Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABAJO FIN DE GRADO
GRADO EN INGENIERÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

Análisis y desarrollo de heurísticas y guías de usabilidad de RESTFUL APIs y aplicación a un caso práctico

Estudiante: Nerea Vázquez Callejón
Dirección: David Alonso Ríos
Eduardo Mosqueira Rey

A Coruña, septiembre de 2020.

A mi padre

Agradecimientos

A mi familia y amigos por acompañarme y apoyarme durante toda la carrera.

A mis tutores, Eduardo Mosqueira y David Alonso por darme la oportunidad de realizar este proyecto de investigación y por haberme ayudado y guiado en todo momento. Gracias.

Nerea.

Resumen

La usabilidad se refiere a la capacidad de un software de ser comprendido, aprendido, usado y ser atractivo para el usuario, en condiciones específicas de uso (ISO/IEC 9126). Por tanto, integrar la usabilidad es uno de los factores clave en cualquier proceso para el desarrollo de software. El objetivo principal de este Trabajo de Fin de Grado es desarrollar un conjunto de heurísticas y guías de usabilidad para el diseño de APIs RESTful a partir de un análisis exhaustivo sobre los principios del estilo REST dentro de un modelo expandido de usabilidad aplicable a cualquier producto o sistema. Para demostrar la eficacia de este conjunto, se estudió la usabilidad en un caso práctico, precisamente en la API REST para desarrolladores de Twitter, mediante una evaluación heurística, una técnica introducida por Nielsen y Molich en 1990 que consiste en examinar la calidad de uso de un sistema a partir del cumplimiento de un conjunto de heurísticas. Previamente, caracterizamos el contexto de uso a partir de las características de los usuarios, tareas y entorno que están implicados en una API REST para definir con mayor precisión los problemas encontrados mediante el estudio de usabilidad. Por último, propusimos varias mejoras para resolver algunas de las carencias de usabilidad detectadas.

Abstract

Usability refers to the ability of a software to be understood, learned, used and attractive to the user, under specific conditions of use (ISO/IEC 9126). Therefore, integrating usability is one of the key factors in any software development process. The main objective of this end of degree project is to develop a set of heuristics and usability guidelines for the design of RESTful APIs based on an exhaustive analysis of the principles of the REST style within an expanded usability model applicable to any product or system. To demonstrate the effectiveness of this set, we studied usability in a case study, specifically the REST API for Twitter developers, using heuristic evaluation, a technique introduced by Nielsen and Molich in 1990, which consists of examining the quality of use of a system based on compliance with a set of heuristics. Previously, we characterized the context of use from the characteristics of users, tasks and environment that are involved in an API REST to define more precisely the problems found through the usability study. Finally, we proposed several improvements to solve some of the detected usability shortcomings.

Palabras clave:

- Usabilidad
- Interfaz de programación de aplicaciones
- API
- RESTful
- REST
- HTTP
- Desarrollo de heurísticas y guías
- Estudio de usabilidad
- Evaluación heurística

Keywords:

- Usability
- Application programming interface
- API
- RESTful
- REST
- HTTP
- Development of heuristics and guides
- Usability study
- Heuristic evaluation

Índice general

1 Introducción **1**

1.1 Transferencia de Estado Representacional (REST) . . . 2
1.2 Objetivos . . . 4
1.3 Estructura de la memoria . . . 5

2 Estado del arte **7**

2.1 Modelo de usabilidad . . . 9
2.2 Campo de aplicación: APIs RESTful . . . 11

3 Metodología **13**

3.1 Desarrollo de heurísticas y guías de usabilidad . . . 14
3.1.1 Caracterización del sistema . . . 14
3.1.2 Creación y organización de heurísticas y guías . . . 14
3.2 Evaluación heurística en un caso práctico . . . 15
3.2.1 Caracterización del contexto de uso y evaluación heurística . . . 15

4 Planificación **17**

4.1 Recursos . . . 19
4.2 Tiempo . . . 19
4.3 Costes . . . 19

5 Diseño de heurísticas y guías de usabilidad **23**

5.1 Caracterización del sistema . . . 23
5.1.1 Solicitud HTTP/1.1 . . . 24
5.1.2 Respuesta HTTP/1.1 . . . 25
5.2 Análisis bibliográfico sobre RESTful APIs . . . 27
5.3 Mapeo entre la taxonomía de usabilidad y la literatura de APIs RESTful . . . 29
5.3.1 Primera clasificación de la información . . . 29

5.3.2 Segunda clasificación de la información 30
5.4 Codificación de la información en forma de heurísticas y guías 30
5.4.1 Integración de guías 31
5.4.2 Creación de heurísticas 32
5.5 Heurísticas y guías de usabilidad 32

6 Aplicación a un caso práctico **49**
6.1 Caso práctico 49
6.2 Caracterización del contexto de uso 50
6.3 Evaluación heurística de usabilidad 52
6.4 Análisis de resultados 58
6.5 Propuestas de mejora 59

7 Discusión y conclusiones **63**

A Publicación en un congreso **69**

Lista de acrónimos **81**

Glosario **83**

Bibliografía **85**

Índice de figuras

1.1 Arquitectura de un servicio web REST 4

2.1 Primeros niveles de la taxonomía de usabilidad 10
2.2 Primeros niveles de la taxonomía de contexto de uso 11

4.1 Plan de tareas del trabajo el 10 de agosto de 2019 20
4.2 Plan de tareas del trabajo el 26 de octubre de 2019 21
4.3 Plan de tareas del trabajo el 3 de mayo de 2020 21
4.4 Plan de tareas del trabajo el 6 de julio de 2020 22

5.1 Web API 23
5.2 Taxonomía de elementos del caso de estudio 27

6.1 Resultados de la evaluación heurística 59

Índice de tablas

2.1 Primera y segunda versión de las heurísticas de Nielsen 7
2.2 Comparación de modelos tradicionales de usabilidad con algunos de los "modelos extendidos de usabilidad" [1] 9

4.1 Estimación de costes del trabajo 19

5.1 Cognoscibilidad parte I 33
5.2 Cognoscibilidad parte II 34
5.3 Cognoscibilidad parte III 35
5.4 Cognoscibilidad parte IV 36
5.5 Operatividad parte I 37
5.6 Operatividad parte II 38
5.7 Eficiencia parte I 39
5.8 Eficiencia parte II, Robustez y Seguridad parte I 40
5.9 Seguridad parte II 41
5.10 Seguridad parte III 42
5.11 Satisfacción subjetiva parte I 43
5.12 Ejemplos (verde) y contraejemplos (rojo) de las heurísticas de usabilidad (parte I) 45
5.13 Ejemplos (verde) y contraejemplos (rojo) de las heurísticas de usabilidad (parte II) 46
5.14 Ejemplos (verde) y contraejemplos (rojo) de las heurísticas de usabilidad (parte III) 47
5.15 Ejemplos (verde) y contraejemplos (rojo) de las heurísticas de usabilidad (parte IV) 48

6.1 Criterios de evaluación 53
6.2 Evaluación heurística (parte I) 53
6.3 Evaluación heurística (parte II) 54

6.4 Evaluación heurística (parte III) 55
6.5 Evaluación heurística (parte IV) 56
6.6 Evaluación heurística (parte V) 57
6.7 Propuestas de mejora (parte I) 60
6.8 Propuestas de mejora (parte II) 61
6.9 Propuestas de mejora (parte III) 61
6.10 Propuestas de mejora (parte IV) 61
6.11 Propuestas de mejora (parte V) 61

Capítulo 1

Introducción

REST (Representational State Transfer) es un estilo de arquitectura software para sistemas hipermedia distribuidos que describe cualquier interfaz entre sistemas para obtener o ejecutar operaciones sobre datos. Este estilo fue presentado por Roy T. Fielding, uno de los principales autores de la especificación del protocolo de transferencia de hipertexto (HTTP) en el año 2000 [2]. Para indicar que se cumple completamente con los principios del estilo, se utiliza el término RESTful, que habitualmente se emplea como sinónimo de REST.

Una interfaz de programación de aplicaciones (API) es un conjunto de definiciones y protocolos que se utilizan para desarrollar e integrar el software de una o varias aplicaciones.

En la actualidad, son muchas las empresas que presentan una API RESTful para así extender la implantación de los servicios que ofrece a otras empresas y desarrolladores. Esto es corroborado por los principales sitios web como Google, Facebook o Twitter, que ahora están implementando servicios REST para proporcionar un fácil acceso a sus valiosos recursos de datos, mientras promueven y enriquecen sus negocios.

Para que los usuarios que utilicen una API puedan alcanzar sus objetivos con eficiencia y efectividad es necesario garantizar su usabilidad. Para ello, proponemos un conjunto de heurísticas y guías de usabilidad para una API RESTful, utilizando una taxonomía organizada jerárquicamente, formada por atributos que constituyen la usabilidad [3], para ordenar la información y detectar deficiencias.

Posteriormente, las heurísticas y guías desarrolladas se probarán aplicándolas a un caso práctico, particularmente a la API para desarrolladores de Twitter, y finalmente realizaremos una evaluación heurística que nos indicará qué aspectos de la API no cumplen con los principios y características de la usabilidad.

1.1 Transferencia de Estado Representacional (REST)

Roy T. Fielding presentó, en su tesis doctoral sobre la web [2], el estilo de arquitectura de transferencia de estado representacional cuya finalidad era guiar el diseño y desarrollo de una arquitectura para la web moderna.

El estilo de arquitectura REST consiste en:

- **Estilo de arquitectura cliente-servidor**: la gestión y almacenamiento de los datos se centran en el servidor, por lo tanto, mejoramos la portabilidad de la interfaz de usuario en múltiples plataformas, aunque lo interesante es que la separación permite que las dos partes evolucionen de manera independiente.

- **Protocolo cliente-servidor sin estado**: cada solicitud del cliente al servidor contiene toda la información necesaria para procesar la petición. El estado de la sesión se mantiene completamente en el cliente. Con esta restricción, en primer lugar, se mejora la visibilidad ya que un sistema no necesita tener en cuenta información adicional para determinar la naturaleza completa de la solicitud. En segundo lugar, la confiabilidad también se ve mejorada, porque se facilita la recuperación de fallos parciales. Por último, al no tener que almacenar el estado entre las solicitudes, el servidor puede liberar recursos rápidamente y su implementación se simplifica ya que no tiene que administrar el uso de recursos entre las solicitudes, de modo que la escalabilidad también se ve mejorada.

- **Almacenamiento en caché**: si una respuesta es almacenable en caché, se le concede a la caché del cliente reutilizar los datos de las respuestas para solicitudes posteriores que sean equivalentes. La ventaja es que elimina, ya sea parcial o completamente, las interacciones, mejorando así la eficiencia, la escalabilidad y el rendimiento que percibe el usuario al reducir la latencia de respuesta.

- **Interfaz uniforme**: al aplicar el principio de generalidad de ingeniería de software a la interfaz del servidor, se simplifica la arquitectura general del sistema y se mejora la visibilidad de las interacciones. REST está definido por cuatro restricciones de interfaz: identificación de recursos, manipulación de recursos a través de representaciones, mensajes auto-descriptivos e hipermedia como motor de estado de la aplicación (conocido como principio de HATEOAS: *Hypermedia as the Engine of Application State*).

- **Sistema en capas**: un sistema en capas se organiza jerárquicamente. Cada capa proporciona servicios a la capa superior y utiliza servicios de la capa inferior. Este tipo de sistema reduce el acoplamiento al ocultar todas las capas internas y mejora la capacidad de reutilización. Estas capas pueden ofrecer funcionalidades adicionales, como

equilibrio de carga, cachés compartidos o seguridad.

REST utiliza varios tipos de conectores para encapsular el acceso a recursos y la transferencia de sus representaciones. Algunos de estos conectores son: el servidor, cliente o caché, entre otros.

Algunos de los componentes que intervienen en una acción de aplicación pueden ser: el servidor origen y el agente de usuario. Los componentes REST se comunican mediante la transferencia de una representación de un recurso en un formato de tipo de datos estándar que son seleccionados en función de las necesidades del cliente o del servidor.

Los elementos de arquitectura que definen REST podemos destacar los siguientes (Figura 1.1 [1]):

- **Recursos**: son la abstracción clave de la información. Un recurso puede ser un documento, una imagen, una colección de otros recursos, un servicio, etc. Es decir, cualquier información que se pueda nombrar.

  Podemos distinguir entre recursos estáticos y dinámicos. Los recursos estáticos son los que su conjunto de valores se mantiene en el tiempo después de su creación, en cambio los recursos dinámicos tienen un grado alto de variación a lo largo del tiempo. Esto es importante mencionarlo, ya que dos recursos que se comportan de manera diferente pero llegan a tener el mismo valor en un momento dado, son distintos y es necesario que ambos recursos puedan ser identificados y referenciados independientemente.

- **Identificador de recursos uniforme (URI)**: se utilizan para identificar un recurso en particular que está involucrado en una interacción entre componentes. REST proporciona una interfaz genérica y uniforme para acceder y manipular el conjunto de datos de un recurso.

- **Representaciones**: los componentes realizan acciones sobre un recurso utilizando una representación para capturar el estado actual de ese recurso y transferirla entre ellos. Una representación está formada por datos, enlaces hipermedia que pueden ayudar a los clientes en la transición al siguiente estado deseado del recurso, metadatos para describir los datos, y en ocasiones, metadatos adicionales para verificar la integridad del mensaje. El formato de datos de una representación se conoce como tipo de medio, que identifica una especificación que define como se procesará dicha representación.

[1] Imagen traducida al español de la imagen obtenida de [4]

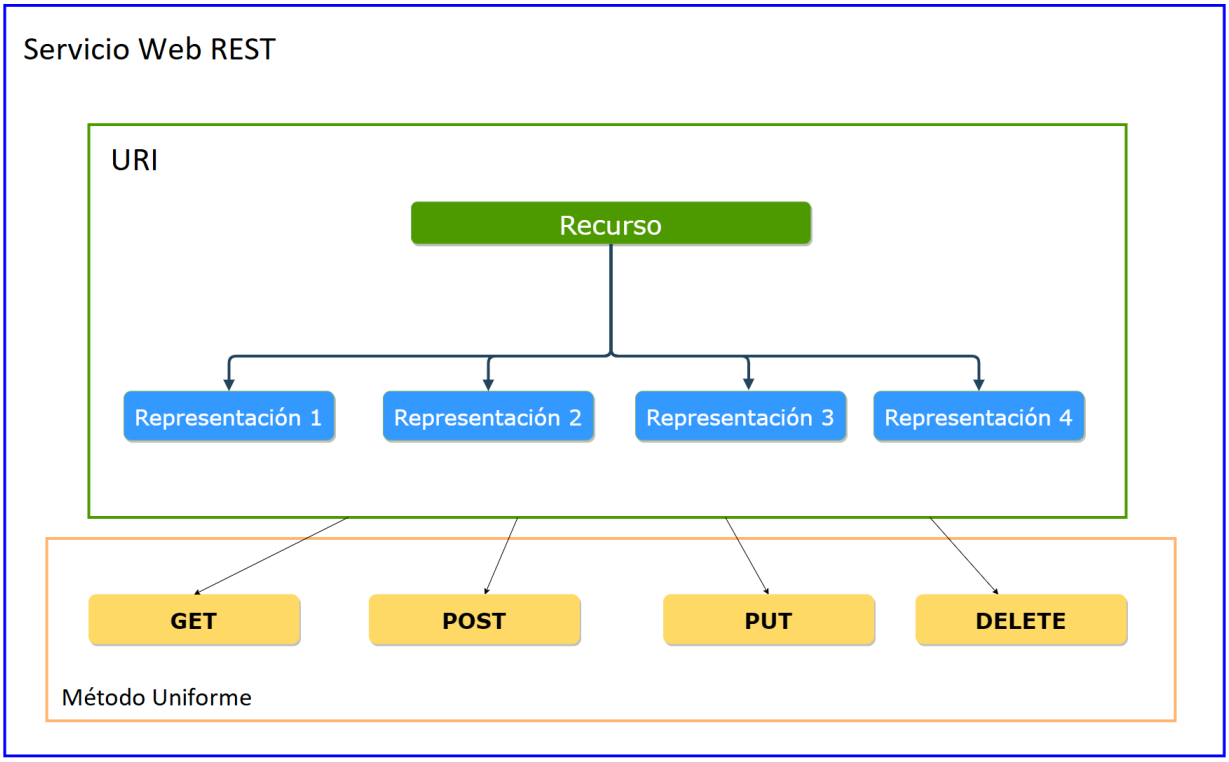

Figura 1.1: Arquitectura de un servicio web REST

Por lo tanto, este estilo de arquitectura logra separar las responsabilidades de estilo cliente-servidor sin el problema de la escalabilidad del servidor y, mediante una interfaz genérica, permite la encapsulación y la evolución de los servicios.

Cabe destacar que, entre las restricciones que componen REST, solo se establecen pautas de diseño a alto nivel, por lo que no hay ninguna regla que obligue el uso de un protocolo específico de transferencia de información. En la actualidad, este estilo se usa para describir cualquier interfaz entre sistemas que utilicen directamente HTTP, ya que a menudo este protocolo se utiliza para el desarrollo de servicios web (Figura 1.1) que consiste en una colección de protocolos abiertos y estándares usados para intercambiar datos entre aplicaciones o sistemas.

1.2 Objetivos

La finalidad de este proyecto es desarrollar un conjunto de heurísticas y guías detalladas de análisis de usabilidad de APIs RESTful, que consisten en reglas generales sobre principios de usabilidad acompañadas de varias pautas descritas de manera ordenada sobre los fundamentos del estilo REST aplicados en APIs web. Con esto, tratamos de realizar una contribución novedosa al estado del arte, ya que no existen trabajos que permitan analizar si una API

RESTful puede ser usada por usuarios cumpliendo los principios de la interacción persona-computador. Los objetivos son los siguientes:

1. **Realizar un análisis bibliográfico sobre la usabilidad de APIs RESTful**. Es necesario hacer un amplio estudio bibliográfico de las publicaciones que traten el tema de la usabilidad de APIs RESTful o, al menos, que incluyan recomendaciones de como crear dichos APIs de forma adecuada.

2. **Desarrollar heurísticas y guías de usabilidad de APIs RESTful dentro de un modelo expandido de usabilidad**. Siguiendo dicho modelo, se desarrollarán heurísticas y guías de usabilidad que permitan realizar un estudio heurístico de la usabilidad de una API de este estilo.

3. **Evaluación heurística sobre un caso práctico**. Para comprobar que las heurísticas desarrolladas son correctas, se aplicarán a un caso práctico de una API existente y se estudiará su usabilidad y se propondrán mejoras si fuera necesario.

4. **Estudiar como el contexto de uso puede afectar a la usabilidad**. Todo análisis de usabilidad tiene que tener en cuenta el contexto de uso en el que el producto a analizar va a ser usado. En las pruebas sobre el caso práctico se tendrá en cuenta como el contexto puede afectar e influir en el análisis de usabilidad.

1.3 Estructura de la memoria

En este apartado describiremos brevemente cada capítulo de la memoria.

- **Capítulo 1: Introducción**. Se introduce el estilo de arquitectura REST y se explica cada uno de los objetivos del trabajo.

- **Capítulo 2: Estado del arte**. Se presentan el modelo de usabilidad y contexto de uso que se han utilizado a lo largo del trabajo. Se comenta varios estudios relacionados, que consisten principalmente en el desarrollo de heurísticas de usabilidad a partir de las de Nielsen y se presenta qué es la evaluación heurística.

- **Capítulo 3: Metodología**. En este capítulo se describe la metodología seguida para el desarrollo de heurísticas y guías y su aplicación a un caso práctico. Está basada en dos artículos publicados de los tutores del trabajo.

- **Capítulo 4: Planificación**. Se muestra el plan de tareas del trabajo, gestionadas bajo la metodología Kanban. Se indican los recursos utilizados durante el trabajo y se hace una estimación de su coste.

- **Capítulo 5: Diseño de heurísticas y guías de usabilidad**. En este capítulo se describe paso por paso la principal contribución del trabajo, es decir, el conjunto de heurísticas y guías de usabilidad para el diseño de APIs RESTful.
- **Capítulo 6: Aplicación a un caso práctico**. Validamos y demostramos la eficacia del conjunto anterior estudiando la usabilidad en un caso práctico mediante una evaluación heurística. También se propone varias mejoras para solventar algunos de los problemas de usabilidad detectados en el caso práctico.
- **Capítulo 7: Discusión y conclusiones**. Se muestran varias reflexiones importantes sobre los resultados obtenidos de la evaluación heurística y se expone todo el trabajo realizado.

Capítulo 2

Estado del arte

El objetivo que persigue la usabilidad es que la experiencia del usuario, al interactuar con un sistema o servicio, sea lo más satisfactoria posible. Por ello, evaluar la usabilidad alcanza un papel importante en cualquier ciclo de vida de desarrollo de un producto interactivo.

La evaluación heurística es un método de estudio de la usabilidad sin usuarios. Consiste en examinar la calidad de uso de un sistema por parte de varios evaluadores expertos, a partir del cumplimiento de un conjunto de heurísticas, que son básicamente reglas generales.

Esta técnica fue introducida por primera vez por Nielsen y Molich [5] en 1990, para analizar interfaces de usuario. Como se puede ver en la tabla 2.1, inicialmente reunieron nueve heurísticas de usabilidad que posteriormente fueron revisadas por Nielsen para formar un conjunto nuevo de heurísticas, manteniendo sus significados y descripciones, e incluir una nueva sobre *ayuda y documentación*.

Nielsen and Molich (1990)	Nielsen (1995)
Simple and natural dialogue	Aesthetic and minimalist design
Speak the user's language	Match between system and the real world
Minimize the user's memory load	Recognition rather than recall
Be consistent	Consistency and standards
Provide feedback	Visibility of system status
Provide clearly marked exits	User control and freedom
Provide shortcuts	Flexibility and efficiency of use
Good error messages	Help users recognize, diagnose, and recover from errors
Prevent errors	Error prevention
	Help and documentation

Tabla 2.1: Primera y segunda versión de las heurísticas de Nielsen

Las heurísticas de Nielsen están destinadas a ser generalmente aplicables a una amplia variedad de interfaces de software, pero muchos autores propusieron diferentes conjuntos de heurísticas para un contexto específico o modificaron y complementaron las propuestas

inicialmente para satisfacer varias necesidades en particular.

Alguno de ellos como Toribio-Guzmán et al. [6], extendieron las heurísticas de Nielsen para evaluar la usabilidad de una red social privada que monitorizaba el progreso diario de los pacientes por parte de sus familiares. En cambio, Mirel y Wright [7], las adaptaron al campo de la bioinformática para garantizar un conjunto de características en varias herramientas. Otros autores como Kölling y McKay [8] crearon una lista nueva de heurísticas para estudiar la usabilidad en herramientas de programación para principiantes y así proporcionar información útil a los diseñadores de este tipo de sistemas.

En el análisis del estado del arte del trabajo de Alonso-Ríos et al. [9], nos encontramos con más autores que siguieron esta dinámica con respecto a las heurísticas de Nielsen, como Zhang et al. [10], que las fusionaron con las "ocho reglas de oro" de Shneiderman y Plaisant [11] para evaluar la usabilidad de un software aplicado dentro del campo de la medicina y asegurar la seguridad de los pacientes, Sutcliffe y Gault [12], que las extendieron con los principios de la realidad virtual, propuestos por Sutcliffe y Kaur, para aplicárselas a las interfaces de usuario del entorno virtual o el trabajo de Baker et al. [13], donde ajustaron la metodología de evaluación heurística tradicional para crear heurísticas nuevas e identificar de manera rápida y económica los problemas de usabilidad dentro de sistemas de software colaborativo.

Bajo el mismo punto de vista de los trabajos anteriores, hemos desarrollado un conjunto de heurísticas y guías de usabilidad para hacer una evaluación heurística al campo de las APIs RESTful. Para ello se ha seguido el modelo de usabilidad propuesto en *Usability: A Critical Analysis and a Taxonomy* de Alonso-Ríos et al. [3], donde se describe el concepto de usabilidad mediante una taxonomía detallada organizada jerárquicamente, que detallaremos en el punto siguiente. Contiene descripciones precisas de los atributos que forman la usabilidad, pudiéndose aplicar de manera general a cualquier tipo de producto o sistema.

Puesto que los modelos tradicionales y más conocidos, como es el modelo introducido por Jakob Nielsen en 1993 [14], o algunos expuestos por la Organización Internacional de Normalización (ISO) (ISO/ IEC 9126-1, 2001, ISO 9241-11, 1998), no son los suficientemente completos para abarcar totalmente la definición de usabilidad, algunos investigadores como Seffah et al. [15], Winter et al. [16], Bevan [17] y Alonso-Ríos et al. [3] construyeron sus propios modelos, nombrados por Lewis [1] como "modelos extendidos de usabilidad", intentando incluir todas las interpretaciones existentes de este término para evitar ambigüedades y contradicciones a la hora de realizar cualquier estudio de usabilidad. En la tabla 2.2 mostramos a modo de comparación varios de los "modelos extendidos de usabilidad" y los modelos tradicionales para demostrar que a pesar de que parecen muy diferentes entre ellos, en realidad los conceptos están muy relacionados.

Nielsen (1993)	ISO (1998)	ISO/IEC (2001)	Seffah et al.(2006)	Alonso-Ríos et al.(2009)
Learnability	Effectiveness	Understandability	Learnability	Knowability
Memorability	Efficiency	Learnability	Effectiveness	Operability
Efficiency	Satisfaction	Operability	Usefulness	Efficiency
Errors		Attractiveness	Accessibility	Robustness
Satisfaction		Usability compliance	Universality	Safety
			Efficiency	Subjective satisfaction
			Productivity	
			Safety	
			Satisfaction	

Tabla 2.2: Comparación de modelos tradicionales de usabilidad con algunos de los "modelos extendidos de usabilidad" [1]

2.1 Modelo de usabilidad

El desarrollo de heurísticas y guías ha sido dentro de un modelo expandido de usabilidad [3], cuyos objetivos son los siguientes: "cubrir todos los aspectos de la usabilidad evitando contradicciones y redundancias, estar estructurado jerárquicamente en varios niveles, poder ser aplicado a cualquier tipo de producto y proporcionar definiciones para cualquier atributo y subatributo que integran la usabilidad" (p. 48).

La taxonomía consta de seis atributos principales, que posteriormente se desglosan en varios subatributos, que a su vez están estructurados jerárquicamente en niveles (Figura 2.1):

- **Cognoscibilidad**: se define como la propiedad del sistema mediante la cual el usuario puede entender, recordar y aprender a utilizar el sistema. Este atributo se subdivide en claridad, consistencia, memorabilidad y asistencia. Los tres primeros se aplican a aspectos formales, conceptuales y al funcionamiento de las tareas del usuario y del sistema.

- **Operatividad**: capacidad del sistema de adaptar y proporcionar las funcionalidades que los usuarios necesitan. El atributo se subdivide en completitud, precisión, universalidad y flexibilidad. Dentro de universalidad nos encontramos con Accesibilidad y Universalidad cultural, mientras que el atributo de flexibilidad se subdivide en Controlabilidad y Adaptatividad.

- **Eficiencia**: es la capacidad del sistema de proporcionar buenos resultados a cambio de los recursos invertidos. Se subdivide en eficiencia respecto al esfuerzo humano, al tiempo de ejecución de tareas, a recursos ocupados y a costes económicos. Cada una se descompone en más subatributos (ej. esfuerzo físico o mental, recursos materiales o humanos,etc.).

- **Robustez**: se define como la capacidad del sistema de ser resistente a errores y a situaciones adversas, y se desglosa en ROBUSTEZ ante fallos internos, ante usos incorrectos por parte del usuario, ante abusos por parte de terceros y ante problemas en el entorno.
- **Seguridad**: establece la capacidad del sistema de evitar que se sufra riesgos y daños al utilizarlo. Está compuesto por SEGURIDAD para el usuario, para terceros y para el entorno. Los dos primeros se subdividen aún más en SEGURIDAD FÍSICA, SEGURIDAD LEGAL, CONFIDENCIALIDAD y SEGURIDAD DE BIENES MATERIALES.
- **Satisfacción subjetiva**: define la capacidad del sistema de producir sensaciones de agrado e interés en los usuarios. Está compuesta por INTERÉS y ESTÉTICA. Este último se separa para cada uno de los cinco sentidos (VISUAL, ACÚSTICA, TÁCTIL, OLFATIVA y GUSTATIVA).

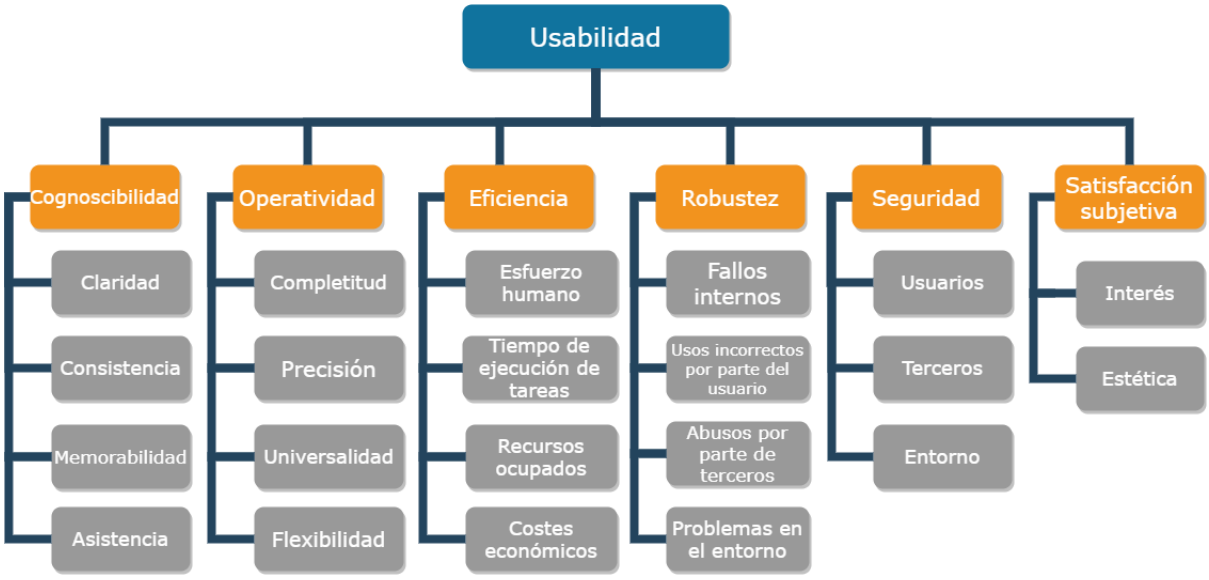

Figura 2.1: Primeros niveles de la taxonomía de usabilidad

La usabilidad depende del contexto específico del uso (ISO 9241–11), es decir, las características particulares de los usuarios, tareas y entornos. Esto significa que un producto o sistema no es utilizable en todos los contextos caracterizados por diferentes usuarios, tareas, equipamiento o entorno. Por ejemplo, si tenemos una herramienta de gestión de aprendizaje, que es utilizada por alumnos para entregar tareas y que los docentes puedan revisarlas, la usabilidad del sistema será diferente para ambos, ya que cada uno desempeña un rol y realiza tareas con fines diferentes.

El modelo detallado anteriormente se complementa con una taxonomía de contexto de uso adicional [18] que sigue los mismos principios. El contexto de uso se usa para identificar

los aspectos más relevantes en un estudio de usabilidad.

Los atributos de primer nivel están formados por usuario, tarea y entorno que se refieren a: persona que interactúa directa o indirectamente con el sistema, labor de trabajo que el usuario realiza al interactuar con el sistema y factores externos que afectan al uso del sistema, respectivamente. Estos a su vez se subdividen de nuevo, como se muestra en la Figura 2.2.

Figura 2.2: Primeros niveles de la taxonomía de contexto de uso

2.2 Campo de aplicación: APIs RESTful

La mayoría de empresas y organizaciones integran una API REST en su negocio para ofrecer sus servicios o productos. Este tipo de APIs cada vez se hacen más relevantes en la actualidad, pero es muy escasa la información formal publicada sobre como implementarla correctamente. Esto es importante destacarlo ya que, en contraste con las APIs tradicionales, una API REST es independiente del lenguaje de implementación y su diseño no está estandarizado.

Esto hizo que profesionales del campo propusieran buenas prácticas de diseño y que con ellas se cumplieran completamente los principios del estilo, dando lugar a APIs RESTful.

Algunos de los autores que nos han permitido adoptar una perspectiva teórica de los principios fundamentales para el diseño e implementación de APIs REST ha sido Amodeo

[19] que propone varias preferencias de diseño para implementar operaciones CRUD sobre datos y el uso de hipermedia en una API REST, profundizando también en cómo diseñar el URI para acceso a recursos dependiendo el volumen de datos o cómo definir vistas o consultas predefinidas para dar un acceso a datos más rápido, entre otras.

El enfoque que ha seguido Massé [20] ha sido el de plantear reglas sobre principios de diseño. Define un número de reglas para cada elemento involucrado en una API REST. Estas reglas van desde el diseño de URIs y sus parámetros de consulta hasta principios de diseño para abordar las preocupaciones comunes de los clientes, como puede ser la de proporcionar una autorización de acceso a recursos de manera segura mediante el estándar OAuth, pasando por el diseño de metadatos dentro de las cabeceras, las interacciones con HTTP y las representaciones de recursos y errores en las respuestas.

Por otro lado, Hradil y Sklenakz [21], abordan de manera general los mismos aspectos que el trabajo anterior pero con la peculiaridad de que dedica varias reglas al diseño de la documentación de la API.

Otra corriente que siguen los estudios que tratan sobre guías de implementación es la de detectar un conjunto de patrones y antipatrones de diseño. Palma et al. [22][23][24] definieron un proceso de detección de patrones y antipatrones en APIs RESTful. Entre los patrones propuestos encontramos algunos como: usar jerarquías entre nodos en el URI o evitar URIs desordenados y difíciles de leer. Algunos de los antipatrones que propusieron son: olvido de hipermedia como parte de la representación de un recurso o ignorar el almacenamiento en caché de la respuesta a una solicitud, entre otros.

Por lo tanto, lo que se ofrece en este trabajo es una manera de cómo cubrir todos los aspectos de la usabilidad en las APIs REST a partir de los principios de diseño del estilo recopilados mediante bibliografía publicada, como es la que hemos explicado anteriormente, ya que no existe ningún trabajo que trate explícitamente de la usabilidad en este tipo de APIs. Esto ha sido una gran motivación para centrarnos en el tema y dar mayor trascendencia al trabajo.

Aun así, destacamos trabajos como el de Murphy et al. [25], donde analizan guías de diseño de compañías altamente conocidas comparando y contrastando cada una de ellas para dar por sentado la calidad y usabilidad, o el de Yamamoto et al. [26], en el que proponen un modelo de calidad para ceñirse al concepto de capacidad de aprendizaje y estabilidad al cambio, desde el punto de vista del usuario, y posteriormente validarlo mediante un estudio empírico sobre la usabilidad en varias APIs.

Capítulo 3
Metodología

Nuestra metodología se basa en el desarrollo de un conjunto de heurísticas de manera sistemática y generalizable, dentro de un modelo expandido de usabilidad para organizar la información y transformarla en heurísticas y guías, que posteriormente serán aplicadas a un caso práctico para estudiar su usabilidad mediante una evaluación heurística, de manera que también se valide este conjunto.

La generalización de nuestra metodología significa que puede ser aplicada a dominios muy diferentes, algo que es realmente útil en dominios donde casi no existe información publicada, como es el caso del estudio de usabilidad para APIs RESTful.

Distinguimos por lo tanto, dos grandes tareas independientes:

- **Desarrollo de heurísticas y guías de usabilidad**, que pueden ser reutilizables, de tal forma que se puedan aplicar en otros muchos contextos, como puede ser en las fases tempranas del diseño. Para este procedimiento, nos hemos apoyado en la metodología expuesta por el trabajo de Alonso-Ríos et al.[27], donde crean un conjunto de heurísticas y guías para APIs generales dentro del mismo modelo de usabilidad utilizado en este trabajo.

  La metodología consiste generalmente en **caracterizar el sistema** para identificar todos los elementos relevantes sobre las tareas del usuario y del sistema, hacer un **análisis exhaustivo de la bibliografía**, **mapearlo con la taxonomía de usabilidad** y **transformar la información en heurísticas y guías**.

- **Evaluación heurística en un caso práctico**, donde se aplican las heurísticas creadas en la tarea anterior para hacer una evaluación heurística a un caso particular. Cabe destacar que, de manera similar al desarrollo de heurísticas y guías, nos hemos apoyado en la metodología seguida en otro estudio de Alonso-Ríos et al.[9], donde se **caracteriza el contexto de uso** para atender qué elementos son relevantes para cubrir la usabilidad dentro de un caso específico y finalmente estudiarla mediante **evaluación heurística**

a partir del mismo modelo de usabilidad mencionado anteriormente.

Cada proceso seguido en la metodología, será detallado a continuación.

3.1 Desarrollo de heurísticas y guías de usabilidad

3.1.1 Caracterización del sistema

Para analizar y abstraer las características de las APIs RESTful, procedemos a diseñar una taxonomía compuesta por los elementos más relevantes, es decir, los que están involucrados en una solicitud y respuesta HTTP, y que por lo tanto, es la parte principal de una API REST (Figura 5.2). Es necesaria para tener cada característica específica del sistema claramente diferenciada y así, detectar qué elementos del sistema son significativos para el estudio de usabilidad. El procedimiento que hemos seguido para definirla, ha sido el de ir dividiendo cada elemento en varios niveles hasta llegar a un nivel de profundidad idóneo para mapearlo con la taxonomía de usabilidad.

De esta manera, podemos conectarlo fácilmente con el análisis de requisitos y así, integrar la usabilidad en el proceso de desarrollo del software. Cada elemento que pertenece a la solicitud y a la respuesta dentro de la taxonomía, está definido en el estándar W3C [28], dentro de los cuales, podemos diferenciar varias partes que forman una solicitud, como es la línea de inicio, compuesta por un método de solicitud y un URI. Como veremos en los puntos siguientes, dentro de nuestro estudio de usabilidad, los elementos y estructura que forman el URI juegan un papel importante, debido a que la línea de inicio proporciona toda la información relevante con respecto a la acción y su configuración correspondiente, ordenada por el usuario. Estos, coexisten con la cabecera y cuerpo del mensaje, necesarios para obtener información adicional con respecto a la solicitud y definir un recurso auto-descriptivo para almacenar o modificar un conjunto de datos, respectivamente.

De manera similar, una respuesta está formada por una cabecera y un cuerpo del mensaje, con la peculiaridad de que este último podría estar formado por una descripción detallada de los problemas surgidos al lanzar una solicitud. De igual forma, lo que caracteriza una respuesta es su código, determinado por la línea de estado.

Finalmente, aparte de todo lo dicho anteriormente, dos aspectos importantes cara al usuario, son la documentación y versión de la API.

3.1.2 Creación y organización de heurísticas y guías

En primer lugar, procedemos a eliminar cada rama de la taxonomía de usabilidad que no es relevante para nuestro caso de estudio. Por ejemplo, varios atributos de la taxonomía se dividen acorde con los cinco sentidos, como es el caso de la claridad. En este caso, nos

centraríamos en elementos que son claramente visuales. Otro ejemplo, sería la satisfacción subjetiva, que se divide en estética e interés. Este último también es eliminado, ya que el interés del usuario no depende del sistema, sino de sus propias necesidades.

En segundo lugar, realizamos un estudio exhaustivo de la bibliografía, comentada en el capítulo anterior, donde se plantea fundamentalmente principios de diseño, buenas prácticas, detección de patrones y antipatrones de APIs RESTful. Para organizar dicha información, mapeamos cada atributo de usabilidad con un elemento del sistema (Figura 5.2), y la clasificamos donde corresponda, de manera que se tenga en cuenta todos los aspectos de usabilidad para cada uno de ellos. Por ejemplo, una regla que se repite mucho en la literatura de APIs REST es la de no incluir la extensión del archivo o formato como parte del URI, por lo que lo clasificamos dentro del atributo que es resultado del mapeo entre ambas taxonomías (Figuras 2.1 y 5.2): *"Cognoscibilidad - Claridad formal - URI"*, y así hasta abordar toda la taxonomía del sistema.

El paso siguiente consiste en convertir la información organizada en heurísticas y guías. Acorde con las terminología de Nielsen, una heurística es definida como una regla general, mientras que, una guía puede definir aspectos específicos hasta principios generales. Por ejemplo, si tenemos varias guías de usabilidad clasificadas dentro de *"Cognoscibilidad - Claridad formal - URI"* y que consisten en: *"usar guiones, no usar guiones bajos, no usar mayúsculas, no usar unicode, no incluir la extensión del archivo o del formato"* estas pueden resumirse en una heurística más general: *"Las URI deben ser claramente visibles"*. Finalmente, si algún atributo de usabilidad no ha sido abordado por la bibliografía, abstraemos información de literatura relacionada para crear sus heurísticas y guías correspondientes.

3.2 Evaluación heurística en un caso práctico

Para validar el desarrollo anterior, realizamos una evaluación heurística a un caso práctico, teniendo en cuenta el contexto de uso para interpretar los resultados de usabilidad.

3.2.1 Caracterización del contexto de uso y evaluación heurística

Hacer un análisis del contexto de uso antes de realizar la evaluación heurística es necesario para atender qué aspectos de los usuarios, sistema y entorno son relevantes para cubrir la usabilidad dentro de un caso específico. Para ello, seguimos la taxonomía de los atributos que definen un contexto de uso (Figura 2.2), centrándonos, por ejemplo, en cualidades del usuario, como la experiencia en sistemas similares y conocimiento del dominio, sus características físicas o las demandas cognitivas para completar con éxito una tarea, entre otros. Identificando los aspectos que pueden dar problemas de usabilidad dentro del contexto de uso de nuestro caso específico, podremos situar los resultados de la evaluación heurística dentro del contexto

e interpretarlos con mayor precisión.

El objetivo de la evaluación heurística es comprobar que los elementos del caso práctico y su contexto de uso satisfacen los atributos de usabilidad. Para esto, aplicamos las heurísticas con sus respectivas guías de usabilidad, de manera que para cada una obtenemos un resultado. Por ejemplo, para la heurística mencionada anteriormente, *"Las URI deben ser claramente visibles"*, la cual implica que se cumpla las siguientes guías de usabilidad, *"usar guiones, no usar guiones bajos, no usar mayúsculas, no usar unicode, no incluir la extensión del archivo o del formato"*. La URI podría cumplir parcialmente, totalmente o escasamente la heurística aplicada.

Por último, considerando las características del contexto de uso y los resultados anteriores, procedemos a sacar las conclusiones del estudio de usabilidad, de tal forma que, para el caso anterior, si el URI no respeta cada guía que conlleva la heurística aplicada, podríamos decir que habría problemas de usabilidad en torno a las demandas cognitivas para construir una solicitud o en la estabilidad visual del entorno.

Capítulo 4
Planificación

Para hacer un seguimiento de cada tarea y visualizar el flujo de trabajo del proyecto, utilizamos Kanban, una de las metodologías ágiles más conocidas. Consideramos que era la más adecuada para un proyecto de investigación, ya que no hay intervalos de tiempo ni iteraciones definidas.

Dentro de esta metodología no existen puntos de inicio o final definidos para cada elemento de trabajo, de manera que todos son independientes entre sí sin una duración de tiempo determinada.

La dinámica para la planificación y el control de las tareas es la de ir asignando cada una bajo un estado, normalmente definidos como: *por hacer*, *haciendo* y *hecho*.

Al controlar qué tareas están activas en un momento dado, nos permite ver el progreso del proyecto y detectar los cuello de botella mientras que nos acercamos al final del proyecto de forma incremental.

Para llevar a cabo la planificación, nos hemos apoyado en *Microsoft Planner*, una herramienta de Office 365 que permite una gestión completa y ordenada de tareas.

El proyecto está dividido en varias fases que definen cada proceso importante de su desarrollo, formadas por un conjunto de tareas:

- **Análisis bibliográfico sobre la usabilidad de RESTful APIs**
  - Análisis de libros.
  - Análisis de revistas.
  - Análisis de webs.
  - Obtención de los resultados del análisis.

- **Desarrollo de heurísticas y guías de usabilidad de RESTful APIs dentro de un modelo expandido de usabilidad**
  - Diseño de la taxonomía de elementos de una solicitud y respuesta HTTP.
  - Análisis de las taxonomías de usabilidad y contexto de uso.
  - Desarrollo de guías para cada atributo dentro de la taxonomía de usabilidad con ejemplos y contraejemplos.
  - Revisiones.
- **Estudio de cómo el contexto de uso puede afectar a la usabilidad**
  - Estudio del contexto de uso.
- **Prueba de las heurísticas y guías sobre un caso práctico**
  - Estudio de potenciales candidatos al caso práctico.
  - Selección del API objetivo.
  - Aplicación de heurísticas en el API objetivo.
  - Análisis de resultados.
  - Aplicación del contexto de uso.
  - Propuesta de mejoras.

También, hemos definido varias tareas para la gestión y actividades burocráticas del trabajo y su publicación en un congreso.

- **Gestión del trabajo**
  - Reuniones con los tutores.
  - Solicitud de prórroga.
  - Entrega de la memoria.
- **Elaboración de la memoria**
  - Definir la estructura de la memoria y redacción de cada capítulo.
- **Publicaciones**
  - Publicación en un congreso.

En las figuras 4.1, 4.2, 4.3 y 4.4 se muestra el plan de tareas en distintos instantes de tiempo.

En los apartados siguientes se describe los recursos, tiempo y costes invertidos en este trabajo.

4.1 Recursos

Sobre los recursos humanos contamos con tres investigadores: una alumna y dos directores del trabajo. Con respecto a los materiales necesarios, incluimos el ordenador portátil de la alumna y los que han proporcionado los directores, como libros y artículos necesarios para la realización del trabajo.

4.2 Tiempo

El tiempo dedicado al trabajo ha sido un total de **275 horas**, pero no de manera constante, ya que al inicio del trabajo, la alumna que ha realizado el proyecto estaba cursando varias asignaturas y trabajando al mismo tiempo. A partir de la mitad del desarrollo del proyecto, se pudo aumentar el tiempo de dedicación.

4.3 Costes

Para estimar los costes, se tiene en cuenta los recursos humanos y materiales del trabajo. Según Indeed, la media salarial para un investigador en España es de 23.640€/año [29], que son unos 1970€ al mes, y 12,31€/hora. Como el tiempo necesario para la elaboración del trabajo ha sido un total de 275 horas, el coste por investigador es de 3385,25€. En la tabla 4.1 se muestra el coste final del trabajo.

Recursos	**Coste**
Investigador	3385,25€
Ordenador portátil	600,00€
Total	**3985,25€**

Tabla 4.1: Estimación de costes del trabajo

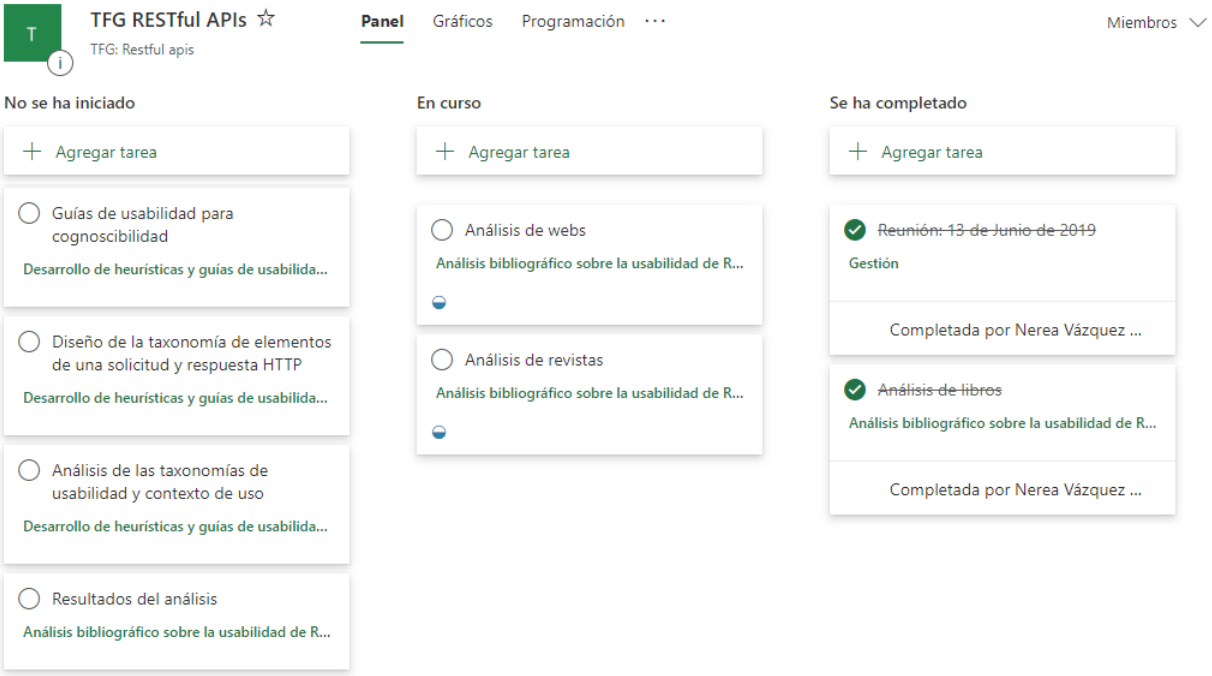

Figura 4.1: Plan de tareas del trabajo el 10 de agosto de 2019

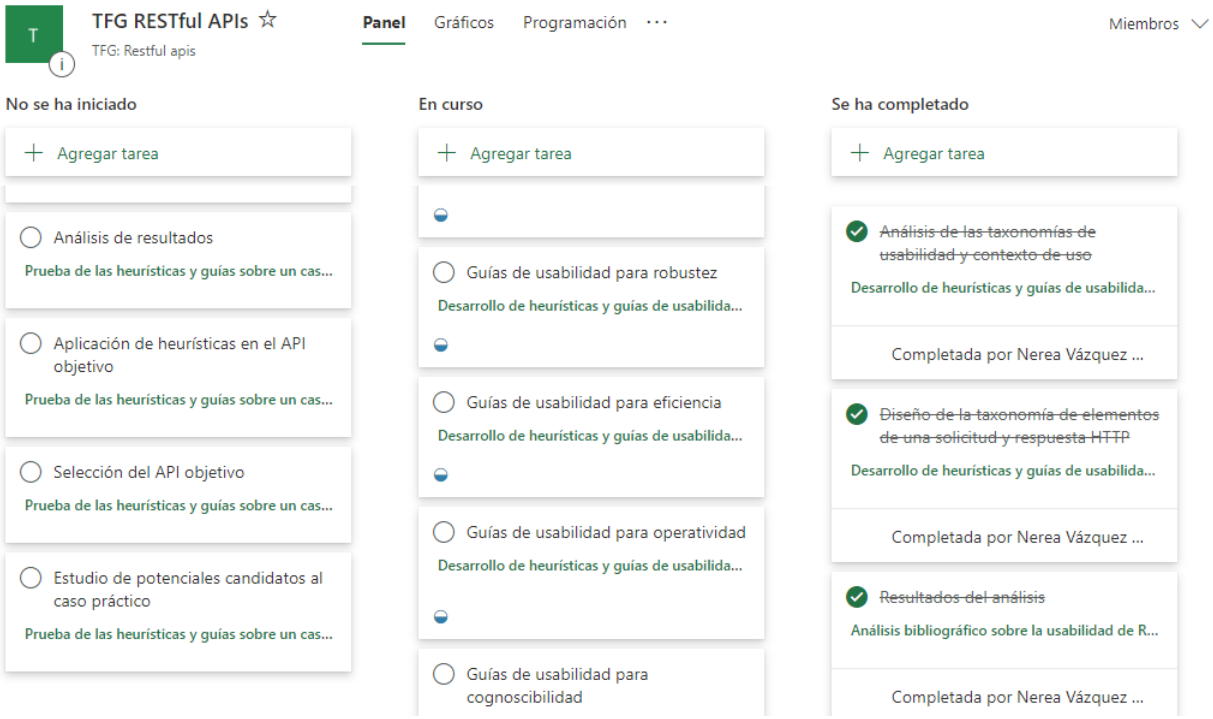

Figura 4.2: Plan de tareas del trabajo el 26 de octubre de 2019

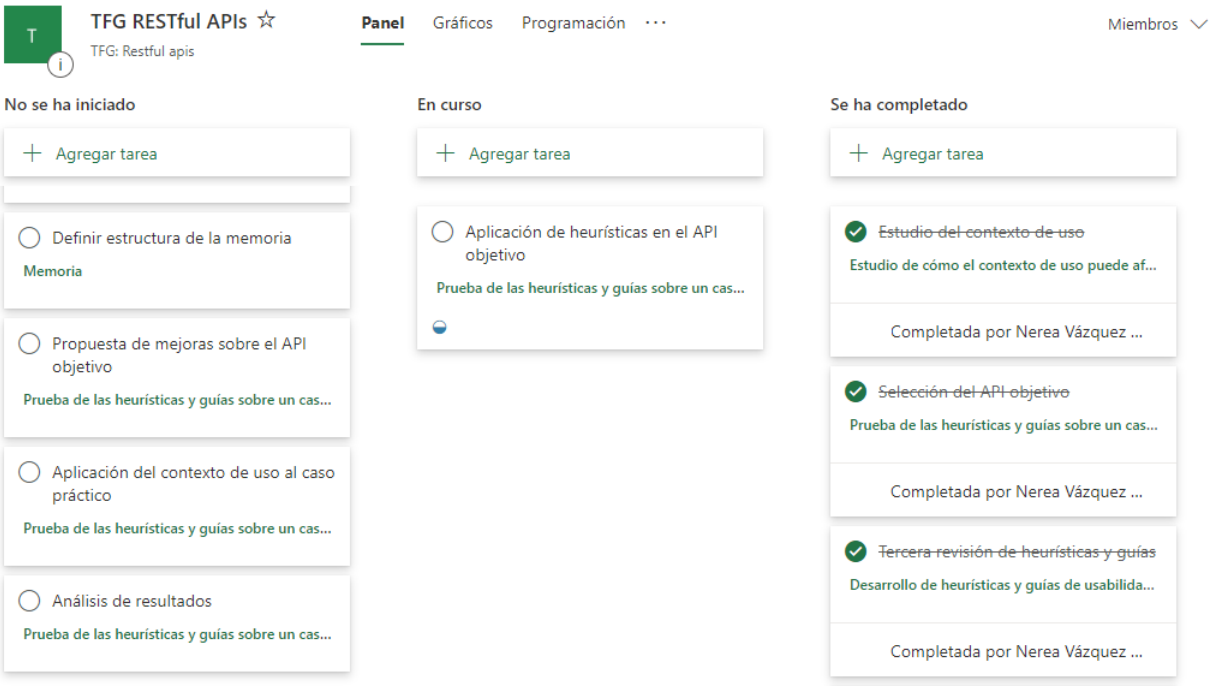

Figura 4.3: Plan de tareas del trabajo el 3 de mayo de 2020

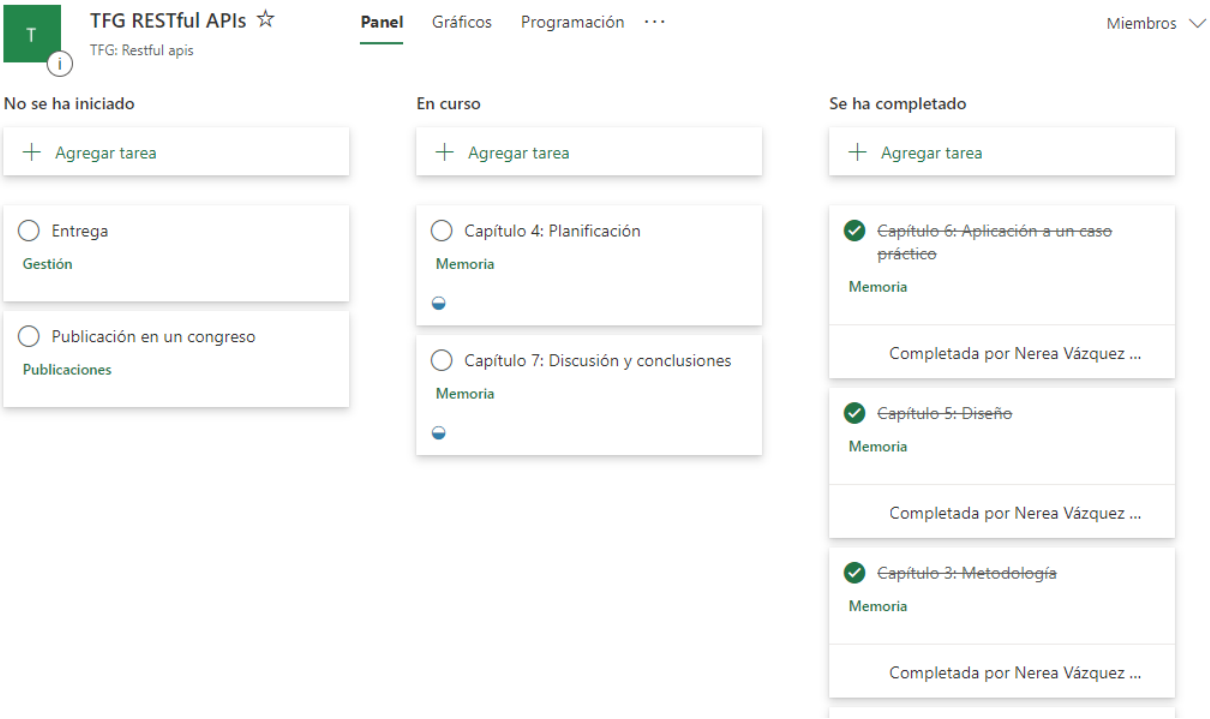

Figura 4.4: Plan de tareas del trabajo el 6 de julio de 2020

Capítulo 5

Diseño de heurísticas y guías de usabilidad

En este capítulo explicaremos cómo hemos aplicado la metodología para el desarrollo del conjunto de heurísticas y guías de usabilidad propuestas en este trabajo.

5.1 Caracterización del sistema

Un servidor web recibe las necesidades de un sitio o cualquier otra aplicación. Los clientes utilizan interfaces de programación de aplicaciones para comunicarse con los servicios web. Generalmente, una API expone varias acciones sobre un conjunto de datos, para facilitar el intercambio de información entre las interacciones cliente-servidor. Como se muestra en la Figura 5.1, una API web es la cara de un servicio web, escucha y responde directamente a las solicitudes de los clientes. El estilo de arquitectura REST, se aplica comúnmente al diseño de APIs para servicios web modernos. En teoría, REST no está vinculado a ningún protocolo de comunicación, pero siempre se implementa sobre HTTP, ya que su autor lo definió pensando en este protocolo, teniendo en cuenta sus puntos fuertes. Una API REST puede ser utilizada por prácticamente cualquier lenguaje de programación, haciendo que el cliente y el servidor sean independientes entre sí. Seguidamente, detallaremos cada parte que determina el comportamiento de una API REST.

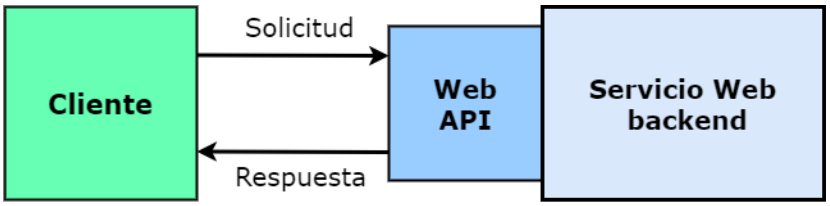

Figura 5.1: Web API

5.1.1 Solicitud HTTP/1.1

Las solicitudes HTTP son mensajes enviados por un cliente para iniciar una acción en el servidor. Dado el estándar W3C [28], está formada por una línea de inicio, campos de cabeceras de solicitud y un cuerpo del mensaje :

- **Su línea de inicio** está compuesta por un método HTTP que describe la acción a realizar sobre el identificador uniforme de recursos. Los métodos más usados son los siguientes:
  - GET: se utiliza para solicitar una representación de un recurso específico.
  - PUT: se utiliza para actualizar completamente un recurso existente.
  - POST: se utiliza para crear un recurso nuevo causando un cambio en el estado o efectos secundarios en el servidor.
  - DELETE: se utiliza para eliminar un recurso ya creado.
  - HEAD: este método es idéntico a GET, excepto que el servidor no envía el cuerpo del mensaje en la respuesta.
  - OPTIONS: se utiliza para describir las opciones de comunicación para un recurso en concreto.

  Los métodos HTTP pueden clasificarse en métodos **seguros de sólo lectura**, que no alteran el estado del servidor, como lo son GET, HEAD y OPTIONS, **almacenables en caché**, que almacenan las respuestas para reutilizarlas, como el caso de GET y POST, e **idempotentes**, si los resultados en el servidor son los mismos al ejecutar una o varias veces dicho método como los son PUT, DELETE y los métodos de solicitud seguros que hemos mencionado.

  Las opciones para integrar el identificador de recursos (URI) dependen de la naturaleza de la solicitud. La forma más común es utilizarlo para identificar un recurso en un servidor o puerta de enlace de origen. En este caso, la ruta absoluta del URI debe transmitirse como el URI de solicitud, y la ubicación de red del URI (autoridad), transmitirse en el campo host de la cabecera.

  Finalmente, se completa con la versión de HTTP, la cual define la estructura de los mensajes, indicando la versión que espera que se use para la respuesta.

- **Los campos de la cabecera** de solicitud permiten que el cliente pase información adicional sobre ella, y sobre el mismo cliente al servidor. Estos campos actúan como modificadores de solicitud, con semántica equivalente a los parámetros en una invocación de método en un lenguaje de programación. Estas pueden clasificarse en varios grupos:

  - Cabeceras generales, como *Date o Cache-Control*, que afectan al mensaje como una unidad completa.
  - Cabeceras de petición, como *User-Agent* o *Accept*, que modifican la petición especificándola con mayor detalle, o restringiéndola condicionalmente, como *If-None-Match*.
  - Cabeceras de entidad, que contienen información sobre el cuerpo de la entidad, como el tamaño del contenido o su tipo MIME, como puede ser *Content-Type*.

- **El cuerpo del mensaje** se utiliza para transportar el cuerpo de la entidad asociado con la solicitud. Las peticiones que reclaman datos, como GET, HEAD o OPTIONS, normalmente, no necesitan ningún cuerpo. Algunas peticiones pueden mandar peticiones al servidor con el fin de actualizarlo, como es el caso del método PUT o POST.

A continuación se muestra un ejemplo de una solicitud con cada una de sus partes mencionadas.

```
POST /v1/team/players HTTP/1.1      <-- Línea de inicio
Host: api.example.com               <-- Cabeceras
Accept: application/json,application/xml
Content-Type: application/json


{                                   <-- Cuerpo del mensaje
    "player": {
        "name": "Osvaldo",
        "surname": "Fernández",
        "age": 24
    }
}
```

5.1.2 Respuesta HTTP/1.1

Después de recibir e interpretar un mensaje de solicitud, un servidor responde con un mensaje de respuesta HTTP. Dado el estándar W3C [28], está formada por una línea de estado, campos de cabeceras de respuesta, y un cuerpo del mensaje:

- **Su línea de estado** está compuesta por la versión del protocolo, seguido de un código de estado, formado por un entero de 3 dígitos indicando el éxito o fallo de la petición, acompañado de una descripción breve de lo que significa. El primer dígito del código define la clase de respuesta:

  - 1xx: Informativo - Solicitud recibida, proceso continuo.
  - 2xx: Éxito - la acción se recibió, entendió y aceptó con éxito.
  - 3xx: Redirección - Se deben tomar medidas adicionales para completar la solicitud.
  - 4xx: Error del cliente - la solicitud contiene una sintaxis incorrecta o no se puede realizar.
  - 5xx: Error del servidor - El servidor no pudo cumplir una solicitud aparentemente válida.

- **Los campos de la cabecera** de respuesta permiten que el servidor pase información adicional sobre la respuesta que no se puede colocar en la línea de estado. Estos campos proporcionan información sobre el servidor y sobre el acceso adicional al recurso identificado por el URI de la solicitud.

  Las cabeceras para respuestas se clasifican de manera similar que las de una solicitud. Algunos ejemplos de campos de cabeceras específicos para una respuesta son *ETag*, que específica la versión actual de un recurso o *Location*, que se utiliza para redirigir al cliente a una ubicación que no sea el URI de solicitud para completar la solicitud o identificar un recurso nuevo.

- **El cuerpo del mensaje** es semejante al de una solicitud. Las respuestas con un código de estado como 201 o 204 normalmente prescinden de él.

A continuación se muestra un ejemplo de una respuesta con cada una de sus partes.

```
HTTP/1.1 201 Created            <-- Línea de estado
Location: /v1/team/players/4335 <-- Cabeceras
Content-Type:application/json
Date: Thu, 30 Oct 2019 01:18:09 GMT
Content-Length: 72


{                               <-- Cuerpo del mensaje
  "data": {
    "id": 4335,
    "message": "Create successfully"
  }
}
```

La taxonomía de los elementos del sistema (Figura 5.2) ha sido creada a partir de cada elemento que hemos explicado anteriormente, y que consideramos que debería de estar in-

volucrado para garantizar la usabilidad en este tipo de sistemas. En los apartados siguientes, describiremos cómo ha intervenido en el desarrollo de heurísticas.

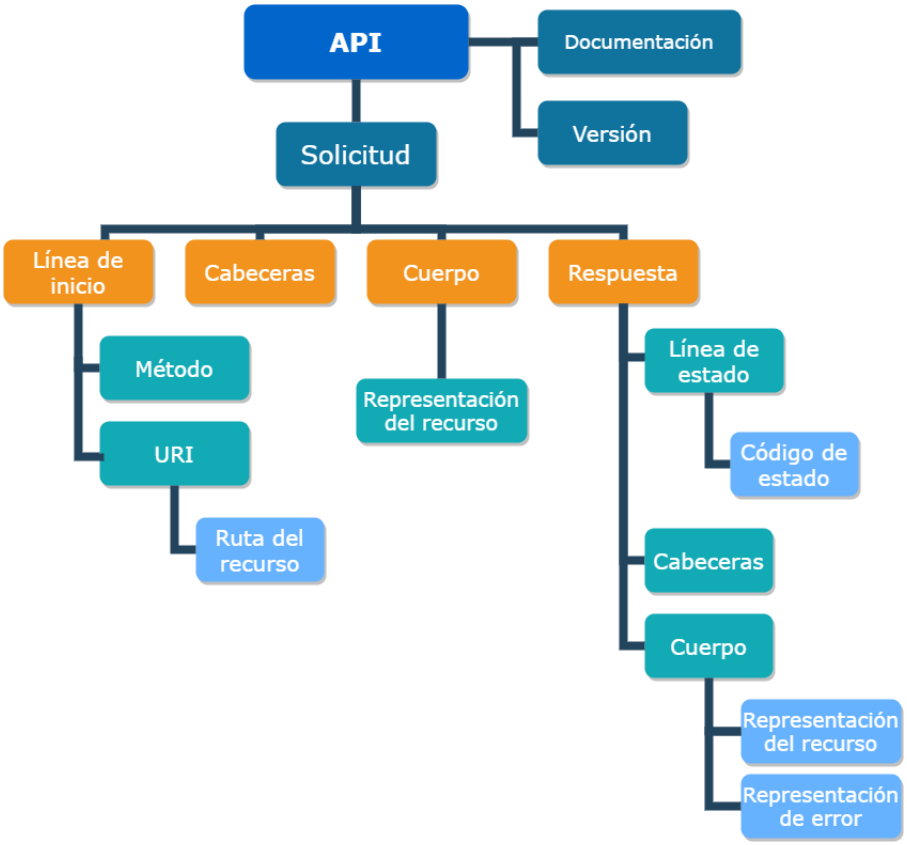

Figura 5.2: Taxonomía de elementos del caso de estudio

5.2 Análisis bibliográfico sobre RESTful APIs

Previamente al estudio bibliográfico, eliminamos las ramas de la taxonomía de usabilidad [3] que consideramos que no son relevantes o que no encajan en nuestro caso, tal y como mencionamos en la metodología del capítulo anterior.

Estos atributos contemplan la percepción de los elementos mediante los sentidos, enfocándonos solamente en los elementos que son consistentes y claramente visuales. Otros que a priori no nos han parecido necesarios, son la interactividad de la asistencia, es decir, la asistencia que proporciona el sistema para responder a las acciones del usuario. Este tipo de APIs se basan en el intercambio de mensajes, y la única información útil para que los usuarios averigüen cómo utilizarla es su documentación o las descripciones de error dentro del cuerpo del

mensaje de la respuesta. El usuario no demanda ayuda explícita al sistema. La controlabilidad del flujo de trabajo también es eliminada. El cliente y el servidor actúan como entidades separadas, realizando actividades o tareas independientes, por lo cual, no podemos hacer un seguimiento de cómo el servidor ha procesado la solicitud. La única manera es enviar solicitudes adicionales para ver el estado de un recurso o esperar la respuesta de la solicitud, pero esto está bajo la responsabilidad del usuario y no del sistema. Por último, prescindimos de la parte que se centra en captar y mantener la atención y curiosidad intelectual del usuario dentro de la satisfacción subjetiva, ya que el interés de este, no depende del propio sistema, sino en qué medida puede el sistema satisfacer las necesidades del usuario.

Seguidamente, nos disponemos a hacer un estudio en profundidad sobre la literatura de APIs REST en artículos publicados, libros, críticas sobre el tema, blogs en la web, etc. Algunos de los recursos utilizados han sido bases de datos como *IEEE Xplore, Scopus, ACM Digital Library* y buscadores como Google académico.

Generalmente, la bibliografía trata sobre principios y reglas de diseño, buenas practicas y recomendaciones. Cabe destacar que también existe aquella que detalla la implementación de un servicio web REST, pero en estos casos, resulta difícil abstraer guías de diseño para la interfaz, ya que no es el tema principal y aparece mencionado de forma implícita.

Para tomar decisiones entre qué guías incluir dentro del conjunto de heurísticas, a veces nos resultó muy útil consultar varios blogs, como ha sido el de Roy T. Fielding [30], que aclara dudas respecto a los principios REST que definió y valora cómo los desarrolladores usan este estilo de arquitectura en diferentes proyectos y estudios publicados.

Después del análisis, llegamos a la conclusión de que la información es muy limitada y poco organizada.

Muchos autores proponen principios REST diferentes para llegar al mismo objetivo, e incluso a veces es necesario completar la información apoyándose en otro estudio diferente, y los trabajos que comprenden todo el dominio, son bastante extensos y complejos, de modo que requiere de una experiencia considerable dentro de los servicios web y del diseño de protocolos de red.

En el capítulo 2, mencionamos que no existe bibliografía explícita sobre la usabilidad de este tipo de APIs, por lo cual, consideramos oportuno comenzar a clasificar la información de una fuente extensa, que abarcara todos los elementos importantes del lado del cliente y servidor, dentro del mapeo entre la taxonomía de usabilidad y elementos del sistema, y así garantizar la usabilidad mediante los fundamentos REST.

5.3 Mapeo entre la taxonomía de usabilidad y la literatura de APIs RESTful

5.3.1 Primera clasificación de la información

Comenzamos a examinar y a clasificar cada guía de diseño propuesta en el trabajo de Massé [20], *REST API Design Rulebook*. Este autor, compuso una lista amplia de reglas para abarcar el diseño de identificadores de recursos, interacciones con HTTP, metadatos proporcionados mediante cabeceras, representaciones de recursos dentro del cuerpo del mensaje y soluciones para lidiar las preocupaciones del cliente. Destacamos algunas de ellas, como las centradas en el formato general del URI: *los guiones bajos no deben usarse en URI, se deben preferir las letras minúsculas en las rutas URI* y *las extensiones de archivo no deben incluirse en los URI*, que implican cómo diseñar URIs limpios de manera que un cliente pueda identificar los recursos de manera precisa y mejorar su legibilidad, de forma que las insertamos dentro de *Claridad - Claridad formal de los elementos - URI*.

Otra como, *se debe usar una forma consistente para representar errores y se debe usar una forma consistente para representar respuestas de error*, apoyadas con ejemplos, indica que una manera consistente de representar errores en el cuerpo del mensaje, es devolver una descripción en modo texto de la causa del error junto a un ID interno, por lo que estaríamos frente aspectos de *Asistencia - Idoneidad del contenido de la documentación - Representación de errores*.

En *Flexibilidad - Controlabilidad - Configurabilidad - Configurabilidad de aspectos técnicos - Representación del recurso*, introducimos las reglas del tipo: *la negociación del tipo de medio debe admitirse cuando hay varias representaciones disponibles* y *los componentes de consulta de un URI debe usarse para admitir respuestas parciales*, que permite al usuario personalizar el sistema de la forma que considere apropiada, en este caso, para ahorrar ancho de banda y acelerar la interacción cliente-servidor, recortando los datos de respuesta con el parámetro de consulta y permitiendo que los clientes negocien un formato mediante el envío de una cabecera *Accept* con el formato deseado.

Un último ejemplo sería las guías enfocadas en proteger los recursos confidenciales y a restringir las operaciones sobre los datos, como: *OAuth puede usarse para proteger recursos* y *las soluciones de administración de la API se pueden usar para proteger los recursos*, indicándonos así, que el estándar OAuth proporciona una autorización segura para los clientes y que las soluciones de administración de la API, como usar un proxy inverso, esconden características del servidor de origen y analiza cada solicitud entrante. Por lo que estaríamos dentro de *Seguridad del usuario / Seguridad de terceros - API*.

Estas guías también cubren aspectos de cómo usar los métodos HTTP dentro de una solicitud o qué cabeceras utilizar para disminuir la latencia de respuesta, que hemos repartido

entre los atributos de usabilidad y sistema: *Precisión - Métodos* y *Eficiencia respecto a tiempo de ejecución de tareas - Respuesta del sistema - Respuesta a una solicitud*, respectivamente.

El resultado de esta primera aproximación es una base sólida y consistente de guías, que nos permite enfocar correctamente la distribución futura de pautas a lo largo de las taxonomías, por lo que en seguida, procedemos a abstraer más guías de la bibliografía, de modo que apoyemos las que han sido previamente vistas y completar todos los atributos de usabilidad.

5.3.2 Segunda clasificación de la información

Analizamos los principios de diseño, patrones y antipatrones propuestos por varios trabajos comentados en el capítulo 2 [21][22][23][24], y seguimos el mismo proceso detallado anteriormente. Muchas de estas apoyan las guías anteriores, o incluso se crean nuevas con respecto a la documentación de la API, como la de insistir que esta debe de ir acompañada de una documentación con ejemplos de uso en diferentes lenguajes de programación, apoyado también por un estudio de investigación de Sohan et al.[31], donde descubrieron que los desarrolladores de clientes de API REST enfrentan problemas de productividad cuando la documentación carece de ejemplos de uso. El enfoque de otros estudios que nos han servido para completar el desarrollo de guías, son los que explican cómo implementar un servidor REST, mostrando aspectos de más bajo nivel. Entre estos, podemos destacar los trabajos de Allamaraju [32], Giessler et al.[33] o Wilde y Pautasso [34], entre otros. De estos, abstrajimos recomendaciones como el de usar tipos MIME consistentes para representar los cuerpos del mensaje como lo son XML, JSON o Multipart/Related, este último en caso de usar un objeto compuesto en el cuerpo del mensaje, especificar las versiones de la API en la ruta URI del recurso, o cómo estructurar el URI cuando haya relaciones jerárquicas entre recursos.

También, tuvimos en cuenta fundamentos REST que siguieron varios autores como Rodríguez et al.[35], para proponer una solución basada en una API REST o analizar un contexto específico para estudiar si cumple los requisitos del estilo. Algunas de estos, establecen cómo nombrar los recursos, fomentar la negociación de formatos en la solicitud, administrar recursos mediante un conjunto uniforme de operaciones definidas por HTTP y como usar hipermedia como motor de estado.

5.4 Codificación de la información en forma de heurísticas y guías

Una vez que tuvimos todas las guías distribuidas a lo largo del mapeo entre la taxonomía de usabilidad y del sistema, observamos que todos los criterios y recomendaciones que fueron propuestos por varios autores podrían asignarse con éxito a un atributo, e incluso a varios.

Esto demuestra que la taxonomía es lo suficientemente completa como para abarcar toda la información de la literatura REST, a pesar de que esta es extremadamente especializada y la taxonomía de usabilidad es de propósito general.

5.4.1 Integración de guías

El siguiente paso es unificar las guías para aquellos atributos de usabilidad cubiertos por varios autores, o tomarlas directamente de la literatura, y resolver contradicciones.

Varios autores proponían semánticamente la misma recomendación para cubrir un principio REST, pero los ejemplos de uso eran diferentes, por lo que elegimos aquellos que más se repetían entre los estudios analizados, y el que consideramos que era el más completo y claro de comprender. Un ejemplo ha sido el de incluir links dentro del cuerpo del mensaje para llevar al cliente a recursos y acciones relacionados con la petición y para cumplir una de los características más importantes del estilo REST, que es el principio de HATEOAS. Finalmente decidimos que estos deberían ir dentro de una sección llamada *links* dentro del cuerpo del mensaje, para que el cliente pudiera recuperarlos con facilidad. Otro, ha sido el de cómo construir una consulta parametrizada dentro del URI para ordenar los resultados. Muchos autores no consideraron importante añadir esta funcionalidad dentro de la API y los que sí, lo hacían de forma que usaban una lista de atributos separados por comas con *sort* como parámetro URI seguido de un signo + como prefijo para un orden ascendente o un signo - para un orden descendente. En varios blogs especializados, lo hacían de manera que el atributo del parámetro URI lo escribían de forma: *sort_by=asc(email)* o *sort_by=email&order_by=desc*, cosa que ignoramos porque nos parecía bastante engorroso para construir la consulta desde algún framework de programación o captar su significado.

Algunas de las discrepancias que nos surgieron fueron si se debe usar el método HTTP PATCH, que sirve para modificar parcialmente un recurso. La mayoría de los autores no consideraban este método para realizar dicha función dentro de los principios REST. Indagando sobre el tema, decidimos no incluirlo dentro del conjunto de pautas, ya que PATCH no es un método idempotente. Esto quiere decir que peticiones sucesivas idénticas pueden tener efectos diferentes, y al actualizar parcialmente unos campos específicos de un recurso, este depende completamente de que el cliente use correctamente dicho método, además que habría que hacer comprobaciones adicionales en las cabeceras de las respuestas y asegurarse de la presencia de *Accept-Patch*, el cual indica que PATCH está permitido en el recurso identificado por el URI de la solicitud. La ventaja es que se ahorraría ancho de banda al no enviar todo el recurso completo, pero insistimos en qué sólo se debería de usar PUT para este fin.

Otra ha sido sobre donde y cómo mostrar la versión actual de la API. Optamos por que forme parte de la ruta del recurso dentro del URI, ya que indica la versión actual del conjunto de recursos, y de esta manera, ahorraríamos muchas confusiones al cliente, aparte de que

especificarla dentro de un parámetro de consulta o a través de una cabecera personalizada en la petición y respuesta, conlleva el aumento del tráfico y coste computacional al tener que existir lógica para discriminar las distintas versiones soportadas.

Para los atributos que no han sido cubiertos por la literatura, o que han quedado un poco escasos de información, los completamos con las guías, que se han podido adaptar a este contexto, del trabajo que nos ha servido como inspiración para realizar el desarrollo de heurísticas y guías [9], y con información adicional fuera de la literatura REST. Por ejemplo, para completar las guías sobre seguridad, cogimos información de varios blogs que trataban de cómo evitar ataques de securización en APIs web. Cabe destacar que algunas guías han sido creadas sin estar apoyadas por ningún recurso bibliográfico, como en el caso de: *usar parámetros de filtrado, ordenación, selección de campos y paginación en la base del recurso del URI para que su estructura sea lo más clara y ordenada posible*, para mantener una claridad conceptual de la estructura del URI, entre otras.

5.4.2 Creación de heurísticas

Finalmente, para cada conjunto de guías, le asignamos una heurística que las resuma de manera general, de tal forma que, por ejemplo, para el conjunto de guías enfocadas en la eficiencia respecto a tiempo de ejecución de tareas, como es la de dar una respuesta a una solicitud, las resumimos de forma: *Se debe proporcionar metadatos específicos en la cabecera que ayuden a procesar y reducir la latencia de la respuesta a una solicitud*, así hasta completar cada atributo de usabilidad con sus guías correspondientes.

Cabe destacar que las heurísticas CCC-2 y CCF-2 han sido creadas sin estar apoyadas por ningún recurso bibliográfico, ya que hemos considerado que la estructura de la ruta del recurso debe ser fácil de comprender y de visualizar dado que pertenece al URI, uno de los elementos más relevantes para la usabilidad en este tipos de APIs.

5.5 Heurísticas y guías de usabilidad

En este apartado mostraremos el conjunto de heurísticas y guías de usabilidad desarrollado en este trabajo, acompañados de varios ejemplos y contraejemplos con el fin de enriquecer y aclarar cada una de ellas.

Cada heurística está identificada por un código, con el que podremos identificarla posteriormente en el grupo de tablas de ejemplos y contraejemplos.

Código	Heurísticas	Guías	Ref.
Cognoscibilidad			
Claridad - Claridad conceptual - URI			
CCC-1	Los elementos del URI se deben comprender con facilidad.	- No usar una barra diagonal al final. - Usar la barra diagonal para indicar una relación jerárquica entre recursos. - Usar punto y coma para indicar elementos no jerárquicos. - Usar *ampersand* para separar parámetros dentro del URI. - No usar abreviaturas.	[21][22] [35][32] [36]
CCC-2	La estructura base de la ruta de un recurso debe ser fácil de comprender.	- Usar parámetros de filtrado, ordenación, selección de campos o paginación en la base del recurso para que su estructura sea lo más clara y ordenada posible.	
Claridad - Claridad formal - URI			
CCF-1	Las URI deben ser claramente visibles.	- Usar guiones. - No usar guiones bajos. - No usar mayúsculas. - No usar unicode. - No incluir la extensión del archivo o del formato.	[32][35] [20][22]
CCF-2	La estructura base de la ruta de un recurso debe ser claramente visible.	- Si tenemos un recurso formado por muchos campos, se debe crear vistas o consultas predefinidas de los campos que el cliente pueda solicitar con mas frecuencia. Así, nos evitamos URIs muy largos. - No usar jerarquías de recursos muy grandes.	

Tabla 5.1: Cognoscibilidad parte I

Código	Heurísticas	Guías	Ref.
Cognoscibilidad			
Consistencia - Consistencia conceptual - Ruta del recurso			
CCC-3	Los términos usados en la ruta de un recurso, deben ser consistentes con el método HTTP de la solicitud.	- Usar un sustantivo singular para representar un recurso. - Usar sustantivos en plural para representar colecciones. - No usar términos CRUD. - Para solicitudes PUT y DELETE, el último nodo del URI de la solicitud debe ser singular. - Para las solicitudes POST, el último nodo debe ser plural. - Usar verbos o frases verbales para representar controladores sobre un recursos.	[22][36] [20][23] [33][35]
Consistencia - Consistencia conceptual - URI			
CCC-4	El URI debe tener una estructura consistente cuando identifique recursos relacionados jerárquicamente.	- Para especificar un recurso se debe usar dos URIs. La primera para determinar el conjunto de estados, es decir, una colección específica, y la segunda para seleccionar un estado dentro de esa colección. En el caso de que no sea necesario especificar una colección, bastará con especificar un recurso en concreto. Por lo tanto, cada recurso del URI tiene que estar relacionado jerárquicamente con los demás.	[33][22] [23]
Consistencia - Consistencia conceptual - Cuerpo del mensaje			
CCC-5	El cuerpo de la solicitud y respuesta debe estar representado por formatos consistentes.	- Utilizar tipos MIME consistentes: HTML, XML, JSON o Atom. - Los objetos JSON deben estar bien formados. - Si el cuerpo del mensaje está formado por un objeto compuesto, usar el tipo MIME *Multipart/Related*.	[34][20] [37][33] [21]

Tabla 5.2: Cognoscibilidad parte II

Código	Heurísticas	Guías	Ref.
Cognoscibilidad			
Consistencia - Consistencia formal - URI			
CCF-3	Se debe percibir con facilidad que el URI hace referencia a una API con su versión correspondiente.	- El URI debe identificar al propietario del servicio. - El URI debe tener un subdominio llamado api. - Si una API proporciona un portal para desarrolladores, el URI debe tener un subdominio etiquetado como desarrollador. - Especificar las versiones de la API en el URI del recurso, de manera VX, siendo X el número de la versión.	[20][37] [32][38] [39]
Memorabilidad - Memorabilidad formal y conceptual - URI			
CMF-1	Los términos y parámetros de consulta en el URI, deben ser fáciles de usar y de recordar. Los URI deben ser lo más explicativos posible.	- Para el filtrado, usar términos cortos como parámetros URI y que estén relacionados con el recurso. - Para la ordenación, usar el parámetro *sort*, seguido de una lista de atributos separados por comas con signo + como prefijo, para un orden ascendente, o un signo -, para un orden descendente. - Para la selección de campos, usar *fields* como parámetro seguido de una lista separada por comas de atributos. - Para la paginación, usar los parámetros *offset* y *limit*. - Si la complejidad de los requisitos de paginación o filtrado de un cliente excede las capacidades de formato simple de la parte de consulta dentro del URI, considere diseñar un recurso controlador especial que se asocie con una colección. - Usar términos descriptivos en los URIs.	[38][33] [20][40] [21]

Tabla 5.3: Cognoscibilidad parte III

Código	Heurísticas	Guías	Ref.
Cognoscibilidad			
Asistencia - Idoneidad del contenido de la documentación - Representación de errores			
CAI-1	La API debe mostrar información útil del error cuando hay un problema en la solicitud.	- El cuerpo del mensaje de la respuesta debe contener: código de estado, código interno, un mensaje para el desarrollador que describa la causa del error y ayuda para resolverlo, un mensaje para mostrar al usuario y un hipervínculo para obtener más información del problema. - No devolver código de éxito cuando se devuelve un error en el cuerpo de la respuesta. - No mostrar información obsoleta.	[20][37] [33][32]
Asistencia - Idoneidad del contenido de la documentación - Documentación de la API			
CAI-2	La documentación debe definir y describir cada elemento de la API y mostrar ejemplos de uso.	- Definir de manera estándar y reconocida los tipos MIME, códigos de estado y métodos HTTP. - Definir cada recurso con sus atributos y relaciones de enlace. - Definir detalladamente cada operación HTTP admitida para cada recurso. - Definir parámetros de consulta. - Definir cómo usar las credenciales de autenticación/autorización. - Mostrar ejemplos de solicitudes y respuestas. - Mostrar ejemplos de consultas parametrizadas en los URI. - Mostrar ejemplos implementados en los lenguajes más usados del dominio. - No mostrar información redundante ni obsoleta.	[37][33] [32][21]

Tabla 5.4: Cognoscibilidad parte IV

Código	Heurísticas	Guías	Ref.
Operatividad			
Completitud - Representación del recurso			
OC-1	La representación del recurso en la respuesta debe estar acompañado por un conjunto de links, formados por aquellos URIs que están relacionados con él.	- El recurso debe de estar acompañado por un conjunto de URIs que nos lleven a recursos relacionados con la petición para que el cliente pueda navegar por la información y esté lo más desacoplado posible del servidor, es decir, la API debe cumplir con el principio de HATEOAS. -El conjunto de URIs debe de formar parte de la representación del recurso y estar dentro de una sección llamada *links*.	[24][35] [20][41] [21][30]
Precisión - Método			
OP-1	Se debe utilizar correctamente los métodos HTTP en una solicitud.	- El método GET debe usarse para recuperar un recurso. - El método PUT debe utilizarse para insertar un recurso nuevo o actualizar/reemplazar recursos mutables ya creados. - El método POST debe utilizarse para crear un nuevo recurso dentro de una colección. - El método DELETE debe utilizarse para eliminar un recurso, que posteriormente quedará inaccesible para los clientes. - El método POST también se debe utilizar para modelar operaciones con efectos secundarios que no encajen con los otros métodos. - El método OPTIONS debe usarse para recuperar metadatos que describen las interacciones disponibles de un recurso. - El método HEAD debe usarse para recuperar metadatos del estado actual de un recurso.	[20][37] [34][36] [41][33] [35]

Tabla 5.5: Operatividad parte I

Código	Heurísticas	Guías	Ref.
Operatividad			
Precisión - Código de estado			
OP-2	Usar códigos de estado HTTP que más se ajuste a la respuesta de una solicitud.	- Devolver códigos de estado HTTP. - Para que sea lo más preciso posible, no usar sólo un código de éxito y uno de error.	[20][37] [34][36] [33][41]
Universalidad - Universalidad cultural - Convenciones culturales - Cuerpo del mensaje			
OUC-1	La API debe evitar el uso de elementos (términos, formatos, ortografía, etc.) que no se reconocen universalmente y estar escrita en el lenguaje más usado por las TI.	- Usar expresiones universales que no estén limitadas a un idioma en particular. - Usar términos estándar. - Usar sistema de codificación de caracteres UTF-8. - Implementar la API en inglés.	[27]
Flexibilidad - Adaptabilidad - Adaptación a los entornos - API			
FAA-1	Cuando se lanza una versión nueva de la API, esta debe ser compatible con la anterior durante un periodo de tiempo para que los clientes puedan adaptarse correctamente.	- Una versión nueva debe de convivir con la anterior durante un periodo mínimo de tiempo.	[37]
Flexibilidad - Controlabilidad - Configurabilidad - Configurabilidad de aspectos técnicos - Representación del recurso			
FCC-1	La API debe ser capaz de representar un recurso en varios formatos. La API debe proporcionar estados parciales de los recursos.	- Se le debe permitir a los clientes negociar el formato del cuerpo del mensaje mediante el encabezado *Accept*. - Se debe admitir solicitudes parciales de estados de un recurso mediante consultas parametrizadas.	[20][37] [30][24] [33][35] [41][38]

Tabla 5.6: Operatividad parte II

Código	Heurísticas	Guías	Ref.
Eficiencia			
Eficiencia respecto a tiempo de ejecución de tareas - Respuesta a una solicitud - Respuesta del sistema			
EER-1	Se debe proporcionar metadatos específicos en la cabecera que ayuden a procesar y reducir la latencia de la respuesta a una solicitud.	- La cabecera de la respuesta debe de contener *Content-Length*. - La cabecera de la respuesta debe contener *Content-Type*. - Cuando se acaba de crear un recurso, en la cabecera de la respuesta debe contener *Location* para especificar el URI del recurso recién creado. - Usar *Cache-Control, Expires* y *Date* en la cabecera de las respuestas para fomentar el almacenamiento en caché, siempre y cuando los recursos no tengan información confidencial o personal del cliente o la información no cambie con mucha frecuencia - Si no se quiere usar el almacenamiento en caché para las respuestas, en vez de usar *Cache-Control:no-cache* usar un valor muy pequeño de *max-age*. - Usar cabeceras de almacenamiento en caché con tiempo de expiración para respuestas exitosas a solicitudes GET y HEAD. - Usar cabeceras de almacenamiento en caché para respuestas 3xx y 4xx, es decir, en redirecciones y errores del cliente. - En la cabecera de la respuestas a solicitudes GET, usar *ETag* o *Last-Modified*.	[20][36] [24][41]
Eficiencia respecto a tiempo de ejecución de tareas - Respuesta a una solicitud - Acciones del usuario			
EEA-1	Proporcionar vistas de consultas más comunes como recursos.	- Crear consultas predefinidas, filtros o vistas sobre una colección.	[19]

Tabla 5.7: Eficiencia parte I

Código	Heurísticas	Guías	Ref.
Eficiencia			
Eficiencia respecto a costes económicos - Costes de consumo, recursos humanos y del sistema - API			
EEC-1	Los costes económicos para el uso de la API, si corresponde, deberían ser razonables.	- El coste de la licencia de uso debe ser adecuado para los servicios ofrecidos y estar acorde con los precios del mercado. - El uso de la API no debe implicar costes excesivos de personal o del equipo.	[27]
Robustez			
Robustez ante usos incorrectos por parte del usuario - Solicitud y Respuesta			
RU-1	La API debe hacer un uso correcto del protocolo HTTP.	- No usar corchetes cuando tenemos campos de una matriz dentro de una consulta parametrizada en el URI. - Las cabeceras personalizadas deben tener fines informativos. Si la información que está transmitiendo a través de un cabecera HTTP personalizada es importante para interpretar correctamente la solicitud o respuesta, se debe incluir en el cuerpo del mensaje.	[20][42]
Seguridad			
Seguridad del usuario y de terceros - API - Confidencialidad del usuario			
SUC-1	La API no debe comprometer la confidencialidad de la información personal de los clientes.	- La API no debe acceder a los datos almacenados en la maquina del usuario que no sean necesarios para su funcionamiento. - La API no debe proporcionar datos personales a terceros sin tener consentimiento.	[27]

Tabla 5.8: Eficiencia parte II, Robustez y Seguridad parte I

Código	Heurísticas	Guías	Ref.
Seguridad			
Seguridad del usuario y de terceros - API - Confidencialidad del usuario			
SUC-2	La API debe proporcionar conexiones seguras entre los clientes y el servidor.	- Usar OAuth 2.0 como método de autorización. - Usar autenticación basada en token. - Usar TLS. - Si hay información confidencial en la respuesta, se debe de usar *Cache-control:private* en la cabecera, para evitar que la caché se comparta con otros clientes. - No mostrar datos confidenciales en el URI. - Configurar el servidor para atender solo solicitudes HTTP.	[20][37] [38][32] [43][21]
Seguridad del usuario y de terceros - API - Seguridad legal			
SUL-1	La API no debe poner al usuario en problemas legales.	- No permitir el acceso a datos protegidos, como por ejemplo, contenido protegido por derechos de autor, contenido confidencial, etc.	[27]
SUL-2	La API debe indicar claramente su licencia de uso.	- Mostrar qué tipo de licencia tiene la API.	[27]

Tabla 5.9: Seguridad parte II

Código	Heurísticas	Guías	Ref.
Seguridad			
Seguridad del entorno - API			
SE-1	La API debe estar implementada en un entorno seguro.	- Usar proxy inverso o una puerta de enlace entre cliente y servidor. - Hacer un seguimiento de uso. - Marcar un límite de uso. - No se debe acoplar el diseño con la implementación del servicio. - Validar los tipos MIME en la cabecera *Content-Type* de la solicitud. - No admitir que la cabecera *Accept* venga vacía desde una solicitud. - Si nuestra API expone sólo el método GET sobre un recurso, no se debe permitir que la solicitud contenga cabeceras como: *X-HTTP-Method-Override* o *X-Method-Override*, ya que puede sobrescribir el método que se utiliza sobre el URI. - El servidor debe enviar la cabecera: *X-Content-Type-Options* con valor *nosniff* para asegurarse que el navegador no intente detectar otros tipos MIME diferentes al del valor de *Content-Type* y así evitar posibles ataques XSS. - Cuando la API devuelva información sobre la causa de un error, estas explicaciones no deben ser muy explícitas. - Proteger acciones privilegiadas sobre recursos contra el uso no autorizado.	[20][30][44][45][46]

Tabla 5.10: Seguridad parte III

Código	Heurísticas	Guías	Ref.
Satisfacción subjetiva			
Estética visual - API			
SSE-1	La API debe ser lo suficientemente atractiva para que el usuario encuentre cómodo su uso.	- La API no debe ser estéticamente desagradable, como por ejemplo: tener nombres raros, usar caracteres especiales de manera inapropiada, etc.	[27]

Tabla 5.11: Satisfacción subjetiva parte I

A continuación se muestra el conjunto de ejemplos y contraejemplos de algunas de las heurísticas y guías de usabilidad en tablas aparte debido a la dificultad de mostrar toda la información en una única tabla. Los ejemplos correctos se presentan en color verde y los contraejemplos en color rojo.

Código	Ejemplos y contraejemplos
CCC-1	No incluir la barra diagonal al final: /customers/1234 /invoices/1/items /customers/1234/ URI con relaciones jerárquicas entre recursos y colecciones: /customer/orders/order1 /university/faculty/professors URI con parámetros: /search?word=Antarctica&limit=30 URI sin relacion jerárquica entre recursos: /co-ordinates;w=39.001409,z=-84.578201
CCC-2	URI con una ruta de recurso ordenada: /profiles?sort=-education&fields=name /profiles/sort/desc/name
CCF-1	URI siempre en minúscula, sin la extensión del archivo o formato y con guiones: /my-folder/my-doc api.invoicehome.com/customers/1 /_NEWCustomer/_photo01.jpg/ /customers/1234.pdf
CCF-2	URI con número decente de parámetros de consulta: /profiles/news/science?fields=name,summary/profiles/news/history /books/news?category=science&fields=name,author,summary...
CCC-3	Representación de un recurso: /newspapers/media/page?id=123 Representación de una colección: /leagues/seattle/teams Uso de métodos: POST /team/players DELETE /team/player Uso de sustantivos en vez de términos CRUD: POST /players/age?id=123 PUT /update/players/age?id=123 DELETE /university/deleteCenter?id=1 URI con controlador sobre un recurso: /users/search /students/morgan/register

Tabla 5.12: Ejemplos (verde) y contraejemplos (rojo) de las heurísticas de usabilidad (parte I)

Código	Ejemplos y contraejemplos
CCF-3	Nombre de dominio completo en el URI: www.soccer.**restapi**.org www.soccer.**api**.org Nombre de dominio completo en el URI para un portal de desarrolladores: http://**developer**.soccer.restapi.org Versionado de la API: api.example.com/exampleAPI/messaging/**v1**/ para la versión 1.x. api.example.org/**v2**/customer/1234 para la versión 2.x. api.example.com/exampleAPI/messaging/1/ api.example.com/exampleAPI/messaging/1.1/
CCC-4	Uso de dos URI para la representación de un recurso: /competence-service/profiles/123 Primera URI: competence-service/profiles Segunda URI: 123 /leagues/seattle/teams/trebuchet/players/claudio?fields=age Rrimera URI: leagues/seattle/teams/trebuchet/players Segunda URI: claudio?fields=age URI con relación jerárquica entre recursos: /university/faculty/professors /leagues/seattle/teams /players/123 /leagues/seattle/teams/trebuchet/players/claudio /profs/university/faculty /seattle/leagues/trebuchet/players/claudio?fields=age

CCC-5	Objeto JSON bien formado:
	`{ "firstName" : "Osvaldo", "lastName" : "Alonso", "number" : 6, "birthdate" : "1985-11-11" }`
	Uso de Content-Type: multipart/related:
	`boundary=foo_bar_baz --foo_bar_baz Content-Type: application/json; { "name": "myObject" } --foo_bar_baz Content-Type: image/jpeg [JPEG\_DATA] --foo_bar_baz--`

Tabla 5.13: Ejemplos (verde) y contraejemplos (rojo) de las heurísticas de usabilidad (parte II)

<table>
<tr><th>Código</th><th>Ejemplos y contraejemplos</th></tr>
<tr><td>CMF-1</td><td>URI con parámetro de filtrado:
/tickets?state=open
URI con parámetro de ordenación:
/profiles?sort=-education,+experience
URI con parámetros de selección:
/profiles?fields=name,experience
URI con parámetros de paginación:
/profiles?offset=0&limit=10
URI con un recurso controlador para paginación y filtrado:
/users/search
URI auto-descriptiva:
/customers/1
/all/1
/songs?n=rhapsody</td></tr>
<tr><td>CAI-1</td><td>Forma correcta:
<pre>HTTP/1.1 404 NOT FOUND
/* More header information */
{
    "error" : {
    "responseCode" : 404,
    "errorCode" : 107,
    "messages" : {
        "developer" : "The resource "profile" could not be found.",
        "user" : "An error occurred while requesting the
    information. Please contact our technical support."
    },
    "additionalInfo": ".../competenceservice/errors/107"
}</pre>Forma incorrecta:
<pre>/*Avoid returning success code with an error in the body */
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8
<error>
    <message>Account limit exceeded.</message>
</error></pre></td></tr>
<tr><td>FCC-1</td><td>Negociación del formato del recurso:
GET /universities
Accept: application/json

GET /universities.json
Solicitud parcial de un recurso:
GET /profiles?fields=name,experience</td></tr>
</table>

Tabla 5.14: Ejemplos (verde) y contraejemplos (rojo) de las heurísticas de usabilidad (parte III)

Código	Ejemplos y contraejemplos

OC-1	Uso correcto en el cuerpo del mensaje 1:
	`"name":"UniTN",`
	`"links":{`
	`"faculty-centers":"/universities/1/faculty-centers"`
	`}`
	Uso correcto en el cuerpo del mensaje 2:
	`"firstName":"Carlos",`
	`"lastName":"Fernández",`
	`"links":{`
	`"team":{`
	`"href":"api.soccer.restapi.org/teams/seattle",`
	`"rel":"api.relations.wrml.org/soccer/team"`
	`},`
	`"addToFavorites":{`
	`"href":"api.soccer.restapi.org/users/42/favorites/{name}",`
	`"rel":"api.relations.wrml.org/soccer/team"`
	`},`
	`}`

EEA-1	Posibles consultas predefinidas: GET /tickets/recentlyclosed GET /book/news
RU-1	Consulta de parámetros de una matriz en el URI: /authors?name=kay,xing /authors?name[]=kay&name[]=xing Cabecera personalizada: Poner 'X-' antes del nombre de nuestra cabecera, como por ejemplo: X-myMetadata: metadata
OP-1	Uso de GET sobre una colección y un recurso específico: GET /users GET /users/john Uso de POST para crear un recurso dentro de una colección: POST /leagues/seattle/teams/trebuchet/players POST /users Uso de PUT para modificar un recurso específico: PUT /users/john PUT /accounts/40200c9a66/buckets/objects/4321 Uso de DELETE para eliminar un recurso específico: DELETE /accounts/40200c9a66/buckets/objects/4321 DELETE /users/john Uso de POST para modelar una operación sobre un recurso: POST students/morgan/register

Tabla 5.15: Ejemplos (verde) y contraejemplos (rojo) de las heurísticas de usabilidad (parte IV)

Capítulo 6

Aplicación a un caso práctico

En este capítulo detallaremos cómo hemos aplicado la segunda parte de la metodología explicada en el capítulo 3 para realizar un estudio de usabilidad a un caso práctico mediante una evaluación heurística, con el fin de validar el conjunto de heurísticas y guías desarrolladas en este trabajo.

A continuación, describiremos el caso de estudio elegido, que en este caso ha sido el API REST para desarrolladores de Twitter [47], qué aspectos caracterizan el contexto de uso para realizar un estudio de usabilidad riguroso y los resultados finales de la evaluación heurística con varias propuestas de mejora.

6.1 Caso práctico

Twitter es una de las redes sociales más importantes que existen actualmente, puesto que ha conseguido a millones de usuarios en todo el mundo, siendo imprescindible para las empresas y personas que quieran tener una presencia influyente en internet. Esta plataforma permite que los usuarios publiquen mensajes de texto plano de corta longitud dentro de su página principal, llamados tweets. Muchos usuarios utilizan la red social con el fin de dar a conocer sus servicios y productos, difundir información sobre su empresa, crear nuevos contactos o anunciar ofertas y promociones. Otros incluso, sólo la utilizan como medio de entretenimiento y estar informado de lo que ocurre alrededor del mundo.

Particularmente, examinaremos la API que presenta esta red social, que permite acceder a partes de su servicio, para permitir a los desarrolladores integrar su software con dicha plataforma. La API proporciona un amplio conjunto de datos compartidos por los usuarios y permite a los desarrolladores a administrar su información privada. Esta información se divide en cinco grupos [48]:

- **Cuentas y usuarios**: Permite a los desarrolladores administrar la información de su perfil y la configuración de su cuenta, como silenciar o bloquear usuarios y administrar

usuarios y seguidores.

- **Tweets y respuestas**: Los desarrolladores pueden publicar y acceder a tweets y respuestas públicas buscando por palabras clave específicas o solicitando un conjunto de tweets de alguna cuenta en particular.
- **Mensajes directos**: Acceso a las conversaciones privadas de usuarios que han otorgado permiso de forma explícita.
- **Anuncios**: Le permite a los desarrolladores ayudar a las empresas a crear y administra automáticamente campañas de anuncios.
- **Herramientas y SDK**: Proporcionan herramientas para desarrolladores y editores de software que permiten integrar los contenidos de Twitter en las páginas web.

Para realizar la evaluación heurística, hemos analizado un conjunto de solicitudes y respuestas que pertenecen a los grupos de *Cuentas y usuarios* y *Tweets y respuestas*.

Concretamente, las solicitudes analizadas permiten seguir, buscar y obtener información de usuarios [49], y crear, recuperar y eliminar Tweets, Retweets y dar *Like* a los anteriores [50].

6.2 Caracterización del contexto de uso

Para hallar las características del contexto de uso y estudiar cómo se comporta realmente los usuarios, las tareas y entorno dentro de nuestro caso de estudio, y así conocer qué aspectos son importantes para nuestro estudio de usabilidad, recorremos la taxonomía definida por Alonso-Ríos et al. [18], detallado en el capítulo 2.

Las características de un desarrollador de software, que en nuestro caso de estudio hablamos del usuario que interactúa con el sistema son:

- **Formación académica y conocimiento del dominio del sistema**: El usuario no tiene por qué tener experiencia en APIs REST, pero sí que es necesario que esté formado técnicamente dentro de las TI y haya adquirido competencias profesionales con respecto al uso del protocolo HTTP, desarrollo de aplicaciones web, y lo que es más importante, que conozca perfectamente las restricciones del estilo REST para utilizar y aprovechar el sistema correctamente.
- **Características físicas**. El usuario utiliza la vista para comprender el funcionamiento del sistema y para recibir su *feedback*, de modo que nos centraremos particularmente en capacidades y discapacidades relacionadas con la vista.

- **Características cognitivas**: Para el uso de este tipo de sistemas, es importante considerar las características mentales del usuario como el razonamiento y la memoria.

Las tareas que realizan los usuarios dentro del sistema consisten en construir solicitudes HTTP para operar sobre los datos que expone la API e interpretar y manipular correctamente los datos devueltos en las respuestas. Las características más relevantes son:

- **Complejidad**: Las tareas del sistema se deben realizar a partir un conjunto estable de acciones mediante una interfaz uniforme de acceso a datos implementada bajo los principios REST, que consisten principalmente en identificación de recursos, manipulación de recursos a través de representaciones, mensajes auto-descriptivos e hipermedia como motor de estado.

  Esto implica que los métodos HTTP se usen correctamente sobre URIs consistentes y bien definidos. Los datos deben ser representados por formatos estándar dentro de mensajes auto-descriptivos, de tal forma que contenga información suficiente para saber cómo procesar el mensaje. Por último, para que los usuarios estén mínimamente acoplados al sistema y tengan un acceso rápido a los datos, este debe ajustarse al principio HATEOAS.

  Generalmente, las tareas no deben ser muy complejas si se ajustan a los principios mencionados anteriormente. Si la API está acompañada de una documentación amplia que describa de forma completa cada elemento involucrado en una tarea y ejemplos de uso de la misma, los usuarios tendrán una visión favorable de uso de la API.

- **Demandas cognitivas**: Las tareas no deben abusar de las capacidades cognitivas de los usuarios, como el razonamiento y la memoria.

- **Seguridad legal y confidencialidad**: Las tareas no deben acceder a datos protegidos para no causar problemas legales. Los datos confidenciales de los usuarios deben estar protegidos, de la misma manera que se debe restringir el acceso a los recursos de los usuarios que no estén autorizados.

Nuestro entorno dentro del contexto de uso respecta al servidor que procesa cada solicitud y respuesta. Sus características son:

- **Idoneidad de los equipos lógicos**: La eficiencia, robustez y seguridad juegan un papel importante dentro del entorno del sistema. El entorno debe fomentar el uso de caché cuando se procese cada mensaje entre cliente y servidor. Esto implica que el rendimiento sea favorable y que por lo tanto, los usuarios puedan acceder a los datos de la forma más eficiente posible.

También, debe ser capaz de recuperarse ante fallos internos, haciendo uso de un protocolo sin estado, lo que significa que los mensajes de los usuarios tendrán toda la información necesaria para que el servidor procese dichas peticiones, sin que rastree el estado sobre las anteriores.

Los clientes que interactúan con el entorno, deben hacerlo de manera segura para evitar ataques y problemas de securización. La API debe permitir que los usuarios utilicen métodos de autenticación y autorización para operar sobre los datos. Es recomendable que siga otros tipos de directrices para reforzar la seguridad, como por ejemplo, no acoplar la implementación del servicio con la interfaz, o alguna solución que implique hardware adicional entre las comunicaciones cliente-servidor.

6.3 Evaluación heurística de usabilidad

Para validar el conjunto de heurísticas y guías de las tablas 5.1 - 5.11, se aplicaron al caso práctico que hemos explicamos previamente para hacer una evaluación heurística y estudiar su usabilidad.

En las tablas siguientes mostraremos los resultados con cada heurística identificada por su código asignado en el capítulo 5, y clasificadas de acuerdo a las categorías explicadas en la tabla 6.1.

Para aplicar cada heurística dentro de la API y comprobar su funcionamiento, hemos utilizamos *Postman* [51], un cliente HTTP dirigida a desarrolladores web que permite realizar peticiones a cualquier API.

	Comentarios
*	El caso práctico **no cumple** con el conjunto de guías.
**	El caso práctico cumple **parcialmente** el conjunto de guías.
***	El caso práctico **cumple** con el conjunto de guías.

Tabla 6.1: Criterios de evaluación

Heurística		**Evaluación Heurística**
Cognoscibilidad		
Claridad conceptual		
CCC-1 Los elementos del URI se deben comprender con facilidad.	***	Los elementos del URI se comprenden con facilidad.
CCC-2 La estructura base de la ruta de un recurso debe ser fácil de comprender.	***	La estructura de la ruta del recurso se comprende con facilidad.
Claridad formal		
CCF-1 Las URI deben ser claramente visibles.	*	Los URI no son claramente visibles porque usan la extensión del formato y guiones bajos.
CCF-2 La estructura base de la ruta de un recurso debe ser claramente visible.	***	Las rutas de los recursos no utilizan jerarquías muy largas, de manera que su estructura se puede visualizar con claridad.
Consistencia conceptual		
CCC-3 Los términos usados en la ruta de un recurso, deben ser consistentes con el método HTTP de la solicitud.	*	Los términos usados en los URI no son consistentes con el método HTTP de la solicitud. Utiliza términos CRUD y abusa del uso de controladores sobre recursos.
CCC-4 El URI debe tener una estructura consistente cuando identifique recursos relacionados jerárquicamente.	**	Las relaciones jerárquicas entre recursos tienen un significado consistente, pero se abusa de controladores sobre recursos.

Tabla 6.2: Evaluación heurística (parte I)

Heurística		Evaluación Heurística
Cognoscibilidad		
Consistencia conceptual		
CCC-5 El cuerpo de la solicitud y respuesta debe estar representado por formatos consistentes.	***	Se utilizan tipos MIME consistentes para representar el cuerpo de la solicitud y respuesta.
Consistencia formal		
CCF-3 Se debe percibir con facilidad que el URI hace referencia a una API con su versión correspondiente.	**	La versión de la API no se percibe con facilidad, aparece de forma ambigua.
Memorabilidad		
CMF-1 Los términos y parámetros de consulta en el URI, deben ser fáciles de usar y de recordar.	***	Usa términos descriptivos y controladores sobre recursos fáciles de recordar. Los parámetros de paginación son fáciles de usar.
Asistencia		
CAI-1 La API debe mostrar información útil del error cuando hay un problema en la solicitud.	*	La información del error cuando hay un problema en la solicitud es escasa. Sólo se muestra un mensaje para describir el error.
CAI-2 La documentación debe definir y describir cada elemento de la API y mostrar ejemplos de uso.	**	La documentación a veces resulta un poco incompleta. No define cada recurso con sus atributos y algunas solicitudes no tienen ejemplos de respuestas y de posibles códigos de estado.
Operatividad		
Completitud		
OC-1 La representación del recurso en la respuesta debe estar acompañado por un conjunto de links, formados por aquellos URIs que están relacionados con él.	*	Las representaciones de los recursos son incompletas porque no hay URIs a acciones y a recursos relacionados.

Tabla 6.3: Evaluación heurística (parte II)

Heurística		**Evaluación heurística**
Operatividad		
Precisión		
OP-1 Se debe utilizar correctamente los métodos HTTP en una solicitud.	*	No se utiliza de manera precisa los métodos HTTP cuando se quiere eliminar y actualizar un recurso. En varias solicitudes GET se recomienda que se utilice POST si la solicitud es muy larga.
OP-2 Usar códigos de estado HTTP que más se ajuste a la respuesta de una solicitud.	**	A veces los códigos de estado no son precisos con la respuesta.
Universalidad cultural		
OUC-1 La API debe evitar el uso de elementos (términos, formatos, ortografía, etc.) que no se reconocen universalmente y estar escrita en el lenguaje más usado por las TI.	***	La API puede ser usada por usuarios con diferente cultura porque utiliza términos estándar, el sistema de codificación de caracteres UTF-8 y está implementada en inglés.
Flexibilidad		
FAA-1 Cuando se lanza una versión nueva de la API, esta debe ser compatible con la anterior durante un periodo de tiempo para que los clientes puedan adapatarse correctamente.	***	La API permite a los desarrolladores a adaptarse cuando hay una versión nueva. Además, indica en qué fechas está disponible y cuando queda obsoleta.
FCC-1 La API debe ser capaz de representar un recurso en varios formatos. La API debe proporcionar estados parciales de los recursos.	*	Los desarrolladores no pueden configurar las solicitudes para negociar el formato del cuerpo del mensaje, ni pueden solicitar estados parciales de recursos.

Tabla 6.4: Evaluación heurística (parte III)

Heurística		Evaluación heurística
Eficiencia		
EER-1 Se debe proporcionar metadatos específicos en la cabecera que ayuden a procesar y reducir la latencia de la respuesta a una solicitud.	**	Se usan cabeceras que ayudan a procesar y a reducir el tiempo de respuesta a una solicitud, pero siempre se utiliza *Cache-control:no-cache, no-store*.
EEA-1 Proporcionar vistas de consultas más comunes como recursos.	***	Los desarrolladores acceden a los datos de manera eficiente porque la API proporciona vistas de solicitudes más comunes como recursos.
EEC-1 Los costes económicos para el uso de la API, si corresponde, deberían ser razonables.	***	La API es eficientemente económica. La versión premium varía entre 149$/mes a 2,499$/mes, según el nivel de acceso necesario.
Robustez		
RU-1 La API debe hacer un uso correcto del protocolo HTTP.	***	Las cabeceras personalizadas en las respuestas tienen fines informativos, por lo tanto, la API es robusta al uso incorrecto del protocolo HTTP por parte de los desarrolladores.
Seguridad		
SUC-1 La API no debe comprometer la confidencialidad de la información personal de los clientes.	***	La API no compromete la confidencialidad de la información personal.
SUC-2 La API debe proporcionar conexiones seguras entre los clientes y el servidor.	***	La API proporciona conexiones seguras a los desarrolladores, ya que no muestra datos confidenciales en el URI, utiliza TLS, evita compartir la caché con otros clientes utilizando *Cache-control:private* en la cabecera de la respuesta, utiliza OAuth como método de autorización y la autenticación basada en token.
SUL-1 La API no debe poner al usuario en problemas legales.	***	La API no permite que los desarrolladores accedan a contenido confidencial.
SUL-2 La API debe indicar claramente su licencia de uso.	***	Se muestra qué requisitos son necesarios para acceder a cada versión de la API: estándar, premium, empresarial y de anuncios.

Tabla 6.5: Evaluación heurística (parte IV)

Heurística		Evaluación heurística
Seguridad		
SE-1 La API debe estar implementada en un entorno seguro.	***	La API garantiza la seguridad del servidor porque existe un límite de solicitudes, no se acopla su diseño con la implementación del servicio, se utiliza la cabecera *X-Content-Type-Options* con valor *nosniff*, no devuelve información muy explícita que implique a la implementación del servicio cuando hay un error y protege acciones privilegiadas sobre recursos mediante un token de sesión.
Satisfacción subjetiva		
SSE-1 La API debe ser lo suficientemente atractiva para que el usuario encuentre cómodo su uso.	***	La API es estéticamente agradable.

Tabla 6.6: Evaluación heurística (parte V)

6.4 Análisis de resultados

Dados los resultados de la evaluación heurística (Figura 6.1), unos de los problemas más significativos de usabilidad que hemos detectado han sido en los atributos de la cognoscibilidad, particularmente en la asistencia que proporciona la API para dar información útil a los desarrolladores. Encontramos que la documentación para algunas solicitudes está incompleta y un poco desestructurada (CAI-2) y la información de error dentro de la respuesta es muy escasa, mostrando una descripción breve de su causa (CAI-1).

Otro problema detectado que afecta negativamente a la usabilidad, se corresponde con la operatividad de la API. En ocasiones, no utiliza correctamente los métodos HTTP sobre los datos identificados en el URI y por lo tanto, afecta a la precisión de las tareas, como el uso de POST sobre un conjunto de datos que requiere una petición segura como GET, o en vez de DELETE para eliminar un recurso (OP-1). Esto también repercute a que los código de estado devueltos en la respuesta no sean coherentes con el método HTTP que se utiliza.

Esto puede confundir a los desarrolladores que sepan utilizar adecuadamente una API REST, de manera que tengan que realizar esfuerzos a mayores para investigar por qué la API no se ajusta al protocolo HTTP cuando define un método sobre un conjunto de datos (como POST en vez de DELETE).

De nuevo, con respecto a la precisión de la API, en algunos casos, los códigos de estado no sirven para entender el significado de la respuesta (OP-2). En las solicitudes POST para crear recursos nuevos, se utiliza 200 OK, en vez de 201 Created, e incluso se llega a utilizar un código de éxito en vez de uno de error, como ocurre con la solicitud que permite añadir usuarios a la lista de amigos, devolviendo un 200 OK cuando se añade otra vez un usuario a la lista de amigos, en lugar de un 403 Forbidden.

Las representaciones de los recursos dentro del cuerpo del mensaje no cumplen con el principio de HATEOAS (OC-1), de modo que los desarrolladores están fuertemente acoplados a los servicios que ofrece la API, e implica a que estén obligados a que lleven a cabo un mantenimiento particular para cada recurso por si en un futuro el URI cambia.

Además, se desaprovecha uno de los puntos fuertes de HTTP que es el uso de la cabecera *Accept* (FCC-1), que permite negociar el formato de la representación del recurso dentro de la respuesta, para que los desarrolladores no tengan que añadir código adicional en la parte del cliente de sus aplicaciones.

Generalmente, la API no presenta problemas de claridad y consistencia, pero sí cabe destacar que el uso de guiones bajos, utilizar la extensión del formato del recurso (CCF-1), abusar de controladores y no utilizar los sustantivos correctamente para representar los recursos en el URI (CCC-3), impide que los desarrolladores perciban con exactitud qué recursos están implicados en la solicitud.

Estos problemas de usabilidad afectan principalmente al razonamiento del usuario y a la complejidad de las tareas dentro del contexto de uso. Aunque, como vemos en la figura 6.1, la usabilidad también es afectada ligeramente por la eficiencia del entorno al utilizar el valor *Cache-control:no-cache* en la cabecera de las respuestas.

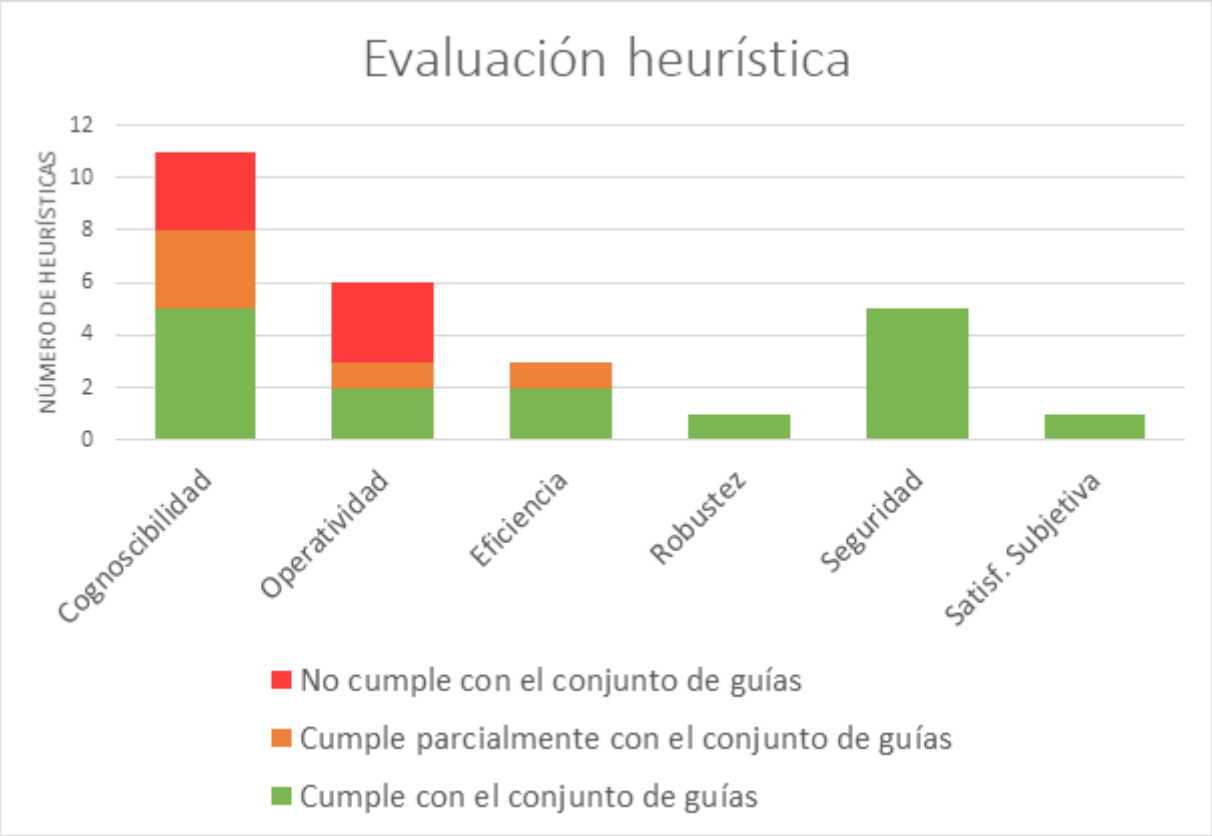

Figura 6.1: Resultados de la evaluación heurística

6.5 Propuestas de mejora

En este apartado mostraremos varias propuestas de mejora para algunos problemas encontrados a partir de las heurísticas CCC-3, CCC-4 y OP-1 dado que su solución no es inmediata al aplicar dichas heurísticas sobre la API.

Las soluciones que proponemos se centran generalmente en usar correctamente los métodos HTTP, sobre todo el uso de POST y PUT, que en ocasiones resulta bastante confuso y en transformar cada elemento que interviene en el URI para convertirlo en una interfaz genérica, y así evitar problemas de consistencia entre el significado de cada término que identifica un conjunto de datos.

Las mejoras se han hecho sobre los dos grupos de solicitudes de la API a los que hemos

aplicado la evaluación heurística [49][50]. Cada recurso implicado en estas solicitudes son explicados a continuación:

- **/friendships**: contiene información detallada sobre las relaciones del usuario que envía la solicitud.
- **/friends**: contiene información de usuarios que sigue el usuario que envia la solicitud.
- **/users**: contiene información detallada de un usuario.
- **/statuses**: contiene información de cualquier tweet publicado.
- **/favorites**: conjunto de tweets favoritos del usuario que envía la solicitud. También conocidos por los tweets a los que ha dado *like*.

En los URIs siguientes, intervienen más parámetros de los que hemos usado, cómo *count* y *cursor* que permiten paginar los resultados o los que son específicos para configurar cada solicitud. Nos hemos limitado a utilizar los mínimos posibles para que los ejemplos sean más legibles y fáciles de entender, ya que añadir uno o más parámetros sería algo inmediato.

Estos son: *screen_name* que sirve para especificar el nombre del usuario y se comporta de forma similar al ID de su cuenta, y *retweets*, que activa o desactiva los retweets de un usuario en particular.

Cabe destacar que la API crea y modifica los recursos mediante consultas parametrizadas en vez de definirlos en el cuerpo del mensaje de la solicitud. Esto es así para evitar ataques de inyección de SQL, que consisten en insertar sentencias SQL maliciosas obteniendo acceso a datos sensibles en la base de datos.

Solicitudes originales
POST friendships/create.json?screen_name={screen_name}
POST friendships/update.json?screen_name={screen_name}&retweets={true||false}
POST friendships/destroy.json?screen_name={screen_name}
GET friendships/no_retweets/ids.json
Mejoras
POST friendships?screen_name={screen_name}
PUT friendships/{screen_name}?retweets={true||false}
DELETE friendships/{screen_name}
GET friendships/no-retweets

Tabla 6.7: Propuestas de mejora (parte I)

Solicitudes originales
GET friends/list.json
GET friends/ids.json
Mejoras
GET friends?list=true para devolver una colección de objetos
GET friends?list=false para devolver una colección de IDs

Tabla 6.8: Propuestas de mejora (parte II)

Solicitudes originales
GET users/show.json
Mejoras
GET users/{id}
GET users/{screen_name}

Tabla 6.9: Propuestas de mejora (parte III)

Solicitudes originales
POST statuses/update.json?status={tweet_message}
GET statuses/retweeters/ids.json
Mejoras
POST statuses?status={tweet_message}
GET statuses/retweeters

Tabla 6.10: Propuestas de mejora (parte IV)

Solicitudes originales
POST favorites/create.json?id={tweet_id}
POST favorites/destroy.json?id={tweet_id}
GET favorites/list.json
Mejoras
POST favorites?id={tweet_id}
DELETE favorites/{tweet_id}
GET favorites

Tabla 6.11: Propuestas de mejora (parte V)

Capítulo 7

Discusión y conclusiones

El objetivo principal de este trabajo ha sido desarrollar un conjunto de heurísticas y guías de usabilidad para el diseño de APIs RESTful y validarlo mediante una evaluación heurística aplicada sobre un caso práctico.

Para desarrollar el conjunto de heurísticas hemos hecho un análisis exhaustivo de la bibliografía sobre principios de diseño, buenas prácticas y recomendaciones de APIs REST, ya que prácticamente no existe información precisa sobre como garantizar la usabilidad en APIs de este estilo.

A partir de la información recopilada del análisis, creamos 27 heurísticas acompañadas por varias guías detalladas dentro de un modelo expandido de usabilidad que cubre todos los aspectos del término estructurados jerárquicamente en varios niveles.

Para comprobar que las heurísticas y guías que hemos desarrollado son correctas y puedan aplicarse adecuadamente, hemos realizado una evaluación heurística aplicándolas sobre el API REST de Twitter para estudiar su usabilidad. Previamente, caracterizamos el contexto de uso para detectar y definir los problemas de usabilidad con mayor precisión, ya que en este paso, estudiamos todas las particularidades de los usuarios, tareas y entorno que intervienen dentro del caso práctico.

Sí que es verdad que el grupo de solicitudes y respuestas que hemos estudiado es reducido en comparación con el tamaño general de la API, pero hemos elegido aquellas que realizan las tareas esenciales de la red social, y que nos ha servido para encontrar varias deficiencias considerables de usabilidad.

Después de analizar y comentar los problemas de usabilidad detectados mediante de la evaluación heurística, propusimos varias mejoras sobre la API estudiada para resolver algunas de estas carencias. Por lo tanto, podemos asegurar que nuestro conjunto de heurísticas y guías de usabilidad funciona correctamente para cumplir su propósito.

Aplicar nuestras heurísticas a la API REST para desarrolladores de Twitter, nos ha ayudado a detectar qué partes presentan problemas de usabilidad. En vista a los resultados de

la evaluación heurística, vimos que los atributos de usabilidad más afectados fueron los que pertenecen a la cognoscibilidad y a la operatividad, es decir, lo que definen cómo de fácil es comprender y aprender a utilizarla, y cómo ésta proporciona las funcionalidades que los desarrolladores se esperan de ella. En cuanto a la eficiencia, no hubo problemas relevantes, pero sí cabe destacar que no se cumplieron completamente con las heurísticas que aseguran la usabilidad con respecto a este atributo.

Algunos de estos, están ligados al uso incorrecto del estilo REST, especialmente, evitar el almacenamiento en caché, el uso de hipermedia dentro del cuerpo de la respuesta y no presentar un conjunto de operaciones estables sobre los datos.

Desafortunadamente, no es extraño que la mayoría de APIs REST públicas de compañías reconocidas mundialmente utilicen el término REST para describir su interfaz de acceso a servicios, cuando en realidad, sólo se limitan a usar el protocolo HTTP sin ninguna restricción, tal y como nos cuentan Renzel et al.[52]. Muchos desarrolladores asocian REST con HTTP, pero esto es un error. No hay nada en las restricciones REST que haga obligatorio el uso de HTTP como protocolo de transferencia, de hecho, es correcto usar otros protocolos, como SMTP [53]. Todo esto es consecuencia de la falta y desorganización de la información sobre las características técnicas, a pesar de la relevancia e interés que tienen estos sistemas en la actualidad.

Por lo tanto, esto podría ser motivo suficiente por el cual no existen estudios ni pautas enfocadas a la usabilidad de las APIs REST, sumándole también, las discrepancias a cerca de este término. Como dijo Lewis en su crítica [1], la medición de la usabilidad es compleja porque la usabilidad no es una propiedad específica de un producto o sistema que podamos conocer con facilidad y la mayoría del trabajo sobre la ingeniería de usabilidad es confidencial, ya que las empresas prefieren no compartir qué carencias de usabilidad encuentran en sus productos o servicios, cómo las han detectado y solucionado.

Sin embargo, existen muchos estudios significativos que describen una metodología para garantizar la calidad de experiencia del usuario y estudiar la usabilidad en cualquier tipo de sistema. Por esta razón, nos hemos basado en varios estudios publicados por los profesores que han tutelado este trabajo, tal y como mencionamos en el capítulo 3, para abordar cada atributo de la usabilidad dentro de las heurísticas que proponemos.

Este proyecto nos ha servido para comprender, fundamentalmente, la complejidad que se esconde detrás del campo de la usabilidad y todos los principios de la transferencia de estado representacional. También, para aprender a estudiar la usabilidad mediante uno de los métodos más usados como es la evaluación heurística.

Cabe destacar que no partimos de una gran experiencia en APIs REST. No obstante, aparte de contribuir con la usabilidad en este tipos de sistemas, podríamos estar presente ante un conjunto de guías que resuman todos los aspectos importantes para comenzar con la imple-

mentación de cualquier API RESTful sin tener en cuenta la experiencia del desarrollador ya que son muy fáciles de aplicar.

Apéndices

Apéndice A

Publicación en un congreso

TRAS haber finalizado este Trabajo de Fin de Grado, hemos preparado un artículo científico sobre el mismo para enviar al congreso *43rd International Conference on Software Engineering* [54], que se celebrará desde el día 23 al 29 de mayo de 2021 en Madrid. Actualmente estamos a la espera de la aceptación del artículo.

Heuristics and Usability Guidelines for RESTful APIs and Application to a Case Study

Nerea Vázquez-Callejón
CITIC
Universidade da Coruña
A Coruña, Spain
nerea.callejon@udc.es

Eduardo Mosqueira-Rey
CITIC
Universidade da Coruña
A Coruña, Spain
eduardo@udc.es

David Alonso-Ríos
CITIC
Universidade da Coruña
A Coruña, Spain
dalonso@udc.es

***Abstract*—According to ISO/IEC 9126, usability refers to a software's capability for being understood, learned, used, and attractive to the user under specific conditions. Therefore, ensuring usability should be a priority in software development. The main goal of this paper is to develop a set of usability heuristics and guidelines for designing RESTful APIs. This is done through a comprehensive analysis of REST principles using an extended, general-purpose usability model as a guide. The efficacy of these heuristics and guidelines was tested with a case study, namely, the REST API for Twitter developers. This usability study consisted in a heuristic evaluation, which is a technique introduced by Nielsen and Molich in 1990. Prior to this, it is necessary to characterize the REST API's context of use according to the characteristics of the users, the tasks and the environments. The goal is to gain a better perspective on the problems found during the usability study.**

***Index Terms*—Usability, API, RESTful, Development of heuristics and guidelines, Heuristic evaluation**

I. Introduction

The term REST (Representational State Transfer) is a type of software architecture for distributed hypermedia systems that describes any interface between systems for obtaining or executing operations on data. It was introduced by Roy T. Fielding, one of the main contributors to the Hypertext Transfer Protocol (HTTP) in 2000 [1].

An Application Programming Interface (API) is set of definitions and protocols that are used to develop and integrate the software of one or many applications.

Currently, many companies offer a RESTful API (RESTful is synonymous with REST) to extend the services it offers to other companies or developers. This is borne out by the websites for prominent corporations such as Google, Facebook or Twitter, who are developing REST services to facilitate access to their valuable data resources, adding value to their businesses.

If the users of these APIs want to perform their tasks with effectiveness and efficiency, it is necessary to ensure a good level of usability. In order to pursue this goal, we propose a set of heuristics and guidelines for a RESTful API, using a hierarchical usability taxonomy [6] as a guide to structure the process and detect deficiencies.

The resulting heuristics and guidelines will be subsequently tested with a practical case study. In particular, a REST API for Twitter developers. A heuristic evaluation will be conducted, which will identify parts of the API that do not comply with good practices in usability.

A. Representational State Transfer (REST)

Roy T. Fielding's PhD research on the internet [1] introduced the Representational State Transfer (REST) paradigm, with the aim of guiding the design and development of an architecture for the modern web. This type of architecure consists mainly in the following: a **stateless client-server architecture**, that is, every request to the server contains all the necessary information for processing the request; **cache storage**, so that the client's cache memory can reuse the data from previous responses for future, equivalent requests; a **uniform interface** defined by four restrictions, namely, resource identification, manipulation of resources through representations, the **HATEOAS** (Hypermedia as the Engine of Application State) principle; and a **layered system** to provide additional functionalities, such as balance of load or shared caches.

The REST architecture is defined by elements such as: **resources**, which can be documents, images, a collection of other resources, a service, and so on. A **Uniform Resource Identifier (URI)**, which is used to indentify a particular resource in a client-server interaction, and **representations**, which consist of data, hypermedia links that take the clients to next desired state of the resource, metadata to describe the data and, occasionally, additional metadata for verifying the integrity of the message. The data format of a representation is known as media type.

This type of architecture separates client-server responsibilities avoiding the problem of server scalability and, via a generic interface, allows the encapsulation and the evolution of the services.

The restrictions that comprise REST only establish high-level design guidelines. Therefore, there is no rule to enforce the use of a specific protocol for transferring information. Currently, this paradigm is used to describe any interface between systems that directly uses HTTP, as this protocol is often use for deploying web services consisting in a collection of open, standard protocols that are used for exchanging data between applications or systems.

II. State of the art

A. Heuristic Evaluation

Heuristic evaluation is a usability study technique with no users. It consists in expert evaluators assessing a system's quality in use based on a set of heuristics, which are essentially rules of thumb.

This technique was introduced by Nielsen and Molich [2] in 1990 in order to evaluate user interfaces.

They initially came up with nine usability heuristics that were subsquently revised by Nielsen to add a new one about *help and documentation*. He also changed the names of the heuristics while retaining their meanings and descriptions.

Nielsen's heuristics are meant to be applicable to a wide range of software interfaces but, nevertheless, many authors chose to create different heuristics for particular contexts, or tried to improve them in some way to suit specific purposes.

Some of these authors, such as Toribio-Guzmán et al. [3], extended Nielsen's heuristics to assess the usability of a private social network that monitored the daily progress of patients by their relatives. Or Mirel and Wright [4], who adapted it to the field of bioinformatics to ensure a number of characteristics for several tools.

On the other hand, Kölling and McKay [5] proposed a new list of heuristics to evaluate usability in programming tools for beginners, in order to offer useful information to the developers of these types of systems.

Along these lines, we have developed a set of usability heuristics and guidelines with the aim of conduction a heuristic evaluation in the domain of RESTful APIs. In order to produce them, we have followed the extended usability model proposed by Alonso-Ríos et al. [6], which describes the concept of usability comprehensively and hierarchically. This taxonomy will be explained in the next section.

"Extended usability models"–a term introduced by Lewis [11]–appeared because traditional, well-known usability models like the one proposed by Nielsen in 1993 [7], or those of the International Organization For Standardization (ISO/ IEC 9126-1, 2001, ISO 9241-11, 1998), have been criticized for not being detailed enough to cover all the facets of usability. Therefore, researchers like Seffah et al. [8], Winter et al. [9], Bevan [10] and Alonso-Ríos et al. [6] tried to create their own models to encompass all the existing definitions of the term. Figure I shows a comparison of several of these extended models and the traditional models. It can be seen that all the concepts are closely related even though they may occasionally seem different.

B. Usability Model

The heuristics and guidelines that were obtained in this research have been derived from an expanded usability model [6] that aims to synthesize all the usability attributes from the literature without redundancy. It is a general-purpose model and it is also hierarchically structured into levels of detail [12].

The taxonomy has six main usability attributes, which are further broken down into subattributes (Fig.1). The attributes are:

TABLE I
COMPARISON OF TRADITIONAL USABILITY MODELS WITH SOME OF THE "EXTENDED USABILITY MODELS" [11]

Nielsen (1993)	ISO (1998)	ISO/IEC (2001)	Seffah et al.(2006)	Alonso-Ríos et al.(2009)
Learnability	Effectiveness	Understandability	Learnability	Knowability
Memorability	Efficiency	Learnability	Effectiveness	Operability
Efficiency	Satisfaction	Operability	Usefulness	Efficiency
Errors		Attractiveness	Accessibility	Robustness
Satisfaction		Usability compliance	Universality	Safety
			Efficiency	Subjective satisfaction
			productivity	
			Safety	
			Satisfaction	

- **Knowability**: defined as the capability of being understood, remembered and learned. It is subdivided into CLARITY, CONSISTENCY, MEMORABILITY AND HELPFULNESS. All but the last one are applicable to formal and conceptual elements, and to the functioning of user tasks and system tasks.
- **Operability**: the capability of providing and adapting to the functionalities the users need. It is subdivided into COMPLETENESS, PRECISION, UNIVERSALITY AND FLEXIBILITY.
- **Efficiency**: the capability of giving good results in return for the resources invested. It is subdivided into EFFICIENCY regarding human effort, task execution time, occupied resources and economic costs. Each of these is broken down into further subattributes (e.g., physical and mental effort, material or human resources, etc.).
- **Robustness**: the capability of being resistant to error and adverse situations. It is subdivided into ROBUSTNESS to internal error, to incorrect uses from the user, to abuse from third parties and to environment problems.
- **Safety**: the capability of avoiding risk and damage. It is subdivided into SAFETY for the user, for third parties and for the environment, The first two are further broken down into BODILY SAFETY, LEGAL SEGURIDAD, CONFIDENCIALITY AND SAFETY OF MATERIAL ASSETS.
- **Subjective satisfaction**: the capability of being pleasurable to use and interesting to the users. It is subdivided into INTEREST and AESTHETICS. The latter is applicable to the five senses (i.e., VISUAL, ACOUSTIC, TACTILE, OLFACTIVE AND GUSTATIVE).

Usability always depends on the specific context of use (ISO 9241–11). That is, the particular characteristics of the users, tasks and environments. For example, consider an educational tool for managing a course. The students submit tasks and the teachers grade them. The usability of the tool will be different for each type of user, as they play different roles and perform different tasks with different goals.

The usability model described above is therefore complemented with an additional context-of-use taxonomy [13] based on the same principles.

The main attributes are USER, TASK AND ENVIRONMENT. Again, these attributes are further broken down into subattributes, as shown in Fig. 2.

C. Field of application: RESTful APIs

Even though RESTful APIs have become increasingly relevant, there is a dearth of formal information on how to implement them with correctness. It is important to note this because, in contrast with traditional APIs, a REST API is language-independent and its design is not standardized.

There are however some authors who have guided us into adopting a theoretical perspective of the main principles for the design and implementation of REST APIs. Firstly, Amodeo [14] suggested some preferences for designing and implementing CRUD (*Create, Read, Update and Delete*) operations on data and the use of hypermedia in a REST API, focusing also on how to design the URI for accessing resources based on the volume of data.

The approach followed by Massé [15] is to propose rules on design principles. The author defines a number of rules for each element involved in a REST API. These rules are very diverse, including: the design of URIs and their query parameters; design principles for typical customer concerns such as safe OAuth authorization for accessing resources; metatata design in headers; HTTP interactions; and the representation of resources and errors in responses.

Another trend in the research for implementation guidelines is identifying a set of design patterns and anti-patterns. In particular, Palma et al. [16] [17] [18] defined a process for their detection in RESTful APIs. The proposed patterns include items such as using hierarchies between nodes in the URI, or avoiding disorganized, hard-to-read URIs. Analogously, the proposed anti-patterns include, for exaple, forgetting hypermedia as part of the representation of a resource, or ignoring cache storage in the response to a request.

In consequence, the aim of this paper is to propose a way to cover all the relevant usability aspects for REST APIs based on the design principles that are gathered in the literature, which typically takes the form of the examples outlined above. To our knowledge, no published document has explicitly tried to offer a systematic and comprehensive view of the whole range of usability criteria for RESTful APIs. Nevertheless, we should mention advances such as the research by Murphy et al. [19], which analyzes design guidelines for famous companies, comparing and contrasting each one in terms of quality and usability. Or the work of Yamamoto et al. [20], which proposes a quality model circumscribed to the concept of learning capability and stability to change–from the point of view of the user–in order to validate it later through an empirical study on the usability in several APIs.

III. METHODOLOGY

Our methodology consists of two independent tasks. Firstly, we develop a set of heuristics and guidelines guided by an expanded usability model. Secondly, we use them in a heuristic evaluation for a real-world case study. The construction process of the heuristics and guidelines is systematic and generalizable, which means that it can be applied to any domain, even when there is little published work, as with RESTful APIs. The goal of the case study is to test the methodology empirically.

Both the construction the heuristics and the usability evaluation are inspired by the methodologies followed in the work of Alonso-Ríos et al. [12] [21], which generally consists in **characterizing the system** in order to identify the elements that are relevant to the user tasks and the system; conducting an **extensive literature review**; **mapping the documentation to the usability taxonomy**; **transforming the information into heuristics and guidelines**, **characterizing the context of use** to identify which aspects to focus on for specific scenarios; and finally, to perform a **heuristic evaluation using the previously constructed heuristics**. The next few subsections describe in detail these steps.

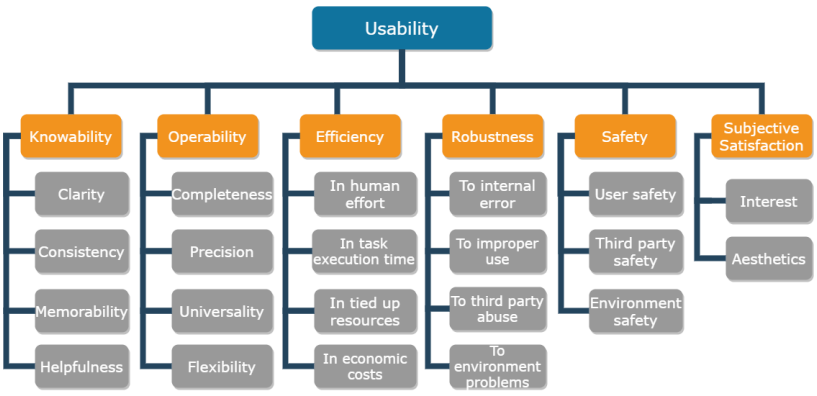

Fig. 1. Usability taxonomy, first level

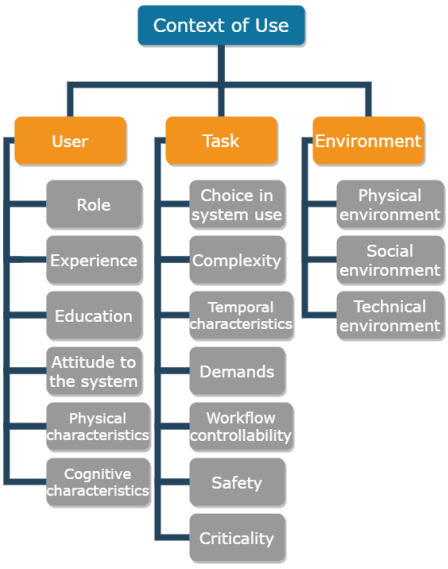

Fig. 2. Context of use taxonomy, first levels

A. Development of usability heuristics and guidelines

The first step is to characterize the system by means of a taxonomy of the elements that must be analyzed. This taxonomy is specific to this field of application and consists of the elements involvedm in an HTTP request and response. That is, the main parts of a REST API. The process has been analytical, breaking down each element into several sub-levels of detail until the taxonomy is specific enough to be mapped against the usability taxonomy. Along the way, we prune the branches of the usability taxonomy that are not relevant to this field of application (for example, some attributes of the taxonomy refer to the five senses–hearing, etc.).

Next, we conduct an extensive literature review. As mentioned, the literature is mainly focused on things like design principles, good practices, pattern and anti-pattern matching, and so on. We try to fit all this information into the attributes of the usability taxonomy and the components of the taxonomy of elements. For example, a recurring rule in the literature is to avoid file extensions and formats as parts of the URI, so we match this rule with *"Knowability - Formal clarity - URI"*, and we repeat this process until we have covered all the available literature, checking that all the rules can be mapped into usability criteria.

The next step is to translate these results into explicit, more easily readable heuristics and guidelines. According to Nielsen's definitions, a heuristic is a rule of thumb, whereas guidelines can range from very specific aspects to general principles. Also, if we end up with multiple guidelines categorized into *"Knowability - Formal clarity - URI"*–such as *"do not use underscores, capital letters, unicode, file extension or formats"*–these can be further grouped into a more general heuristic: *"URIs must be clearly visible"*. Finally, if we find out that there is a usability attribute from the taxonomy that is not explicitly addressed by the literature, we consider the possibility of filling this gap by extrapolating information from literature into new heuristics and guidelines.

B. Heuristic evaluation methodology of a case study

So far, we have obtained a set of heuristics and guidelines. However, in order to use them for a specific case study and be able to interpret the results, we need to analyze the intended context of use. That is, the characteristics of the users, the tasks and the environment are more relevant to this setting. In order to do this, we use the context-of-use taxonomy in Fig. 2.

We can them perform the heuristic evaluation, which checks whether the elements under study comply with basic usability principles. For example, for the previously mentioned heuristic, that is, *"URIs must be clearly visible"*, which also corresponds to the previously mentioned guidelines about underscores, capital letters, and so on, the evaluation detects whether these criteria are met completely, partially or not at all.

IV. Design of usability heuristics and guides

A. System characterization

REST is not linked to any communication protocol, but most of the time it is implemented over HTTP, since its author defined it with this protocol in mind, taking into account its strengths. Fig.3 shows the taxonomy of elements of an HTTP request and response in accordance with the W3C standard [22], which determines the behavior of a REST API, and therefore, has served us to detect which elements of the system are significant for the usability study and to develop our heuristics and guidelines with accuracy. It should be noted that we also added the documentation and version of the API as these are two important aspects for the user.

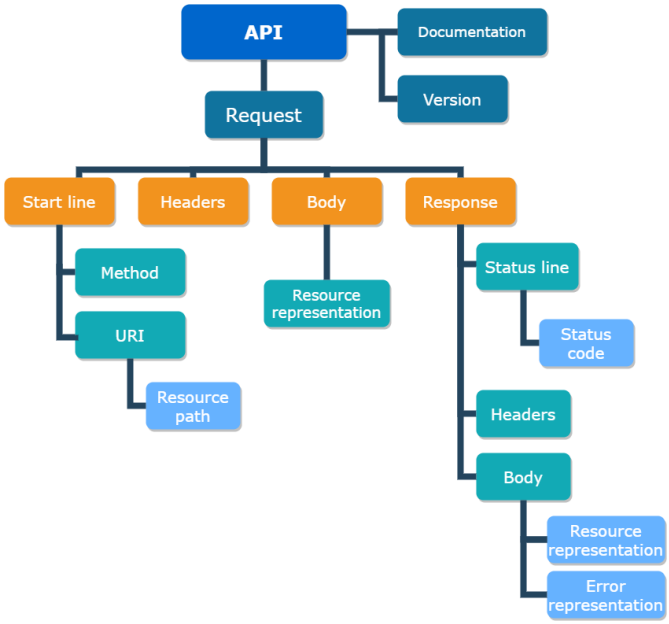

Fig. 3. Taxonomy of elements of the case study

B. Bibliographic analysis and mapping between usability taxonomy and RESTful API literature

Prior to the bibliographic study, we pruned the branches of the usability taxonomy (Fig.1) that we considered not relevant or that did not fit in our case, as we mentioned in the methodology. For example, for the attribute of subjective satisfaction, which is divided into aesthetics and interest, we eliminate the latter because the user's interest does not depend on the system, but on his or her own needs.

With respect to bibliographic analysis, it generally deals with design principles and rules, good practices and recommendations, as mentioned in the state of the art. One of the problems we have encountered with this study is that many authors propose different REST principles to achieve the same goal.

As we mentioned in previous sections, there is no explicit bibliography on usability in this type of API, so we considered it appropriate to start classifying information from an extensive

source, covering all the important elements on the client and server side, within the mapping between the usability taxonomy (Fig. 1) and system elements (Fig. 3), as explained in the methodology, and thus ensure usability through REST fundamentals.

We began to examine and classify each design guide proposed in the Massé work [15], which is composed of an extensive list of rules covering the design of resource identifiers, interactions with HTTP, metadata provided through headers, representations of resources within the body of the message, and solutions to address the client's concerns. The result of this first approach is a solid and consistent basis of guidelines, which allows us to focus correctly on their future distribution across taxonomies.

We then proceed to abstract more guides from the bibliography, so that we support those that have been previously seen and complete all the attributes of usability. We can take as an example the case of the attribute 'Helpfulness - Suitability of documentation content - API documentation', where we are concerned with finding guidelines related to API documentation since in the research study by Sohan et al. [23] they found that developers of REST API clients face productivity problems when the documentation lacks usage examples.

Other studies that have served to complete the development of guides are those that explain how to implement a REST server, showing lower level aspects. Among these, we can highlight the works of Allamaraju [24], Giessler et al. [25] or Wilde and Pautasso [26], among others.

Also, we took into account studies such as those by Rodriguez et al. [28], which propose a solution based on an API REST and analyze a specific context to study whether it meets style requirements.

C. Information coding in the form of heuristics and guides

Once we had all the guidelines distributed throughout the mapping between the usability and system taxonomy, we observed that all the criteria and recommendations that were proposed by various authors could be successfully assigned to one attribute, and even to several. This shows that the taxonomy is comprehensive enough to encompass all the information in the REST literature, even though the literature is extremely specialized and the usability taxonomy is general in purpose.

The next step is to unify guidelines for those usability attributes covered by several authors, or take them directly from the literature, and resolve contradictions. Several authors semantically proposed the same recommendation to cover a REST principle, but the examples they use were different, so we chose those that were most repeated among the studies analyzed and that we considered to be more complete and clearer to understand.

For example, we can cite the case of how to build a parameterized query within the URI to sort the results. Many authors did not consider it important to add this functionality within the API and most of them did it, as it comes in the CMF-1 heuristic (Table III). Another alternative was proposed by several specialized blogs in which the URI parameter attribute was written as follows `sort_by=asc(email)` or `sort_by=email&order_by=desc`. We decided to discard the latter proposal as it seemed to be a rather cumbersome way of constructing the query.

Some of the discrepancies we encountered were about where and how to display the current version of the API. We finally chose to make it part of the resource path within the URI, as this indicates the current version of the resource set, and thus saves the client a lot of confusion. In addition, specifying it within a query parameter or through a custom header in the request and response, leads to an increase in traffic and computational cost as there has to be logic to discriminate the different supported versions.

For the attributes that have not been covered by the literature, or that have been left a little short of information, we complete them using the methodology discussed in the study that has served as inspiration for this work [21], and with additional information outside the REST literature. For example, to complete the guides about security, we took information from several blogs that tried to avoid securitization attacks on web APIs.

Finally, for each set of guides, we assign a heuristic that summarizes them in a general way. For example, for the set of guidelines focused on task execution time efficiency, such as responding to a request, we summarize them all in the following heuristic: *Specific metadata should be provided in the header to help process and reduce the latency of the response to a request.*

It should be noted that some of them have been created without being supported by any bibliographic resource, as in the case of the CCC-2 and CCF-2 heuristics.

D. Results

In the tables II, III, IV, V and VI we show the final result of our work. We also include examples and counter-examples for those heuristics but for space reasons we do not include them in this paper. As a sample, for the CCC-2 heuristic ("*The basic structure of a resource path should be easy to understand*") an example will be: `/profiles?sort=-education&fields=name` and a counter-example: `/profiles/sort/desc/name`.

TABLE II
USABILITY HEURISTICS AND GUIDES (PART I)

Code	Heuristics	Guides	Ref.
Knowability - Conceptual clarity - URI			
CCC-1	The elements of the URI should be easy to understand.	- Do not use a slash at the end. - Use the slash to indicate a hierarchical relationship between resources. - Use semicolons to indicate non-hierarchical elements. - Use *ampersand* to separate parameters within the URI. - Do not use abbreviations.	[27] [16] [28] [24] [30]
CCC-2	The basic structure of a resource path should be easy to understand.	- Use filtering, sorting, field selection or paging parameters in the resource path to make its structure as clear and orderly as possible.	
Knowability - Formal clarity - URI			
CCF-1	URIs must be clearly visible.	- Use dashes. - Do not use underscores. - Do not use capital letters. - Do not use unicode. - Do not include the file extension or format.	[24] [28] [15] [16]
CCF-2	The basic structure of a resource path must be clearly visible.	- If you have a resource that consists of many fields, you must create predefined views or queries of the fields that the customer may request most frequently. This way, we avoid very long URIs. - Do not use very large resource hierarchies.	
Knowability - Conceptual consistency - Resource path			
CCC-3	Terms used in a resource path must be consistent with the HTTP method of the request.	- Use a singular noun to represent a resource and a plural noun to represent collections. - Do not use CRUD terms. - For PUT and DELETE requests, the last node of the URI of the request must be singular, for POST it must be plural. - Use verbs or verb phrases to represent controllers over a resource	[16] [30] [15] [17] [25] [28]
Knowability - Conceptual consistency - URI			
CCC-4	The URI must have a consistent structure when identifying hierarchically related resources.	- Two URIs must be used to specify a resource. The first one to determine the set of states, that is, a specific collection, and the second one to select a state within that collection. - In the event that it is not necessary to specify a collection, it is sufficient to specify a particular resource.	[25] [16] [17]

TABLE III
USABILITY HEURISTICS AND GUIDES (PART II)

Code	Heuristics	Guides	Ref.
Knowability - Conceptual consistency - Body of the message			
CCC-5	The body of the request and response should be represented by consistent formats.	- Use consistent MIME types: HTML, XML, JSON or Atom. - JSON objects must be well formed. - If the message body consists of a composite object, use the MIME type *Multipart/Related*.	[26] [15] [31] [25] [27]
Knowability - Formal consistency - URI			
CCF-3	It should be easily perceived that the URI refers to an API with its corresponding version.	- The URI must identify the owner of the service and have a subdomain called api. - If an API provides a portal for developers, the URI must have a subdomain labeled as developer. - Specify the API versions in the resource's URI, in `Vx` form, with `x` being the version number.	[15] [31] [24] [32] [45]
Knowability - Formal and conceptual memorability - URI			
CMF-1	The querying terms and parameters of the URI should be easy to use and remember. URIs should be as explanatory as possible.	- For filtering, use short, resource-related parameters. - For sorting, use the "`sort`" parameter, followed by a list of attributes prefixed with "+" or "–" for an ascending and descending order. - For field selection, use `fields` as parameter. - For paging, use the `offset` and `limit` parameters. - If the complexity of a client's paging or filtering exceeds the simple formatting capabilities of the query part within the URI, consider designing a controller resource. - Use descriptive terms in URIs.	[32] [25] [15] [33] [27]
Knowability - Helpulness Suitability of documentation content Representation of errors			
CAI-1	The API should display useful error information when there is a problem in the request.	- The body of the response message should contain: The body of the response message should contain: status code, internal code that identifies it within the API documentation, a message to the developer describing the cause of the error and how to solve it, a message to show to the user, and a hyperlink to get more information about the problem. - Do not return success code when an error is returned in the body of the response. - Do not show obsolete information.	[15] [31] [25] [24] [12]

TABLE IV
USABILITY HEURISTICS AND GUIDES (PART III)

Code	Heuristics	Guides	Ref.
Knowability - Helpfulnes Suitability of documentation content API Documentation			
CAI-2	The documentation should define and describe each element of the API and show examples of use.	- Define standard and recognized MIME types, status codes and HTTP methods. - Define each resource with its attributes and link relationships. - Define each HTTP operation supported for each resource and query parameters. - Define how to use authentication/authorization credentials. - Show examples of requests, responses and queries parameterized in the URI. - Show examples implemented in the most used languages of the domain. - Do not show redundant or obsolete information.	[31] [25] [24] [27] [12]
Operability - Completeness - Resource representation			
OC-1	The representation of the resource in the response must be accompanied by a set of links, formed by those URIs that are related to it.	- Within the representation of the resource there must be a section called *links* formed by URIs that lead to resources related to the request, i.e., the API must comply with the HATEOAS principle.	[18] [28] [15] [34] [27] [29]
Operability - Precision - Methods			
OP-1	HTTP methods should be used correctly in a request.	- GET: to recover a resource. - PUT: to insert a new resource or update/replace already created mutable resources. - POST: to create a new resource within a collection or to model operations with side effects that do not fit with other methods. - DELETE: to remove a resource. - OPTIONS: to retrieve metadata describing the available interactions of a resource. - HEAD: to retrieve metadata from the current state of a resource.	[15] [31] [26] [30] [34] [25] [28]
Operability - Precision - Status code			
OP-2	Use HTTP status codes that most closely match the response of a request.	- Return HTTP status codes. - To be as accurate as possible, do not use only one success code and one error code.	[15] [31] [26] [30] [25] [34]
Operability - Cultural universality - Cultural conventions - Message body			
OUC-1	The API should avoid the use of elements (terms, formats, spelling, etc.) that are not universally recognized and be written in the language most used by IT.	- Use universal expressions that are not limited to a particular language. - Use standard terms. - Use UTF-8 character encoding system. - Implement the API in English.	[12]
Operability - Flexibility - Asptiveness to environments - API			
FAA-1	When a new version of the API is released, it must be compatible with the previous one for a period of time.	- A new version should coexist with the previous one for a minimum period of time so that customers can adapt correctly.	[31]

TABLE V
USABILITY HEURISTICS AND GUIDES (PART IV)

Code	Heuristics	Guides	Ref.
Operability - Flexibility - Controllability Technical configurability - Resource representation			
FCC-1	The API must be able to represent a resource in several formats. The API must provide partial status of resources.	- Customers should be allowed to negotiate the format of the message body using the *Accept* header. - Partial requests for the status of a resource must be admitted by means of parameterized queries.	[15] [31] [29] [18] [25] [28] [34] [32]
Efficiency - Efficiency in task execution time Responding to a request - System response			
EER-1	Specific metadata should be provided in the header to help process and reduce the latency of the response to a request.	- The header of the response must contain *Content-Length* and *Content-Type*. - When a resource has just been created, the response header must contain *Location* to specify the URI of the newly created resource. - Use *Cache-Control, Expires* and *Date* in the response header to encourage caching, as long as the resources do not have confidential or personal client information or the information does not change too frequently - If you do not want to use the cache for the responses, instead of using *Cache-Control:no-cache* use a very small value of *max-age*. - Use caching headers with expiration time for successful responses to GET and HEAD requests and for redirects and client errors, i.e. 3xx and 4xx responses. - In the header of the responses to GET requests, use *ETag* or *Last-Modified*.	[15] [30] [18] [34]
Efficiency - Efficiency in task execution time Responding to a request - User actions			
EEA-1	Provide views of more common queries as resources.	- Create predefined queries, filters or views on a collection.	[14]
Efficiency - Efficiency in economic costs Consumables, human resources and system resources - API			
EEC-1	The economic costs for the use of the API, if applicable, should be reasonable.	- The cost of the license must be adequate for the services offered and be in line with market prices.	[12]
Robustness - Robustness to user improper use. Request and Response			
RU-1	The API should make correct use of the HTTP protocol.	- Do not use square brackets when we have fields from an array in a parameterized query in the URI. - Custom headers should be for informational purposes. If the information being transmitted through a custom header is important for interpreting the request or response, it should be included in the body of the message.	[15] [35]

TABLE VI
USABILITY HEURISTICS AND GUIDES (PART V)

Code	Heuristics	Guides	Ref.
Safety - User and third party safety - API User confidentiality			
SUC-1	The API must not compromise the confidentiality of clients' personal information.	- The API should not access data stored on the user's machine that is not necessary for its operation or provide personal data to third parties without consent.	[12]
SUC-2	The API must provide secure connections between the clients and the server.	- Use OAuth 2.0 as an authorization method and token-based authentication. - Use TLS. - If there is confidential information in the response, use *Cache-control:private* in the header to prevent the cache from being shared with other clients. - Do not show confidential data in the URI. - Configure the server to handle HTTP requests only.	[15] [31] [32] [24] [36] [27]
Safety - User and third party safety - API Legal safeguarding			
SUL-1	The API should not put the user in legal trouble.	- Do not allow access to protected data, such as copyrighted content, confidential content, etc.	[12]
SUL-2	The API should clearly indicate its license of use..	- MShow what type of license the API has.	[12]
Safety - Environment safety - API			
SE-1	The API must be implemented in a secure environment.	- Use reverse proxy or a client-server gateway. - Monitor and set a limit for use. - The design should not be coupled with the implementation of the service. - Validate the MIME types of the request's *Content-Type* header. - Do not allow the header *Accept* to come empty from a request. - If our API exposes only the GET method on a resource, the request should not be allowed to contain headers like: *X-HTTP-Method-Override* or *X-Method-Override*, as it may overwrite the method. - The server must send the header: *X-Content-Type-Options* with value *nosniff* to ensure that the browser does not try to detect other MIME types different from the value of *Content-Type* and thus avoid possible XSS attacks. - When the API returns information about the cause of an error, these explanations should not be too explicit. - Protecting privileged actions on resources against unauthorized use.	[15] [29] [37] [38] [39]
Subjective satisfaction - Visual esthetics - API			
SSE-1	The API must be attractive enough for the user to find it comfortable to use.	- The API should not be aesthetically unpleasant, such as having strange names, using special characters inappropriately, etc.	[12]

V. HEURISTIC EVALUATION IN A CASE STUDY

A. Context of use characterization

To find the characteristics of the context of use and study how users, tasks and environment really behave within our case study, and thus know what aspects are important for our usability study, we go through the context-of-use taxonomy (Fig.2).

Regarding the user, who in our case would be a software developer who interacts with the system, some of the most important aspects we have found regarding the context of use have been the ACADEMIC TRAINING AND KNOWLEDGE OF THE SYSTEM DOMAIN.

As for the PHYSICAL AND COGNITIVE CHARACTERISTICS we will focus particularly on abilities and disabilities related to sight, reasoning and memory, while for the tasks performed by users within the system, which consist of building HTTP requests to operate on the data exposed by the API and interpret and manipulate correctly the data returned in the responses, we consider their COMPLEXITY relevant. It should also be mentioned that these tasks should not abuse the user's COGNITIVE ABILITIES and should provide legal security and confidentiality.

Finally, regarding the technical environment, which concerns the server that processes each request and response, its important characteristics are related to the SUITABILITY OF THE LOGIC DEVICES, in particular the EFFICIENCY, ROBUSTNESS AND SECURITY as they play an important role within the system environment.

B. Heuristic evaluation

To validate our set of heuristics and guidelines, we proceeded to apply them to the Twitter REST Developer API to perform a heuristic evaluation and study its usability. The API allows access to parts of its service for developers to integrate their software with that platform.

We have analyzed a set of requests and answers that belong to the groups of *Accounts and users* and *Tweets and answers* that allow to follow, search and obtain information from users [40], and to create, recover and delete Tweets, Retweets and give *Like* to the previous ones [41].

In the tables VII and VIII we will show the results of the usability study with each heuristic identified by its code assigned in the previous section, and classified according to three categories: the guidelines are completely fulfilled ("***"), partially fulfilled ("**") and barely fulfilled ("*").

In view of the results of the heuristic evaluation, we saw that the usability attributes most affected were those pertaining to knowability and operability, that is, what defines how easy it is to understand and learn how to use it, and how it provides the functionalities that developers expect from it. As far as efficiency is concerned, there were no relevant problems, but it should be pointed out that the heuristics that ensure usability with respect to this attribute were not completely fulfilled.

These usability problems mainly affect the user's reasoning and the complexity of the tasks within the context of use.

TABLE VII
HEURISTIC EVALUATION (PART I)

Code		Heuristic evaluation
Knowability		
Conceptual clarity		
CCC-1	***	The elements of the URI are easily understood.
CCC-2	***	The structure of the resource path is easily understood.
Formal clarity		
CCF-1	*	URIs are not clearly visible because they use the format extension and underscores.
CCF-2	***	Resource paths do not use very long hierarchies, so their structure can be clearly visualized.
Conceptual consistency		
CCC-3	*	The terms used in the URIs are not consistent with the HTTP method of the request. It uses CRUD terms and abuses the use of resource controllers.
CCC-4	**	Hierarchical relationships between resources have a consistent meaning, but abuses of controllers over resources.
CCC-5	***	Consistent MIME types are used to represent the body of the request and response.
Formal consistency		
CCF-3	**	The version of the API is not easily perceived, it appears in ambiguous form.
Memorability		
CMF-1	***	Uses descriptive terms and easy-to-remember resource controllers. The paging parameters are easy to use.
Helpfulness		
CAI-1	*	The error information when there is a problem in the application is scarce. Only one message is displayed to describe the error.
CAI-2	**	The documentation is sometimes a little incomplete. It does not define each resource with its attributes and some requests do not have examples of responses and possible status codes.
Operability		
Completeness		
OC-1	*	Resource representations are incomplete because there are no URIs to actions and related resources.
Precision		
OP-1	*	HTTP methods are not used accurately when you want to remove and update a resource. In several GET requests it is recommended to use POST if the request is very long.
OP-2	**	Sometimes the status codes are not accurate with the response.

TABLE VIII
HEURISTIC EVALUATION (PART II)

Code		Heuristic evaluation
Operability		
Cultural universality		
OUC-1	***	The API can be used by users of different cultures because it uses standard terms, the UTF-8 character encoding system and is implemented in English.
Flexibilidad		
FAA-1	***	The API allows developers to adapt when there is a new version. In addition, it indicates on what dates it is available and when it will become obsolete.
FCC-1	*	Developers cannot configure requests to negotiate the format of the message body, nor can they request partial resource status.
Efficiency		
EER-1	**	Headers are used to help process and reduce the response time to a request, but always use *Cache-control:no-cache, no-store*.
EEA-1	***	Developers access data efficiently because the API provides views of more common requests as resources.
EEC-1	***	The API is economically efficient. The premium version varies from 149$/month to 2,499$/month, depending on the level of access required.
Robustness		
RU-1	***	The custom headers in the responses are for informational purposes, so the API is robust to developers' misuse of the HTTP protocol.
Safety		
SUC-1	***	The API does not compromise the confidentiality of personal information.
SUC-2	***	The API provides secure connections to developers, as it does not display confidential data in the URI, uses TLS, avoids sharing the cache with other clients by using *Cache-control:private* in the response header, uses OAuth as the authorization method and token-based authentication.
SUL-1	***	The API does not allow developers to access confidential content.
SUL-2	***	It shows what requirements are needed to access each version of the API: standard, premium, business and advertising.
SE-1	***	The API guarantees the security of the server because there is a limit of requests, it does not match its design with the implementation of the service, it uses the 'X-Content-Type-Options' header with the value 'nosniff', it does not return very explicit information involving the implementation of the service when there is an error and it protects privileged actions on resources by means of a session token.
Subjective satisfaction		
SSE-1	***	The API is aesthetically pleasing.

VI. DISCUSSION AND CONCLUSIONS

The aim of this work was to develop a set of usability heuristics and guidelines for the design of RESTful APIs, which were tested empirically with a real-world case study.

In order to arrive at this goal, we firstly conducted an extensive literature review on the usability of REST APIs, since formal documentation on this specific topic is scarce and dispersed.

Based on the documentation we gathered, we created 27 heuristics that we also further complemented with additional guidelines. All of this was explicitly connected to an expanded usability model that tries to organize all the facets of usability hierarchically.

In order to test the validity of the heuristics and the guidelines, we performed a heuristic evaluation for Twitter's REST API for developers. We also examined the intended context of use for this particular field of application, so we could interpret the results more precisely.

Using our heuristics in the heuristic evaluation helped us

to identify many usability problems. Some are related to an incorrect use of the REST style. In particular, avoiding cache storage, using hypermedia in the response's body, and not presenting a stable set of operation on the data. Unfortunately, it is not unsual that the public REST APIs of major companies use the term "REST" when they are in fact simply using the HTTP protocol without restrictions, as demonstrated by Renzel et al. [43]. There are no restrictions which force the use of HTTP as a transfer protocol. As a matter of fact, using other protocols, such as SMTP [44] is correct.

All of this is a consequence of the scarcity of information on technical characteristics, and how disorganized it is, in spite of the current relevance and interest of these systems.

This may be reason enough to explain the dearth of studies and recommendations focused on REST APIs, taking also into account the lack of consensus on the term itself. As explained by Lewis [11], measuring usability is complex for two reasons: usability is not a specific, easily grasped property of a product, and, unfortunately, most usability studies are confidential, as businesses are reluctant to disclose the usability defects of their products.

Finally, and advancing further in our goal to ensure the usability of REST APIs, we should also recommend the use of OpenAPI specifications in order to pursue good designs, complete documentation, and an easy way to distribute our API to the community and our workgroup, as suggested by Deshpande [45].

ACKNOWLEDGMENT

This work has been supported by the regional goverment of Galicia (Xunta de Galicia, Spain) under project ED431C 2018/34 and the Ministry of Science and Innovation of the government of Spain under project PID2019-107194GB-I00.

CITIC as a Research Center of the University System of Galicia is funded by the Ministry of Education, University and Vocational Training of the Xunta de Galicia through the European Regional Development Fund (ERDF) with 80% of the funds from ERDF Galicia Operational Program 2014-2020 and the 20% remaining from the General Secretariat of Universities (Ref. ED431G 2019/01).

REFERENCES

[1] R. T. Fielding and R. N. Taylor, "Architectural Styles and the Design of Network-Based Software Architectures," University of California, Irvine, 2000.

[2] J. Nielsen and R. Molich,"Heuristic evaluation of user interface," 1990.

[3] J. M. Toribio-Guzmán, A. García-Holgado, F. Soto-Pérez, F. García-Peñalvo and M. Franco,"Study of the Usability of the Private Social Network SocialNet using Heuristic Evaluation," pp. 1–5, September 2016.

[4] B. Mirel and Z. Wright,"Heuristic Evaluations of Bioinformatics Tools: A Development Case," Human-Computer Interaction. New Trends pp. 329-.338, 2009.

[5] M. Kölling and F. McKay,"Heuristic Evaluation for Novice Programming Systems," ACM Transactions on Computing Education, vol. 16, pp. 1–30, June 2016.

[6] D. Alonso-Ríos and A. Vázquez-García and E. Mosqueira-Rey and V. Moret-Bonillo, "Usability: A Critical Analysis and a Taxonomy," Intl. Journal of Human–Computer Interaction, vol. 26, pp. 53–74, January 2010.

[7] J. Nielsen, "Usability Engineering," Morgan Kaufmann Publishers Inc., April 1993.

[8] A. Seffah, M. Donyaee, R. Kline and H. Padda, "Usability measurement and metrics: A consolidated model," Software Quality Journal, vol. 14, pp. 159–178, June 2006.

[9] S. Winter, S. Wagner and F. Deissenboeck, "A Comprehensive Model of Usability," pp. 106–122, January 2007.

[10] N. Bevan, "Extending Quality in Use to Provide a Framework for Usability Measurement," pp. 13–22, July 2009.

[11] J. Lewis, "Usability: Lessons Learned ... and Yet to Be Learned," International Journal of Human-Computer Interaction, vol. 30, September 2014.

[12] E. Mosqueira-Rey, D. Alonso-Ríos, V. Moret-Bonillo and D. Álvarez-Estévez, "A systematic approach to API usability: Taxonomy-derived criteria and a case study," Information and Software Technology, vol. 97, pp. 46–63, December 2017.

[13] D. Alonso-Ríos, A. Vázquez-García, E. Mosqueira-Rey and V. Moret-Bonillo, "A Context-of-Use Taxonomy for Usability Studies," Intl. Journal of Human–Computer Interaction, vol. 26, pp. 941–970, October 2010.

[14] E. Amodeo, "Principios de diseño de APIs REST," Leanpub, 2013.

[15] M. Massé, "REST API Design Rulebook," O'Reilly Media, Inc., 2011.

[16] F. Palma, J. Gonzalez-Huerta, M. Founi, N. Moha, G. Tremblay and Y. G. Guéhéneuc, " Semantic Analysis of RESTful APIs for the Detection of Linguistic Patterns and Antipatterns," International Journal of Cooperative Information Systems, vol. 26, no. 2, p. 37, May 2017.

[17] F. Palma, J. Gonzalez-Huerta, N. Moha, Y. G. Guéhéneuc and G. Tremblay, "Are RESTful APIs Well-designed? Detection of their Linguistic (Anti) Patterns," 13th International Conference on Service Oriented Computing, pp. 171–187, November 2015.

[18] F. Palma, J. Dubois, N. Moha and Y. G. Guéhéneuc, "Detection of REST Patterns and Antipatterns: A Heuristics-Based Approach," 12th International Conference on Service Oriented Computing, pp. 230–244, November 2014.

[19] L. Murphy, T. Alliyu, M. B. Kery, A. Macvean and B. A. Myers, "Preliminary Analysis of REST API Style Guidelines," 8th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU'2017) at SPLASH 2017, October 2017.

[20] R. Yamamoto, K. Ohashi, M. Fukuyori, K. Kimura, A. Sekiguchi, R. Umekawa, T. Uehara and M. Aoyama, "A Quality Model and Its Quantitative Evaluation Method for Web APIs," 25th Asia-Pacific Software Engineering Conference (APSEC), pp. 598–607, December 2018.

[21] D. Alonso-Ríos, E. Mosqueira-Rey and V. Moret-Bonillo, "A Systematic and Generalizable Approach to the Heuristic Evaluation of User Interfaces," International Journal of Human–Computer Interaction, vol. 34, no. 12, pp. 1169–1182, January 2018.

[22] R. Fielding and J. Reschke (2014, June) Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing [Online] Available: https://tools.ietf.org/html/rfc7230

[23] S. Sohan, F. Maurer, C. Anslow and M. Robillard, "A study of the effectiveness of usage examples in REST API documentation," IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 53–61, October 2017.

[24] S. Allamaraju, "RESTful Web Services Cookbook," O'Reilly, 2010.

[25] P. Giessler, M. Gebhart, D. Sarancin, R. Steinegger and S. Abeck, "Best Practices for the Design of RESTful Web Services," International Conference on Software Engineering, vol. 10, pp. 392–397, November 2015.

[26] E. Wilde and C. Pautasso, "REST: From Research to Practice," Springer, 2011.

[27] J. Hradil and V. Sklenakz, "Practical Implementation of 10 Rules for Writing REST APIs," Journal of Systems Integration, vol. 8, no. 1, pp. 45–54, 2017.

[28] C. Rodriguez, M. Baez, F. Daniel, F. Casati, J. Trabucco, L. Canali and G. Percannella, "REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices," International Conference on Web Engineering, pp. 21–39, June 2016.

[29] R. Fielding, Untangled musings of Roy T. Fielding [Online] Available: https://roy.gbiv.com/untangled/tag/rest

[30] J. Webber, S. Parastatidis and I. Robinson, "REST in Practice - Hypermedia and Systems Architecture," O'Reilly, January 2010.

[31] Open Mobile Alliance, "Guidelines for RESTful Network APIs," 2012.

[32] L. Armentano (2017, September) Buenas prácticas para el Diseño de una API RESTful Pragmática [Online] Available: https://elbauldelprogramador.com/buenas-practicas-para-el-diseno-de-una-api-restful-pragmatica/
[33] Microsoft (2018, January) Diseño de API web [Online] Available: https://docs.microsoft.com/es-es/azure/architecture/best-practices/api-design
[34] H. Brabra, A. Mtibaa, F. Petrillo, F. Merle, L. Sliman, N. Moha, W. Gaaloul, Y. Guéhéneuc, B. Benatallah and F. Gargouri, "On Semantic Detection of Cloud API (Anti)Patterns," Information and Software Technology, November 2018.
[35] Moesif (2019, March) REST API Design Best Practices for Parameter and Query String Usage [Online] Available: https://www.moesif.com/blog/technical/api-design/REST-API-Design-Best-Practices-for-Parameters-and-Query-String-Usage/
[36] S. Vergara (2019, June) ¿Cuál es el mejor método de autentificación en un API REST? [Online] Available: https://www.itdo.com/blog/cual-es-el-mejor-metodo-de-autentificacion-en-un-api-rest/
[37] Red Hat, Seguridad de las API [Online] Available: https://www.redhat.com/es/topics/security/api-security
[38] N. Martín (2020, January) Seguridad en tus APIs: cómo evitar ataques de securización [Online] Available: https://www.paradigmadigital.com/dev/seguridad-apis-como-evitar-ataques-seguridad/
[39] G. Levin (2016, December) Top 5 REST API Security Guidelines [Online] Available: https://blog.restcase.com/top-5-rest-api-security-guidelines/
[40] Twitter, Follow, search, and get users [Online] Available: https://developer.twitter.com/en/docs/accounts-and-users/follow-search-get-users/overview
[41] Twitter, Post, retrieve, and engage with Tweets [Online] Available: https://developer.twitter.com/en/docs/tweets/post-and-engage/overview
[42] A. Neumann, N. Laranjeiro and J. Bernardino, "An Analysis of Public REST Web Service APIs," IEEE Transactions on Services Computing, pp. 1–1, June 2018.
[43] D. Renzel, P. Schlebusch and R. Klamma, "Today's top RESTful services and why they are not restful," 13th international conference on Web Information Systems Engineering, pp. 354–367, November 2012.
[44] SMTP, SMTP API documentation [Online] Available: https://www.smtp.com/smtp-api-documentation/
[45] T. Deshpande (2018, July) RESTful API Design — Step By Step Guide [Online] Available: https://medium.com/better-programming/restful-api-design-step-by-step-guide-2f2c9f9fcdbf

Lista de acrónimos

REST *Transferencia de Estado Representacional*

HTTP *Hypertext Transfer Protocol*

HATEOAS *Hypermedia As The Engine Of Application State*

API *Application programming interface*

URI *Uniform resource identifier*

CRUD *Create, Read, Update and Delete*

XML *eXtensible Markup Language*

JSON *JavaScript Object Notation*

MIME *Multipurpose Internet Mail Extensions*

SDK *Software development kit*

SMTP *Simple Mail Transfer Protocol*

TI *Tecnología de la información*

Glosario

Usabilidad Medida de la calidad de la experiencia que tiene un usuario cuando interactúa con un producto o sistema.

Heurísticas Conjunto de principios que sigue un experto para realizar una investigación.

Evaluación Heurística Técnica que permite examinar la calidad de uso de un sistema por parte de varios evaluadores expertos, a partir del cumplimiento de un conjunto de heurísticas.

Principio de HATEOAS Consisten en que un cliente interactúa con una aplicación de red completamente a través de hipermedia proporcionadas dinámicamente por los servidores de aplicaciones.

Contexto de uso Condiciones en las que un producto o sistema es utilizado, atendiendo principalmente a los factores que influyen en su uso y en el grado de satisfacción de los usuarios.

Bibliografía

[1] J. Lewis, "Usability: Lessons Learned ... and Yet to Be Learned," *International Journal of Human-Computer Interaction*, vol. 30, 09 2014.

[2] R. T. Fielding and R. N. Taylor, "Architectural Styles and the Design of Network-Based Software Architectures," Ph.D. dissertation, 2000.

[3] D. Alonso-Ríos, A. Vázquez-García, E. Mosqueira-Rey, and V. Moret-Bonillo, "Usability: A Critical Analysis and a Taxonomy," *Intl. Journal of Human–Computer Interaction*, vol. 26, pp. 53–74, 2010.

[4] Arquitectura Servicio Web REST. [En línea]. Disponible en: https://exequielc.files.wordpress.com/2012/10/rest-web-service.jpg

[5] J. Nielsen and R. Molich, "Heuristic evaluation of user interfaces," 1990.

[6] J. M. Toribio-Guzmán, A. García-Holgado, F. Soto-Pérez, F. García-Peñalvo, and M. Franco, "Study of the Usability of the Private Social Network SocialNet using Heuristic Evaluation," pp. 1–5, 09 2016.

[7] B. Mirel and Z. Wright, "Heuristic evaluations of bioinformatics tools: a development case," pp. 329–338, 2009.

[8] M. Kölling and F. McKay, "Heuristic Evaluation for Novice Programming Systems," *ACM Transactions on Computing Education*, vol. 16, pp. 1–30, 06 2016.

[9] D. Alonso-Ríos, E. Mosqueira-Rey, and V. Moret-Bonillo, "A Systematic and Generalizable Approach to the Heuristic Evaluation of User Interfaces," *International Journal of Human–Computer Interaction*, vol. 34, no. 12, pp. 1169–1182, 2018.

[10] J. Zhang, T. Johnson, V. Patel, D. Smith, and T. Kubose, "Using usability heuristics to evaluate patient safety of medical devices," *Journal of biomedical informatics*, vol. 36, pp. 23–30, 02 2003.

[11] B. Shneiderman and C. Plaisant, *Designing the user interface: strategies for effective human-computer interaction*, 01 2010.

[12] A. Sutcliffe and B. Gault, "Heuristic evaluation of virtual reality applications," *Interacting with Computers*, vol. 16, pp. 831–849, 08 2004.

[13] K. Baker, S. Greenberg, and C. Gutwin, "Heuristic Evaluation of Groupware Based on the Mechanics of Collaboration," *Lecture Notes in Computer Science*, vol. 2254, 05 2001.

[14] J. Nielsen, *Usability Engineering*, 01 1993.

[15] A. Seffah, M. Donyaee, R. Kline, and H. Padda, "Usability measurement and metrics: A consolidated model," *Software Quality Journal*, vol. 14, pp. 159–178, 06 2006.

[16] S. Winter, S. Wagner, and F. Deissenboeck, "A Comprehensive Model of Usability," pp. 106–122, 01 2007.

[17] N. Bevan, "Extending Quality in use to Provide a Framework for Usability Measurement," pp. 13–22, 07 2009.

[18] D. Alonso-Ríos, A. Vázquez-García, E. Mosqueira-Rey, and V. Moret-Bonillo, "A Context-of-use Taxonomy for Usability Studies," *Intl. Journal of Human–Computer Interaction*, vol. 26, pp. 941–970, 10 2010.

[19] E. Amodeo, *Principios de diseño de APIs REST*. Leanpub, 2013. [En línea]. Disponible en: https://leanpub.com/introduccion_apis_rest

[20] M. Masse, *REST API Design Rulebook*. O'Reilly Media, Inc., 2011.

[21] J. Hradil and V. Sklenakz, "Practical Implementation of 10 Rules for Writing REST APIs," *Journal of Systems Integration*, vol. 8, no. 1, pp. 45–54, 2017.

[22] F. Palma, J. Gonzalez-Huerta, M. Founi, N. Moha, G. Tremblay, and Y. G. Guéhéneuc, "Semantic Analysis of RESTful APIs for the Detection of Linguistic Patterns and Antipatterns," *International Journal of Cooperative Information Systems*, vol. 26, no. 2, p. 37, 2017.

[23] F. Palma, J. Gonzalez-Huerta, N. Moha, Y. G. Guéhéneuc, and G. Tremblay, "Are RESTful APIs Well-designed? Detection of their Linguistic (Anti) Patterns," pp. 171–187, 2015.

[24] F. Palma, J. Dubois, N. Moha, and Y. G. Guéhéneuc, "Detection of REST Patterns and Antipatterns: A Heuristics-Based Approach," pp. 230–244, 2014.

[25] L. Murphy, T. Alliyu, M. B. Kery, A. Macvean, and B. A. Myers, "Preliminary Analysis of REST API Style Guidelines," 2017.

[26] R. Yamamoto, K. Ohashi, M. Fukuyori, K. Kimura, A. Sekiguchi, R. Umekawa, T. Uehara, and M. Aoyama, "A Quality Model and Its Quantitative Evaluation Method for Web APIs," pp. 598–607, 12 2018.

[27] E. Mosqueira-Rey, D. Alonso-Ríos, V. Moret-Bonillo, I. Fernández-Varela, and D. Álvarez Estévez, "A systematic approach to API usability: Taxonomy-derived criteria and a case study," *Information and Software Technology*, vol. 97, pp. 46 – 63, 2018.

[28] Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. [En línea]. Disponible en: https://tools.ietf.org/html/rfc7230

[29] Salarios para empleos de Investigador/a en España. [En línea]. Disponible en: https://es.indeed.com/salaries/investigador-Salaries?period=yearly

[30] Untangled musings of Roy T. Fielding. [En línea]. Disponible en: https://roy.gbiv.com/untangled/tag/rest

[31] S. Sohan, F. Maurer, C. Anslow, and M. Robillard, "A study of the effectiveness of usage examples in REST API documentation," pp. 53–61, 10 2017.

[32] S. Allamaraju, *RESTful Web Services Cookbook.*, 01 2010.

[33] P. Giessler, M. Gebhart, D. Sarancin, R. Steinegger, and S. Abeck, "Best Practices for the Design of RESTful Web Services," *Proceedings - International Conference on Software Engineering*, vol. 10, pp. 392–, 11 2015.

[34] E. Wilde and C. Pautasso, *REST: From Research to Practice*, 01 2011.

[35] C. Rodriguez, M. Baez, F. Daniel, F. Casati, J. Trabucco, L. Canali, and G. Percannella, "REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices," pp. 21–39, 2016.

[36] J. Webber, S. Parastatidis, and I. Robinson, *REST in Practice - Hypermedia and Systems Architecture.*, 01 2010.

[37] O. M. Alliance, "Guidelines for RESTful Network APIs," 2012.

[38] Buenas prácticas para el Diseño de una API RESTful Pragmática. [En línea]. Disponible en: https://elbauldelprogramador.com/buenas-practicas-para-el-diseno-de-una-api-restful-pragmatica/#6-filtrado-ordenaci%C3%B3n-y-b%C3%BAsqueda-en-los-resultados

[39] RESTful API Design — Step By Step Guide. [En línea]. Disponible en: https://medium.com/better-programming/restful-api-design-step-by-step-guide-2f2c9f9fcdbf

[40] Diseño de API web. [En línea]. Disponible en: https://docs.microsoft.com/es-es/azure/architecture/best-practices/api-design

[41] H. Brabra, A. Mtibaa, F. Petrillo, P. Merle, L. Sliman, N. Moha, W. Gaaloul, Y.-G. Guéhéneuc, B. Benatallah, and F. Gargouri, "On Semantic Detection of Cloud API (Anti)Patterns," *Information and Software Technology*, 11 2018.

[42] REST API Design Best Practices for Parameter and Query String Usage. [En línea]. Disponible en: https://www.moesif.com/blog/technical/api-design/REST-API-Design-Best-Practices-for-Parameters-and-Query-String-Usage/

[43] ¿Cuál es el mejor método de autentificación en una API REST? [En línea]. Disponible en: https://www.itdo.com/blog/cual-es-el-mejor-metodo-de-autentificacion-en-un-api-rest/

[44] Seguridad de las API. [En línea]. Disponible en: https://www.redhat.com/es/topics/security/api-security#:~:text=Las%20API%20de%20REST%20utilizan,cifrados%20y%20no%20se%20modifiquen.

[45] Seguridad en tus APIs: cómo evitar ataques de securización. [En línea]. Disponible en: https://www.paradigmadigital.com/dev/seguridad-apis-como-evitar-ataques-seguridad/

[46] Top 5 REST API Security Guidelines. [En línea]. Disponible en: https://blog.restcase.com/top-5-rest-api-security-guidelines/

[47] Twitter API reference index. [En línea]. Disponible en: https://developer.twitter.com/en/docs/api-reference-index

[48] Información sobre las APIs de Twitter. [En línea]. Disponible en: https://help.twitter.com/es/rules-and-policies/twitter-api

[49] Follow, search, and get users. [En línea]. Disponible en: https://developer.twitter.com/en/docs/accounts-and-users/follow-search-get-users/overview

[50] Post, retrieve, and engage with Tweets. [En línea]. Disponible en: https://developer.twitter.com/en/docs/tweets/post-and-engage/overview

[51] Postman. the Collaboration Platform for API Development. [En línea]. Disponible en: https://www.postman.com/

[52] D. Renzel, P. Schlebusch, and R. Klamma, "Today's top RESTful services and why they are not restful." pp. 354–367, 11 2012.

[53] SMTP API documentation. [En línea]. Disponible en: https://www.smtp.com/smtp-api-documentation/

[54] 43rd International Conference on Software Engineering. [En línea]. Disponible en: https://conf.researchr.org/home/icse-2021