

MÓDULO DE PROYECTO



**ALONSO
DE AVELLANEDA**
Alonso de Avellaneda

Técnico Superior en Desarrollo de Aplicaciones Web

Zirty, aplicación de chat en tiempo real

Autor: Alberto Collantes Sánchez

Profesor tutor: Pablo Martín Ruiz

Alcalá de Henares, Junio de 2025

Índice

Introducción	3
Estudio económico	4
Estudio legal	8
Estudio viabilidad cronológica	10
Análisis y diseño	11
Conclusiones	15
Bibliografía	16
Anexos	17

Introducción

La aplicación a desarrollar consiste en un chat en tiempo real utilizando la arquitectura Blazor Web Assembly hosted con .Net en la parte del backend, más concretamente la versión 9.0. Aún siendo su principal funcionalidad la de chat en tiempo real cuenta con persistencia de datos y personalización de perfil de usuario entre otras cosas. La aplicación pretende mantener los datos actualizados pero sin consultar constantemente a una base de datos, más bien mediante eventos y alguna consulta puntual.

La idea de usar esta tecnología fue motivada por una charla en el módulo de desarrollo web en entorno cliente. Nuestro profesor y tutor Pablo nos comentó que la tecnología de .Net para el backend y el framework de Blazor (lo que viene siendo el entorno Microsoft) era muy demandado por las empresas y estaba en auge, aún así muchos desarrolladores sobretodo españoles ignoraban esta tecnología y lo que más había era programadores con origen indio y/o sudamericano y en España era un terreno un poco inexplorado. Esta charla me dio ganas de aprender sobre el framework de Blazor y cuando descubrí la librería de Socket.io en el módulo de Desarrollo web en entorno cliente y de nuevo nuestro profesor nos comentó uno de los usos que se le podía dar a esta librería que permitía una comunicación muy rápida entre dos o más clientes o entre un nodo emisor y varios receptores... Me pregunté si había un equivalente a Socket.io para Blazor, y en efecto lo había, se llama SignalR y así ideé la aplicación.

En lo referente al nombre: al ser un miembro de la tan nombrada generación Z cuando entré en el instituto a los 12 años descubrí el internet y las redes sociales y en aquel momento la más grande era Tuenti, la actual operadora de móvil. El nombre de mi aplicación es una referencia a aquella red social que en mi opinión no tenía una esencia tan tóxica como las que hay ahora.

Estudio de la viabilidad económica del proyecto

Cálculo de costes

Los materiales para montar una aplicación de este tipo dependen siempre de lo mismo: la magnitud que se desea alcanzar y de ese motivo depende el segundo que es el presupuesto.

Puesto que la aplicación que he montado de momento no pretende más que ser un tfg es gratis pero como se verá en la presentación no funciona todo lo rápido que debería porque los recursos que se han utilizado han sido gratis. A continuación os pondré en contexto.

Pongámonos en el supuesto de que esta misma aplicación pretende llegar a más personas. El equipo de desarrollo debería contar con algún miembro más con el coste que ello conlleva, pongamos que contratamos a un Junior por 1200 euros.

En cuanto al servidor de hosting hay un millón de formas distintas de desplegar una aplicación web, yo he elegido una plataforma gratis que se llama Render. En esta plataforma se brinda un plan gratuito y uno de pago. A continuación pondré la tabla de precios

Hobby	Professional	Organization	Enterprise
For personal projects and small-scale applications.	For teams building production applications.	For teams with higher traffic demands and compliance needs.	For mission critical applications with complex needs.
\$0 USD <small>per user/month plus compute costs*</small>	\$19 USD <small>per user/month plus compute costs*</small>	\$29 USD <small>per user/month plus compute costs*</small>	Custom pricing
Start deploying	Select plan	Select plan	Get in touch
Deploy full-stack apps in minutes	Everything in Hobby, plus: 500 GB of bandwidth included	Everything in Professional, plus: 1TB of bandwidth included	Everything in Organization, plus: Centralized team management
Fully-managed datastores	Collaborate with 10 team members	Unlimited team members	Guest users
Custom domains	Unlimited projects & environments	Audit logs	SAML SSO & SCIM
Global CDN & regional hosting	Horizontal autoscaling	SOC 2 Type II certificate	Guaranteed uptime
Get security out of the box	Test with preview environments	ISO 27001 certificate	Premium support
Email support	Isolated environments		Customer success
	Chat support		

*Compute costs: Pay only for provisioned resources, with transparent pricing based on CPU and service activation prorated to the second.

Como se puede ver disponemos de distintos planes, nosotros usaremos el de Organization en el que pagaremos 29 dólares al mes para los cálculos de viabilidad que haremos más adelante. También comenté que este servicio de hosting si bien no es muy profesional es una opción económica y por eso se ha elegido esta. Existen otros proveedores de hosting como por ejemplo Azure App Services que tienen planes mucho más potentes pero también mucho más caros (algunos ascienden a los 3000 euros al mes).

Necesitaremos un dominio (buscaremos por ejemplo zirty.com)

The screenshot shows a search interface for 'zirty.com'. At the top, there's a search bar with the query 'zirty.com'. Below it, a prominent result is shown for 'zirty.com' with a green checkmark, labeled as 'PREMIUM'. To the right of this result are the price '€1,410.18' and a link 'Renews at €14.97/yr'. A large 'Add to cart' button is also present. Below this main result, there's a navigation bar with tabs: 'Domains' (selected), 'Auctions', 'Handshake', 'Generator', and 'Beast Mode'. Underneath the tabs, a section titled 'Suggested Results' lists several other domain variations with their prices, discounts, and 'Add to cart' buttons:

Domain	Discount	Price	Renewal Price	Add to Cart
zirty.com	PREMIUM	€1,410.18	Renews at €14.97/yr	Add to cart
zirty.net	13% OFF	€11.45/yr	Retail €13.21/yr	Add to cart
zirty.org	42% OFF	€6.60/yr	Retail €11.45/yr	Add to cart
zirty.ai	11% OFF	€79.35	Renews at €82.00/yr	Add to cart
zirty.to	25% OFF	€26.44/yr	Retail €35.26/yr	Add to cart

https://www.godaddy.com/es-es/domainsearch/find?domainToCheck=zirty.com

zirty.com

DOMINIO PREMIUM

zirty.com

1.409,18 € +21,99 €/año

Valor Estimado 1.487 € ⓘ

Compralo ahora Alquila para comprar Llama al +34 91 198 05 24 para recibir asistencia

Por qué es genial

- ✓ Utiliza la extensión .com.
- ✓ "Zirty" es de 15 caracteres o menos.

Perspectivas de dominios

DOMINIO PREMIUM

Compra ahora a este precio o alquila para comprar. [Más información](#)

Los dominios incluyen Protección de privacidad gratis para siempre. ⓘ

en GoDaddy. ⏲ Cada 4 minutos alguien compra un .AI en GoDaddy. ⏲ Cada 2 minutos alguien compra un .CO en GoDaddy.

zirty.es Sujeto a restricciones. ⓘ

10,99 € 0,01 € Únicamente el primer año con un plazo de 2 años ⓘ

Si comparamos entre los distintos dominios encontramos una diferencia de precios muy grande entre los TLD (Top Level Domains)

Dado que nuestra aplicación pretende usarse únicamente en España elegiremos el dominio Zirty.es con un precio de 0,01 el primer año. Este dominio ha sido encontrado en GoDaddy.com

Otra cosa que deberíamos tener en cuenta es nuestro almacenamiento de datos puesto que nuestra aplicación cuenta con persistencia de los mismos. Nuestra aplicación persiste en local en una base de datos mongo pero para nuestro servidor de hosting no procede una base de datos en local puesto que son despliegues a través de repositorios de Github por tanto elegiremos Mongo Atlas que también tiene su pricing fijado el cual pondré a continuación.

Pay-as-you-go! Clusters are billed hourly with monthly invoices.



Amazon Web Services



Microsoft Azure



Google Cloud

Cluster Tier	Storage	RAM	vCPUs	Base Price
M10	10 GB	2 GB	2 vCPUs	\$0.08/hr
M20	20 GB	4 GB	2 vCPUs	\$0.20/hr
M30	40 GB	8 GB	2 vCPUs	\$0.54/hr
M40	80 GB	16 GB	4 vCPUs	\$1.04/hr
M50	160 GB	32 GB	8 vCPUs	\$2.00/hr
M60	320 GB	64 GB	16 vCPUs	\$3.95/hr
M80	750 GB	128 GB	32 vCPUs	\$7.30/hr

Como podemos ver esta vez el precio es por horas y depende de los recursos de los que podremos hacer uso y del proveedor. Para nuestro caso puesto que no tenemos cargas muy pesadas a la base de datos y las que son mas pesadas son una vez como puede ser el login, subir una foto a la base de datos en base64 o al buscar usuarios elegiremos el Tier M20 por asegurarnos que vaya bien. En la presentación se verá como va con el modo gratis de Mongo Atlas.

El resto de programas usados o licencias son gratuitos asi que estos son los gastos que deberíamos tener en cuenta.

Retorno de inversión

Si sumamos los costes fijos que tendríamos mensuales si eligiéramos los planes de precios de los que hemos hablado en el punto anterior nos encontramos que la cuantía asciende a 180 euros al mes aproximadamente. Eso obviando los gastos en personal de los cuales no hablaremos para hacer un poco más rentable el proyecto.

A continuación estudiaremos las distintas maneras que tendría nuestra aplicación de generar ingresos seguido de un pequeño estudio de mercado y de los principales competidores.

Las aplicaciones web normalmente generan la mayoría de sus ingresos con publicidad en ellas, por ejemplo en el sitio web de Wallapop nos encontramos que en cualquier búsqueda que hagamos el main es desplazado un poco hacia abajo para mostrar un banner de publicidad justo debajo de la barra de búsqueda o a la derecha del scroll que contiene los productos encontrados. Otros muchos sitios lo que hacen es al hacer un click dentro de la página se abre automáticamente una ventana o pestaña nueva que carga una página que normalmente no tiene nada que ver. Esta clase de publicidad genera ingresos que varían según el número de personas que visitan abren esos sitios. Tuenti por ejemplo en 2011 generó 10 millones de euros en publicidad pero también hay que tener en cuenta que en 2011 se estimó que aproximadamente el 15 % del tráfico de internet en España pasaba por Tuenti.

Otra forma de generar ingresos muy utilizada en sitios web son los planes de suscripciones los cuales ofrecen una serie de ventajas a cambio de pagar una cuantía ya sea mensual, anual o una sola vez. Una implementación de esto en una aplicación de un tipo parecido a la nuestra podría ser Discord que ofrece personalización extra del perfil mediante un pago pero eso no procede en una aplicación como la nuestra.

Si hacemos una previsión de ingresos nos encontraríamos que para generar 2-4 euros con publicidad programática con servicios como Google AdSense necesitaríamos alrededor de unos 1000 visitas. Dado que al principio sería nuestra única fuente de ingresos vamos a calcular cuantas visitas necesitaríamos alrededor de 45000 visitas para cubrir gastos. Para saber si nuestra aplicación obtendría esa cantidad de visitas necesitamos compararnos con nuestros principales competidores los cuales serían Discord, Whatsapp o Telegram y puesto que nuestra aplicación no dispone de ninguna función exclusiva que no tengan estas otras tres podríamos asegurar con casi total certeza que no triunfaría.

Estudio de la viabilidad legal del proyecto

Cumplimiento normativo

La Unión Europea tiene unas normas muy estrictas en cuanto a lo que el tratamiento de datos en internet se refiere:

El más importante es el Reglamento General de Protección de Datos el cual nos obliga a :

- Siempre a informar claramente al usuario qué datos se recogen y para qué fin.
- Solicitar el consentimiento expreso para registrar y tratar datos.
- Permitir el derecho de acceso, modificación y eliminación de datos.

- Proteger los datos mediante medidas de seguridad técnicas
- Firmas contratos si usamos servicios externos como encargados del tratamiento

No hace falta decir que toda esta normativa es la típica que nos ponen en el recuadro de acepto los términos y condiciones la cual si no aceptamos al hacer registro no podremos registrarnos valga la redundancia.

La Ley de Servicios de la Sociedad de la Información y del Comercio Electrónico aplica si el sitio web tiene servicios comerciales como compras... y/o comunicaciones electrónicas.

Nos obliga a mostrar el típico banner de aceptar/rechazar las cookies no esenciales que suelen ser de publicidad para mostrar publicidad personalizada y nos obliga también a incluir un aviso legal indicando quien es el responsable de la web (no hace falta que sea una persona física)

Licencias de software

Todo el software utilizado por nuestra aplicación es de código abierto. Como frameworks se utilizan .Net 9.0 y Blazor WASM los cuales tienen licencias MIT (licencia de código abierto popular que permite a los usuarios utilizar, copiar, modificar, fusionar, publicar, distribuir, sublicenciar y/o vender copias del software, siempre que se incluya el aviso de derechos de autor y la licencia original) En cuanto a las librerías se utilizan SignalR y MudBlazor que también son de código abierto.

Propiedad intelectual

Si bien es cierto que la idea surgió totalmente de mi solo con dos comentarios de mi profesor, no me puedo tomar la licencia de decir que el proyecto es totalmente original puesto que he utilizado ideas y recursos de terceros.

Las vistas utilizadas en mi proyecto (mis componentes .razor) salieron de modificar unas vistas de un programador que colgó un proyecto de Blazor usando la librería MudBlazor para mostrar las capacidades de esta librería. Este es el repositorio en cuestión: [Repositorio](#). El código que subes a Github si no incluyes una licencia explícita el proyecto está bajo "todos los derechos reservados", lo que significa que nadie puede legalmente copiar, modificar, distribuir o usar tu código, ni siquiera para

usos personales. Pero abajo del todo en el repositorio de este señor se puede ver que indica que su proyecto es de uso libre.

También sería injusto no nombrar las inteligencias artificiales las cuales están tan en boca de todos y mas en nuestro sector. En ellas he encontrado un montón de información que me ha ayudado a crear una aplicación en un entorno que para mi era completamente nuevo.

No tengo previsto aplicar ninguna licencia pero si tuviera que aplicar una debería ser de uso libre MIT puesto que yo uso vistas (aunque las haya modificado a mi gusto) de un tercero.

Estudio de la viabilidad cronológica del proyecto

El proyecto empezó de manera muy caótica y eso se podrá ver en el código (espero que cuando alguno de ustedes lo revise lo pueda tener mas ordenado). Sin ir más lejos la idea de usar MudBlazor nació buscando aplicaciones de chat en Blazor para inspirarme y encontré esta librería porque tenía componentes específicos para hacer chats como bubbles para los chats donde se mostraban los avatares al lado de los mensajes...

Empecé la aplicación con una vista sin mirar nada porque quería avanzar lo máximo posible antes de empezar las prácticas y empecé a hacer el registro. La fase de análisis fue mucho después cuando implementé los chats y con ello SignalR y busqué la manera de hacerlo lo más eficiente posible (en un principio los chats no tendrían persistencia para hacer la aplicación más rápida y eficiente). Al diseño como tal le dediqué un par de días donde elegí las librerías y encontré las vistas del repositorio antes mencionado.

En cuanto a los cambios entre lo presentado y lo que se quería hacer en un principio son varios sobretodo de funcionalidad:

- En un principio como comenté no quería que los chats fueran persistentes.
- Cuando me decidí a implementar persistencia en los chats pensé en una manera eficiente de hacerlo: quería que los chats hicieran una sola llamada a la base de datos cuando ambos usuarios cerrasen ese chat, intentaré explicarlo más adelante porque aquí creo que no procede.
- Tenía pensado en un principio implementar una funcionalidad para hacer videollamadas entre usuarios y por tiempo no he podido mirarlo
- En general tenía pensado que la aplicación tuviera más funcionalidades y vistas

Las tareas que llevaron más de lo esperado fue sobretodo la funcionalidad del chat persé puesto que como ya he dicho quería hacerla eficiente y lo más rápida posible y me ha tocado rediseñar el esquema de SignalR un par de veces junto con la interfaz del chat puesto que tenía que mostrar las fotos de perfil de nuestro usuario y el amigo con el que se está hablando y la imagen en base64 era muy grande para pasarlo por signalR asi que me tocó hacer unos condicionales anidados para mostrar la foto del user o el amigo y tambien mostrar la imagen solo en el primer mensaje al igual que si las horas de un mensaje que estaba encima de otro no variaba no mostrarl... Otra tarea que me dio muchísimos dolores de cabeza y me rompió el código por completo fueron las inyecciones de clases (otra cosa que también comentaré más adelante)

Análisis y diseño del proyecto

Descripción de la arquitectura web

Mi aplicación está basada en Blazor Web Assembly hosted, es una arquitectura que nos brinda un modelo SPA el cual tiene algunas diferencias que he podido experimentar y me ha costado entender respecto a un modelo tradicional de un framework de front como puede ser React y uno de back como puede ser Node.

- La principal diferencia es el lenguaje, en Blazor WASM usamos C# para la parte del cliente y la del server.
- Otra que a mi me costó hacerme a la idea es que comparten puerto, corren a la vez como si fueran un único proyecto, el front “cuelga” del back, al lanzar el server automáticamente llama al cliente y compila (esto me generó los problemas con las inyecciones de dependencias)
- Al ser más “de nicho” no hay tanta información de estos frameworks, todo lo que engloba a Microsoft está muy cerrado por decirlo de alguna manera.

Tecnologías y herramientas utilizadas

- Frontend: Blazor Web Assembly
- Backend: host .Net 9.0
- Base de datos: Mongo Compass en local y Atlas en el despliegues
- Integración y pruebas: He realizado pruebas manuales de funcionamiento de toda la app
- Seguridad: Bcrypt principalmente
- Despliegue y Hosting: Render.com

Análisis de usuarios

Los usuarios que usarán la app son usuarios finales estándar, similares a los de cualquier red social o aplicación de mensajería. No se contemplan perfiles administrativos complejos, ya que el enfoque es simple, directo y centrado en la experiencia de usuario

Requisitos

- Funcionales:

La aplicación debe ser capaz de manejar una sesión mediante el ciclo de vida normal y corriente de cualquier aplicación web (Login y Logout).

Tenemos que ser capaces de registrarnos persistiendo los datos en la base de datos

La aplicación nos debe permitir chatear con otros usuarios

Debemos de ser capaces de personalizar nuestro perfil

Debemos poder acceder a la misma a través de la url ChatApp

- No funcionales:

Que los datos de la aplicación se mantengan actualizados sin necesidad de interacción por el usuario (recibir eventos como eliminaciones de las listas de otros usuarios)

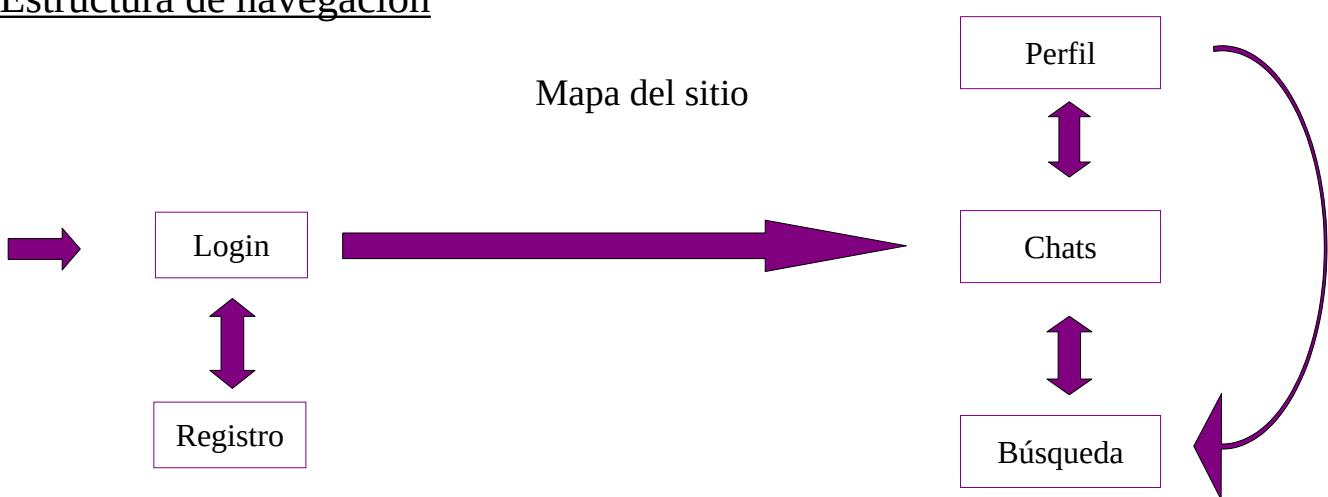
Que la aplicación sea rápida y eficiente

Que sea visible en distintos tipos de dispositivos

Los datos como contraseñas no deben ser almacenados en texto plano

Información sensible como la dirección de correo no puede ser visible por cualquier usuario

Estructura de navegación



Organización de la lógica de negocio

- Lógica del backend: En cuanto a la lógica del backend pues tenemos una serie de modelos que se encuentran solo en el back (punto importante: los modelos del cliente si no se les asigna un namespace son compartidos con el server pero por el contrario a los del server no se pueden acceder desde el cliente) principalmente la mayoría de modelos son las entidades que luego se van a persistir en base de datos. Tenemos una clase Mapper, y lo típico, una configuración para el cliente de mongo, un servicio para operaciones contra base de datos y un servicio Rest para la llamada a la api que se usará en la aplicación.
En nuestro Program.cs que es la clase principal tanto del proyecto del servidor como el del cliente definimos los endpoints que llaman a los controladores para devolver los distintos datos. Lo interesante de verdad es la clase ChatHub.cs que es la que gestiona los eventos de signalR, me gusta pensar que es como si definieras endpoints pero para SignalR, yo por ejemplo en el cliente tengo un ChatService que define una variable tipo HubConnection y desde esa variable se usa el método InvokeAsync para llamar a los métodos definidos en el ChatHub del servidor por su nombre.
- Conexiones a APIs y servicios externos: como era un punto obligatorio para el trabajo me decidí por usar una muy ligera, gratis y que me devolviera algún dato problemático que en mi caso son las fechas. Utilizo WorldTimeAPI para quitarme problemas con las fechas y sobretodo con las zonas horarias. Esto acabó en mi contra porque en mi servidor donde desplegué la aplicación no me dejaba hacer fetch a apis externas

Modelo de datos simplificado

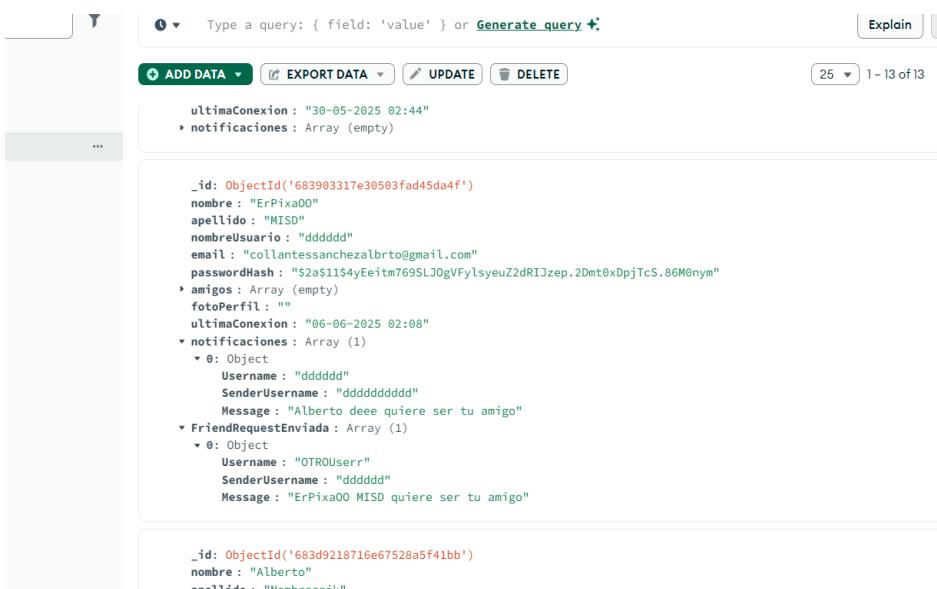
Elegí una base de datos no relacional para agilizar el proceso de diseño y puesto que los modelos que iba a guardar no son muy complejos ni guardan relaciones complejas entre si.

Mi base de datos se compone de una colección de usuarios que guarda objetos Usuario los cuales están compuestos de las siguientes propiedades:

- Nombre: nombre del usuarios
- Apellido
- nombreUsuario: esto es el username y es la clave en la colección, se utiliza como si fuera una PrimaryKey puesto que no puedo guardar dos usuarios con

el mismo username y es lo que voy a utilizar para casi cualquier operación contra bdd

- email
- passwordHash: contraseña encriptada
- amigos: es un array de strings en el que se guardan los usernames de los amigos agregados, lo suyo hubiera sido guardar el objeto User correspondiente pero eso lo haría más lento puesto que hay veces que no necesito la imagen de perfil que es lo más pesado y costoso de trabajar con ello en lo que a recursos se refiere
- fotoPerfil: imagen hasheada en base64
- ultimaConexion: en string y con la zona horaria de Madrid (obtenida a través de la api)
- notificaciones: es un array de objetos FriendRequest que tiene 3 propiedades:
 1. Username: hace referencia al receptor de la solicitud
 2. SenderUsername: del que manda la petición de amistad
 3. Message: un mensaje meramente informativo
- FriendRequestEnviada:
 1. Username: el usuario que recibe la solicitud
 2. SenderUsername: el que manda la solicitud, en este caso nosotros
 3. Message: mensaje descriptivo



The screenshot shows a MongoDB query interface with the following details:

- Query bar: Type a query: { field: 'value' } or [Generate query](#).
- Buttons: ADD DATA, EXPORT DATA, UPDATE, DELETE.
- Pagination: 25 | 1 - 13 of 13.
- Document 1 (Expanded):
 - ultimaConexion : "30-05-2025 02:44"
 - notificaciones : Array (empty)
 - ...
 - _id: ObjectId('683903317e30503fad45da4f')
nombre : "ErPixaOO"
apellido : "MISD"
nombreUsuario : "ddddd"
email : "collantessanchezalbrto@gmail.com"
passwordHash : "\$2a\$11\$4yEeitm769SLJ0gVFylsyeuZ2dRIJzep.2Dmt0xDpjTcS.86M0nym"
amigos : Array (empty)
fotoPerfil : ""
ultimaConexion : "06-06-2025 02:08"
notificaciones : Array (1)
 - 0: Object
 - Username : "ddddd"
 - SenderUsername : "ddddd"
 - Message : "Alberto dee quiere ser tu amigo"
FriendRequestEnviada : Array (1)
 - 0: Object
 - Username : "OTROUserr"
 - SenderUsername : "ddddd"
 - Message : "ErPixaOO MISD quiere ser tu amigo"
- Document 2 (Collapsed):
 - _id: ObjectId('683d9218716e67528a5f41bb')
nombre : "Alberto"
apellido : "NombreApellido"

La otra colección que utilzo en mi base de datos es una llamada chats la cualm almacena objetos Chat que se componen de dos propiedades:

1. Un _id el cual se compone de los nombres de los dos usuarios que comparten el chat separados por una _ y ordenados alfabéticamente, esto de nuevo tiene mucha importancia puesto que ese mismo id es la “url” que usamos en SignalR para crear Groups entre esos usuarios
2. Un array de objetos ChatMessage los cuales se componen de cuatro propiedades: Username, Message, Time y IsRead

```
_id: "ddddd0000_Pixita0000"
Mensajes: Array (1)
  0: Object
    Username : "pixita0000"
    Message : "aaaaa"
    Time : "06-06-2025 01:40"
    IsRead : false
```

Conclusiones

En lo referente a los resultados obtenidos se ha logrado construir una aplicación lo suficientemente robusta, no dabría decir si es eficiente o no puesto que en un entorno real con un deploy en condiciones habría que ver el rendimiento y a mi gusto atractiva visualmente aparte de polivalente puesto que se ha diseñado para usarse en móvil también.

En los retos encontrados se pueden destacar unos cuantos:

El intentar desenvolverse en un entorno completamente desconocido ha sido agotador, he tenido que leer mucha información y constantemente me he ido enfrentando a nuevos problemas relacionados con el entorno. Por ejemplo: he tenido que aprender C# desde 0 y empezar a programar con un framework directamente lo que me ha dado problemas tan básicos que han sido difíciles de encontrar porque lo obvio es muy difícil de buscar. Os pongo en contexto: Yo desde un principio quería hacer código modularizado usando servicios e inyectándolos para poder utilizarlos en mis componentes. Esto no sería un problema si no fuera porque yo desde el principio he usado

@rendermode="InteractiveWebAssembly". ¿Qué es esto? Esto es la nueva forma de definir un modelo WASM con un pre-renderizado desde el servidor, el cual te da una primera carga desde el server en lo que tarda en cargar Blazor. ¿Qué problema ha habido con esto? Yo a la hora de registrar en el módulo de dependencias lo hacía sólo en el cliente puesto que solo lo iba a usar ahí y claro, al hacer f5 sobre una vista con una inyección de un modelo lo intentaba cargar desde el servidor, si no lo registraba en el servidor me tiraba un error y cuando lo registraba en el servidor caía en el error de copiar la clase en el proyecto del server cambiándole el namespace haciendo así que aunque la clase del servidor y la del cliente fueran una copia la una de la otra, lo detectase como dos clases distintas. Me costó mucho darme cuenta de cómo compartir modelos entre el cliente y el servidor y aun así no lo hago de la manera correcta puesto que se deberían guardar los modelos compartidos en un proyecto shared aparte y acceder a los mismos desde cualquiera de los otros dos proyectos. Me obcequé en hacerlo lo más simple posible con las menores cargas al servidor y eso me hizo equivocarme muchas veces y con ello perder tiempo. El otro caso que quería comentar era que al hacer la persistencia de chats lo quería hacer de tal manera que solo uno de los usuarios hiciera una petición al server para persistir cuando la sala de chat (Group de SignalR) se quedara vacía. Perdí un día pensando e intentando implementar esta idea y al final tiré la toalla y me decanté por persistir cada mensaje cuando se enviaba.

Otra de las dificultades ha sido el entender un framework tan distinto como es Blazor, el enfrentarme a problemas de una librería que no conoce mucha gente y sobretodo el solapar la formación en el centro de trabajo con hacer un trabajo tan pesado como este.

He aprendido a desenvolverme en un entorno nuevo, siento que he aprendido a programar y que este trabajo, aunque no sea perfecto, es completamente mio. En cuanto a las mejoras futuras tengo pensado implementar funcionalidades hasta que me canse del entorno.

Bibliografía

- Microsoft. (2024). *ASP.NET Core Blazor documentation*. Disponible en: <https://learn.microsoft.com/en-us/aspnet/core/blazor>
- Microsoft. (2024). *SignalR for ASP.NET Core*. Disponible en: <https://learn.microsoft.com/en-us/aspnet/core/signalr>
- MongoDB Inc. (2024). *MongoDB Atlas Documentation*. Disponible en: <https://www.mongodb.com/docs/atlas/>

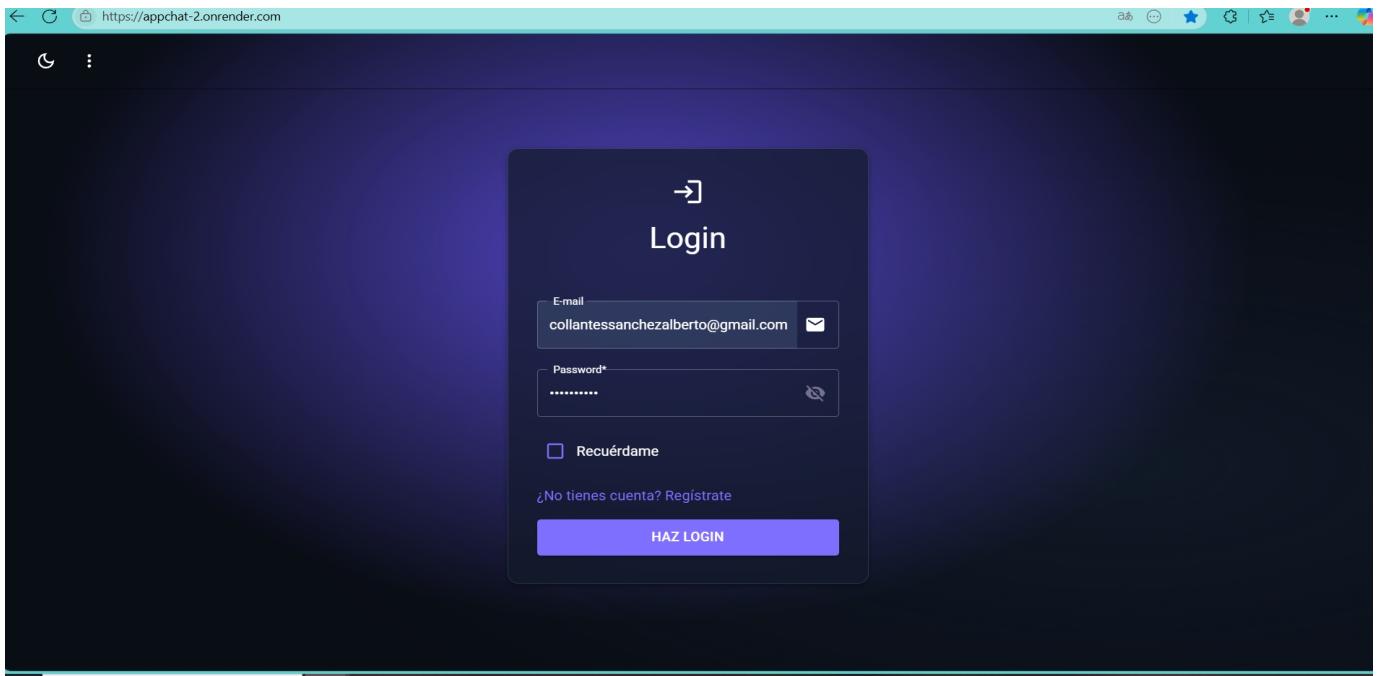
- MongoDB C# Driver Team. (2024). *MongoDB .NET Driver documentation*. Disponible en: <https://mongodb.github.io/mongo-csharp-driver/>
- MudBlazor Team. (2024). *MudBlazor - Blazor Component Library*. Disponible en: <https://mudblazor.com/>
- Render. (2024). *Render Hosting Documentation*. Disponible en: <https://docs.render.com/>

Anexos

Manual de usuario

Vamos a hacer un análisis de todas las vistas de la app y sus componentes

El ciclo de vida de la app nace en la vista login, a la cual accede el usuario siempre que entra en la app, que el usuario esté logueado es un requisito para poder navegar por la aplicación

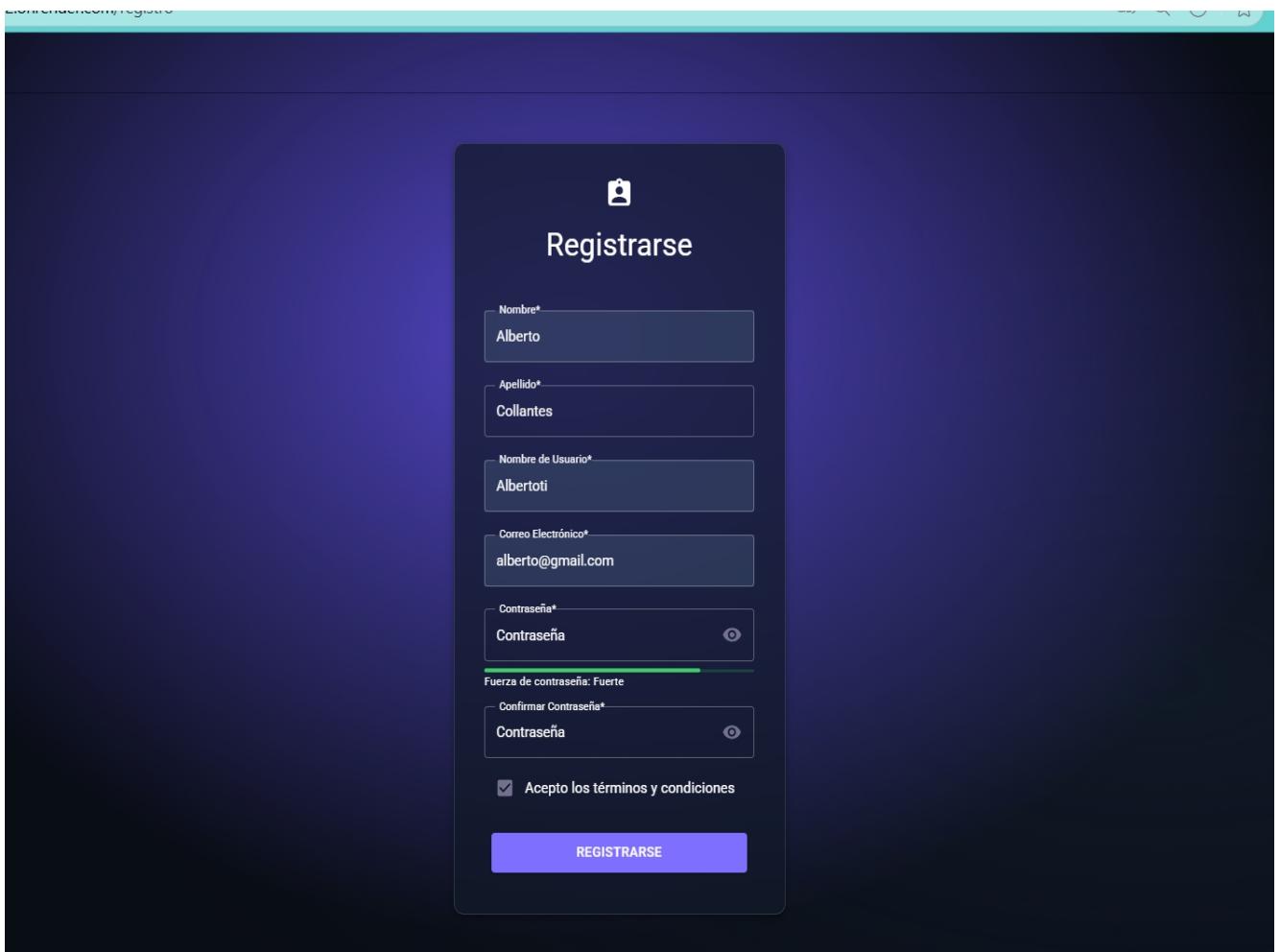


Como podemos observar tenemos un formulario simple que cuenta con dos campos, uno para el email y otro para la password.

Podemos ver también un link que dirige a la vista del registro.

Y por encima de nuestras vistas tendremos siempre una navbar que contendrá elementos para facilitar la navegación así como la funcionalidad de cambiar el tema de la app de claro a oscuro respectivamente. Esta navbar cambiará cuando hagamos login mostrándonos nuestra imagen de perfil

Pasamos a la vista de registro



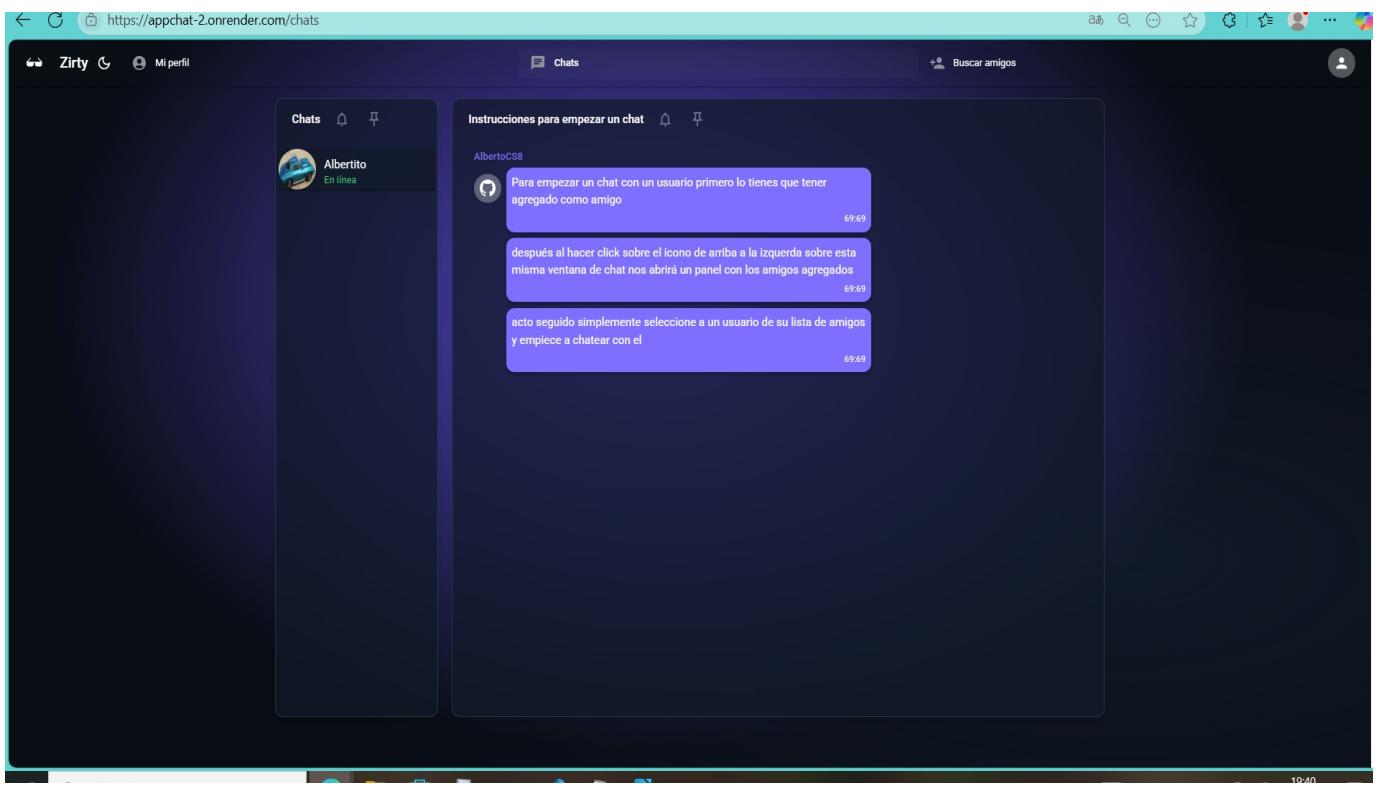
En esta vista como no hay mucha diferencia con el login destacaremos solo dos cosas:

la barra de fuerza de la contraseña que cada vez que se escribe en ella llama a una función que calcula un valor del 0 al 100 dependiendo de los requisitos típicos que cumpla tu contraseña (letra mayúscula y minúscula, un número, y que tenga 8 o más caracteres)

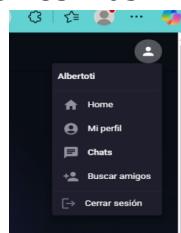
y la imposibilidad de registrarte varias veces con un mismo email o un mismo username puesto que son datos que suelen ser tratados como únicos en bases de datos.

Si hacemos registro nos manda directamente al login y si nos logueamos vamos directamente a nuestro componente de chats

Ahora vamos a pasar a analizar exhaustivamente nuestra vista porque es la principal de la aplicación



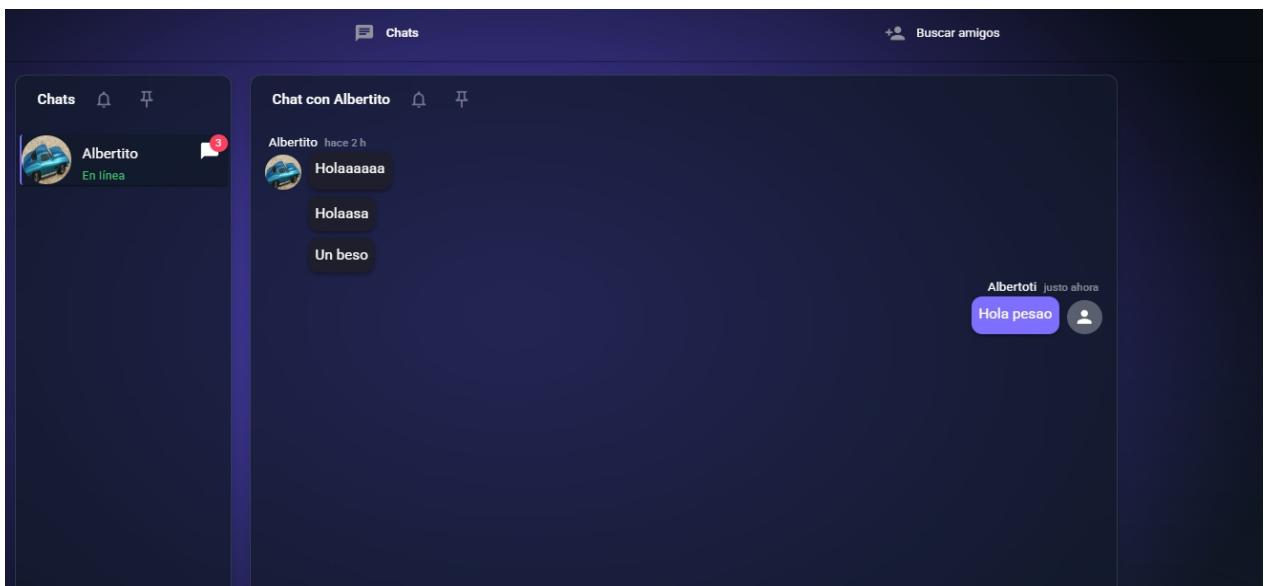
Como podemos ver nuestra navbar ha cambiado y ahora nos muestra los elementos de navegación necesarios para movernos por la página así como un ícono que muestra nuestra foto de perfil (en este caso no tenemos pero la cambiaremos más adelante) que nos sirve también de menú desplegable para navegar



En el lado izquierdo encontramos una lista de amigos que se cargan con los datos que hemos pedido al servidor cuando hacemos login, esa lista permanece inmutable a menos que un usuario te acepte una petición de amistad o todo lo contrario, te borre de su lista de amigos, los eventos en tiempo real como pueden ser envíos de mensajes o conexiones son enviados a través de signalR para actualizar la interfaz del otro usuario

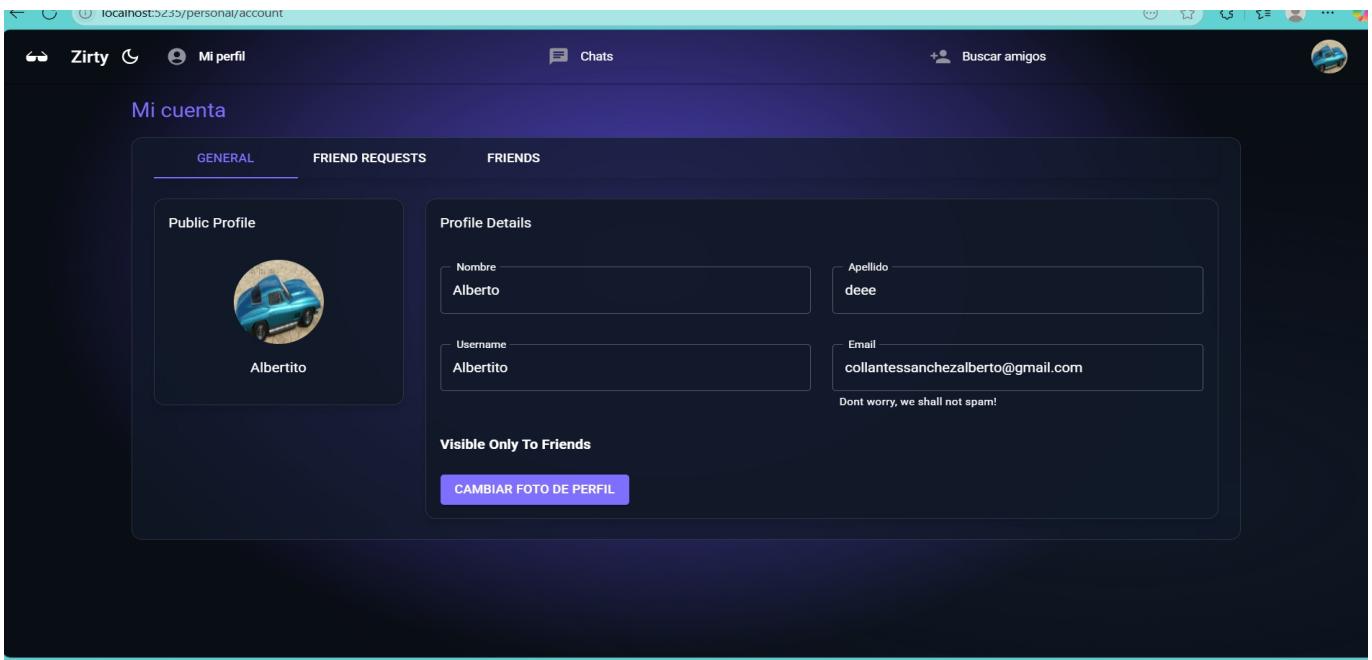


Y al hacer click en el usuario deseado y solo en ese momento se hace una petición al servidor para cargar el chat con ese usuario y se enlazan los dos usuarios a través de signalR para poder enviarse mensajes

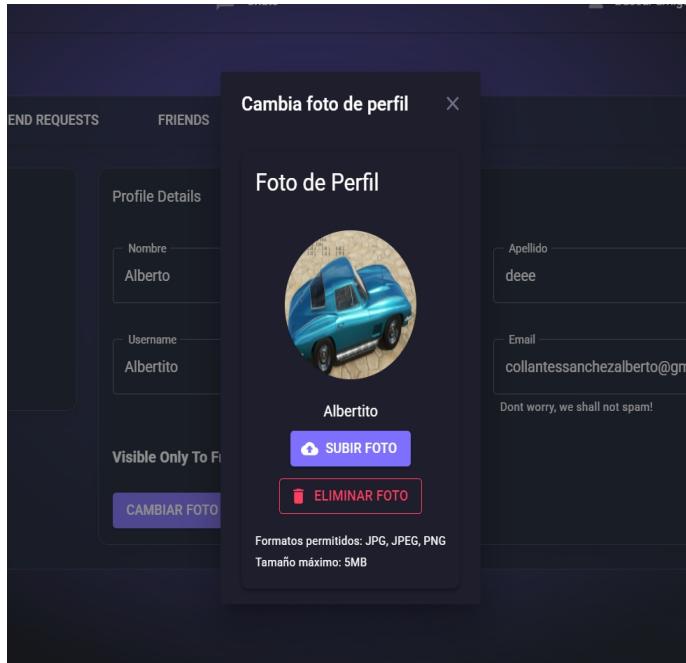


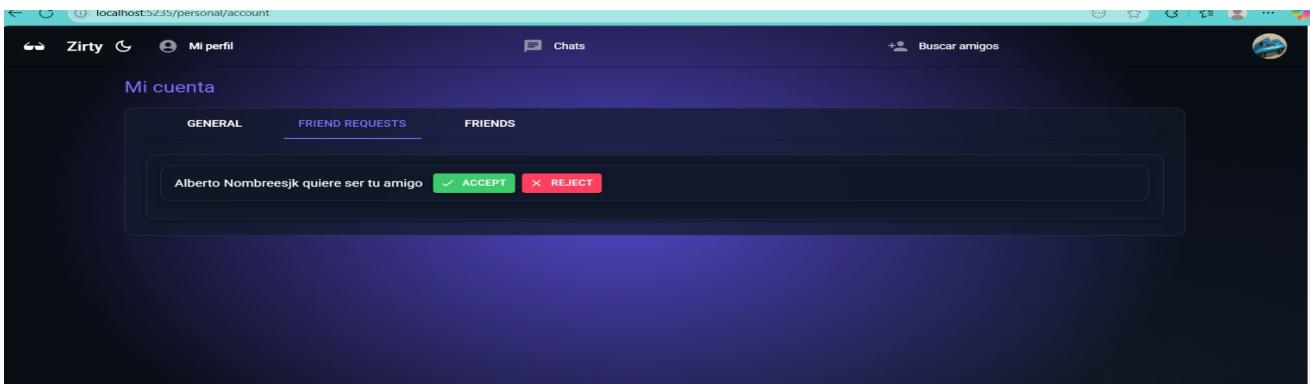
Como se puede ver cada mensaje renderiza la imagen de perfil del usuario y muestra aproximadamente hace cuánto se ha mandado el mensaje.

A continuación veremos la vista del perfil



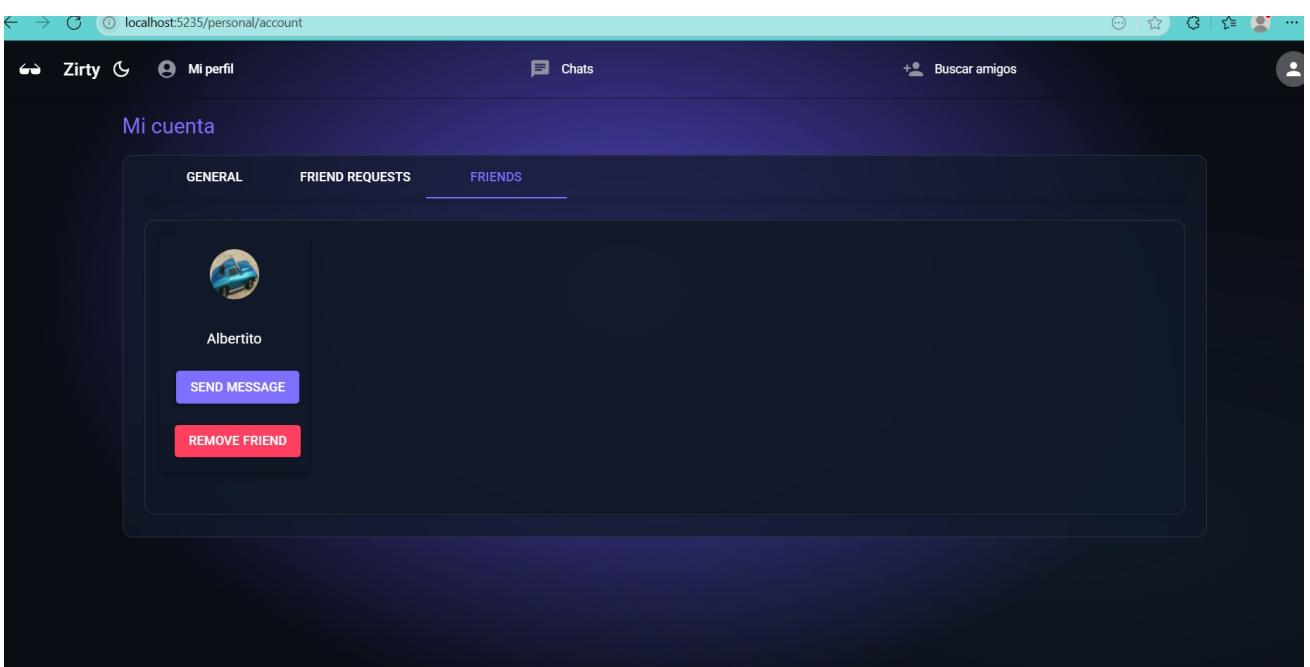
Tenemos la información de nuestro perfil que no es visible en ningún otro sitio de la app como el email, si pulsamos en cambiar foto de perfil se nos abre un modal que nos permite subir una foto y meterla en base de datos para que sea nuestra nueva imagen de perfil





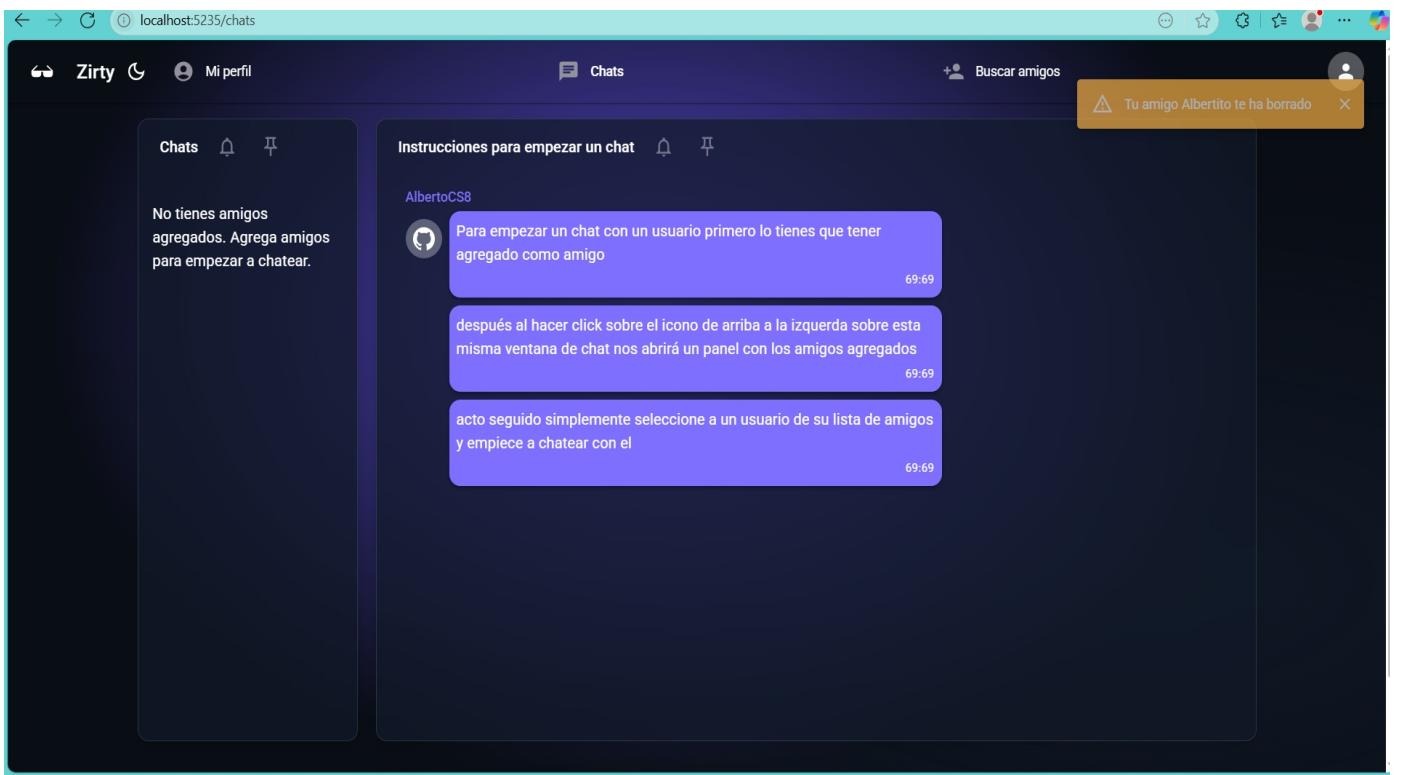
Este es el panel de solicitudes de amistad, cuando un usuario está en otra parte de la aplicación (suelen estar en chats) y recibe una solicitud de amistad es notificado al momento de que tiene una solicitud de amistad y cuando va al panel de friend request ve esto que le dice quien le ha solicitado amistad y si quiere aceptarla o rechazarla. Cuando acepta el usuario se le agrega un amigo en base de datos y si está conectado se le agrega a su “almacenamiento local”.

Si el usuario accede a la siguiente vista vería algo como esto

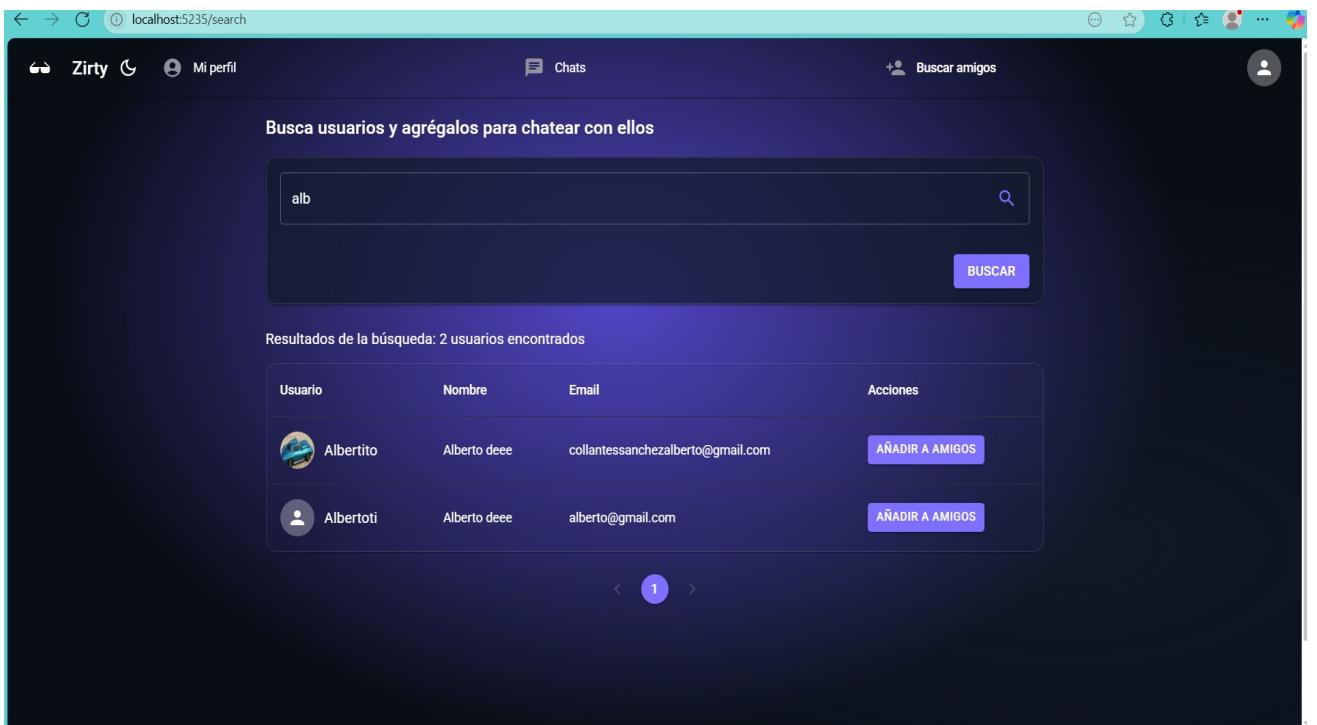


En esta vista se ven los usuarios que tienes agregados como amigos y te da la opción de cargar su chat para enviarles un mensaje o eliminarlos de tu lista de amigos.

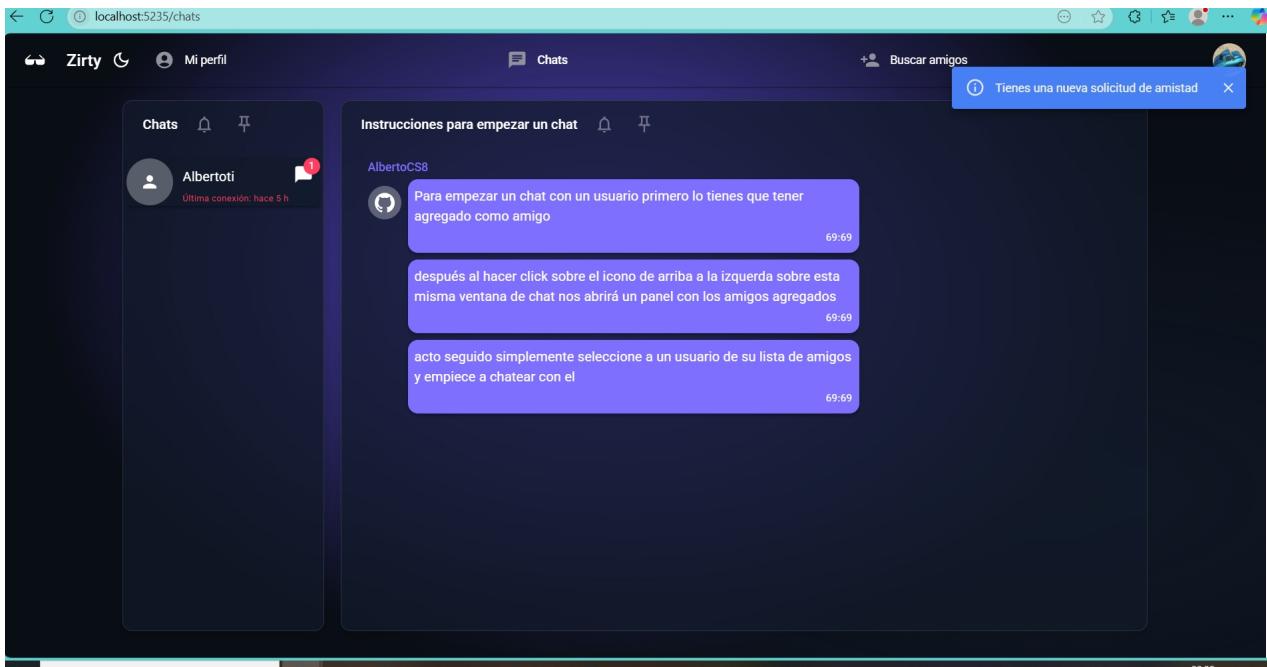
Si el usuario elimina a un amigo de su lista se envía un evento por SignalR y refresca la interfaz del usuario destino eliminando también de su almacenamiento local al usuario



Por ultimo presentaremos la última vista que es la vista que usaremos para buscar otros usuarios en la aplicación



y si pulsamos en añadir a amigos si el usuario está conectado recibe la notificación



Guía de instalación y despliegue

El proyecto estará alojado en mi github al que podréis acceder clicando al siguiente enlace: [repositorio](#)

Primero necesitaréis descargaros .Net, más en concreto la versión más reciente que es la 9.0

Para correrlo en local únicamente tendréis que clonaros el repositorio en vuestro local y desde la carpeta del server (./pruebaMudBlazor) ejecutar el comando dotnet run y el proyecto ya estará ejecutándose en local.

Para desplegarlo la cosa se complica algo más pero gracias a que render es un servidor de despliegue muy simplificado se hace bastante fácil si tienes el conocimiento necesario.

Primero nos tendremos que loguear en render.com. Cuando nos logueemos nos encontraremos con esta pantalla

The screenshot shows the Render dashboard at <https://dashboard.render.com>. The left sidebar includes sections for Projects, Blueprints, Environment Groups, Integrations, and Workspace (Billing, Settings). The main area has a header with a search bar, a 'New' button, and upgrade options. The 'Overview' section features a 'Projects' card for 'Alberto' (No active services) and a 'Create new project' button. Below is an 'Ungrouped Services' section with a table showing one service: AppChat-2 (Deployed, Docker, Frankfurt, 11h).

Esta pantalla es nuestro dashboard, primero tendremos que crearnos un nuevo servicio

The screenshot shows the 'Add new' modal on the Render dashboard. It lists various service types: Static Site, Web Service (selected), Private Service, Background Worker, CronJob, Postgres, Key Value, Project (PRO), and Blueprint. The 'Web Service' option is highlighted.

Si la cuenta con la que te has logueado es la que está asociada a tu cuenta de github detecta automáticamente los repositorios que tienes. Elegimos el de nuestra aplicación de chat y veremos que la aplicación detecta automáticamente (más o menos) el entorno en el que está basado el repositorio.

The screenshot shows the Render.com dashboard at <https://dashboard.render.com/web/new>. The user is creating a new web service. The interface includes:

- Project**: Optional. A dropdown menu labeled "Select a project..." with a placeholder "Select an environment...".
- Language**: Docker.
- Branch**: main.
- Region**: Frankfurt (EU Central) is selected, indicated by a purple outline. Other options include "Deploy in a new region".
- Root Directory**: Optional. The path "./pruebaMudBlazor" is entered.
- Dockerfile Path**: Dockerfile.

Como podemos ver nos ha detectado Docker de forma nativa y si abrimos el desplegable veremos que no hay opción para seleccionar .Net, ¿Por qué ocurre esto? Render no tiene soporte nativo para .Net así que tendremos que instalar mediante un DockerFile .Net y configurar el entorno. El DockerFile se encuentra en mi repositorio así que con poner el directorio y la ubicación del DockerFile simplemente tenemos que darle a siguiente y Render se encargará de desplegarlo y darnos una url para que podamos acceder.