Aplicación web de katas de código

David Marchena Blanco
Grado de Ingeniería Informática
Desarrollo web

Gregorio Robles Martínez
Santi Caballe Llobet

Fecha Entrega

Licencias alternativas (elegir alguna de las siguientes y sustituir la de la página anterior)

A) Creative Commons:

Esta obra está sujeta a una licencia de Reconocimiento-NoComercial-SinObraDerivada 3.0 España de Creative Commons

Esta obra está sujeta a una licencia de Reconocimiento-NoComercial-CompartirIgual 3.0 España de Creative Commons

Esta obra está sujeta a una licencia de Reconocimiento-NoComercial 3.0 España de Creative Commons

Esta obra está sujeta a una licencia de Reconocimiento-SinObraDerivada 3.0 España de Creative Commons

Esta obra está sujeta a una licencia de Reconocimiento-CompartirIgual 3.0 España de Creative Commons

Esta obra está sujeta a una licencia de Reconocimiento 3.0 España de Creative Commons

B) GNU Free Documentation License (GNU FDL)

Copyright © AÑO TU-NOMBRE.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free

C) Copyright

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Descripción del trabajo*
Nombre del autor:	*Nombre y dos apellidos*
Nombre del consultor/a:	*Nombre y dos apellidos*
Nombre del PRA:	*Nombre y dos apellidos*
Fecha de entrega (mm/aaaa):	*MM/AAAA*
Titulación::	*Plan de estudios del estudiante*
Área del Trabajo Final:	*El nombre de la asignatura de TF*
Idioma del trabajo:	
Palabras clave	*Máximo 3 palabras clave, validadas por el director del trabajo (dadas por los estudiantes o en base a listados, tesauros, etc.)*
Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados i conclusiones del trabajo.*	

Abstract (in English, 250 words or less):

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 de al menos el 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

---

1 *Test-driven development*

2 *User experience*

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

aplicación de los detalles técnicos y de infraestructura, como: clean 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

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

soporte casi completo a ES2015[5], de forma que su motor javascript nos permita enviar un código final de la aplicación más optimo y ofrecer katas en dicho lenguaje sin tener que hacer transpilación en tiempo de ejecución. En formato browserlist el soporte se resumiría en:

```
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
```

1.2.1.3 CSS

Son las siglas para referirse a *Cascading Style Sheets*, el lenguaje para dar crear 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.

El uso 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.

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

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 esquéma 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].

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 branch locales y remotos.
- Mejor rendimiento que un sistema centralizado.
- Es gratuito y *open-source*.

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)

1.2.3.3 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 codigo…

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

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

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.

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 automatico lighthouse, transpilacion 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 backend 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 backend
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

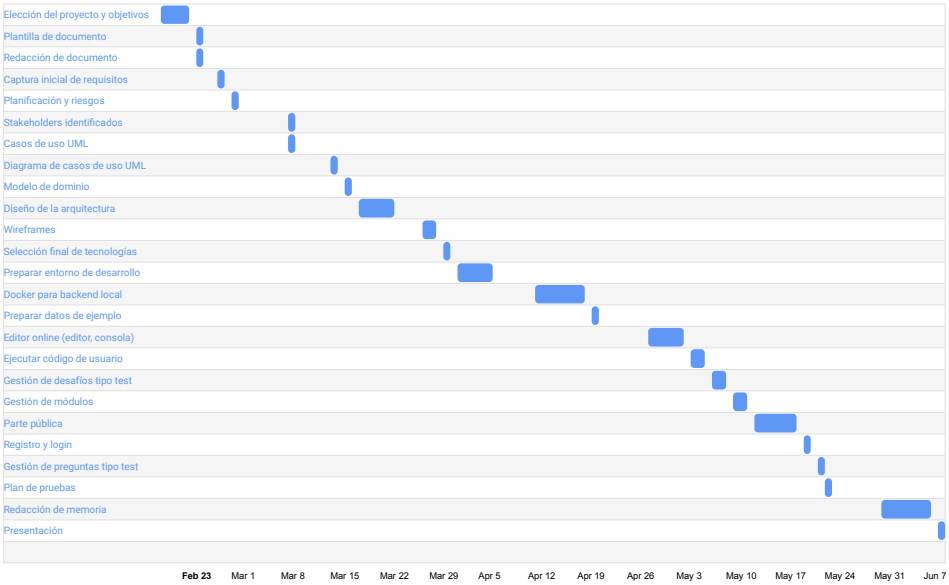

Figura 1: Diagrama de Gantt

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 reevaluar 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 backend 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.
- RF13, 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

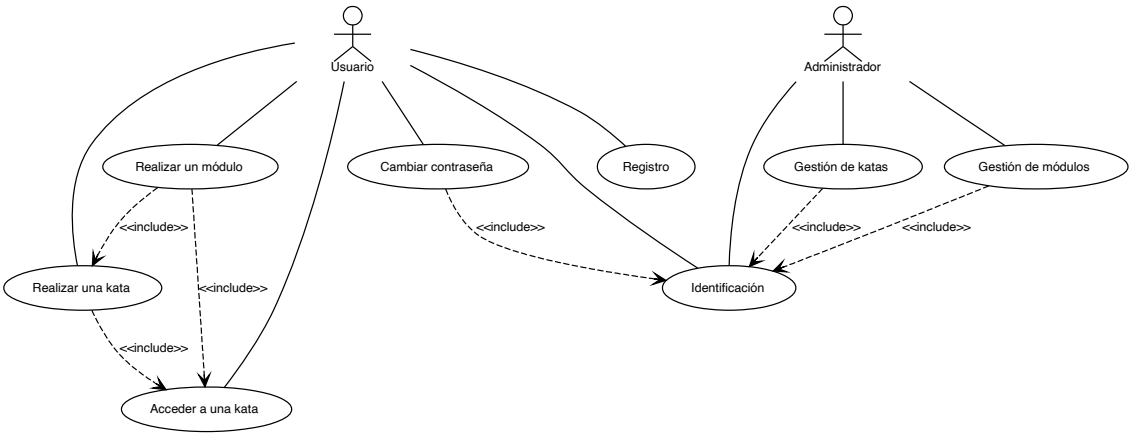

Figura 2: Diagrama de casos de uso

CAPÍTULO 3

Diseño

3.1 Arquitectura de la aplicación

Tal y como se describía en la captura de requisitos, para el diseño de la aplicación se va a intentar utilizar un diseño dirigido por el modelo (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, envio 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).

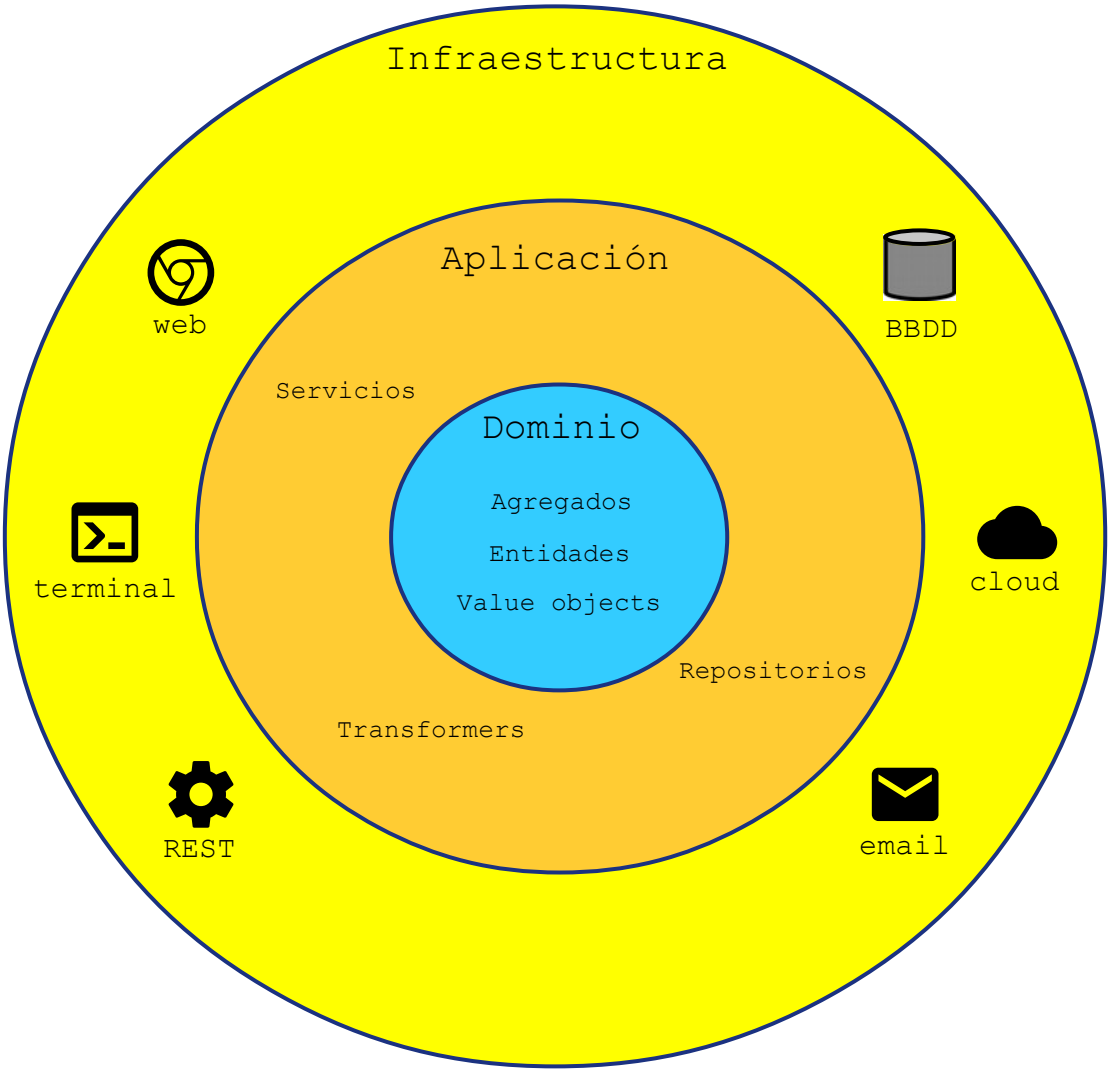

Figura 3: Arquitectura DDD en capas concéntricas

3.1.1 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 raiz 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.*

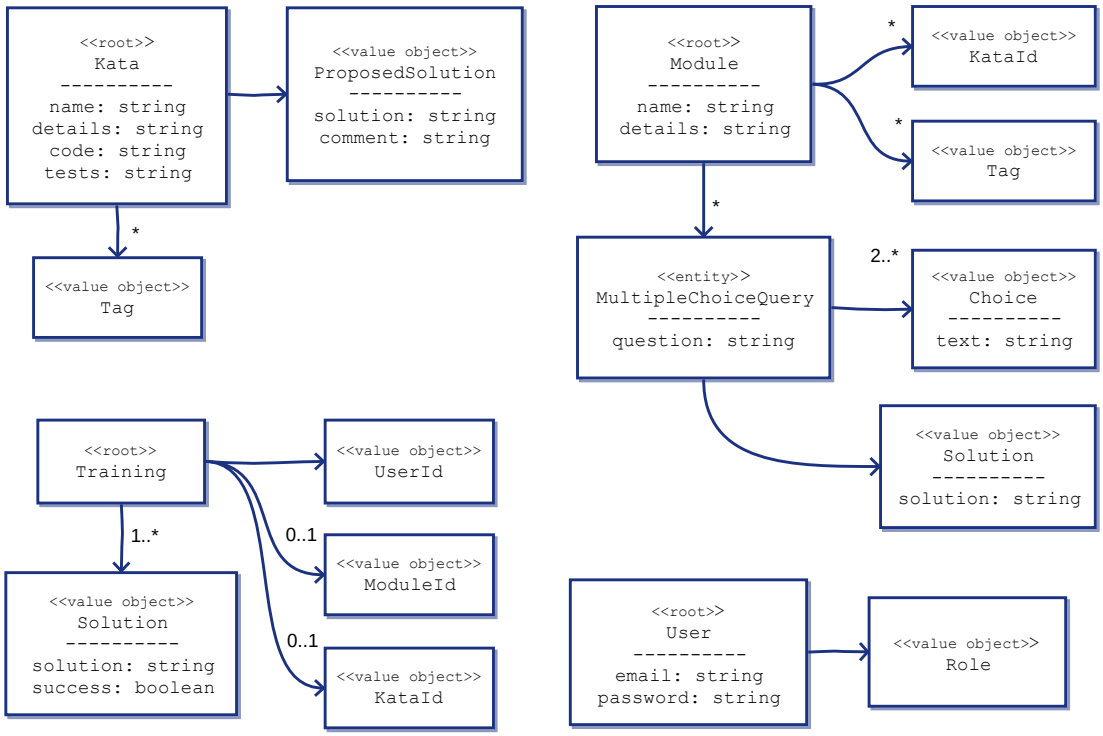

Figura 4: Core Domain inicial

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 subdomios:

- *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:

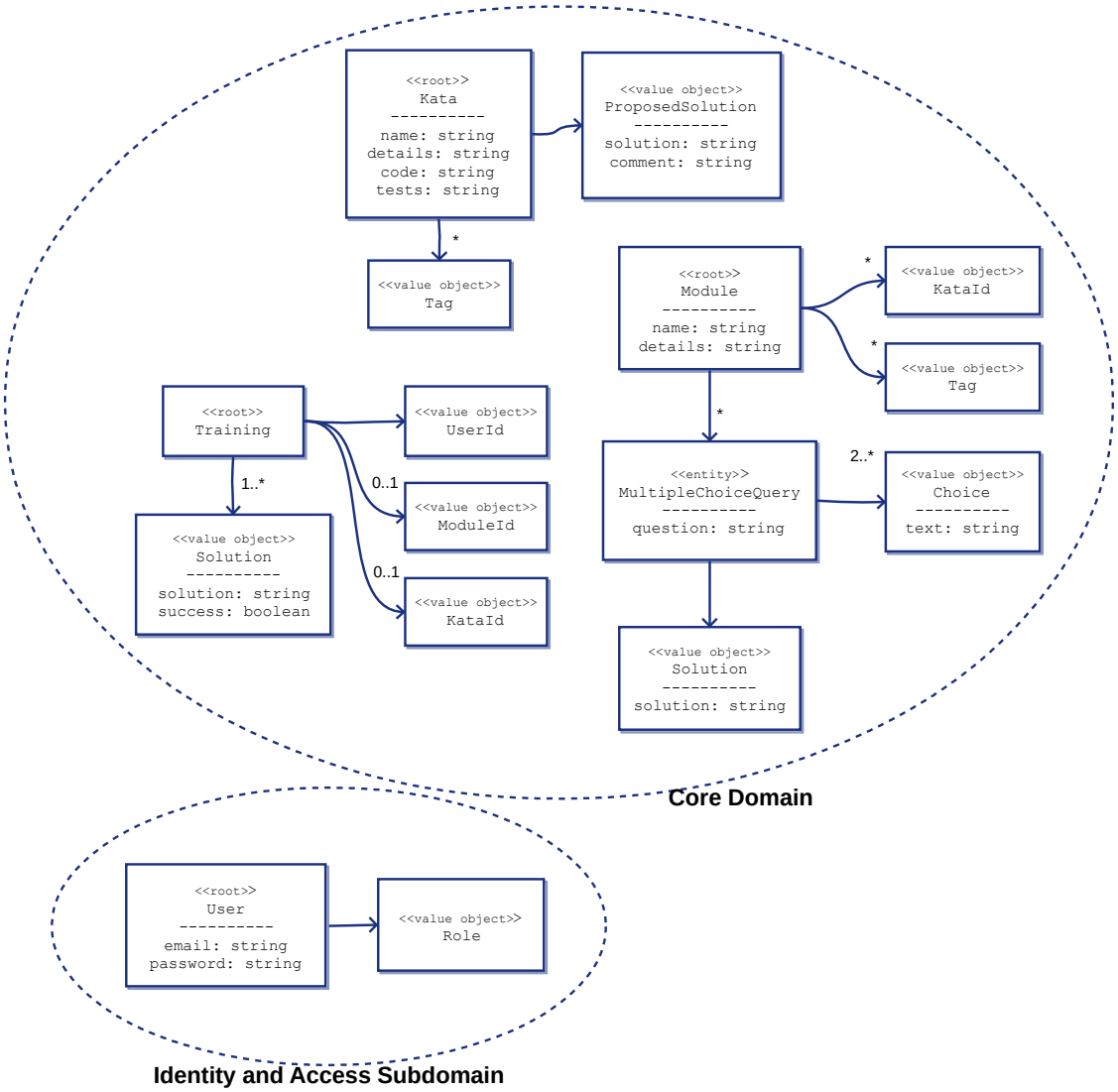

Figura 5: Núcleo segregado

3.1.2 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.

3.1.2.1 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`

3.1.2.2 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 prototipos de Javascript.

Sin embargo, en este proyecto no se considera necesario el uso de

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/

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.

3.1.2.3 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*.

3.1.2.4 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`

3.1.3 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.

3.1.3.1 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.

3.1.3.2 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 el 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 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 reusable, 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 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

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".*

"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).

ANEXO I

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.

Glosario

BUNDLER Herramienta que permite combinar multiples 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* (tambien 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.

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.

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.

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

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

GARRETA, M. y MOR, E., [sin fecha]. *Diseño centrado en el usuario*. S.l.: Universitat Oberta de Catalunya.

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/.

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

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.

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

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