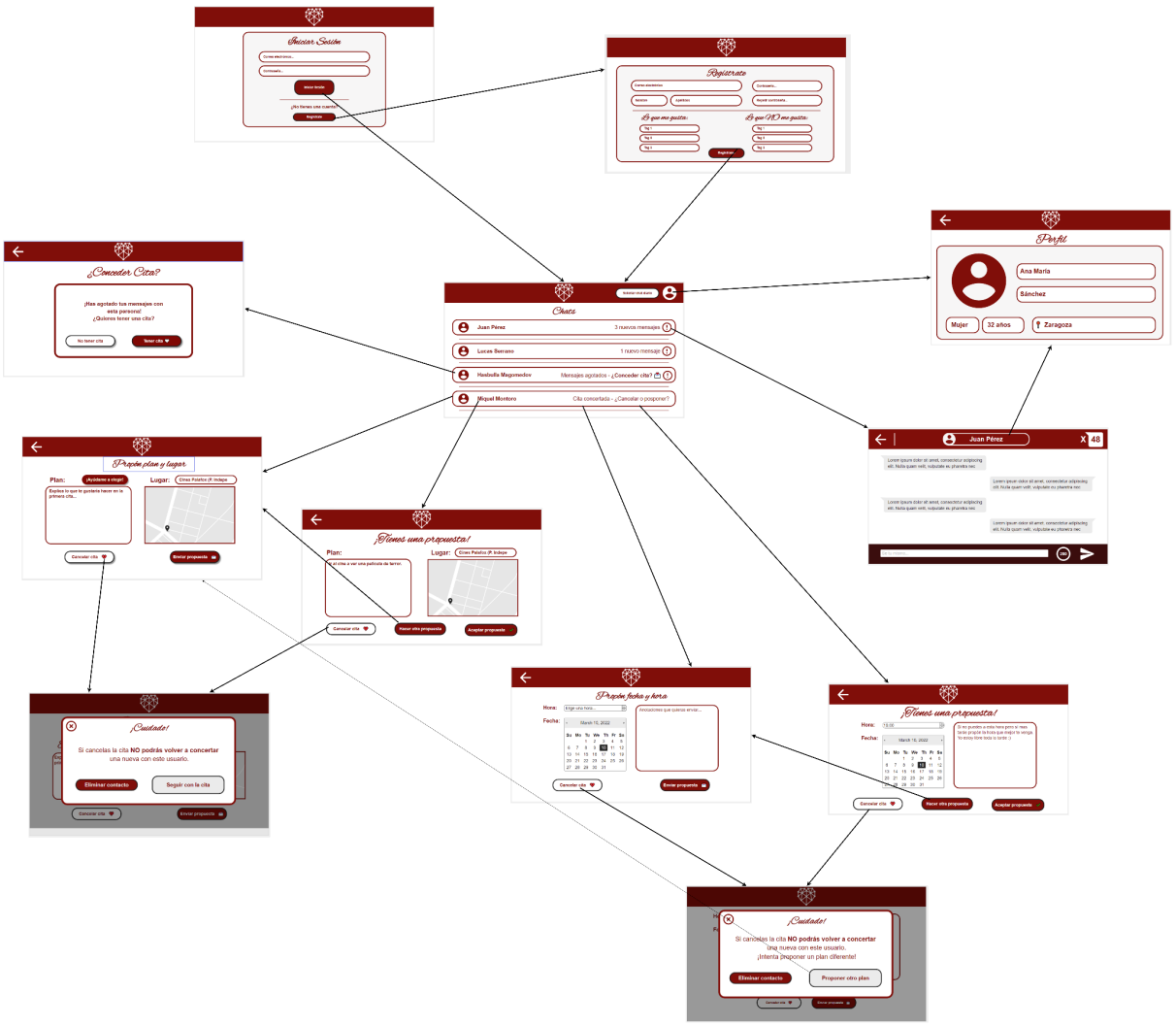


Diagrama de Asignación de trabajo

Vista de Asignación de trabajo		
Segmento	**Subsistema**	**Organización**
Frontend	UI y lógica de presentación -App Component Lógica de dominio Despliegue Heroku	Marcos Garralaga César Borja
Backend	API -Hubs Lógica de aplicación Model -Dao, -VO	Jesús Roche Juliana Zordan Clara Cerdán