UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

# Desarrollo de una aplicación web Full Stack para la reserva de barcos y viajes

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Pérez Rico, Roberto

Tutor/a: Sáez Barona, Sergio

CURSO ACADÉMICO: 2022/2023

# Resumen

El auge de los servicios ha facilitado el crecimiento de nuevos modelos de negocio, tales como el alquiler por días de embarcaciones, y de nuevas plataformas online en el mercado.

Es por esto, que se ha decidido el desarrollo de una aplicación web llamada Vela; que permita al usuario buscar, filtrar y reservar embarcaciones, tanto para la reserva por días como viajes creados dentro de la propia aplicación. Esta aplicación está enfocada a dar una solución de software para el desarrollo de aplicaciones en empresas emergentes (start-ups). Se tratarán diferentes temas tales como la integración y despliegue continuo, pagos online, detección y corrección de fallos en tiempo real, versionado de la aplicación, metodologías ágiles y desarrollo en diferentes entornos para ofrecer una posibilidad a otros equipos dentro de nuestra empresa a realizar pruebas en un entorno real.

**Palabras clave:** reservas, MEVN stack, start-ups, Github Actions, CI/CD, scrum, pagos online

# Abstract

The boom in services has facilitated the growth of new business models, such as boat rentals for days, and new online platforms in the market.

It is for this reason that the development of a web application called Vela has been decided; that allows the user to search, filter and book boats, both for booking for days and trips created within the application itself. This application is focused on providing a software solution for the development of applications in small companies (start-ups). Different topics will be covered such as integration and continuous deployment, online payments, real-time bug detection and correction, versioning of the application, agile methodologies, and development in different environments to offer a possibility to other teams within our company to carry out tests. in a real environment.

**Keywords:** bookings, MEVN stack, start-ups, CI/CD, Github Actions, scrum, online payments

# Tabla de contenidos


1.1 MOTIVACIÓN ..... 1
1.2 OBJETIVOS ..... 1
1.3 ESTRUCTURA DE LA MEMORIA ..... 1

**2 ESTADO DEL ARTE ..... 3**

2.1 RESUMEN DE VELA ..... 3
2.2 IDEA DE NEGOCIO ..... 3
2.3 ESTUDIO ESTRATÉGICO ..... 3
*2.3.1 Análisis de competidores ..... 3*
*2.3.2 Identificación del público objetivo ..... 6*
*2.3.3 Modelo de negocio ..... 6*
*2.3.4 Evaluación de la propuesta de valor de nuestro modelo de negocio ..... 6*

**3 ESPECIFICACIÓN DE REQUISITOS ..... 7**

3.1 INTRODUCCIÓN ..... 7
3.2 REQUISITOS FUNCIONALES ..... 7
3.4 REQUISITOS NO FUNCIONALES ..... 7

**4 ANÁLISIS ..... 8**

4.1 MARCO TEÓRICO Y ANTECEDENTES ..... 8
*4.1.1 Metodología de trabajo ..... 8*
*4.1.1.1 Metodología de trabajo: Scrum ..... 8*
*4.1.1.2 Metodología de trabajo: Kanban ..... 9*
*4.1.1.3 Metodología a usar en el proyecto ..... 9*
*4.1.2 Alcance ..... 9*
*4.1.3 Tecnologías y procedimientos en el proyecto ..... 10*
4.2 DIAGRAMA ENTIDAD-RELACIÓN ..... 10
4.3 CASOS DE USO ..... 11

**5 DISEÑO Y ESTRUCTURA DE LA APLICACIÓN ..... 16**

5.1 ARQUITECTURA POR CAPAS EN EL BACKEND ..... 16
*5.1.1 Intención ..... 16*
*5.1.2 Motivación ..... 16*
*5.1.3 Solución propuesta ..... 16*
5.2 ARQUITECTURA MODELO-VISTA-CONTROLADOR EN EL FRONTEND ..... 17
*5.2.1 Intención ..... 17*
*5.2.2 Motivación ..... 18*
*5.2.3 Solución propuesta ..... 18*

**6 IMPLEMENTACIÓN ..... 20**

6.1 BACKEND ..... 20
*6.1.1 Configuración de entorno ..... 21*
*6.1.2 Loaders ..... 22*
*6.1.3 Carpetas ..... 24*
*6.1.4 Implementación de un proveedor de pagos online ..... 26*
6.2 FRONTEND ..... 30
*6.2.1 Enrutamiento ..... 30*
*6.2.2 Vistas ..... 31*
*6.2.3 Módulos ..... 38*

*6.2.4 Componentes* ... *40*
*6.2.5 Implementación de Stripe en el Frontend* ... *40*

**7 DESPLIEGUE, PRUEBAS E IDENTIFICACIÓN Y CORRECCIÓN DE ERRORES EN TIEMPO REAL** ... **41**

7.1 BACKEND ... 41
*7.1.1 Configuración del entorno de prueba* ... *41*
*7.1.2 Pruebas e integración continua* ... *42*
*7.1.3 Despliegue mediante Github Actions* ... *46*
7.2 FRONTEND ... 50
7.3 DETECCIÓN Y CORRECCIÓN DE ERRORES EN TIEMPO REAL ... 54
*7.3.1 Sentry* ... *54*
*7.3.2 Datadog* ... *57*

**8 ANÁLISIS DE LA METODOLOGÍA USADA EN EL PROYECTO** ... **59**

**9 CONCLUSIONES** ... **61**

9.1 RELACIÓN PROYECTO-ESTUDIOS CURSADOS ... 61

**10 TRABAJO FUTURO Y LÍNEAS DE MEJORA** ... **63**

**11 REFERENCIAS** ... **65**

**12 ANEXOS** ... **67**

12.1 ANEXO A: PRUEBA DE CONCEPTO DE UNA ARQUITECTURA HEXAGONAL ... 67
12.2 ANEXO B: OBJETIVOS DE DESARROLLO SOSTENIBLE ... 69

**GLOSARIO** ... **71**

# Índice de figuras


FIGURA 1: INTERFAZ PRINCIPAL CLICK&BOAT 4
FIGURA 2: FORMULARIO DE RESERVA BOATAROUND 5
FIGURA 3: RESERVA DE UNA EMBARCACIÓN SAMBOAT 5
FIGURA 4: MARCO DE TRABAJO SCRUM 9
FIGURA 5: DIAGRAMA ENTIDAD-RELACIÓN DEL NEGOCIO DE VELA 11
FIGURA 6: CASOS DE USO USUARIO 12
FIGURA 7: CASOS DE USO CAPITÁN 12
FIGURA 8: CASOS DE USO ADMINISTRADOR 13
FIGURA 9: ARQUITECTURA EN CAPAS EN NODEJS 16
FIGURA 10: ARQUITECTURA POR CAPAS BACKEND DE VELA 17
FIGURA 11: INTERACCIÓN ARQUITECTURA MODELO-VISTA-CONTROLADOR EN EL FRONTEND 18
FIGURA 12: CAMPOS NO COMPARTIDOS ENTRE ENTORNOS 22
FIGURA 13: ESTRUCTURA DE UN TOKEN JWT DE GENERADO POR NUESTRO BACKEND 23
FIGURA 14: LOADER PARA LAS LIBRERÍAS RELATIVAS A EXPRESS Y CARGA DE RUTAS 24
FIGURA 15: CARGA DE NUESTRAS RUTAS PARA NUESTRAS ENTIDADES Y MODELOS DE DOMINO 24
FIGURA 16: SELECCIÓN DE ENTORNO SEGÚN PARÁMETRO ENTORNO 25
FIGURA 17: EJEMPLO DE SERVICIO PARA LA RESERVA DE UN BARCO 26
FIGURA 18: EJEMPLO DE CONTROLADOR PARA LA RESERVA DE UN VIAJE 26
FIGURA 19: URL DEL WEBHOOK DE STRIPE PARA NUESTRO ENTORNO DE PRODUCCIÓN 27
FIGURA 20: EVENTOS REGISTRADOS EN EL WEBHOOK DE STRIPE 28
FIGURA 21: CASOS DE USO SEGÚN EVENTOS ENVIADOS POR STRIPE A NUESTRO WEBHOOK 28
FIGURA 22: CLAVE SECRETA DE FIRMA PROPORCIONADA POR STRIPE PARA VALIDAR SUS LLAMADAS A NUESTRA API (WEBHOOK SECRET) 29
FIGURA 23: SERVICIO PARA VALIDACIÓN DE LA FIRMA DE STRIPE 29
FIGURA 24: CONFIGURACIÓN EXTRA NECESARIA PARA EL ENDPOINT DE STRIPE 29
FIGURA 25: ESTRUCTURA DE CARPETAS PARA EL APARTADO DE VISTAS DE VELA EN EL FRONTEND 31
FIGURA 26: PÁGINA POR DEFECTO FRONTEND VELA – URL / 31
FIGURA 27: INICIO DE SESIÓN DE VELA - URL /AUTH/LOGIN 32
FIGURA 28: PÁGINA PRINCIPAL FRONTEND VELA – URL /APP/HOME 32
FIGURA 29: BÚSQUEDA DE BARCOS PARA ALQUILAR – URL /APP/BOATS 32
FIGURA 30: BÚSQUEDA DE BARCOS CON FILTRO - URL /APP/BOATS 33
FIGURA 31: INFORMACIÓN SOBRE UN BARCO - URL APP/BOATS/:ID 33
FIGURA 32: BÚSQUEDA DE BARCOS POR UN TIPO DE EMBARCACIÓN 34
FIGURA 33: INFORMACIÓN SOBRE UNA EMBARCACIÓN – URL /APP/BOATS/:ID 34
FIGURA 34: FORMULARIO DE PAGO DE LA EMBARCACIÓN 35
FIGURA 35: RESERVA DE LA EMBARCACIÓN REALIZADA – URL /APP/PROFILE 35
FIGURA 36: CASO DE USO AÑADIR RESERVA A GOOGLE CALENDAR 36
FIGURA 37: RECIBO DEL PAGO REALIZADO 36
FIGURA 38: VIAJES DISPONIBLES - URL /APP/TRIPS 37
FIGURA 39: RESERVA DE VIAJE - URL /APP/BOOKING/:ID 37
FIGURA 40: PORTAL DE ADMINISTRACIÓN PARA BARCOS - URL /APP/ADMIN/BOATS 37
FIGURA 41: BARRA DE NAVEGACIÓN DE VELA 38
FIGURA 42: VUEX STORE 38
FIGURA 43: INTERACCIÓN ENTRE LOS COMPONENTES DE VUE Y NUESTRA STORE DE VUEX 39
FIGURA 44: EJEMPLO MÓDULO VUEX DE BARCOS 39
FIGURA 45: CARGA DE MONGODB EN MEMORIA 42
FIGURA 46: TEST UNITARIO DE RESERVA DE UN VIAJE (SERVICIO) 44
FIGURA 47: TESTS DE INTEGRACIÓN RESERVA DE UN BARCO 45

Figura 48: Test de integración - Reserva de un barco con capitán 45
Figura 49: Test de integración - Fallo validación de negocio el barco no se encuentra disponible para esas fechas 46
Figura 50: Workflow para nuestra integración continua para todas las Pull Request abiertas contra master o nuestra rama principal de producción 47
Figura 51: Flujo de trabajo para nuestro CI en todas las Pull Requests contra la rama master 47
Figura 52: Flujo de trabajo en proceso en Github Actions 48
Figura 53: Escenario satisfactorio para nuestro CI en una pull request 48
Figura 54: Despliegue con éxito en ambos entornos después de integrar nuevo código a nuestra rama principal 48
Figura 55: Test end-to-end para el registro de un usuario en Cypress 51
Figura 56: Flujo de trabajo Cypress en ejecución 53
Figura 57: Flujo de trabajo Cypress satisfactorio, despliegues en Vercel completados y no se han detectado claves secretas 53
Figura 58: Flujo de trabajo Cypress en ejecucción en la consola de Github Actions 54
Figura 59: Interfaz gráfica de Cypress para lanzar los tests en nuestro entorno local - Test de Registro de usuario 54
Figura 60: Interfaz Vela API en Sentry 55
Figura 61: Interfaz de un error en Sentry 56
Figura 62: Interfaz de un error de validación de campos en Sentry 56
Figura 63: Interfaz de un error en el componente de perfil de usuario en la parte frontal 57
Figura 64: Interfaz de sentry - Análisis detallado del error 57
Figura 65: Ejemplo panel de Datadog 58
Figura 66: Panel de Trello al finalizar todas las iteraciones 60
Figura 67: Arquitectura Hexagonal para el negocio de Vela 67

# Índice de tablas

TABLA 1: C.U. 1. CREAR UNA CUENTA EN LA APLICACIÓN ..... 13
TABLA 2: C.U. 2. INICIAR SESIÓN COMO USUARIO ..... 13
TABLA 3: C.U. 3. NAVEGAR A LA RUTA DE BARCOS ..... 13
TABLA 4: C.U. 4. CAMBIO DE EMAIL ..... 13
TABLA 5: C.U. 5. ADMINISTRAR ENTIDADES ..... 14
TABLA 6: C.U. 6. CAMBIO DE NOMBRE DE USUARIO ..... 14
TABLA 7: C.U. 7. USO DE LOS FILTROS DISPONIBLES PARA ACOTAR LA BÚSQUEDA DE EMBARCACIONES ..... 14
TABLA 8: C.U. 8. ACCEDER A LA INFORMACIÓN DE UNA EMBARCACIÓN ..... 14
TABLA 9: C.U. 9. RESERVA DE UNA EMBARCACIÓN ..... 14
TABLA 10: C.U. 10. RESERVA DE UN VIAJE ..... 15
TABLA 11: C.U. 11. VISUALIZAR SUS VIAJES Y RESERVAS DE EMBARCACIONES PASADAS ..... 15
TABLA 12: C.U. 12. VISUALIZACIÓN DEL RECIBO DE PAGO ..... 15
TABLA 13: C.U. 13. AÑADIR LA RESERVA DE UNA EMBARCACIÓN A GOOGLE CALENDAR ..... 15
TABLA 14: C.U. 14. ACEPTAR UNA PROPUESTA DE VIAJE ..... 15
TABLA 15: ESTRUCTURA DE CARPETAS EN EL FRONTEND ..... 18
TABLA 16: COMPARACIÓN SQL VS NOSQL ..... 20

# Siglas

**HTML** Lenguaje de marcado de hipertexto (Hyper Text Markup Language)

**DOM** Modelo de objeto de documento (Document Object Model)

**API** Interfaz de programación de aplicaciones (Application Programming Interface)

**REST** transferencia de estado representacional (Representational State Transfer)

**CRUD** Crear, Leer, Actualizar, Eliminar (Create, Read, Update, Delete)

**SQL** Lenguaje de Consulta Estructurada (Structured Query Language)

**NOSQL** No solo SQL (Not only SQL)

**JSON** Notación de objeto de JavaScript (JavaScript Object Notation)

**JS** Javascript

**XML** Lenguaje de marcado extensible (Extensible Markup Language)

**JWT** Token web JSON (JSON Web Token)

**CORS** Intercambio de Recursos de Origen cruzado (Cross-Origin Resource Sharing)

**IP** Protocolo de internet (Internet Protocol)

**UI** Interfaz de usuario (User Interface)

**DDD** Diseño dirigido por el dominio (Domain Driven Design)

**POC** Prueba de concepto (Proof Of Concept)

**YAML** Otro lenguaje de marcas (Yet Another Markup Language)

**SEPA** Zona única de pagos en Euros (Single Euro Payments Area)

# 1 Introducción

En la última década, el auge de los servicios ha transformado la forma en que las personas interactúan con los bienes. El surgimiento de nuevas tecnologías y plataformas en línea ha permitido el crecimiento de nuevos modelos de negocio, como el alquiler por días de embarcaciones, que antes eran inaccesibles para muchas personas. Además, la accesibilidad y la facilidad de uso de estas plataformas han impulsado el crecimiento de una economía colaborativa, donde los usuarios pueden compartir recursos y servicios entre sí.

Aunque la mayoría de sectores se han adoptado a las nuevas tecnologías, pienso que Vela, la aplicación a desarrollar, es la propuesta de valor ideal para crear mayor competencia saludable en el mercado de alquiler de embarcaciones marítimas y de viajes. Se quiere brindar una experiencia única y accesible para los amantes de la navegación y los viajes en el mar.

## 1.1 Motivación

Una de las mayores motivaciones para la realización de este trabajo ha sido poder dar una solución efectiva al desarrollo de una aplicación que contemple el uso de las tecnologías actuales más punteras y servicios a nuestra disposición. Se quiere desarrollar una aplicación que contemple pagos online, un despliegue e integración continuo, automatización de tareas para la reducción de costes, y la posibilidad de dar un entorno de prueba a otros equipos más allá del de tecnología dentro de nuestra empresa.

## 1.2 Objetivos

Se pretende dar una solución efectiva sobre el desarrollo de una aplicación usando tecnologías punteras y metodologías ágiles para modelos de negocio orientado a nuevas empresas emergentes (start-ups) y adquirir mejores conocimientos sobre los temas a tratar. En este trabajo final de grado, se abordarán los siguientes puntos:

- El auge de los servicios y su impacto en el mercado del alquiler por día.
- La aparición de nuevos modelos de negocio y plataformas como servicios en línea en este mercado.
- La exploración de soluciones de software que utilizan tecnologías punteras para una empresa emergente.
- La implementación de soluciones para el desarrollo de aplicaciones que contemplen diferentes entornos de desarrollo.
- La implementación de pagos online dentro de la aplicación usando uno de los mayores proveedores de servicio de pago en el mercado.
- La automatización de despliegue e integración continuos en el desarrollo de aplicaciones, una buena cultura de tests, versionado de la aplicación y la configuración necesaria para un entorno de prueba.
- La detección y corrección de errores en tiempo real en los entornos de producción y desarrollo.
- El uso de metodologías ágiles, estimación de esfuerzos y objetivos a abordar.

## 1.3 Estructura de la memoria

Este trabajo se divide en diversas partes:

- **Capítulo 1:** Establecer la motivación y los objetivos de la propuesta.
- **Capítulo 2:** Establecer un estado del arte para evaluar la idea de negocio realizando un análisis de competidores, sus fortalezas y debilidades, etc…

- **Capítulo 3:** Establecer un marco teórico y antecedentes.
- **Capítulo 4:** Realizar un análisis del problema a abordar
- **Capítulo 5:** Proponer una solución viable de arquitectura y estructura de la aplicación.
- **Capítulo 6:** Solución propuesta e implementación.
- **Capítulo 7:** Realizar un análisis e implementación para la configuración de una integración y despliegue continuo, y la detección de errores en tiempo real.
- **Capítulo 8:** Metodologías usadas en el proyecto.
- **Capítulo 9:** Conclusiones del proyecto.
- **Capítulo 10:** Trabajo futuro y líneas de mejora.
- **Capítulo 11:** Referencias.
- **Capítulo 12:** Anexos con la implementación de una prueba de concepto centrada en arquitectura hexagonal con DDD con un bus de eventos y los objetivos de desarrollo sostenible.

# 2 Estado del arte

## 2.1 Resumen de VELA

Se pretende desarrollar una aplicación llamada **Vela** la cual consta de dos partes:

- Parte frontal donde el usuario podrá interaccionar con la aplicación y sus diferentes casos de uso.
- Parte de atrás (*backend*) donde se ejecutarán los casos de uso con su respectiva lógica de negocio.

En la parte frontal de Vela, los usuarios podrán acceder a una amplia variedad de embarcaciones para alquilar o diferentes viajes en barco disponibles en la aplicación. A través de una interfaz intuitiva y amigable, podrán explorar diferentes opciones, verificar la disponibilidad de una embarcación, filtrar por comunidad autónoma para el alquiler de embarcaciones, realizar reservas de forma rápida y sencilla. Además, contarán con la posibilidad de la búsqueda de viajes con diferentes paradas y destinos que podrán visualizar en la interfaz.

La parte de atrás de Vela se encargará de ejecutar los casos de uso. Aquí se encontrará la lógica de negocio necesaria para gestionar reservas de viajes o de embarcaciones, el procesamiento de los datos del usuario o la gestión por parte de un administrador de las embarcaciones, los usuarios, los viajes, etc…

Vela siempre tratará de implementar los estándares de seguridad más modernos y confiables. La seguridad debe ser una prioridad fundamental para cualquier aplicación ya que debemos proteger la información personal y garantizar transacciones seguras a nuestros usuarios.

## 2.2 Idea de negocio

El objetivo de cualquier idea de negocio es tratar de obtener un beneficio económico, para esto es primordial hacer un estudio de otras posibles soluciones ya en el mercado, identificar que nuestra idea es plausible en el mercado y si tiene un hueco en el.

Para determinar si nuestra idea de negocio tiene viabilidad, es necesario realizar un análisis exhaustivo de la competencia y del público objetivo al que nos dirigimos. A continuación, se detallan los pasos clave para evaluar la idea de negocio de Vela para posteriormente realizar un estudio estratégico:

- **Análisis de la competencia**: Es fundamental investigar y comprender a fondo las soluciones existentes en el mercado que podrían ser similares a Vela.
- **Identificación del público objetivo**: Queremos definir cuál es nuestro público y que necesidades queremos satisfacer.
- **Modelo de negocio**: Queremos definir de qué trata nuestro servicio y que necesita para cumplirse.
- **Evaluación de la propuesta de valor de nuestro modelo de negocio**: Una vez que se haya identificado a la competencia y al público objetivo, debemos analizar si ofrecemos valor al mercado.

## 2.3 Estudio estratégico

El primer paso que se ha dado para la evaluación de la idea de negocio es un estudio estratégico.

### 2.3.1 Análisis de competidores

A continuación, se exponen algunos competidores u otras aplicaciones relacionadas con uno de los objetivos del proyecto.

***Click&Boat***

Se trata de una de las empresas líderes de alquiler de embarcaciones. Trata el modelo de negocio de *Airbnb,* pero usando barcos en su lugar. Da la opción de poder registrar tu embarcación para poder rentabilizarla o alquilar embarcaciones de otras personas por día.

Click&Boat[1] ofrece en alquiler veleros, lanchas, neumáticos, chalanas, catamaranes y motos de agua. Elige la duración del alquiler con total flexibilidad (un día, una semana, ...). Dispone de una aplicación móvil. Se trata de una aplicación muy versátil, con una interfaz muy amigable para el usuario. Tras probarla, la aplicación permite pagos en línea, después de que el propietario de la embarcación haya aceptado la solicitud de reserva. Una vez esto haya sucedido, se disponen de 24h para realizar el pago en línea.

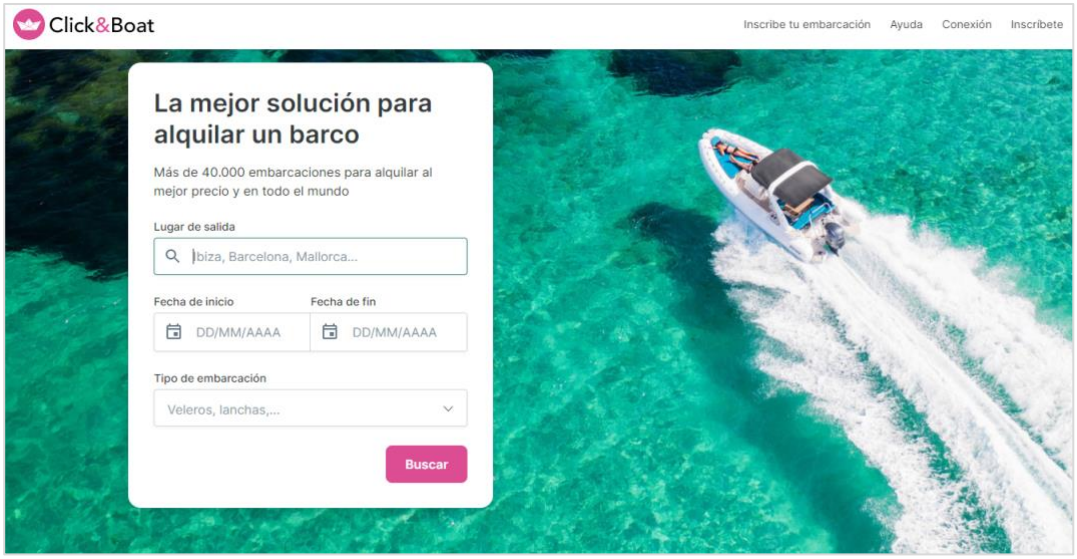


*Figura 1: Interfaz principal Click&Boat*

***BoatAround***

BoatAround[2] es una empresa que usa un modelo de negocio donde predomina la ayuda al cliente, es decir, ofrece posibilidad de cancelación, seguro de depósito, e incluso la posibilidad de poder viajar con una mascota, con un cargo adicional. Todas las reservas están aseguradas por si la embarcación pudiese sufrir cualquier tipo de daño. Sin embargo, no incluye seguros a todo riesgo y solo cubre ciertos daños. En este caso, si se dispone de la posibilidad de reservar automáticamente la embarcación después de rellenar el formulario de reserva.

[1] https://www.clickandboat.com/es/
[2] https://www.boataround.com/es

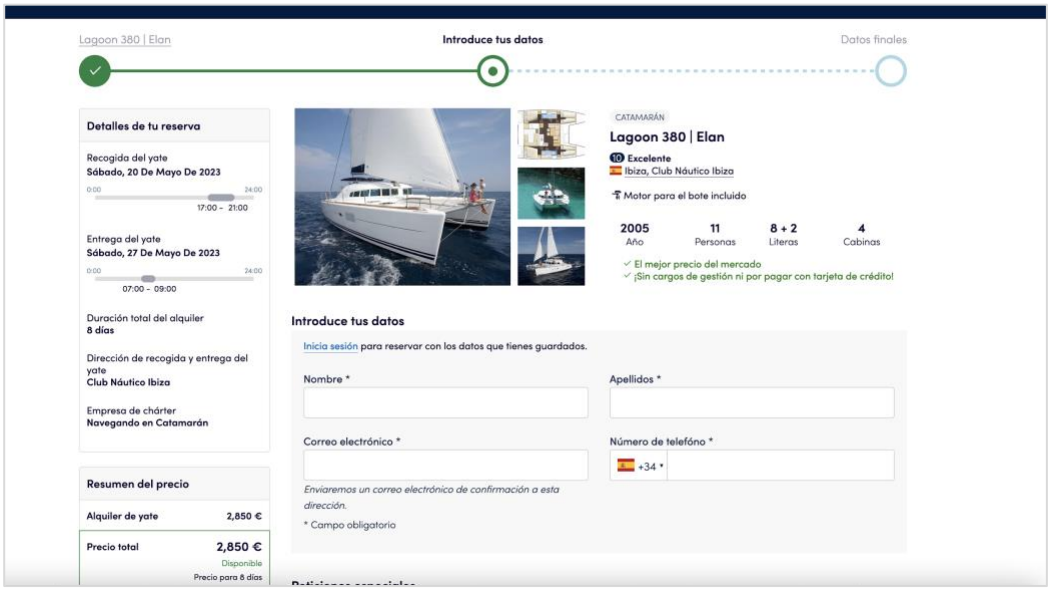


*Figura 2: Formulario de reserva BoatAround*

Esta aplicación usa *Stripe*[3] como librería para realizar pagos online y tiene implementadas diferentes tipos de pagos muy usados en otros países: *iDeal* (plataforma de pago muy usada en Países Bajos), *GiroPay* (plataforma de pago usada en Alemania), entre otras…

### *SamBoat*

SamBoat[4] es una aplicación que sigue el mismo modelo de negocio que las dos anteriores, pero introduce la posibilidad de reservar con un capitán. Cabe destacar que esta aplicación usa la empresa *MangoPay*[5] para la realización de pagos en línea con tarjeta, competencia de Stripe.

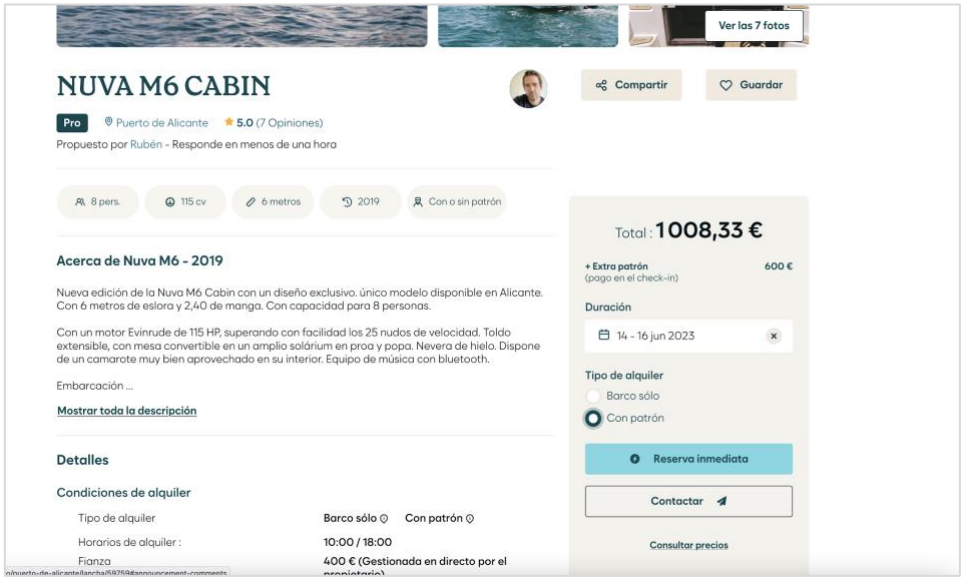


*Figura 3: Reserva de una embarcación SamBoat*

[3] https://stripe.com/
[4] https://www.samboat.es/
[5] https://mangopay.com/

### 2.3.2 Identificación del público objetivo

En el desarrollo de cualquier idea de negocio debemos identificar al público al que nos dirigimos. En nuestro caso, se compone de las personas que tengan interés en el alquiler de barcos por día o viajes por día. Es por esto que definiremos nuestro público objetivo como las personas que incluyan alguna de las siguientes características: turistas y viajeros, familias o grupos, propietarios de embarcaciones, personas que les guste el mar, etc...

### 2.3.3 Modelo de negocio

Vela trata de un SaaS (*Software as a Service*), este modelo nos permite distribuir nuestro producto desde la nube (web y tiendas de aplicaciones) y proporcionarle a nuestros clientes. Este modelo de negocio se ha vuelto muy famoso ya que nos trae algunas ventajas:

- **Escalabilidad**: Permite a las empresas ajustar fácilmente sus servicios según la demanda de clientes, sin necesidad de invertir en infraestructura adicional.
- **Accesibilidad y disponibilidad**: Los usuarios pueden acceder al software SaaS desde cualquier lugar y en cualquier momento, siempre que tengan conexión a Internet.
- **Actualizaciones y mantenimiento simplificados**: El proveedor de servicios se encarga de mantener el software actualizado y funcional, liberando a los clientes de esa responsabilidad.
- **Personalización y configuración**: Aunque los productos SaaS suelen ser estándar, muchos proveedores permiten cierto grado de personalización para adaptarse a las necesidades de cada cliente.

### 2.3.4 Evaluación de la propuesta de valor de nuestro modelo de negocio

Para el desarrollo de una propuesta de valor debemos identificar qué aspectos hacen únicos y valiosos a nuestra aplicación frente a nuestros competidores. Esto implica tanto la funcionalidad como la experiencia de usuario, seguridad, confianza y valor económico, entre otros aspectos…

Vela quiere destacar por la experiencia personal e intuitiva de la aplicación, ofrecer una amplia variedad de embarcaciones, proporcionar seguridad a la hora de realizar reservas, estas se aceptarán automáticamente después de realizar el pago, y ofrecer un servicio de reserva el cual intenta siempre reducir sus precios y costes frente a sus competidores.

# 3 Especificación de requisitos

## 3.1 Introducción

Para una especificación de requisitos necesitamos crear un documento que describa el sistema a hacer, pero no cómo hacerlo. Los requisitos son una condición o capacidad que necesita uno para conseguir un objetivo determinado. La definición de objetivos es vital para un buen análisis y desarrollo del software.

La fase de requisitos se divide en tres grandes fases:

- Definir los requisitos funcionales y no funcionales de software.
- Definir los requisitos de las interfaces software con el resto del sistema.
- Integrar los requisitos en un documento de especificación aparte.

En este documento vamos a hacer un análisis de los requisitos funcionales y no funcionales.

## 3.2 Requisitos funcionales

En este apartado se tiene como objetivo el formalizar las funcionalidades de nuestra aplicación. Se detallan los requisitos funcionales de los usuarios con más permisos en orden creciente, es decir, 4 tipos de roles con distintos privilegios o permisos: Usuarios, Tripulación, Capitán, Administrador.

- Usuarios:
  - Registro e inicio de sesión.
  - Filtro, búsqueda y reserva de barcos.
  - Búsqueda y reserva de viajes.
  - Cambiar nombre de usuario.
  - Cambio de email de una manera segura.
  - Visualizar sus reservas de viajes y barcos futuros o pasados.
  - Crear una ruta en forma de borrador.
  - Guardar reserva en Google Calendar.
  - Mostrar el recibo asociado a la reserva.
  - Dar la posibilidad de convertirse en capitán.
- Tripulación/Capitán
  - Aceptar un viaje para poder llevarlo a cabo.
- Administrador
  - Completas funcionalidades CRUD para nuestro modelo de dominio.

## 3.4 Requisitos no funcionales

Los requisitos no funcionales que especifican criterios que pueden usarse para juzgar la operación de un sistema en lugar de sus comportamientos específicos, ya que estos corresponden a los requisitos funcionales. Se describen algunos para nuestro primer prototipo:

- Interfaz web para móvil y ordenador (interfaz responsiva).
- Retroalimentación al usuario de los tiempos de carga.
- Información al usuario concisa en caso de errores.
- Lectura y carga rápida.
- Fiabilidad en los datos.

# 4 Análisis

## 4.1 Marco teórico y antecedentes

Se pretende desarrollar una aplicación de reservas de barcos y viajes; con el fin de crear una aplicación robusta, segura y escalable. Se trata de una aplicación web destinada al uso a través de navegadores. La aplicación se compone, a grandes rasgos, de un portal para la reserva de viajes y barcos, y un portal de administración, donde los usuarios con permiso podrán hacer las típicas operaciones CRUD en función de sus permisos.

### 4.1.1 Metodología de trabajo

El uso de una metodología de trabajo adaptada a un proyecto determinado es clave para lograr los objetivos que se plantean. Se plantea el desarrollo de la aplicación mediante el uso dos metodologías diferentes:

- **Scrum:** intenta dividir el proyecto mediante *sprints*, que se planifican y revisan de manera continuada.
- **Kanban:** introduce el concepto de tabla para saber el estado actual del punto en el que se encuentran las tareas a realizar de forma rápida y sencilla.

Para dar un mayor contexto de estas dos metodologías, vamos a indagar un poco más de lo que trata cada una.

#### 4.1.1.1 Metodología de trabajo: Scrum

Se trata de una manera para la gestión de proyectos, se establecen entregas de producto progresivas. Este *framework* ágil busca conseguir la máxima rentabilidad por cada iteración de trabajo. Se basa en *sprints*, iteraciones de 1 a 4 semanas para la organización del equipo y la realización de tareas. Se puede definir de la siguiente manera:

1. Se van añadiendo tareas o épicas (una tarea que engloba subtareas usando una metodología de despliegue continuo) en el backlog (parte donde se dejan fijados los requisitos o tareas por parte de producto).
2. Se crea una reunión para la organización del siguiente *sprint*, donde se van a estimar una cantidad de horas, que puede cambiar en función de los días de vacaciones de los integrantes del equipo. Normalmente, se suele dedicar una media de 4h por día trabajado para el desarrollo de tareas. El resto de las horas se estiman que serán para análisis, reuniones, contratiempos etc... Se escogen tareas que completen las horas estimadas y se añaden al *sprint* para poder ser llevadas a cabo. Una vez finalizada la reunión se da comienzo al sprint.
3. Se realizan reuniones diarias para ver el estado de las tareas en curso, dudas a producto sobre alguna definición de una tarea, o cualquier otra cuestión que pueda surgir entre miembros del equipo.
4. Se da por finalizado el *sprint*, se realiza una revisión y retrospectiva donde se expone temas tales como si la estimación ha sido la correcta, sobre si se han estimado bien el nivel de esfuerzos en función de las tareas realizadas, si se han completado todas las tareas, o se ha sobreestimado y no se han podido cumplir todas las estimadas. Se dan una serie de métricas como el porcentaje de horas de las tareas completadas respecto al número de horas estimadas. También, se realiza una revisión interna dentro del equipo donde este expondrán que cosas han podido fallar, cosas a seguir mejorando, cosas a seguir realizando, cosas que

deberíamos de dejar de hacer, etc… Una aplicación muy versátil para la realización de la retrospectiva es *MetroRetro*[6].

5. Se da comienzo al siguiente *sprint* empezando de nuevo por el **punto 1**, donde también se tienen en cuenta las tareas que no hayan podido ser finalizadas en el anterior *sprint*.


*Figura 4: Marco de trabajo Scrum*

#### 4.1.1.2 Metodología de trabajo: Kanban

Esta estrategia se basa en la elaboración de una tabla con, como mínimo, tres columnas; pendiente de iniciar progreso, en progreso, terminada. Podríamos añadir diferentes columnas tales como, pendientes de revisión o pendientes de despliegue. Estas tareas deben estar al alcance de todos los miembros del equipo para evitar la repetición de ellas. Esta estrategia proporciona una mejora en la productividad y eficiencia de todo el equipo de trabajo.

#### 4.1.1.3 Metodología a usar en el proyecto

Se pretende usar en el desarrollo de este proyecto una metodología que junte lo mejor de Scrum y lo bueno de Kanban. Se definirán primeramente las tareas a realizar en un ***backlog*** (trabajo incompleto o cosas de las que debemos ocuparnos). Se pretenden realizar **6 iteraciones** en forma de *sprints*, y se priorizan las tareas previamente definidas en el *backlog* por iteración. Se intentará medir el nivel de esfuerzo de las tareas a realizar y solamente se escogerán las tareas que se puedan llevar a cabo según las horas de trabajo planteadas para esa iteración.

### 4.1.2 Alcance

Este trabajo se enfoca en el desarrollo de un portal de reservas, que es parte de una aplicación web. Para esto, se utilizan tres componentes principales:

- Un **servidor** que ofrece servicios web al cliente, se conecta y opera con la base de datos.
- Un **cliente** donde el usuario navega y utiliza para obtener información y recursos del servidor, así como para realizar acciones sobre ella.
- Una **base de datos** que contiene toda la información manejada por la aplicación y permite su obtención y modificación por parte del servidor.

[6] https://metroretro.io/

Además, se utilizan dos plataformas como servicio para el despliegue de la aplicación, lo que implica alojamiento y computación en la nube para externalizar la configuración del entorno del servidor. Se necesita un servidor y una base de datos por cada entorno que vamos a desarrollar; en nuestro caso: **entorno de producción, entorno de desarrollo y entorno local**. El entorno de producción será donde se ejecuta la aplicación en vivo y está disponible para los clientes finales. Es el entorno que debe ser altamente confiable, escalable, con integridad de datos y seguro. Por otro lado, el entorno de desarrollo es utilizado por los desarrolladores u otro equipo relativo a nuestra empresa para trabajar en nuevas funcionalidades o realizar pruebas. Proporciona un ambiente controlado y un servicio para otros equipos más allá del de tecnología. Y, por último, el entorno local será el entorno en el cual cada desarrollador tenga persistido sus propios datos y pueda colaborar y trabajar de manera individual en el desarrollo de la aplicación.

### 4.1.3 Tecnologías y procedimientos en el proyecto

Para alcanzar los objetivos de una aplicación robusta, escalable y eficiente se pretende crear la aplicación Vela en un stack llamado **MEVN**. Este es uno de los más usados, con mayor soporte y actualizados actualmente. Se compone de las siguientes tecnologías:

- **MongoDB**: base de datos no relacional. [1]
- **Express.js**: framework para recibir peticiones HTTP/S y crear REST APIs. [2]
- **Vue.js**: framework de desarrollo usando virtual DOM en la parte frontal. [3]
- **Node.js**: entorno de tiempo de ejecución de Javascript. [4]

El stack MEVN (MongoDB, Express.js, Vue.js y Node.js) se seleccionó debido a su popularidad, soporte y actualizaciones. Estas tecnologías permiten desarrollar una aplicación web robusta y eficiente, aprovechando las ventajas de una base de datos no relacional, un framework backend flexible y escalable, y un framework frontend ágil, de fácil aprendizaje y de alto rendimiento.

Para conseguir el objetivo del desarrollo de una aplicación en torno a un equipo, se darán soluciones para que estos puedan colaborar de manera eficiente en el desarrollo de la aplicación, se plantearan soluciones de automatización de integración y despliegue contiuo, y como gestionar la resolución de errores que puedan ocurrir en un entorno de producción.

La elección de tres entornos de desarrollo (producción, desarrollo y entorno local) se basa en la necesidad de tener un entorno controlado para el desarrollo y pruebas de nuevas funcionalidades, así como un entorno de producción confiable y seguro para los usuarios. Esto facilita el trabajo colaborativo del equipo de desarrollo y garantiza la integridad de los datos.

La solución técnica propuesta con modelos de desarrollo con integración y despliegue continuo buscan optimizar el proceso de desarrollo, facilitar la detección y resolución de errores, y garantizar la entrega eficiente de nuevas funcionalidades y actualizaciones a los usuarios.

En resumen, el desarrollo de esta aplicación usando el stack **MEVN** busca dar una solución escalable, automatizada y eficiente al desarrollo de aplicaciones y busca satisfacer las necesidades de los usuarios en cuanto a actualizaciones constantes. Con esta infraestructura, ofrecemos una plataforma segura, intuitiva y actualizada para nuestros usuarios.

## 4.2 Diagrama entidad-relación

Se presenta a continuación el diagrama entidad-relación.

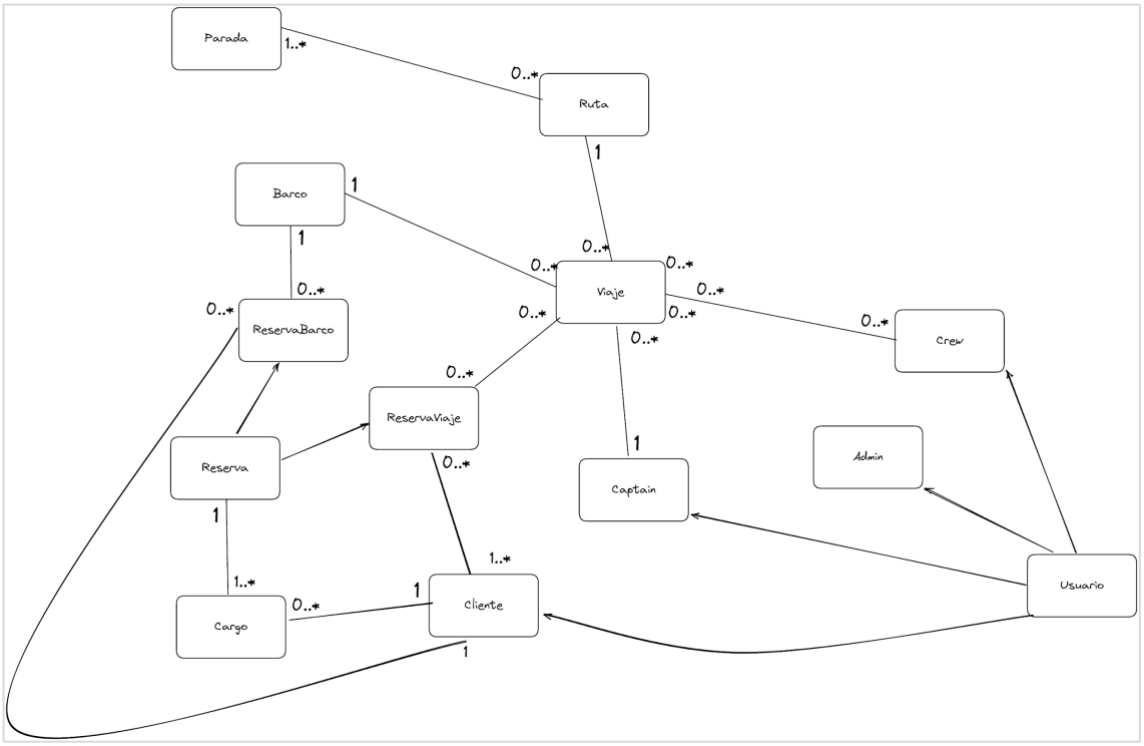


*Figura 5: Diagrama entidad-relación del negocio de Vela*

Como se aprecia, las entidades: Cliente, Capitán, Admin y Crew extienden de Usuario, y las entidades reserva viaje y reserva barco extienden de reserva. Toda reserva debe tener al menos un cargo asociado a ella.

## 4.3 Casos de uso

En este apartado se van a definir los casos de uso.

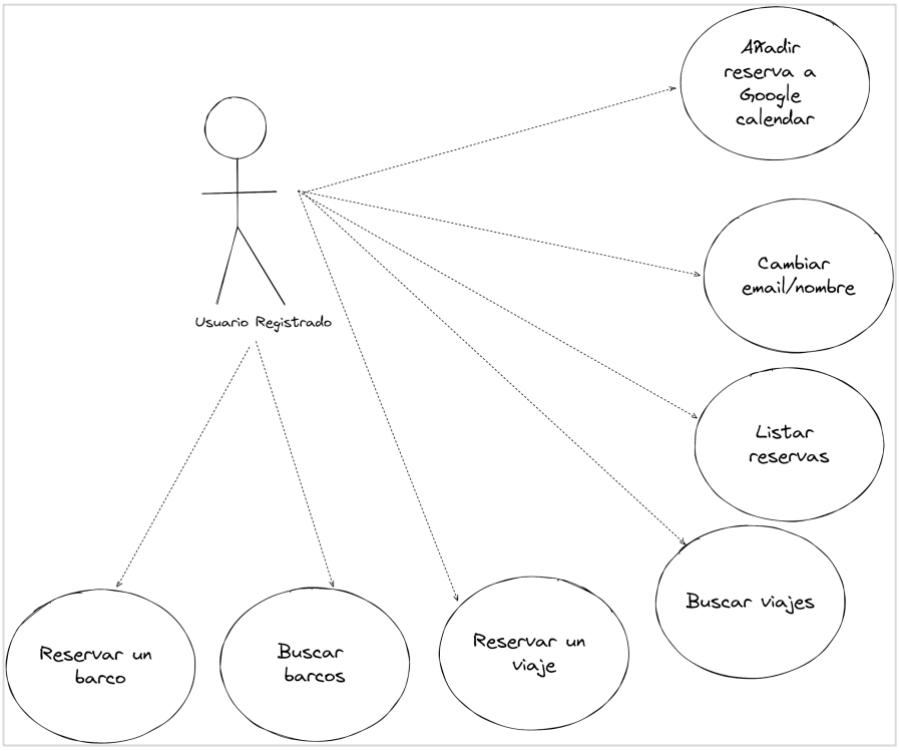


*Figura 6: Casos de uso usuario*

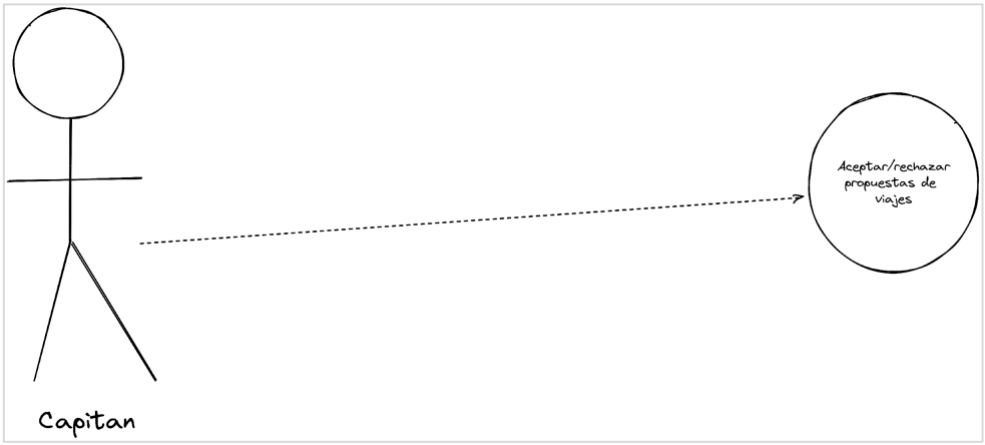


*Figura 7: Casos de uso capitán*

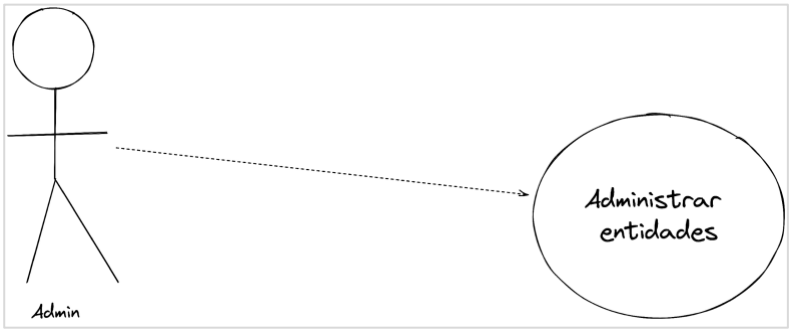


*Figura 8: Casos de uso administrador*

Vamos a señalar algunos casos de uso de forma más detallada a continuación:

**C.U. 1.** Crear una cuenta por parte del usuario

| | |
|---|---|
| Descripción | El usuario podrá crear una cuenta |
| Actor | Usuario |
| Precondición | Estar situado en la ruta de registro y rellenar los campos correspondientes |
| Postcondición | El usuario habrá sido registrado y entrará dentro del portal de reservas. |

*Tabla 1: C.U. 1. Crear una cuenta en la aplicación*

**C.U. 2.** Iniciar sesión como usuario

| | |
|---|---|
| Descripción | El usuario podrá iniciar sesión |
| Actor | Usuario |
| Precondición | Estar situado en la ruta de inicio de sesión y rellenar los campos correspondientes con datos de una cuenta existente. |
| Postcondición | El usuario entrará dentro del portal de reservas. |

*Tabla 2: C.U. 2. Iniciar sesión como usuario*

**C.U. 3.** Navegar a la ruta de barcos

| | |
|---|---|
| Descripción | El usuario podrá visualizar los barcos disponibles. |
| Actor | Usuario |
| Precondición | Estar situado en la ruta de barcos. |
| Postcondición | El usuario podrá visualizar la flota y poder filtrar. |

*Tabla 3: C.U. 3. Navegar a la ruta de barcos*

**C.U. 4.** Cambio de email

| | |
|---|---|
| Descripción | El usuario podrá modificar su email. |
| Actor | Usuario |
| Precondición | Estar situado en la ruta de perfil de usuario. |
| Postcondición | El usuario deberá recibir un email en la nueva dirección para validarla, y posteriormente, abrirla para hacer el cambio efectivo. |

*Tabla 4: C.U. 4. Cambio de email*

**C.U. 5.** Administrar entidades

| | |
|---|---|
| Descripción | El administrador podrá acceder a las rutas bajo admin para poder realizar modificaciones en nuestras entidades y poder visualizar datos, crear nuevas rutas, viajes, barcos, etc… |
| Actor | Admin |
| Precondición | Estar situado sobre alguna ruta para administradores. |
| Postcondición | El administrador podrá ver la tabla con los datos de esa entidad. |

*Tabla 5: C.U. 5. Administrar entidades*

**C.U. 6.** Cambio de nombre de usuario

| | |
|---|---|
| Descripción | El usuario podrá modificar su nombre de usuario en su perfil |
| Actor | Usuario |
| Precondición | Estar situado sobre la ruta de perfil de usuario. |
| Postcondición | El usuario verá reflejado su nuevo nombre en la aplicación con retroalimentación de una alerta de que sus nuevos datos se han persistido en base de datos. |

*Tabla 6: C.U. 6. Cambio de nombre de usuario*

**C.U. 7.** Usar filtros en la búsqueda de embarcaciones

| | |
|---|---|
| Descripción | El usuario podrá usar los filtros disponibles para poder acotar la búsqueda de las embarcaciones a reservar |
| Actor | Usuario |
| Precondición | Estar situado en la ruta de barcos. |
| Postcondición | El usuario verá la nueva consulta realizada con los filtros aplicados. |

*Tabla 7: C.U. 7. Uso de los filtros disponibles para acotar la búsqueda de embarcaciones*

**C.U. 8.** Acceder a la información de una embarcación

| | |
|---|---|
| Descripción | El usuario podrá acceder a la información de un barco para poder ver sus características, disponibilidad y precios |
| Actor | Usuario |
| Precondición | Estar situado en la ruta de barcos. |
| Postcondición | El usuario podrá ver toda la información disponible de la embarcación. |

*Tabla 8: C.U. 8. Acceder a la información de una embarcación*

**C.U. 9.** Reserva de una embarcación

| | |
|---|---|
| Descripción | El usuario podrá realizar la reserva de una embarcación |
| Actor | Usuario |
| Precondición | Estar situado en la ruta de una embarcación disponible y haber rellenado el formulario de reserva y pago. |
| Postcondición | El usuario habrá reservado su embarcación según el formulario rellenado y podrá ver toda la información disponible sobre su reserva en la ruta de su perfil. |

*Tabla 9: C.U. 9. Reserva de una embarcación*

**C.U. 10.** Visualizar la infomracion y reserva de un viaje

| | |
|---|---|
| Descripción | El usuario podrá realizar la reserva de un viaje |
| Actor | Usuario |
| Precondición | Estar situado en la ruta de una viajes, seleccionar un viaje disponible y rellenar el formulario de reserva. |
| Postcondición | El usuario habrá reservado su viaje según el formulario rellenado y podrá ver toda la información disponible sobre su reserva en la ruta de su perfil. |

*Tabla 10: C.U. 10. Reserva de un viaje*

**C.U. 11.** Visualizar sus viajes y reservas de embarcaciones futuras y pasadas

| | |
|---|---|
| Descripción | El usuario podrá ver sus reservas futuras y pasadas tanto de embarcaciones como de viajes. |
| Actor | Usuario |
| Precondición | Estar situado en la ruta de una viajes, seleccionar un viaje disponible y rellenar el formulario de reserva. |
| Postcondición | El usuario podrá ver toda la información disponible sobre su reserva en la ruta de su perfil. |

*Tabla 11: C.U. 11. Visualizar sus viajes y reservas de embarcaciones pasadas*

**C.U. 12.** Visualización del recibo del pago

| | |
|---|---|
| Descripción | El usuario podrá ver el recibo correspondiente a su pago. |
| Actor | Usuario |
| Precondición | Estar situado en la ruta de su perfil y haber realizado una reserva. |
| Postcondición | El usuario podrá ver el recibo correspondiente de la reserva. |

*Tabla 12: C.U. 12. Visualización del recibo de pago*

**C.U. 13.** Añadir la reserva de una embarcación a Google Calendar

| | |
|---|---|
| Descripción | El usuario podrá ver añadir la reserva de su embarcación a Google Calendar. |
| Actor | Usuario |
| Precondición | Estar situado en la ruta de su perfil y haber realizado una reserva de una embarcación. |
| Postcondición | El usuario tendrá una nueva pestaña abierta con los datos correspondientes rellenados para guardar su reserva en Google Calendar. |

*Tabla 13: C.U. 13. Añadir la reserva de una embarcación a Google Calendar*

**C.U. 14.** Aceptar una propuesta de viaje

| | |
|---|---|
| Descripción | El cápitan podrá aceptar o rechazar una propuesta para un viaje |
| Actor | Cápitan |
| Precondición | Estar situado en la ruta de su perfil. |
| Postcondición | El cápitan será parte del viaje aceptado. |

*Tabla 14: C.U. 14. Aceptar una propuesta de viaje*

# 5 Diseño y estructura de la aplicación

## 5.1 Arquitectura por capas en el backend

En este apartado se explica la solución que se implementará para poder realizar un diseño de software que se enfoca en la separación clara y concisa de las responsabilidades del sistema.

### 5.1.1 Intención

La intención de la arquitectura a implementar no es otra más que dar la posibilidad de abstraer capas de nuestro sistema, es decir, que nuestro sistema esté preparado para que sea utilizado por diferentes APIs, interfaces de usuario o servicios web. Al separar las responsabilidades entre capas, y que estas sean independientes, nos da una mayor flexibilidad, mantenibilidad y escalabilidad de nuestro sistema ya que, en un entorno donde tengamos que hacer alguna modificación o nueva implementación en una capa más externa, solo deberemos atacar a esta capa y podrían únicamente verse afectadas las capas más internas a la modificada.

### 5.1.2 Motivación

La arquitectura con capas independientes proporciona flexibilidad, mantenibilidad y escalabilidad al sistema. Permite adaptarse a cambios tecnológicos, solucionar problemas de forma eficiente, escalar por separado y fomentar la reutilización de componentes. Esto garantiza un sistema sólido y eficiente para usuarios y clientes.

### 5.1.3 Solución propuesta

Por lo tanto, y comprendido todo lo anterior, se plantea la siguiente arquitectura:

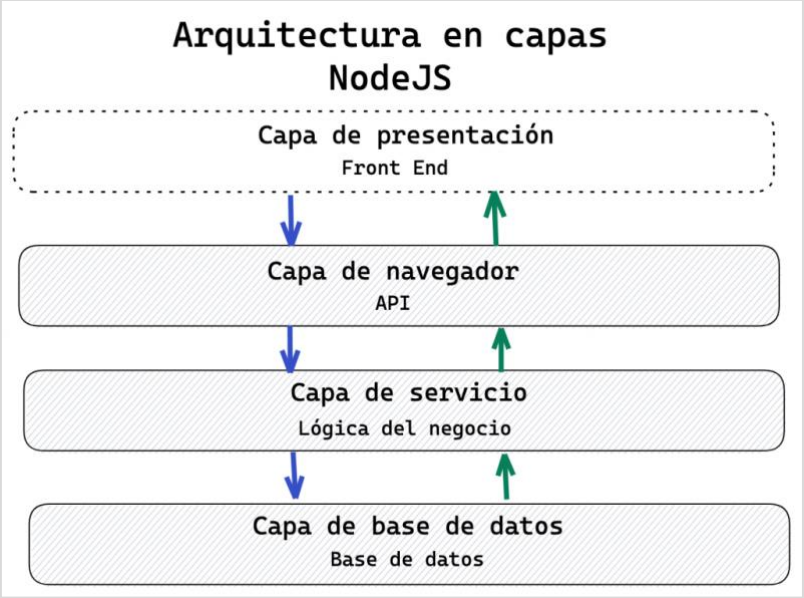


*Figura 9: Arquitectura en capas en NodeJS*

A esta arquitectura la vamos a denominar **arquitectura por capas** [5]. Para el contexto de Vela se ha decidido realizar la siguiente implementación:

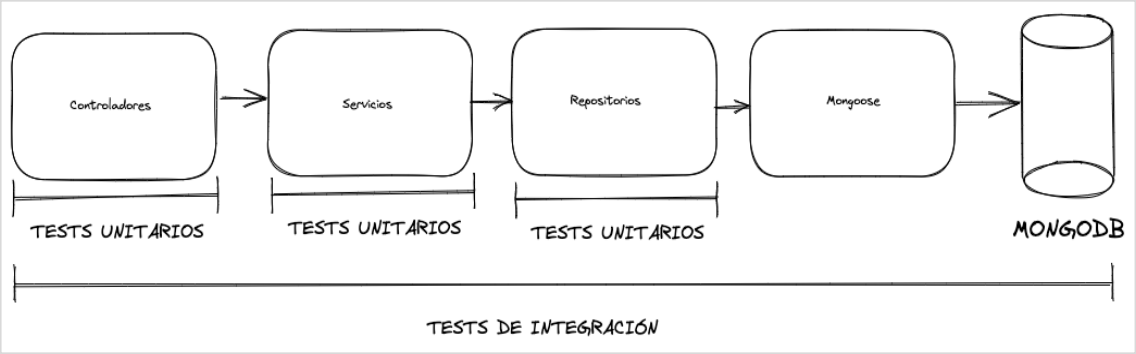


*Figura 10: Arquitectura por capas backend de Vela*

- **Controladores:** Esta capa que se va a encargar de realizar validaciones de nuestra REST API. Es decir, se va a encargar de manejar las llamadas a nuestros endpoints, validar los datos de entrada y enviar los datos recibidos por los servicios.
- **Servicios:** Esta capa se va a encargar de realizar la lógica de negocio y hacer las validaciones o la lógica necesaria para realizar los casos de uso de nuestra aplicación.
- **Repositorios:** Esta capa se va a encargar de realizar las consultas a nuestra base de datos no relacional. Esta capa no tiene que tener lógica de negocio, y debe ser sencilla.
- **Mongoose:** Esta capa[7] es una librería ODM (Object Document Mapper) que nos permite realizar un mapeo de cada colección que tenemos en la base de datos MongoDB a través de esquemas. Los documentos se representan como objetos cuando usamos *mongoose*. Nos facilita el acceso a los datos, a la hora de realizar consultas complejas a la base de datos MongoDB y a estandarizar la estructura de estos mismos datos.
- **MongoDB:** Se trataría de la capa de persistencia de los datos. Aquí vamos a guardar todos los datos de nuestra aplicación.

En particular, comparando la Figura 9 y la Figura 10, la capa de navegador corresponde con nuestra capa de controladores, la capa de lógica de negocio corresponde con nuestra capa de servicios y la capa de base de datos corresponde con nuestra capa repositorios.

Como se aprecia en la Figura 10, cada una de las capas con nuestra implementación debe tener tests unitarios y debemos probar el flujo completo con los tests de integración. Por otro lado, debemos asumir que la capa que nos proporciona *mongoose* es correcta y no tiene fallos. Por norma general, siempre debemos tender a desacoplarse de esta capa lo máximo posible: no debemos realizar implementaciones de métodos en los archivos relativos a los esquemas de nuestras entidades, debemos de minimizar el uso de *virtuals*[8]. Sin embargo, y como todo, en el caso de que nos veamos forzados a realizar implementaciones sobre lo mencionado anteriormente o acoplarnos más a *mongoose*, esto debe estar adecuadamente justificado.

## 5.2 Arquitectura modelo-vista-controlador en el frontend

### 5.2.1 Intención

En la **arquitectura MVC** (modelo-vista-controlador), el modelo se encarga de manejar los datos y la lógica de negocio, la vista se encarga de presentar la información al usuario y el controlador actúa como intermediario entre el modelo y la vista.

[7] https://mongoosejs.com/
[8] https://mongoosejs.com/docs/tutorials/virtuals.html

En Vue.js[9], el modelo se implementa utilizando la instancia de Vue y sus datos reactivos, la vista se implementa utilizando plantillas y componentes, y el controlador se implementa utilizando opciones de componentes y métodos, en nuestro caso será Vuex.

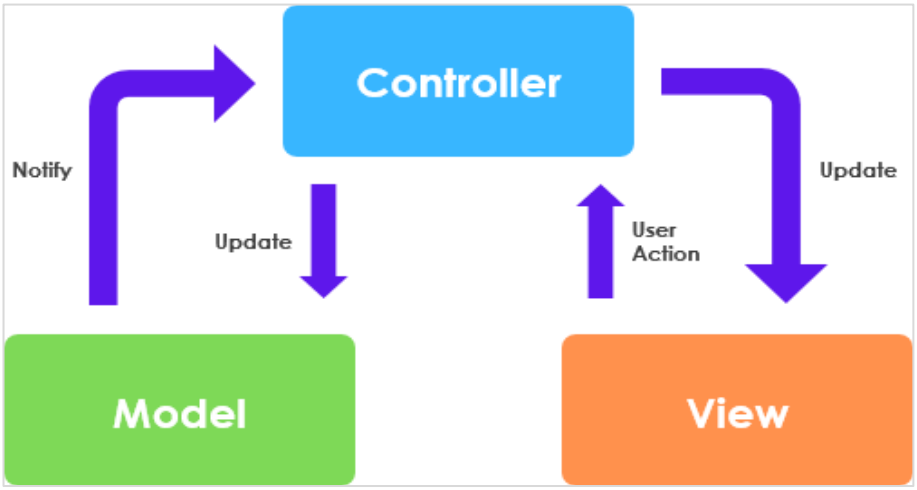


*Figura 11: Interacción arquitectura modelo-vista-controlador en el frontend*

Con el uso de este patrón de diseño, conseguimos la separación clara de responsabilidades, lo que nos da una gran escalabilidad, la posibilidad de reusar código, nos es fácil realizar *mockups* en el caso de los tests *end-to-end*, al promover el código bien estructurado nos facilita el mantenimiento a largo plazo.

### 5.2.2 Motivación

La estructura de la aplicación se divide en tres partes claramente definidas y separadas, lo que permite una mayor modularidad y reutilización de código. Además, esta arquitectura facilita la gestión de cambios y el mantenimiento del código a largo plazo.

### 5.2.3 Solución propuesta

En nuestro caso, vamos a implementar esta arquitectura siguiendo la siguiente estructura de carpetas:

*Tabla 15: Estructura de carpetas en el frontend*

| **Carpeta** | **Función** |
|---|---|
| API | Se encarga de realizar las peticiones HTTP/S a nuestro *backend* o a otros servicios externos. |
| Assets | Se encarga de almacenar imágenes estáticas. |
| Components | Almacena los componentes (o las plantillas) que posteriormente vamos a reusar en nuestras vistas o componentes. |
| Config | Se encarga de guardar claves públicas o de pequeñas configuraciones de nuestra aplicación. |

[9] https://vuejs.org/

| | |
|---|---|
| Data | Datos estáticos que no cambian, como ejemplo, las comunidades autónomas |
| Helpers | Funciones |
| Layouts | Plantillas base que van a ser usadas únicamente por el *router*. |
| Plugins | Definiciones para agregar funcionalidades base a nuestro proyecto. |
| Router | Definición de rutas. Se definen las plantillas y vistas a usar por cada ruta. |
| Sass | Archivos relacionados con el preprocesador de CSS |
| Store | Módulos de nuestra aplicación. |
| Views | Vistas de nuestra aplicación. |
| Cypress | Carpeta relativa a los tests unitarios de componentes y tests de punto a punto (*end-to-end*). |

Con esta estructura, lo que conseguimos es una separación clara de las responsabilidades de cada una de nuestras carpetas. Esto nos beneficia a la hora de reusar código en diferentes puntos de la aplicación y a la hora de hacer cambios en nuestro código.

# 6 Implementación

## 6.1 Backend

El *backend* se basa en el *framework* de express.js[10] bajo el entorno de Node.js. Express.js es un *framework* muy popular para desarrollar *APIs REST*, y tiene infinidad de librerías que ayudan al desarrollo de una aplicación web.

Express funciona mediante capas y *middlewares*. Toda petición HTTP se intentará resolver en función de cómo hayamos estructurado en nuestro código ya que el orden en el que hayamos definido nuestras rutas si importa, se procesan de arriba hacia abajo. La implementación se ha llevado a cabo usando MongoDB[11] como base de datos.

MongoDB es una base de datos no relacional que nos brinda una gran escalabilidad y flexibilidad. Las bases de datos relacionales no se diseñaron para poder hacer frente a la escalabilidad y agilidad que necesitan las aplicaciones modernas, ni para beneficiarse de los sistemas de almacenamiento básicos y de la potencia de proceso que existen hoy en día. A diferencia de una base de datos relacional que se basa en tablas y columnas, MongoDB se basa en colecciones y documentos, un documento puede tener muchos pares de clave-valor, o clave-matriz, o incluso documentos anidados.

A continuación, se expone una tabla sobre la comparativa de una base de datos SQL (base de datos relacional) a una base de datos NoSQL (base de datos no relacional) [6].

*Tabla 16: Comparación SQL vs NoSQL*

| | **Base de datos SQL** | **Base de datos NoSQL** |
|---|---|---|
| **Tipos** | Un tipo (base de datos SQL) con pequeñas variaciones. | Muchos tipos, entre los que se incluyen almacenes de clave-valor, bases de datos de documentos, bases de datos orientadas a columnas y bases de datos de grafos. |
| **Historia sobre el desarrollo** | Se desarrollan a finales de la década de 1970 para respaldar la primera ola de aplicaciones de almacenamiento de datos. | Se desarrollan a finales de la década de 2000 para superar las limitaciones de las bases de datos SQL, sobre todo en lo relativo a escalabilidad, datos multiestructurados, distribución geográfica y sprints de desarrollo ágiles. |
| **Ejemplos** | MySQL, Postgres, Oracle. | MongoDB, Cassandra, HBase, Neo4j. |
| **Modelo de almacenamiento de datos** | Los registros individuales se almacenan en filas de tablas, y cada columna almacena un dato específico sobre ese registro. Los datos relacionados se almacenan en tablas separadas, y luego se | Varía en función del tipo de base de datos. Por ejemplo, los almacenes de clave-valor funcionan de forma parecida a las bases de datos SQL, pero solo tienen dos columnas (clave y valor), con información más compleja, que a veces se almacena como objetos BLOB en las columnas valor. Las |

[10] https://expressjs.com/
[11] https://www.mongodb.com/

| | | |
|---|---|---|
| | combinan cuando se ejecutan consultas más complejas. | bases de datos de documentos evitan por completo el modelo de tablas y filas, almacenando todos los datos relevantes juntos en un único documento en formato JSON, XML u otro formato. |
| **Escalabilidad** | Escalado vertical, es decir, implica la necesidad de incrementar la capacidad de un solo servidor en caso de un aumento en la demanda. Aunque es factible distribuir las bases de datos SQL en varios servidores, esta tarea requiere una participación significativa del equipo de ingeniería y suele resultar en la pérdida de funciones relacionales esenciales como las operaciones JOIN, la integridad de las referencias y las transacciones. | Escalado horizontal, es decir, implica incrementar la capacidad, el encargado de la base de datos simplemente necesita agregar servidores básicos o instancias de la nube. Los datos en la base de datos se distribuyen entre los servidores de acuerdo con las exigencias específicas. |
| **Manipulación de datos** | Lenguaje específico que utiliza instrucciones Select, Insert y Update, p. ej., SELECT fields FROM table WHERE... | A través de APIs orientadas a objetos. |

### 6.1.1 Configuración de entorno

Como se ha comentado previamente en el análisis, vamos a crear una configuración por cada entorno que queremos que nuestra aplicación tenga. En nuestro caso vamos a definir 3 tipos de entornos de desarrollo:

- **Producción**: Entorno que se ejecutará cuando despleguemos la aplicación en internet. Por defecto, este es el entorno de nuestra aplicación real.
- **Desarrollo**: Entorno que usamos para que otros equipos que no pertenezcan al equipo de tecnología pueda realizar pruebas de nuevas funcionalidades, simulación de casos de uso que están dando fallos en producción, etc... Incluimos el entorno local en este apartado también, pero usará una base de datos distinta.
- **Test**: Entorno que usamos cuando lancemos nuestra batería de tests.

En este caso tenemos estas variables que cambian en función del entorno en que lancemos nuestra aplicación.


*Figura 12: Campos no compartidos entre entornos*

Es interesante mencionar que, por norma general, en este apartado aparecen los campos críticos como pueden ser la clave secreta de firma para nuestros tokens JWT, entre otras. Para un mejor manejo y una mayor seguridad, se planteó la implementación de ***Terraform***[12], sin embargo, se llegó finalmente a la conclusión de que tomaría demasiados esfuerzos su implementación.

### 6.1.2 Loaders

Se trata de nuestra primera configuración a la hora de dar servicio por parte de nuestro servidor. Esta estructura es muy útil ya que queremos cargar las librerías principales lo antes posible.

#### *6.1.2.1 MongooseLoader*

Conectamos la base de datos en función de nuestro entorno actual que viene determinado desde nuestro archivo de *config*. Esto cambia en función de cómo lancemos el comando de npm. Tendremos cuatro tipos de DB: base de datos de testing, base de datos local, base de datos de desarrollo y base de datos de producción.

#### *6.1.2.2 PassportLoader*

La librería ***passport*** es un middleware para Express que nos permite la implementación de estrategias de autenticación de una manera rápida, simple y segura.

Este loader [7] es útil para todo lo relativo a las cookies y sesiones que nuestra API va a usar. Aquí definiremos y registramos qué tipo de estrategias podremos usar para autenticar usuarios. En este caso vamos a usar nuestro campo por defecto y único el email del usuario. Hemos definido por defecto dos estrategias que posteriormente usaremos:

- **Estrategia local**: Permite autenticar a los usuarios mediante un email y una contraseña. Para lograr esto, se requiere una *callback* (función que se pasa a otra función como un argumento, que luego se invoca dentro de la función externa) de verificación (*verify*) que reciba estas credenciales y llame a la función "done" para proporcionar la confirmación de autenticación. Para lograr esto usamos la función "*authenticate"*, que a grandes rasgos, se

[12] https://www.ibm.com/topics/terraform

encarga de comprobar si la opción "*limitAttempts*" está habilitada, entre otras validaciones. Si es así, verifica si el usuario ha intentado iniciar sesión demasiado pronto o si ha excedido el número máximo de intentos permitidos. Si se encuentra alguno de estos errores, se llama al *callback* con un error correspondiente.

Luego, la función verifica si el objeto del usuario contiene un valor para el campo "*saltField*". Si no es así, se llama al *callback* con un error *"NoSaltValueStoredError"*.

Si se encuentra un valor de *salt*, se utiliza la función "*pbkdf2*" para generar una versión hash de la contraseña proporcionada por el usuario y se compara con el valor hash almacenado en el objeto del usuario. Si los valores coinciden, el usuario se considera autenticado y se llama al *callback* con el objeto del usuario. De lo contrario, se llama al *callback* con un error "*IncorrectPasswordError*" y se actualiza el número de intentos de inicio de sesión (si "*limitAttempts*" está habilitado). Si el usuario ha excedido el número máximo de intentos, se llama al *callback* con un error "*TooManyAttemptsError*".

En general, la función "*authenticate*" se encarga de varios aspectos de la autenticación de usuarios, como la limitación de intentos de inicio de sesión y la validación de contraseñas.

Almacenamos en nuestra base de datos un *salt* y un *hash* cuando un usuario se registra y cuando inicia sesión, llamamos a la función de autenticar (*authenticate)*.

- **Estrategia JWT**: Una de las autenticaciones basadas en tokens más usadas actualmente. Esta autenticación nos proporciona una seguridad extra ya que, si nuestros tokens por razón ajena a nuestra aplicación se filtrasen, estos expiran y los usuarios deben iniciar sesión de nuevo cada cierto tiempo (establecido por nosotros) para poder renovar su token.

  La estructura de un token JWT generado en la API es:

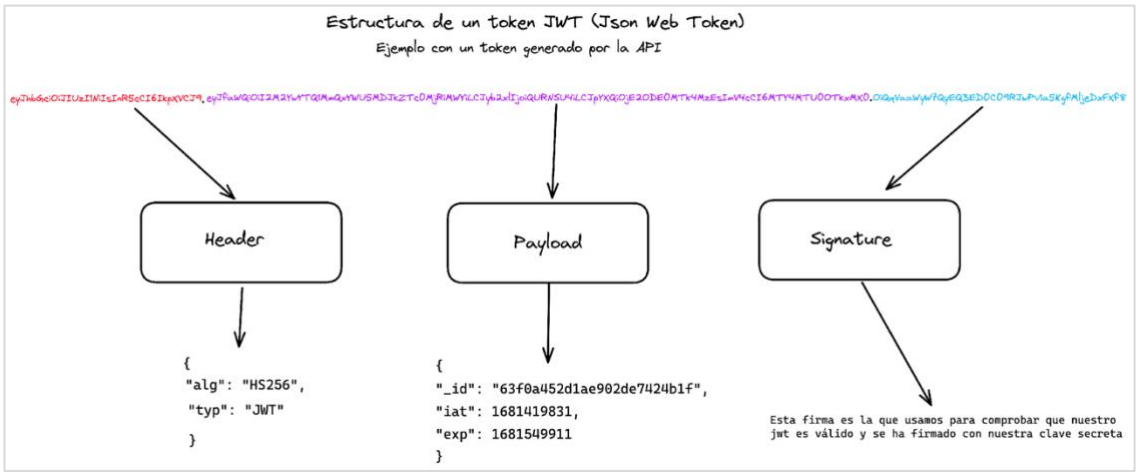


*Figura 13: Estructura de un token JWT de generado por nuestro backend*

#### *6.1.2.3 ExpressLoader*

Aquí definimos las librerías, algunas que nos ayudan para manejar el CORS, el número de peticiones por parte de una IP antes de bloquearla, la encapsulación de la función "*.json()"* para devolver los datos en forma de objetos en formato JSON, etc... Por último, definiremos todas nuestras rutas de la aplicación bajo la ruta */api y* posteriormente el *errorHandler* para poder manejar correctamente errores controlados y no controlados de la aplicación.

```
import morgan from "morgan";   robertprp, 13/12/22, 21:11 • Init
import helmet from "helmet";
import compression from "compression";
import cors from "cors";
import express from "express";
import routes from '../routes/index.js';
import errorHandler from "./errorHandler.js";

export default async ({ app }) => {
    app.use(express.json());
    app.use(express.urlencoded( options: { extended: true }));
    app.use(cors());
    app.use(helmet());
    app.use(compression());
    app.use(morgan( format: 'dev'));
    app.use('/api', routes);
    app.use(errorHandler);
}
```

*Figura 14: Loader para las librerías relativas a express y carga de rutas*

#### *6.1.2.4 SentryLoader*

Aquí vamos a definir todo lo necesario en lo relativo al sistema de monitorización de aplicaciones. Más adelante en el trabajo hablaremos de este sistema.

### 6.1.3 Carpetas

En este apartado definimos la estructura que nuestra API REST va a usar de forma genérica.

#### *6.1.3.1 Rutas*

En nuestro proyecto, tenemos diferentes rutas para los diferentes modelos de dominio o casos de uso que tenemos definidos.

```
import express from "express";
import expressAsyncHandler from "express-async-handler";

import AuthRouter from "../controllers/auth/index.js";   robert
import UserRouter from "../controllers/users/index.js";
import RouteRouter from "../controllers/routes/index.js";
import BoatRouter from "../controllers/boat/index.js";
import TripRouter from "../controllers/trip/index.js";
import ConfigRouter from "../controllers/config/index.js";
import WebhoookRouter from "../controllers/stripe/index.js";

const router = express.Router();

router.use("/config", expressAsyncHandler(ConfigRouter));
router.use("/auth", expressAsyncHandler(AuthRouter));
router.use("/users", expressAsyncHandler(UserRouter));
router.use("/boats", expressAsyncHandler(BoatRouter));
router.use("/trips", expressAsyncHandler(TripRouter));
router.use("/routes", expressAsyncHandler(RouteRouter));
router.use("/webhooks", expressAsyncHandler(WebhoookRouter));

export default router
```

*Figura 15: Carga de nuestras rutas para nuestras entidades y modelos de domino*

Para el acceso a estas rutas se necesitan diferentes permisos y todas están protegidas por autenticación excepto las rutas de inicio de sesión y registro. Se ha usado una función como *middleware* para la comprobación de rol.

En este contexto, si un usuario tiene el rol de "*captain*", se dice que hereda los permisos de los roles "*crew*" y "*user*". Esto significa que el usuario con el rol de "*captain*" tiene todos los permisos y funcionalidades que los usuarios con los roles "*crew*" y "*user*" tienen.

#### *6.1.3.2 Config*

Definimos aquí la configuración de nuestra aplicación que luego usará nuestro frontal, datos que son públicos y constantes, como pueden ser los barcos que actualmente tenemos, las *feature flags* (funcionalidades en proceso de desarrollo que no están públicamente activas), etc…


*Figura 16: Selección de entorno según parámetro ENTORNO*

#### *6.1.3.3 Repositorios*

Se considera la capa de acceso a los datos, desde aquí haremos todas las interacciones a la base de datos. No debemos incluir lógica de negocio en esta capa y debería haber una capa de implementación de nuestro dominio en forma de infraestructura. Por ejemplo, desde esta capa, jamás deberían de lanzarse errores de recurso no encontrado.

#### *6.1.3.4 Servicios*

Se considera la capa que validará la lógica de negocio, crearán los casos de uso correspondientes y validará los datos en función del negocio. Como ejemplo se añade una captura del servicio para hacer una reserva de un barco.

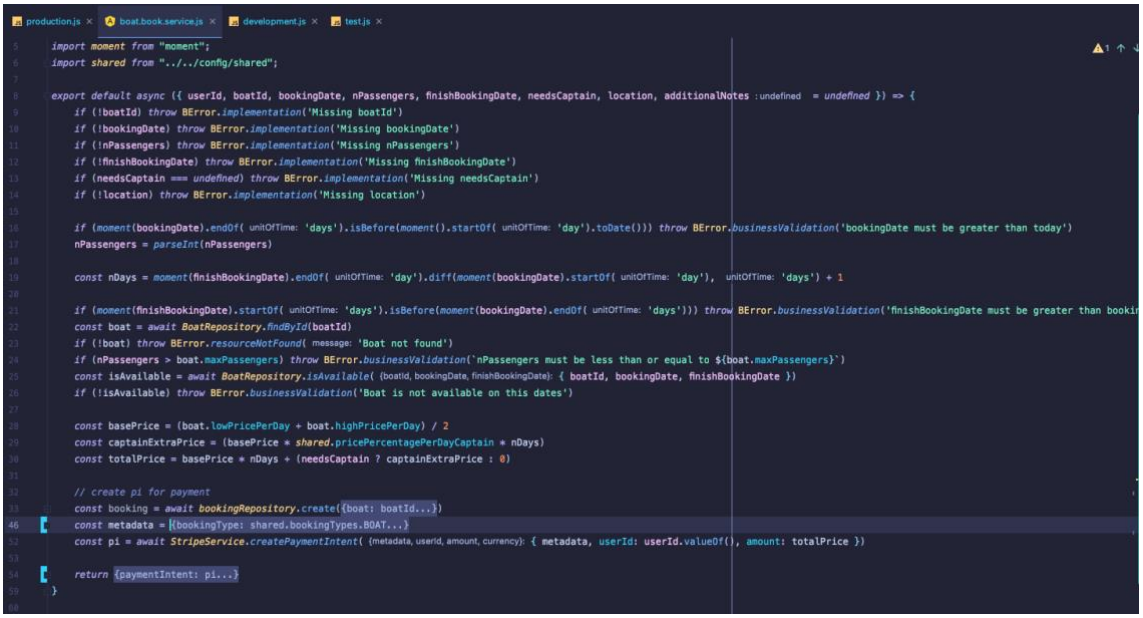


*Figura 17: Ejemplo de servicio para la reserva de un barco*

#### *6.1.3.5 Controladores*

En esta capa se validará todos los datos en las peticiones y se llamarán a los servicios correspondientes según el caso de uso. Este es el punto de entrada para nuestra API REST. Este es el último punto donde definiremos la ruta final para cualquier petición HTTP, cuando se cumpla la condición de la ruta, se ejecutará el caso de uso asociado a esa ruta.


*Figura 18: Ejemplo de controlador para la reserva de un viaje*

### 6.1.4 Implementación de un proveedor de pagos online

En este apartado vamos a hablar sobre la implementación de uno de los proveedores de cobros de software más versátiles y actualizados actualmente. Este, llamado Stripe[13], se trata de una empresa privada tecnológica que se encarga del desarrollo de una infraestructura de software que permite a individuos y empresas recibir pagos por internet proporcionando seguridad y prevención contra el fraude.

[13] https://stripe.com/en-gb-es

Stripe proporciona unan solución sencilla para la implementación de diferentes métodos de pago usados mundialmente. En nuestro caso, solo se va a realizar la implementación de pagos instantáneos con tarjeta, sin guardar datos de esta, a través de webhooks usando la librería para desarrolladores que nos proporciona Stripe[14]. También es posible una implementación sin usar código a través de una pasarela de pago que ofrece este proveedor.

Necesitamos unas claves, una de tipo pública y otra de tipo privada. La clave pública será usada en el frontend y la clave privada será usada en el backend. Inicializamos nuestra librería y exportamos esta constante.

Para hacer un pago instantáneo creamos un objeto llamado *PaymentIntent*[15] donde incluimos algunos datos importantes sobre el cobro [8].

En nuestro caso, vamos a incluir datos como el monto a pagar, la moneda de pago a usar que por defecto siempre usaremos el dólar y el id de cliente creado en stripe. Este, se crea automáticamente cuando un usuario se registra. Este intento de pago va asociado a un único cliente y es el único que puede realizar el pago ya que se genera una clave de cliente secreta para que únicamente el creador de este intento de pago pueda realizar correctamente el cobro. Necesitamos aplicar una multiplicación de 100 a la cantidad total ya que Stripe no trabaja con decimales y únicamente con números enteros, por lo que un dólar con 5 centavos para ellos es una cantidad de 105. Stripe nos permite guardar algunos datos propios para poder categorizar el intento de pago a través de un campo llamado *metadata*.

Posteriormente, esta información se envía al usuario para que pueda usar la clave secreta de cliente para poder realizar el pago usando los datos de su tarjeta. Se hace una petición a la API de Stripe, este validará y lanzará una llamada a nuestro backend para saber que se ha realizado un pago exitoso o fallido. Para esto debemos hacer una configuración mínima en nuestro backend. Primero crearemos el webhook de respuesta para nuestro entorno de producción.

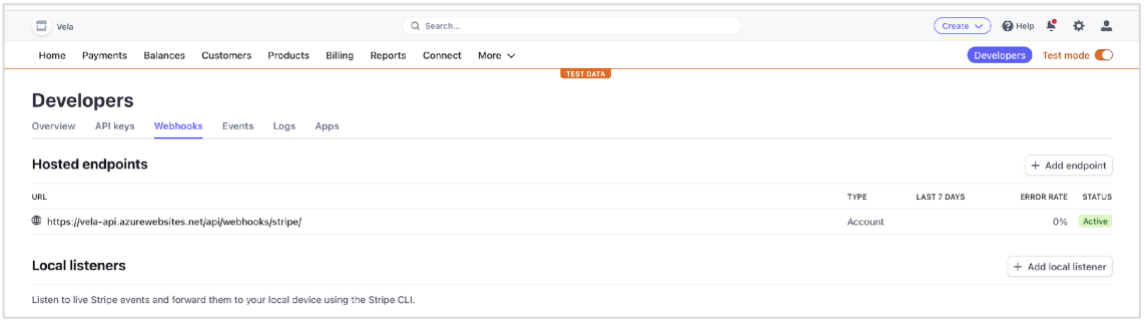


*Figura 19: URL del webhook de stripe para nuestro entorno de producción*

Stripe nos brinda una batería extensa de eventos, pero nosotros registramos los eventos que consideremos importantes para no tener sobrecarga en nuestra aplicación. Solo registramos los siguientes eventos:

[14] https://stripe.com/docs/api
[15] https://stripe.com/docs/api/payment_intents

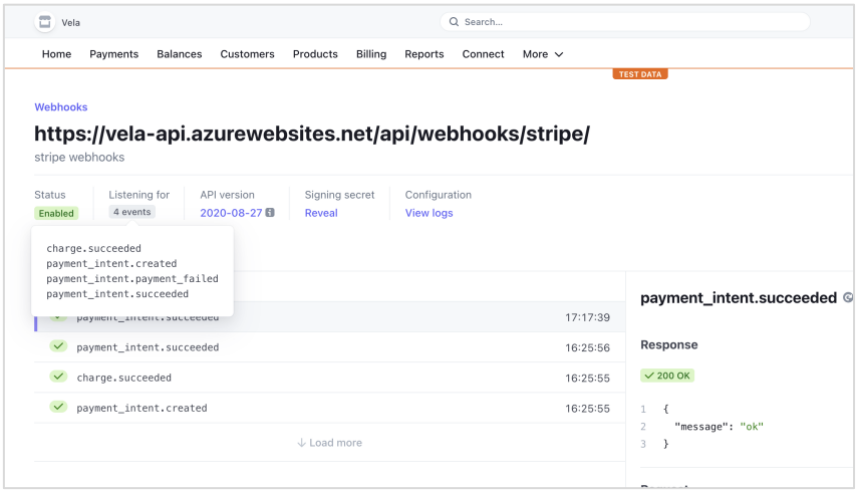


*Figura 20: Eventos registrados en el webhook de stripe*

Y la implementación de este webhook en la API:


*Figura 21: Casos de uso según eventos enviados por stripe a nuestro webhook*

Recibiremos llamadas firmadas con una variable llamada *Signing secret*, que nos servirá para que nuestra aplicación verifique usando nuestra clave secreta si la llamada realmente viene del API de Stripe.

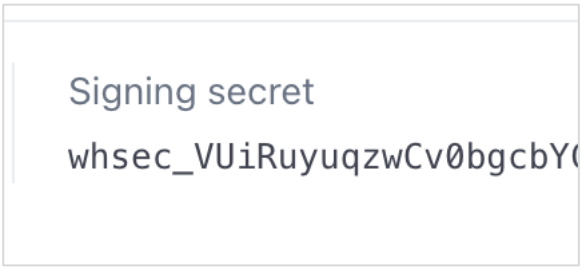


*Figura 22: Clave secreta de firma proporcionada por stripe para validar sus llamadas a nuestra API (webhook secret)*

```
stripe.isValidStripeEventRequest.service.js
import {stripe} from "../../../components/stripe";

export default async (req, res, next) => {
    const stripeSignature = req.headers['stripe-signature'];
    const stripeWebhookSecret = process.env.STRIPE_WEBHOOK_SECRET;
    let event;

    try {
        event = stripe.webhooks.constructEvent(req.rawBody, stripeSignature, stripeWebhookSecret);
    } catch (err) {
        return res.status(400).send(`Webhook Error: ${err.message}`);
    }

    req.event = event;
    next();
}
```

*Figura 23: Servicio para validación de la firma de Stripe*

La documentación para la verificación se encuentra en este enlace[16]. Stripe requiere el cuerpo sin procesar de la solicitud (*rawBody*) para realizar la verificación de la firma. Por lo que al estar usando un framework como *express* que manipula las peticiones antes de procesarlas, necesitamos realizar esta siguiente configuración.

```
app.use(express.json( options: {
    limit: '50mb',
    // Taken from: https://github.com/stripe/stripe-node/issues/341#issuecomment-304733080
    verify: function (req : IncomingMessage , res : ServerResponse , buf : Buffer ) {
        const url = req.originalUrl
        if (url.startsWith('/api/webhooks/stripe')) {   robertprp, 24/5/23, 10:03 • update controller tests
            req.rawBody = buf.toString()
        }
    }}))
```

*Figura 24: Configuración extra necesaria para el endpoint de stripe*

Nuestra API, realizará el caso de uso en función del evento recibido. En este caso solo vamos a ejecutar los eventos de un intento de pago de tipo exitoso o de tipo fallido. Por defecto, stripe envía dos eventos cuando hay un pago exitoso: Intento de pago y cargo exitoso. Se realizan las operaciones de manera asíncrona, entre el cliente y el servidor, es por esto por lo que debemos esperar en el lado del cliente a que nuestra API termine de realizar nuestro la lógica de nuestro caso de uso según el pago (también, recibiremos el ok por parte de stripe al *confirm* que se ha hecho desde la parte frontal).

[16] https://stripe.com/docs/webhooks/signatures

## 6.2 Frontend

El *frontend* se basa en el *framework* de Vue.js[17], uno de los más versátiles y usados junto con React[18] actualmente.

Vue nos ofrece flexibilidad y adaptabilidad a la hora de desarrollar interfaces de usuario. Dependiendo del caso de uso, Vue se puede usar de diferentes maneras:

- Integración como Componentes Web en cualquier página
- Aplicación de página única (SPA), para no realizar recargas en el navegador.
- Posibilidad de SSR, renderizado en el lado del servidor, para mejorar la indexación de nuestra página web en los motores de búsqueda (Google, Bing, etc…).
- Generación de páginas web estáticas.

Para la implementación la aplicación, se han usado diferentes librerías, algunas importantes como:

- **Vuetify**: biblioteca de interfaz de usuario que ofrece componentes de Material Design. [9]
- **Vue-Router**: biblioteca de enrutamiento oficial del lado del cliente que proporciona las herramientas necesarias para asignar los componentes de una aplicación a diferentes rutas de URL del navegador.
- **Vuex**: biblioteca para la gestión de estados globales.
- **Vue Stripe**: componentes relativos al pago online.

### 6.2.1 Enrutamiento

A la hora de declarar rutas, vamos a tender a crearlas con coherencia, es decir, crearemos rutas bajo *admin* que serán rutas protegidas por rol.

Algunas rutas definidas son:

- app/
    - boats/:id
    - trips/:id
    - profile/
    - boats/
    - …
    - admin/
        - users
        - users/:id
        - trips/
        - trips/:id
        - …
- auth/
    - login
    - register

[17] https://vuejs.org/
[18] https://es.react.dev/

Donde el parámetro "*:id"* se trata del identificador de esa entidad en base de datos.

Las rutas dentro de *app/* vienen protegidas por autenticación, para resolver si un usuario tiene los permisos adecuados para esa ruta se verifica primeramente que tenga un token JWT guardado en el almacenamiento local de su navegador y posteriormente, en el caso de administradores, se comprueba si tiene el rol de administrador.

### 6.2.2 Vistas

La implementación de las vistas ha llevado a la siguiente estructura de carpetas:

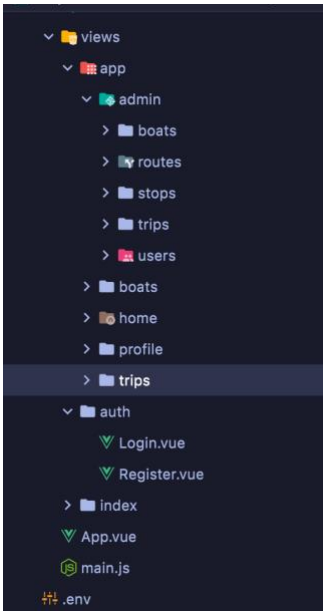


*Figura 25: Estructura de carpetas para el apartado de vistas de Vela en el frontend*

Algunas capturas de la UI de la aplicación:

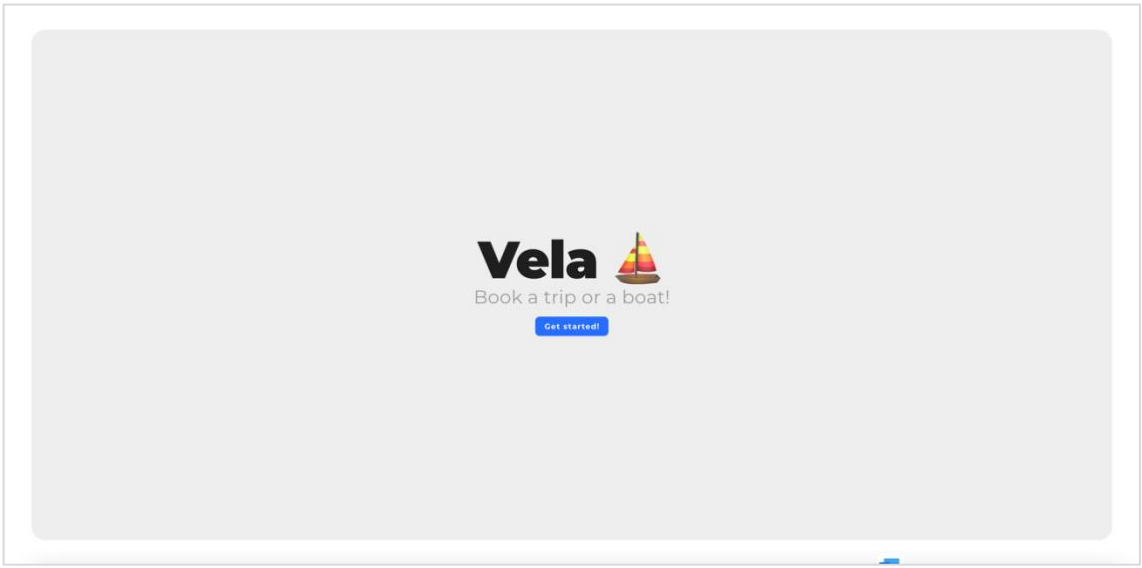


*Figura 26: Página por defecto frontend Vela – URL /*

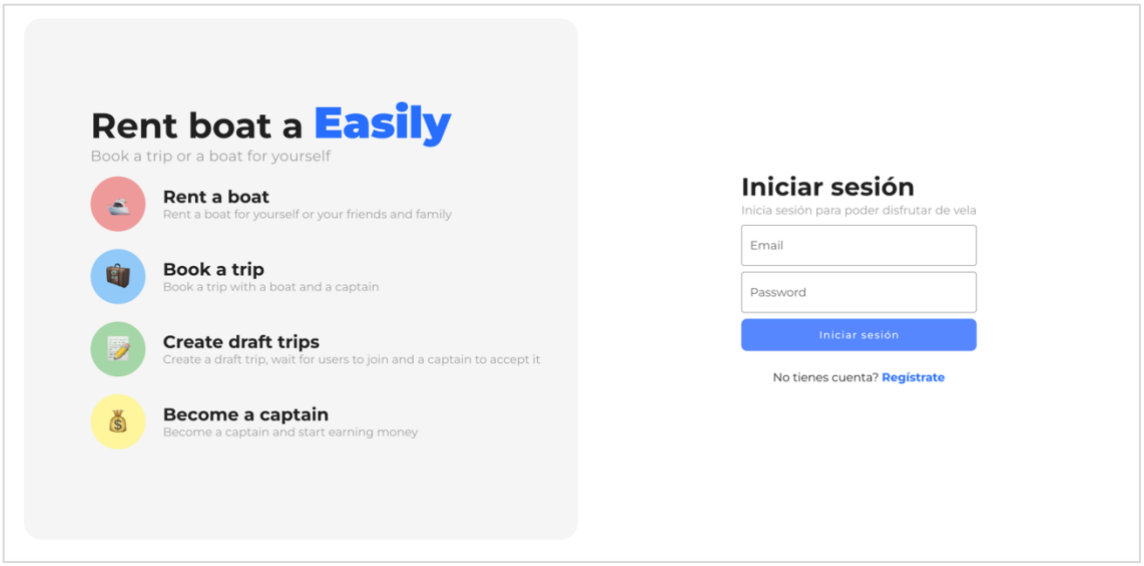


*Figura 27: Inicio de sesión de Vela - URL /auth/login*

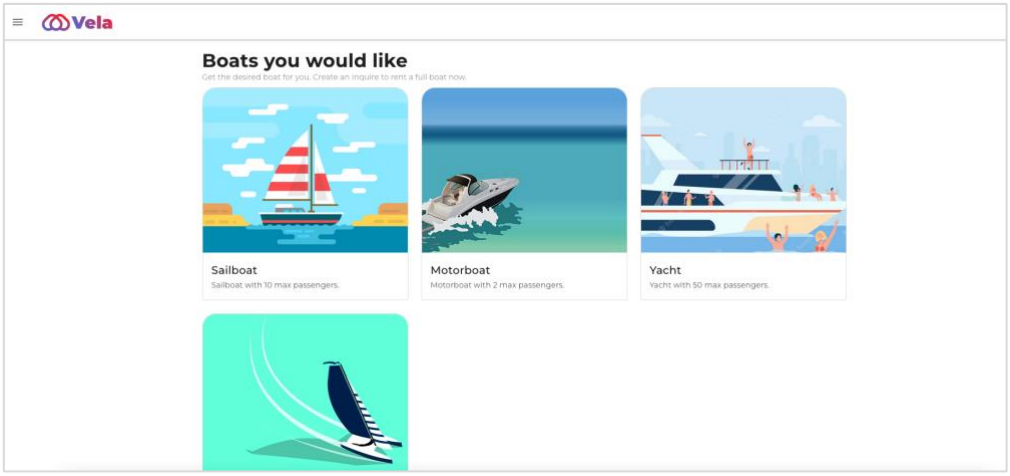


*Figura 28: Página principal frontend Vela – URL /app/home*

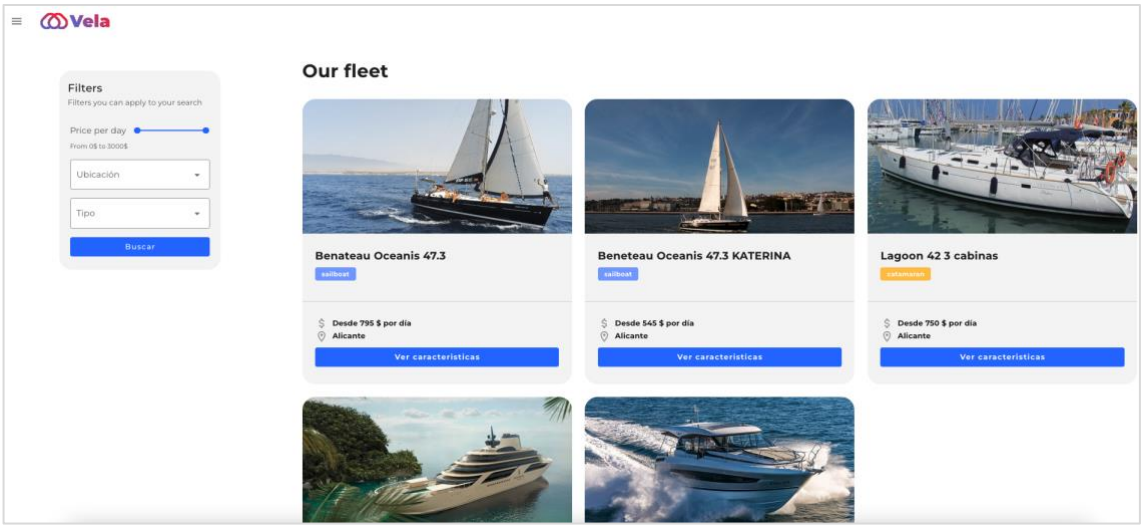


*Figura 29: Búsqueda de barcos para alquilar – URL /app/boats*

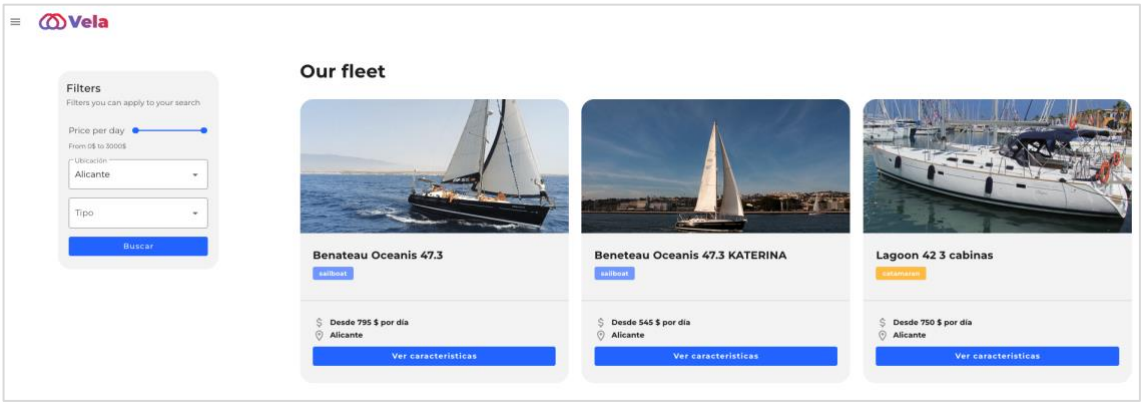


*Figura 30: Búsqueda de barcos con filtro - URL /app/boats*

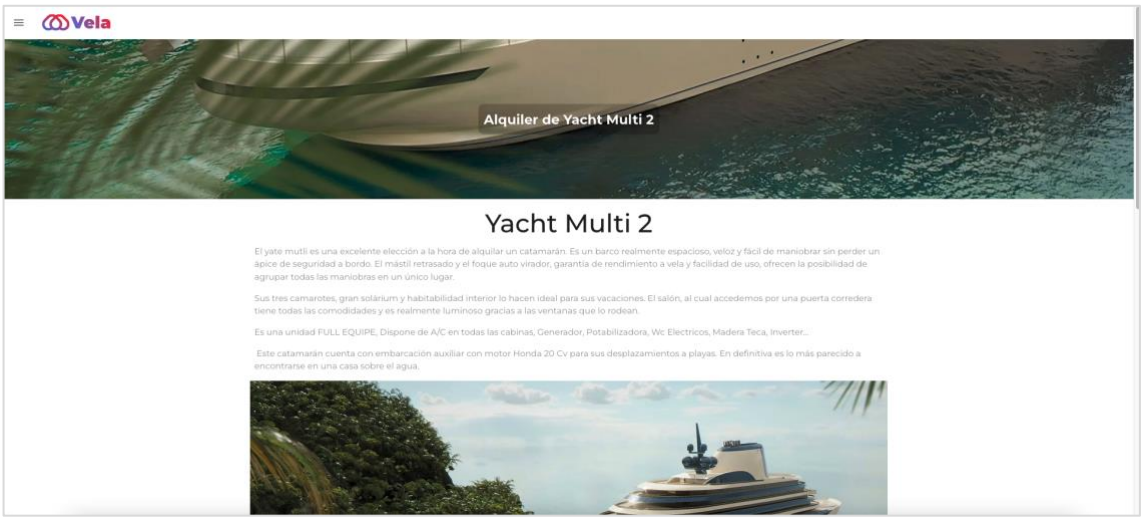


*Figura 31: Información sobre un barco - URL app/boats/:id*

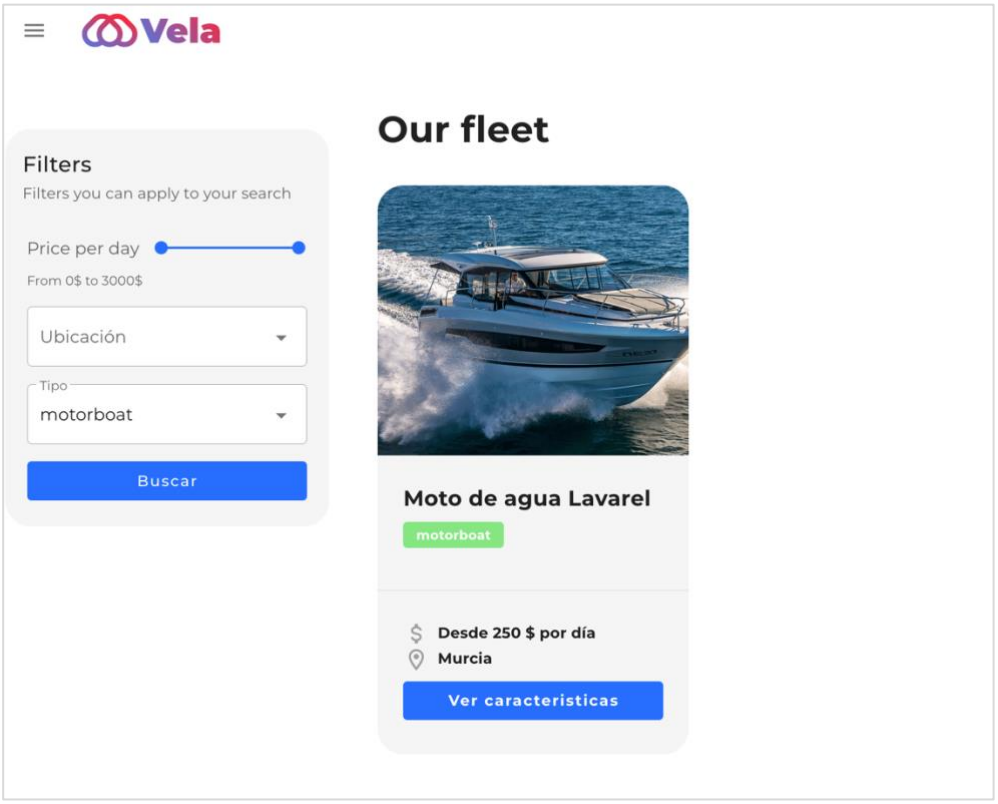


*Figura 32: Búsqueda de barcos por un tipo de embarcación*

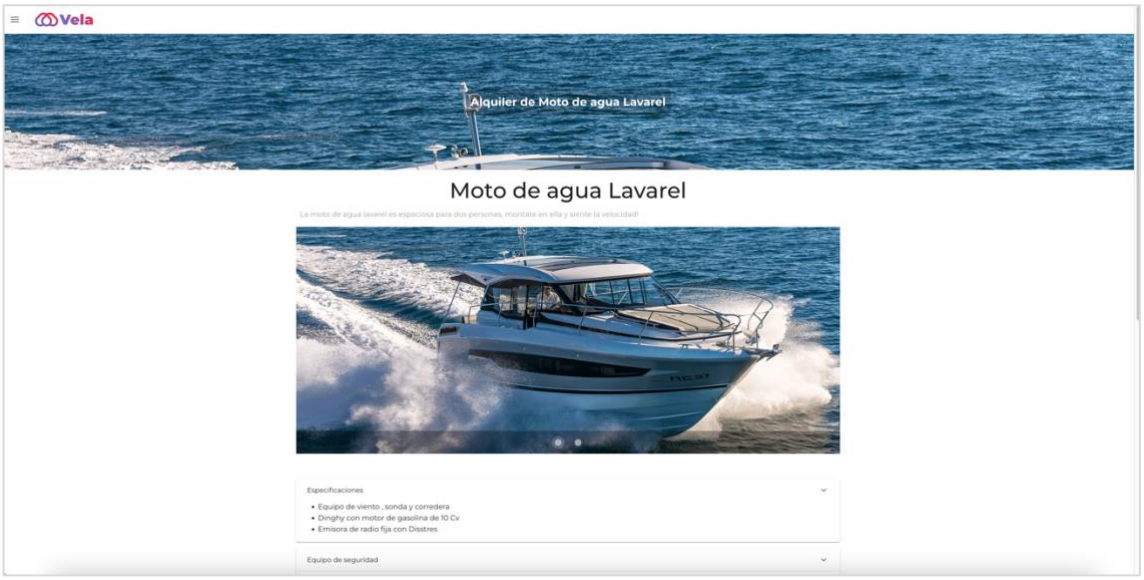


*Figura 33: Información sobre una embarcación – URL /app/boats/:id*

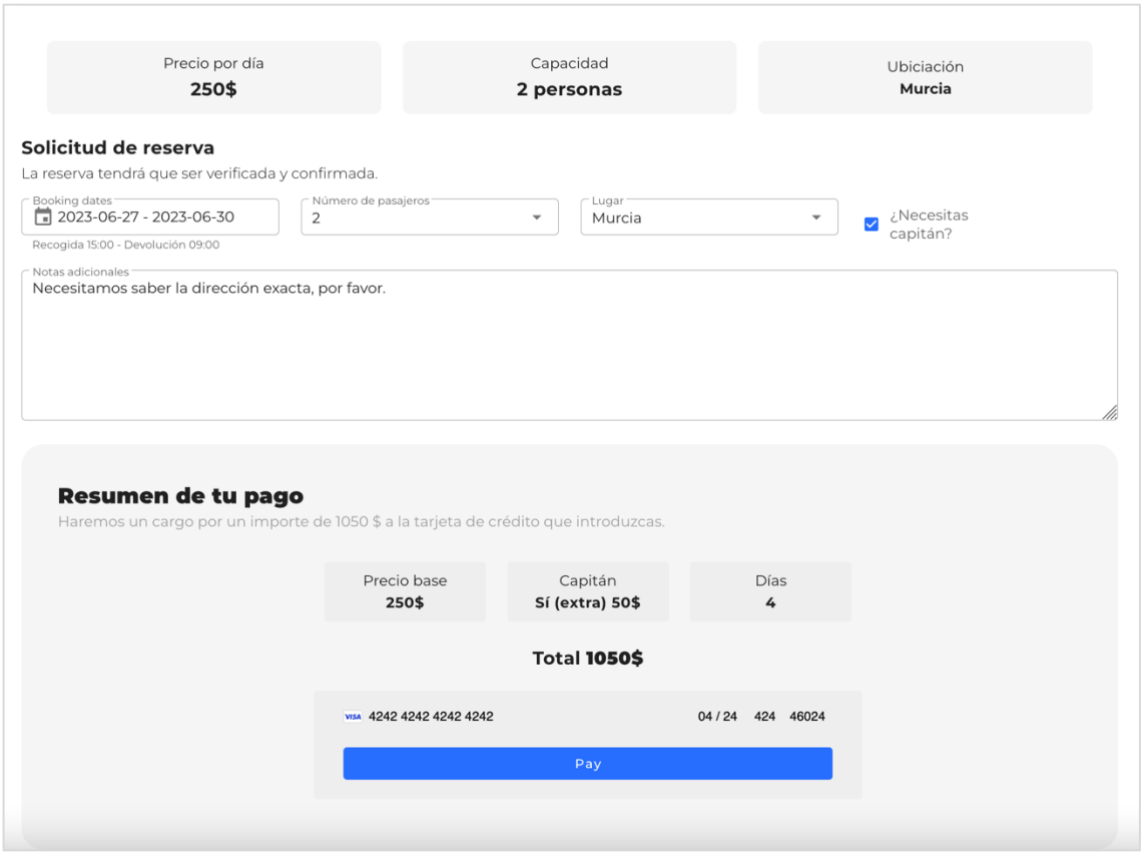


*Figura 34: Formulario de pago de la embarcación*

La tarjeta ***4242 4242 4242 4242*** se trata de una de las tarjetas de testing[19] que nos proporciona stripe para poder hacer un flujo simulando pagos.

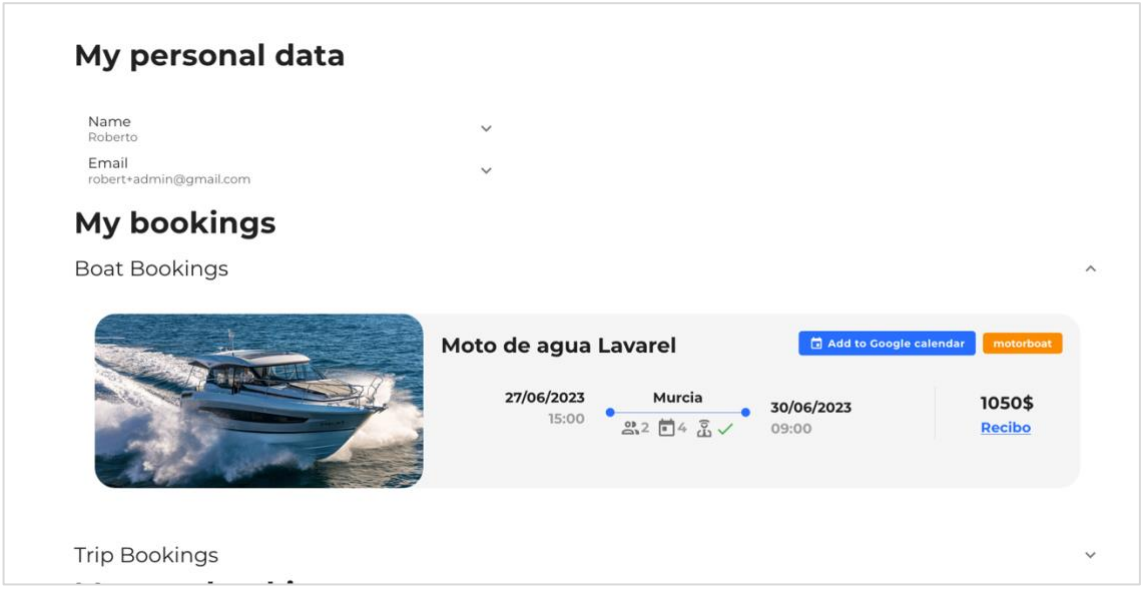


*Figura 35: Reserva de la embarcación realizada – URL /app/profile*

[19] https://stripe.com/docs/testing

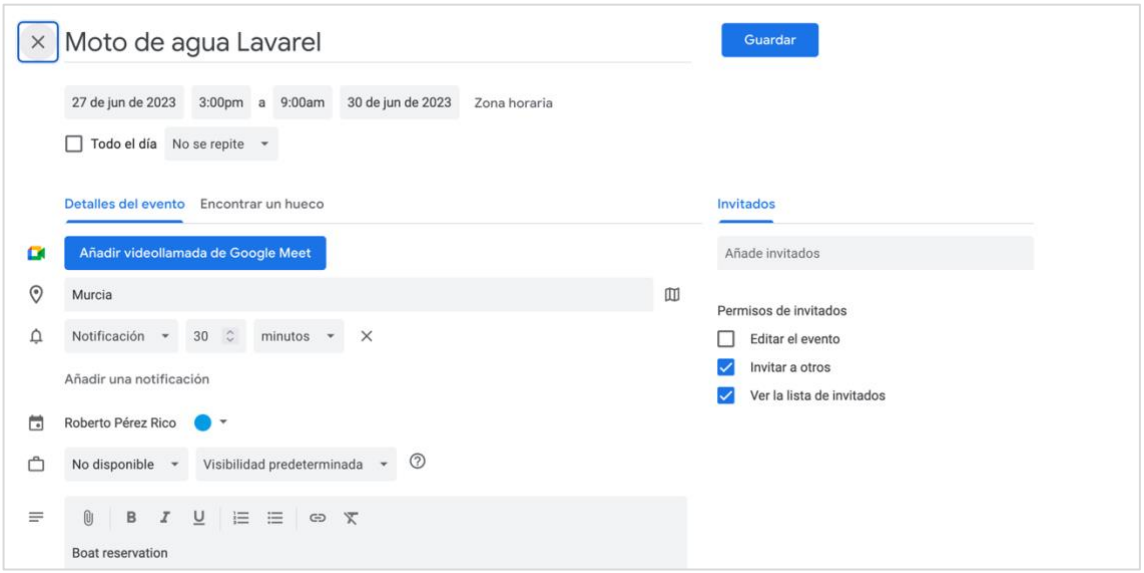


*Figura 36: Caso de uso añadir reserva a Google Calendar*

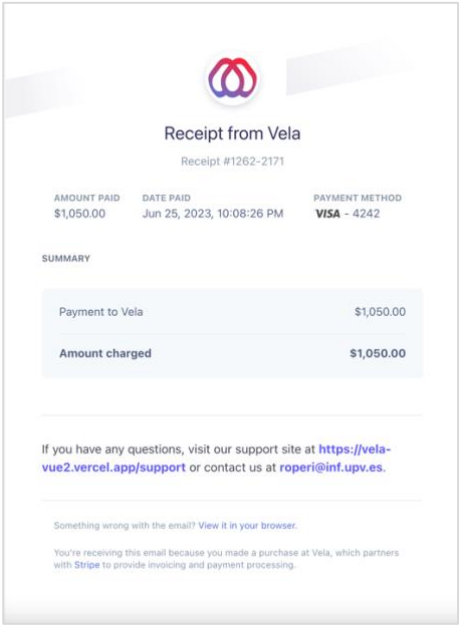


*Figura 37: Recibo del pago realizado*

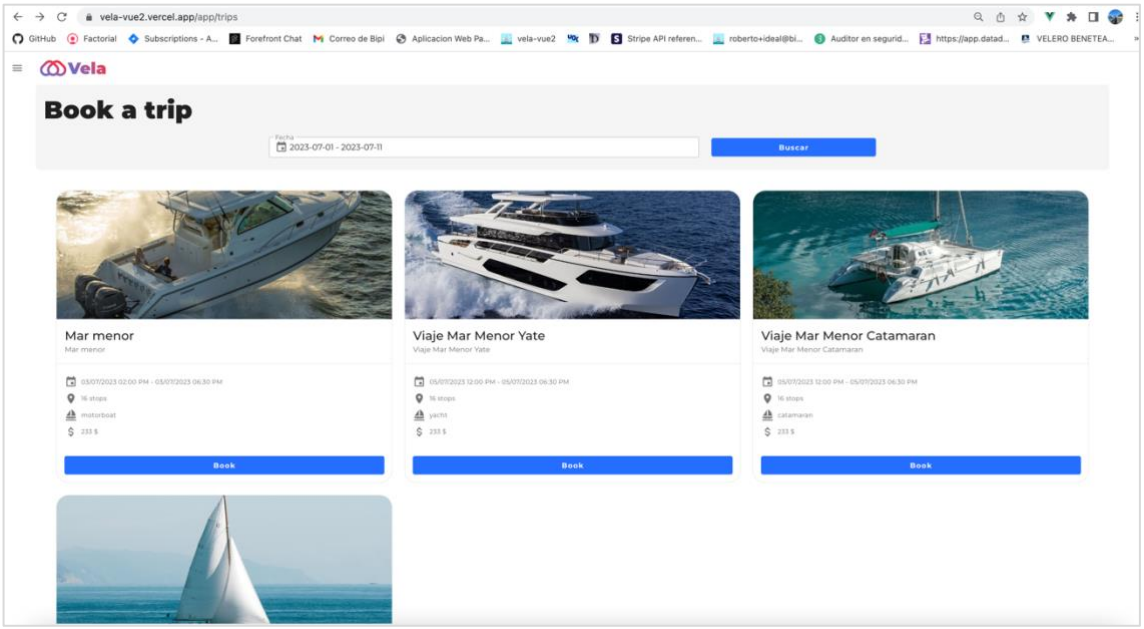


*Figura 38: Viajes disponibles - URL /app/trips*

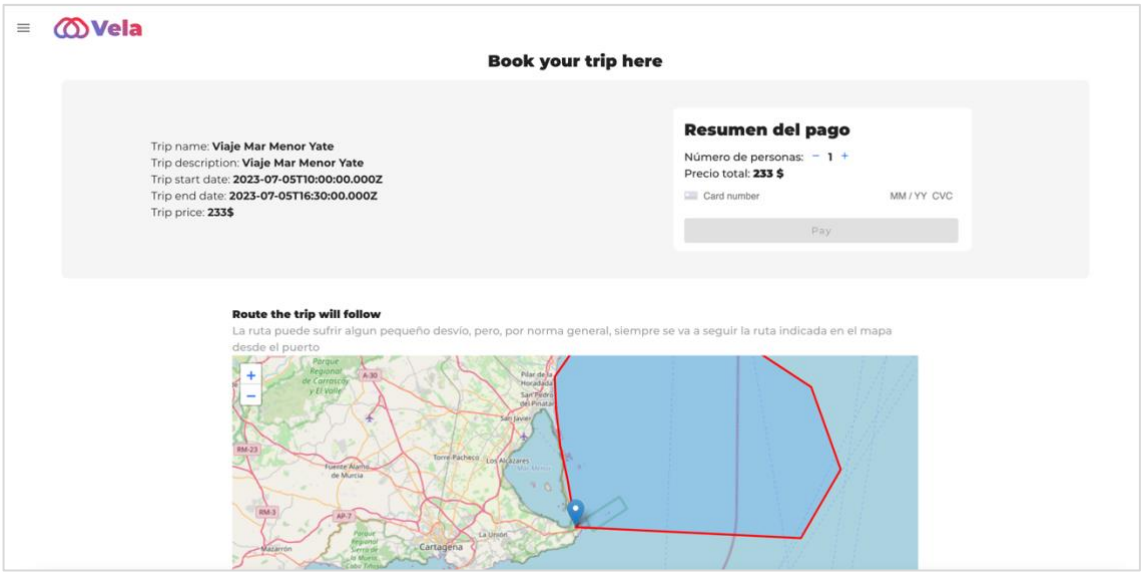


*Figura 39: Reserva de viaje - URL /app/booking/:id*


*Figura 40: Portal de administración para barcos - URL /app/admin/boats*

El administrador posee de 4 rutas diferentes para crear, editar, eliminar o ver las entidades disponibles: usuarios, rutas, barcos y viajes.

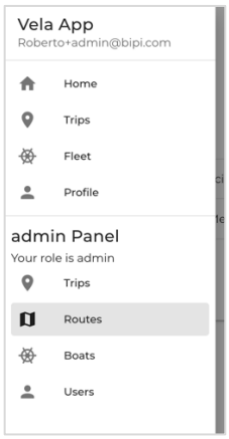


*Figura 41: Barra de navegación de Vela*

### 6.2.3 Módulos

Para el manejo de los estados globales de la aplicación se han usado módulos[20]. Vuex nos permite dividir nuestro estado, mutaciones, acciones y otros métodos en nuevos estados globales.

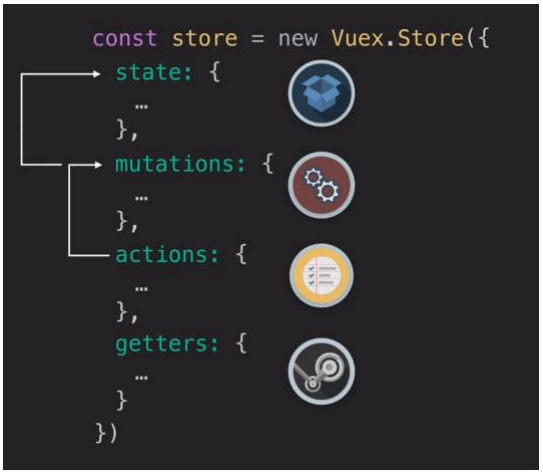


*Figura 42: Vuex Store*

- **State**: Se trata del árbol u objeto inicial de nuestro estado.
- **Mutations**: Desde aquí realizamos el manejo de cambios en el estado u objeto inicial.
- **Actions**: Manejo de interacciones externas con nuestra API u otros servicios. Es una buena práctica llamar a las mutaciones desde las acciones para actualizar los datos de manera directa.
- **Getters**: Acceso rápido con posibilidad de definir pequeñas funciones sobre los datos del estado.

[20] https://vuex.vuejs.org/guide/modules.html

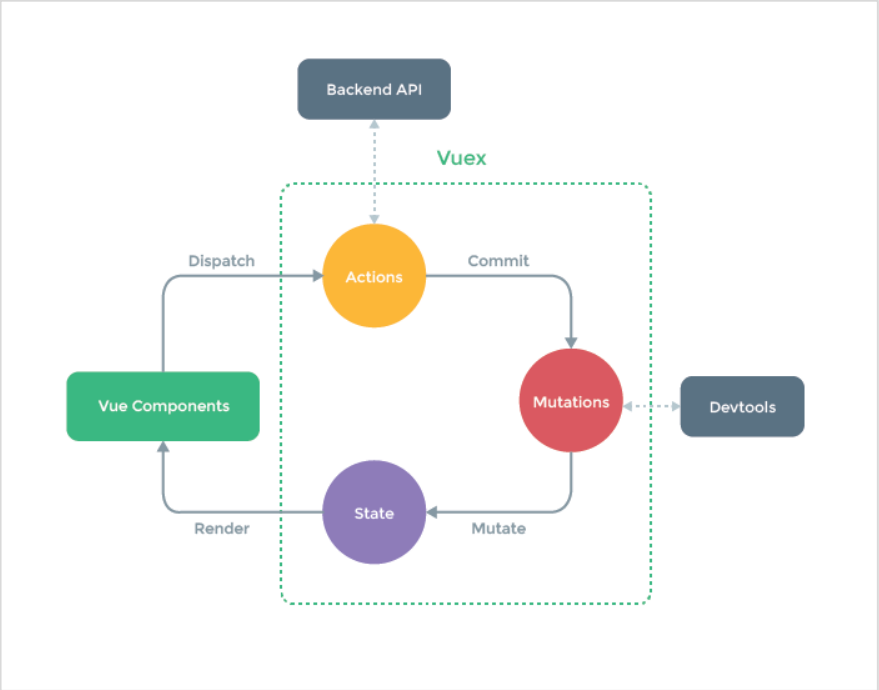


*Figura 43: Interacción entre los componentes de Vue y nuestra store de Vuex*

El uso de Vuex es una muy buena práctica para aplicaciones que escalan en complejidad o aplicaciones a gran escala.

Para Vela se ha subdivido en diferentes módulos según contexto y entidades de nuestra aplicación.

```
const state : {boat: {...}, boatFleet: ...}  = {
    boatFleet: [],
    boat: {},
}

const mutations : {SET_BOAT: function(any, any): void, SET_BOAT_FLEET: function(any, any): void}  = {
    SET_BOAT_FLEET (state, fleet) : void  {...},
    SET_BOAT (state, boat) : void  {...}
}

const actions : {fetchBoat: function({commit: any}, {id: a..., fetchBoatFleet: function({commit: any}, {type:...}  = {
    async fetchBoatFleet ({ commit }, { type, location, price }) : Promise<void>  {...},
    async fetchBoat ({ commit }, { id }) : Promise<void>  {...}
}

const getters : {boatFleet: function(any): []}  = {
    boatFleet: state => state.boatFleet
}

const modules : {admin:  = {
    admin: {
        namespaced: true,
        state: {boat: undefined...},
        mutations: {
            SET_BOATS (state, { boats }) : void  {...}
        },
        actions: {
            async fetchAllBoats ({ commit }, { options }) : Promise<void>  {...},
            async create ({ commit }, data) : Promise<...>  {...}
        },
        getters: {
            results: state => state.boats.results,
            count: state => state.boats.total,
        }
    }
```

*Figura 44: Ejemplo módulo Vuex de barcos*

Como se puede observar en la Figura anterior, un módulo puede tener diferentes submódulos, es decir, se ha definido un módulo interno al contexto de viajes para manejar los estados para la parte de administración de la aplicación.

### 6.2.4 Componentes

Los componentes nos proporcionan la posibilidad de poder reusar nuestro código y crear pequeños elementos HTML que funcionen de manera independiente. Un claro ejemplo que se ha utilizado en el desarrollo de la aplicación para el portal de administración se trata del componente Tabla[21].

### 6.2.5 Implementación de Stripe en el Frontend

Para la implantación de Stripe necesitamos la librería o el SDK (el kit de desarrollo de software) que nos proporciona los elementos UI a inyectar en nuestros componentes. Se ha creado el componente *PaymentIntentForm*[22] para poder manejar de forma global posibles escenarios de pagos.

Emitiremos el evento *pay* cuando Stripe haya validado que todos los campos del formulario son correctos. Esta función llamará a otra acción dentro del módulo de stripe para crear el intento de pago para reservar un barco, por lo que es necesario realizar la llamada a nuestro backend para esto. Posteriormente obtendremos la clave secreta con la cual el cliente debe realizar el *confirm* (una función que nos brinda el SDK para poder realizar la confirmación del cobro por parte del cliente) del intento de pago en el *frontend*. Posteriormente, stripe nos responderá la petición iniciada en el frontend, y nos enviará dos eventos de intento de pago y cargos exitosos al *backend*.

[21] https://gist.github.com/pretended/f4d7cb848e7dbc0268a238b4fa86eabb
[22] https://gist.github.com/pretended/70ee9a36838454d64086c1d634ae6ac2

# 7 Despliegue, pruebas e identificación y corrección de errores en tiempo real

Uno de los puntos más interesantes a la hora de desarrollar una aplicación robusta y confiable se basa en definir buenos métodos para poder distribuir la aplicación a los clientes con frecuencia mediante el uso de automatización en las etapas del desarrollo. Es por eso vital, un proceso de integración y despliegue continuo, el cual llamaremos CI/CD durante este capítulo. [11]

La sigla “CI” se refiere a la integración continua, que es un proceso donde se diseñan, prueban y combinan diferentes tipos de pruebas y nos ayuda a poder detectar errores antes de lanzar a producción a la hora de integrar los nuevos cambios de código.

La sigla “CD” se refiere a la entrega o despliegue continuos, que es un proceso donde se someten a pruebas automáticas de errores y posteriormente, se ejecuta el proceso de despliegue de los cambios para ponerlos disponible a los clientes. El propósito de la distribución y entrega continua es garantizar que la nueva implementación se lleve a cabo con el mínimo esfuerzo mediante automatización.

En este proyecto se ha implementado un workflow CI/CD usando Github Actions. Se va a dividir este punto en dos partes: backend y frontend.

## 7.1 Backend

### 7.1.1 Configuración del entorno de prueba

Para la realización de nuestro entorno de prueba necesitamos hacer una configuración previa y dar un contexto de qué herramientas vamos a usar.

Para la implementación de nuestro CI se han usado algunas librerías tales como:

- ***Mocha***: Se trata de un marco de pruebas muy utilizado en Node.js. Admite pruebas asíncronas, ofrece una variedad de informes y funciona tanto en entornos de desarrollo orientado a pruebas (TDD) como en desarrollo orientado al comportamiento (BDD).
- ***Chai***: Se trata de una librería de aserciones (assertions) y expectativas (expect) que se usa junto con el marco de pruebas Mocha. Proporciona una sintaxis clara y expresiva para realizar afirmaciones sobre el comportamiento y el resultado de la prueba.
- ***Sinon***: Se trata de una librería que tiene como principal función facilitar la creación de objetos ficticios, como stubs (simulaciones) y spies (espías), para simular el comportamiento de dependencias externas al servicio que se está probando. Esto ayuda a aislar las unidades de código que se está probando y controlar su entorno, nos permite realizar pruebas mucho más rápidas y consistentes.

Para lanzar nuestra batería de tests, definimos en nuestro *package.json*, dos scripts:

```
"test": "PORT=8399 ENTORNO=test npx mocha --experimental-modules --es-module-specifier-resolution=node --require mocha.conf.js --timeout 30000 -b --exit \\\"src/**/*.unit.js\\\" \\\"src/**/*.integration.js\\\"",
"test-single": "PORT=8299 ENTORNO=test npx mocha --experimental-modules --es-module-specifier-resolution=node --require mocha.conf.js --timeout 30000 --exit"
```

- ***npm run test***: Este comando lanzará toda la suite de tests que tengan como extensión final *unit.js* o *integration.js.*
- ***npm run test-single <file o regex>:*** Con este comando podemos lanzar simplemente un archivo o un conjunto de archivos según una expresión regular.

Definiremos dos variables globales: **PORT y ENTORNO**, para poder definir los parámetros de entrada de que nuestro entorno de prueba va a usar. En este caso, y al usar la versión de Node.js 16.x.x, podemos usar módulos experimentales, lo que nos permite, la importación de archivos sin usar su extensión o sin incluir el *index.js* en el caso de la carpetas. Definimos un tiempo de ejecución máxima por test de 30000 (ms), un archivo de configuración, y unos parámetros de entrada tales como "*bailing*" para que mocha no reporte automáticamente ante el fallo de una prueba y podamos obtener que otros han podido fallar en nuestro CI, y el parámetro "*exit*" que fuerza a mocha a automáticamente finalizar después de que los tests se completen.

Nuestro CI/CD lanzará toda nuestra batería de tests y comenzará el despliegue en el caso de que todos hayan sido exitosos.

Para cargar nuestra base de datos en memoria necesitamos la siguiente configuración:

```
import { MongoMemoryReplSet } from 'mongodb-memory-server-global'

let mongoServer

async function loadMongoOnMemory () {
    mongoServer = await MongoMemoryReplSet.create({
        replSet: {
            count: 1,
            storageEngine: 'wiredTiger',
            name: 'rs0',
            dbName: 'vela-api-test',
        },
        binary: {
            version: '6.0.4',
        },
    })
    process.env.MONGODB_URI_TEST = mongoServer.getUri()
    return mongoServer.getUri()
}

async function shutDownMongoOnMemory () {
    await mongoServer.stop()
}

/* eslint-disable */
console.log(`Loading mongo on memory`)
await loadMongoOnMemory()
console.log(`Mongo on memory URI: ${process.env.MONGODB_URI_TEST}`)
/* eslint-enable */
```

*Figura 45: Carga de MongoDB en memoria*

### 7.1.2 Pruebas e integración continua

Para llevar a cabo las pruebas, tenemos que verificar el comportamiento de nuestra aplicación sobre una batería de casos de prueba. Estos se orientarán siempre para la búsqueda de errores de software, negocio, implementación... No siempre vamos a poder encontrar todos los errores de nuestra aplicación, pero mediante el uso de técnicas, intentaremos cubrir la mayor cantidad de casos de uso y lógica de esta.

Para encontrar errores, dos técnicas muy usadas son: la técnica de **"caja negra"** y la técnica de **"caja blanca"**.

La técnica de caja negra consiste en crear un flujo sin preocuparnos del comportamiento interno que pueda tener, únicamente centrándonos en las entradas y salidas de dicho flujo.

La técnica de caja blanca, al contrario de las pruebas de caja negra, consiste en verificar el flujo y estructura interna de nuestro código. Estas pruebas necesitan generar todos los caminos lógicos que se puedan producir. Es importante remarcar que, una buena cultura de tests nos proporcionara seguridad y fiabilidad a la hora de lanzar nuestra aplicación a los usuarios.

Hay diferentes tipos de pruebas (unitarias, integración, aceptación, *end-to-end*, …) pero teniendo en cuenta estas, se plantean desarrollar según necesidad de nuestra aplicación los siguientes dos tipos de prueba:

- **Pruebas unitarias**
- **Pruebas de integración**

#### *7.1.2.1 Pruebas unitarias*

Este tipo de tests se basa en la prueba de nuestro servicio como uno solo para poder comprobar su funcionamiento de forma aislada, sin interacción entre otros servicios o repositorios y suelen tratarse de pequeños tests.

Este tipo de pruebas pueden estar asociadas a la técnica de caja blanca ya que probaremos nuestro servicio/repositorio de manera que intentaremos siempre tender a una validación de toda nuestra lógica dentro de la prueba.

Como se ha visto en la Figura 10, todas nuestras capas deberían de estar testeadas unitariamente.

Una prueba unitaria que se ha realizado dentro de nuestro CI:

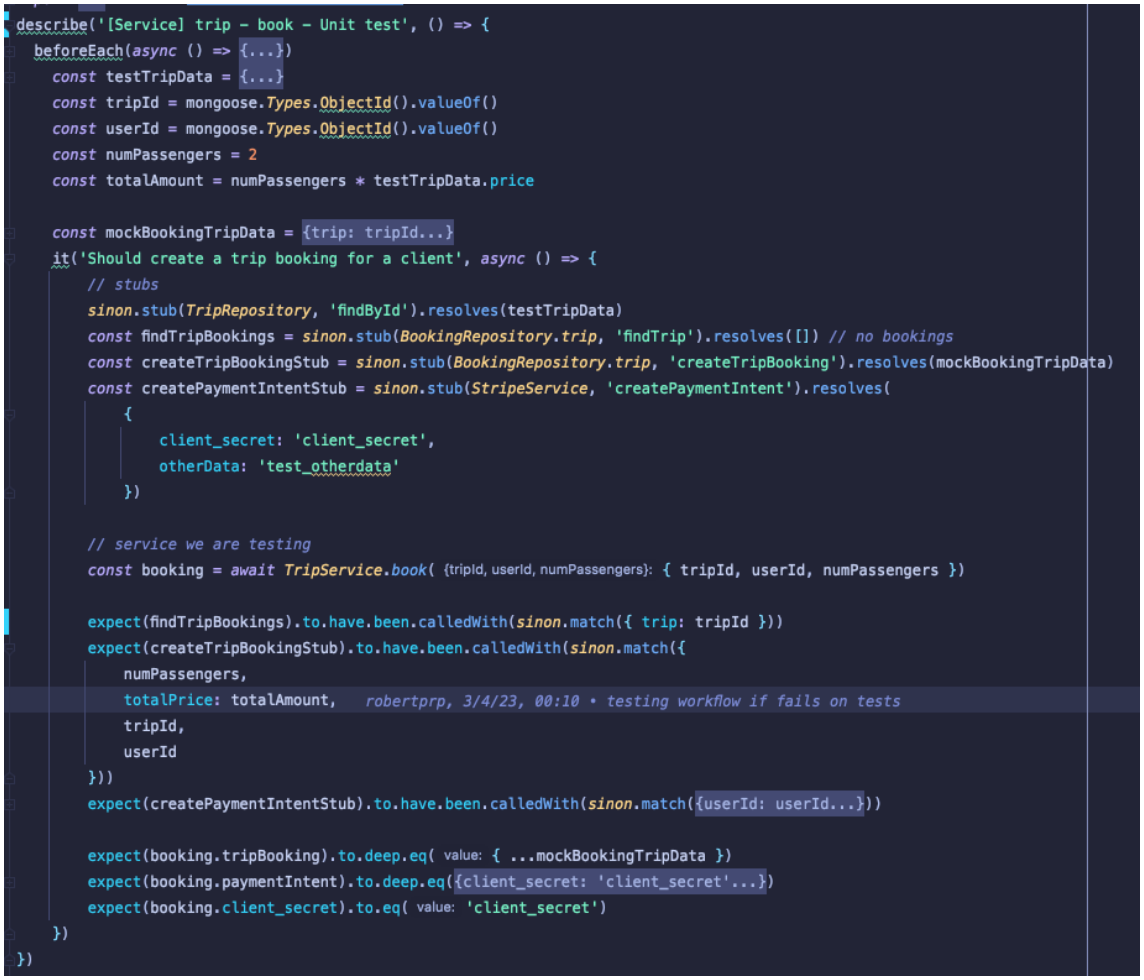


*Figura 46: Test unitario de reserva de un viaje (servicio)*

Como podemos observar en la Figura anterior, creamos copias mediante *stubs* donde imitamos su comportamiento y forzamos sus parámetros de salida de esa función. Esto es muy útil a la hora de reducir la latencia en la ejecución de tests, ya que simplemente imitamos el comportamiento de estos servicios que previamente ya deberían de estar testeados unitariamente. Con esto, conseguimos, que en el caso de que este servicio se modificará o introdujeramos lógica de negocio, este se vería afectado y los tests fallarían.

#### *7.1.2.2 Pruebas de integración*

Este tipo de pruebas pueden estar asociadas tanto a la técnica de caja blanca como de caja negra, dependiendo del enfoque que se utilice durante la ejecución de las pruebas. Ambos enfoques son válidos y pueden proporcionar información valiosa sobre la calidad y el rendimiento del conjunto de nuestros servicios.

Las pruebas de integración son vitales a la hora de probar un flujo completo o un caso de uso. Estas pruebas nos ofrecen fiabilidad para comprobar que nuestra casuística está funcionando correctamente según nuestra lógica de negocio. A continuación, se expone el caso de uso de reservar un barco como test de integración.

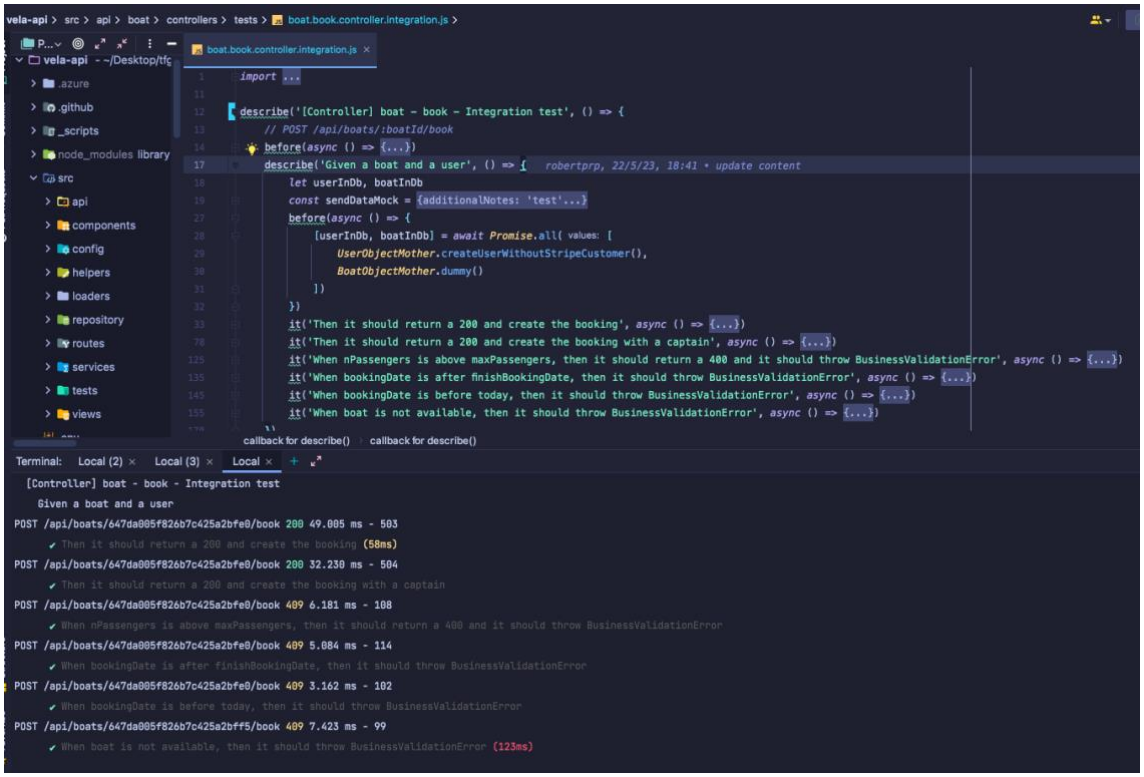


*Figura 47: Tests de integración reserva de un barco*

Como se puede apreciar en la Figura anterior, se han realizado numerosos tests intentando cubrir todas las casuísticas que conlleva este caso de uso. Se ha usado el patrón "***Given-When-Then***" para la mejora en la lectura de cada prueba. Este patrón es muy útil para describir el contexto general del que parte nuestra prueba.


*Figura 48: Test de integración - Reserva de un barco con capitán*

En este caso, es necesario hacer una imitación del servicio de *Stripe* ya que no tenemos acceso a esta librería en un entorno de prueba.


*Figura 49: Test de integración - Fallo validación de negocio el barco no se encuentra disponible para esas fechas*

Como se puede apreciar en la Figura 47 e Figura 49, se ha introducido también un nuevo patrón de testeo llamado "***Object Mother***". Este, nos proporciona mucho orden y claridad a nuestros tests, ofrece mayor contexto al lector, y tiende a la creación de un solo punto de generación de datos dentro de nuestra suite de prueba.

### 7.1.3 Despliegue mediante Github Actions

La garantía de calidad (QA) es una parte muy importante a la hora de contar con una retroalimentación rápida de pruebas, una implementación de calidad y un entregable rápido. La entrega continua es una aproximación en la cual los desarrolladores entregan funcionalidades de forma frecuente a través de la automatización de despliegues. Es por esto por lo que, debemos asegurarnos de haber realizado una buena investigación y planificación para la implementación de un **flujo de trabajo CI/CD**.

Actualmente, una de las soluciones más versátiles y flexibles es **Github Actions**[23]. [12]

Github Actions es una herramienta de integración y entrega continuas (**CI/CD**) que permite a los desarrolladores automatizar el proceso de construcción, prueba y despliegue de aplicaciones. Esta herramienta es muy flexible y fácil de usar, ya que se integra perfectamente con el flujo de Github.

Además, nos ofrece una amplia variedad de plantillas y acciones predefinidas que facilitan la creación de pipelines CI/CD personalizados y ajustados a las necesidades de cada proyecto. Esto nos permite centrarnos en la implementación de funcionalidades y en la mejora continua, en lugar de preocuparnos por la configuración y mantenimiento de la infraestructura.

Como se ha comentado anteriormente en el proyecto, nuestra aplicación trata de 3 entornos diferentes: **producción, desarrollo y test**.

Es por esto por lo que debemos definir un flujo por el que tenemos que pasar para poder llevar código nuevo a producción:

[23] Features • GitHub Actions

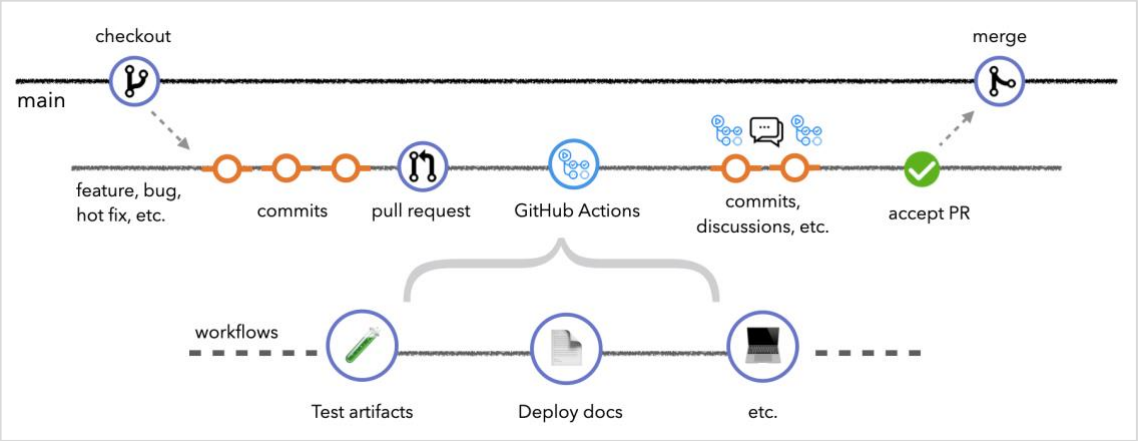


*Figura 50: Workflow para nuestra integración continua para todas las Pull Request abiertas contra master o nuestra rama principal de producción*

Nuestro archivo que automáticamente lanzamos ante cualquier *commit* en una *pull request* cuando queremos integrar en la rama de producción (master):

**Workflow file for this run**

.github/workflows/pr-workflow-ci-cd.yml at 699e28f

```yaml
name: Vela-api test coverage
on:
  pull_request:
    branches:
      - master
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout repository
      uses: actions/checkout@v2
    - name: Use Node.js
      uses: actions/setup-node@v1
      with:
        node-version: '16.x'
    - name: Run install
      run: npm install
    - name: Run tests
      run: npm test
```

*Figura 51: Flujo de trabajo para nuestro CI en todas las Pull Requests contra la rama master*

De esta manera, obligamos a que, al menos, tengamos una revisión por parte de otro usuario de nuestro equipo y que la batería de tests haya pasado satisfactoriamente para poder integrar el código en producción y lanzar el despliegue.

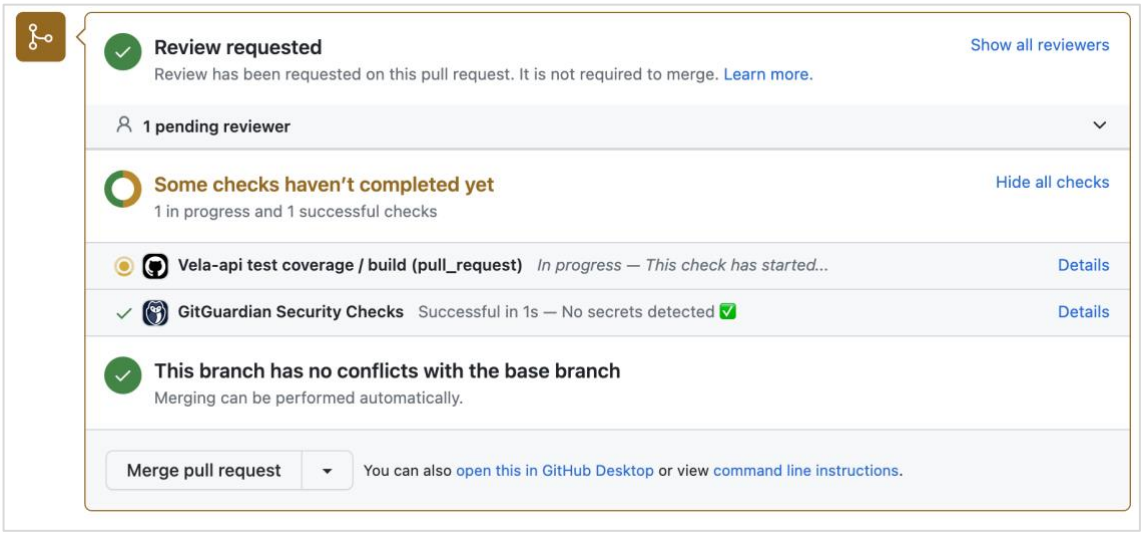


*Figura 52: Flujo de trabajo en proceso en Github Actions*

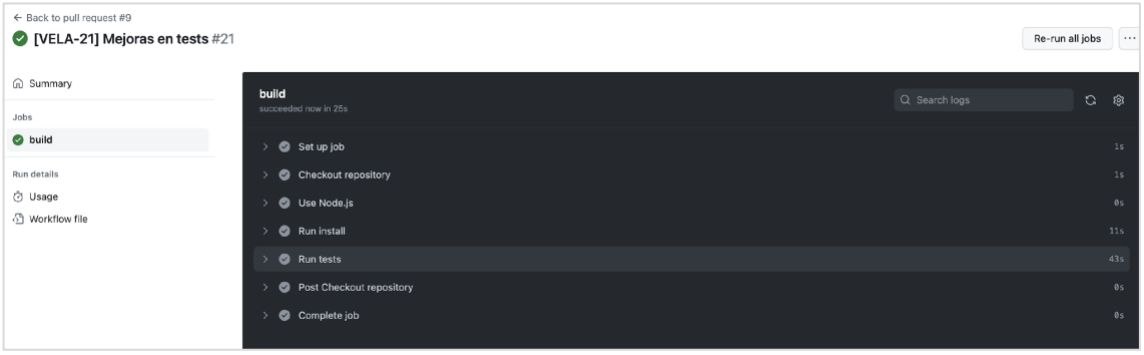


*Figura 53: Escenario satisfactorio para nuestro CI en una pull request*

Un *workflow* o flujo de trabajo se trata de una serie de pasos que se ejecutan en el proceso de CI/CD.

Cuando el CI nos haya brindado un estado satisfactorio, el desarrollador podrá integrar esta rama a la rama de producción, donde lanzaremos dos nuevos *workflows*: despliegue en el entorno de producción y desarrollo.

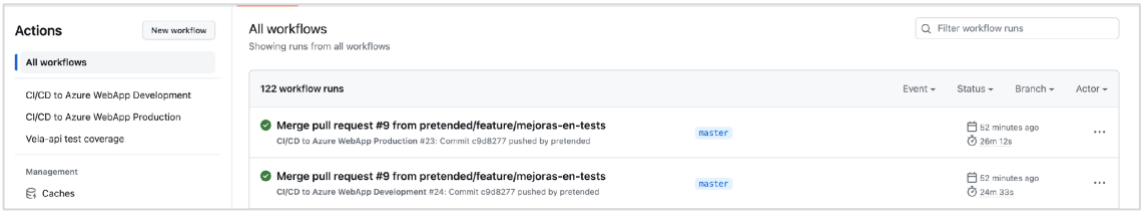


*Figura 54: Despliegue con éxito en ambos entornos después de integrar nuevo código a nuestra rama principal*

Ambos flujos de trabajo son muy parecidos, simplemente cambia la clave de Azure para realizar el despliegue en su correspondiente Web App y la actualización de la versión de nuestra API en producción. Para dar mayor contexto, se muestra el código **YAML** del archivo de despliegue de producción:

```
name: CI/CD to Azure WebApp Production
on:
  push:
    branches:
      - master
  workflow_dispatch: null
env:
  AZURE_WEBAPP_NAME: vela-api
  AZURE_WEBAPP_PACKAGE_PATH: .
  NODE_VERSION: 16.x.x
  MASTER_BRANCH: refs/heads/master
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Set up Node.js
        uses: actions/setup-node@v3
        with:
          node-version: ${{ env.NODE_VERSION }}
          cache: npm
      - name: npm install and run coverage
        run: |
          npm install
          npm test
      - name: Config Github account
        run: |
          git config --global user.email "GitHub@gmail.com"
          git config --global user.name "Github Actions"
      - name: Update package version
        if: github.ref == env.MASTER_BRANCH && job.status == 'success'
        env:
          TOKEN: ${{ secrets.GH_TOKEN }}
        run: >
          git pull
          LAST_COMMIT=$(git log -1 --pretty=%B)
          echo "Last commit $LAST_COMMIT"
            re="(#)+([0-9]+)"
            if [[ $LAST_COMMIT =~ $re ]]; then PULL_NUMBER=${BASH_REMATCH[2]}; fi
            BRANCH_NAME=$(curl -X GET \
            https://api.github.com/repos/pretended/vela-api/pulls/$PULL_NUMBER \
          -H 'Accept: application/vnd.github.v3+json' \
          -H "Authorization: token $TOKEN" \
          -H 'Content-Type: application/json' | sed -n 's|.*"'ref":\ "\([^"]*\)".*|\1|p')
          if [[ $BRANCH_NAME =~ "fix/" ]]; then VWORD=patch; else VWORD=minor; fi
          NEWVERSION=$(npm version $VWORD --no-git-tag-version)
          echo "New version is $NEWVERSION"
          echo "NEW_PACKAGE_VERSION=$NEWVERSION" >> $GITHUB_ENV
          git status
          git commit-a -m "[ci/cd] Version update to ${NEWVERSION}"
          git tag $NEWVERSION && git push --tags
          git push
      - name: Upload artifact for deployment job
        uses: actions/upload-artifact@v3
        with:
          name: node-app
          path: .
  deploy:
    permissions:
      contents: none
    runs-on: ubuntu-latest
    needs: build
    environment:
      name: Production
      url: ${{ steps.deploy-to-webapp.outputs.webapp-url }}
    steps:
      - name: Download artifact from build job
        uses: actions/download-artifact@v3
        with:
          name: node-app
      - name: Deploy to Azure WebApp Production
        id: deploy-to-webapp
        uses: azure/webapps-deploy@v2
```

Con este archivo lanzamos dos flujos de trabajo (*build y deploy*) y conseguimos varias escenarios:

- **Lanzamos de nuevo nuestro CI**, de esta manera nos tenemos más confianza en que no se hayan podido introducir pruebas inestables o "*flaky tests*". Estos nos pueden arrojar resultados válidos y erróneos a pesar de no haber cambiado el código de la prueba. Además, nos aseguramos de que el código integrado en la rama de producción es compatible con todo lo que se haya integrado previamente, es decir, si nuestra rama ha pasado nuestro CI pero fue hace tiempo, es posible que se hayan podido introducir cambios que puedan afectar a nuestra rama y tendríamos un falso CI satisfactorio.
- **Incrementamos la versión** de nuestro *package.json* para así tener trazabilidad a la hora de entrega continua y detección de errores. Tendremos una versión distinta por despliegue, sumaremos uno en caso de que este despliegue no se trate de un *fix*, o simplemente sumaremos al tercer número. Como ejemplo, si nuestra versión es la "1.2.1" y desplegamos una rama que no comience por *fix/*, nuestra nueva versión será la "1.3.0", y en caso de que comenzase por *fix/*, pasaríamos a tener la versión "1.2.2". El versionado de nuestra aplicación es muy útil para poder hacer "***rollback***" a versiones anteriores. Esta acción trata de volver a una versión anterior de nuestra aplicación por X motivos, como, por ejemplo, cuando se han introducido cambios en la rama de producción y esto está afectando al uso general de la aplicación, como podría ser en nuestro caso, el reservar un barco.
- **Despliegue en Azure**: subimos nuestra aplicación para que el *workflow* de despliegue pueda descargarlo posteriormente y realizar el despliegue en la máquina de Azure usando el action *azure/webapps-deploy@v2*.

## 7.2 Frontend

Para la realización de tests en la parte frontal se ha usado la librería **Cypress,** este *framework* nos permite la realización de pruebas *end-to-end* y la realización de pruebas unitarias de componentes. Las pruebas *end-to-end* se tratan de pruebas que realizan el flujo completo en toda su extensión. Este *framework* nos ofrece la posibilidad de grabación de nuestras pruebas, esta funcionalidad es muy útil para cuando se trata de tests que han fallado. En la siguiente Figura se va a realizar el flujo de registro de un nuevo usuario.

```
describe( title: 'Register',  fn: () => {
    const viewports = [
        { name: 'Mobile', viewport: 'iphone-6' },
        { name: 'Desktop', viewport: 'macbook-13' }
    ]
    viewports.forEach(viewport => {
        context( title: `Device: ${viewport.name}`,  fn: () => {
            beforeEach( fn: () => {
                cy.viewport(viewport.viewport)
                cy.fixture('../e2e/fixtures/user-register.json').as( alias: 'userRegister')
                cy.intercept('POST', '**/api/auth/signup', '@userRegister').as( alias: 'signup')
            })
            it( title: 'should register a new user',  config: () => {
                cy.visit('auth/register')
                cy.get('[data-cy="name"]').type( text: 'John Doe')
                cy.get('[data-cy="email"]').type( text: 'John@john.com')
                cy.get('[data-cy="password"]').type( text: 'Password1')
                cy.get('[data-cy="registerButton"]').click()
                cy.wait('@signup')
                cy.url().should( chainer: 'include',  value: '/app/home')
            })
        })
    })
})
```

*Figura 55: Test end-to-end para el registro de un usuario en Cypress*

En este caso, realizamos la prueba del registro de un nuevo usuario exitoso. Una vez que haga *click* en el botón de registro tenemos que imitar la respuesta de nuestro *backend* usando *fixtures* e interceptando la petición. Un *fixture* se trata de un archivo *JSON* con la respuesta que tendríamos por parte de nuestro *backend* para un caso de uso específico, y la función de interceptar se trata de que *cypress* va a intentar esperar a que se lance esa petición para imitarla y dar como respuesta la *fixture*. Una vez que esto se resuelve, comprobaremos que hemos iniciado sesión y estamos en la página principal. Una buena práctica es asignar campos *data-cy* en cada uno de los elementos HTML que vayamos a obtener, como se muestra en la Figura anterior. Nunca deberíamos de obtener el campo HTML a testear usando consultas por etiqueta de clase.

Para el despliegue de la aplicación de frontend, se ha usado **Vercel**[24].

Vercel es una aplicación en la nube que nos permite desplegar, gestionar e incluso escalar nuestras aplicaciones a través de repositorios Git. Se integra con el flujo de Github para que podamos realizar despliegues automáticamente con una pequeña configuración.

El *workflow* [13] que se ha usado para los tests teniendo en cuenta que nuestro proveedor de despliegue es Vercel:

[24] https://vercel.com/

```
name: Deploy PR only VERCEL CI

on:
  pull_request:
    branches:
      - master

jobs:
  run-e2es:
    name: Test E2E Cypress
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v2
      - name: Install dependencies
        run: npm ci
      - name: Uninstall Serve test
        run: npm uninstall -g serve
      - name: Install Serve test
        run: npm i -S serve
      - name: Generate routes for tests
        run: npm run generate:local:test
      - name: Cypress run
        uses: cypress-io/github-action@v5
        with:
          start: npm run serve:static
          browser: chrome
          config-file: cypress.config.js
          config: pageLoadTimeout=10000,baseUrl=http://localhost:3000
          spec: 'cypress/e2e/**/*.{js,jsx,ts,tsx}'
          wait-on: 'http://localhost:3000'
        env:
          NODE_ENV: development
```

De manera general, este *workflow* se ejecuta siempre que se crea una *pull request* hacia la rama principal, en nuestro caso mater. Instalamos las dependencias necesarias, crearemos la *build* de nuestro proyecto en un entorno local y testing. La construcción del proyecto simplemente generará las rutas necesarias y los archivos estáticos para poder levantar el proyecto en un entorno local donde cypress pueda ejecutarse. De última instancia, lanzaremos la acción que nos proporciona cypress para lanzar nuestra suite de tests.

El script *deploy:test* lanza un ejecutable con el siguiente archivo de ejecución *bash*:

```
#!/bin/bash
set -e # Exit immediately if a command exits with a non-zero status
rm -rf ./dist # Clean the files in dist
# Generate statics
npm run generate:local:test # this is the same as: vue-cli-service build --mode development
serve -p 3000 ./dist -L
```

El comando *serve* nos permite crear un servidor en local en la carpeta */dist*, donde se encuentra nuestra *build* o todos los archivos estáticos generados. Este comando trata de una librería[25] de *npm* para poder levantar un servidor de archivos estáticos y lanzar los tests de *cypress* contra este servidor.

[25] https://www.npmjs.com/package/serve

Para cada *commit* que realicemos a nuestra *pull request*, vamos a lanzar de nuevo nuestro CI. Únicamente integraremos a nuestra rama principal las ramas que hayan pasado satisfactoriamente el CI.

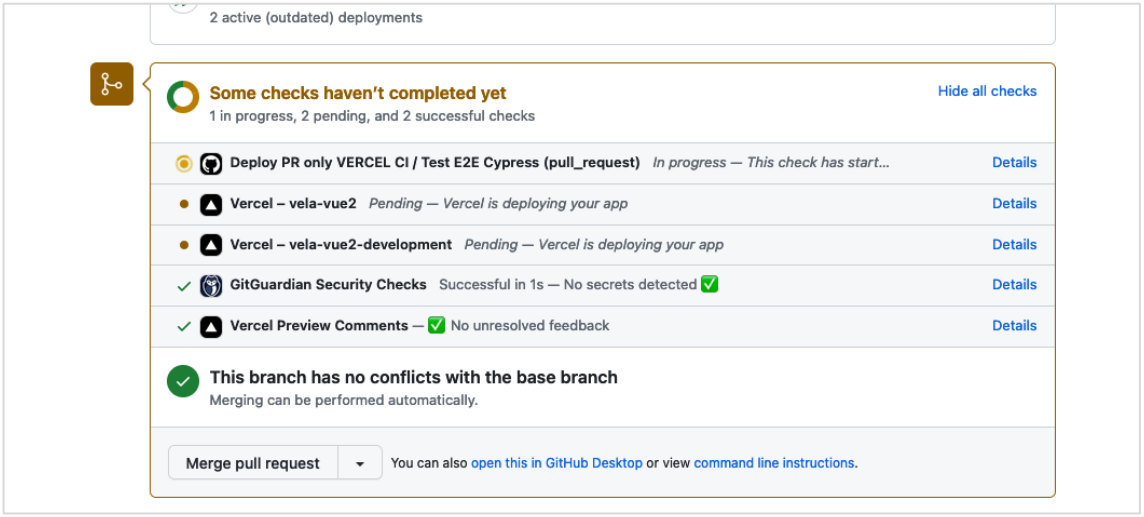


*Figura 56: Flujo de trabajo Cypress en ejecución*

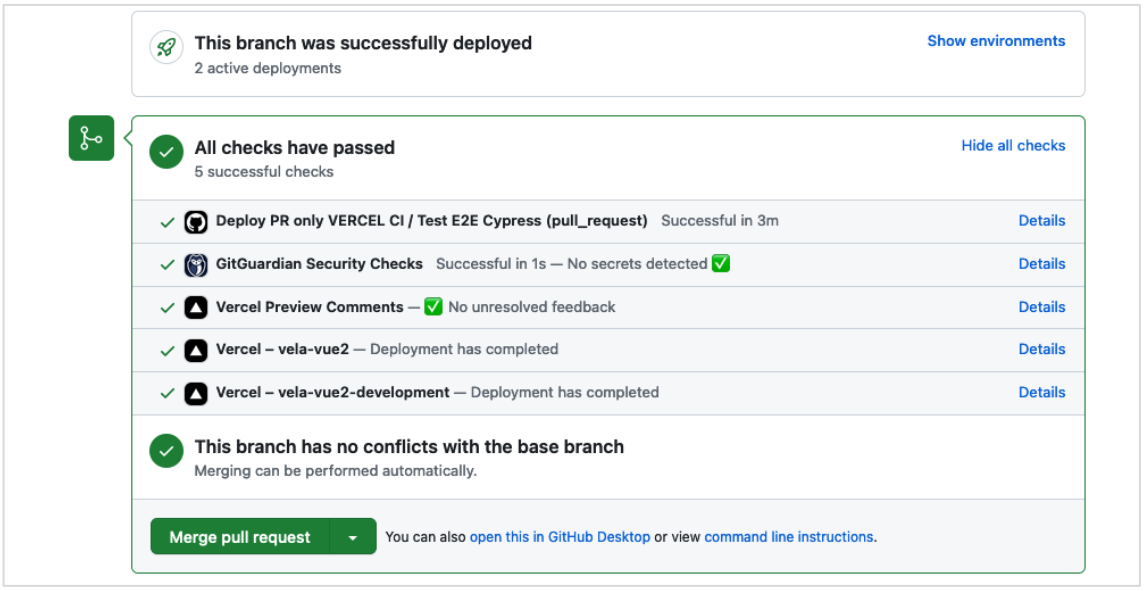


*Figura 57: Flujo de trabajo Cypress satisfactorio, despliegues en Vercel completados y no se han detectado claves secretas*

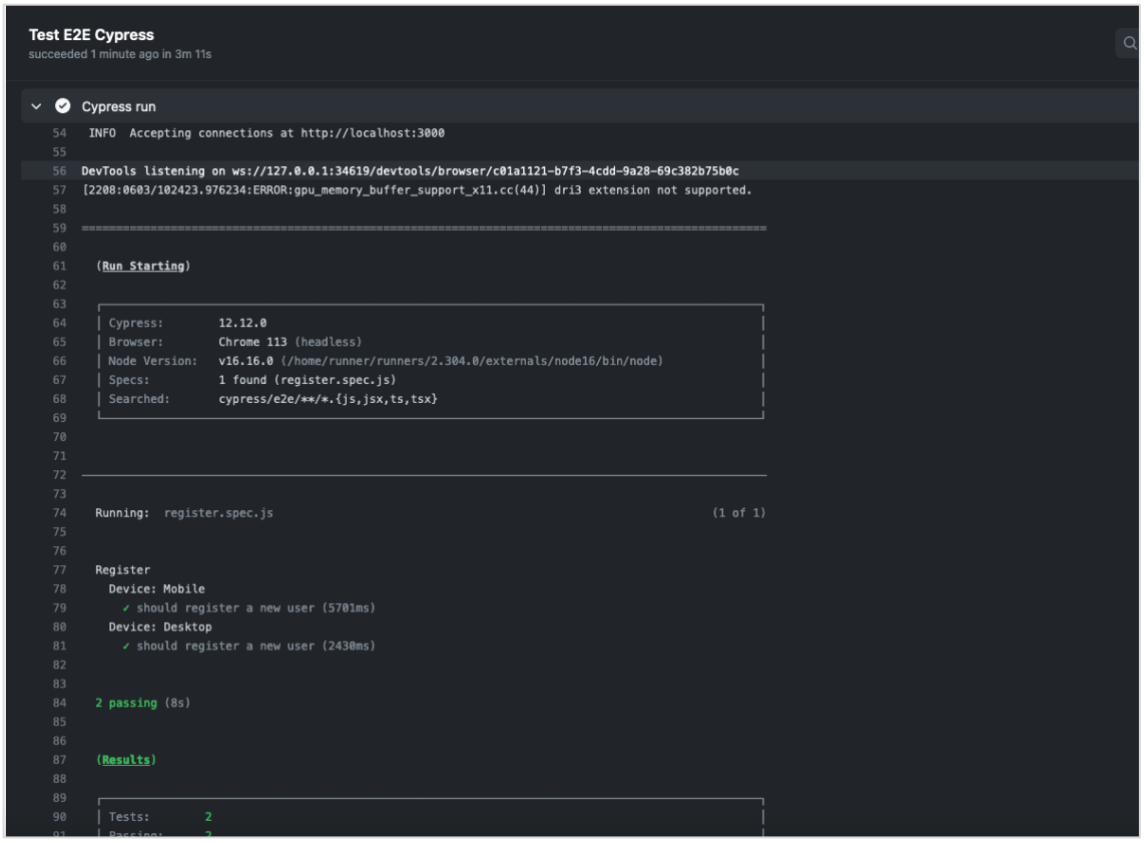


*Figura 58: Flujo de trabajo Cypress en ejecucción en la consola de Github Actions*

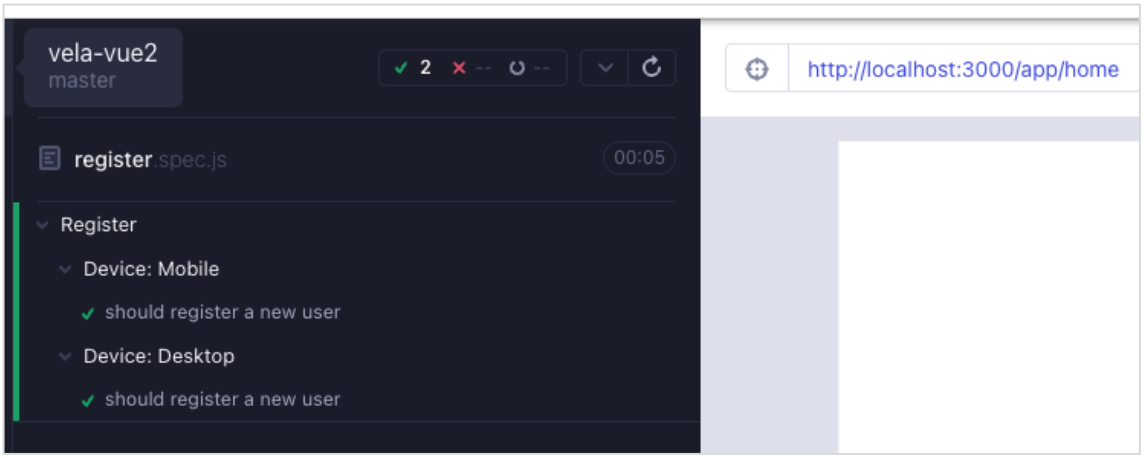


*Figura 59: Interfaz gráfica de Cypress para lanzar los tests en nuestro entorno local - Test de Registro de usuario*

Debemos probar nuestros tests sobre ambas vistas, móvil y ordenador, para poder estar seguros de que nuestro diseño e implementación es responsable con todos los dispositivos y no afecta a la experiencia de usuario.

## 7.3 Detección y corrección de errores en tiempo real

Cuando una aplicación crece en escala y en usuarios necesitamos, también, mejoras en los procesos de detección de errores ya que podemos repercutir de manera severa en la experiencia de usuario. La configuración de un buen sistema de monitorización de fallos nos ayuda a poder mitigar errores que no han sido controlados por nuestro CI.

### 7.3.1 Sentry

Sentry[26] se define como un sistema que está enfocado en la monitorización de aplicaciones en tiempo real, a través de un rastreador que supervisa y responde a los fallos que se puedan presentar en nuestra aplicación. Es de muy fácil configuración para entornos como Node.js, entre otros. Se ha realizado la integración e implementación de Sentry tanto en el *backend* como en el *frontend*.

#### 7.3.1.1 Implementación Backend

Para la implementación de este servicio, se ha seguido la documentación[27] proporcionada por Sentry. Se ha necesitado realizar pequeñas adaptaciones para nuestra API, como puede ser la versión de nuestra aplicación o la posibilidad de añadir más información sobre el usuario cuando se trata de errores autenticados.

Según lo planteado anteriormente, nuestra configuración nos permite:

- **Distinguir entre entornos**: Para poder saber desde que entorno se ha lanzado el error y abordarlo correctamente.
- **Distinguir entre la última versión de nuestra aplicación**: Para poder comprobar en qué versión se ha lanzado el error.
- **Conocer el usuario que ha lanzado el error**: Para poder identificar el usuario que ha causado este error y acotar el análisis de este.

Algunos errores que se han capturado mientras se hacían pruebas en los entornos:

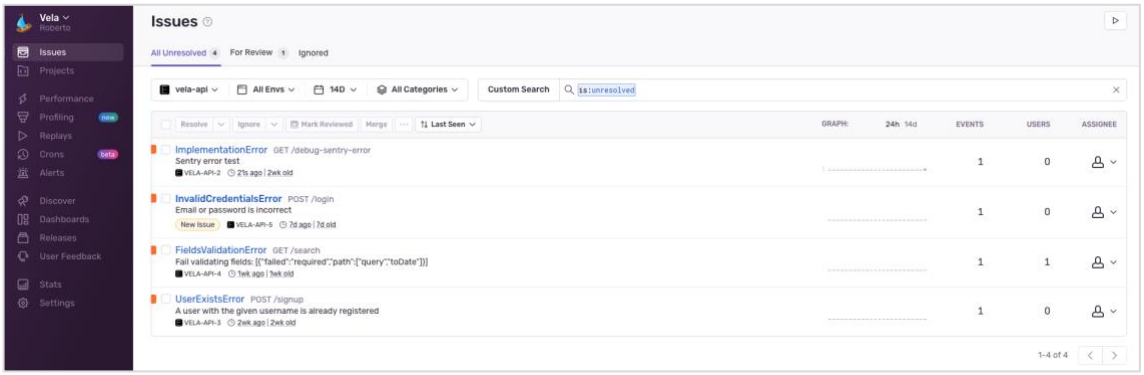


*Figura 60: Interfaz Vela API en Sentry*

En esta interfaz podemos encontrar: errores controlados y errores no controlados. Un error controlado es, por ejemplo, "*UserExistsError*" cuando un usuario se intenta registrar con un email ya en uso. Un error no controlado se trataría de, por ejemplo, un error de implementación por un fallo en el código. Estos tipos de errores jamás deberían de aparecer ya que se relacionan con fallos en el código y no lógica de negocio.

[26] https://sentry.io/
[27] https://docs.sentry.io/platforms/node/

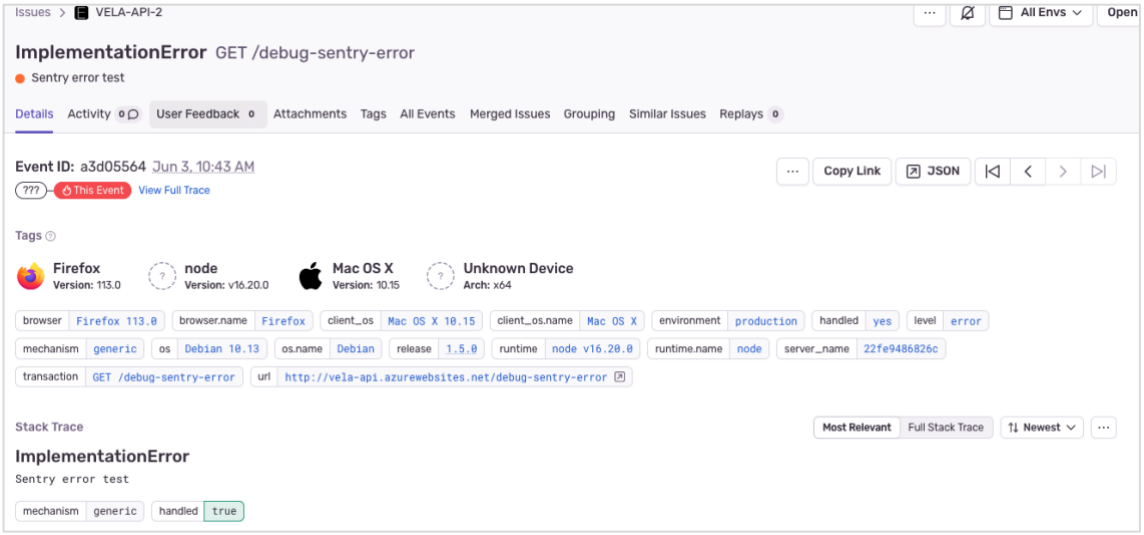


*Figura 61: Interfaz de un error en Sentry*

Como se puede apreciar, podemos visualizar una gran variedad de información de un error. Podemos observar que este se ha lanzado en el entorno de producción, la versión en la cual se ha lanzado el error, la traza completa del error, las cabeceras o la información relativa a la petición HTTP, y en el caso de que se tratase de un usuario autenticado podríamos ver su id para poder hacer un mejor análisis sobre la causa del error.

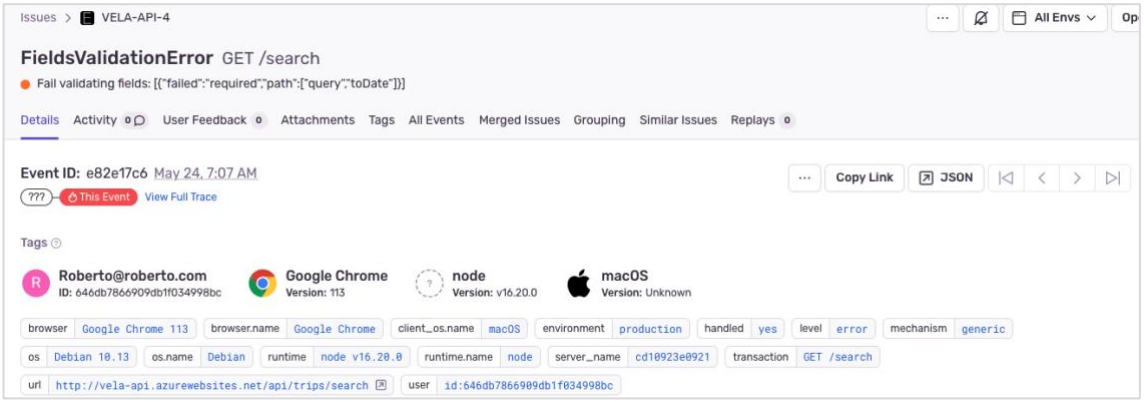


*Figura 62: Interfaz de un error de validación de campos en Sentry*

### 7.3.1.2 Implementación Frontend

Para la implementación en el frontend se ha seguido la siguiente documentación[28].

Se ha creado un error de prueba desde la parte frontal para poder visualizar la implementación realizada:

[28] https://docs.sentry.io/platforms/javascript/guides/vue/

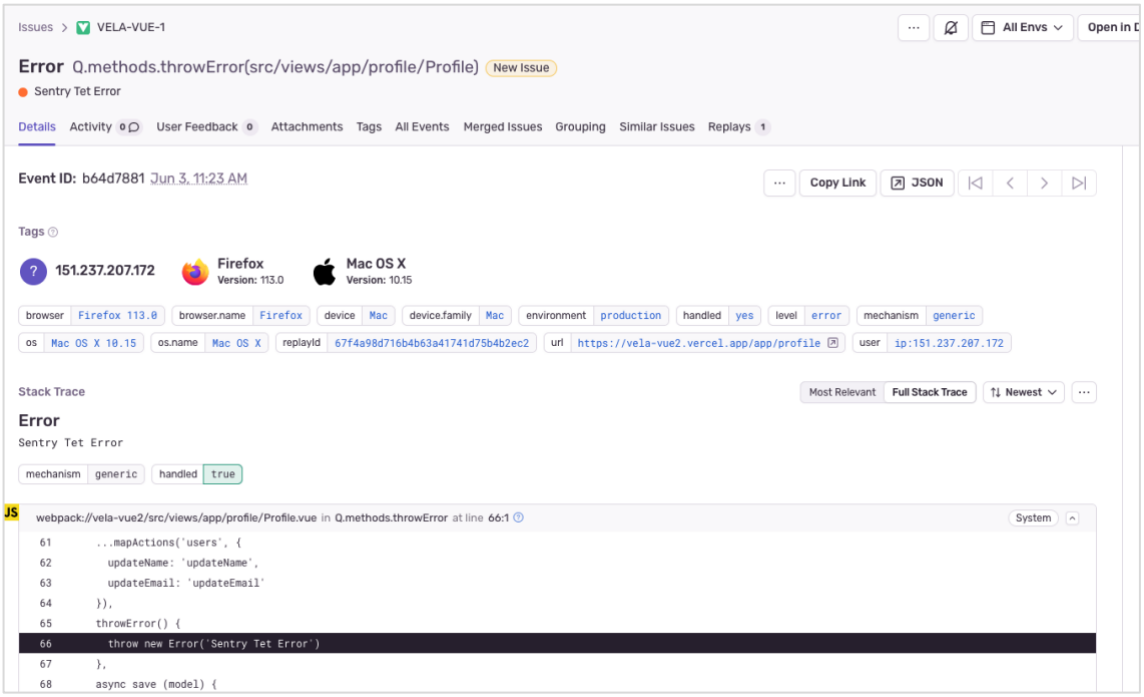


*Figura 63: Interfaz de un error en el componente de perfil de usuario en la parte frontal*

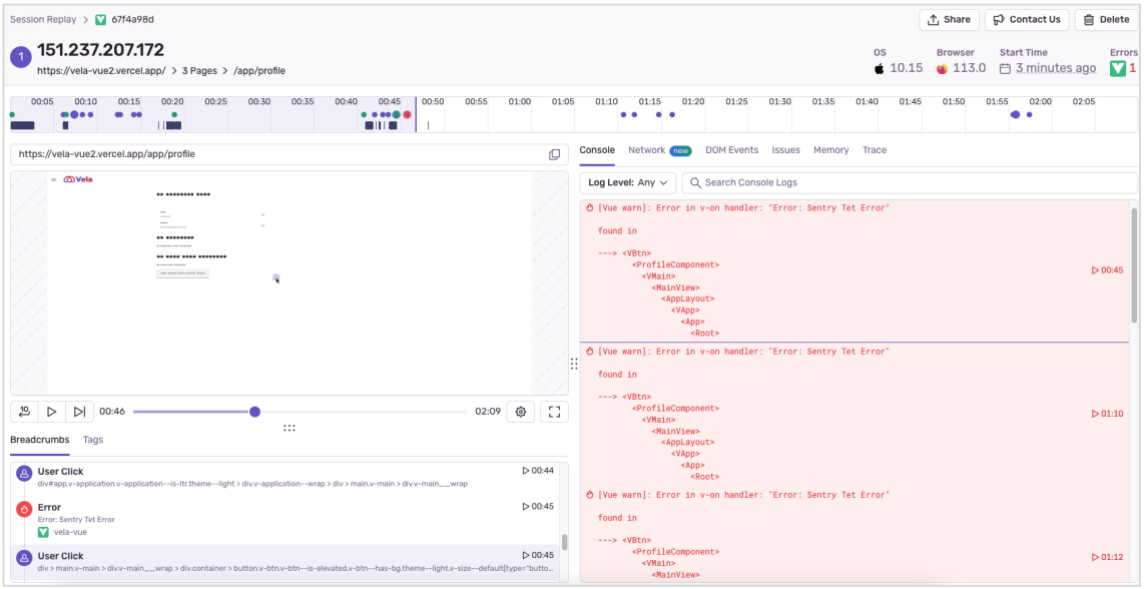


*Figura 64: Interfaz de sentry - Análisis detallado del error*

Desde esta parte podemos ver un video completo de toda la traza que ha seguido el usuario hasta que se ha lanzado el error, la consola del navegador, las peticiones a otros servicios externos, los eventos del DOM, los clicks que se han dado, etc…

### 7.3.2 Datadog

Datadog[29] es un servicio en la nube que nos brinda monitorización no solo a nuestra aplicación si no a servidores, bases de datos, herramientas y servicios. Se caracteriza por ser uno de los APM (Gestión del rendimiento de la aplicación) más versátiles actuales en el mercado.

[29] https://datadoghq.eu/

Con este servicio se pueden realizar infinidad de estadísticas y operaciones:

- Podemos hacer un análisis de todas nuestras peticiones HTTP y su tiempo de respuesta para poder mejorarlas.
- Crear plantillas para poder monitorizar en tiempo real las peticiones.
- Incluir logs dentro de nuestros servicios; esto es muy útil si tenemos un servicio como Agendash[30] u otro microservicio para ejecutar procesos en segundo plano.
- Crear un análisis sobre la carga que está realizando nuestro servidor y la carga en el servidor de MongoDB.
- Estado de nuestras peticiones en tiempo real, notificaciones o alertas para errores en un proceso X.

Un ejemplo del panel de Datadog:

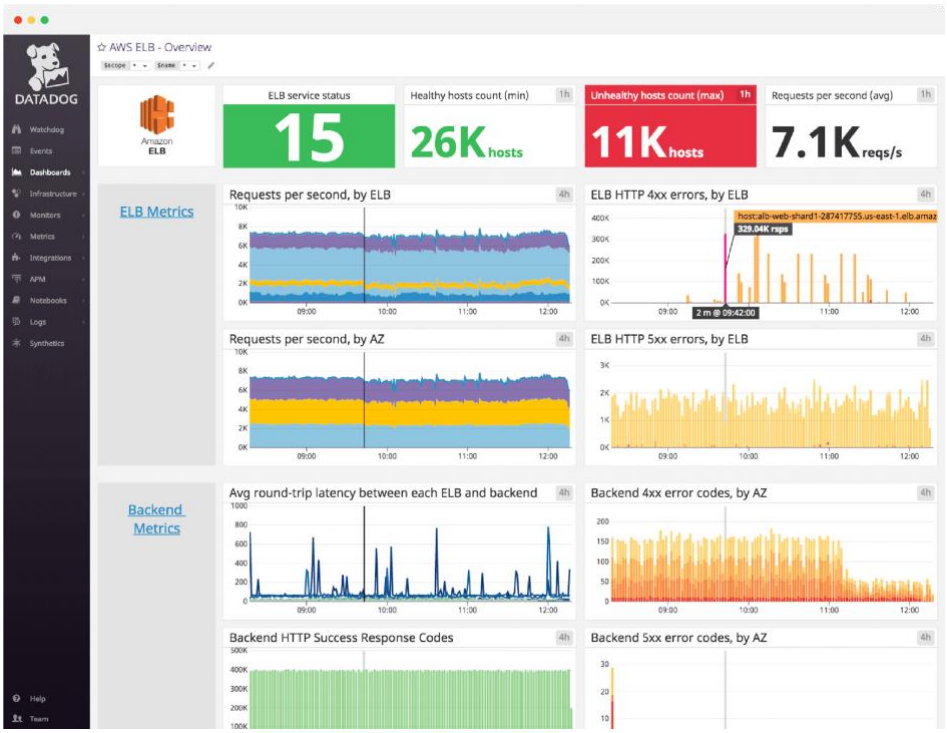


*Figura 65: Ejemplo panel de Datadog*

[30] https://github.com/agenda/agendash

# 8 Análisis de la metodología usada en el proyecto

La elección de una metodología de trabajo ha supuesto un punto fundamental a la hora del desarrollo de la aplicación. Una planificación y organización es primordial a la hora de abordar objetivos. En cuanto a Vela, en el capítulo 4.1.1.3 se habla de la metodología escogida para el desarrollo del proyecto.

Este capítulo se centrará en el análisis y descripción del proceso de desarrollo de software que se ha decidido usar en el desarrollo de este proyecto [15]. Durante el desarrollo se ha notado la necesidad de hacer una metodología de prioridades usando Scrum y Kanban. En el tablero, se iban realizando las tareas que más prioridad tenían. Se decidió realizar 6 iteraciones de 4 semanas cada una:

**Iteración 1**

Damos comienzo al proyecto, en esta iteración hemos definido los requisitos de desarrollo, hemos añadido las tareas al *backlog*, hemos realizado el análisis completo del problema a abordar y hemos comenzado con el desarrollo de la parte del *backend*.

**Iteración 2**

Se han dado por finalizado el modelado de las identidades de nuestro proyecto, se ha realizado la implementación del inicio de sesión y registro de un usuario, se ha desarrollado una parte del frontal junto con la implementación de Stripe en el *backend*.

**Iteración 3**

Se da por finalizada la parte de implementación de Stripe en la parte frontal y *backend*, se finaliza la implementación y el desarrollo de la parte administrativa en la parte frontal.

**Iteración 4**

Se realizan los despliegues para el *backend* y *frontend*, se realizan los tests para ambas partes, así como una automatización de despliegue mediante Github Actions para los diferentes entornos.

**Iteración 5**

Se finaliza la parte de desarrollo en ambas partes, se implementa Sentry para en ambas partes y se realizan pruebas de aceptación.

**Iteración 6**

Se solucionan los errores que hayan podido ocurrir usando nuestro sistema de detección de errores, se crean tareas correspondiente en nuestro tablero y se archivan los errores controlados.

Por cada iteración, se ha realizado una retrospectiva para analizar si hemos cumplido con los objetivos/tareas propuestas al inicio del *sprint* o iteración, y en caso contrario, que ha podido fallar para que esto no ocurriese: una mala definición o análisis de tareas que ha extendido la

estimación de esfuerzo de trabajo de esta, una mala estimación de capacidad de trabajo, alta incertidumbre a la hora de abordar tareas, etc… Posteriormente a la retrospectiva, se añaden las tareas del *backlog* para el siguiente *sprint* a la columna de **"Lista de tareas"**, incluyendo las que no se han podido completar, hasta alcanzar el nivel de esfuerzo para esa iteración y se da comienzo a esta.

Así es como ha quedado el tablero de **Trello**[31] una vez finalizada las 6 iteraciones:

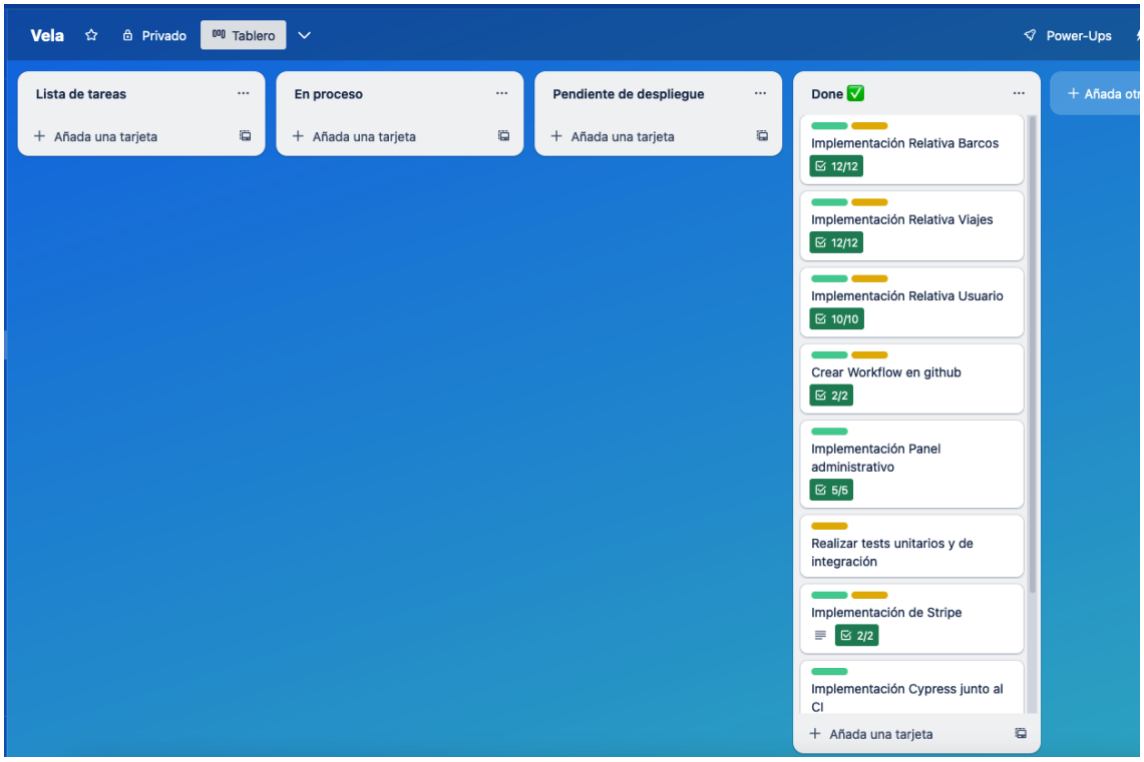


*Figura 66: Panel de Trello al finalizar todas las iteraciones*

**Trello** es un administrador y gestor de proyectos de software que permite a los equipos manejar flujos de trabajo de manera eficiente usando una metodología *Kanban.*

Se han planteado 4 tipos de columnas según la necesidad que se ha visto a la hora del desarrollo del proyecto:

- **Lista de tareas**: Tareas definidas a falta de comenzar.
- **En proceso**: Tareas que están siendo desarrolladas.
- **Pendiente de despliegue**: Tareas que han pasado nuestro CI y están listas para ser desplegadas.
- **Tareas desplegadas**: Tareas desplegadas y en producción. Tareas finalizadas.

Se ha usado el mismo tablero para la realización de la parte del *backend* y frontend. En cada caso, se ha marcado con una etiqueta de color amarillo para las tareas relacionadas con el *backend* y con una etiqueta de color verde para las tareas de *frontend.*

[31] https://trello.com/es

# 9 Conclusiones

El uso de una metodología es clave para una buena organización del proyecto, medir esfuerzos y realizar análisis previos a ciertas implementaciones es clave antes del inicio de cualquier desarrollo. La realización de retrospectivas ayuda a que un equipo pueda plantear sus inquietudes, posibles mejoras en el equipo, fallos durante el desarrollo del sprint, etc…

El uso de varios frameworks de tecnologías bien conocidas ha ahorrado mucho trabajo de programación. Estos framework son herramientas clave para el desarrollo de un proyecto exitoso ya que cuentan con una gran comunidad y soporte detrás.

El uso de mongoose, cómo ODM, para la interacción entre la base de datos MongoDB y nuestra aplicación, ha sido desde mi punto de vista acertada, sin embargo, creo que, es muy fácil acoplarse a esta librería ya que nos facilita mucho el desarrollo a la hora de hacer consultas y a la larga esto se nos puede volver en nuestra contra.

Lo más complicado desde mi punto de vista ha sido la puesta en marcha de las automatizaciones o el pipeline CI/CD para ambos proyectos (frontend y backend). Ha sido retador la creación de un CI satisfactorio para luego realizar un despliegue de una nueva versión, ya sea por problemas con Azure o por implementaciones difíciles por mi parte. El despliegue de la parte frontal ha sido muy sencillo, pero, sin embargo, la automatización del CI ha llevado mayores esfuerzos de lo esperado y analizado. Debido a que tenemos un acoplamiento con Vercel, necesitamos bloquear la rama principal y añadir cualquier nuevo despliegue desde una pull request a esta.

La implementación de Stripe ha sido relativamente sencilla en ambas partes, creo que esta librería es muy sencilla de usar, con muchísima documentación por detrás y buen soporte.

La implementación de Sentry ha sido muy útil para poder encontrar errores que no se estuvieran teniendo en cuenta mientras se realizan pruebas de aceptación en nuestro entorno de desarrollo o no se hayan detectado por nuestro CI. Cometemos errores continuamente a la hora de desarrollar, pero está en nuestras manos el poder identificarlos y solucionarlos en la mayor brevedad posible.

En mi opinión el proyecto, tras todas las iteraciones de desarrollo, se puede concluir que el desarrollo de la aplicación ha sido todo un éxito y se han cumplido las metas y objetivos planteados a su comienzo. Se ha creado una aplicación robusta por ambas partes, segura, y con una automatización muy alta para disminuir al máximo posible tiempos de desarrollo o despliegues.

## 9.1 Relación proyecto-estudios cursados

Primero de todo, es importante mencionar que gran parte de mi conocimiento con la automatización de despliegues y pagos online vienen de la experiencia trabajando para una empresa start-up y conocimientos adquiridos por mí de manera autodidacta. Sin embargo, el trabajo desarrollado en esta aplicación está relacionado con una asignatura cursada en el grado.

La asignatura de Bases de datos ha sentado las bases para poder entender cómo organizar datos de una manera flexible y accesible. Si bien es cierto que en este proyecto se ha usado una base

de datos no relacional y que en esta asignatura se enseñan bases de datos relacionales, ha sido de ayuda a la hora de entender mejor las bases de datos NoSQL como MongoDB.

Otra asignatura que considero de gran relevancia ha sido la asignatura de Ingeniería del software, cursada en tercer curso, ya que me ha dado las capacidades y asentado las bases para participar eficazmente en proyectos de desarrollo de software desempeñando todas las actividades descritas en el trabajo como son: requisitos, análisis, implementación, pruebas y/o mantenimiento del software.

Además, considero la competencia transversal *Aprendizaje permanente* muy relevante para este trabajo desarrollado. A pesar de que algunos de los temas tratados se introdujeron por algunas asignaturas que curse, estos no fueron con suficiente profundidad. Por lo tanto, al estar aprendiendo de manera autodidacta por mi cuenta, con interés y ganas por aprender nuevas tecnologías, he podido obtener una comprensión mucho más profunda de los temas del proyecto: Stripe, Github Actions, MongoDB, CI/CD, Vue, Cypress, etc… Otra competencia transversal relevante es *Análisis y resolución de problemas,* ya que durante el desarrollo de este trabajo se ha necesitado realizar análisis previos de las posibles tecnologías a usar para aumentar la probabilidad de éxito del proyecto. Una buena organización, análisis y definición han sido clave para el desarrollo de requisitos y la resolución de los problemas planteados.

# 10 Trabajo futuro y líneas de mejora

Durante el desarrollo de la aplicación, se han detectado ciertas mejoras que se podrían haber realizado si se hubiera hecho un mejor análisis y planteamiento. Estas propuestas deberían ser analizadas y comprobar su viabilidad.

Desde un primer momento se ha de tener claro que una metodología adaptada al desarrollo de un proyecto es clave para determinar el éxito y el nivel de esfuerzo para la realización de tareas de calidad.

Se plantea la posibilidad de haber realizado una implementación de una arquitectura hexagonal, esta se complementa muy bien con un sistema de desarrollo dirigido por el dominio (DDD), para más detalles sobre esta propuesta consulta el anexo al final del trabajo.

Durante el desarrollo del proyecto se ha notado la necesidad de poder trabajar con un lenguaje fuertemente tipado, como puede ser **Typescript**. Este es un lenguaje de programación libre y de código abierto desarrollado y mantenido por Microsoft. Es un superconjunto de JavaScript, que esencialmente añade tipos estáticos y objetos.

En términos de escalabilidad, se plantea la posibilidad de haber utilizado una arquitectura basada en microservicios en lugar de una arquitectura monolítica. Esta decisión habría permitido una mayor flexibilidad y capacidad de adaptación a medida que la aplicación crece y se requieren cambios y actualizaciones. Sin embargo, hubiéramos necesitado un mayor esfuerzo de trabajo.

En cuanto a los pagos en línea, si bien se ha optado por el uso de Stripe, existen otros proveedores de pago online muy usados y con mejores comisiones por cobro que Stripe. Por cada pago que se realiza en la aplicación con tarjeta, se nos cobra el 1.5% del pago + 0.25 euros. Además, sería beneficiosa la implementación de nuevos tipos de cobro, como puede ser cobro por SEPA, sin embargo, esto requeriría un análisis de viabilidad para esta nueva funcionalidad. Se puede mejorar la manera en la que se gestionan las tarjetas ya que se podría adjuntar la tarjeta al usuario en nuestra aplicación para que este pudiese usarla sin tener que introducirla por cada pago que realice.

En cuanto a los flujos de trabajo, una línea de mejora podría ser el bloqueo de la rama de producción mientras se está realizando un despliegue, ya que, si se realiza uno nuevo mientras hay otro en curso, ambos podrían fallar. Se podría contratar una mejor versión de Azure para obtener una máquina con RAM dedicada, de esta manera, podríamos mejorar el tiempo de despliegue o de respuesta de nuestra API.

En el caso de que una de nuestras capas falle y hayamos creado documentos anteriores al fallo, necesitamos introducir el concepto de sesiones y transacciones. Supongamos que guardamos un documento en una transacción y posteriormente falla; los cambios en este documento no se van a persistir en nuestra base de datos ya que van reflejadas en sesiones. Estas sesiones, en el caso de un caso de uso exitoso, deben de comprometerse para así persistir los cambios y ser visible a cualquier llamada de búsqueda sin sesión. En el caso de un caso de uso fallido, la sesión abortará y desharemos todos los cambios realizados durante la sesión. El uso de transacciones es

una funcionalidad atómica muy útil para poder mantener una consistencia de datos adecuada en nuestra base de datos ante errores no controlados.

# 11 Referencias


[1] MongoDB [En línea]. Disponible: https://www.mongodb.com/ [Último acceso: 25/06/2023].

[2] ExpressJs [En línea]. Disponible: https://expressjs.com/ [Último acceso: 25/06/2023].

[3] Evan You, "Vuejs" Versión 2 [En línea]. Disponible: https://v2.vuejs.org/ [Último acceso: 25/06/2023].

[4] NodeJs [En línea]. Disponible: https://nodejs.org/en [Último acceso: 25/06/2023].

[5] Yari Antonieta, Ctrly Blog [En línea]. Disponible: https://ctrly.blog/es/arquitectura-capas/ [Último acceso: 25/06/2023].

[6] MongoDB, Fundamentos de las bases de datos NoSQL, [En línea]. Disponible: https://www.mongodb.com/es/nosql-explained [Último acceso: 25/06/2023].

[7] rvpran - Geeks For Geeks, Node.js authentication using Passportjs and passport-local-mongoose [En línea]. Disponible: https://www.geeksforgeeks.org/node-js-authentication-using-passportjs-and-passport-local-mongoose/ [Último acceso: 25/06/2023].

[8] Crear un intento de cobro, Documentación de Stripe, [En línea]. Disponible: https://stripe.com/docs/api/payment_intents/create [Último acceso: 25/06/2023].

[9] John Leider, "Vuetify" Versión 2 [En línea]. Disponible: https://v2.vuetifyjs.com/en/getting-started/installation/ [Último acceso: 25/06/2023].

[10] Confirmar un pago con tarjeta, Documentación de Stripe, [En línea]. Disponible: https://stripe.com/docs/js/payment_intents/confirm_card_payment [Último acceso: 25/06/2023].

[11] RedHat, La integración y la distribución continuas (CI/CD), [En línea]. Disponible: https://www.redhat.com/es/topics/devops/what-is-ci-cd/ [Último acceso: 25/06/2023].

[12] Brian Douglas - Github Blog, How to build a CI/CD pipeline with Github Actions in four simple steps, [En línea]. Disponible: https://github.blog/2022-02-02-build-ci-cd-pipeline-github-actions-four-steps/ [Último acceso: 25/06/2023].

[13] Skies.dev, Run Cypress Tests against Vercel Preview Deployments, [En línea]. Disponible: https://www.skies.dev/cypress-ci [Último acceso: 25/06/2023].

[14] Cuerpo de conocimiento de Scrum (guía SBOK), Aprendiendo Scrum [En línea]. Disponible: http://blog.pucp.edu.pe/blog/pamelars/2018/01/19/aprendiendo-scrum/ [Último acceso: 25/06/2023].

[15] ProyectosAgiles.org, Desarrollo iterativo o incremental, [En línea]. Disponible: https://proyectosagiles.org/desarrollo-iterativo-incremental/ [Último acceso: 25/06/2023].

[16] Picodotdev, Introducción a DDD y arquitectura hexagonal con un ejemplo de aplicación en Java [En línea]. Disponible: https://picodotdev.github.io/blog-bitix/2021/02/introduccion-a-ddd-y-arquitectura-hexagonal-con-un-ejemplo-de-aplicacion-en-java/ [Último acceso: 25/06/2023].

# 12 Anexos

## 12.1 Anexo A: Prueba de concepto de una arquitectura hexagonal

Se ha realizado una implementación centrada en el negocio de Vela con DDD, Typescript y arquitectura hexagonal.

El desarrollo dirigido por el dominio o DDD es un concepto que propone la sugerencia de que la representación y terminología utilizada en el código, como los nombres de las clases, métodos y variables, deben ser una manifestación del ámbito o área de negocio al que se refieren.

La arquitectura hexagonal [16] propone un aislamiento del modelo de dominio de los elementos externos con el fin de que, aunque la infraestructura cambie, estos no se vean afectados, y de que sí, hay cambios en el dominio, los elementos externos a este, capa de aplicación y capa de infraestructura, se vean lo más mínimamente afectados. En nuestro caso, vamos a proponer un ejemplo de cómo hubiera sido una implementación de esta arquitectura para nuestro proyecto:


*Figura 67: Arquitectura Hexagonal para el negocio de Vela*

Crearemos carpetas por cada una de nuestras entidades (User, Boat, etc…). Cada entidad tendrá tres capas: Dominio, aplicación e infraestructura.

En la capa de dominio vamos a incluir nuestro modelado de datos y nuestro repositorio asociado a él. En nuestro repositorio aparecerán métodos como búsqueda o creación, pero sin realizar su implementación (clase abstracta). El concepto de repositorio hace que todo lo que se vaya a persistir en nuestra base de datos debe pasar por nuestro patrón repositorio.

En la capa de aplicación vamos a incluir todo lo relativo a nuestros casos de uso según entidad.

En la capa de infraestructura vamos a incluir todo lo relativo a la implementación con librerías o servicios externos, como en nuestro caso puede ser *mongoose*. Lo que se busca con esto es adaptar nuestras capas por si cambian sus requisitos en un futuro, para poder realizar el mínimo número de modificaciones en las capas más internas. Un ejemplo de esto puede ser la migración a una nueva base de datos, simplemente tendríamos que hacer la implementación a la nueva base de datos (como ejemplo, *MySQLBoatRepository*) donde vamos a incluir todas las adaptaciones que tenía la anterior implementación de *MongooseBoatRepository*, con esto, y gracias a una buena inyección de dependencias, tendremos las mínimas modificaciones posibles de nuestra nueva implementación de repositorio en nuestro proyecto.

Una buena práctica para el modelado de datos es usar el concepto de *ValueObject.* Este concepto nos permite encapsular elementos de nuestro dominio que representan un valor por sí mismo, por ejemplo, si queremos que una cadena X cumpla ciertos criterios, podemos crear

nuestra propia implementación de cadena, pero centrado en nuestro criterio o negocio. Los *ValueObject* deberían de estar lo más cerca posible a la capa de dominio para intentar mantener la integridad de nuestros datos en base de datos. Algunos beneficios que nos aportan para nuestro día a día son:

- **Inmutabilidad**: Los *Value Objects* son inmutables, lo que significa que una vez que se crean, no pueden ser modificados. Esta inmutabilidad puede ayudar a prevenir errores y hacer que el código sea más fácil de entender, ya que no hay que preocuparse por el cambio de estado.
- **Legibilidad**: Al dejar de utilizar primitivos y empezar con clases, vamos a darle nombres semánticos. También vamos a tener métodos dentro de los *Value Objects* que nos ayuden a entender mejor el código gracias a su nombre.
- **Reutilización**: Los *Value Objects* actúan de imán de nuestro código. Desde el momento que hemos modelado una clase, cualquier lógica que le pertenezca va a ir allí dentro en lugar de estar duplicada en sitios o con alguna abstracción metida por calzador.

Para obtener el contexto completo, puedes visitar el repositorio de vela-api-ddd-hexagonal[32].

[32] Vela API POC DDD y Arquitectura Hexagonal en Typescript

## 12.2 Anexo B: Objetivos de Desarrollo Sostenible

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

| **Objetivos de Desarrollo Sostenibles** | **Alto** | **Medio** | **Bajo** | **No Procede** |
|---|---|---|---|---|
| ODS 1. **Fin de la pobreza.** | | | | **X** |
| ODS 2. **Hambre cero.** | | | | **X** |
| ODS 3. **Salud y bienestar.** | | | | **X** |
| ODS 4. **Educación de calidad.** | | | | **X** |
| ODS 5. **Igualdad de género.** | | | | **X** |
| ODS 6. **Agua limpia y saneamiento.** | | | | **X** |
| ODS 7. **Energía asequible y no contaminante.** | | | | **X** |
| ODS 8. **Trabajo decente y crecimiento económico.** | **X** | | | |
| ODS 9. **Industria, innovación e infraestructuras.** | | **X** | | |
| ODS 10. **Reducción de las desigualdades**. | | | | **X** |
| ODS 11. **Ciudades y comunidades sostenibles**. | | | | **X** |
| ODS 12. **Producción y consumo responsables.** | | | **X** | |
| ODS 13. **Acción por el clima.** | | | | **X** |
| ODS 14. **Vida submarina.** | | | | **X** |
| ODS 15. **Vida de ecosistemas terrestres.** | | | | **X** |
| ODS 16. **Paz, justicia e instituciones sólidas.** | | | | **X** |
| ODS 17. **Alianzas para lograr objetivos.** | | | | **X** |

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

La transformación digital ha revolucionado la forma en que las empresas operan y se relacionan con sus clientes. En este contexto, el desarrollo de una nueva aplicación para pequeñas empresas se convierte en una oportunidad para poder impulsar el crecimiento económico. La implementación de soluciones de software punteras y la automatización de procesos repetitivos son elementos clave para poder crear un impacto en la empresa, mejorando la eficiencia y productividad de los empleados, lo que se traduce a una mejor economía para la empresa y la posibilidad de invertir este dinero en otros aspectos de empresa.

Es por esto por lo que este proyecto se relaciona principalmente con el objetivo de desarrollo sostenible, Trabajo decente y crecimiento económico, ya que se plantea una solución o línea posible al desarrollo de una nueva aplicación para pequeñas empresas. Esta tendría un gran impacto en la empresa de manera positiva, ya que la implementación se traduciría en un mayor crecimiento económico y la creación de nuevo empleo de calidad.

Además, la implementación de pagos en línea y la detección de errores pueden mejorar la experiencia de usuario de forma general y aumentar la confianza en la aplicación, lo que puede tener un impacto positivo en el éxito de la empresa.

La implementación de soluciones de software que utilicen tecnologías punteras puede fomentar la innovación (ODS 9, Industria, innovación e infraestructuras), esto se traduce en mejores infraestructuras, industrias más tecnificadas y una mayor innovación que podría repercutir de manera significativa en la economía de un país. En el desarrollo de esta aplicación siempre se ha apostado por la digitalización, innovación, mejora de la calidad de trabajo y el uso de las tecnologías actuales más punteras.

Es importante destacar que la aplicación desarrollada en este proyecto se ha concebido desde sus inicios con una visión sostenible. La apuesta por la digitalización, la innovación y el uso de tecnologías punteras permite optimizar recursos y minimizar el impacto ambiental. La reducción de la dependencia de procesos físicos y el fomento de prácticas de producción y consumo responsables se alinean con el ODS 12, Producción y consumo responsables, y contribuyen a construir una sociedad más sostenible.

En resumen, este TFG se relaciona principalmente con el ODS 8, Trabajo decente y crecimiento económico, debido a su especial enfoque al desarrollo de una aplicación para start-ups o pequeñas empresas que quieran crecer su producto. Sin embargo, también puede contribuir indirectamente al ODS 9, Industria, innovación e infraestructuras, al promover la implementación de soluciones tecnológicas avanzadas y punteras en el desarrollo de la aplicación como puede ser la automatización de tareas repetitivas usando flujos de trabajo CI/CD que impulsa la modernización de las infraestructuras tecnológicas y fomentar la innovación en el sector empresarial. Al haber creado un enfoque de desarrollo sostenible que se refleja en la implementación de pagos en línea, lo cual implica la reducción de transacciones físicas en papel y la promoción de prácticas financieras más eficientes y seguras, este TFG, también se relaciona con el ODS 12 (Producción y consumo responsables).

# Glosario

## *B*

***backend***
El backend es el encargado de procesar toda la información que alimenta a un frontend. Se compone de marcos, bases de datos o códigos. Para que un sitio web o aplicación opere efectivamente, se requiere mucha información y datos que se almacenan en «la parte trasera» de un sistema informático. En oposición al frontend, el usuario no puede ver o acceder a esta información. 10, 19, 22, 23, 34, 35, 50, 60, 70, 71, 72

## *C*

***callback***
es una función que se pasa a otra función como un argumento, que luego se invoca dentro de la función externa para completar algún tipo de rutina o acción 26, 27

***commit***
Captura una instantánea de los cambios preparados en ese momento del proyecto. 56, 62

## *F*

***framework***
Un framework es un esquema o marco de trabajo que ofrece una estructura base para elaborar un proyecto con objetivos específicos, una especie de plantilla que sirve como punto de partida para la organización y desarrollo de software. 15, 23, 38, 39, 59, 69, 72

***frontend***
El frontend o «desarrollo del lado del cliente» se refiere a la práctica de producir HTML, CSS y JavaScript. Estos tres elementos se encargan de dar forma a la parte frontal de un sitio web o aplicación. Esto incluye los fondos, colores, texto, animaciones o efectos. 20, 21, 34, 39, 50, 60, 66, 71, 72

## *H*

***hash***
Una función resumen,  en inglés hash function,  también conocida con los híbridos función hash o función de hash, convierte uno o varios elementos de entrada a una función en otro elemento 27

## *P*

***pull request***
Un Pull Request es una función de GitHub que permite a los colaboradores solicitar la revisión y aprobación de sus cambios antes de fusionarlos en la rama principal de desarrollo. 56, 61, 62, 72

## *S*

***salt***
En criptografía, una sal (en inglés salt) es un conjunto de bits aleatorios que se usan como una de las entradas en una función derivadora de claves. La otra entrada es habitualmente una contraseña. La salida de la función derivadora de claves se almacena como la versión cifrada de la contraseña. Una sal también puede usarse como parte de una clave en un cifrado u otro algoritmo criptográfico 27

## *W*

***webhook***

Un webhook es una función de devolución de llamadas que se basa en el protocolo HTTP para que dos interfaces de programación de aplicaciones (API) se comuniquen mediante eventos de forma ligera 35, 36, 37

***workflow***

Un workflow, o flujo de trabajo en español, es un conjunto de actividades relacionadas, que son completadas en un determinado orden para alcanzar un objetivo de la organización. 4, 56, 57, 58, 59, 60, 61