

*Proyecto*

*de CFGS*

*2º DAM*



## **Pequeño ensayo expositivo sobre la fase de codificación**

### **Introducción**

El presente trabajo pretende mostrar la funcionalidad conseguida durante esta fase del ciclo de vida del software. Es decir, ahora se entraría en una pequeña iteración para la mejora de la calidad del software; así como su deployment.

No obstante, este proyecto se consideró desde una perspectiva ágil. Por tanto, en el desarrollo de este trabajo me he visto obligado en la necesidad de usar herramientas complementarias para encontrar fallos en el razonamiento de la idea y en la implementación. Por ejemplo, en la parte de desarrollo de la API, Postman fue de mucha ayuda. Pero en el desarrollo del frontend, las herramientas de desarrollo del propio navegador también fueron útiles.

Todo ello, mezclado con un exhaustivo análisis de mensajes de errores en las diferentes pilas ha dado como resultado, la implementación de estas funcionalidades.

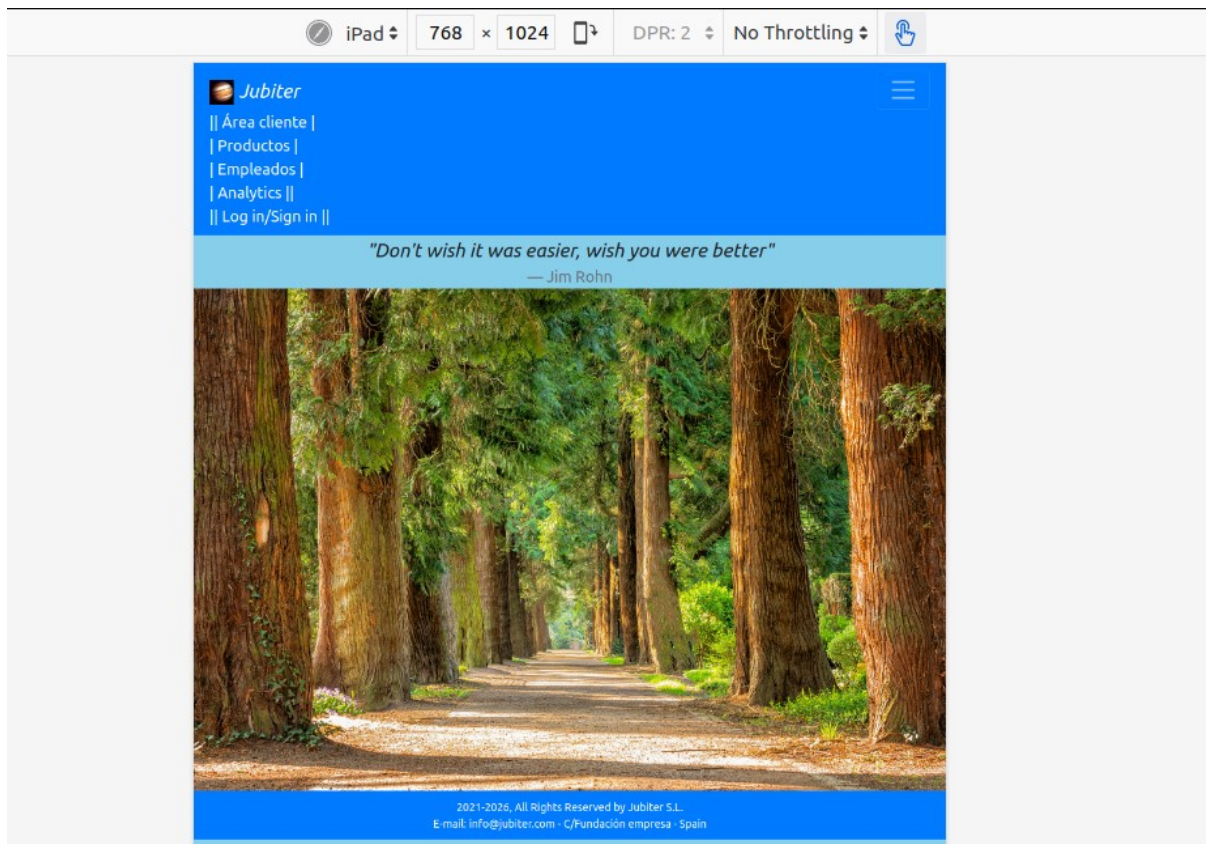
**Nota:** se ha comentado el código, cada clase y componente, de una manera más o menos completa. Por tanto, se intentará ahondar en aquellos conceptos que den sentido al conjunto; ya que el código se acompañará por medio de un enlace (pesa demasiado).

### **Estructuración del proyecto (frontend capa JS/React): funcionalidades.**

Esta no las he podido empaquetar aún, pero sí que se podrían organizar en función del recurso que se carga en la SPA.

Por tanto, se podrían considerar las siguientes divisiones:

- inicio → se cargará el componente Home.js.



Página de inicio, se trata de conseguir responsiveness.  
Fuente: elaboración propia.

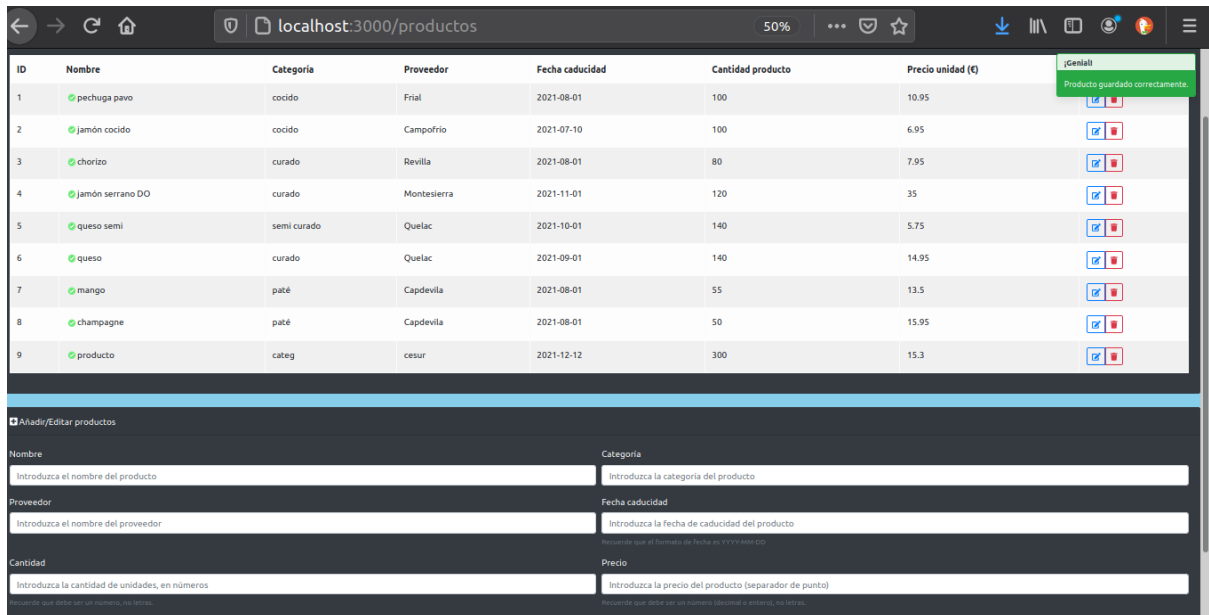
- Productos :

Este componente permite eliminar un producto de la lista, (editar aún no).

En la parte inferior al rellenarlo y pulsar el botón de guardar se guarda.

Tanto al guardar como al eliminar, componentDidMount() hace que aparezca inmediatamente. Además se informa con un mensaje verde o rojo, respectivamente.

2º Desarrollo Aplicaciones Multiplataforma – Proyecto – Desarrollo  
 Docente: Santiago Matín-Palomo García  
 Alumno: Alberto Ruiz Moreno



The screenshot shows a web browser at localhost:3000/productos. The top part is a table with 9 products. The bottom part is a form titled 'Añadir/Editar productos' with fields for Name, Category, Provider, Expiry Date, Quantity, and Price.

ID	Nombre	Categoría	Proveedor	Fecha caducidad	Cantidad producto	Precio unidad (€)
1	pechuga pavo	cocido	Frial	2021-08-01	100	10.95
2	jamón cocido	cocido	Campofrio	2021-07-10	100	6.95
3	chorizo	curado	Revilla	2021-08-01	80	7.95
4	jamón serrano DO	curado	Montesierra	2021-11-01	120	35
5	queso semi	semi curado	Quelac	2021-10-01	140	5.75
6	queso	curado	Quelac	2021-09-01	140	14.95
7	mango	paté	Capdevila	2021-08-01	55	13.5
8	champagne	paté	Capdevila	2021-08-01	50	15.95
9	producto	categ	cesur	2021-12-12	300	15.3

**Añadir/Editar productos**

Nombre:

Categoría:

Proveedor:

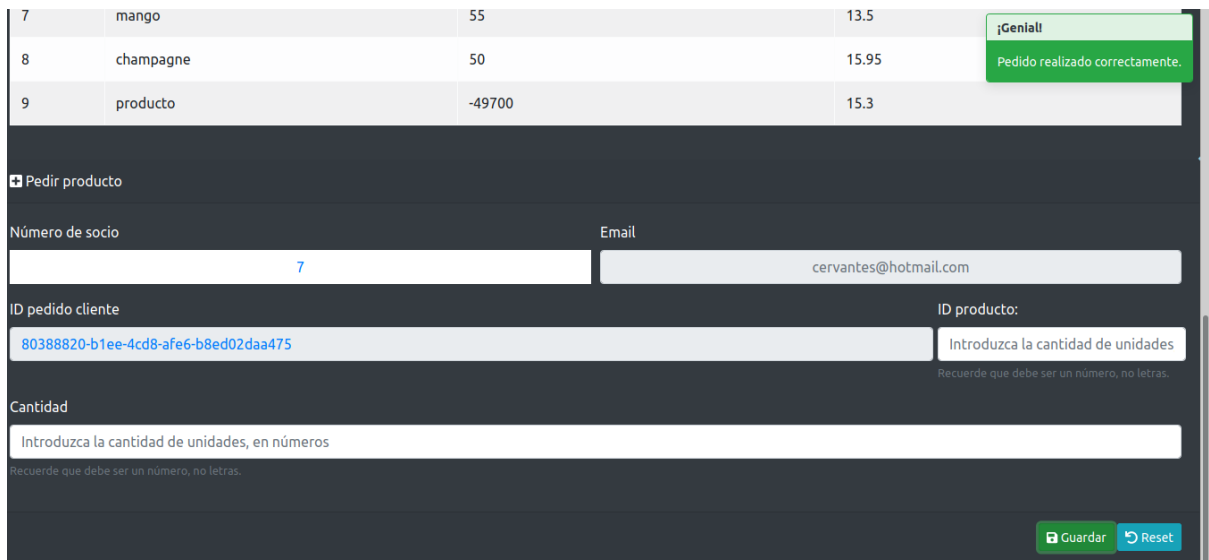
Fecha caducidad:

Cantidad:

Precio:

Productos une una lista de los productos con un formulario de relleno.  
 Fuente: elaboración propia.

- Área de clientes:



The screenshot shows a web browser at localhost:3000/clientes. The top part is a table with 3 products. The bottom part is a form titled 'Pedir producto' with fields for Number of associate, Email, ID of client order, ID of product, and Quantity.

7	mango	55	13.5
8	champagne	50	15.95
9	producto	-49700	15.3

**Pedir producto**

Número de socio:

Email:

ID pedido cliente:

ID producto:

Cantidad:

Área de clientes.  
 Fuente: elaboración propia.

Se puede comprobar que el artículo añadido anteriormente si se hace un pedido, se resta automáticamente a la vez que informa del estado de la petición.

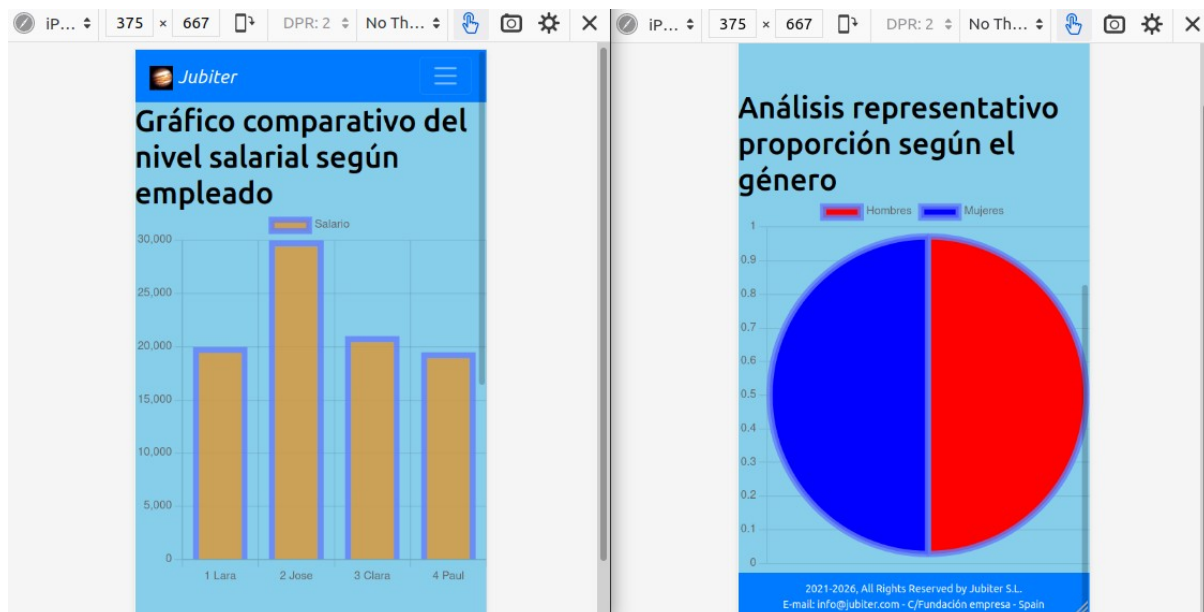
En este caso el formulario simula un previo registro, ya que tiene campos de sólo lectura.

#### - Zona de empleados:

Aquí se implementaría la lógica con la seguridad de los datos sensibles.

#### - Analytics:

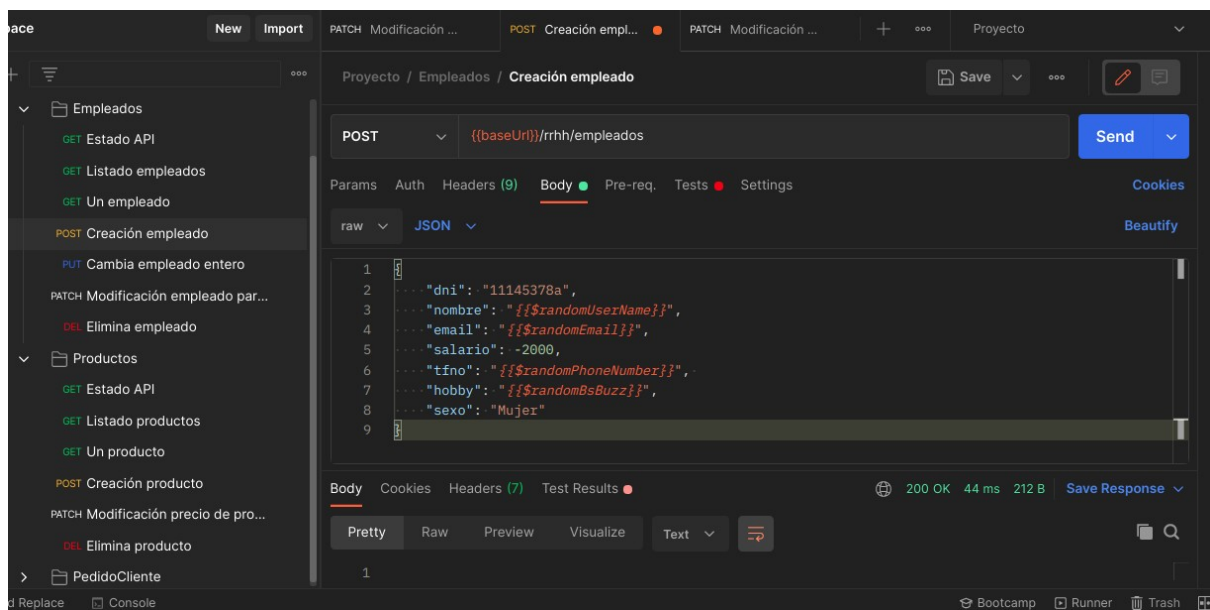
Aquí se ve el estado inicial del gráfico de barras y el de tarta. Los datos son tomados de la propia información de la empresa. Se ve, también, que se ajusta a la pantalla.



Analytics previo POST Postman.  
Fuente: elaboración propia.

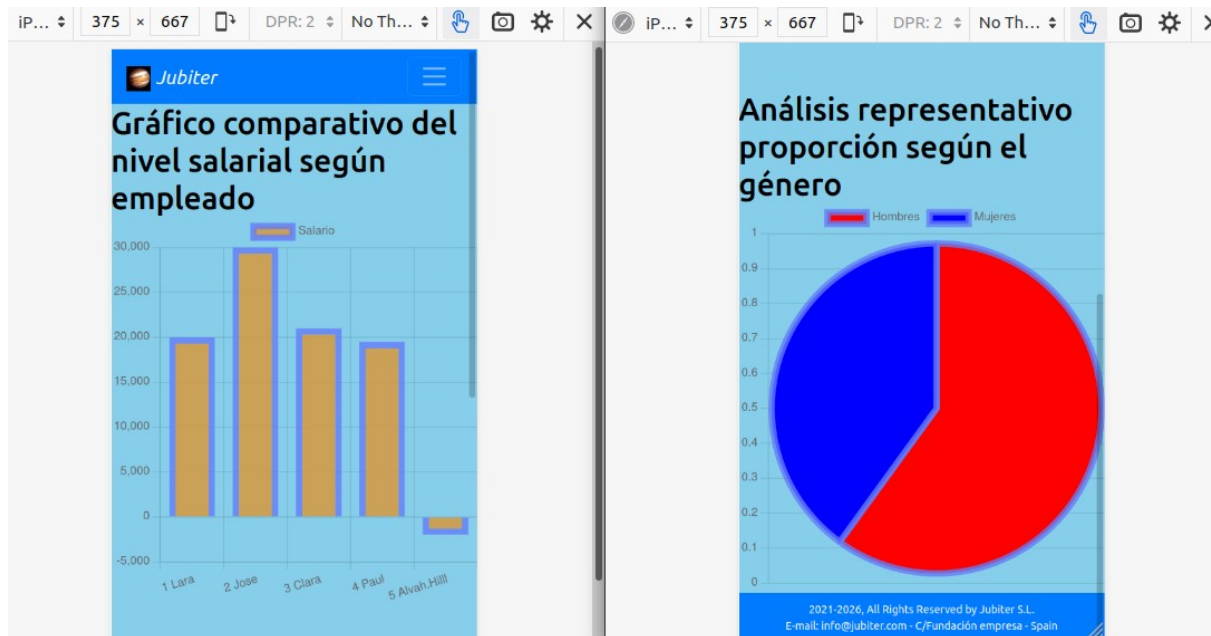
Esta siguiente imagen presenta al cliente Postman, que ha sido una herramienta fundamental para entender el funcionamiento entre una API backend y un cliente.

Se expone que se han realizado varias colecciones: Clientes, Empleados, Productos y PedidoCliente. A excepción de la última, que sólo tiene un método, el resto están prácticamente todos (los que están implementados en Java). Queda pendiente el automatizar los test y quizá extraerlo con Newman.



POST desde Postman.  
Fuente: elaboración propia.

Como resultado se obtiene lo siguiente:



Resultado POST Postman.  
Fuente: elaboración propia.

Se pueden comprobar varias cosas interesantes, una que el gráfico de barras permite valores negativos; y segundo, que funcionan ambas lógicas de manera correcta.

En ambos casos, se ha realizado un pequeño tratamiento de los datos propios de la empresa: tomar valores desde la API, pasarlos a arrays exponerlos como resultado. Esto permitiría a la empresa tomar decisiones más informadas.

Mencionar que la librería que he usado fue Chart.js, porque me daba error D3.js.

Se ha usado React-bootstrap para conseguir esos matices de adaptación a los dispositivos.

Y se usó el cliente axios para el traspaso HTTP.

### **Estructuración del proyecto (backend capa Java): funcionalidades.**

Obviando el propio pom.xml que es donde están todas las dependencias que se necesitan y otras opcionales; y el archivo de propiedades en el directorio resources, en donde se exponen otro tipo funcionalidades (enlace con BBDD, configuración Hibernate, configuración del servidor y seguridad). El proyecto se ha dividido en los siguientes bloques/paquetes<sup>1</sup>:

- **com.jubiter** → se encuentra la clase desde donde arranca la aplicación.
- **com.jubiter.config** → están las clases de configuración que inicializan los valores de las tablas de la base de datos.
- **com.jubiter.seguridad** → es una clase que controlaría el acceso a la API de manera completa, según el PATH o por roles. De implementación básica, pero sirve conceptualmente para estructurar siguientes implementaciones. Se podría decir que es una feature en estado de desarrollo para apendizarla a producción próximamente.
- **com.jubiter.logica** → se trata de un conjunto de clases e interfaces que pretenden aportar la lógica de las relaciones entre las partes. Se ha hardcodeado un poco y no se han hecho uso de ellas, pues no he sabido resolver algunos errores de conexión que tenía. Sin embargo, la lógica es aplicada porque la aplicación funciona.
- **com.jubiter.exception** → define algunas excepciones personalizadas y que son usadas en el aplicativo.

---

1 Ordenados en función de la implementación y cercanía desde la perspectiva de un cliente usuario.



- **com.jubiter.modelo** → se encuentran todos los modelos de objetos que son usados. Otros serían implementados en una futura etapa. Especial mención tendría la que establece una relación ternaria entre las clases almacén-pedido cliente-producto.

En cada clase, se muestran las relaciones y anotaciones tanto de Hibernate como de Spring o Javax.

- **com.jubiter.controlador** → como su nombre indica, se encuentran los controladores de la aplicación.

- **com.jubiter.service** → es el enlace entre el controlador y el repositorio. Es donde se encuentran relaciones lógicas. Por ejemplo, la que permite hacer un pedido y que éste reste la cantidad del producto directamente.

- **com.jubiter.repository** → es el paquete que establecerá la conexión con entre el servicio y la base de datos. Se implementan diferentes interfaces (la propia CrudRepository o alguna subclase).

### **Estructuración del proyecto (backend capa persistencia PostgreSQL y Docker): funcionalidades.**

Aunque no se ha presentado una imagen, sí que implícitamente se puede ver el uso de PostgreSQL y Docker a través de los ficheros que acompañan al proyecto.

Esta parte sirve para relacionar más la parte de codificación con la automatización de los procesos intermedios, IoT, scripting y las redes inteligentes.

El hecho de tener a PostgreSQL en un contenedor, hace pensar en que se realizan test de integración de manera casi constante y si demasiados recursos. Al menos es así, como me lo he tomado.

En este ámbito hay muchas maneras distintas de hacer las cosas, por ejemplo, para inicializar PostgreSQL se puede hacer entrando al contenedor manualmente, importándolo, que se inicialice cuando se inicia el directorio principal de carga de la imagen, a través de la aplicación Java (como se hizo finalmente), con scripts que corran fuera del contenedor (archivos ejecutables ./create... ./remove ...) y parece ser que cada a habrán nuevas formas. Sin embargo, se viene a la mente simplemente shell Linux/Unix.

**Nota:** en Windows funciona de forma similar, pero se necesita instalar una capa de software intermedia que haga creer a docker que corre en Linux.

### **Requisitos.**

Para el frontend: node.

Para el backend: mvn, jdk/jre/

Para persistir: o bien PostgreSQL, o Docker con la imagen.

Se tratará de hacer un deployment independiente.

### **Lista de referencias más relevantes**

Spring.io

Javaguides.

JavaBrains.

AmigosCode.

Baeldung.

Oracle.

StackOverflow.

Documentación: React, Bootstrap, Chart.js.