Práctica Profesional Supervisada

Leonel Alejandro Rebollini

Desarrolló de una plataforma web para gestión de órdenes en Diserglass

Instituto de Ingeniería y Agronomía

Carrera: Ingeniería en Informática

2025

Universidad Nacional Arturo Jauretche

Instituto de Ingeniería y Agronomía

Carrera de Ingeniería en Informática

PRÁCTICA PROFESIONAL SUPERVISADA
Informe final

DESARROLLÓ DE UNA PLATAFORMA WEB PARA GESTIÓN DE ÓRDENES EN DISERGLASS

Leonel Rebollini

Florencio Varela, agosto 2026

DESARROLLO DE UNA PLATAFORMA WEB PARA GESTIÓN DE ÓRDENES EN DISERGLASS

Estudiante

- **Apellido y Nombres**: Rebollini, Leonel Alejandro
- **Correo electrónico**: LeoRebollini@gmail.com

Organización donde se realiza la Práctica Profesional Supervisada

- **Diserglass**

Tutor Organizacional

- **Apellido y Nombres**: Mogni, Victor
- **Correo electrónico**: Victor@diserglass.com.ar

Docente Supervisor

- **Apellido y Nombres**: Osio, Jorge
- **Correo electrónico**: Jorge.Osio@ing.unlp.edu.ar

Docente tutor del taller de apoyo para la producción de textos académicos

- **Apellido y Nombres**: Kelly, Carolina
- **Correo electrónico**: kellygcarolina@gmail.com

Coordinador de la carrera de Ingeniería en Informática

- **Apellido y Nombres**: Morales, Martin
- **Correo electrónico**: martin.morales@unaj.edu.ar

Resumen

La presente Práctica Profesional Supervisada (PPS) se llevó a cabo en la empresa de ***Diserglass***, con el objetivo de desarrollar una aplicación web destinada a optimizar la gestión de pedidos internos, el seguimiento de órdenes y la organización de tareas administrativas.

El proyecto fue realizado en conjunto con **Alan Marengo**, quien abordó principalmente el *backend,* mientras que el presente informe se centra en el desarrollo del *frontend*.

Durante el proyecto se tomaron decisiones arquitectónicas y se seleccionaron tecnologías modernas como *React, Tailwind CSS y .NET*. Se aplicaron buenas prácticas como *Clean Architecture* y una clara separación en capas.

El diseño de interfaces se realizó en *Figma, con* vistas armadas para los distintos dispositivos, y se implementó una *API REST* para la gestión de datos internos. Además, se desarrolló un componente de sincronización encargado de integrar los datos de los sistemas existentes con la nueva plataforma.

También, se configuraron mecanismos de validación, autenticación, auditoría y seguridad. Finalmente, se desplegó la solución en un entorno de producción, acompañada de su respectiva documentación funcional y técnica.

Abstract

Development of a Web Platform for Order Management at Diserglass

This Supervised Professional Practice (SPP) was carried out at the company **Diserglass**, aiming to develop a web platform designed to optimise the management of internal orders, order tracking, and administrative task organisation.

The project was carried out in collaboration with **Alan Marengo**, who primarily focused on the backend, while the present report is centred on the development of the frontend.

Key architectural decisions were made throughout the Project, and modern technologies such as *React*, *Tailwind CSS*, and *.NET* were selected. Best development practices were applied, including Clean Architecture and a clear separation of concerns.

The user interfaces were designed in Figma with support for multiple devices, and a REST API was implemented to manage and power the new platform. Additionally, a synchronization component was developed to integrate data from existing systems with the new platform.

In addition, validation, authentication, auditing, and security mechanisms were also configured. Finally, the solution was deployed in a production environment, along with its corresponding functional and technical documentation.

Tabla de contenido

1.Introducción

1.1. Justificación

El presente proyecto surge como respuesta a una necesidad que se presentaba en la empresa de *Diserglass*, la cual se dedica a la fabricación y distribución de vidrio.

Antes de la implementación del sistema, *Diserglass* gestionaba sus pedidos de forma fragmentada y desorganizada, estos se recibían por múltiples canales sin un formato estandarizado, como puede ser *WhatsApp* o llamadas telefónicas, lo cual dificultaba la trazabilidad de los pedidos y generaba errores operativos. Por otra parte, no existía una plataforma que permitiera la visualización y seguimiento de órdenes, lo que limitaba el acceso del cliente a información actualizada sobre sus pedidos.

La solución desarrollada hacía estos problemas mencionados fue la implementación de una plataforma web desde la cual se puede tener acceso a la visualización de órdenes históricas y actuales, además de poder realizar seguimiento de las mismas y, en una siguiente etapa, se realizará el módulo para la toma de pedidos mediante la plataforma web. Esta evolución hacía lo digital representa un impacto significativo tanto a nivel organizacional como en la relación con sus clientes, generando mayor orden, agilidad en sus procesos y reducción de errores; con expectativas de mejora en la productividad.

Desde lo personal, elegí esta propuesta como Práctica Profesional Supervisada (PPS) porque me permitió adquirir diversos conocimientos y experiencia en distintos campos, desde el desarrollo *frontend* donde tuve que realizar el diseño en *Figma* hasta su implementación técnica, como en reuniones con el cliente, colaborar en la toma de decisiones, y asumir compromisos con fechas de entrega. Fue una experiencia laboral única, que me desafió constantemente.

Durante el proyecto, adquirí conocimientos en herramientas como *Axios*, *Redux* y *Figma*, además de profundizar en otras como *React*, *TypeScript* y *Tailwind CSS*. Con el fin de entender los distintos aspectos del ciclo completo de desarrollo de software colaboré ocasionalmente con tecnologías *.NET 8*, *Entity Framework* y *Dapper*.

A nivel tecnológico, el software desarrollado puede ser replicado en otras pequeñas y medianas empresas del rubro industrial, las cuales aún no han adoptado sistemas digitales para la gestión de pedidos o visualización de órdenes. Con estas expectativas, el sistema podría convertirse en una

plataforma base con potencial de crecimiento y aplicación en diferentes entornos productivos.

Por último, el impacto del proyecto en el campo económico y organizacional: permite reducir tiempos y errores, mejorar la atención al cliente y preparar las bases para una evolución digital. A partir de esto, desde el punto de vista de la empresa hubo una mejoría en su forma de operar; desde lo académico y profesional, ofrece un caso concreto de cómo la tecnología puede transformar procesos reales.

1.2. Objetivos

1.2.1. Objetivo general

Diseñar e implementar una plataforma web para la empresa *Diserglass*, que optimice la forma en que sus clientes interactúan con los procesos de compra. En una primera instancia el sistema les brinda la posibilidad de registrarse, consultar el estado actual de sus pedidos y revisar el historial de órdenes realizadas.

Esta solución se plantea como un punto de partida hacía una mayor digitalización en un rubro que no suele priorizar este tipo de inversiones tecnológicas. Por otra parte, se plantea la incorporación del módulo para la toma de pedidos.

El sistema será desarrollado bajo una arquitectura limpia, utilizando tecnologías modernas tanto en *frontend* como en *backend*, priorizando la escalabilidad y mantenibilidad del código; facilitando nuevas incorporaciones y garantizando una experiencia de usuario eficiente, segura y escalable.

1.2.2. Objetivos específicos

- Diseñar la interfaz de usuarios en herramientas como Figma, respetando criterios de usabilidad y accesibilidad.
- Implementar la interfaz del sistema utilizando tecnologías como *React*, *TypeScript* y *Tailwind CSS*.
- Desarrollar el flujo de registro y autenticación de usuarios, permitiendo registrarse aquellos que verifiquen su Clave Única de Identificación Tributaria (CUIT) y utilizando verificación mediante contraseña de un solo uso (OTP, por sus siglas en inglés *One-Time Password*) vía correo electrónico.

- Permitir que cada usuario visualice exclusivamente sus órdenes activas e históricas, brindando un estado actual de la orden y permitiendo realizar filtrados según distintos criterios.
- Integrar la comunicación con la interfaz de programación de aplicaciones (API, por sus siglas en inglés *Application Programming Interface*) desarrollada para obtener y actualizar información en tiempo real.
- Participar en reuniones con el cliente y colaborar en la toma de decisiones del sistema.
- Adaptar y mejorar la interfaz a partir de la retroalimentación (*feedback*) recibida por parte de los usuarios y clientes.
- Preparar el entorno de producción para el despliegue en *Internet Information Service* (IIS)
- Documentar el desarrollo y pruebas realizadas en la plataforma.
- Planificar el desarrollo de un módulo futuro para la cotización de pedidos, orientado a reducir la carga operativa de la empresa.

1.3. Alcance

La presente propuesta de alcance abarca el diseño e implementación del **Frontend** de una aplicación web destinada a mejorar la gestión de pedidos en la empresa de *Diserglass*. Este alcance comprende la elaboración de las vistas en *Figma*, la toma de decisiones junto al cliente, hasta el desarrollo técnico del sistema desde el lado del cliente.

El sistema permitirá a los usuarios registrarse, visualizar el estado actual de sus órdenes y acceder al historial de pedidos realizados. El trabajo incluirá reuniones con el cliente para la toma de decisiones, diseño de interfaces en *Figma*, y el desarrollo técnico utilizando tecnologías modernas como *React*, *TypeScript* y *Tailwind CSS*. Además, se va a implementar un sistema de autenticación mediante *Json Web Token* (*JWT*), validaciones de errores y pruebas de funcionamiento para asegurar la correcta visualización en distintos dispositivos, con el fin de garantizar una buena experiencia de usuario.

Desde el punto de vista arquitectónico, y cómo solicitud del cliente, se tuvo en cuenta la incorporación de un componente adicional denominado **Sincronizador**, el cual se instala en la infraestructura de la empresa. Este módulo interno es el único autorizado para comunicarse con la *API* principal, y se encarga de mantener actualizada la base de datos del sistema con la información sobre los clientes registrados y sus pedidos. Esta decisión busca

mejorar la seguridad encapsulando la información sensible de la empresa; estableciendo un flujo de sincronización controlado donde la ***API*** nunca inicia peticiones, sino que responde únicamente a las solicitudes del *Sincronizador*.

1.4. Requerimientos

1.4.1. Requerimientos funcionales

- **RF1 – Registro de clientes**

  El sistema debe permitir que los clientes creen sus propias cuentas mediante un formulario de registro.

- **RF2 – Autenticación segura**

  La plataforma debe implementar un sistema de inicio de sesión basado en *JWT*, permitiendo a los usuarios acceder de forma segura.

- **RF3 – Visualización de pedidos actuales**

  El sistema debe brindar al cliente el estado actual de sus órdenes

- **RF4 – Visualización del historial de pedidos**

  Los usuarios deben poder acceder a un historial de pedidos realizados durante el último año.

- **RF5 – Sincronizador de datos**

  El sistema debe recibir actualizaciones desde el **Sincronizador** instalado en la estructura de la empresa. Este módulo envía información actualizada a la *API* principal sobre los clientes y pedidos.

- **RF6 – Composición de cada pedido**

  La plataforma debe permitir visualizar la composición de cada pedido, incluyendo los materiales, dimensiones y pedidos.

- **RF7 - Diseño responsivo**

  Se debe poder visualizar correctamente la interfaz en distintos dispositivos (*desktop*, *tablet* y *mobile*)

- **RF8 – Manejo de errores**

  El sistema debe validar los datos ingresados por el usuario en formularios, y en caso de ser necesario, mostrar el mensaje de error adecuado.

- **RF9 – Planificación del módulo de cotización**

  Definición y alcance preliminar del módulo destinado a la cotización de pedidos. El desarrollo de este módulo queda planificado para una etapa posterior a la PPS.

1.4.2. Requerimientos no funcionales

- **RNF1 – Tecnologías modernas**

  El sistema debe estar desarrollado con tecnologías modernas como *React*, *TypeScript* y *Tailwind CSS* en el *frontend*, *.NET 8*, *C#* y *PostgreSQL* en el *backend*.

- **RNF2 – Arquitectura limpia (*Clean Architecture*)**

  El proyecto debe implementar los principios de arquitectura limpia, permitiendo la escalabilidad, mantenibilidad y separación de responsabilidades.

- **RNF3 – Seguridad de la información**

  Debe garantizarse el encriptado de contraseñas, autenticación segura y encapsulamiento de la información sensible.

- **RNF4 – Diseños en Figma**

  El cliente debe dar una aprobación de la interfaz previamente diseñada en *Figma*.

- **RNF5 – Disponibilidad**

  La plataforma debe estar disponible permanentemente para los clientes.

- **RNF6 – Control de versiones y metodología ágil**

  El desarrollo debe realizarse mediante una metodología ágil (*Scrum*), gestionando tareas con *Jira* y *GitHub* para el control de versiones.

- **RNF7 – Despliegue en IIS**

  El entorno de producción debe estar preparado para ejecutarse en los servidores internos de la empresa utilizando *IIS*.

- **RNF8 – Pruebas de funcionamiento**

  Se deben realizar pruebas de funcionamiento para comprobar el correcto comportamiento de la plataforma en cuanto a vistas, flujos de acción y majeo de errores.

- **RNF8 – Documentación técnica**

  El sistema debe tener una documentación técnica que describa el desarrollo, pruebas realizadas y procesos relevantes.

2. Antecedentes y marco teórico

En la actualidad, el desarrollo tecnológico en las empresas es una tendencia creciente que busca mejorar la experiencia del cliente y optimizar los procesos internos. Sin embargo, si nos referimos particularmente en los parques industriales locales como el que forma parte la empresa Diserglass este tipo de inversiones todavía no están generalizadas, es por eso que el presente trabajo propone un avance significativo, posicionándose como una iniciativa pionera que podría escalar a otras empresas del mismo rubro.

Como se mencionó anteriormente, *Diserglass* no contaba con ningún sistema digital que permitiera a sus clientes realizar un seguimiento de sus pedidos. Estos procesos se realizaban de forma manual, por lo que este sistema no solo representa una mejora operativa, sino que también implica un cambio en la interacción con sus clientes.

En la planificación se optó por un enfoque técnico basado en tecnologías modernas, las cuales están ampliamente aceptadas por la comunidad del desarrollo de software. El frontend se desarrolla utilizando *React*, una biblioteca de *JavaScript* especializada para construir interfaces de usuario; complementada con *TypeScript*, que se basa en *JavaScript*, sin embargo, este ofrece sintaxis para tipos y puede captar errores, entre otras características. Para el diseño de la interfaz se utilizó *Tailwind CSS*, un *framework* de clases utilitarias que permite construir interfaces responsivas de forma ágil.

Además, el diseño de las interfaces fue prototipado en *Figma* y presentado al cliente para obtener una aprobación del mismo en una etapa temprana del proyecto. Por otra parte, la comunicación con el servidor se gestiona a través de *Axios*, una librería que se encarga de realizar solicitudes *Hypertext Transfer Protocol* (*HTTP)* de manera eficiente y estructura. Asimismo, el sistema de autenticación se implementa mediante *JWT* (*Json Web Token*), que permite validar sesiones de manera segura, sin necesidad de almacenar el estado del lado del servidor.

Para el manejo del estado global, se desarrolló utilizando *Redux Toolkit*, el cual es ideal para estados complejos que deben ser accesibles en toda la aplicación, por otra parte, se empleó *React Context* para estados locales o específicos de un componente, es decir, donde no se necesita de la complejidad de *Redux*.

Una parte fundamental del sistema es un módulo denominado **Sincronizador**, es que no se puede considerar una *API*, ya que no recibe solicitudes, sino un componente independiente desplegado en la infraestructura de producción de

la empresa. Su función es mantener actualizada la base de datos del sistema a través del consumo de la ***API***; es decir, el Sincronizador es el que se comunica con la *API* para mantenerla actualizada. Asimismo, el frontend se comunica con la *API* y la información limitada de la misma, lo que refuerza la seguridad de la base de datos de la empresa, manteniendo la información sensible fuera de exposición. En este esquema, el flujo de comunicación siempre es iniciado por el Sincronizador*,* limitando la *API* a únicamente recibir y responder.

En conclusión, las decisiones técnicas tomadas tuvieron un fuerte apoyo en la revisión de documentación oficial (*React*, *Redux Toolkit*, *JWT*, etc), en recursos promovidos por la comunidad en *GitHub* y *Stack Overflow,* y en el aprendizaje de cursos especializados ofrecidos en *Udemy*.

3. Planificación

3.1. Diseño de solución

En las primeras etapas, se evaluó una arquitectura donde la ***API*** se comunicaría directamente al sistema interno de Diserglass mediante un *socket*. Sin embargo, el área de seguridad informática de la empresa determinó que no era posible realizar esto debido a los riesgos asociados a abrir canales de comunicación directa entre el sistema interno y servicios expuestos a internet.

Entonces, surgió como respuesta de diseño un componente denominado **Sincronizador**, el cual su implementación se encuentra dentro de la infraestructura interna de la empresa, en su dominio privado. El Sincronizador es el único módulo autorizado para comunicarse de forma activa con la *API* pública, alojada en *IIS*. Este componente realiza consultas periódicas a la *API* sobre su última actualización para compararla con la del mismo, en el caso de que no coincidan el Sincronizador le envía la información actualizada a la *API*, esto pueden ser nuevas órdenes o cambios de estados en órdenes existentes. La información pasa a ser visible para los clientes desde el momento en que la base de datos de la *API* es actualizada, es decir, hay un lapso de tiempo entre que se genera una orden y puede visualizarse, esto se debe a que la frecuencia de sincronización de datos se ajustó de forma que no genere sobrecarga en los servidores de producción, considerando las restricciones de procesamiento y los volúmenes de datos involucrados.

Desde el punto de vista arquitectónico, el sistema se compone de tres módulos principales: ***Frontend***, ***API*** y **Sincronizador**.

En el presente informe se desarrolla en profundidad el *Frontend*, mientras que el *backend*, que incluye la *API* y el Sincronizador, será abordado con mayor detalle en la presentación de la Práctica Profesional Supervisada correspondiente a **Alan Marengo.**

No obstante, se incluirán las referencias necesarias para comprender la interacción entre los módulos y, en el caso del Sincronizador, se presenta un breve análisis de su implementación en *.NET*, destacando buenas prácticas de desacoplamiento y seguridad, así como su impacto en la arquitectura general del sistema.

3.1.1. Funcionalidades principales del sistema

El sistema fue diseñado para cubrir las necesidades de los clientes de *Diserglass* en la gestión y consultas de órdenes, garantizando seguridad, accesibilidad y usabilidad. Entre las funciones clave se encuentran:

- Registro de usuarios restringido a clientes existentes mediante validación de CUIT.
- Validar el correo electrónico mediante verificación *OTP* como capa adicional de seguridad.
- Iniciar sesión utilizando autenticación basada en *JWT,* sin almacenamiento innecesario de estado en el servidor.
- Visualizar el estado de sus órdenes en el último año, con opciones de filtrado por fecha, estado y antigüedad.
- Navegación intuitiva en cualquier dispositivo debido a un diseño responsivo y versiones específicas para *desktop, tablet* y *mobile.*
- Ver la composición de órdenes seleccionada, de manera intuitiva y dinámica.
- Protección de rutas críticas para asegurar que solo usuarios autenticados accedan a la información sensible.

Este conjunto de funcionalidades responde a criterios de eficiencia como a la experiencia de usuarios.

3.1.2. Hipótesis y supuestos

- Los clientes deben contar con conexión a internet para poder ingresar y visualizar la información.
- Los usuarios utilizarán dispositivos compatibles con los estándares modernos (*React*, *TypeScript* y *TailwindCSS*), sin requerir tecnologías obsoletas.
- El Sincronizador debe operar en condiciones óptimas dentro de la red interna de *Diserglass*, es decir, debe respetar las ventanas de mantenimiento y los procesos internos de copia de seguridad (*backup*).

3.1.3. Dificultades previstas y alternativas de solución

- **Resistencia a la digitalización en el sector industrial:** Se diseñaron interfaces amigables, responsivas y altamente optimizadas para que puedan ser utilizados incluso en celulares de baja gama, sin penalizar la experiencia de usuarios.
- **Validación de identidad de usuarios externos:** Se incorporó un validador de CUIT antes del registro, que permite identificar clientes de la empresa, como mecanismo adicional de seguridad.

- **Desconocimiento técnico de los usuarios:** La aplicación evita sobrecarga de librerías y prioriza la compatibilidad, asegurando que funcione correctamente en la mayoría de los dispositivos y navegadores.

La solución adoptada refleja el equilibrio entre los requerimientos técnicos, las buenas prácticas y las necesidades particulares de Diserglass, además, sienta las bases para la implementación del módulo para la cotización de pedidos.

3.1.4. Frontend

Tecnologías utilizadas

El ***Frontend*** se desarrolló utilizando *React con TypeScript* y *Tailwind CSS* como tecnologías principales, priorizando la flexibilidad y el rendimiento de una *Single Page Application (SPA).* La elección de estas herramientas fue debido a la necesidad de un desarrollo ágil, con tipado fuerte para reducir errores y estilos altamente personalizables para mantener coherencia visual en toda la aplicación.

Por otra parte, se adoptó una arquitectura modular inspirada en *Clean Architecture,* asegurando la separación de responsabilidades y facilitando la escalabilidad futura.

En cuanto al manejo del estado global, se optó por una combinación entre *Redux Toolkit* y *Context API*, seleccionando uno según el alcance y persistencia requerida para cada dato.

Por último, para la comunicación con la *API* se empleó *Axios**,*** por su capacidad de estandarizar peticiones, manejar errores y soportar operaciones como cancelación o *timeout.*

Diseño responsivo y adaptabilidad

Se definió un enfoque completamente responsivo para garantizar una experiencia de usuario óptima en *desktop, tablet y mobile.*

La decisión de rediseñar para cada tipo de dispositivo, en lugar de solamente ajustar el tamaño, se basó en el hecho de que, se podría suponer que la mayoría de los clientes operaban desde computadoras de escritorio, sin embargo, no existía certeza de que no accedieran desde *tablets* o *mobiles*.

Este rediseño permite que, en pantallas grandes, los datos se muestren de forma amplia y detallada, mientras que en dispositivos móviles se adopte una estructura compacta y visualmente jerarquizada, que favorece la navegación y la accesibilidad.

Paginación de datos

Se implementó un sistema de paginación tras un análisis funcional y pruebas de rendimiento, en el cual se determinó la cantidad de órdenes a mostrar según el tipo de dispositivo.

El *Frontend* es el encargado de solicitar la cantidad de órdenes necesarias y el número de página, enviando esta información como metadatos en el encabezado *HTTP*. Esta metodología mejora la performance general al evitar la carga innecesaria de datos.

SEO y alcance público

No se consideró optimizar la plataforma para motores de búsqueda (*SEO*, por sus siglas en inglés Search Engine Optimization), dado que el sistema no está orientado al público general, sino exclusivamente a clientes existentes de Diserglass.

El registro está restringido mediante una validación de CUIT, lo que garantiza que solo clientes existentes en la base de datos interna puedan ingresar. Sin embargo, la aplicación cuenta con enlaces en el pie de página a las redes sociales de la empresa y un sitio institucional externo con *SEO* propio de la empresa.

Dado que los datos manejados en esta plataforma están vinculados a gestiones internas de clientes y órdenes, no existe un beneficio directo en *indexarlos* para buscadores. El enfoque de visibilidad orgánica de la marca se mantiene en el sitio principal de *Diserglass,* donde se presenta la empresa, sus servicios y su propuesta comercial.

Seguridad de la información

La plataforma fue diseñada con un enfoque de seguridad por capas, integrando medidas tanto en *frontend* como en *backend* para proteger los datos sensibles de clientes y sus órdenes.

El acceso solo se permite a clientes existentes de *Diserglass* mediante la validación previa de su CUIT, luego se añade otra capa de autenticación mediante la verificación *OTP.*

Se utilizó *Json Web Token (JWT)* para la autorización y persistencia de sesión, lo que evita el almacenamiento innecesario de estado en el servidor y garantiza un control granular sobre el acceso a recursos.

Por otra parte, se consideró la protección de rutas críticas como una medida necesaria de seguridad, mientras que prácticas adicionales como la limpieza de *inputs* y la desactivación de autocompletado en formularios sensibles fueron

evaluadas, pero no implementadas en esta primera fase, al no manejar datos que lo requieran de manera urgente. Sin embargo, se proyecta su incorporación en futuras etapas, especialmente con la funcionalidad de cotización de pedidos, donde la sensibilidad de la información será mayor.

Validación de datos y manejo de errores

Las validaciones y manejo de errores se implementaron en conjunto con la ***API***. La *API* se encarga de validar la información relacionada a la base de datos y responde con un formato estándar *{ Property: X, Error: Y }.*

Por su parte, el frontend realiza validaciones de estructura (tipo, formato, longitud) antes de enviar los datos, además, gestiona los errores derivados de conectividad o funcionamiento de la *API*.

3.1.5. Sincronizador

El ***Sincronizador*** desempeña un rol fundamental en la arquitectura del sistema, su principal función es mantener actualizada la base de datos de la *API* sin comprometer la seguridad interna de la empresa. Esto es debido a su diseño desacoplado, el cual permite que no haya una dependencia directa entre los sistemas, promoviendo así una arquitectura robusta y escalable.

3.2. Plan de ejecución del proyecto

Planificación y Organización

Duración: 3 semanas.

- Reuniones iniciales y armado de requerimientos.
- Análisis funcional y técnico.
- Creación del repositorio y configuraciones del entorno.
- Planificación ágil con Sprint Planning.
- Uso de Jira y organización modular de tareas (Frontend, Backend, Sincronizador, Testing).

Resultado esperado

Documento de objetivos y requerimientos funcionales, documento de diseño general y descripción funcional del sistema, *stack tecnológico definido, enumeración de tareas y asignación de responsabilidades.*

Responsables

Alan Marengo y Leonel Rebollini.

Diseño y Estructura del Proyecto

Duración: 2 semanas.

- Aplicación de *Clean Architecture* (estructura de carpetas, separación en capas).
- Definición del flujo de navegación y experiencia del usuario (*UX*)
- Diseño de vistas en *Figma* (pantallas para *desktop, tablet y mobile* de *login*, recuperación, órdenes, *dashboard*).
- Diseño de la base de datos y análisis de la base de datos local de Diserglass.
- Definición de *endpoints* de la API REST.

Resultado esperado

Estructura inicial del proyecto definida, prototipo de vistas diseñadas, *endpoints* documentados y base de datos diseñada.

Responsables

Alan Marengo y Leonel Rebollini.

Desarrollo del Frontend

Duración: 5 semanas.

- Desarrollo de componentes reutilizables en React.
- Implementación de vistas (flujo de *login*, ordenes...).
- Integración con backend mediante Axios.
- Aplicación de diseño responsivo con TailwindCSS.
- Rediseño de la vista mobile para mejorar la experiencia de usuario.

Resultado esperado

Rutas definidas, estilos aplicados, vistas desarrolladas para *desktop/tablets/mobile* y comunicación con el *backend* preparada.

Responsables

Leonel Rebollini.

Desarrollo de la API y Sincronizador

Duración: 5 semanas.

- *Desarrollo de la API REST con autenticación y endpoints protegidos.*
- Creación del módulo *Sincronizador* en *.NET.*
- Pruebas locales de comunicación *Sincronizador ⇄ API.*

Resultado esperado

API funcional y disponible para consumir con el frontend, pruebas de conexión completadas.

Responsables

Alan Marengo

Pruebas, Auditoría y Seguridad

Duración: 3 semanas.

- Pruebas de integración (*Frontend ⇄ Backend)*
- Validación del *Sincronizador* y manejo de errores.
- Integración de *Firebase* para auditoría.
- Registro de logs, auditoría de eventos e inconsistencias.
- Revisión de seguridad (tokens, validaciones, rate limit).

Resultado esperado

Sistema completo funcionando correctamente en entorno local, sistema de auditoría implementado. Listo para entregar.

Responsables

Alan Marengo y Leonel Rebollini.

Despliegue y Puesta en Producción

Duración: 4 semanas.

- Configuración en servidor de IIS y dominio.
- Configuración de DNS privado de la empresa.
- Pruebas de funcionamiento y performance.
- *Feedback* del cliente y ajustes finales.

Resultado esperado

Plataforma web operando correctamente, accesible para los usuarios finales.

Responsables

Alan Marengo y Leonel Rebollini.

Documentación y Manual de Usuarios

Duración: 1 semana.

- Documentación técnica del sistema, redactada y estandarizada en *Confluence*.
- Manual de usuarios.

Resultado esperado

Informe completo y manual de usuarios, entregados.

Responsables

Alan Marengo y Leonel Rebollini.

4. Desarrollo

4.1. Introducción al desarrollo

En esta sección se describe el proceso de implementación de la solución propuesta para la empresa de Diserglass, se optó por dividir el trabajo, en donde el desarrollo backend fue asumido por **Alan Marengo**, mientras que el desarrollo frontend fue abordado por mí. La metodología de trabajo fue iterativa, permitiendo planificar, ajustar y validar funcionalidades a medida que avanzaba el desarrollo.

4.2. Organización del trabajo

La organización de trabajo se realizó utilizando un enfoque iterativo por medio de *Jira*, se aplicó la metodología *Scrum* en donde se plantearon objetivos semanales y se utilizó *GitHub* para la gestión de versiones y seguimiento de tareas. Además, la vinculación entre *Jira y GitHub* permitió automatizar el ciclo de desarrollo.

Cada tarea en Jira incluía:

- Un **identificador único** con prefijo y número secuencial (DW-01, DW-02…).
- Una **etiqueta** que indica su categoría: *UI/UX*, *Testing*, *Frontend*, *Backend Nube*, *Backend Sincronizador*.
- **Responsables** de llevar a cabo la tarea.

El flujo de trabajo se estableció de la siguiente manera:

- **Inicio de tarea***:* desde Jira se generaba automáticamente una rama en GitHub siguiendo el formato DW-XX-descripción-tarea. La tarea cambia de estado de *“Tareas por hacer”* a *“En curso”*.
- **Desarrollo**: el trabajo se realizaba en *Visual Studio* o *Visual Studio Code* sobre la rama asignada.
- **Integración y cierre**: al crear un *Pull Request* vinculado a la tarea, *Jira* actualizaba su estado a *“Finalizada”*.
- **Cierre de tareas padre**: cuando todas las subtareas de una tarea principal se completan, *Jira* marca automáticamente la tarea padre como *“Finalizada”*.

4.3. Arquitectura del Frontend y diseño de datos

4.3.1. Estructura del Frontend

El proyecto ***Frontend*** adopta *Clean Architecture*, donde se organiza en carpetas modulares que agrupan responsabilidades comunes. Esta organización brinda escalabilidad, mantenimiento, reutilización de componentes y lógica compartida.

Carpetas principales

- *adapters:* inicialmente planeado para mapear o transformar datos entre *backend* y *frontend*, aunque terminó quedando sin uso hasta el momento.
- *assets:* contiene recursos estáticos como imágenes, íconos, etc.
- *components:* incluye componentes reutilizables globalmente como *Footers, Input, Label.*
- *contexts:* define los contextos utilizados durante el flujo de registro, entre otros.
- *hooks:* almacena hooks reutilizables.
- *interceptors:* configuración de interceptores para *Axios* (por ejemplo, para agregar el *token* a los *headers*).
- *models:* define los tipos y modelos de datos utilizados tanto para el formulario, como para respuestas del *backend*, *tokens*, errores, etc.
- *pages:* cada subcarpeta representa una página o vista del sistema. Cada una tiene su propia estructura:
    - *Adapters*
    - *Components*
    - *Hooks*
    - *Interceptors*
    - *Models*
    - *Redux*
    - *Styles-components*
    - *Utilities*
- *redux:* contiene los *slices* de estado global, como *authSlice*, etc.
- *services:* funciones que manejan las llamadas a la ***API,*** por ejemplo, *AuthRegisterService.*
- *styled-components:* se inicializó *Tailwind CSS* y se definieron estilos globales para cuando no se usa *Tailwind CSS* directamente.
- *utilities:* funciones auxiliares utilizadas en todo el proyecto, por ejemplo, *handleApiError.*

4.3.2. Modelado de datos

Durante el desarrollo del Frontend se definieron distintas interfaces en *TypeScript* ubicadas en la carpeta "*models*", que permitieron estructurar la información de forma clara y tipada, facilitando el desarrollo seguro y legible.

Estas interfaces se pueden agrupar en tres grandes categorías:

Modelos de dominio

Representan datos reales del negocio, generalmente obtenidos desde la *API*.

- *OrderType*, *OrderComposition*, *OrdersState*: representan el detalle de una orden, su composición (material – dimensión - cantidad) y el estado del pedido, respectivamente.

  Hooks relacionados

  - *useFetchOrders:* se encarga de armar los parámetros para consultar a la *API*.
  - *useFiltersOrders:* realiza la aplicación de filtros en la visualización de órdenes.
  - *usePaginatedOrders:* tiene la responsabilidad de realizar los cambios de página, así como el nombramiento de la *URL.*
  - *useOrders:* maneja los cambios de estados que pueden expandir/contraer una orden.

  Estado global *(Redux)*

  - *OrdersSlice:* almacena el estado global de las órdenes, incluyendo la lista de órdenes recuperadas, los metadatos como la página actual, total de páginas y filtros aplicados, y gestiona el estado de la interfaz de usuario (*UI*) como *loading, error,* etc.
  - *OrdersThunks:* se encargan de llamar a la *API* y actualizar el estado global.
- *AuthState*: contiene el estado de autenticación del usuario, como el *token*, email, nombre completo, identificación y nombre de la compañía.

  Hooks relacionados

  - *useAuth*: en caso de login exitoso, se encarga del almacenamiento de los datos en *localStorage*, y ejecuta *dispatch(loginSucess)* para actualizar el estado global.
  - *useUser*: es el responsable del proceso de logout, eliminando los datos del *localStorage* y actualizando el estado global mediante *dispatch(logout)*.

Estado global *(Redux)*

- *AuthSlice:* define el estado *AuthState* como estructura principal. Utiliza *Redux Toolkit* para centralizar los datos del usuario autenticado, donde su función es almacenar y actualizar el estado de autenticación durante el flujo de navegación.

Estado global *(Context API)*

- *AuthContext:* complementa a *Redux* en las operaciones de autenticación, gestionando el estado en pasos específicos del flujo de *login*, como el registro, la recuperación de contraseña y validaciones de formularios.

Modelos de formularios

Intervienen en los procesos de registro o *login*.

FormValues, *RegisterData*: estructuran los datos cargados por el usuario.

- *RegisterData* representa la versión final que se envía a la *API, la cual contiene nombre, contraseña, email e identificación*.
- *FormValues utiliza* campos opcionales (útiles para validación progresiva), además, incluye la confirmación de la contraseña.

Hooks relacionados

- *useFlowLoginForm:* inicializa los estados generales para el flujo de *login* o registro, incluyendo los valores de los campos del formulario y el estado de errores.

Estado global *(Redux)*

- *AuthSlice:* recibe los datos de registro construidos a partir de *RegisterData*, utilizando la estructura de *AuthState, durante los procesos de loginSucess* y *logout*.

Estado global *(Context API)*

- Se utiliza para persistir los datos parciales del formulario a lo largo del flujo de registro, recuperación o cambio de contraseña. Es decir, *RegisterData* es almacenado en este contexto para que todas las páginas del proceso puedan acceder y modificar su estado.

- *ValidationOptions*: contiene los estados de validación de campos (nombre, email, password, etc.) que permiten una retroalimentación clara al usuario.

Modelos de respuesta y errores del backend

Permiten interpretar y adaptar las respuestas de la *API* al *Frontend.*

- *BackendError*, *RegisterResponse*, *DecodedToken*: facilitan el manejo de errores específicos del servidor, lectura de *tokens* y el manejo de respuestas exitosas o con validaciones fallidas.

  Hooks relacionados

  - *useAuthGuard*: se utiliza *DecodedToken* para verificar la validez del token mediante la función *getTokenExpirationTime()*. Si detecta que el token ha expirado, dispara el *logout* y redirecciona al usuario al *login*.
  - Los formularios implementan sus propios *hooks*, como *useLoginForm* o *useRegisterForm*, estos se encargan de hacer los llamados a los servicios que consumen los endpoints.

  Estado global *(Redux)*

  - *OrdersSlice:* incluye una estructura para almacenar errores para casos en los que se requiera mostrar errores relacionados con las órdenes.

  Estado global *(Context API)*

  - *AuthContext:* cumple un rol importante en el manejo de errores durante el flujo de *login*, registro y recuperación. Este mantiene el estado de los errores durante el transcurso de páginas, a través de objetos como *formsErrors* y *setFormErrors* para la actualización.

4.3.3. Páginas

Cada carpeta de página representa una vista específica y en cápsula su propia lógica y componentes asociados. Todas siguen la misma estructura, implementando subcarpetas como *components, hooks, services, etc,* según va siendo necesario.

A continuación, se describen brevemente las páginas desarrolladas:

- *Welcome:* página de presentación diseñada para *móvil* o *tablet vertical*. Sirve como pantalla de inicio, y permite al usuario ingresar directamente al menú de iniciar sesión o al flujo de registro.
- *CuitVerification:* página que permite verificar si el CUIT del usuario pertenece a los clientes de *Diserglass*, primer paso del registro. Encapsula la lógica previa al registro formal.

- *Register:* maneja el formulario de registro. Interactúa con *AuthContext* para manejar las validaciones, errores y acciones, además, obtiene el CUIT cargado en *CuitVerification*, Utiliza *handleApiError* desde *utilities*, y guarda errores del formulario en el contexto.
- *Login:* vista de *login* con sus propios *hooks* y servicios. Utiliza el contexto de autenticación para iniciar sesión y mostrar errores en tiempo real.
- *OTPVerification:* página donde el usuario ingresa el código OTP. Centraliza tanto la vista como la lógica de verificación y reenvío.
- *PasswordRecovery:* agrupa tres vistas relacionadas con recuperación de contraseña:
    - *ForgotPassword:* formulario para solicitar recuperación de cuenta mediante email.
    - *ResetPassword:* ingresar nueva contraseña.
    - *PasswordChanged:* confirmación de contraseña cambiada correctamente.

  Esta división permite un control claro del flujo de recuperación.
- *Orders*: vista principal para la visualización de órdenes. Incluye componentes visuales como tablas o tarjetas, que son exclusivos de esta página.

4.4. Implementación

4.4.1. Desarrollo de la API

Introducción breve

Si bien es cierto que el *backend* será abordado en profundidad por **Alan Marengo**, se incluye a continuación una introducción general a los aspectos más relevantes del *backend*, ya que su funcionamiento es esencial para el correcto desarrolló del ***Frontend*** y la lógica de negocio del sistema.

Tecnologías utilizadas

La ***API*** fue desarrollada bajo el enfoque de una *Web API* en *.NET 8*, aplicando los principios de *Clean Architecture* con una estructura basada en cinco capas. Se utilizaron diversas bibliotecas y herramientas para facilitar la modularidad, validación y consultas:

- *Entity Framework Core:* para operaciones *CRUD* generales.
- *Dapper:* en caso de consultas complejas o con filtros.
- *MediatR:* para implementar el patrón *CQRS* y manejar *comandos/queries* desacoplados.
- *FluentValidation:* para validar la entrada de datos de forma estructurada.

- *AutoMapper:* para mapear entre entidades del dominio y *Data Transfer Objects (DTOs).*
- *MinimalAPI:* para simplificar la exposición de endpoints.

Diseño de endpoints

Los endpoints siguen una convención estandarizada basada en prefijos claros:

- `/api/v1` → versión de la *API*.
- `{grupo}` → módulo funcional (ejemplo: orden, usuario).
- `{caso-de-uso}` → acción o recurso específico (ej. crear, detalle/{id}).

Ejemplo completo:

`/api/v1/order/GetOrders`

La API tiene tres grupos principales de endpoints:

- *Users*: autenticación y gestión de usuarios.
- *Orders*: visualización de órdenes asociadas al cliente autenticado.
- *Synchronization*: recepción de datos por parte del sincronizador interno.

Interacción con el Frontend

El Frontend consume esta *API* para manejar todo el flujo de autenticación y consulta de órdenes. Las respuestas siguen un estándar basado en convenciones propuestas por **Robert C. Martin**, definido:

- En caso exitoso, se devuelve la información requerida de manera directa.
- En caso de error, se utiliza el formato *{ Property: , Errors: }*, para mantener claridad y trazabilidad de errores.

Seguridad y autenticación

El sistema emplea *JWT* como mecanismo de autenticación. Una vez autenticado, el usuario recibe un token que contiene:

- *token*: cadena codificada de autorización.
- *email*: correo del usuario.
- *fullName*: nombre completo del usuario.
- *identification*: CUIT del usuario.
- *companyName*: nombre de la empresa asociada.

El token tiene una validez extendida. Además, no se implementó un sistema de *refresh tokens* debido a que la naturaleza del sistema es de acceso eventual, no crítico.

Observaciones finales

La *API* fue diseñada desde cero para cumplir de la mejor manera las necesidades del cliente, priorizando una estructura limpia, escalable y segura. Se encuentra desplegada en *IIS* bajo el dominio de *Diserglass,* por otra parte, su implementación fue acompañada de reuniones funcionales frecuentes para definir claramente cada caso de uso.

4.4.2. Desarrollo del Sincronizador

Introducción

El **Sincronizador** es un s*chedule component* interno que actúa como puente entre los sistemas de gestión de *Diserglass* y la ***API***. Su función principal es, mediante ejecuciones programadas, iniciar la comunicación con la *API* para consultar y detectar cambios en los datos, en el caso de haber, enviar las actualizaciones correspondientes. Esto garantiza que la información de la plataforma web esté actualizada varias veces al día.

Por solicitud del cliente, no se implementó como un ejecutable independiente, sino que se encuentra desplegado bajo el mismo entorno *IIS* que la *API*, lo que simplificó su administración, además, permitió reforzar su seguridad como por ejemplo con certificados *SSL*.

Tecnologías utilizadas

El Sincronizador está desarrollado en *.NET 8*, utilizando la misma base tecnológica y arquitectónica, y aplicando los mismos estándares que la *API,* aunque con un alcance más reducido debido a su naturaleza de servicio interno. Las herramientas y librerías aplicadas fueron*:*

- *Dapper:* para consultas rápidas y eficientes a la base de datos interna.
- *Entity Framework Core*: para la construcción de tablas necesarias y gestión de las mismas.
- *Firestore:* para el registro centralizado de auditoría de eventos.

A diferencia de la *API,* el Sincronizador no expone *endpoints* ni implementa patrones como *CQRS* o *MediatR,* ya que su lógica se centra en operaciones internas y controladas.

Seguridad

El canal de comunicación entre el Sincronizador y la *API* está protegido mediante *API Key*, donde se valida cada solicitud.

Si se considera que el sincronizador opera dentro de un entorno controlado y no interactúa con usuarios externos, podemos discernir que este método es suficiente.

Flujo de funcionamiento

1. *Inicio de proceso*: el Sincronizador ejecuta una tarea programada (s*chedule*) que inicia la comunicación con la *API.*
2. *Obtención de referencias:* la *API* devuelve un listado con la última actualización de cada cliente.
3. *Comparación de datos*: el Sincronizador compara esta información con la base de datos interna.
4. *Detección de cambios*:
    - Si existen órdenes nuevas o cambios de estado, las prepara para ser enviadas.
    - Si no hay actualizaciones, se registra el evento “sin cambios” en la auditoría.
5. *Envío de datos*: las órdenes nuevas y actualizadas se transmiten a la *API* en llamadas separadas.
6. *Confirmación y cierre:* una vez procesada la información por la *API,* se registra el resultado en el sistema de auditoría.

Aunque la comunicación es bidireccional, la *API* solo responde. El control y la iniciación siempre corresponden al sincronizador.

Observaciones finales

El Sincronizador es una pieza fundamental en el flujo de datos entre los sistemas internos y la *API.* Su diseño simple y orientado a tareas programas asegura estabilidad y seguridad. Además, permite una trazabilidad total, sin requerir intervención manual.

4.4.3.Sistema de auditoría

El sistema de auditoría fue implementado de forma centralizada para abarcar tanto la ***API*** como el ***Sincronizador***, garantizando una trazabilidad completa de los procesos ejecutados.

Para ello, se integró por medio de *Firebase* como repositorio de registros, lo que permite conservar un historial cronológico detallado de cada operación realizada.

Cada evento es registrado con un código estandarizado, compuesto por la fecha, hora y un identificador de la operación.

Por ejemplo `2025-02-28_17-C1`, donde:

- `2025-02-28` → Fecha de registro.
- `17` → Hora (en formato de 24 horas).
- `C1` → Código de operación.

La codificación se definió según el tipo de operación:

- **Clientes**
  - *C1*: Inicio de sincronización de clientes.
  - *C2*: Respuesta de la *API.*
  - *C3*: Finalización de la sincronización de clientes.
- **Órdenes nuevas**
  - *O1*: Inicio de inserción de órdenes.
  - *O2*: Respuesta de la *API.*
  - *O3*: Finalización de la inserción de órdenes.
- **Órdenes actualizadas**
  - *OU1*: Inicio de actualización de órdenes.
  - *OU2*: Respuesta de la *API.*
  - *OU3*: Finalización de la actualización.

El sistema también registra los eventos cruzados entre la *API* y el *Sincronizador*, de modo que exista una trazabilidad de qué se envió, cuándo se envió y cómo fue procesado. Esto permite, en casos de inconsistencias, encontrar rápidamente el origen del problema.

4.4.4. Desarrollo del Frontend

Introducción breve

El desarrollo del ***Frontend*** tuvo como objetivo proporcionar una interfaz intuitiva, responsiva y eficiente, la cual está optimizada para su uso tanto en dispositivos móviles como de escritorio.

Este módulo se encarga de recibir las interacciones del usuario, realizar validaciones iniciales y comunicarse con la ***API***. El Frontend fue diseñado para garantizar una experiencia fluida, aplicando prácticas de modularidad y separación de responsabilidades, de esta manera el código resultante será escalable, mantenible y adaptable a futuras evoluciones del sistema.

La relación con la *API* se basa en el consumo de *endpoints* mediante solicitudes *HTTP*, sumando manejo centralizado de autenticación, control de errores y estados globales, para asegurar una consistencia en el flujo de datos y minimizar las solicitudes.

Tecnologías utilizadas

El desarrolló se realizó con un conjunto de tecnologías y herramientas seleccionadas por su rendimiento, respaldo en el sector y compatibilidad con los requerimientos del proyecto:

- *React 18*: *Framework* de *JavaScript* para la construcción de interfaces de usuarios basadas en componentes reutilizables. Ofrece un flujo de datos unidireccional y un alto rendimiento en renderizado.
- *Vite*: Herramienta moderna de *build* y servidor de desarrollo, optimizada para proyectos con *React, Tailwind CSS y TypeScript.* Permite integración fluida con librerías y herramientas del ecosistema, configuración mínima y soporte nativo para módulos *ECMAScript (ESM)*, además de proporcionar recarga en caliente (*Hot Module Replacement*) para una experiencia de desarrollo ágil.
- *TypeScript*: *Superset* tipado estáticamente de JavaScript, que añade tipado fuerte y permite la detección temprana de errores, lo que facilita la escalabilidad y el mantenimiento del código.
- *Tailwind CSS*: Framework de estilos *utility-first* que facilita la creación de interfaces responsivas y coherentes, evitando la sobrecarga de *CSS* personalizado.
- *Redux Toolkit* y *Context API*: Combinación empleada para el manejo de estado global. *Context API* se utiliza para estados acotados a un módulo o vista específica, mientras que *Redux Toolkit* gestiona estados compartidos entre múltiples vistas, como el estado de autenticación o la lista de órdenes.
- *Axios:* Cliente *HTTP* que simplifica la comunicación con la *API*, permitiendo interceptores para añadir encabezados de autenticación, manejo unificado de errores y cancelación de solicitudes.
- *ESLint*: Herramienta de análisis estático para mantener la consistencia y calidad en el desarrollo.
- *React Router*: Librería para manejo de rutas y navegación entre vistas de manera declarativa

Estructura y organización del proyecto

La estructura de carpetas y módulos del Frontend sigue el esquema descrito en el apartado de *Arquitectura del Frontend y diseño de datos*, brindando modularidad y separación de responsabilidades, propias de *Clean Architecture.*

Esta organización garantiza escalabilidad, mantenibilidad y reutilización de tanto componentes como lógica en todo el sistema.

Interacción con el backend

Resumen funcional

El Frontend actúa como capa de presentación, valida datos de entrada, presenta la *UI* y consume los servicios expuestos por la *API*. La comunicación

se realiza mediante llamadas *HTTP* por medio de *Axios* a *endpoints* con la convención:

`/api/v1/{grupo}/{caso-de-uso}`

Los módulos principales con los que interactúa el Frontend son:

- `/api/v1/order` → gestión y consulta de órdenes.
- `/api/v1/user` → autenticación y operaciones relacionadas con el usuario.

Ejemplo relevante de caso de uso para la consulta de órdenes

GetOrders

Endpoint completo*:*

`GET /api/v1/order/GetOrders`

Parámetros de consulta*:*

- *PageSize*: tamaño de página.
- *PageNumber*: número de página.
- *Date*: filtro por fecha.
- *OrderStatus*: filtro por estado de la orden.
- *OrderSort*: criterio de ordenamiento, ascendente o descendente.
- *LegacyCode*: código del cliente.

Requiere *token JWT* en el encabezado de la solicitud se agrega:

`Authorization: Bearer <token>`

Respuesta: listado de órdenes, junto con metadatos de paginación.

Cliente HTTP y organización de llamadas

Se utiliza *Axios* directamente en cada servicio (por ejemplo, *OrdersService, AuthService*) para construir y enviar las solicitudes *HTTP*.

Actualmente, se cuenta con un interceptor global implementado para manejar códigos de estado y mensajes de error. Los interceptores para autorización y otros aspectos comunes están desarrollados, pero no han sido integrados todavía, ya que por el momento solo existe un servicio que requiere autenticación.

En futuras iteraciones, cuando se incorporen más servicios que requieran autenticación, se planea migrar todas las peticiones hacía una instancia de *Axios* centralizada con dichos interceptores, con el objetivo de:

- Reducir duplicación de código.
- Unificar el manejo de autenticación.
- Mejorar la mantenibilidad del proyecto.

Manejo de errores y adaptación de respuestas

El tratamiento de errores sigue el formato y criterios descritos en el apartado de *Validación de datos y manejo de errores.*

En este contexto específico, el Frontend adapta las respuestas para integrarlas a la *UI*, por ejemplo:

- Asociando *Property* al campo correspondiente en los formularios.
- Mostrando *Error* como mensaje contextual.
- Traduciendo errores genéricos de red a mensajes comprensibles para el usuario final.

Se destaca nuevamente que, el *Frontend* se encarga de las validaciones de estructura (tipo, formato, longitud) antes de enviar los datos.

Estado y flujo de datos

El *Frontend* utiliza dos herramientas para gestionar el estado compartido, por una parte, *Redux Toolkit* para estados globales y *Context API* para estados específicos de flujos más locales.

Redux Toolkit

Se implementa con un *store* que agrupa dos *slices* principales:

- authSlice
  - El estado inicial contiene *token*, *email*, *fullName*, *identification*, *companyName* y *isAuthenticated*.
  - *Reducers:*
    - *loginSucess:* carga el estado con los datos de autenticación.
    - *logout:* limpia el estado y elimina los datos del localStorage.
- *ordersSlice*
  - El estado inicial contiene las órdenes, estados de procesos como cargando, error, orden expandida, fecha, filtros, ordenamiento, página actual, ítems por página y páginas totales.
  - *Reducers: setCurrentPage, setSizePage, toggleOrderDetail, setFilters.*
  - *Extra reducers:* manejan las fases *pending, fulfilled, rejected* de *fetchOrders*, actualizando el estado según la respuesta de la *API.*

Context API

Se utiliza para mantener el estado en procesos que requieren persistencia temporal durante formularios, es decir, de manera más local.

- *AuthContext*
  - Gestiona el flujo de registro multi-paso, incluyendo verificación de email y CUIT.
  - Mantiene variables como *identification*, *otp*, *email*, *formErrors*, *registerData*, *isLoading*, entre otras, junto a sus *setters.*
  - Durante el registro, persisten temporalmente los datos en *sessionStorage* para que sobrevivan a recargas de página.
- *ResendTimerContext*
  - Actúa únicamente en el componente *OTPVerification*.
  - Controla el temporizador para reenviar el código *OTP,* manteniendo estados como *timeleft*, *showResendButton* y funciones como *resetTimer* que permite los reenvíos de códigos al email.

Seguridad y manejo de autenticación

El sistema utiliza *JWT* (*Json Web Token*) para la autenticación y autorización. Tras su inicio de sesión exitoso, la *API* genera un *token* que se almacena en el *localStorage* del cliente junto con sus datos. Esta elección se tomó de forma deliberada porque simplifica la implementación y el manejo de autenticación en una *SPA*, evitando la configuración de *cookies* y problemas asociados a *CORS* con credenciales, además de que el nivel de información al que otorga acceso es limitado (por el momento, solo permite la visualización de órdenes).

Por otra parte, el riesgo de robo de *token* mediante *XSS* se considera bajo en el contexto actual, y mitigable con buenas prácticas de sanitización y control de dependencias. Cabe destacar que no se utilizan *refresh tokens*, solo *tokens* con tiempo de expiración extendidos.

Se consideran medidas cuando aumente la sensibilidad de la información manejada, entre ellas están:

- Validaciones y sanitización más estrictas.
- Políticas de seguridad de contenido (CSP) configuradas desde la *API*.
- Restricción de ejecución de *scripts* no autorizados.

El flujo de autenticación incluye:

- **Login exitoso**: El usuario envía sus credenciales, la *API* responde con un *JWT* y datos del perfil.

- **Persistencia de sesión**: El *token* y los datos se guardan en *localStorage* y en el estado global (*authSlice*).
- **Logout**: Se limpia el estado global y el almacenamiento local, cerrando la sesión.
- **Protección de rutas**: El acceso a vistas privadas depende de *isAuthenticated*.

Manejo de expiración del token

El cliente cuenta con métodos y *hooks* especializados para garantizar que el *token* esté vigente antes de permitir acceso:

- *isAuthenticated*: retorna un booleano según el estado de autenticación y la vigencia del *token.*
- *getTokenExpirationTime*: obtiene la marca de tiempo de expiración del *JWT*.
- *useAuthGuard*: verifica la autenticación y expiración, si el *token* caducó, redirige al *login*, si sigue vigente, permite acceso.
- *useRedictIfAuthenticated*: se utiliza en la pantalla inicial, si está autenticado, redirige automáticamente a la página de órdenes.

Este enfoque permite que la espiración se gestione del lado del cliente, sin dependencia de la respuesta de la *API*, brindando una mejor experiencia de usuario.

Optimización y rendimiento

Se han aplicado diversas estrategias con el fin de optimizar el sistema. Entre ellas, podemos destacar la optimización de la interacción con la *API* para evitar recargas o peticiones innecesarias a través del uso de *useDebounce,* un *hook* que retrasa la actualización de un valor evitando así que se generen múltiples solicitudes al *backend* por cambios rápidos de filtros o campos de búsqueda. Este patrón se combina con la variable de inicialización de *useResponsivePagination*, que ajusta la paginación de forma adaptativa según el tamaño de la pantalla, asegurando que un cambio de resolución no realice varias llamadas consecutivas a la *API*.

En cuanto a la navegación, se implementaron hooks centralizados para mejorar la mantenibilidad y reducir redundancia:

- *useAppNavigation*: encapsula el uso de *useNavigate* y expone funciones (*goTo*…) para dirigir a cualquier página definida en el flujo general, incluyendo la conducción a *OTPVerification* con el estado necesario para diferenciar si el proceso es de registro o recuperación de contraseña.

- *usePasswordRecoveryNavigation*: replica este patrón dentro del módulo de recuperación de contraseña, asegurando que la navegación de este mantenga el contexto adecuado (por ejemplo, ir a *OPTVerification* con el estado "*passwordRecovery*" para definir el *endpoint* correspondiente).

La responsividad también dispara *hooks* específicos:

- *useResponsiveRedirect* determina si la pantalla es ancha o estrecha, en el caso de ser estrecha muestra la vista de *Welcome* al iniciar, sino redirige directamente a *Login.*
- *useResponsivePagination* detecta el tipo de dispositivo (móvil, tablet, pantalla de escritorio pequeña o grande) y ajusta dinámicamente la cantidad de elementos por página, reduciendo recursos innecesarios y optimizando la experiencia de usuario. También cumple con la función de determinar si es adecuado mostrar las órdenes en lista o tarjetas.

Rutas protegidas y control de navegación

El enrutamiento del Frontend está organizado con *React-router-dom* para prevenir accesos no autorizados y manejar la navegación de forma consistente. Se utiliza un componente *PrivateRoute* que envuelve las rutas restringidas (por ejemplo, */Orders*), validando que el usuario esté autenticado para tener acceso.

Además, cualquier intento de acceso a una ruta restringida o inexistente (/*) provoca una redirección controlada a la página de inicio (/).

El módulo de recuperación de contraseñas cuenta con un enrutador anidado que permite gestionar sus propias pantallas (*ForgotPassword, OTPVerification, ResetPassword, PasswordChanged*) sin perder el control del flujo.

Estás prácticas contribuyen a mejorar la experiencia del usuario, optimizar el consumo de recursos y garantizar una navegación consistente en toda la aplicación.

Diseño y experiencia de usuario (UI/UX)

El diseño de la interfaz se encaminó a ofrecer una experiencia intuitiva y consistente, asegurando que el usuario pueda interactuar con el sistema de manera sencilla. Una de las prioridades fue las validaciones en los formularios: los campos muestran mensajes de error específicos justo debajo del *input* correspondiente, en color rojo y con un lenguaje claro. Por ejemplo:

- *"El nombre solo debe contener letras"* para nombres incorrectos.
- *"El CUIT solo debe contener números"* para identificaciones inválidas.
- *"La contraseña debe ser alfanumérica y mayor a 7 caracteres"* para claves que no cumplen la política definida.

- *"Este campo es obligatorio"* para los campos requeridos.

En procesos críticos como la verificación de códigos *OTP*, se incluyen instrucciones adicionales para evitar confusión:

- Se notifica el email al que fue enviado el código.
- Se informa el tiempo restante para solicitar un nuevo envío (*"Puede solicitar uno nuevo en {email} en {timeLeft} segundos"*).
- Se habilita el copiado y pegado del código en un solo paso, distribuyéndolo automáticamente entre los distintos campos para el código.

La organización de los componentes también fue planificada para optimizar la interacción. Los elementos de filtrado, búsqueda, ordenamiento y selección de fechas se agrupan de manera más centralizada, mientras que la información estática se presenta en bloques separados y consistentes. Los botones incluyen iconografía, puntero (*pointer*) y cambios visuales al pasar el cursor (*hover states*) para mejorar la percepción de interactividad.

Por otra parte, la implementación de componentes reutilizables para elementos comunes, (*Header*, *Footer*, *inputs*, botones…), permitió estandarizar el estilo y funcionalidad, optimizando el tiempo de desarrollo, reduciendo la duplicidad de código y permitiendo un mantenimiento del diseño más rápido y consistente.

Asimismo, la estructura de navegación mantiene rutas legibles y predecibles, como *Orders/Page/number,* lo que facilita la comprensión y permite la posibilidad de compartir enlaces directos a vistas específicas.

Estas decisiones de diseño guían al usuario en cada paso, reduciendo la carga cognitiva y minimizando la posibilidad de errores.

Observaciones finales

Durante el desarrollo del *Frontend* se mantuvo un enfoque iterativo, donde se revisaban y ajustaban los componentes, *hooks* y servicios a medida que se detectaban mejoras o inconsistencias.

Se priorizó la consistencia visual y funcional, evitando elementos que pudieran generar comportamientos inesperados en producción.

El código se organizó con una estructura clara que facilita la mantenibilidad y escalabilidad, asegurando que los distintos módulos (autenticación, paginación, navegación, filtros, etc.) puedan evolucionar sin afectar el resto del sistema.

Las decisiones técnicas, como el uso de hooks generales y especializados, así como utilidades centralizadas, aportaron a reducir la duplicación de código.

Si bien aún hay optimizaciones planificadas, como la incorporación de medidas para datos de mayor sensibilidad, la base actual es sólida y permite la incorporación de estos cambios sin una gran reestructuración.

4.4.5. Pruebas y validación

Durante la etapa de desarrollo se llevaron a cabo distintas pruebas para verificar el correcto funcionamiento, su estabilidad en distintos entornos y su eficiencia en la utilización de recursos.

Pruebas funcionales en entorno de desarrollo y producción

Se ejecutaron pruebas funcionales sobre los módulos principales tanto en la etapa de desarrollo como en la de producción. Esto incluyó la creación de usuarios, inicio de sesión, recuperación de contraseñas, carga de órdenes, navegación por las vistas, filtrado y búsquedas de órdenes.

En producción se repitieron las pruebas de flujo completo para verificar que no aparecieran errores tras la migración.

Detección y resolución de incidencias en producción

Durante las pruebas que se realizaron posteriormente al despliegue, gracias a la trazabilidad del sincronizador se detectó un problema en el módulo de sincronización de órdenes, en el cual ciertas órdenes no se cargaban correctamente.
La causa de este problema fue debido a la existencia de caracteres especiales dentro de la columna de números de pedidos denominada *RIF,* esto se debe a que es un campo compuesto por un carácter que simboliza el tipo de orden y el número de orden, un escenario no contemplado inicialmente debido a su baja frecuencia de aparición y a la ausencia de comunicación previa sobre su existencia.

A través de una reunión con el cliente se determinó que:

- Las órdenes con *RIF* que contienen “R” debían ser incluidas.
- Otras órdenes que incluían diversos patrones debían ser descartadas.

Ejemplos:

- Inclusión: *1301-R*
- Descarte: *1651-A*

Pruebas de responsividad y compatibilidad

Se evaluó el comportamiento de la aplicación en distintos dispositivos (celulares, *tablets*, *notebooks* y monitores) y navegadores (*Mozilla Firefox*, *Opera GX*, *Google Chrome* y *Microsoft Edge*).

En este proceso se detectó una incompatibilidad con la visualización de íconos *SVG* (por ejemplo, *eye-close / eye-open)* en el campo de contraseña. Para resolverlo, se adoptó la librería *lucide-react* para el manejo, unificando y actualizando los iconos para garantizar consistencia y compatibilidad.

Validación y corrección del manejo de expiración de tokens

Durante las primeras pruebas, se identificó un problema en el *hook useAuthGuard* debido a la diferencia en la duración de los *tokens* entre el entorno de desarrollo (5 minutos) y el de producción (10 meses). Esto ocasionaba que los *timeout* encargados de gestionar el cierre de sesión quedarán fuera de rango, provocando un bucle donde el usuario era enviado al *login*, pero al mismo tiempo era detectado como autenticado y redirigido nuevamente a la vista principal (*/orders*).

Detalles técnicos:

- El valor máximo soportado por *setTimeout* es de *2147483647* milisegundos (aproximadamente más de 24 días).
- Se implementó una condición que, si la expiración supera este límite, se divide el conteo en intervalos más cortos mediante *setInterval*.
- Cuando el tiempo restante era menor o igual al tiempo máximo, se retornaba al manejo estándar con *setTimeout*.

Este conjunto de pruebas permitió identificar y resolver fallos críticos antes de la entrega final, asegurando un producto estable y funcional.

4.4.6. Manejo de errores y adaptación de respuestas

El manejo de errores se diseñó con el objetivo de mostrar mensajes claros y contextualizados al usuario, evitando la exposición de datos internos del backend y manteniendo un tratamiento consistente de los errores en toda la aplicación.

Flujo centralizado de tratamiento de errores

Todas las respuestas de error que afectan a los formularios pasan por la función *handleApiError.*

Este método invoca a *parseBackendError*, que se encarga de:

- Interpretar el mensaje proveniente del *backend*.
- Extraer la propiedad y el mensaje mediante un patrón estandarizado (*Property: X, Error: Y*).

- Convertir el nombre de la propiedad del backend a su equivalente de frontend usando el mapeo *backendToFrontendErrorMap* (por ejemplo, *emailuser* → *email*, *verificationcode* → *otp*).

El resultado es un objeto clave-valor listo para ser asignado al estado *formErrors*.

Cabe destacar que, si se detectan errores asociados a campos específicos, estos se muestran directamente bajo el campo correspondiente.

Gestión de errores generales

Cuando el error no corresponde a un campo particular, como pueden ser problemas de red o fallos en el servidor, *handleApiError* registra el mensaje en *formErrors.general*, el cual se visualiza en un área más general del formulario.

Ejemplos de estos mensajes:

- *"No se pudo conectar con el servidor. Verifica tu conexión."*
- *"Error en la respuesta del servidor."*
- *"Ocurrió un error inesperado en la solicitud."*

Validaciones en el Frontend

Antes del envió a la ***API***, se ejecutan validaciones de estructura para minimizar las llamadas al backend:

- *validateForm*: comprueba campos requeridos, formatos (nombre, email, contraseña, etc.) y valida la confirmación de contraseña.
- *validateOTP*: asegura que el código OTP sea estrictamente numérico y de 4 dígitos.

Los errores detectados por estas validaciones también se cargan en *formErrors* y se muestran en pantalla sin necesidad de contactar al servidor.

Tratamiento global con interceptores

Se configuró un interceptor de respuestas de *Axios* que captura todos los errores *HTTP* y genera un objeto simple con estado y mensaje. Este método permite:

- Manejar de manera centralizada el formato de errores en todos los servicios que usen la misma instancia de *Axios*.
- Diferenciar entre errores de servidor (5xx), de cliente (4xx) y de red.
- Definir un mensaje genérico en caso de respuestas no controladas.

Mensajes informativos no relacionados a errores

En algunos casos particulares, se incluyen mensajes de orientación para el usuario, como:

- Cuando no hay órdenes disponibles: *"Si no puede visualizar las órdenes, tenga paciencia, en lo que resta del día se estará sincronizando con la aplicación."*.
- Mensajes temporales en botones como "*Cotizar pedidos*" generando interés en futuras funciones.

Gracias a este conjunto de validaciones se puede garantizar que los errores que se producen están previamente procesados, categorizados y mostrados de manera clara y segura.

4.4.7. Documentación

En cuanto a la documentación, actualmente se hizo entrega de un **manual de usuario** que incluye tanto las instrucciones como la guía de uso del sistema, asegurando que cualquier persona pueda replicar el proceso de despliegue y entender la manera de funcionar. Además, el equipo realizó directamente la instalación en el entorno productivo, lo cual complementa la documentación con una validación práctica.

Por último, está pendiente la construcción de una **documentación técnica** estandarizada en *Confluence*, que garantizará la trazabilidad y mantenimiento del proyecto.

5. Verificación

5.1. Verificación del diseño de solución planteado

Durante la etapa de verificación, se evaluó el cumplimiento de los objetivos propuestos inicialmente, que consistía en desarrollar una aplicación que permitiera a los clientes visualizar y seguir el estado de sus órdenes. Además, brindar un flujo consistente a través de validaciones y manejo de errores, garantizando la experiencia del usuario.

En la etapa final se verificó que el sistema desarrollado cumplió con lo planteado, las pruebas que se incluyeron fueron:

- **Pruebas funcionales**: validación de registro, inicio de sesión, visualización de órdenes y manejo de errores.
- **Pruebas de integración**: comunicación entre el ***Frontend*** y la ***API*** de manera estable, verificando forzosamente la traducción de errores técnicos a mensajes entendibles para el usuario.
- **Pruebas de rendimiento del Sincronizador**: evaluación de los tiempos de carga y actualización de datos, se consideró el uso de recursos del servidor y el horario en que la empresa realiza *backup* para no sobrecargar.

La solución resultante es mantenible y escalable, permitiendo incorporar nuevas funciones sin necesidad de realizar grandes reestructuraciones.

5.2. Evaluación de impacto

Impacto económico

- Reducción de la carga operativa del personal debido a que el cliente puede realizar seguimiento de sus órdenes de manera independiente.
- Reducción de carga en el servidor gracias a la optimización del proceso de sincronización.
- Potencial incremento en la satisfacción del cliente, conduciendo a la retención del mismo.
- Potencial incremento de clientes y reducción de costos operativos a mediano plazo, al automatizar parte del proceso de venta con la integración de un módulo de cotización desde la misma plataforma.

Impacto ambiental

- Menor necesidad de impresión de documentos debido a la digitalización de consulta de órdenes.
- Uso eficiente de recursos del servidor.

- Disminución de consumo de combustible para obtener información, debido a la digitalización.

Impacto social

- Mejora en la experiencia del cliente, ofreciendo una plataforma fácil, rápida y disponible en todo momento.
- Proporciona de manera accesible información relevante, como el seguimiento de órdenes, mejorando la satisfacción del usuario.
- Facilita el acceso a la información, mediante formularios simples y mensajes claros, sin requerir capacitación previa.

5.3. Evaluación de condiciones de seguridad e higiene

El proyecto se desarrolló principalmente de manera remota, con reuniones virtuales periódicas con el cliente. De forma ocasional se realizaron encuentros presenciales en la empresa, cumpliendo con las medidas de seguridad e higiene correspondiente.

En cuanto a las medidas de seguridad implementadas:

- **Control de errores y validaciones** para asegurar la integridad de datos y prevenir comportamientos inesperados.
- **Restricciones de acceso** a la información interna de la empresa.
- **Comunicación segura** entre la empresa, la plataforma y el cliente, para minimizar el riesgo de filtración de datos.

5.4. Consideraciones no previstas en la planificación inicial

Durante el despliegue del proyecto surgieron situaciones técnicas que no estaban contempladas en la etapa de planificación, algunos ejemplos:

- **Limitación de la cantidad de órdenes sincronizadas**: se estableció un límite (último año) a pedido del cliente, debido a la carga excesiva de datos afectaba el rendimiento del servidor.
- **Restricciones del dominio privado**: el dominio de la empresa restringía la comunicación del sincronizador hacía la *API*, lo que requirió ajustes en la configuración de red y permisos internos.

- **Suspensión del servicio por inactividad**: a solicitud del cliente, el sincronizador se desarrolló como una "*API*" sin *endpoints* en vez de un ejecutable. Debido a esto, el servicio entraba en estado de suspensión por inactividad (incluso con configuraciones de *IIS* para evitarlo). Para mantenerlo activo, se implementó un mecanismo de autopedidos periódicos.

6. Conclusión

6.1. Reflexión personal sobre la experiencia

La PPS me permitió consolidar y ampliar conocimientos técnicos en:

- Integración frontend-backend con manejo centralizado de errores y validaciones.
- Optimización de procesos para mejorar el rendimiento y la escalabilidad.
- Aplicación de buenas prácticas en seguridad y comunicación entre sistemas.

A pesar de que el proyecto presentó tecnologías y escenarios nuevos, el proceso de adaptación fue más ágil gracias a las bases adquiridas en distintas materias de la carrera, como *Ingeniería de software*, *Proyecto de software* y *Base de datos*, entre otras que aportaron conceptos fundamentales para la metodología de trabajo, estructuración de datos y desarrolló del sistema.

En el plano no técnico, se fortalecieron habilidades de comunicación, trabajo en equipo y adaptación a imprevistos.

Esta experiencia aportó un valor significativo a mi desarrollo como profesional, brindándome herramientas para afrontar proyectos de mayor complejidad. La realización de la PPS será un antecedente importante para mi futuro laboral, especialmente en ámbitos donde el aprendizaje constante sea un factor determinante para el éxito.

7. Bibliografía

Martin, Rober Cecil (2012). Código Limpio: Manual de Estilo para el Desarrollo Ágil de Software. Madrid, España, Pearson Educación.

Atlassian (s.f.). “Confluence Data Center Documentation”. Disponible en: https://confluence.atlassian.com/doc/confluence-documentation-135922.html. [Consulta: 30 de junio de 2025]

Automapper Project (2024). “AutoMapper”. Disponible en: https://docs.automapper.io/en/latest/. [Consulta: 24 de febrero de 2025].

Microsoft (s.f.). “C# documentation”. Disponible en: https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp. [Consulta: 5 de febrero de 2025]

CodePen (2024). “CodePen”. Disponible en: https://codepen.io/topic/react/templates. [Consulta: 12 de agosto de 2025].

Donweb (2024). “Hosting y Dominios”. Disponible en: https://donweb.com/. [Consulta: 8 de julio de 2025].

Firebase (2024). “Firebase developer Documentation”. Disponible en: https://firebase.google.com/docs. [Consulta: 28 de mayo de 2025].

FluentValidation Project (2024). “FluentValidation”. Disponible en: https://docs.fluentvalidation.net/. [Consulta: 24 de febrero de 2025].

Mailtrap (2024). “Email Delivery Platform built for products that send big”. Disponible en: https://mailtrap.io/. [Consulta: 21 de marzo de 2025].

Meta (2024). “React”. Disponible en: https://react.dev/. [Consulta: 5 de febrero de 2025].

Microsoft (2024). “Documentation de .NET”. Disponible en: https://learn.microsoft.com/es-es/dotnet/. [Consulta: 5 de febrero de 2025].

Microsoft (2024). “Arquitecturas de aplicaciones web comunes”. Disponible en: https://learn.microsoft.com/es-es/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures. [Consulta: 5 de febrero de 2025].

Microsoft (s.f.). “TypeScript is JavaScript with syntax for types.”. Disponible en: https://www.typescriptlang.org. [Consulta: 5 de febrero de 2025]

PostgreSQL Global Development Group (2024). “Documentation”. Disponible en: https://www.postgresql.org/docs/. [Consulta: 5 de febrero de 2025].

Shadcn (2024). “The Foundation for your Design System”. Disponible en: https://ui.shadcn.com/. [Consulta: 3 de marzo de 2025].

Youtube (2022). “Clean Architecture en Frontend – Parte 1”. Disponible en: https://youtu.be/5LqhlCd2_nE. [Consulta: 5 de febrero de 2025].

Youtube (2022) “Figma tutorial para principiantes | 👏 Aprende Diseño Web UI de manera simple a través de Figma”. Disponible en: https://youtu.be/bIK7PIdlLTU. [Consulta: 7 de febrero de 2025].

Youtube (2022) “Como diseñar una página web bonita y sin saber de diseño | Figma”. Disponible en: https://youtu.be/Mx8h0SRu-Iw. [Consulta: 7 de febrero de 2025].

Youtube (2021) “Figma - Como Diseñar una App de Gastos”. Disponible en: https://youtu.be/fl9lZ0Ggb14. [Consulta: 7 de febrero de 2025].

Youtube (2021) “¿Cómo crear enlaces en tus diseños de Figma?”. Disponible en: https://youtu.be/I-o48K9BjKU. [Consulta: 12 de febrero de 2025].

Youtube (2021) “🌈 Cómo crear un enlace a una imagen de Figma”. Disponible en: https://youtu.be/CC42loY7DPM. [Consulta: 12 de febrero de 2025].

Youtube (2023) “Add links into icon in figma | Expert Azi | Put URL in icon Figma”. Disponible en: https://youtu.be/FgufK9-Dxmw. [Consulta: 12 de febrero de 2025].

Youtube (2021) “#FIGMATUTORIAL 👉 Cómo crear un SEARCH BAR paso a paso”. Disponible en: https://youtu.be/YEesTrEVdgY. [Consulta: 11 de febrero de 2025].

8. Glosario

8.1. Términos generales

- **API Key**: Clave única utilizada para autenticar solicitudes hacía una *API*.
- **Backup**: Copia de seguridad de datos para su recuperación en caso de pérdida o corrupción.
- **Booleano**: Tipo de dato que solo puede tomar dos valores: verdadero o falso.
- **Build**: Proceso de compilación, empaquetado y optimización del código para su despliegue.
- **Dashboard**: Interfaz gráfica que muestra información resumida y relevante, generalmente en tiempo real.
- **ECMAScript (ESM)**: Estándar que define el lenguaje *JavaScript***,** con especificaciones de sintaxis y funcionalidades.
- **Endpoints**: Puntos finales de una *API* que definen rutas específicas para acceder a datos o funcionalidades.
- **Extra reducers**: *Reducers* adicionales definidos fuera de la propiedad principal *reducers* en *Redux Toolkit.*
- **Feedback**: Retroalimentación. Respuesta o devolución que proporciona un usuario o cliente ante una acción o funcionalidad.
- **Firebase**: Plataforma de *Google* que ofrece servicios en la nube para desarrollo de aplicaciones web o móviles.
- **Footer**: Parte inferior de una página o aplicación, que suele contener enlaces, información legal o de contacto.
- **Framework**: Estructura de trabajo que proporciona un conjunto de herramientas y convenciones para el desarrollo.
- **Header**: Parte superior de una página o aplicación, generalmente con navegación y *branding*.
- **Hooks**: Funciones especiales en *React* que permiten usar estado y otras características sin escribir clases.
- **Hot Module Replacement (HMR)**: Funcionalidad que permite actualizar módulos de una aplicación sin recargar toda la página.
- **Hover states**: Estilos o comportamientos que se aplican a un elemento cuando el cursor pasa sobre él.
- **Inputs**: Campos de entrada en una interfaz donde el usuario puede introducir datos.
- **Pointer**: Indicador visual o cursor que señala un elemento en la interfaz.
- **Reducer**: Función pura que determina cómo cambia el estado en respuesta a acciones.

- **Refresh tokens**: Tokens que permiten obtener nuevos tokens de acceso sin volver a iniciar sesión.
- **Schedule**: Proceso o conjunto de instrucciones que se ejecutan en momentos programados.
- **Schedule component**: Componente especializado para programar eventos o acciones en un sistema específico.
- **Scrum**: Metodología ágil de gestión de proyectos enfocada en entregas iterativas e incrementales.
- **Setters**: Funciones que permiten modificar valores almacenados en un estado u objeto.
- **Slices**: Segmentos del estado global manejados por Redux Toolkit.
- **Socket**: Canal de comunicación que permite el intercambio de datos entre dos sistemas o procesos.
- **Stack**: Conjunto de tecnologías, herramientas y lenguajes que se utilizan en un proyecto de desarrollo.
- **Store**: Contenedor centralizado que gestiona el estado de una aplicación.
- **Superset**: Herramienta de visualización y análisis de datos de código abierto.
- **Timeout**: Periodo máximo de espera antes de cancelar una operación por inactividad o falta de respuesta.
- **Tipada**: Característica de un lenguaje o sistema que define y restringe el tipo de dato que puede almacenar o manipular.
- **Token**: Cadena de caracteres utilizada para autenticar y autorizar solicitudes en un sistema.
- **Utility-first**: Filosofía de *CSS* que prioriza clases pequeñas y reutilizables para construir interfaces.

8.2. Acrónimos

- **API:** Application Programming Interface (Interfaz de Programación de Aplicaciones).
- **CRUD**: Create, Read, Update, Delete (crear, leer, actualizer, borrar).
- **CSP**: Content Security Policy.
- **CUIT**: Clave Única de Identificación Tributaria.
- **DTO**: Data Transfer Object (objeto de transferencia de datos).
- **ESM**: ECMAScript Modules.
- **HTTP**: Hypertext Transfer Protocol.
- **IIS**: Internet Information Services.
- **JSON**: JavaScript Object Notation.
- **JWT**: JSON Web Token.
- **OTP**: One-Time Password (contraseña de un solo uso).

- **SEO**: Search Engine Optimization (optimización de una sola página).
- **SPA**: Single Page Application (aplicación de una sola página).
- **SVG**: Scalable Vector Graphics.
- **UI**: User Interface (interfaz de usuario).
- **URL**: Uniform Resource Locator (localizador uniforme de recursos).
- **UX**: User Experience (experiencia del usuario).

9.Anexos

9.1.1.Anexo 1 – Pantalla de inicio de sesión (Desktop)

Figura 1. Pantalla de inicio de sesión (Desktop).

La Figura 1 presenta la pantalla de inicio con campos de correo y contraseña que incluyen validaciones en tiempo real, opción de mostrar/ocultar contraseña, enlace de recuperación y mensajes de error contextuales. En la parte inferior, se incluye un pie de página con enlaces hacia la página oficial y redes sociales de la empresa (Facebook e Instagram)

Nota. Elaboración propia, basada en la plataforma web desarrollada para Diserglass.

9.1.2.Anexo 2 – Pantalla de inicio de sesión (Tablet)

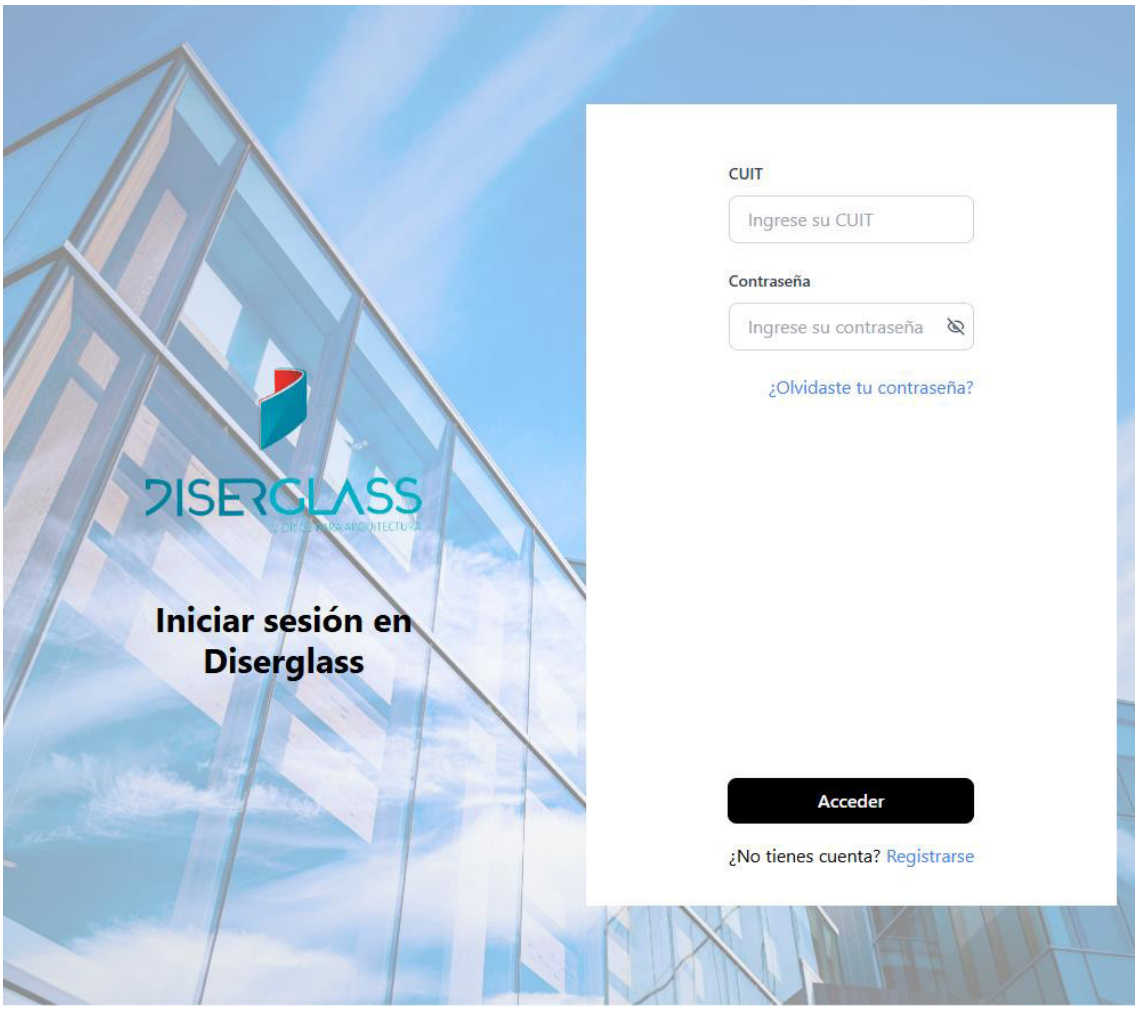

Figura 2. Pantalla de inicio de sesión (Tablet).

La Figura 2 muestra campos de correo y contraseña con validaciones en tiempo real, posibilidad de mostrar/ocultar la contraseña, enlace de recuperación y mensajes de error contextuales. Su diseño se adapta al formato vertical, optimizando la usabilidad en pantallas intermedias.

Nota. Elaboración propia, basada en la plataforma web desarrollada para Diserglass.

9.1.3.Anexo 3 – Pantalla de inicio de sesión (Mobile)

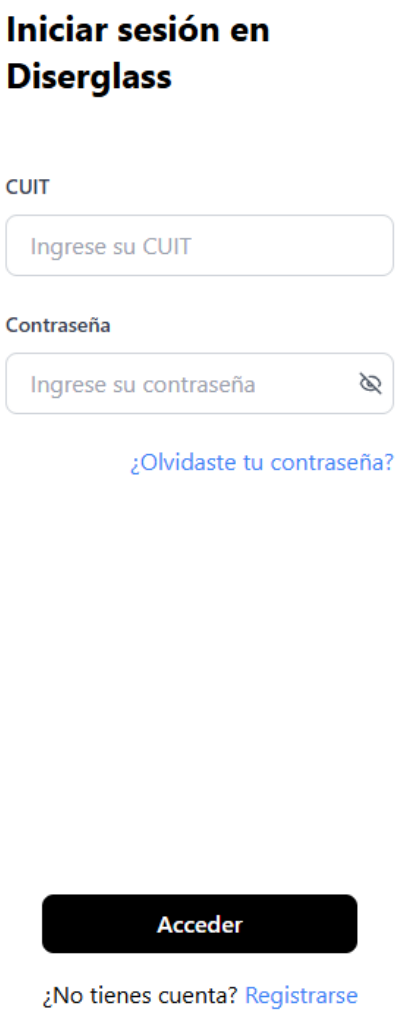

Figura 3. Pantalla de inicio de sesión (Mobile).

La Figura 3 contiene campos de correo y contraseña que incluyen validaciones en tiempo real, opción de visualizar la contraseña, enlace de recuperación y mensajes de error contextuales. Su diseño esta adaptado a pantallas pequeñas para facilitar el acceso desde dispositivos móviles.

Nota. Elaboración propia, basada en la plataforma web desarrollada para Diserglass.

9.1.4.Anexo 4 – Pantalla de órdenes (Desktop)

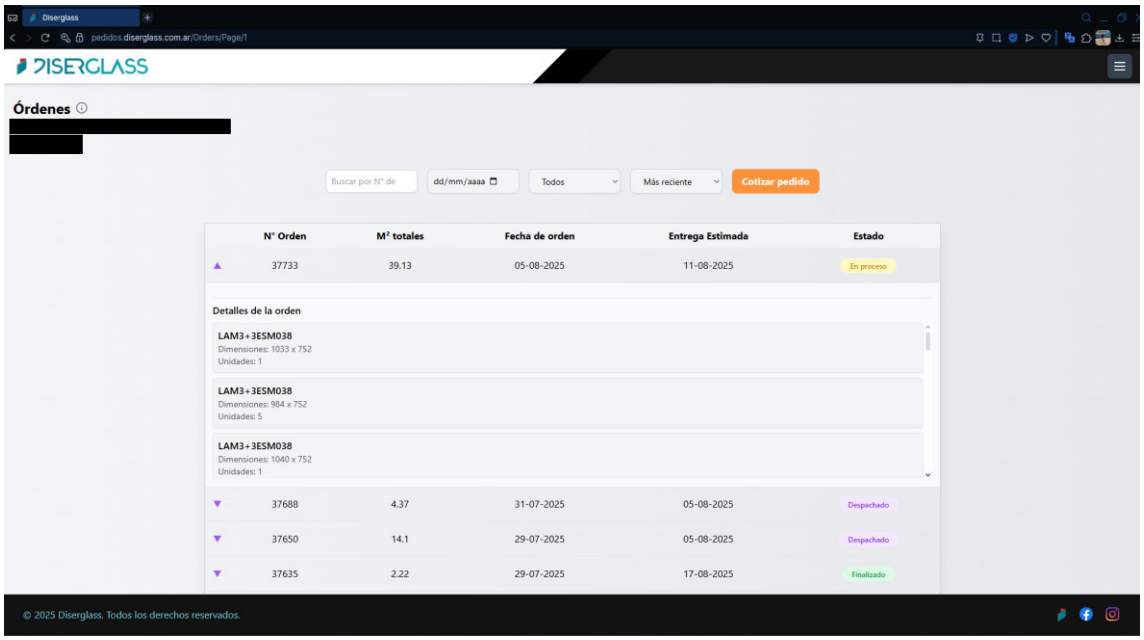

Figura 4. Pantalla de órdenes (Desktop).

La Figura 4 presenta una barra de navegación superior con el logotipo y menú de usuario (información de la cuenta y opción de cierre de sesión). Debajo se muestra el título “Órdenes” acompañado de información básica de la cuenta y un icono informativo que despliega detalles del sistema. En el área central se ubican los filtros, buscador y el botón para cotizar pedidos (funcionalidad en evolución). La sección principal muestra un listado de órdenes actuales e históricas, con opción de expandir cada una en formato acordeón para visualizar detalles. Finalmente, en el pie de página se incorporan enlaces a la página oficial y redes sociales.

Nota. Elaboración propia, basada en la plataforma web desarrollada para Diserglass.

9.1.5.Anexo 5 – Pantalla de órdenes (Tablet)

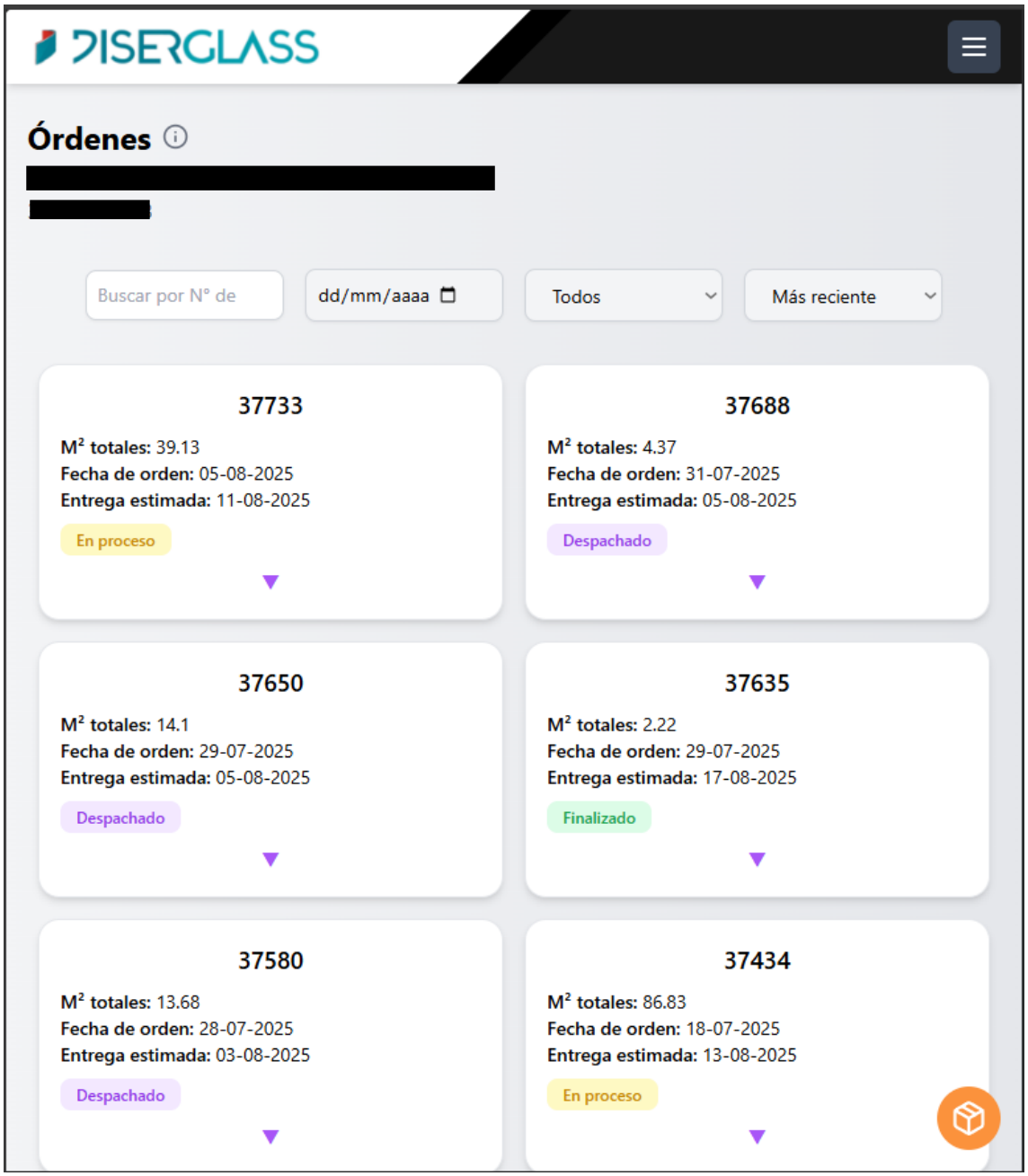

Figura 5. Pantalla de órdenes (Tablet).

La Figura 5 mantiene la estructura de la vista desktop, conservando la barra de navegación, título, ícono informativo, filtros, buscador y selector de ordenamiento. Las órdenes se presentan en tarjetas individuales distribuidas en cuadrícula de dos columnas, cada una con opción de expandirse para ver detalles. En la esquina inferior derecha se incluye el acceso flotante a la función de cotización.

Nota. Elaboración propia, basada en la plataforma web desarrollada para Diserglass.

9.1.6. Anexo 6 – Pantalla de órdenes (Mobile)

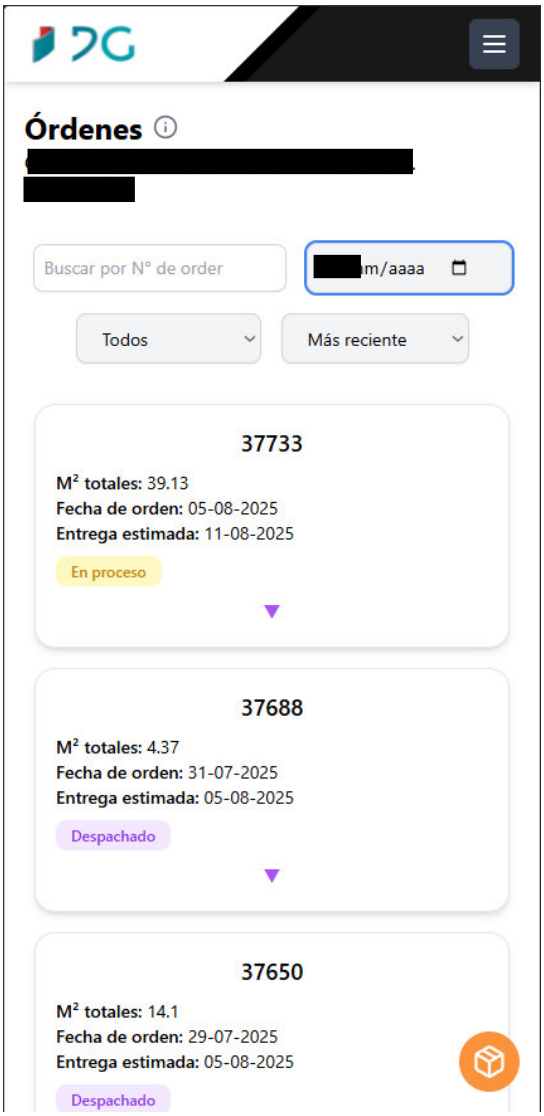

Figura 6. Pantalla de órdenes (Desktop).

En la figura 6 la pantalla se reorganiza en una sola columna para optimizar la visualización en pantallas reducidas. Se mantiene la barra de navegación con logotipo abreviado y menú de usuario desplegable, seguido del título “Órdenes”, información de la cuenta e ícono informativo. Los filtros, buscador y selector de ordenamiento se disponen en formato vertical. Las órdenes se muestran en tarjetas apiladas, cada una con la información principal y opción de expandir para ver detalles. En la esquina inferior derecha se incorpora el acceso flotante a la función de cotización.

Nota. Elaboración propia, basada en la plataforma web desarrollada para Diserglass.

9.1.7. Anexo 8 – Nota final sobre el despliegue del sistema web

La aplicación web se encuentra desplegada en producción y disponible en la *URL:* https://pedidos.diserglass.com.ar*.* Sin embargo, se debe tener en cuenta que el acceso es limitado debido a la verificación por CUIT solo para clientes existentes.