Aplicación web de katas de código

David Marchena Blanco
Grado de Ingeniería Informática
Desarrollo web

Gregorio Robles Martínez
Santi Caballe Llobet

12/06/2020

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Aplicación web de katas de código*
Nombre del autor:	*David Marchena Blanco*
Nombre del consultor/a:	*Gregorio Robles Martínez*
Nombre del PRA:	*Santi Caballe Llobet*
Fecha de entrega (mm/aaaa):	*06/2020*
Titulación:	*Grado de Ingeniería Informática*
Área del Trabajo Final:	*Desarrollo web*
Idioma del trabajo:	*Castellano*
Palabras clave	*JavaScript* *DDD* *TDD*

Resumen del Trabajo:

La finalidad de este Trabajo es construir una aplicación web para adquirir y afianzar conocimientos de programación en JavaScript a través de ejercicios. Servirá para cubrir una necesidad vista en el ámbito de la empresa: mejorar la comprensión del lenguaje a las nuevas incorporaciones de una forma práctica.

Se utiliza una metodología de diseño guiado por el dominio para construir una arquitectura limpia. En la misma debe existir una clara separación de intereses entre capas y desacoplamiento entre la lógica de aplicación y los detalles de infraestructura.

Se asegura la calidad mediante automatizaciones, analizadores de código y siguiendo un desarrollo TDD.

El MVP resultante cumple con el propósito y con los objetivos de experiencia de usuario y de rendimiento web.

El autor ha cubierto sus objetivos de aprendizaje y experiencia en el uso de TDD y DDD.

Abstract:

The aim of this project is to develop a web application for improving the coding skills and the understanding of JavaScript by means of coding katas. It will meet the needs of an enterprise in new hires onboarding.

A Domain-driven Design is followed in order to build a clean architecture. There

must be a separation of concerns between layers. Besides, the business logic has to be abstracted out from infrastructure.

With the purpose of assuring the quality of the product some QA tools are used: linters, formatters… In addition, Test-driven development is chosen as development process.

This resulting product is a MVP (minimum viable product) that fulfills the main goal of the project. Furthermore, the targeted UX and web performance have been achieved.

The author accomplished his goals of learning and gaining experience working with TDD and DDD.

A mi pareja y a mi hija.
Gracias por tu apoyo, Arrate.
Sin ti nada de esto habría sido posible.
Por las horas de juego que te debo, Lur.

Índice

Lista de figuras

Lista de tablas

CAPÍTULO 1

Introducción

Este trabajo de fin de grado (en adelante TFG) trata del desarrollo de una aplicación web de katas de programación en Javascript.

El término “kata” se refiere a ejercicios que ayudan a mejorar las habilidades de un programador. En este proyecto, las katas podrán validarse de forma automática cumpliendo una serie de aserciones predefinidas.

Por otro lado, se permitirá agrupar varias katas en módulos, de tal forma que puedan crearse diferentes tutoriales con fin educativo.

1.1 Objetivo principal

La idea para este TFG surge de conversaciones con compañeros de trabajo, sobre cómo ayudar a los recién llegados a mejorar y afianzar sus conocimientos de JavaScript (en adelante JS).

La popularidad alcanzada en los últimos años y su omnipresencia en el desarrollo web hace que muchos profesionales tengan que hacer frente a tareas relacionadas con JS. Su principal ventaja, la versatilidad para hacer cualquier cosa, es a la vez su mayor debilidad (Gómez 2019). Si le sumamos una relativamente rápida curva de aprendizaje para realizar ciertas funcionalidades, tenemos un caldo de cultivo perfecto para dar por sentado muchos conocimientos y caer en malas prácticas al afrontar tareas más complicadas.

De ahí que el principal objetivo sea el disponer de una herramienta práctica y sencilla con la que poder reafirmar conocimientos sin abandonar a los desarrolladores frente a una lista de libros o enlaces a sitios web.

1.1.1 Subobjetivos

- Desarrollar el TFG siguiendo una metodología TDD[1] y, como comenta Martin Fowler, esperar alcanzar una cobertura superior al 80% (Fowler 2012). La motivación para ello reside en conocer empresas donde las pruebas automáticas se consideran un coste adicional y se dejan fuera del flujo de desarrollo. De hecho la tónica habitual es que los planes de pruebas manuales se realicen al final de la entrega de un proyecto y, durante el mantenimiento o en nuevas iteraciones, se pruebe sólo la parte afectada, provocando en ocasiones errores debidos a efectos colaterales. En este proyecto quiero comprobar si el coste adicional de TDD se compensa con calidad y el ahorro en la solución de errores.
- Montar un entorno de desarrollo que disponga de las herramientas necesarias para una implementación ágil del proyecto: *linters*, *testing* automatizado, servidor de desarrollo con *live-reload*...
- Conseguir una experiencia de usuario (en adelante UX[2]) satisfactoria, para lo cual, además de un diseño de interacción intuitivo, será importante un rendimiento web aceptable, tanto en tests tipo Lighthouse[3] como en rendimiento percibido.
- Conseguir que la UX del editor de código online sea lo más parecida posible a uno de escritorio. Es importante cuidar la accesibilidad para que la aplicación web sea usable mediante teclado, ya que el usuario objetivo (desarrolladores) está acostumbrado a manejarse únicamente con teclado.
- La idea inicial era que el sitio web fuese estático, pero con el objetivo de poner a prueba los conocimientos adquiridos a lo largo del Grado, se implementará un gestor web para crear los diferentes retos y colecciones.
- Preparar una imagen de docker con el *backend* necesario (Linux, Nginx, Node.js, BBDD). De esta manera podremos tener un entorno de trabajo limpio y el producto final será posible probarlo en cualquier máquina o desplegarlo a un proveedor como DigitalOcean.
- Diseñar la aplicación utilizando DDD con fines de aprendizaje y práctica, debido al interés del autor del TFG en el diseño dirigido por el dominio y las arquitecturas que intentan desacoplar la lógica de una aplicación de los detalles técnicos y de infraestructura, como: *clean*

---

1 *Test-driven development*

2 *User experience*

3 *Lighthouse scoring guide: <https://developers.google.com/web/tools/lighthouse/v3/scoring#perf-color-coding>.*

architecture, *hexagonal architecture*, *onion architecture*...

1.2 Tecnologías que se van a utilizar

1.2.1 Cliente

1.2.1.1 Vue.js

Es un framework Javascript progresivo para construir interfaces de usuario. El término progresivo se refiere a que la librería incluye sólo las funcionalidades básicas de renderizado y sistema de componentes, pudiendo añadirse otras mediante otras librerías.

Ha sido elegido frente a otras alternativas muy extendidas como React o Angular por su rápido renderizado, la agilidad en su desarrollo e incluir de base la funcionalidad de *mixin*, que favorece la composición y la reutilización de código. A pesar de que React ha hecho un gran esfuerzo en este aspecto con la inclusión de sus Hooks[4], el enfoque en este y otros aspectos de Vue resulta más amigable al desarrollador.

1.2.1.2 ES2020

Es el nombre abreviado de ECMAScript 2020, la última revisión del estándar acordada por el Ecma TC39 (*Ecma International, Technical Committee 39 – ECMAScript*). Es decir, la última versión de lo que comúnmente se conoce como Javascript.

En 2015 apareció la versión ES6, que supuso una revolución al estándar ES5 que llevaba prácticamente inalterado desde 2009. Desde el estándar ha seguido un desarrollo más ágil, con una versión cada año. Es por ello que se abandonó el número de versión por el año de la publicación.

En este proyecto se utilizará el último estándar, garantizando la compatibilidad con navegadores mediante el uso de un transpilador.

Se ha decidido que los navegadores soportados serán aquellos que dan soporte casi completo a ES2015[5], de forma que su motor javascript nos

4 *https://reactjs.org/docs/hooks-intro.html*

5 *ES6 compatibility table <https://kangax.github.io/compat-table/es6>*

permita enviar un código final de la aplicación más óptimo y ofrecer katas en dicho lenguaje sin tener que hacer transpilación en tiempo de ejecución.

1.2.1.3 CSS

Son las siglas para referirse a *Cascading Style Sheets*, el lenguaje para enriquecer la presentación de un documento HTML.

Al igual que ocurre con ECMAScript, CSS es un estándar vivo que cada vez presenta novedades más interesantes. Por ello en el código de la aplicación incluirá especificaciones modernas, incluyendo:

- *Candidate Recommendations*
- *Working Drafts*
- y posiblemente algún *Editor Draft* que evite utilizar preprocesadores CSS alejados del estándar como Sass.

1.2.1.4 Axios

Es un cliente HTTP basado en promesas.

Una promesa es un objeto que representa la terminación o error de una operación asíncrona y evita el uso de los clásicos *callbacks*.

Un abuso de *callbacks* puede acabar en lo que se conoce como el *Callback hell* o *Pyramid of Doom*, un antipatrón que aparece al encadenar muchos niveles de operaciones asíncronas. En cambio, las promesas ofrecen un código más limpio y *debuggeable*, además de otras funcionalidades impensables para *callbacks,* como poder esperar a la finalización de una lista de promesas.

1.2.2 Servidor

1.2.2.1 Node.js

Entorno en tiempo de ejecución, de código abierto, basado en el motor V8 de Google. Gracias a él, el código Javascript podrá ser ejecutado tanto en servidor como en cliente, pudiendo reutilizar código de la aplicación en ambos entornos.

Además, al instalarse en el equipo utilizado para escribir el código fuente, permitirá el uso de herramientas de desarrollo como servidor

local, *linters*, *bundlers*…

1.2.2.2 Express

Es un framework de aplicaciones web para Node.js.

En un inicio se estudió como alternativa el uso de Fastify por su prometedor rendimiento, pero finalmente fue descartado porque Express dispone de multitud de documentación oficial y no oficial, amplio soporte y lleva años como solución estable.

1.2.2.3 NGINX

Aunque pueda pensarse que un servidor Node.js es *production-ready*, instalar por delante un *reverse proxy* proporciona ciertas ventajas. En nuestro caso nos interesa por mejoras en el rendimiento para servir estáticos, encriptación SSL, compresión… (Hunter 2019)

Se ha decidido usar NGINX frente a Apache por su mejor rendimiento rendimiento: hasta 2,5 veces más rápido al servir contenido estático (Goyal 2019).

1.2.2.4 Postgres

En un principio se dudó entre utilizar una base de datos relacional tradicional o una NoSQL. Ésta última es muy útil cuando se quiere comenzar a usarla pronto de forma iterativa y disponer de un esquema flexible. Este no es el caso de este TFG, ya que se utilizará un modelo en cascada y el desarrollo de la aplicación debería comenzar una vez finalizada la fase de diseño.

Sin embargo, desde su versión 9.2, Postgres soporta de forma nativa el tipo de datos JSON con un rendimiento nada desdeñable (Bonefeste 2019). Eso ofrece lo mejor de ambos enfoques en una opción solida y estable como Postgres.

Además, dispone de soporte para UUID (*Universally Unique Identifiers*), lo cual resulta muy útil para una arquitectura basada en DDD, ya que permite crear la clave de una nueva entidad sin que sea responsabilidad exclusiva de la base de datos[6].

6 *A priori parece que sea más óptimo usar un numérico incremental como clave de una entidad, y puede que sea así. Pero también podría darse el caso de que sólo sea una micro-optimización.* («Is using a UUID as a primary key in Postgres a performance hazard?» 2017)

1.2.3 Stack tecnológico para desarrollo

1.2.3.1 VSCodium

Desde la aparición de SublimeText en 2007, muchos IDEs han seguido su estela. Pero no había aparecido ninguno que, para su desarrollador objetivo, pudiera hacerle sombra. Hasta que apareció Visual Studio Code (VSCode) de Microsoft y empezó a ganarle terreno por méritos propios.

VSCode es código libre y está construido sobre Electron, Node y HTML/CSS. Sin embargo su rendimiento es mucho mejor comparado con otros IDE de tecnología similar, como Atom. Además tiene un ecosistema de plugins muy extenso.

Por todo ello, se desarrollará utilizando VSCodium (proyecto libre, basado en Microsoft VSCode y con todas sus ventajas), cuyo binario es publicado bajo licencia MIT y está libre de la telemetría y *tracking* de Microsoft.

1.2.3.2 Git

Git es un software de control de versiones distribuido muy extendido y familiar para todos aquellos que hayan utilizado GitHub o GitLab. Sus puntos fuertes son:

- Alta seguridad e independencia por sus repositorios locales.
- Flujo de trabajo mas eficiente mediante sus ramas locales y remotas.
- Mejor rendimiento que un sistema centralizado.
- Es gratuito y *open-source*.

1.2.3.3 Webpack

Un proyecto como éste necesita de un bundler para generar los estáticos necesarios para la aplicación web. Las opciones consideradas han sido Webpack y Parcel.

El más veterano de los tres es Webpack. De uso muy extendido pero con una curva de aprendizaje más pronunciada debido a la complejidad que puede alcanzar su configuración.

Parcel surgió como una alternativa rápida y sin configuración explícita. Lo que hace que tras su instalación pueda comenzar a usarse rápidamente. Esto ha hecho que gane mucha popularidad.

Finalmente, se ha decidido utilizar Webpack para poder ajustar la configuración a las necesidades del proyecto y adquirir más experiencia en el bundler con mayor cuota en el mercado.

1.2.3.4 Babel

Como se ha comentado anteriormente, el uso de ES2020 hace necesario el uso de un transpilador de código para dar el soporte adecuado a los navegadores elegidos.

En este terreno, no sería posible estudiar otra alternativa diferente de Babel.

1.2.3.5 Postcss

Una definición sencilla para postcss sería que es similar a Babel pero aplicado a CSS. Sin embargo es más que eso, es un preprocesador CSS basado en plugins, por lo que su aplicación puede ir desde transpilar CSS estándar a navegadores sin soporte nativo hasta optimizar, añadir estructuras y bucles de control en el CSS, integrar imágenes, verificar un estilo de código…

Antes de postcss, ya existían preprocesadores como Less o Sass. No obstante obligaban a escribir las hojas de estilo en otro lenguaje, dificultando la migración. Con postcss, en cambio, está en manos del desarrollador elegir si seguir el estándar (pudiendo eliminar la dependencia de postcss en el momento en que su código se soportado por los navegadores objetivo) o alejarse de él todo lo que sea necesario para un determinado proyecto.

1.2.4 Herramientas para documentación

1.2.4.1 LibreOffice

Para la redacción de la memoria y las diferentes entregas es necesario un procesador de textos lo suficientemente completo.

LibreOffice es actualmente la solución libre y de código abierto que tomó el relevo a OpenOffice como la alternativa por excelencia a Microsoft Office.

1.2.5 Zotero

A pesar de que LibreOffice dispone de la funcionalidad para gestionar una bibliografía, el resultado no llega a cumplir su objetivo.

Zotero, en cambio, es gestor libre de referencias bibliográficas muy ágil y completo con grandes ventajas:

- Es estable y tiene una buena experiencia de usuario.
- Dispone de una herramienta muy útil: añadir un libro junto con todos sus datos a partir de un identificador, como el ISBN, entre otros.
- Soporta multitud de notaciones bibliográficas.
- Cuenta con un plugin para la integración con LibreOffice.

1.2.6 Pencil[7]

Se trata de una herramienta de prototipado gratuita y de código abierto con soporte para OS X.

Inicialmente se elige para el desarrollo de los *wireframes* de la aplicación pero una vez instalada se observa su utilidad para crear diagramas y gráficos durante el diseño de la arquitectura.

1.3 Planificación temporal

Como bien se pregunta Jeffries sobre las estimaciones, ¿es prever mejor que dirigir? (Jeffries 2015). Seguramente él opinaría que en un proyecto de este calado lo ideal sería desarrollar siguiendo una metodología ágil e ir sumando valor al producto en cada sprint. Siguiendo un desarrollo *feature-by-feature* el producto podría evolucionar e ir adaptándose en base a la experiencia adquirida durante el desarrollo y la satisfacción de los usuarios finales.

De todas formas, dado que este proyecto se realizará bajo un marco académico en el que hay designados cuatro hitos de entrega, se utilizará un modelo en cascada. Así, el proyecto se dividirá en cuatro etapas: Plan de trabajo, Análisis y diseño, Desarrollo y Documentación.

7 *Pencil 3.1.0 <https://pencil.evolus.vn/>*

Tarea	Fase	Inicio	Fin	Dias	Horas	Descripción
PEC1		20/feb	06/mar		39h	Plan de trabajo
PT1	Plan	20/feb	24/feb	5	15h	Elección del proyecto y objetivos
PT2	Plan	25/feb	25/feb	1	3h	Preparación de plantilla de documento
PT3	Plan	25/feb	26/feb	2	7h	Redacción de documento y elección de tecnologías
PT4	Plan	28/feb	29/feb	2	6h	Captura inicial de requisitos (historias de usuario)
PT5	Plan	01/mar	02/mar	2	8h	Planificación y riesgos
PEC2		09/mar	10/abr		96h	Primer hito
ANA1	Análisis	09/mar	09/mar	1	1h	Describir stakeholders identificados
ANA2	Análisis	09/mar	10/mar	2	7h	Casos de uso UML a partir de historias
ANA3	Análisis	15/mar	16/mar	2	8h	Diagrama de casos de uso UML
ANA4	Análisis	17/mar	18/mar	2	6h	Modelo de dominio
DIS1	Diseño	19/mar	24/mar	6	24h	Diseño de la arquitectura de la aplicación
DIS2	Diseño	28/mar	30/mar	3	12h	Wireframes
DIS3	Diseño	31/mar	01/abr	2	8h	Selección final de tecnologías en base al análisis y tras estudiar alternativas: BBDD, ORM o ODM, bundler, servidor de aplicaciones…
DEV1	Desarrollo	02/abr	07/abr	6	30h	Documentarse y preparar entorno de desarrollo: linters (estilo de código, accesibilidad...), test runner, test automático Lighthouse, transpilación de ES2020 y >css3, bundler, servidor de desarrollo con hot reloading…
PEC3		13/abr	29/may		153h	Segundo hito
DEV2	Desarrollo	13/abr	20/abr	8	32h	Documentarse y preparar docker para *back end* local
DEV3	Desarrollo	21/abr	21/abr	1	3h	Preparar datos de ejemplo para la base de datos y mocks

DEV4	Desarrollo	22/abr	26/abr	5	20h	API *back end*
DEV5	Desarrollo	29/abr	04/may	6	24h	Editor online (editor, consola)
DEV6	Desarrollo	05/may	07/may	3	12h	Ejecutar código fuente de usuario y comprobar aserciones
DEV7	Desarrollo	08/may	10/may	3	12h	Gestión de katas
DEV8	Desarrollo	11/may	13/may	3	12h	Gestión de módulos
DEV9	Desarrollo	14/may	20/may	7	30h	Parte pública
DEV10	Desarrollo	21/may	22/may	2	8h	Registro y login
DEV11	Desarrollo	23/may	23/may	1	4h	Gestión de preguntas tipo test
DEV12	Desarrollo	24/may	25/may	2	8h	Plan de pruebas
PEC4		01/jun	12/jun		36h	Memoria final
MEM1	Docs	01/jun	08/jun	8	32h	Redacción de memoria y correcciones
MEM2	Docs	09/jun	09/jun	1	4h	Presentación
TFG					324h	

Tabla 1: Planificación

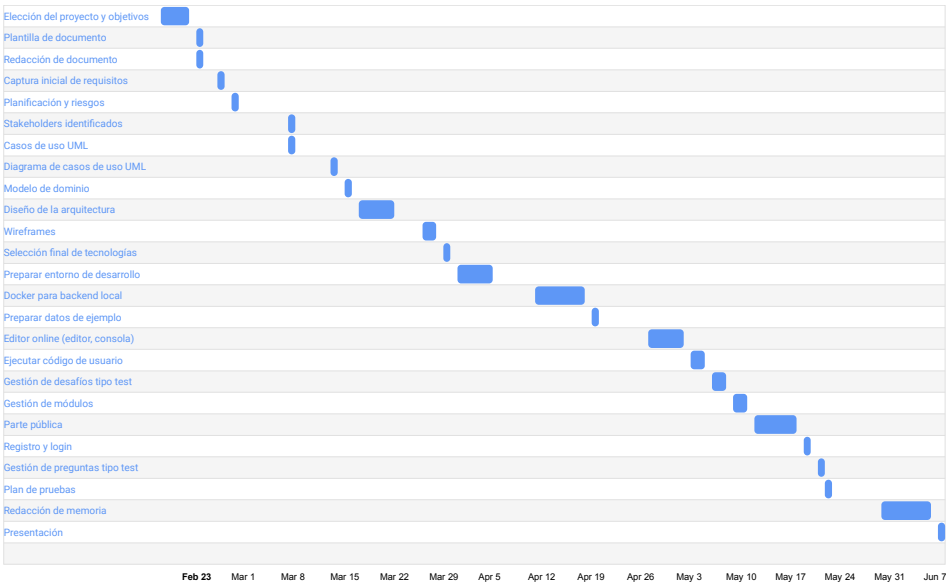

Figura 1: Diagrama de Gantt

1.4 Evaluación de riesgos

Riesgo	Impacto	Probabilidad	Medidas
Cargas familiares	Alto	Alta	Evitarlo adelantando trabajo los fines de semana o festivos de la madre. Mitigarlo eliminando funcionalidades, requisitos de menor valor o revisando alcance de tareas.
Cargas laborales	Medio	Alta	Mitigarlo eliminando funcionalidades, requisitos de menor valor o revisando alcance de tareas.
Mala planificación (riesgo negativo)	Alto	Medio	Mitigarlo eliminando funcionalidades, requisitos de menor valor o revisando alcance de tareas.
Mala planificación (riesgo positivo)	Medio	Medio	Explotarlo mejorando la calidad o añadiendo nuevos requisitos.
Requisitos excesivos	Medio	Bajo	Definir y revaluar con precisión requisitos que hayan podido quedar poco detallados o con un alto grado de exigencia y cuyo impacto no sea crítico.
Problemas relacionados con el desarrollo	Medio/ Bajo	Medio	Evitarlo mediante la búsqueda de documentación. Mitigar el impacto mediante la búsqueda alternativas tecnológicas o redefinición del requisito dependiendo de su importancia.
Pérdida o avería del ordenador de trabajo	Alto	Baja	Evitarlo mediante la subida habitual de la documentación y el código fuente del TFG a un repositorio remoto.

Tabla 2: Riesgos detectados

CAPÍTULO 2

Análisis

2.1 Stakeholders

Se han detectado los siguientes stakeholders:

- Usuario (o visitante)
- Usuario registrado
- Administrador
- Desarrollador

2.1.1 Usuario

Son todos aquellos visitantes del sitio web que acceden por primera vez para practicar o intentar superar una kata. Su principal interés es adquirir conocimiento, para lo cual el administrador debe proveerles de katas de calidad y con soluciones suficientemente instructivas. A parte del contenido, es indispensable que la experiencia de usuario fomente que prueben nuevos retos. Los puntos críticos de la UX serán la velocidad de carga, una interfaz atractiva, un diseño de interacción intuitivo y un editor de código lo más parecido a su aplicación de escritorio.

2.1.2 Usuario registrado

Un usuario es aquel que ha decidido registrarse para guardar sus avances en un módulo o kata. Los intereses de este *stakeholder* son el poder retomar en otro momento sus prácticas principalmente y quizá el disponer de un histórico o estadísticas de sus módulos completados.

2.1.3 Administrador

Serán los encargados de crear las diferentes katas y módulos que estarán disponibles en la parte pública. La gestión tiene que ser usable y les facilite la tarea. Por otro lado, la aplicación debe darles las herramientas suficientes para comprobar que la solución de código que proponen es correcta. Dado que dar de alta una kata puede suponer bastantes datos, es de su interés que no se puedan eliminar por error o en su defecto que pueda recuperarse la información.

2.1.4 Desarrollador

Al tratarse de un proyecto académico, el autor de este trabajo es también un *stakeholder*. Entre sus intereses están el producir un producto de calidad junto con la puesta en práctica de los conocimiento del Grado y su ampliación a otros nuevos. De ello derivarán principalmente requisitos de proceso, como por ejemplo el uso de TDD.

2.2 Obtención de requisitos

En primera instancia se realiza un *brainstorming* para obtener una lista de requisitos candidatos y asignarles un valor y un coste.

El valor de un requisito se encuentra entre una escala de 1 (muy bajo) y 5 (muy alto). Mientras, el coste estimado va de 1 (muy poco costoso) a 5 (muy costoso).

Dichas estimaciones servirán para priorizar o desechar aquellos de menor relación valor/coste durante las fases de análisis y diseño o en caso de incurrir en alguno de los riesgos descritos en “Evaluación de riesgos” que comprometa planificación descrita anteriormente en el apartado “Planificación temporal”

2.2.1 Requisitos funcionales

Descripción	Código	Valor	Coste
Stakeholder: Usuario			
Como usuario, quiero ver un listado de katas sueltas o módulos (grupos de katas)	RF1	5	2
Como usuario, quiero poder filtrar entre katas o	RF2	3	1

módulos			
Como usuario, quiero poder filtrar por nombre	RF3	3	1
Como usuario, quiero poder filtrar por etiquetas	RF4	2	2
Como usuario, quiero iniciar una kata	RF5	5	4
Como usuario, quiero comprobar si he hecho bien la kata	RF6	5	4
Como usuario, quiero ver la solución y saber qué he hecho mal en una kata que no haya superado	RF7	5	3
Como usuario, quiero volver a intentar la kata (o módulo) desde el principio	RF8	3	1
Como usuario, quiero poder guardar el estado de la kata (o módulo) sin registrarme y retomarlo más tarde	RF9	2	4
Como usuario, quiero registrarme y guardar el resultado del reto (o módulo)	RF10	3	3
Stakeholder: Usuario registrado			
Como usuario registrado, quiero poder ver las katas y módulos que he hecho	RF11	2	3
Como usuario registrado, quiero volver a intentar una kata que ya he hecho	RF12	2	1
Como usuario registrado, quiero poder cambiar mi contraseña	RF13	3	1
Como usuario registrado, quiero ver estadísticas de mis katas (tiempo, porcentaje de acierto, intentos…)	RF14	2	5
Como usuario registrado, quiero poder recuperar el acceso si olvido mi contraseña. Funcionalidad “He olvidado mi contraseña”.	RF15	3	5
Stakeholder: Administrador			
Como administrador, quiero gestionar katas aisladas	RF16	5	5
Como administrador, quiero gestionar un módulo (grupo de katas)	RF17	4	4
Como administrador, quiero poder añadir preguntas de tipo test a un módulo	RF18	3	2
Como administrador, quiero publicar (hacer visible) una kata o módulo	RF19	5	1
Como administrador, quiero archivar (no disponible para nuevos intentos) una kata o módulo	RF20	2	1

Como administrador, quiero eliminar una kata o módulo	RF21	3	1
Como administrador, quiero añadir documentación previa y/o posterior a una kata a través de markdown	RF22	4	3
Como administrador, quiero ver los usuarios registrados	RF23	1	2
Como administrador, quiero ver estadísticas de una kata o módulo	RF24	2	5

Tabla 3: Requisitos funcionales

2.2.2 Requisitos no funcionales

Descripción	Código	Valor	Coste
Requisitos de usabilidad y humanidad			
Navegación fluida entre las diferentes katas o preguntas de un modulo (sin recarga completa)	RNF1	4	2
Conseguir que la UX del editor de código online sea lo más parecida posible a uno nativo de escritorio	RNF2	5	5
Requisitos de cumplimiento			
Conseguir un rendimiento web de al menos 90[8] en Lighthouse.	RNF3	3	3
Permitir tolerancia a fallos de red.	RNF4	2	4
Requisitos operacionales y de entorno			
El sitio web será usable en todos los dispositivos. Aunque la mejor UX de un editor de código se obtiene en un ordenador y no se recomendará su uso, los módulos con sólo preguntas tipo test tienen en los móviles un claro *target*.	RNF5	3	2
El sitio web soportará las últimas dos versiones *major* de los principales navegadores: Chrome, Firefox, Safari, Edge, Chrome Android y Safari iOS.	RNF6	1	1
El *back end* de la aplicación estará disponible como imagen de Docker, facilitando el desarrollo en nuevos equipos y su posterior despliegue a servidores	RNF7	4	3

8 *Lighthouse scoring guide: 90 to 100 = fast* <*https://developers.google.com/web/tools/lighthouse/v3/scoring#perf-color-coding*>.

La web deberá ser usable sólo con teclado debido a que el usuario objetivo son desarrolladores, muchos de los cuales están acostumbrados a trabajar sólo con teclado para agilizar su trabajo. Aunque sea deseable lograr un nivel AA según lo especificado en las WCAG 2.1. no será un requisito.	RNF8	3	3
Requisitos de seguridad			
El acceso a la zona privada de usuario y la administración se realizará por email y contraseña.	RNF9	4	2
El sitio web usará exclusivamente HTTPS	RNF10	4	2
La ejecución del código generado por el usuario para un ejercicio no podrá jamás ejecutarse en el servidor, a pesar de que el servidor Node.js sea capaz, para evitar agujeros graves de seguridad.	RNF11	3	1
La ejecución del código generado por el usuario deberá ejecutarse en cliente en un sandbox seguro para mejorar la seguridad.	RNF12	4	4
La contraseña se guardará codificada.	RNF13	3	1

Tabla 4: Requisitos no funcionales

2.2.3 Requisitos de proceso

Descripción	Código	Valor	Coste
Desarrollar siguiendo una metodología TDD	RP1	4	2
Conseguir una cobertura de tests de al menos 80%	RP2	3	2
Disponer de linting automatizado de código	RP3	4	1
Disponer de test runner automatizado para los test unitarios	RP4	2	1
Disponer de un test automático para validar el requisito RNF3 (Lighthouse)	RP5	2	2
Disponer de un servidor de desarrollo dinámico que reaccione a modificaciones en el código de cliente (hot reloading)	RP6	4	3
Desarrollar siguiendo un diseño dirigido por el dominio (DDD)	RP7	5	4

Tabla 5: Requisitos de proceso

2.2.4 Requisitos descartados

En primera instancia se ha decidido descartar una serie de requisitos del *brainstorming*:

- RF9: Inicialmente parece sencillo implementarlo a través del LocalStorage del navegador. Sin embargo, el valor que aporta no es mucho y se prevé que los ajustes necesarios tengan un coste superior. Posiblemente se pueda abordar si el desarrollo va por delante de la planificación.
- RF14, RF24: Aunque a priori parece muy interesante el recoger datos y mostrar estadísticas tanto al usuario como al administrador, su coste en tiempo sería excesivo tanto en la captura de datos como en una visualización atractiva (gráficas, etc). Además es una funcionalidad que no está en el *core* del producto esperado.
- RF15: Se descarta este requisito por el coste en tiempo asociado. Queda fuera del TFG montar un servidor de correo para implementar la funcionalidad “He olvidado mi contraseña”. De cara a una implantación en producción sería necesario (o disponer de una alternativa de login mediante OAuth).
- RF23: Se descarta en inicio este requisito por el escaso valor que aporta si no existen acciones relacionadas con la gestión de usuarios (ver estadísticas, cambiar contraseña...)
- RNF4: Se elimina el requisito aunque se desarrollará teniéndolo en cuenta. La comunicación cliente-servidor no es muy constante, por lo que no es un requisito crítico.

2.3 Casos de uso

2.3.1 Acceder a una kata

Los requisitos funcionales RF1-4 y RF11-12 pueden englobarse en un único caso de uso:

Caso de uso: Acceder a una kata

Actor principal: Usuario

Escenario principal de éxito:

1) El usuario pide ver las katas (y módulos) disponibles.
2) El sistema muestra un listado de katas con nombre y etiquetas asignadas a cada una.
3) El cliente accede a una kata (o módulo).
4) El sistema muestra la información inicial de la kata. Sólo los detalles introductorios, no el código.

Extensiones:

1a) En caso de estar identificado, el usuario pide ver sólo los que ya ha realizado.
2a) El usuario pide solo ver las katas aisladas.
2a1) El sistema muestra un listado de katas aisladas.
2b) El usuario pide solo ver los módulos.
2b1) El sistema muestra un listado de módulos.
2c) El usuario pide ver solo aquellos con una determinada etiqueta.
2c1) El sistema muestra un listado cumpliendo la condición.
2d) El usuario introduce un texto
2d1) El sistema reacciona y muestra un listado que variará a medida que el usuario escribe.

2.3.2 Realizar una kata o módulo

De los requisitos funcionales RF5-8 extraemos dos casos de uso: uno para katas aisladas y otro para módulos.

Caso de uso: Realizar una kata

Actor principal: Usuario

Precondición:

El usuario se encuentra en ya en la página de la kata

Escenario principal de éxito:

1) El usuario comprueba la información inicial de la kata y pide iniciarla.
2) El sistema muestra el editor online y una consola de resultados.
3) El usuario pide testar el código.
4) El sistema muestra en la consola los resultados del test.
5) El usuario pide comprobar la kata.
6) El sistema ejecuta el código y felicita al usuario.
7) El sistema le ofrece la opción de identificarse o registrarse para guardar el resultado. Se da paso a los respectivos casos de uso si así lo desea el usuario.

Extensiones:

6a) El sistema ejecuta el código, el cual arroja un error, e informa al usuario por consola. Muestra además la opción de abandonar y ver la solución.

6a1) El usuario modifica su código y vuelve al paso 5.

6a1a) El usuario pide abandonar.

6a1a1) El sistema muestra la solución.

Caso de uso: Realizar un módulo

Actor principal: Usuario

Precondición:

El usuario se encuentra en ya en la página del módulo

Escenario principal de éxito:

1) El usuario comprueba la información inicial del módulo y pide iniciarlo.
2) El sistema muestra una kata (en primera instancia la marcada como inicial) y se pasa al caso de uso “Realizar una kata”.
3) El usuario pide pasar al siguiente ejercicio del módulo. Si quedan, se vuelve al paso 2.
6) El sistema muestra un resumen del módulo y el porcentaje de aciertos.

Extensiones:

2a) El sistema muestra una pregunta tipo test.

2a1) El usuario elige una de las respuestas propuestas.

2a2) El sistema valida la respuesta y muestra si ha sido correcta o no.

2.3.3 Registro de usuario

Caso de uso: Registro

Actor principal: Usuario

Escenario principal de éxito:

1) El usuario pide registrarse.
2) El sistema muestra el formulario de registro.
3) El usuario escribe su email y la contraseña por duplicado (para comprobar que se ha rellenado correctamente).
4) El sistema guarda el nuevo usuario.

Extensiones:

3a) La contraseña no coincide, así que el sistema lo comunica. Se vuelve al paso 2.

3b) El email ya existe y el sistema se lo indica al usuario. Se vuelve al paso 2.

2.3.4 Identificación

Caso de uso: Identificación

Actor principal: Usuario

Escenario principal de éxito:

1) El usuario pide identificarse.
2) El sistema muestra el formulario de login, solicitando email y contraseña.
3) El usuario rellena el formulario y envía los datos.
4) El sistema inicia la sesión y vuelve a la página en la que estaba el usuario.

Extensiones:

3a) La contraseña no es correcta, así que el sistema lo comunica. Se vuelve al paso 2.
3b) El email no existe y el sistema se lo indica al usuario. Se vuelve al paso 2.

2.3.5 Cambiar contraseña

Caso de uso: Cambiar contraseña

Actor principal: Usuario registrado

Precondición:

El usuario se encuentra ya identificado

Escenario principal de éxito:

1) El usuario pide cambiar contraseña.
2) El sistema muestra solicita la contraseña actual y la nueva contraseña. Ésta última por duplicado.
3) El usuario rellena el formulario y envía los datos.
4) El sistema modifica los datos y comunica al usuario que la operación se ha realizado con éxito.

Extensiones:

3a) La contraseña no es correcta, así que el sistema lo comunica. Se vuelve al paso 2.
3a) La nueva contraseña, que se ha escrito dos veces, no coincide, así que el sistema lo comunica. Se vuelve al paso 2.

2.3.6 Gestión de katas

A partir de los requisitos RF16 y RF22 obtenemos el siguiente caso de uso:

Caso de uso: Gestión de katas

Actor principal: Administrador

Precondición:

El administrador ya se encuentra identificado

Escenario principal de éxito:

1) El administrador pide modificar una kata.
2) El sistema muestra y permite editar los datos actuales del *core* de la kata: código inicial, aserciones que debe superar y solución propuesta.
3) El administrador envía los datos.
4) El sistema comprueba que la solución cumple las aserciones y el código inicial falla.
5) El sistema muestra la documentación inicial y, de forma optativa, una documentación de la solución.
6) El administrador escribe la documentación.
7) El sistema solicita un nombre y permite añadir etiquetas para mejorar el filtrado de katas.

Extensiones:

1a) El usuario pide hacer un alta
1a1) El sistema no buscará datos y los formularios futuros aparecerán vacíos.

2.3.7 Gestión de módulos

A partir de los requisitos RF17, RF18 y RF22 obtenemos el siguiente caso de uso:

Caso de uso: Gestión de módulos

Actor principal: Administrador

Precondición:

El administrador ya se encuentra identificado

Escenario principal de éxito:

1) El administrador pide modificar una kata.
2) El sistema muestra el nombre del modulo, sus etiquetas asociadas y un listado ordenado de katas o preguntas tipo test.
3) El administrador pide añadir una kata, se inicia el caso de uso “Gestión de katas” y se vuelve al paso 2.
4) El sistema actualiza el listado de katas.
4) El sistema comprueba que la solución cumple las aserciones y el código inicial falla.

5) El sistema muestra la documentación inicial y, de forma optativa, una documentación de la solución.
6) El usuario escribe la documentación.
7) El sistema guarda los datos.

Extensiones:

1a) El administrador pide un alta de modulo.
1a1) Inicialmente el sistema mostrará los formularios vacíos de cada pantalla.
3a) El administrador pide añadir una pregunta tipo test
3a1) El sistema solicita un texto para la pregunta, las diferentes opciones y la solución correcta y, opcionalmente, una explicación de la solución.
3a2) El usuario envía los datos.
3a3) El sistema añade la pregunta y se vuelve al paso 2.

2.4 Diagrama de casos de uso

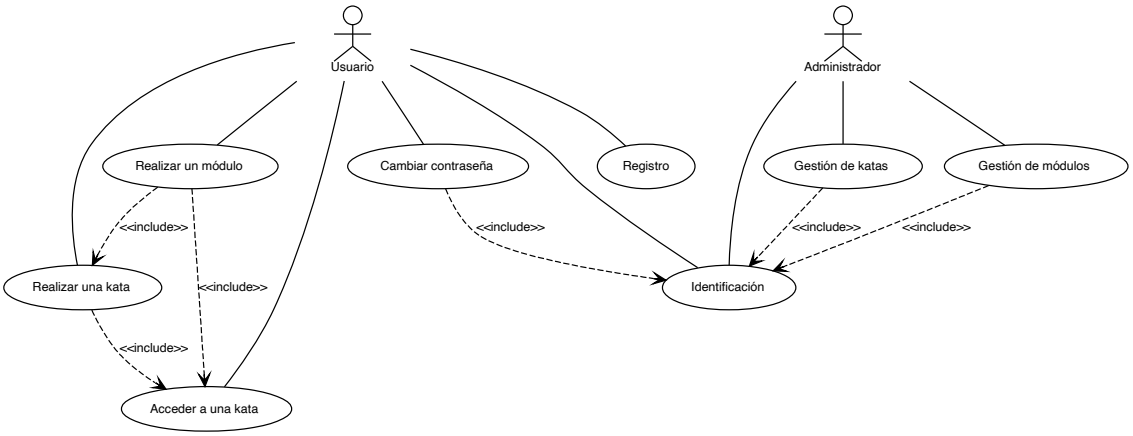

Figura 2: Diagrama de casos de uso

CAPÍTULO 3

Diseño

3.1 Diseño guiado por el dominio

Tal y como se describía en la captura de requisitos, para el diseño de la aplicación se va a intentar utilizar una metodología denominada *Domain-Driven Design* (a partir de ahora DDD) en vez de basarlo en una clásica arquitectura cliente-servidor en tres capas (presentación, aplicación, administrador de datos).

DDD puede aplicarse sobre una clean architecture y estructurar inicialmente en tres capas concéntricas:

- Dominio, que encapsula las entidades y las reglas de negocio de la aplicación.
- Aplicación, que contiene la lógica de la aplicación. Por un lado será la responsable de lidiar con los aspectos de infraestructura (transacciones, seguridad, envío de emails…) y, por otro, de llevar a cabo los diferentes casos de uso.
- Infraestructura, que proveerá a aplicación y dominio de adaptadores para interactuar de forma independiente con bases de datos, frameworks, interfaz de usuario… De esta forma conseguiremos que un cambio de infraestructura no repercuta en cambios en las demás capas.

Aplicando el principio de inversión de dependencias la capa de infraestructura se sitúa por encima de todas las demás, lo que la habilita para implementar interfaces de todas las capas inferiores y liberarlas de tener que conocer detalles de infraestructura y acoplarse a la misma.

Teniendo en cuenta que en la capa de infraestructura podría contener implementaciones de, por ejemplo, cliente de base de datos o interfaz de usuario (web, consola, servicio web…), es posible visualizar mejor el concepto de que las capas sean concéntricas. Así, un flujo que acabase

en una operación de escritura en base de datos implicaría: petición desde la interfaz de usuario (infraestructura) a un servicio (aplicación) para realizar una acción del modelo de negocio (dominio); tras los cambios en el modelo, el servicio (aplicación) se encarga de persistirlos en base de datos a través de la implementación de un repositorio (infraestructura).

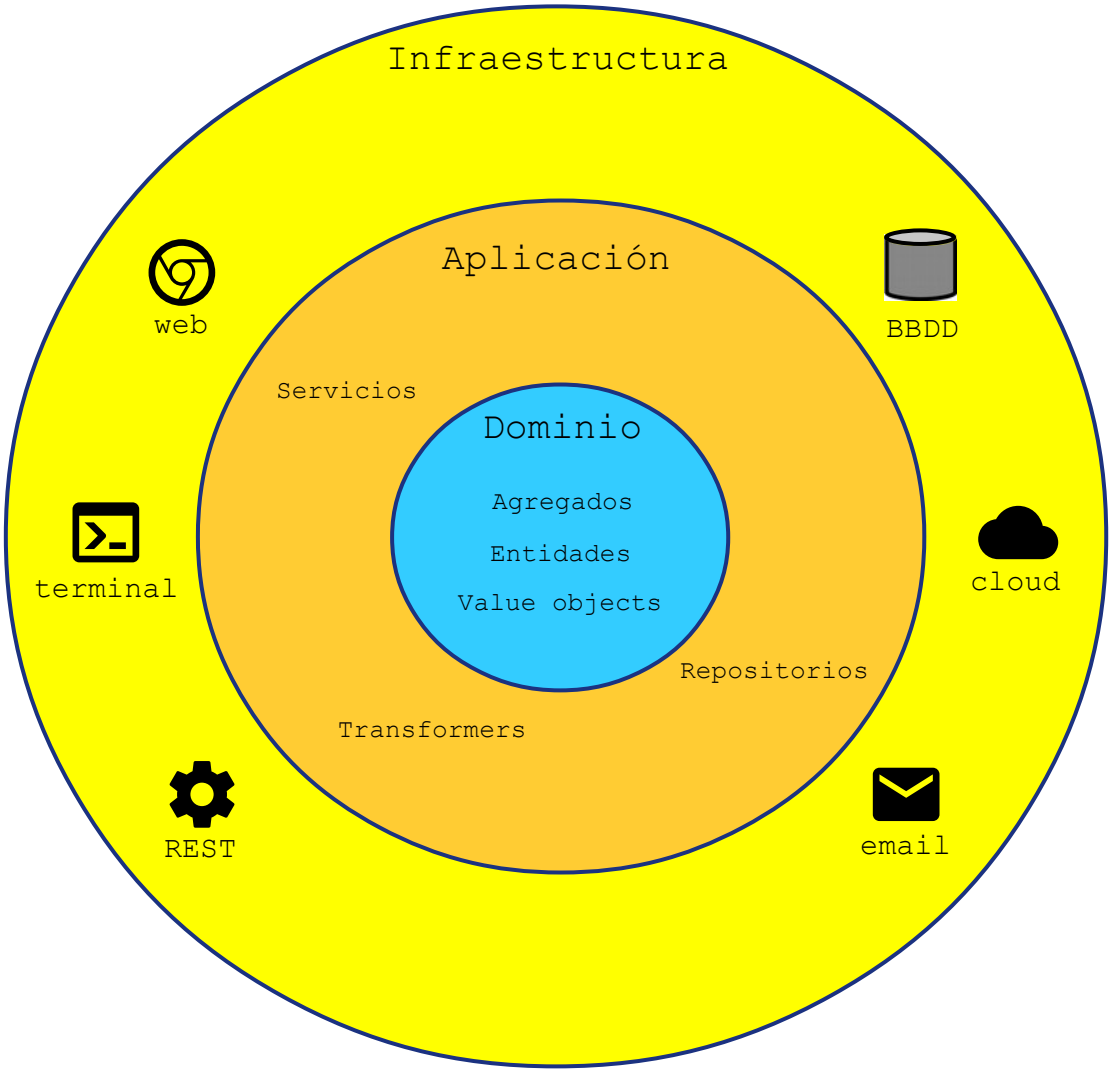

Figura 3: Arquitectura DDD en capas concéntricas

3.2 Modelo de dominio

Eric Evans cimentó DDD en la idea de que en un proyecto pueden

convivir diferentes contextos (*Bounded Contexts*), dentro de cada cual todos los actores (expertos en negocio, *product owners*, desarrolladores…) utilizarán un lenguaje ubicuo (*Ubiquitous Language*) para referirse al modelo y la lógica de negocio.

La primera acción será crear un modelo para el núcleo de la aplicación. Para ello DDD define tres tipos de objetos:

- *Entity*: Representa un concepto del modelo cuya individualidad viene determinada por un identificador único. Dos entidades del mismo tipo son diferentes si no poseen el mismo identificador, aunque todos sus atributos sean iguales.
- *Value Object*: Modela un concepto inmutable. Son sus atributos los que determinan su identidad y cuando se comparan dos se hace a través del valor de todos sus atributos. Un ejemplo de ello es una localización (latitud y longitud).
- *Aggregate*: Es una composición de una o más entidades. También puede contener *Value Objects* en su interior. Cada agregado posee una entidad raíz que determinará el nombre del mismo.

Para definir los agregados, se tendrán en cuenta las *Aggregate Rules of Thumb* (Vernon 2016, p. 81):

> *1. Protect business invariants inside Aggregate boundaries.*
> *2. Design small Aggregates.*
> *3. Reference other Aggregates by identity only.*
> *4. Update other Aggregates using eventual consistency.*

Otro punto útil a la hora de detectar agregados puede extraerse de “Patterns, principles, and practices of domain-driven design”, donde Millett expone que sólo los agregados pueden obtenerse directamente consultando a base de datos. Mientras, todos sus objetos de dominio internos sólo pueden ser accedidos a través del agregado. De esta forma, cargando completamente el agregado, se protege la integridad de su contexto. (Millett 2015, p. 458). Teniendo todo esto en cuenta, un primer diseño[9] del dominio de la aplicación sería el observado en la Figura 4: Core Domain inicial.

9 *Debe considerarse que los diseños de modelo presentados no son un diagrama de clases de UML aunque se asemejen a primera vista. Siguiendo las pautas de la bibliografía, no se desarrollan patrones, si no que se representan las relación existente entre conceptos del modelo de dominio.*

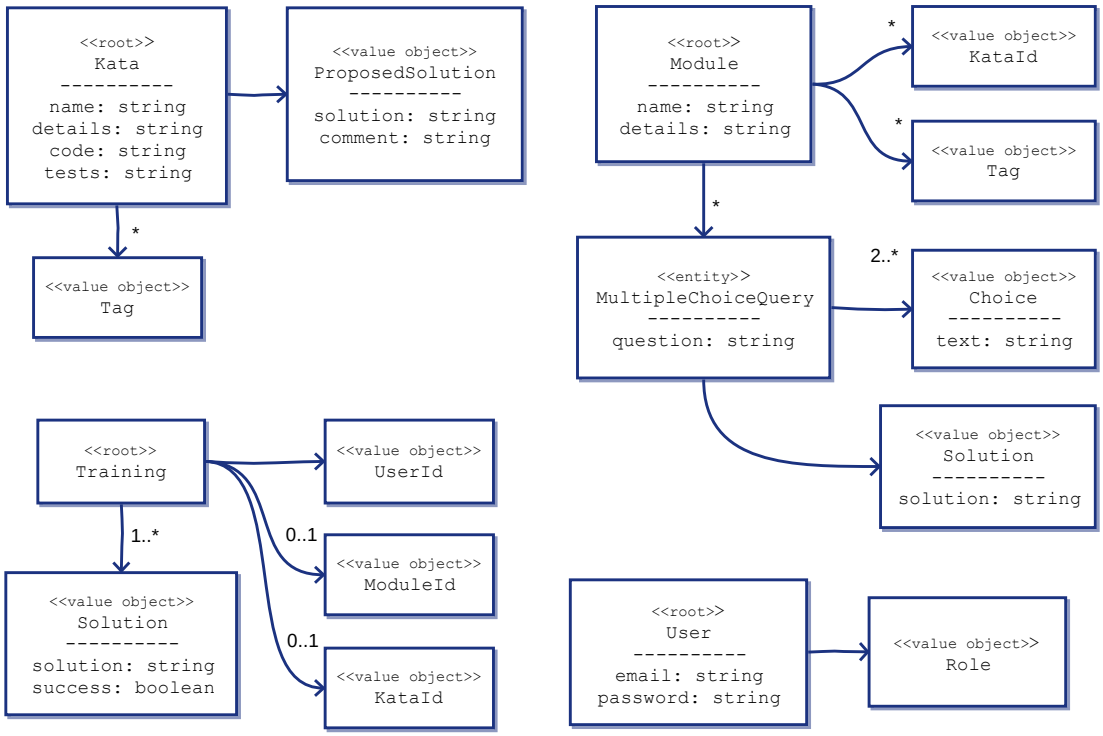

Figura 4: Core Domain inicial

Sin embargo, un punto en el que se hace especial hincapié al utilizar DDD es identificar el dominio principal de la aplicación. En este caso, si se analizan los casos de uso detectados se puede observar que identificación y registro no son muy diferentes de cualquier otra aplicación con usuarios. Por tanto, sería posible refactorizar el diseño utilizando lo que Evans denomina un núcleo segregado:

> *Elements in the model may partially serve the CORE DOMAIN and partially play supporting roles. CORE elements may be tightly coupled to generic ones. The conceptual cohesion of the CORE may not be strong or visible. All this clutter and entanglement chokes the CORE. Designers can't clearly see the most important relationships, leading to a weak design. (Evans 2003)*

Dentro de un dominio pueden existir tres tipos de subdominios:

- *Core Domain* o aquél que sirve el propósito principal de una aplicación, lo que la define en si misma.
- *Supporting Subdomain* o aquél que necesita de un desarrollo específico, pero de menor calado, ya que no es estratégico. Aunque sigue siendo un modelo importante sin el cual producto no sería

exitoso.

- *Generic Subdomain* o solución que puede venir de fuera de la plataforma o ser externalizada.

Dicho esto, el contexto de identificación y acceso podría tratarse de un subdominio genérico, como se observa en la siguiente figura:

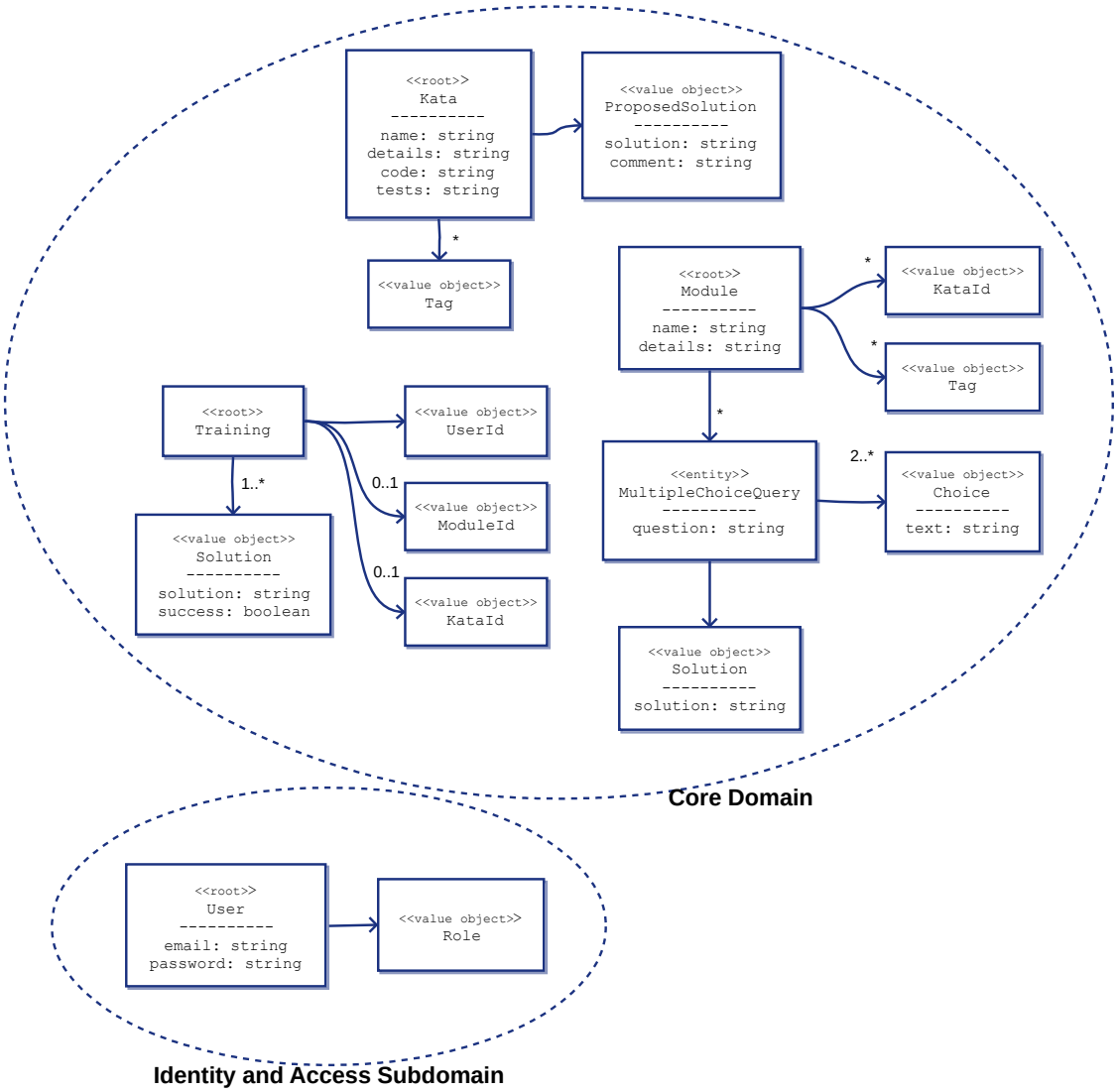

Figura 5: Núcleo segregado

3.3 Capa de aplicación

La capa de aplicación estará construida entorno a la capa del modelo y

será cliente de las entidades y agregados del ésta. Para Evans, esta capa está cimentada en los servicios de aplicación. Martin, en cambio, habla de casos de uso, pero desarrolla un concepto muy similar. Sin embargo, antes de entrar en dichos servicios, se examinarán otros conceptos de la capa de aplicación que serán de utilidad para aquéllos.

3.3.1 Repositorios

Normalmente, en DDD, un repositorio se refiere al almacenamiento y recuperación de agregados del dominio.

El almacenamiento se utiliza a través de inyección de dependencias bien en la capa de dominio o bien en la de aplicación, dependiendo del diseño elegido.

En este proyecto, se realizará es uso de repositorios se llevará a cabo en la capa de aplicación, aunque la implementación de los repositorios se realizará en infraestructura. Es por ello que esta capa se deberán crear los contratos y abstracciones para infraestructura.

Recopilando los agregados detectados en el diseño del dominio, la lista de repositorios necesarios quedará así:

- `kataRepo`
- `moduleRepo`
- `trainingRepo`
- `userRepo`

3.3.2 Factorías

En el punto anterior se ha visto la necesidad de establecer una serie de contratos para las dependencias que deben inyectarse desde la capa de infraestructura.

Como es sabido, en Javascript no existe el concepto de interfaz, es por ello que a medida que aquél ha ido ganando popularidad han aparecido *supersets*[10] como Typescript[11], que suple de tipos e interfaces a proyectos y desarrolladores para los que no encaja el sistema basado en

10 *Término inglés para referirse a lenguajes que se contruyen sobre otros a fin de extenderlos con funcionalidades no nativas.*

11 *Creado por Microsoft. Como muchos otros, al no disponer de soporte nativo en navegadores o Node.js, es necesario compilarlo a Javascript.* https://www.typescriptlang.org/

prototipos de Javascript.

Sin embargo, en este proyecto no se considera necesario el uso de Typescript. La complejidad que añadiría no compensa el disponer de soporte de interfaces.

Como se avanzó al comienzo de este documento, Javascript dota al desarrollador de una gran versatilidad y es posible definir un contrato sin utilizar una interfaz. Elliot en su libro “Programming JavaScript Applications” (Elliott 2014, p.73) desarrolla cómo utilizar funciones factoría para asegurar que se cumple una determinada interfaz.

De este modo, el diseño contemplará la creación de un directorio `factories` donde declarar todas las interfaces de las dependencias necesarias en esta capa. Posteriormente la capa de infraestructura utilizará dichas factorías para proveer a las capas inferiores de los repositorios necesarios.

3.3.3 Transformers

La capa de aplicación dispondrá además de una serie de módulos JS que transformarán objetos de la capa de dominio en los que finalmente consumirán los clientes de la capa de infraestructura.

Los principales propósitos de estos *transformers* serán: desacoplar infraestructura de dominio y evitar que desde el exterior se conozca o se acceda a funcionalidades internas de la capa de dominio. Cabe recordar que la capa de modelo no albergará un modelo anémico, sino que los objetos tendrán métodos para cumplir las reglas de negocio. En ocasiones, dichos métodos simplemente actuarán sobre los datos del objeto pero podría darse el caso que lanzaran eventos de dominio o tuviesen otros efectos secundarios. Es por ello que los objetos de dominio no viajarán fuera de la capa de aplicación sino que ésta proveerá a infraestructura de los DTO necesarios mediante los *transformers*.

De igual forma, en ocasiones será indispensable disponer de la transformación inversa, obtener un agregado a partir de un DTO. En tales caso, se implementará en el mismo *transformer*.

3.3.4 Application services

Los *application services* no son ficheros que albergan operaciones CRUD, sino métodos que ayudan a completar el flujo de un caso de uso. Su principal objetivo será el de encargarse de que los casos de uso

cumplan con los requisitos propios de la aplicación: almacenamiento, transacciones, transformación de datos para consumidores… Estos requisitos son ajenos a los del modelo de negocio y muchas veces transparentes para los expertos del modelo.

Existen muchas patrones de implementar los servicios de aplicación: Orientado a objetos, *Command Processor*, *Publish/Subscribe*, Petición/Respuesta... Dicho esto, no existe un ranking de mejores patrones y, “en muchas ocasiones, la solución más simple es una buena opción” (Millett 2015, p.701).

En la aplicación de katas, los servicios serán objetos de cada caso de uso que incluirán los métodos necesarios para realizar sus pasos.

Revisando los casos de uso detectados, se puede extrapolar la siguiente lista de servicios:

- `listService`
- `doKataService`
- `doModuleService`
- `manageKataService`
- `manageModuleService`
- `userService`

Tras revisar más detenidamente los casos de uso es posible extraer nuevas abstracciones que serán necesarias para los servicios:

- `authSession`, que permitirá a la aplicación conocer si un usuario se ha autenticado mientras dure la sesión. Deberá disponer de los siguientes métodos:
  - `saveAuthentication(user: object): void`
  - `isAuthenticated(): boolean`
  - `isAdmin(): boolean`
  - `discardAuthentication(): void`

- `codeTester`, de tal forma que la capa de infraestructura pueda proveer a las capas inferiores de la implementación necesaria para validar si el código de una kata pasa los test definidos para misma, sin importar si la ejecución del código se realiza en cliente, en servidor o en un servicio de terceros. Dispondrá inicialmente de un único método, el cual recibirá el código y lo tests a realizar junto con una función donde escribir la traza resultante de los tests si es necesaria:

- `test(code: string, tests: any, output: function): boolean`

Una vez identificadas estas abstracciones es posible listar los servicios de aplicación junto con sus dependencias y la firma de cada uno de sus métodos:

- `BrowseService`
  `(dependencias: kataRepo, moduleRepo)`
  - `getAllKatas(): kataDto[]`
  - `getAllKatasDoneByUser(userId: string): kataDto[]`
  - `getAllKatasWithTag(tag: string): kataDto[]`
  - `getAllModules(): moduleDto[]`
  - `getAllModulesDoneByUser(userId: string): moduleDto[]`
  - `getAllModulesWithTag(tag: string): moduleDto[]`
  - `getKataWithId(kataId: string): kataDto`
  - `getModuleWithId(moduleId: string): moduleDto`
- `DoKataService`
  `(dependencias: kataRepo, codeTester, authSession)`
  - `testCode(kataId: string, code: string): boolean`
  - `saveTraining(kataId: string, userId: string, solution: object): void`
- `DoModuleService`
  `(dependencias: moduleRepo, kataRepo, codeTester, authSession)`
  - `testSolution(multiChoiceQueryId: string, solution: string): boolean`
  - `saveTraining(moduleId: string, userId: string, solution: object[]): void`
- `ManageKataService`
  `(dependencias: kataRepo, codeTester, authSession)`
  - `testProposedSolution(proposedSolution: object, tests: any, output: function): boolean`
  - `save({ name: string, details: string, code: string, tests: string, tags: string[], proposedSolution: object }): void`
  - `update({ id: string, name: string, details: string, code: string, tests: string, tags: string[], proposedSolution: object }): void`
  - `delete(kataId: string): void`

- `ManageModuleService`
  `(dependencias: moduleRepo, kataRepo, codeTester, authSession)`
  - `addKata(moduleId: string, kataDto: object)`
  - `addMultipleChoiceQuery(moduleId: string, question: string, choices: string[], solution: string): string`
  - `save(moduleDto: object): void`
- `UserService`
  `(dependencias: userRepo, authSession)`
  - `login(email: string, password: string): boolean`
  - `singup(email: string, password: string): void`
  - `changePassword(password: string, oldPassword: string): void`
  - `logout(): void`

3.4 Capa de infraestructura

Como se ha avanzado en apartados anteriores, la capa de infraestructura se encargará de implementar las dependencias de las capas inferiores y los clientes que interactuarán con la capa de aplicación.

3.4.1 Clientes

Los clientes se implementarán en un directorio denominado `ui` (acrónimo inglés de *User Interface*). En este proyecto se considerarán como interfaces de usuario tanto el cliente web como la API REST que se desplegará en el servidor para persistir los datos en base de datos.

3.4.2 Repositorios

Los repositorios se agruparán por tipo dentro de una carpeta `repos`. Los tipos detectados en el diseño son:

- `db`: interacción con la base de datos en servidor.
- `api`: interacción cliente – API REST.
- `mock`: para testing.

Dentro de cada carpeta se incluirá un repo para cada agregado, que se

implementará a partir de su correspondiente factoría.

Para disminuir la interacción con la API y mejorar el rendimiento para el usuario, debería incorporarse una caché. Existen varias opciones para dicho propósito:

- Que sea el cliente web quien tenga la responsabilidad de controlar la caché y decidir si llamar o no a los servicios de aplicación correspondientes.
- Que sean los repositorios de API REST los que implementen internamente una caché.
- Implementar un nuevo tipo de repositorio `cacheableAPI` que extienda los de tipo API.
- Implementar una función a través de la cual sea posible componer varios repositorios del mismo agregado en uno solo, de tal forma las operaciones de escritura se ejecuten en todos los repositorios y las de lectura devuelvan el primer resultado disponible de la lista de repositorios.

Las dos primeras opciones añadirían complejidad junto con la nueva responsabilidad, por lo que a priori las otras parecen dar más robustez a la aplicación. Entre éstas, la última opción parece la más versátil, ya que sigue el segundo principio postulado por los Gang of Four[12] a mediados de los noventa: “Favorecer la composición de objetos sobre la herencia”. Según este principio, la herencia puede romper la encapsulación y favorecer que un cambio en la implementación del padre fuerce a modificar el hijo, lo que limita la flexibilidad y la reutilización del código.

Componer varios repositorios parece ser la opción que permita obtener los beneficios que Gang of Four comentan en su libro "Design Patterns: Elements of Reusable Object-Oriented Software" (Gamma et al. 1995):

- Mantener a cada componente centrado en una tarea.
- Disponer de componentes más pequeños y menos propensos a convertirse en monstruos inmanejables.
- Obtener un diseño más reutilizable, pudiendo, en este caso concreto, componer otros repositorios diferentes si surge la necesidad sin tener que crear nuevas jerarquías de objetos.

Dicho esto, se implementará un nuevo tipo de repositorio para gestionar la cache de cliente y una solución para poder componer sus repositorios

12 *Gang of Four es el nombre colectivo con el que comúnmente se conoce a los cuatro autores del clásico libro "Design Patterns: Elements of Reusable Object-Oriented Software".*

con los de API. Para ello se implementará la siguiente función:

```
composeRepos(… repos: repo[]): repo
```

Ésta utilizará un enfoque funcional parecido al que utiliza Elliot en el libro “Composing Software” (Elliott 2017): se crearán *pipelines* de funciones en las que el resultado de un método sea el input del siguiente método.

Teniendo en cuenta que la salida de un método no equivaldrá a la entrada del método correspondiente del siguiente repositorio será necesario envolverlos en una función que lo permita y además sea posible gestionar los dos tipos de flujos: lectura (se obtiene el resultado del primer repositorio que disponga) y escritura (se ejecutará la función de todos los repositorios).

Capítulo 4

Desarrollo del proyecto

4.1 Entorno de desarrollo

En este TFG existen bastantes requisitos de proceso que se refieren a automatizaciones para asegurar la calidad del producto obtenido así como del proceso de desarrollo:

- Desarrollar siguiendo una metodología TDD
- Conseguir una cobertura de tests de al menos 80%
- Disponer de linting automatizado de código
- Disponer de test runner automatizado para los test unitarios
- Disponer de un servidor de desarrollo dinámico que reaccione a modificaciones en el código de cliente (*hot reloading*)

La idea de este TFG es ir un paso más allá de instalar un IDE (como Eclipse, Netbeans o Visual Studio Code) y ponerse a programar, sino disponer de una configuración lo más portable posible que automatice (o al menos facilite) parte de la carga de trabajo del desarrollador. El objetivo principal es bajar la tasa de errores evitando derivar la responsabilidad de tareas automatizables a la mano humana.

Por otro lado, como ya se ha comentado, usar ES2020 o nuevas especificaciones de CSS hacen necesario el uso de transpiladores para asegurar el soporte a los navegadores indicados al inicio.

Todo esto hace del entorno de desarrollo una pieza clave para llevar al cabo el proyecto cumpliendo los objetivos especificados.

4.1.1 Formato de código

Para asegurar la consistencia del estilo de código se usarán:

- EditorConfig[13], para definir un estándar de formato de código independiente del IDE utilizado. Toda la configuración (indentación por tabulador o espacios, tamaño de la misma…) se centraliza en un único archivo `.editorconfig`.
- Prettier[14]: Es un formateador de código que actualizará la estructura del código de un archivo, con la posibilidad de automatizarlo cada vez que se guarde. De esta forma, el programador se libera de la responsabilidad de mantener un estilo de código consistente a lo largo proyecto. Prettier pretende ser dogmático y dejar sólo unas pocas reglas configurables en un archivo `.prettierrc`. El objetivo de esto es acabar con los debates sobre estilos de código en un equipo.

4.1.2 Linters

Un *linter* es una herramienta de análisis de código que permite detectar (y en ocasiones solucionar automáticamente) errores de programación, de estilo de código, bugs, patrones sospechosos de ser mejorados…

Para este proyecto se utilizarán ESLint[15] y Stylelint[16] para analizar Javascript y CSS respectivamente.

Ambos disponen de la posibilidad de usar plugins para extender su funcionalidad y de cargar configuraciones de terceros. Gracias a esto último no es necesario configurar una a una todas las reglas, sino importar las utilizadas por empresas como Google o Airbnb y usarlas directamente o como base para personalizar algunas reglas.

4.1.2.1 Configuración de ESLint

El archivo `.eslintrc` es el que recoge la configuración que se utilizará al analizar los archivos. Un aspecto muy interesante de este linter es que la configuración del mismo tiene un sistema jerárquico y en cascada, lo que permite crear nuevos `.eslintrc` dentro de un directorio que extiendan configuraciones de niveles superiores y se aplique dicha configuración solo para los descendientes de dicho directorio.

Para este proyecto se utilizarán las siguientes configuraciones:

---

13 *https://editorconfig.org/*

14 *https://prettier.io/*

15 *https://eslint.org/*

16 *https://stylelint.io/*

- `eslint:recommended`: Reglas recomendadas por ESLint.
- `eslint-config-airbnb`: Configuración seguida por equipo de desarrollo de Airbnb, muy prolífico en compartir buenas prácticas.
- `eslint-config-prettier`: Configuración que deshabilita reglas que entran en conflicto con Prettier.
- `plugin:jest/all`: Configuración de buenas prácticas al codificar tests para Jest.
- `plugin:vue/recommended`: Reglas recomendadas para el desarrollo con Vue.
- `plugin:vue-a11y/recommended`: Reglas para asegurar una accesibilidad mínima en componentes Vue.

4.1.2.2 Configuración de Stylelint

Dentro del archivo `.stylelintrc` se hará referencia a las siguientes configuraciones:

- `stylelint-config-standard`: Añade a las reglas base de Stylelint otras extraídas de guías de estilo CSS muy comunes: Google, Idiomatic CSS, Airbnb...
- `stylelint-a11y/recommended`: Para detectar patrones que perjudiquen a la accesibilidad del sitio web, como deshabilitar el resaltado del foco al navegar por teclado.
- `stylelint-config-rational-order`: Permite reordenar las diferentes declaraciones CSS en base a un orden concreto, en este caso el *rational order*[17]. Éste agrupa las declaraciones por grupos en base a su propósito: posicionamiento, modelo de caja, tipografía, visual, animación o miscelánea. Mucho mejor para el desarrollo que otros órdenes, como el alfabético.
- `stylelint-prettier/recommended`: Configuración que deshabilita reglas que entran en conflicto con Prettier.

4.1.3 TDD

Sobre la base de disponer una guía de estilo de código y linters que detecten posibles bugs se puede empezar a desarrollar código con una mayor seguridad.

---

17 *Agrupa las declaraciones por grupos en base a su propósito. posicionamiento, modelo de caja, tipografía, visual, animación o miscelánea. Mucho mejor para gestionar*

Aún así, se trabajará en la medida de lo posible siguiendo una metodología TDD para aumentar la robustez del código y hacer más manejables las refactorizaciones, lo que hace indispensable un framework de pruebas unitarias.

4.1.3.1 Jest

De entre las opciones disponibles en el mercado se ha decidido utilizar Jest[18] por ser una suite que cubre las necesidades del proyecto:

- VSCodium dispone de extensiones para trabajar con él y ejecutar automáticamente los test, permitiendo hacer TDD. (Requisito RP1)
- Genera informes de cobertura de código de forma sencilla. Además permite configurar unos límites mínimos de cobertura por debajo de los cuales el test lanza un error. (Requisito RP2)
- Provee de un sistema sencillo para sustituir dependencias por mocks.
- Permite extender los *matchers* de tests para generar evaluaciones que se repitan a lo largo del código. Muy útil en este proyecto para evaluar que un método devuelve un DTO o un modelo; o que un repositorio cumpla una determinada interfaz antes de comprobarlo en ejecución.

La configuración de Jest se realizará a través de dos archivos:

- `jest.config.js`: datos de configuración.
- `jest.setup.js`: código que se ejecutará antes de lanzar los test y que declarará los *matchers* específicos del proyecto.

Para mejorar el desarrollo con TDD y cambiar ágilmente entre código testado y test, éste se codificará en un archivo contiguo (`[filename].test.js`) al archivo objetivo (`[filename].js`).

4.1.3.2 Vue Test Utils

Para poder realizar pruebas unitarias sobre componentes de Vue.js se utilizará la librería oficial de *testing*.

Para integrar Vue Test Utils con Jest será necesario instalar vue-jest para que éste soporte los Single-File Components[19] de Vue.

---

18 *https://jestjs.io/*

19 *Un Single-File Component (SFC) es un archivo con extensión .vue que integra todo el código necesario para definir un componente de Vue. Los códigos HTML, Javascript y CSS se especifican respectivamentes en las etiquetas <template>, <script> y <style>.*

Vue Test Utils necesita de un navegador para poder montar los componentes y realizar las diferentes pruebas. Sin embargo, la recomendación oficial es utilizar JSDOM, un entorno implementado en puro javascript (y por tanto ejecutable en Node.js) que permite generar el DOM de un HTML e interactuar con él como en cualquier navegador. No llega a ser un navegador completo como PhantomJS, pero es ligero y muy útil para tareas de test.

Gracias a Jest no será necesario instalar JSDOM, porque ya se encarga el framework de instalarlo y ponerlo a punto. Bastará con indicar al inicio del test que Jest debe utilizarlo:

```
/**
 * @jest-environment jsdom
 */
```

4.1.3.3 Mocks

Los mocks son objetos que imitan el comportamiento de objetos reales de una aplicación de forma que puedan sustituir a éstos en un test. Son especialmente útiles cuando dichos objetos reales no son aptos para un test unitario porque, por ejemplo, no devuelven un resultado determinista, son costosos o difíciles de crear.

Jest hace que la implementación de mocks sea bastante sencilla. Aún así, para ciertos aspectos más complejos se deberán utilizar librerías extra que faciliten la tarea. Por ejemplo:

- `sequelize-mock` para simular objetos del ORM.
- `jest-mock-axios` para mocks de peticiones de cliente a través de Axios.
- `supertest` para poder realizar test de los endpoint de un servidor Express sin tener que levantar un servidor.

4.1.4 Transpilación de código

El primer paso para dicha tarea será el de crear un archivo `.browserlistrc` para especificar a los transpiladores de código los navegadores soportados por la aplicación:

```
# Browsers that we support
```

```
last 2 Chrome major versions
last 2 Firefox major versions
last 2 Safari major versions
last 2 Edge major versions
last 2 ChromeAndroid major versions
last 2 iOS major versions
```

4.1.4.1 Babel

Babel se encargará de transpilar el código ES2020 en Javascript inteligible para los navegadores. Sin embargo, para soportar algunas características modernas del lenguaje será necesario el uso de *polyfills*[20].

La mejor solución para añadir polyfills es core-js, una librería modular que permite a Babel incluir sólo los necesarios para los navegadores objetivo.

Así la configuración quedará en el archivo `.babelrc.js`:

```
module.exports = {
  presets: [
    [
      '@babel/preset-env',
      {
        useBuiltIns: 'entry',
        corejs: '3.6',
      },
    ],
  ],
};
```

4.1.4.2 Postcss

Se seleccionarán los plugins:

- `postcss-import`: permite separar la hoja de estilo en diferentes archivos parciales y utilizar *imports* para generar el código final.
- `autoprefixer`: añade nuevas declaraciones CSS con los prefijos propietarios necesarios (`-webkit-`, `-moz-`, `-ms-`, `-o-`).
- `postcss-preset-env`: permite convertir CSS moderno en uno que puedan entender los navegadores definidos en el browserlist. Se configurará para permitir utilizar Working Drafts.
- `cssnano`: para minificar el código.

20 *Código que implementa una funcionalidad que no es soportada por un navegador.*

La configuración se almacenará en el archivo `.postcssrc.js`:

```
module.exports = {
  plugins: {
    'postcss-import': {},
    autoprefixer: {},
    'postcss-preset-env': {
      stage: 2,
    },
    cssnano: {
      preset: 'default',
    },
  },
};
```

4.2 Infraestructura del servidor

En el desarrollo de aplicaciones web, la coordinación entre el área de sistemas y de desarrollo es fundamental. Cuanto menor sea, y es habitual que sean áreas bastante estancas entre sí, los problemas en el despliegue de nuevas versiones de un software pueden ser frecuentes.

Dichos errores son en gran medida responsabilidad del área de desarrollo por una batería de pruebas deficiente. Aún así, en ocasiones se debe a que la configuración del equipo local o del servidor de desarrollo difieren de los servidores de staging y producción. Independientemente de la responsabilidad, acaban con el despliegue de un error en producción y la consiguiente búsqueda entre logs en diferentes máquinas. Aunque pueda parecer lo contrario, son tan habituales que han hecho del *“It Works On My Machine”* uno de los memes por excelencia entre los profesionales del sector.

Para mejorar este aspecto del desarrollo de software surgió, entre otros, el movimiento DevOps: una metodología ágil basada en la integración de administradores de sistemas y desarrolladores. Se caracteriza, entre otras cosas, por incluir en el ciclo de vida de las aplicaciones:

- Integración continua (CI) y ejecución de pruebas automáticas cada vez que se sube código nuevo al control de versiones
- Entrega continua (CD) o despliegue automático y frecuente de nuevas versiones de la aplicación
- Infraestructura como código, para definir los sistemas de un modo que se puedan administrar de la misma manera que el código fuente de la aplicación y almacenarlo en un control de versiones. Esto además

permite implementar los sistemas de un modo confiable, repetible, automatizable y ajeno a errores humanos.

En resumen, el objetivo principal es que los desarrolladores se centren su tarea y puedan desplegar nuevas versiones en cuestión de segundos con menor grado de incertidumbre.

En este proyecto se persigue un objetivo similar: centrarse en el desarrollo evitando errores ocasionados por desarrollar bajo OS X e hipotéticamente desplegar en un servidor linux o por el simple hecho de diferir en la instalación manual de las diferentes herramientas en otra máquina. Es por ello que se tomará nota de cómo aborda la metodología DevOps para plantear la infraestructura.

Aunque no esté contemplada en el plan de trabajo, dado que se va a seguir una metodología TDD, no sería muy difícil integrar una herramienta de CI como en un puro DevOps,. Sin embardo, el uso de soluciones como Ansible, Chef, Terraform... para conseguir un CD se aleja mucho del alcance y del valor que aportaría a los objetivos del proyecto.

Toda la configuración se incluye dentro de un `vagrantfile` de manera que pueda replicar la misma en cualquier ordenador. Se consideró una solución factible para este proyecto al ser entregable, multiplataforma y fácilmente instanciable.

Finalmente, siguiendo un enfoque más práctico, se utilizará Docker como infraestructura como código. Toda la configuración se realiza en formato YAML en un `dockerfile`, lo que hace que sea versionable. Además, es fácilmente replicable en otra máquina de desarrollo o en un servidor final, ya que Docker tiene gran soporte y se integra muy bien con plataformas como Digital Ocean o Amazon ECS.

Docker se basa en contenedores de imágenes que encapsulan todo lo necesario para ejecutar una aplicación: código fuente, ejecución, herramientas de sistema, librerías y configuración. Como se puede observar en la Figura 6: Comparativa entre contenedores Docker y máquinas virtuales[21], los contenedores son una alternativa más ligera ya que no necesitan una copia completa del sistema operativo para cada aplicación. Todos los contenedores comparten el kernel de la máquina host, corriendo como procesos aislados.

21 *Fuente de la imagen:* «What is a Container?»

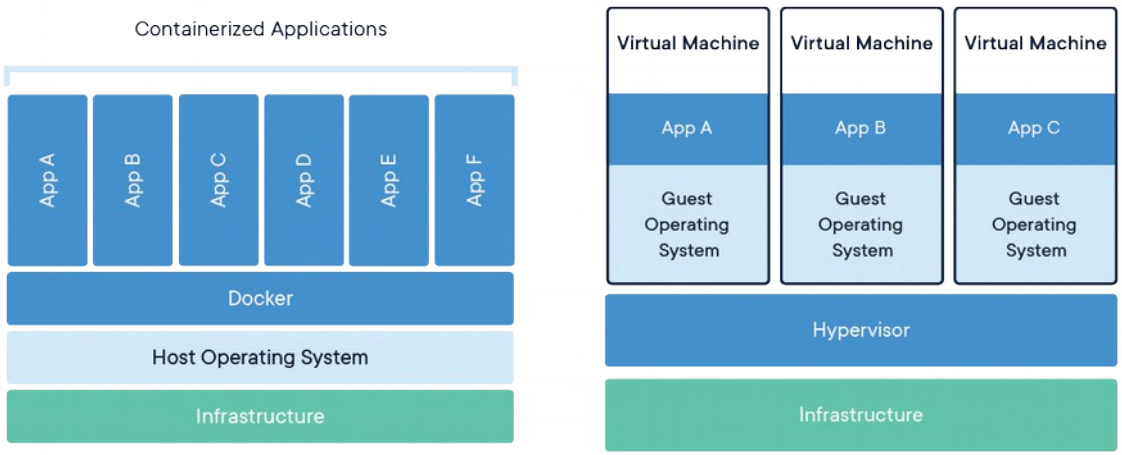

Figura 6: Comparativa entre contenedores Docker y máquinas virtuales

Usando la analogía de un lenguaje orientado a objetos es posible comprender mejor el funcionamiento de Docker. Una imagen de docker sería como la definición de una clase. Mientras, los contenedores generados a partir de aquélla podrían considerarse como instancias de una clase.

Por tanto lo primero que habrá que realizar será la definición de la imagen de Docker que servirá como servidor de la aplicación de katas.

4.2.1 Configuración de la imagen de Docker

Un Dockerfile debe comenzar con la instrucción FROM, que crea un nuevo escenario y establece la imagen base para las siguientes instrucciones.

Los requisitos para el servidor de la aplicación de katas son:

- Una distribución de linux como sistema operativo. Se utilizará Alpine linux, una distribución ligera y muy popular para imágenes Docker.
- Node.js, en su última versión LTS (*Long Term Support*).
- Nginx.
- Postgres, versión 9.2 o superior, para disponer de soporte a JSON.

En vez de partir de una base limpia de Alpine e instalar el resto de software, parece más conveniente hacerlo sobre una imagen oficial de Node.js sobre Alpine. De esta forma, la infraestructura partirá de una base más robusta. El hecho de elegir como base Node.js se debe a que la aplicación estará desarrollada para ejecutarse en dicho entorno,

mientras que el resto de herramientas (Postgres, Nginx) son decisiones puntuales y más factibles de sufrir cambios.

En este punto, la siguiente decisión a tomar es si instalar manualmente Postgres y Nginx sobre la misma imagen. Dado que la instalación de Postgres no es una tarea sencilla y existen imágenes oficiales con las diferentes versiones, lo más oportuno es utilizar Docker Compose[22], una herramienta para definir y ejecutar aplicaciones de múltiples contenedores. Esta solución aporta varias ventajas:

- Menor probabilidad de errores derivados de una instalación manual.
- Postgres y Nginx se ejecutarán en imágenes oficiales, con el consiguiente soporte y mantenimiento.
- Facilita la migración de la infraestructura a nuevas versiones o a soluciones alternativas. Por ejemplo, migrar de Nginx a Apache cambiando sólo la imagen que se encarga de ello.

En conclusión, la configuración del Docker que definirá la infraestructura del servidor se basará en dos archivos:

- `Dockerfile`: Definirá la imagen principal que albergará el código de la aplicación y estará basada en la imagen `node:14-alpine` para ejecutar la misma.
- `docker-compose.yaml`: Donde se especificarán los diferentes contenedores necesarios para la aplicación::
  - `app`: a partir de la imagen principal definida en el punto anterior.
  - `webserver`: servicio basado en la imagen `nginx:mainline-alpine`.
  - `postgres`: servicio basado en la imagen `postgres:12-alpine`.

4.2.2 Creación de un certificado SSL auto-firmado

Uno de los requisitos no funcionales de la aplicación es que usará HTTPS (RNF10). Por tanto, para el desarrollo del proyecto se deberá utilizar un certificado auto-firmado.

Para empezar habrá que generar una clave para la autoridad de certificación (a partir de ahora CA, según sus siglas inglesas).

22 *https://docs.docker.com/compose/*

```
openssl genrsa -des3 -out ca.key 2048
```

Será necesario rellenar los datos de la CA. Hecho esto, con la nueva clave será posible generar un certificado para la CA. Éste recibirá una validez de 10 años para evitar su caducidad durante el desarrollo.

```
openssl req -x509 \
  -new -sha256 -nodes -days 3650 \
  -key ca.key \
  -out ca.crt
```

Una vez lista la CA, es momento del certificado del servidor. El primer paso será el de generar la clave privada del mismo y una CSR (*Certificate Signing Request*) para que la CA pueda generar el certificado:

```
openssl req \
  -new -sha256 -nodes -out server.csr \
  -newkey rsa:2048 -keyout server.key \
  -config server.csr.cnf
```

En este paso será necesario disponer de un archivo configuración de OpenSSL `server.csr.cnf` para poder importar los datos del servidor sin tener que introducirlos en la terminal:

```
[ req ]
default_bits       = 2048
prompt             = no
default_md         = sha256
distinguished_name = req_distinguished_name

[ req_distinguished_name ]
C            = ES
ST           = State/County/Region
L            = City
O            = Organization
OU           = OrganizationUnit
emailAddress = hi@devdomain.com
CN           = localhost or dev domain
```

Una vez se disponga de la CSR se podrá generar el certificado X.509 v3 auto-firmado:

```
openssl x509 -req \
  -in server.csr \
  -CA ca.crt -CAkey ca.key -CAcreateserial \
```

```
  -out server.crt -days 825 -sha256 -extfile v3.ext
```

En esta ocasión será necesario disponer de un nuevo archivo de configuración para definir extensiones del certificado como por ejemplo nombres alternativos para los que el certificado es valido. Su formato se asemejará al siguiente ejemplo:

```
authorityKeyIdentifier=keyid,issuer
basicConstraints=CA:FALSE
keyUsage = digitalSignature, nonRepudiation, keyEncipherment,
dataEncipherment
subjectAltName = @alt_names

[alt_names]
DNS.1 = domain.com
DNS.2 = www.domain.com
DNS.3 = app.domain.com
```

Tras completar este paso bastará con añadir la clave privada y el certificado del servidor al contenedor de Nginx y añadir, entre otras configuraciones de ssl, los datos del certificado en el archivo `nginx.conf`:

```
ssl_certificate /etc/ssl/certs/server.crt;
ssl_certificate_key /etc/ssl/private/server.key;
```

En caso de utilizar un dominio de desarrollo en lugar de `localhost` se añadirá en el archivo de `hosts` del ordenador utilizado para desarrollar.

Para simplificar el proceso se crearán unos *shell scripts* y se dejarán en una carpeta `scripts` del proyecto.

4.2.3 Configuración de Nginx

Debido a los requisitos no funcionales RNF3 y RNF10 se decide utilizar HTTP/2.

Por un lado, HTTP/2 incrementa el rendimiento. Entre otras mejoras, es capaz de reutilizar una conexión y multiplexar varias peticiones en la misma. De esa forma se evita el bloqueo Head-of-Line de anteriores versiones, por el que un navegador disponía de un numero limitado de conexiones y nuevas peticiones debían esperar a que alguna anterior liberase una conexión.

Por otro lado, los navegadores actuales decidieron soportar el nuevo

protocolo sólo sobre una capa TLS[23], una versión más segura que el antiguo SSL. No es para nada un problema en el caso de esta aplicación, entre cuyos requisitos está el funcionar solo bajo HTTPS.

Para habilitar HTTP/2 es necesario especificarlo en el archivo `nginx.conf` e indicar la ruta del certificado y la clave privada del servidor:

```
server {
  listen 443 ssl http2;
  listen [::]:443 ssl http2;

  ssl_certificate /etc/ssl/certs/server.crt;
  ssl_certificate_key /etc/ssl/private/server.key;

  ssl_protocols TLSv1.2 TLSv1.1 TLSv1;
}
```

Para cumplir completamente con el requisito RNF10 se configura el servidor para redirigir siempre a HTTPS:

```
server {
  listen 80;
  listen [::]:80;

  location / {
    rewrite ^ https://$host$request_uri? permanent;
  }
}
```

En la configuración del puerto 443 se incluye además la del proxy inverso que redirija las peticiones al contenedor docker en el que se encuentra la aplicación. Para ello se utiliza con la directiva `proxy_pass`.

```
location @app {
  proxy_pass http://app:3000;
  …
}
```

En el cliente se va a utilizar un router para implementar una *single page application (SPA)*, es decir, se mapearán diferentes urls con componentes vue, sin intervención del servidor. Dichas urls serán exclusivas de cliente y no tendrán reflejo en el servidor. Para evitar que al recargar la página el servidor responda con un 404, Nginx se encargará de redirigir toda petición a la raíz del proxy (donde escucha la SPA) excepto las que vayan a la API REST, a un *asset* o al endpoint del

23 *Transport Layer Security: sucesor del protocolo SSL para la comunicación segura basada en certificados digitales.*

hot reloading del servidor de desarrollo:

```
location = / {
  try_files $uri @app;
}

location /api {
  try_files $uri @app;
}

location /__webpack_hmr {
  try_files $uri @app;
}

location ~* \.(.*)$ {
  try_files $uri @app;
}

location / {
  rewrite ^/(.+)$ / break;
  try_files $uri @app;
}
```

4.3 Base de datos

4.3.1 ORM: Sequelize

La creación y explotación de la base de datos se realizará a través de un ORM. En concreto se utiliza Sequelize[24], un ORM para Node.js basado en promesas.

Es necesario definir un archivo `.sequelizerc` como el siguiente:

```
require('@babel/register');
const path = require('path');

module.exports = {
  'config': path.resolve(
    'src/infrastructure/db',
    'config'
  ),
  'migrations-path': path.resolve(
    'src/infrastructure/db',
    'migrations'
  ),
  'models-path': path.resolve(
    'src/infrastructure/db',
```

24 *https://sequelize.org*

```
    'models'
  ),
  'seeders-path': path.resolve(
    'src/infrastructure/db',
    'seeders/development'
  )
}
```

En el código anterior se puede vislumbrar los diferentes tipos de módulos que necesita el ORM:

- Una configuración para poder conectarse al servidor de base de datos.
- Los modelos, para que el ORM sea capaz de transformar entidades de base de datos en objetos del ORM y viceversa.
- *Migrations*, u operaciones sobre la estructura que debe realizar el ORM. Los archivos son como un registro de las operaciones que se han ido realizando sobre la estructura de la base de datos y suelen tener la fecha como prefijo, ya que se puede indicar al ORM que realice nuevas o deshaga hasta una concreta. Por ejemplo: creación o eliminación de tablas, añadir o eliminar nuevas columnas…
- *Seeders*, o dicho de otra manera, cargas de datos. Bien sean datos de ejemplo para el desarrollo o unos datos iniciales reales para poner a punto producción.

Los dos últimos puntos se utilizarán en contadas ocasiones, sin embargo los modelos serán consumidos por los repositorios correspondientes para leer o escribir en la base de datos.

4.3.2 Estructura de la base de datos

Como se comentó el apartado Modelo de dominio del capítulo de Diseño, el diagrama diseñado servía para representar la relación existente entre conceptos de modelo de dominio, sin importar su representación en un almacén de datos, como una BBDD. Ésta se contempla según la arquitectura, como un detalle de infraestructura.

Por otro lado, como comenta Vernon, “generalmente, existe una relación uno a uno entre un Agregado y un repositorio.” (Vernon 2013, p.401). Según esta máxima, contemplada en el apartado Repositorios del anterior capítulo, cada Agregado detectado en la aplicación dispondrá de su propio repositorio, es decir, el acceso a los mismos se hará por separado sin importar dónde se persistan. Es por ello que el diseño de la base de datos no era crucial en la anterior fase.

En este punto del desarrollo sí se hace necesario y, dado que se va a usar un ORM, la decisión de usar una misma base de datos es clara.

Examinando los modelos necesarios para el ORM del punto anterior y la base de datos de la que se dispone (Postgres), el diseño de la misma se corresponde con el diagrama de la Figura 7: Diagrama de la base de datos.

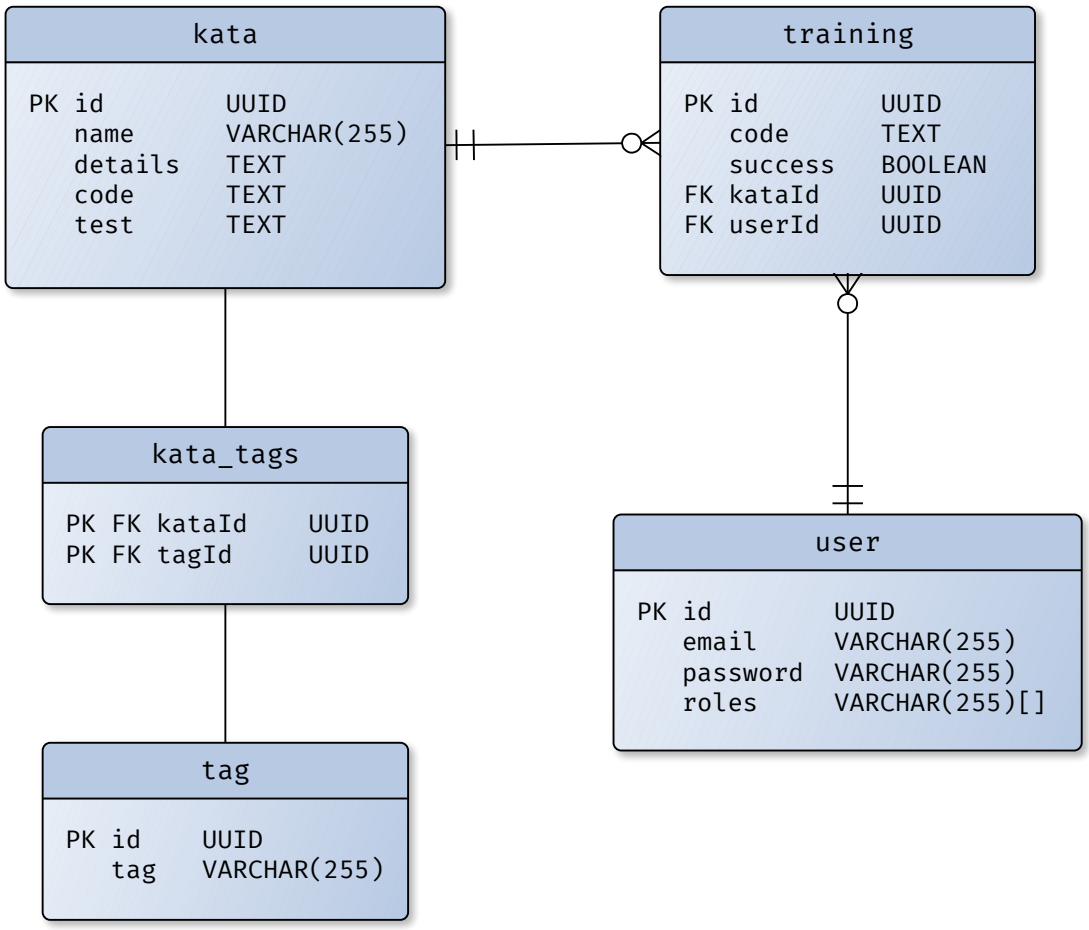

Figura 7: Diagrama de la base de datos

Es primordial indicar que dicho diagrama representa el diseño de base de datos necesario tras las decisiones tomadas en la Segunda replanificación, que puede encontrarse en el Anexo Replanificaciones. Aquéllas hace que el Agregado “Module” desaparezca y, con él, la necesidad de representarlo en base de datos.

4.4 Estructura del código

4.4.1 Entorno de desarrollo

Todos los archivos de configuración del entorno de desarrollo comentados en el apartado Entorno de desarrollo se almacenan en la raíz del proyecto, excepto ciertos `.eslintrc.js` que declaran reglas específicas para algunos directorios.

Se crea una carpeta `__mocks__`, donde se depositarán algunos mocks de ámbito general.

4.4.2 Docker

Los archivos `Dockerfile` y `docker-compose.yml` que se utilizarán para levantar el servidor se incluyen también en la raíz del proyecto.

Por otro lado, se crea un directorio `docker` para asociar algunos de los volúmenes utilizados por los contenedores. De esta forma podemos permitir que los contenedores accedan a ficheros de los mismos sin tener que copiarlos en tiempo de construcción. Se utilizará por ejemplo para el archivo `nginx.conf` o los certificados.

4.4.3 Aplicación

El código fuente de la aplicación se incluye en el directorio `src`. En su interior se encuentran sendos directorios con las capas de dominio, aplicación e infraestructura.

El directorio `domain` sólo contiene los modelos de la aplicación. Mientras, el grueso de la lógica de negocio se encuentra en el directorio `application`: servicios, factorías, transformadores para comunicación de datos con el exterior, excepciones para los distintos errores, creación de una instancia de aplicación con la inyección de sus dependencias...

Dentro de `infrastructure` existen diversos directorios para agrupar código fuente por su tipología. Los principales son:

- `repos`: donde se localizan los repositorios para leer o persistir Agregados en base de datos, la API REST o la cache de cliente. Al ser dependencias de infraestructura, trabajan con DTOs en vez de con objetos del dominio.

- `db`: para recoger todo lo relacionado con el ORM para creación y explotación de la base de datos.
- `ui`: donde se incluyen tanto la API REST como el cliente web. En la arquitectura diseñada, ambos se consideran interfaces de usuario con la única diferencia que utilizan instancias de la aplicación con dependencias diferentes (repositorios, authSession...). En realidad, constituyen aplicaciones independientes que se comunican entre ellas y comparten las mismas reglas de negocio. Gracias a esta arquitectura sería posible, por ejemplo, inyectar al cliente web un repositorio basado en LocalStorage y tener una aplicación local totalmente funcional sin depender de un servidor.

4.4.4 Bundler

Por último, debido al uso de Webpack como *bundler*, se crea un directorio homónimo para albergar las configuraciones del mismo.

Para el proceso de los estáticos de cliente se opta por crear una configuración base sobre la cual se construyen las finales destinadas a desarrollo y producción. La razón para disponer de dos versiones son que producción necesita de optimizaciones que para desarrollo consumirían demasiado tiempo en despliegue continuo. Además el código para el *hot reloading* de desarrollo es innecesario y contraproducente para producción.

4.5 Sesiones de usuario

Para la autenticación de usuarios se utilizan tokens de basados en el estándar abierto JWT[25]. Éstos se generan en el servidor mediante una clave privada y se envían al cliente, quién tendrá que enviarlo en las cabeceras de todas aquellas peticiones que requieran autenticación. El token se decodificará de nuevo en el servidor con la clave que sólo éste posee para comprobar si el token es válido o si ha expirado.

Los token además incluyen información en su interior. De esta forma el servidor puede comprobar si un usuario posee los roles necesarios para una determinada petición.

25 *JSON Web Token* <https://jwt.io/>

4.6 API REST

La API toma como base los principios de una arquitectura REST[26]:

- Arquitectura cliente-servidor: separando completamente interfaz de usuario del almacenamiento de datos.
- Ausencia de estado: las peticiones son atómicas e inteligibles por el servidor sin depender de ningún contexto o sesión.
- Cacheable.
- Interfaz uniforme: mediante una serie de restricciones para interactuar con los recursos.
- Sistema por capas: de tal forma que un cliente no pueda saber si interactúa con el servidor final o con un intermediario.

El API REST de la aplicación utilizará la URI como identificador de un recurso y métodos HTTP) como representación de la operación que se desea realizar sobre un recurso (GET, POST, PUT, PATCH, DELETE). El éxito o error de aquélla se representa con el código de estado HTTP de la respuesta.

Los códigos de estado tienen a su vez asociado una de las excepciones definidas en la aplicación. De esta forma es posible generar un código de estado en la API tras una respuesta de error de la aplicación *back end*. Y a su vez, la aplicación de cliente podrá traducir dicho código de estado a la excepción. Gracias a este proceso, es posible dentro de la arquitectura diseñada enviar una misma excepción desde servidor a cliente.

HTTP	Éxito	HTTP status	Excepción
GET	Sí	200	-
POST	Sí	201	-
PUT	Sí	200	-
PATCH	Sí	200	-
DELETE	Sí	204	-
*	No	400	`BadRequestError`
*	No	401	`UnauthorizedError`

26 *Representational state transfer*

*	No	403	`ForbiddenError`
*	No	404	`NotFoundError`
*	No	409	`ConflictError`
*	No	500	`Error`

Tabla 6: API REST: Métodos HTTP, códigos y errores equivalentes

4.7 Optimización del rendimiento web

4.7.1 Compresión

Para acelerar la carga de una web es comprimir los archivos de texto (html, js, css…). En este proyecto se habilita en Nginx la compresión al vuelo con Gzip. Además se otra versión con Brotli[27] (una alternativa moderna que mejora en un 20-30% la tasa de compresión de gzip) para los navegadores que lo soporten.

4.7.2 *Code splitting*

Para evitar que un visitante deba bajar todo el código Javascript de una SPA en la primera carga, aunque no lo vaya a necesitar, se aplica una técnica denominada *code splitting*. Mediante la misma el código generado por el bundler se separa en varios archivos que se cargarán bajo demanda.

Para marcar qué código puede separarse se sustituye el `import` habitual:

```
import KataAdmin from '../pages/KataAdmin.vue',
```

Por una importación dinámica:

```
const KataAdmin = () ⇒
  import(
    /* webpackChunkName: "kata-admin" */ '../pages/KataAdmin.vue'
  );
```

27 *https://github.com/google/brotli*

Los beneficios se pueden ver mejor a través de los resultados de un analizador de *bundles*.[28]

Un análisis inicial arroja que el peso total del JS de la aplicación es de 170.87 KB, una vez aplicada la compresión con gzip.

Figura 8: Análisis del bundle JS inicial

Además de ese dato, el análisis permite comprobar la contribución al peso final de los diferentes módulos y dependencias que componen la aplicación.

En este caso concreto puede observarse cómo el editor online (`codemirror.js`), cuya funcionalidad no es necesaria hasta seleccionar una kata, aporta mucho peso inicial no esencial. Es el ejemplo más visible de la necesidad de dividir el código que se envía a cliente.

Tras aplicar *code splitting* y realizar un nuevo análisis, el JS que se carga inicialmente pasa a ser de 10.62 KB (código propio: `app`) y 67.01 KB (dependencias: `vendors~app`). Estos 77.63 KB suponen una reducción de casi un 55% respecto al *bundle* inicial.

28 *webpack-bundle-analizer.* <https://github.com/webpack-contrib/webpack-bundle-analyzer>

Figura 9: Análisis de los bundles tras aplicar code-splitting

4.8 Flujo de trabajo de Git

Antes de cerrar este capítulo es indispensable comentar la metodología seguida en el repositorio durante el ciclo de desarrollo.

Dado que el autor de este TFG es el único programador que interviene en el repositorio, se ha optado por un flujo de trabajo ramificado basado en ramas puntuales o por funcionalidad.

Podría haberse utilizado un flujo centralizado, en el que los *commits* se mandasen siempre a una rama *master*. Sin embargo, el trabajar por funcionalidades permite leer mejor la historia del repositorio y descartar o deshacer cambios.

La rama de la que han partido y a la que se han ido integrando todas las ramas ha sido `develop`. Posteriormente con la publicación a producción, la rama central ha sido denominada `master`.

De esta manera, es posible diferenciar cuando un producto esta aún en su desarrollo inicial o posee ya una versión estable en uso. Es un concepto extraído del *GitHub flow*[29], que toma como norma que todo lo que haya en la rama `master` es desplegable en cualquier momento.

El flujo definido por GitHub indica también que el nombre de las ramas

29 https://guides.github.com/introduction/flow/

debe ser descriptivo. Sin embargo, no establece ninguna convención al respecto.

Para este proyecto, la nomenclatura de ramas está influida por Gitflow, un flujo creado por Vincent Driessen en 2010[30]. Aunque hasta el mismo autor ha dejado de recomendar Gitflow, las convenciones de prefijos en ramas para publicación o hotfix[31] siguen siendo muy interesantes. Así se han definido los siguientes prefijos:

- `feature/` para ramas de funcionalidad.
- `hotfix/` para parches rápidos para un despliegue inmediato
- `bugfix/` para parches cuyo objetivo no sea crítico
- `doc/` para ramas que documentación

Se ha establecido también la convención de que los nombres de las ramas irán en minúscula y las palabras serán separadas por un guión. Debido a esto, los prefijos anteriores utilizan una barra (`/`) como separador, para así poderlos distinguir claramente en el nombre.

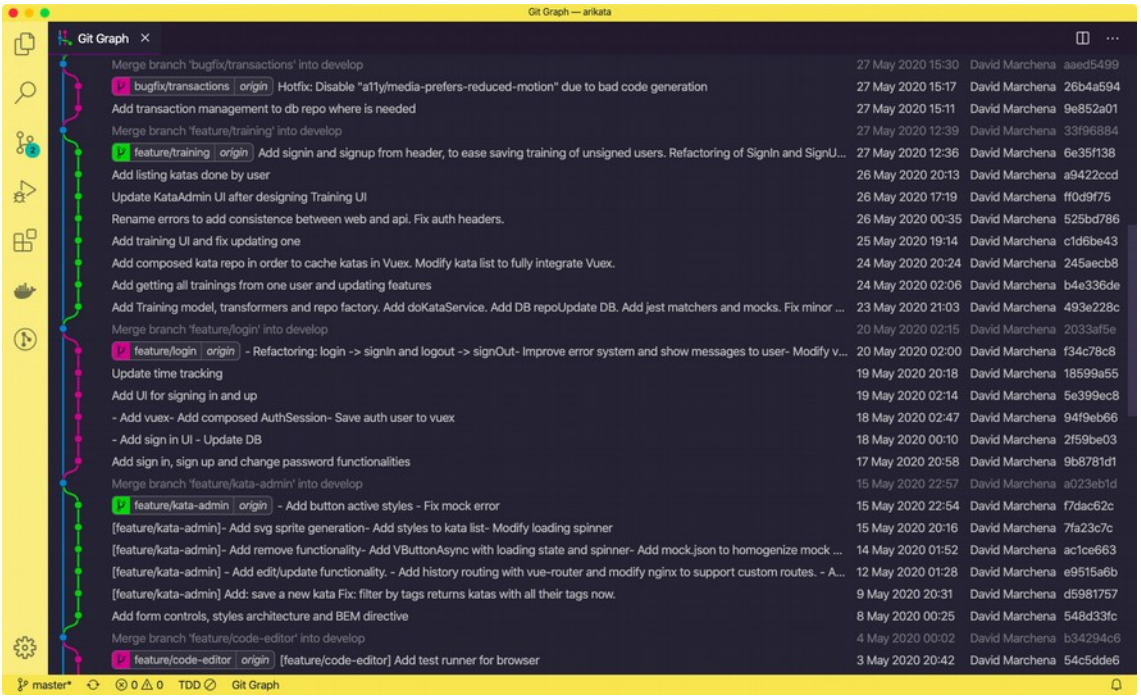

Figura 10: Fragmento del árbol de Git del proyecto en VSCodium

30 https://nvie.com/posts/a-successful-git-branching-model/

31 *Parche rápido para aplicar a un entorno de producción con objeto de solucionar un error detectado en el mismo.*

4.8.1 GitHub Actions e integración continua

Dado que el repositorio remoto contra el que se trabaja está alojado en GitHub, se ha aprovechado una característica recientemente añadida: las acciones. Mediante éstas es posible definir flujos de trabajo en un repositorio de GitHub, de tal forma que se ejecuten una serie de tareas cada vez que se suban los cambios de una rama.

Así, se ha podido disfrutar de integración continua en este proyecto.

Se ha configurado el flujo para que se lancen la tarea de *linting* y los tests para las diferentes versiones de Node.js (10.x, 12.x y 14.x). De esta forma, si se detectaba algún fallo, el autor recibía un correo indicando que era necesario revisar el código.

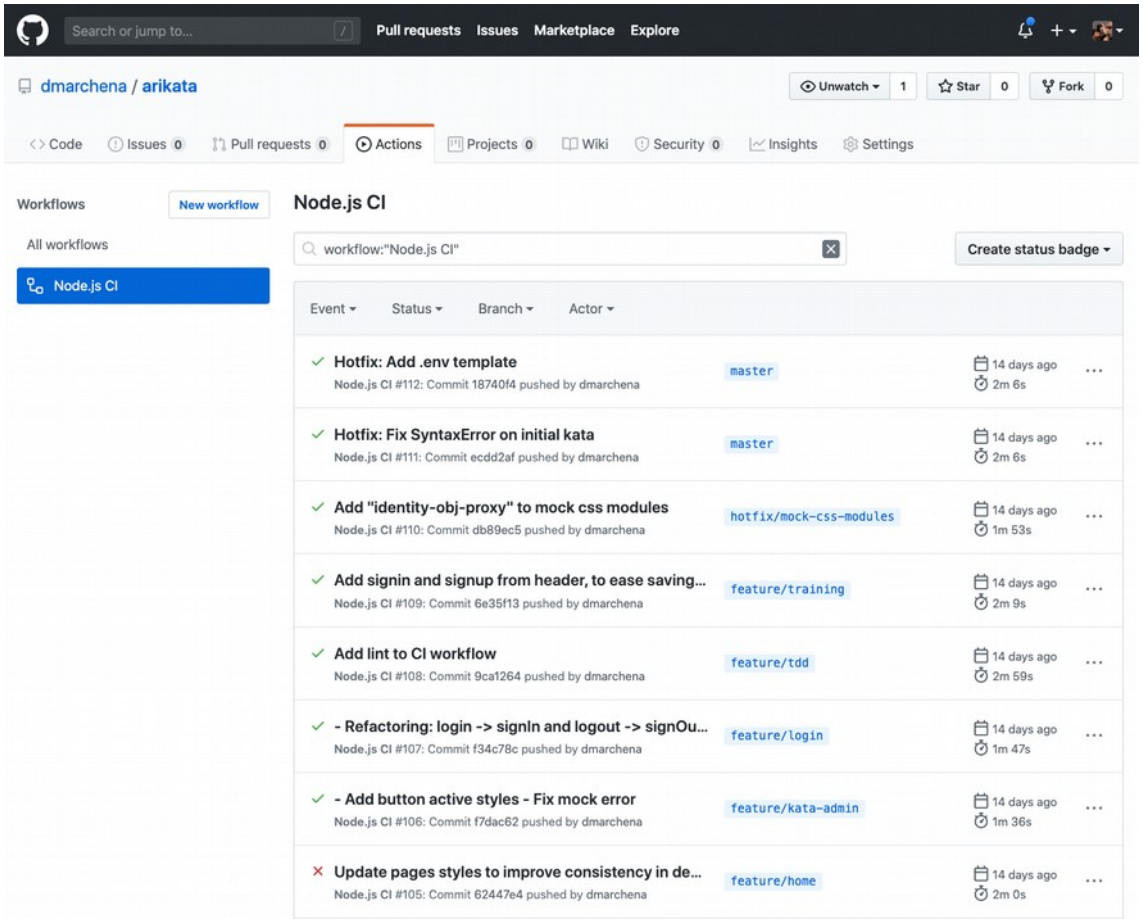

Figura 11: GitHub actions

Capítulo 5

Resultados

5.1 Producto obtenido

5.1.1 Código fuente

El código fuente de la aplicación ha sido liberado bajo licencia MIT[32] en GitHub. Se encuentra disponible en la siguiente url:

https://github.com/dmarchena/arikata

5.1.2 Puesta en marcha

La aplicación necesita seguir los siguientes pasos para levantar el servidor y poder acceder a la misma:

- Clonar el repositorio de GitHub.
- La aplicación funciona con HTTP/2. Por dicho motivo (junto con el requisito RF10) es necesaria[33] la generación de un certificado, almacenar dicho certificado junto con la clave privada en el directorio `/docker/ssl/` y confiar en la autoridad certificadora. La generación se puede realizar de diferentes maneras:
  - Siguiendo los pasos descritos en Creación de un certificado SSL auto-firmado
  - Mediante el script que puede encontrarse en el directorio `/scripts/ssl-self-signed/`.

32 *https://choosealicense.com/licenses/mit/*

33 *«Aunque el estándar HTTP/2 no requiere explicitamente el uso de encriptación, los principales navegadores tomaron la decisión de soportarlo sólo sobre protocolo TLS.»* («An introduction to HTTP/2 - SSL.com» [sin fecha])

- Crear un archivo `.env` a partir de la plantilla `.env.template` para configurar los datos del servidor.
- Levantar el servidor mediante el comando:

```
docker-compose up
```

- Instalar Node.js (LTS o stable) y yarn en la máquina.
- Crear la estructura de la base de datos y alimentarla con unos datos iniciales con los comandos:

```
yarn db:migrate

yarn db:seed
```

- Acceder a la url dominio que se haya especificado en el certificado e incluido en el archivo de hosts.

5.1.3 Servidor de producción

Con objeto de hacer más sencillo probar el producto final y realizar prueba real de cumplimiento del requisito RNF7 la aplicación se encuentra desplegada en un servidor de producción alojado en DigitalOcean.

Como mínimo, durante el tiempo que dure este TFG, será posible acceder a la aplicación a través del siguiente dominio:

https://arikata.dev/

Para poder ver la zona de administración del sitio web se ha habilitado un usuario con permisos de administrador:

```
Email:      uoc@arikata.dev
Password:   oH%sSmIKCX?U&1d.
```

5.1.4 Pantallas

Para poder observar el producto resultante sin arrancar la aplicación o acceder a producción se han realizado una serie de capturas de pantalla.

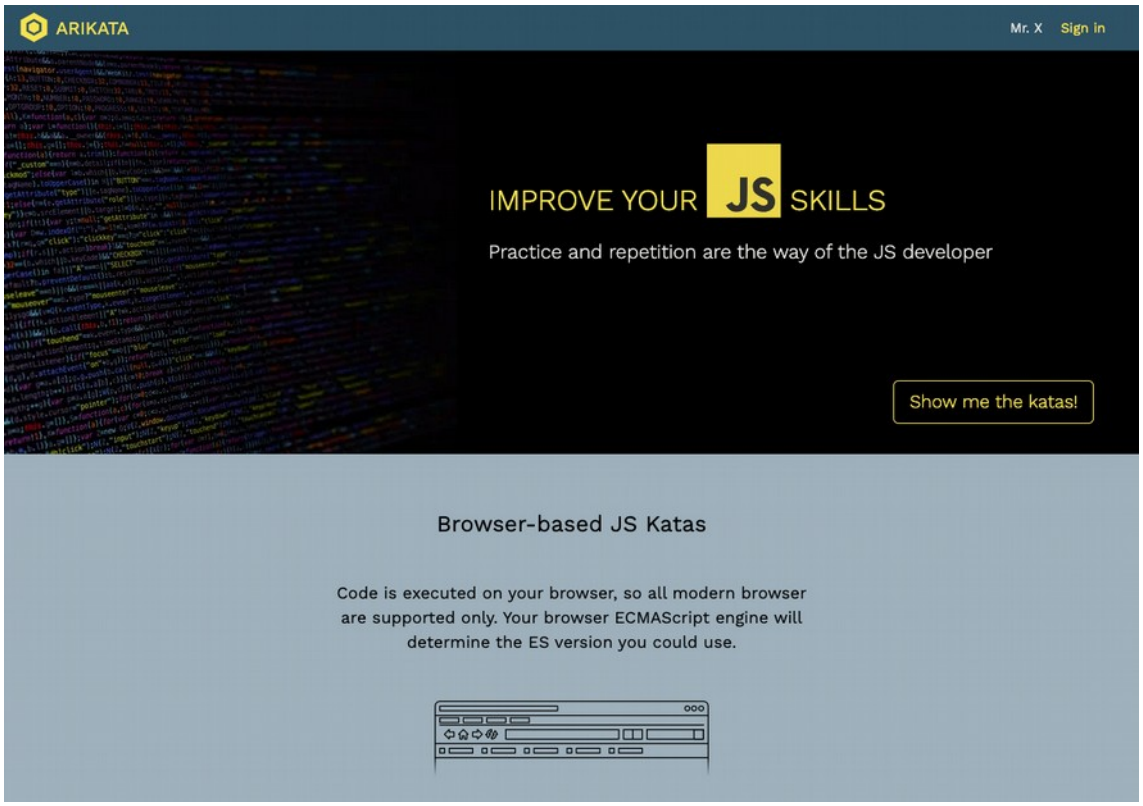

Figura 12: Captura de la página de inicio

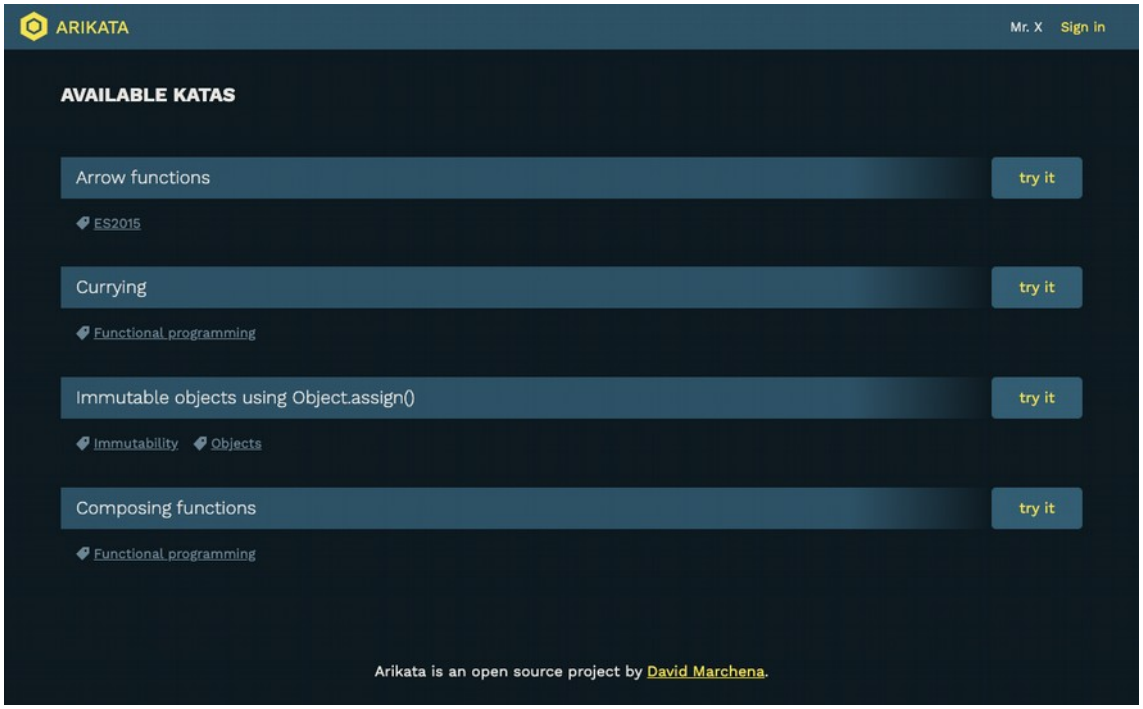

Figura 13: Lista de katas disponibles

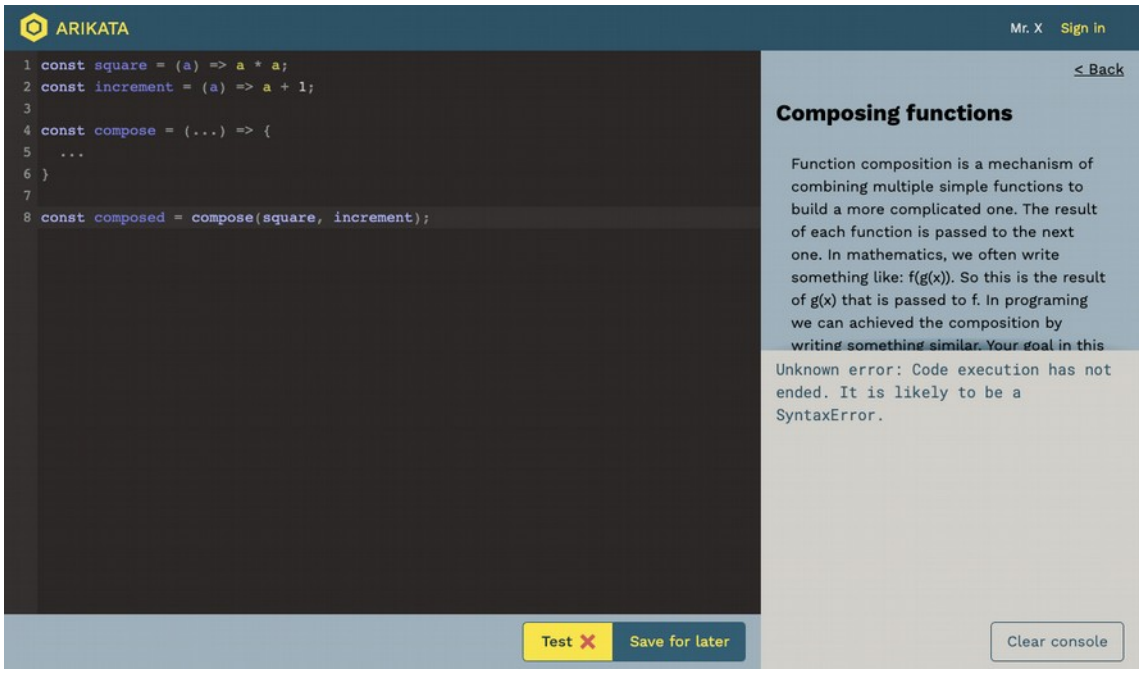

Figura 14: Realización de una kata: test fallido

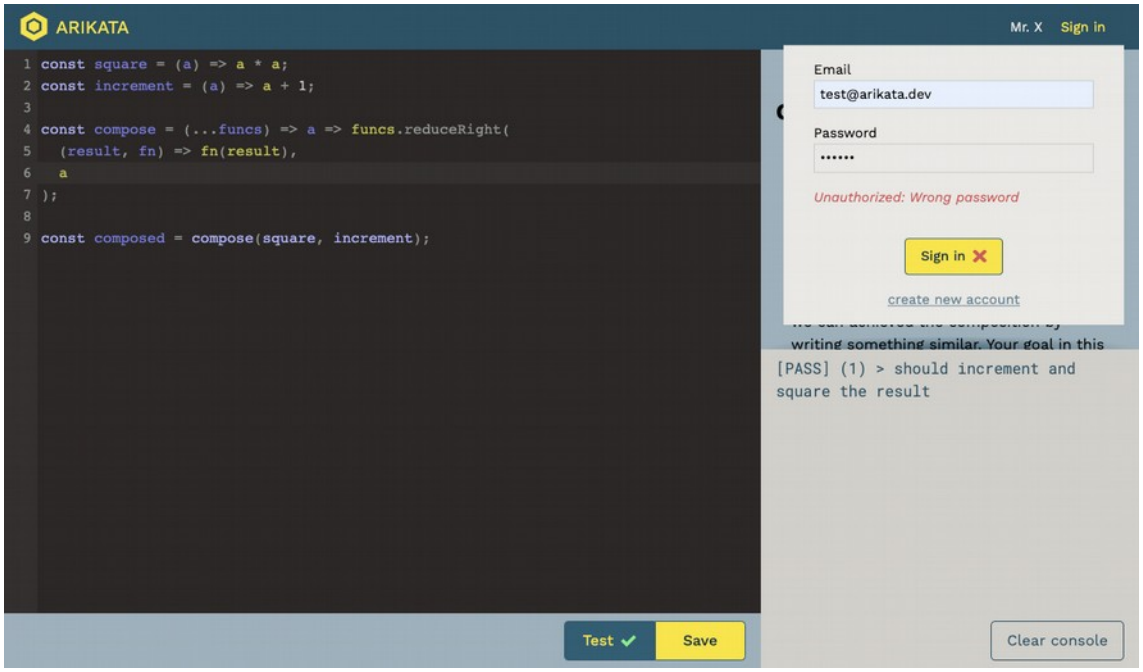

Figura 15: Realización de una kata: test correcto e intento de autenticación

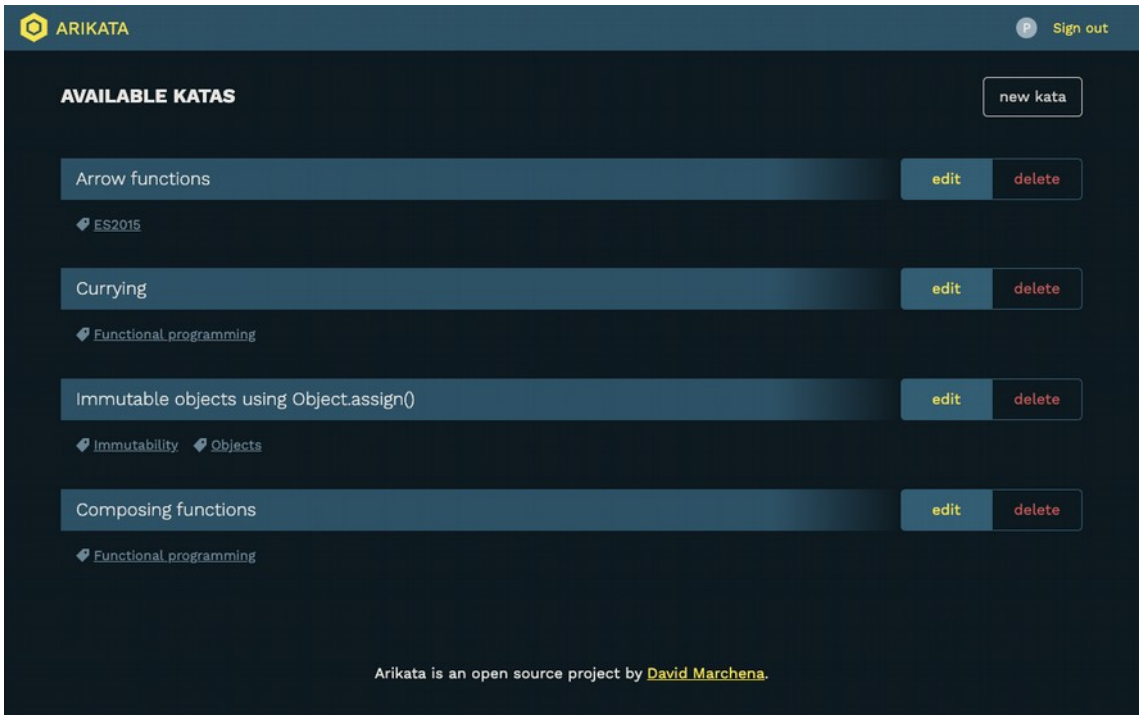

Figura 17: Administración: Lista de katas

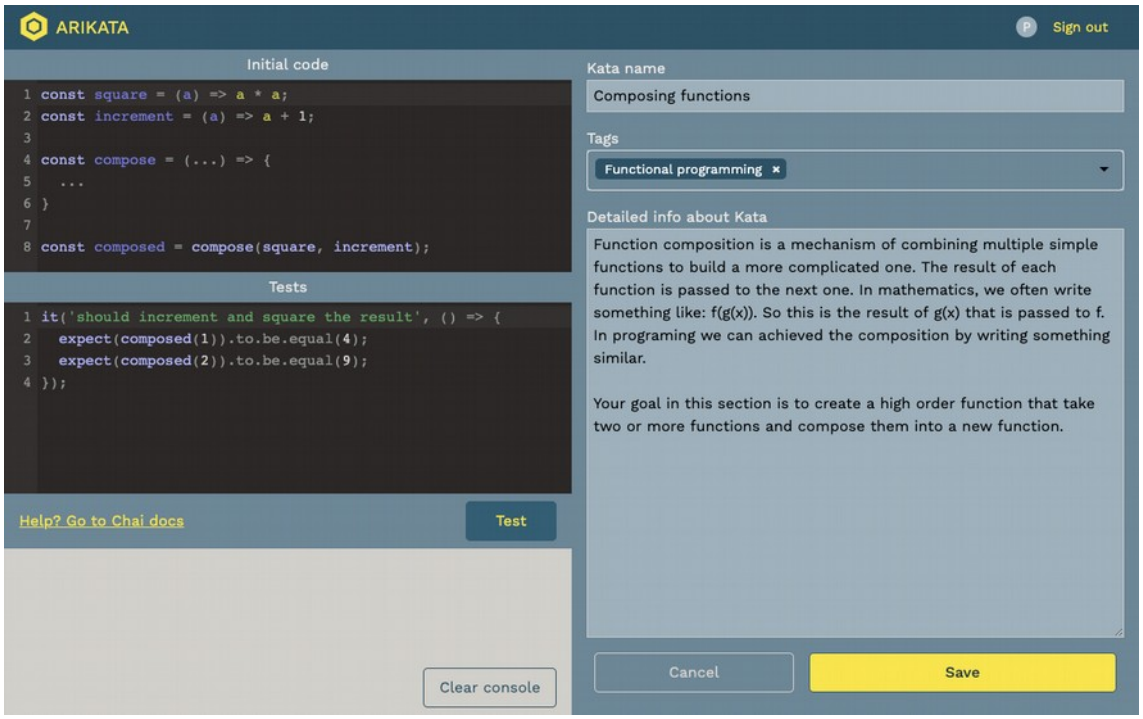

Figura 18: Administración: gestión de una kata

5.2 Tests y cobertura

Durante el desarrollo se ha seguido una metodología TDD. Producto de la misma se han obtenido un total de 55 archivos de test y 224 pruebas.

Con el siguiente comando se lanzaran los tests y el análisis de cobertura:

```
yarn coverage
```

Como puede verse en la Figura 19: Resultados de tests con cobertura, todos los tests son positivos y la cobertura ronda el 62-66%.

Si bien durante el desarrollo la cobertura ha oscilado entorno al 75-80%, cercano a lo marcado en el apartado de Subobjetivos, ha acabado decreciendo al final por las siguientes causas:

- El código fuente relacionado con la interfaz de usuario es más costoso y difícil de testear. En ocasiones eran necesarios mocks demasiado complejos y, en otras, era imposible realizar el test sin un navegador (como el caso de la ejecución de código en un iframe aislado).
- A medida que el hito de entrega se acercaba, inconscientemente se intercalaba TDD puro con un desarrollo más tradicional. Este punto es muy interesante y se abordará en las conclusiones del proyecto.

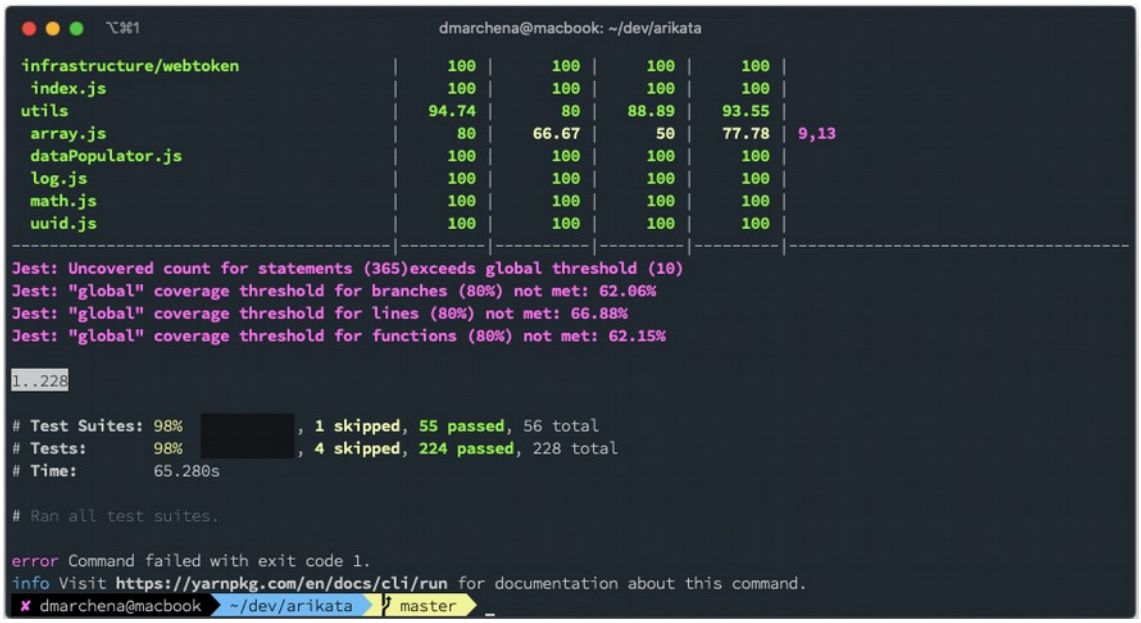

Figura 19: Resultados de tests con cobertura

5.3 Auditoría de rendimiento web

Entre los requisitos no funcionales planteados se encontraba el que la web fuese considerada como “rápida” por Lighthouse (RNF3). Para ello es necesario obtener una puntuación de entre 90 y 100 en la auditoría.

Para tener datos más completos, se han realizado auditorías para la pagina inicial y para el listado de katas en producción. Aunque ambas pantallas implican la ejecución de la aplicación, el listado es más interesante al incluir comunicación con servidor y la base de datos.

Como puede observarse los resultados son más que satisfactorios.

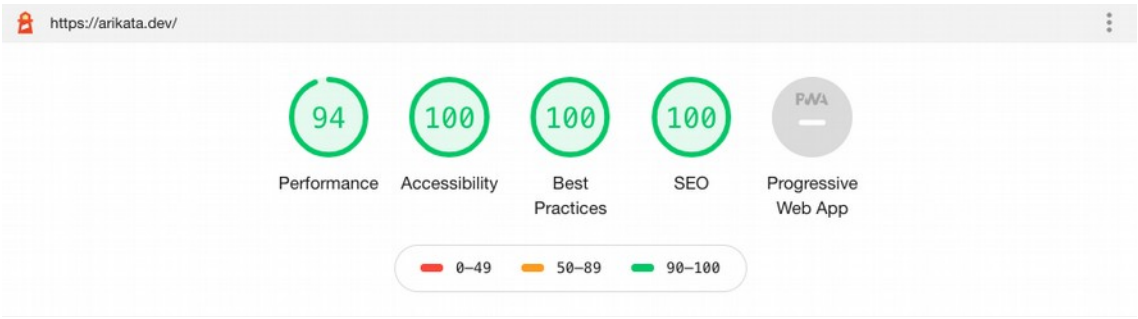

Figura 20: Resultados de Lighthouse para la página inicial

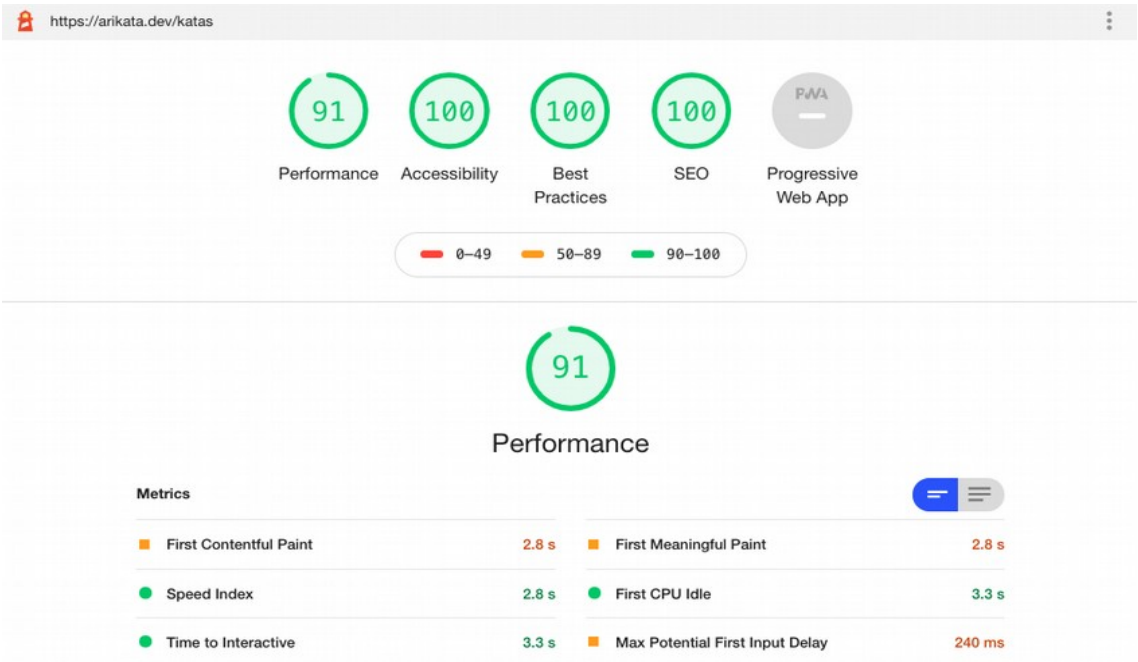

Figura 21: Resultados de Lighthouse para el listado de katas

Capítulo 6

Conclusiones

6.1 Evaluación de los objetivos

Al comienzo de esta memoria se establecieron una serie de objetivos. El principal era el de obtener una herramienta que permitiese a un desarrollador reafirmar conocimientos u obtener nuevos de una forma práctica y sencilla.

A pesar de carencias debidas a las necesarias replanificaciones o errores, tras revisar los resultados obtenidos y las opiniones de compañeros de trabajo al probarla es muy probable que la aplicación sea capaz de cumplir con el objetivo.

Además en el apartado “Subobjetivos”, se establecieron unos objetivos secundarios. De ellos, valoro positivamente los siguientes:

- El entorno de desarrollo diseñado es cómodo de utilizar. Para instalarlo en un nuevo puesto sólo es necesario disponer de Node.js (con yarn) y clonar el repositorio. No es necesario instalar manualmente más herramientas o realizar configuraciones en el IDE.
- La aplicación web tiene una UX bastante cercana a la deseable en inicio. La navegación es ágil y las peticiones asíncronas dan una respuesta coherente al usuario por medio de un icono en el botón pulsado y con un mensaje de error si es necesario. Por otro lado, como se ha podido ver en el capítulo anterior, el rendimiento es muy satisfactorio.
- El editor online es bastante cómodo de utilizar, principalmente gracias a un proyecto como CodeMirror[34]. La navegación por teclado es funcional, salvo el hecho de que cuando el foco se encuentra en el editor se anula la posibilidad de navegar con el tabulador. Para mitigar este hecho se ha habilitado lanzar los test mediante una combinación

34 *https://codemirror.net/*

de teclas (Ctrl + Enter).

- El *back end* en Docker ha permitido desarrollar en la máquina local sobre una infraestructura portable y sin dependencia con la máquina. El despliegue del mismo en el servidor de producción fue una tarea sencilla de un par horas. La mayor parte del tiempo fue dedicada a leer la documentación del proveedor para encontrar una forma de desplegar utilizando un `docker-compose.yml`.

Los subobjetivos restantes es necesario analizarlos en más profundidad.

6.1.1 TDD y cobertura

Seguir un desarrollo dirigido por tests implica un tiempo extra que ha tenido su impacto negativo en la planificación. La experiencia ha sido positiva y, sin duda, volvería a ser un objetivo si el proyecto se iniciase de nuevo. Sin embargo, la planificación debería reflejar ese coste extra y justificarlo.

En resumen, de entre los muchos beneficios observados, hay que destacar los siguientes:

- Mejor definición de interfaces y datos de respuesta esperados.
- Descubrimiento de nuevas necesidades o definiciones incorrectas en la fase de diseño (por ejemplo: en firmas de algunos métodos de servicios de aplicación).
- Favorecer las refactorizaciones en pos de optimizaciones o mayor claridad de código. Una buena cobertura de test hace que se modifique sin miedo código ya funcional. Constituye un remedio muy eficaz frente a códigos que, debido a su complejidad, se han convertido en cajas negras y el pensamiento habitual de “si funciona, no lo toques”.
- Detección rápida de bugs causados por efectos secundarios de nuevos desarrollos.

En el capítulo anterior, en el punto “Tests y cobertura”, ya se adelantó cómo la cobertura ha ido bajando hasta el punto de no alcanzar el 80% esperado.

Como comentaba Fowler: “si estás probando bien y reflexionando, se debería esperar una cobertura de 80 o 90” (Fowler 2012). Esto puede indicar que no se haya seguido del todo bien la metodología.

Por otro lado, Fowler también incide en el hecho de que el principal objetivo de un análisis de cobertura es el de ayudar a determinar qué

partes del código no están siendo testadas. Según él, se podría decir que se están haciendo suficientes tests si se cumple lo siguiente:

- Raramente de despliegan errores a producción, y
- No se duda en cambiar código sólo por miedo a producir errores en producción.

En resumen, para marcar el objetivo como cumplido parece que sería necesario un ciclo de vida más largo (más allá de la primera versión en producción), ya que hay aspectos que indican que el objetivo se cumple y otros que recalcan lo contrario.

En otro orden de cosas, hay que destacar que a veces se han obviado tests porque el coste de generarlos era muy superior al del código a cubrir y el hito apremiaba. De ahí que, al repasar el ciclo de desarrollo, surja a la siguiente reflexión:

Ante un hito y la necesidad de acelerar el desarrollo, ¿qué lleva a un programador a inconscientemente ahorrar tiempo eliminando todos los beneficios que aportan los tests?

Es difícil encontrar una respuesta clara. Habrá una parte que sea debida a falta de experiencia en *testing*; otra, a una mala calidad en el código o en la encapsulación del mismo que lo haga difícil de probar de forma unitaria; y una última, más psicológica, relacionada con la obtención rápida de algo funcional que dé una sensación de avance (aunque sin pruebas de cumplir la especificación y pueda derivar en un mayor coste por solución de errores). También puede ser que, una vez en el mercado laboral, el programador interiorice la creencia de muchos gestores de que las pruebas automáticas son un sobrecoste.

Es sin duda una pregunta compleja que invita a la reflexión y su mera aparición al concluir el proyecto hace que haya merecido la pena elegir esta metodología.

Para finalizar, un último apunte. En base a la experiencia obtenida, TDD es una metodología muy recomendable para cualquier proyecto, sobre todo si la calidad y mantenibilidad es un requisito. Quizá en ciertos proyectos de consultoría (con un ciclo de vida corto o con una fase de mantenimiento por equipos con otras metodologías) no tendría mucho sentido o podría aplicarse solo en zonas críticas. Sin embargo en ámbitos de producto o proyectos con una larga vida sus beneficios superan con creces al coste, pudiendo considerarse como casi indispensable.

6.1.2 Domain-Driven Design

El uso de una arquitectura limpia basada en DDD era una apuesta personal con escasa relevancia al principio del proyecto. A priori se esperaba aplicar ciertos principios, como el de abstraer la aplicación de detalles de infraestructura.

A medida que maduraba la fase diseño y la de desarrollo, fue cogiendo importancia, movido por el interés personal y la oportunidad de disponer de un lienzo en blanco para la aplicación de un marco teórico tan interesante como extenso y complejo.

El resultado final, a pesar de sus carencias, es satisfactorio.

Por un lado, se han asimilado mejor conceptos que en su estudio pueden resultar demasiado abstractos y se ha ganado una visión más completa de lo que supone DDD; en lugar de repetir sin criterio todas y cada una de las técnicas propuestas, lo que acaba llevando a la sobreingeniería.

Por otro, el producto se aproxima a una arquitectura hexagonal de puertos y adaptadores, donde los puertos son puntos en los que el *core* de la aplicación puede interactuar con el exterior. Éstos suelen dividirse en dos grupos:

- Aquellos que exponen funcionalidades. Por ejemplo, los servicios de aplicación con los que interactúan tanto cliente web como el API REST.
- Contratos que definen la visión del *core* de la aplicación del mundo exterior. Es el caso de las factorías de repositorios, que definen la interfaz para crear adaptadores para interactuar con la base de datos o el API REST.

Cierto es que el producto final tiene sus fallos pero permite comprobar en la práctica algunos de los beneficios que aportan este tipo de arquitecturas:

- Al compartir toda la lógica de negocio, el cliente web y la API REST se benefician de cualquier *bug* solucionado en aquélla.
- Migrar la aplicación de un modelo cliente-servidor a otro es mucho más sencillo. Basta inyectar otros adaptadores de repositorios para crear un sitio web estático que utilice Local Storage o una aplicación de escritorio basada en Electron[35].

A pesar de ello, es una arquitectura recomendable para proyectos de más envergadura. Para este caso concreto, es claro que ha añadido

35 https://www.electronjs.org/

complejidad, mayor carga de ejecución de código (en múltiples transformaciones de DTOs por ejemplo). Si el objetivo no fuese académico, hubiese sido un claro error de sobreingeniería.

Entre los fallos destacables están el de haber obtenido un modelo bastante anémico, a pesar del hincapié realizado en la fase de diseño. En parte por no haber utilizado eventos de dominio (para ahorrar complejidad) y por falta de experiencia. Ejemplo de esto último es no haber conocido hasta casi el final que otra de las responsabilidades de los objetos de dominio es el de hacer de factorías para entidades dependientes de un agregado u otros relacionados.

Si el proyecto comenzase de nuevo seguramente se trabajase de otra manera con la inyección de dependencias. En vez de encapsularlas dentro de cada servicio a través de una factoría podría ser centralizarlas en la instancia de aplicación generada y exponerlas si fuese necesario. De esta forma, en vez de generar un `AuthSession` sin funcionalidad para el API REST y controlar los permisos desde el router de Express, éste podría haber interactuado con el `AuthSession` y haber integrado mejor el JSON Web Token en la aplicación.

6.2 Evaluación de los requisitos de la aplicación

6.2.1 Requisitos funcionales

Ciertos requisitos han sido descartados tras ajustes del alcance del proyecto en sendas replanificaciones. Se ha procurado que aquellos fuesen los que menos valor restasen al proyecto por lo que no deben tener mucho peso en esta evaluación.

Sin embargo, al repasar el TFG se han encontrado algunos que no se cumplen debido a errores durante el diseño o el desarrollo. Es el caso de:

- RF3: El filtrado por nombre de Kata desapareció en la fase de diseño al definir las firmas de los métodos de los servicios de aplicación en el punto “Application services”. Ese error hizo que el requisito pasase inadvertido durante el desarrollo hasta que fue demasiado tarde.
- RF7: A pesar de haberlo tenido en cuenta en fase de diseño, el guardar una propuesta de solución y que el usuario pudiese consultarla desapareció durante el desarrollo. Se cree que pudo deberse a intentar acelerar el desarrollo, creyendo que no se trataba

de un requisito sino de una funcionalidad añadida para mejorar la aplicación.

- RF19, RF20: Los requisitos de hacer visible o archivar una kata se omiten en los casos de uso y desaparecen en el diseño como ocurrió con RF3.
- RF22: Al igual que en el caso del RF7, durante el desarrollo no se tuvo en cuenta que el uso de markdown era un requisito explícito.

De los comentados, el más grave es el RF7, ya que supone una clara diferencia en la experiencia de uso para el usuario. A pesar de estar muy satisfecho con el producto obtenido y que el resto de requisitos no cumplidos no afectan demasiado al mismo, el RF7 hace dudar en si considerar como cumplidos los objetivos marcados. Si el proyecto sigue, sin duda será una de las siguientes tareas a abordar.

Si el proyecto volviese a iniciarse, es indudable que la lista de requisitos debería tener un lugar mucho más visible a lo largo del TFG, en vez de quedarse perdida en la documentación. Seguramente hubiese bastado un gesto tan sencillo como haberla impreso en una hoja de papel.

6.2.2 Requisitos no funcionales

Por como se ha visto en el capítulo anterior, estos requisitos han sido cumplidos y con un alto grado de satisfacción. Incluso la tolerancia a fallos de red (RNF4), que había sido descartada en el apartado “Requisitos descartados”, es bastante alta al tratarse de una SPA y haber introducido un almacén de estado con Vuex que hace las veces de cache de katas.

6.2.3 Requisitos de procesos

Se ha hablado largo y tendido anteriormente de los RP1, RP2 y RP7, referidos al uso de TDD y DDD.

Respecto a los demás que seguían vigentes, se han cumplido sin problema.

6.3 Planificación

Durante el desarrollo se detectaron errores que llevaron a la Segunda replanificación.

A final del proyecto, la planificación no se puede considerar un fracaso, pero sobre todo gracias a replanificaciones, sacrificar tareas de desarrollo y ampliar horas de dedicación.

Tras hacer una retrospectiva se han detectado algunas medidas que hubiesen sido de gran ayuda a la hora de planificar mejor y llevar a cabo un seguimiento:

- Revisar la planificación que se realizó al comienzo del proyecto tras la fase de análisis. Sin la captura de requisitos, era muy probable que las tareas de desarrollo no estuviesen correctamente definidas.
- Orientar las tareas de desarrollo a funcionalidades concretas, en vez de disponer de tareas globo que agrupaban todo lo referente a una capa (Ej: parte pública).
- Dividir la fase de desarrollo en *sprints* de dos semanas. Aunque se trate de un proyecto en cascada, esta figura del desarrollo ágil hubiese obligado a realizar un seguimiento de la planificación cada dos semanas y ayudado a detectar posibles desviaciones antes de ser demasiado graves.

Glosario

API

Acrónimo en inglés que significa *application programming interface*. Una API es una capa de abstracción que simplifica la comunicación entre componentes de software y define las peticiones que pueden realizarse, cómo hacerlas, el formato de los datos y otras convenciones adoptadas.

BUNDLER

Herramienta que permite combinar múltiples archivos en uno (o varios, si no es necesario cargar todo al inicio de la página).

CLEAN ARCHITECTURE

Es un término que acuñó Robert C. Martin (también conocido como Uncle Bob y por ser el autor de "Clean Code: A Handbook of Agile Software Craftsmanship") para intentar arquitecturas de software basadas en capas de dominio, negocio e infraestructura como: *Hexagonal Architecture* (también llamada *Ports and Adapters*) de Alistair Cockburn o *Onion Architecture* by Jeffrey Palermo. Como bien explica, su propósito es conseguir un software testable e independiente de cualquier agente externo ajeno al dominio y a la lógica de negocio (interfaz de usuario, frameworks, base de datos…) (Martin 2012)

CRUD

Acrónimo inglés para referirse a las operaciones típicas de persistencia: Create, Read, Update y Delete.

DDD

Del inglés *Domain-Driven Design*, es término introducido por Eric Evans, para denominar a un enfoque en el diseño de un software que pone el foco en como plasmar el esquema mental y el lenguaje utilizado sobre un determinado modelo de negocio. Así, mediante una mejor comunicación y colaboración entre los expertos de un dominio y los técnicos que implementan el software, el

producto resultante está más alineado con los objetivos y puede evolucionar con los mismos.

Docker container Es una instancia en ejecución de una imagen de Docker.

Docker image Es un paquete de software sin estado generado en base a la configuración y comandos especificados en un Dockerfile.

DTO Del acrónimo inglés *Data Transfer Object*. Es un objeto que permite la transferencia de datos entre procesos. Al tratarse de objetos que pueden viajar por la red, deberán de poder ser serializables. Un DTO no tiene por que ser un reflejo de una entidad del modelo. A menudo un DTO puede agrupar los datos de diferentes entidades que son necesarios para, por ejemplo, una vista.

FIRMA La firma de un método indica la información que recibe (parámetros) y la que se devuelve a quién haya solicitado la ejecución del método.

FRAMEWORK Anglicismo para referirse a una plataforma o abstracción que provee de conceptos y funcionalidad genérica para resolver una problemática definida (sin importar si está acotada un aspecto del software o es más generalista). Puede extenderse a través de código del desarrollador pero su código no puede ser modificado.

KATA Es un ejercicio de código que permite a un programador mejorar sus habilidades. El término fue introducido por Dave Thomas, coautor del libro “The Pragmatic Programmer”. («Kata (programming)» 2019)

MIXIN Anglicismo para referirse a una clase que ofrece una determinada funcionalidad que puede ser usada por otras clases sin tener que utilizar herencia y por tanto seguir una jerarquía. Favorece la reutilización de código y la

implementación de software a través de la composición.

MODELO ANÉMICO

Un modelo anémico es aquel en el que las entidades carecen de comportamiento y son simples agrupaciones de datos que, a menudo, mapean una tabla de base de datos. Para autores como Evans o Fowler, es un antipatrón que va en contra de la idea del diseño orientado a objetos, que combina datos y procesamiento juntos (Fowler 2003). Un modelo anémico a su vez puede promover el acoplamiento entre capas si se utilizan las entidades del modelo como DTO. Además, en base a la experiencia laboral del autor de este TFG, puede ocasionar que se distribuyan reglas de negocio por controladores, DAOs o incluso interfaz (dependiendo de la experiencia del desarrollador y de su conocimiento de las reglas). Esto derivaría en un software menos robusto en el cual sería más difícil modificar el flujo de un caso de uso o incluso solucionar errores que impidan el éxito del mismo.

ORM

Del acrónimo inglés Object-relational mapping. Se trata de un modelo de programación que permite mapear la estructura de una base de datos relacional en una estructura que sigue un paradigma de orientación a objetos. A través de la misma será posible interactuar con una base de datos utilizando el lenguaje de programación escogido y los objetos del ORM en vez de consultas SQL.

POLYFILL

Anglicismo para referirse al código que implementa una determinada funcionalidad en aquellos navegadores que no la soporten de manera nativa.

PRINCIPIO DE INVERSIÓN DE DEPENDENCIAS

Es uno de los principios S.O.L.I.D. postulados por Robert C. Martin. El principal propósito de este principio es el de desacoplar módulos de software y establece que un módulo de alto nivel no debería depender de uno de bajo nivel, sino de una abstracción.

SPA

Siglas en inglés de *Single-page application*. Se

refiere a una aplicación web que evita el flujo por defecto de un navegador de cargar nuevas páginas por URL. En una SPA, todo o gran parte del HTML, JS y CSS se descargan en la primera carga y la interacción con servidor se reduce drásticamente. De esta forma la interacción se asemeja más a una aplicación nativa y aumenta la sensación de fluidez por parte del usuario.

TRANSPILADOR

Un transpilador de código es un tipo especial de compilador que traduce un código fuente a otro de un nivel similar de abstracción. («Transpilador» 2019)

Bibliografía

Best Practices for Designing a Pragmatic RESTful API. *Vinay Sahni* [en línea], [sin fecha]. [Consulta: 16 mayo 2020]. Disponible en: https://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api.

BONEFESTE, A., 2019. Postgresql vs. MongoDB for Storing JSON Database. *Sisense* [en línea]. [Consulta: 9 abril 2020]. Disponible en: https://www.sisense.com/blog/postgres-vs-mongodb-for-storing-json-data/.

ELLIOTT, E., 2014. *Programming JavaScript applications*. First edition. Sebastopol, CA: O'Reilly. ISBN 978-1-4919-5029-6. QA76.73.J39 E45 2014

ELLIOTT, E., 2017. *Composing Software: An Exploration of Functional Programming and Object Composition in JavaScript*. S.l.: Independently published. ISBN 978-1-66121-256-8.

ESTRADA, K., 2017. Por que es Vue.js el nuevo framework de moda. *Medium* [en línea]. [Consulta: 10 abril 2020]. Disponible en: https://medium.com/blog-apside/por-que-es-vue-js-es-el-nuevo-framework-de-moda-79de70e13ef5.

EVANS, E., 2003. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston: Addison-Wesley. ISBN 978-0-321-12521-7. QA76.76.D47 E82 2004

FOWLER, M., 2003. AnemicDomainModel. *martinfowler.com* [en línea]. [Consulta: 6 abril 2020]. Disponible en: https://martinfowler.com/bliki/AnemicDomainModel.html.

FOWLER, M., 2012. TestCoverage. *martinfowler.com* [en línea]. [Consulta: 26 febrero 2020]. Disponible en: https://martinfowler.com/bliki/TestCoverage.html.

GAMMA, E., HELM, R., JOHNSON, R. y VLISSIDES, J., 1995. *Design patterns: elements of reusable object-oriented software*. Reading, Mass: Addison-Wesley. Addison-Wesley professional computing series. ISBN 978-0-201-63361-0. QA76.64 .D47 1995

GÓMEZ, M.A., 2019. *Clean Code, SOLID y Testing aplicado a JS* [en línea]. 1ª ed. España: Autoeditado. Disponible en: https://softwarecrafters.io/cleancode-solid-testing-js.

GOYAL, R., 2019. Apache Vs NGINX – Which Is The Best Web Server for You? *ServerGuy.com* [en línea]. [Consulta: 10 abril 2020]. Disponible en: https://serverguy.com/comparison/apache-vs-nginx/.

HOYOS, M., 2019. What is an ORM and Why You Should Use it. *Medium* [en línea]. [Consulta: 1 mayo 2020]. Disponible en: https://blog.bitsrc.io/what-is-an-orm-and-why-you-should-use-it-b2b6f75f5e2a.

HTTP Methods – REST API Verbs – REST API Tutorial. [en línea], [sin fecha]. [Consulta: 13 mayo 2020]. Disponible en: https://restfulapi.net/http-methods/.

HUNTER, T., 2019. Why should I use a Reverse Proxy if Node.js is Production-Ready? *Medium: Intrinsic* [en línea]. [Consulta: 27 febrero 2020]. Disponible en: https://medium.com/intrinsic/why-should-i-use-a-reverse-proxy-if-node-js-is-production-ready-5a079408b2ca.

Is using a UUID as a primary key in Postgres a performance hazard? *GitHub* [en línea], 2017. [Consulta: 5 abril 2020]. Disponible en: https://github.com/atom/teletype-server/issues/25#issuecomment-341031619.

JEFFRIES, R., 2015. *The nature of software development: keep it simple, make it valuable, build it piece by piece*. 1st ed. Dallas, Texas: The Pragmatic Bookshelf. ISBN 978-1-941222-37-9. QA76.76.D47 J444 2015

KAPADNIS, J., 2019. REST: Good Practices for API Design. *Hashmap - Medium* [en línea]. [Consulta: 10 mayo 2020]. Disponible en: https://medium.com/hashmapinc/rest-good-practices-for-api-design-881439796dc9.

Kata (programming). *Wikipedia* [en línea], 2019. [Consulta: 20 marzo 2020]. Disponible en: https://en.wikipedia.org/w/index.php?title=Kata_(programming)&oldid=931390144.

MARTIN, R.C., 2012. The Clean Architecture. *Clean Coder Blog* [en línea]. [Consulta: 12 marzo 2020]. Disponible en: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html.

MILLETT, S., 2015. *Patterns, principles, and practices of domain-driven design*. Indianapolis, IN: Wrox, a Wiley Brand. WROX professional guides. ISBN 978-1-118-71470-6. MLCM 2018/46553 (Q)

MORENO, N., VALLECILLO, A., ROMERO, J.R. y DURÁN, F.J., [sin fecha]. *Arquitectura del software*. S.l.: Universitat Oberta de Catalunya.

OpenSSL Essentials: Working with SSL Certificates, Private Keys and CSRs. *DigitalOcean* [en línea], [sin fecha]. [Consulta: 19 abril 2020]. Disponible en: https://www.digitalocean.com/community/tutorials/openssl-essentials-working-with-ssl-certificates-private-keys-and-csrs.

PRADEL I MIQUEL, J. y RAYA, J.A., [sin fecha]. *Catálogo de patrones*. S.l.: Universitat Oberta de Catalunya.

¿Qué es DevOps? Explicación de DevOps | Microsoft Azure. [en línea], [sin fecha]. [Consulta: 13 abril 2020]. Disponible en: https://azure.microsoft.com/es-es/overview/what-is-devops/.

RUIZ, J., 2016. Qué es DevOps (y sobre todo qué no es DevOps). [en línea]. [Consulta: 12 abril 2020]. Disponible en: https://www.paradigmadigital.com/techbiz/que-es-devops-y-sobre-todo-que-no-es-devops/.

Transpilador. *Wikipedia, la enciclopedia libre* [en línea], 2019. [Consulta: 27 febrero 2020]. Disponible en: https://es.wikipedia.org/w/index.php?title=Transpilador&oldid=121585326.

VERNON, V., 2013. *Implementing domain-driven design*. Upper Saddle River, NJ: Addision-Wesley. ISBN 978-0-321-83457-7. QA76.76.D47 V45 2013

VERNON, V., 2016. *Domain-driven design distilled*. 1st ed. Boston: Addison-Wesley. ISBN 978-0-13-443442-1. QA76.76.D47 V44 2016

ANEXO

Replanificaciones

Primera replanificación

Llevada a cabo el 10 de abril de 2020.

Dentro de los riesgos detectados no se encontraba el de una pandemia global como la acontecida por el COVID-19. El cierre de colegios junto con la declaración en España del estado de alarma han provocado un impacto muy grave en la planificación inicial planteada.

La jornada laboral que se compagina con el desarrollo de este TFG se realiza en remoto sin mayor impacto. Es una fortuna que la cultura de la empresa tuviera ya contemplado el trabajo en remoto, el horario libre y la comunicación asíncrona en el flujo de trabajo. Sin embargo, una hija de dos años en confinamiento hace que aquélla se alargue más de lo normal. Además, el hecho de que ambos progenitores trabajen hace que las jornadas laborables de ambos no puedan llevarse a cabo a la vez al ser imposible poder recurrir a otras personas.

Esto deriva en que las cuatro horas diarias planificadas para el desarrollo del proyecto no están aseguradas de forma constante y que nuevas planificaciones tengan un alto grado de incertidumbre.

Otra de las razones para el desvío es que el diseño de la arquitectura de la aplicación ha llevado más de lo planificado (38h) debido a tareas de investigación y documentación. El diseño dirigido por el dominio es un tema apasionante y que aporta soluciones a problemas comunes en el desarrollo de software. Sin embargo, mucha materia es a veces muy conceptual y su aplicación es sumamente abierta y no existe un decálogo de técnicas y patrones a aplicar.

Autores como Millett ya alertan que uno de los errores comunes al aplicar DDD es el de que problemas sencillos se conviertan en complejos (Millett 2015, p.126). Normalmente las soluciones propuestas en libros y artículos son complejas a fin de mostrar todas las técnicas (Domain Events, CQRS, Event Sourcing…) y sin una suficiente labor de documentación es fácil caer en el error de que son todas necesarias para

llevar a cabo DDD. Dicho esto, el tiempo invertido, dado que el objetivo era de aprendizaje, ha sido muy fructífero.

Otro error común es el de aplicar DDD dentro de una metodología de desarrollo no iterativa (Millett 2015, p.128). Dado que en este proyecto se está siguiendo un desarrollo en cascada, choca con la idea de que un modelo no sea el correcto en un primer intento y que no haya que dejar de experimentar por si se descubren soluciones mejores que hagan evolucionarlo. No ha sido un impedimento para llevar a cabo un diseño que posiblemente no deba evolucionar, pero si que conlleva una repercusión dentro una planificación en cascada.

Tras todo lo comentado anteriormente, se considera necesario aplicar una de las medidas contempladas para otro tipo de riesgos en la Tabla 2: Riesgos detectados: eliminar funcionalidades, requisitos de menor valor o revisando alcance de tareas.

Queda descartada la tarea DIS2, por el escaso valor que aportarían unos wireframes en un proyecto de desarrollo como éste. Dado que no se van a realizar pruebas con usuarios ni involucrar a usuarios objetivo en el desarrollo, disponer de aquéllos no aportará el valor que si tendría dentro de un diseño centrado en el usuario: el de poder probar, corregir y modificar los diseños antes de que se desarrollen completamente (Garreta y Mor).

Se revisa el alcance de la tarea DEV1 y a priori se descarta el requisito RP5 ya que la automatización del test no genera el valor suficiente. Durante el desarrollo el código desplegado en el servidor de desarrollo no está optimizado y puede contener código de desarrollo (hot reloading…). Esto ocasiona que el test automatizado durante el desarrollo pueda devolver resultados no reales. Si para realizar el test hay que levantar el servidor de producción ad-hoc, es escaso el valor de automatizar el test, pudiendo realizarlo manualmente.

Estas decisiones eliminan muy poco valor y acercan el estado actual del TFG a la planificación inicial, aunque siga con algo de retraso. Dado que se prevé que la situación extraordinaria provocada por la pandemia se alargue durante semanas, es muy posible que haga falta tomar nuevas medidas en el futuro.

Segunda replanificación

Llevada a cabo el 19 de mayo de 2020.

Tras un mes de desarrollo se han confirmado algunos de los riesgos detectados durante en el apartado de “Evaluación de riesgos”.

Para empezar, los relativos a cargas familiares y laborales, cuyos efectos se han visto exponenciados por el estado de alarma derivado de la pandemia del COVID-19. Ha sido posible compensar los horarios para que no repercutan en el número de horas destinadas a trabajar en el TFG, sin embargo ha repercutido en la calidad de las mismas debido a realizar muchas de noche, junto al cansancio acumulado.

Sin embargo, el riesgo confirmado cuyo impacto ha sido mayor ha sido el de una mala planificación. Principalmente en dos aspectos:

- No haber dado la importancia necesaria al tiempo extra que añaden:
  - Los tests que se realizan según TDD y sobre todo la preparación de *mocks*. En ocasiones, test fallidos son debidos a que el *mock* no es correcto o no cubre la necesidad (por ejemplo, la librería para el ORM no era capaz de emular en su totalidad el comportamiento del mismo), con el el consiguiente gasto de tiempo en detectar el fallo en el *mock* o desarrollar una solución. En otras, el resultado fallido de un test era demasiado críptico como para encontrar la causa posible, derivando en inversión extra de tiempo.
  - La complejidad añadida de una arquitectura basada en DDD: Objetos de dominio, DTOs, transformadores, repositorios, servicios… En aplicaciones grandes son de gran utilidad y clarifican mucho el flujo de una aplicación, pero cuando se realizó la planificación no existía ni el diseño ni lo que supondría a la hora de estimar ciertas pantallas.

- Mezclar tareas horizontales (a priori centradas en una capa como DEV4 “API Back end” o DEV9 “Parte pública”) con otras más verticales. Para la arquitectura elegida fue un error. Por ejemplo, para la API REST es necesario implementar en dominio, aplicación e infraestructura, sin embargo se concibió como una tarea como si servidor fuese una capa. Por otro lado, no se pudo prever que, si existía una desviación en la planificación, una tarea como la API completa de forma aislada podría derivar en consumir demasiado tiempo y no conseguir un entregable funcional. Durante el desarrollo se ha podido comprobar que trabajar por *features*, era mucho más lógico por el diseño de la arquitectura como por el resultado final.

En conclusión, es necesario adoptar algunas de las medidas contempladas en la evaluación de riesgos. Por un lado, sobrepasar las horas estimadas y, por otro, eliminar funcionalidades en base al valor que aporten al proyecto.

En primer lugar, teniendo el cuenta el tiempo restante, se elimina la funcionalidad de agrupar las katas en módulos e intercalar preguntas de tipo test. Aunque parezca drástico, es necesario recordar el objetivo

principal planteado al comienzo de esta memoria: "*disponer de una herramienta práctica y sencilla con la que poder reafirmar conocimientos sin abandonar a los desarrolladores frente a una lista de libros o enlaces a sitios web*".

Inicialmente, el *core* de la aplicación (la realización de katas) por sí sólo, de estar bien resuelto, cumpliría con el objetivo,. La agrupación en módulos fue añadida con el objetivo de hacer una aplicación más completa. Sin embargo, en la situación actual el filtrado por etiquetas permite suplir un aspecto que aportaban los módulos: agrupar tareas por temática.

En lo referido a las preguntas tipo test que derivaban de la inclusión de los módulos, su eliminación es claramente una perdida de valor de la aplicación. No obstante, un objetivo implícito en el TFG es la demostración de conocimientos adquiridos en el Grado. En este contexto académico el valor aportado por esta funcionalidad es menor que el de implementar correctamente un sistema de "login / registro / sesiones de usuario" o documentar mejor decisiones tomadas. Por tanto, se considera un error dedicar esfuerzos a tareas que hagan que la aplicación parezca más compleja a cambio de un proyecto más simple.

En segundo lugar, se limitará el alcance de la tarea DEV10 "Registro y login", eliminando el requisito RF13 y con él la funcionalidad de cambiar la contraseña. En base a la experiencia de este mes, es previsible que las ocho horas estimadas sean insuficientes. Las razones de ámbito académico esgrimidas anteriormente hacen que sea más interesante el aspecto de la sesión y seguridad que otros cómo el de poder cambiar la contraseña.

Y en tercer y último lugar, la documentación del plan de pruebas se realizará en la siguiente fase. A día de hoy, la cobertura de test es de alrededor del 70%. Esto hace que disponer de un plan de pruebas prefijado antes de la entrega de la PEC3 sea mucho menos crítico que al no disponer de ninguna batería de pruebas. Esto no significa que no se vayan a realizar pruebas de producto final, sino que la documentación de las mismas se prevé realizarla después de la tercera entrega si la planificación lo permite.