ULPGC Universidad de Las Palmas de Gran Canaria | Escuela de Ingeniería Informática

Diseño e implementación de un cuadro de mando para la gestión de aforos en la Playa de las Canteras

Titulación:
Grado en Ingeniería Informática
Autor:
Álvaro Manuel Lorenzo Domínguez
Tutor:
Jose Juan Hernández Cabrera
Cotutora:
Ana María Plácido Castro

Fecha: 14 de julio de 2021
Curso: 2020/2021

Agradecimientos

Quiero agradecer este TFG en primer lugar al Instituto Universitario de Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería, por brindarme la oportunidad de realizar este trabajo en un entorno real.

También quería agradecerle a mis amigos, en especial a Jared, Kevin y Miguel, y a mis compañeros de carrera, en especial a Yonay, por ayudarme en mi primer año y enseñarme lo que necesitaba para avanzar, en las tardes junto con otros alumnos perdidos.

Sobre todo, y para finalizar, agradecerle a Adriana, mi pareja, por ayudarme y apoyarme durante todo este trayecto, y a mi familia, a mi hermano Thanay, por brindarme la ayuda que necesitaba, a mi abuela Yeya, por cuidarme desde que era pequeño, y en especial a mis padres, Carlos e Ina, por traerme hasta donde estoy y estar conmigo hasta el final.

Gracias.

Resumen

Se ha diseñado e implementado una aplicación que calcula los aforos y la capacidad de carga de la playa de Las Canteras. El aforo es el número de personas que pueden estar en un espacio de forma segura, mientras que la capacidad de carga es el número de personas que pueden estar en un espacio natural sin afectar a su ecosistema. Se ha implementado un modelo que obtiene la superficie de diferentes zonas de la playa a partir de la altura de la marea. Tras ello, se realiza una estimación del aforo y capacidad de carga de dichas zonas teniendo en cuenta diversas condiciones. El sistema puede mostrar los cálculos tanto en tiempo real como datos históricos o predicciones.

Abstract

We have designed an app capable of calculating the capacity and loading capacity of Las Canteras beach. The capacity is the maximum number of people that can be in a spot without any security risk, whilst the loading capacity is the number of people that can be in a natural space without affecting its ecosistem. To do this, a model which obtains the surface of various zones of Las Canteras beach from a tide height measure has been included. After calculating the surface, it estimates, given various conditions, the number of people that can be in said zones at the same time. This application is capable of displaying the calculations in real time, as well as historical data and predictions.

Índice general

Índice de figuras

Índice de tablas

Introducción

Actualmente es bien sabido que los espacios naturales se ven afectados directamente por las personas que estén en el mismo. El número de personas que puede haber en un espacio natural sin afectar al ecosistema se conoce como “capacidad de carga”. Este término fue usado por primera vez por Platón, en el libro V de Las leyes, en el que declaró que “no se puede fijar un total adecuado del número de ciudadanos sin considerar la tierra y los estados vecinos”. (Rees y Wackernagel 2001) [1] Posteriormente, se usó para la gestión de poblaciones de ganado. [2]

En diciembre de 2020 se inició el proyecto Investiga las Canteras. [3] Dicho proyecto fue promovido por el Ayuntamiento de Las Palmas de Gran Canaria, así como por la Universidad de Las Palmas de Gran Canaria, con el objetivo de estudiar el desarrollo sostenible de la playa de Las Canteras. En el contexto de la parte del proyecto en el que estoy implicado, este se enfoca en el estudio de la capacidad de carga de la playa.

La capacidad de carga, mencionada anteriormente, se define cómo el número máximo de personas que puede haber en un espacio sin que el ecosistema del mismo se vea afectado. En el caso de las playas este valor no es constante a lo largo del día, dado que las mareas influyen directamente en el espacio disponible de la playa y, por consiguiente, en la capacidad de carga de esta.

En el contexto del proyecto Investiga las Canteras, este se ha planteado con el objetivo de estudiar la capacidad de carga de la playa de Las Canteras para, por un lado, preservar el ecosistema que puede verse afectado por un número de personas excesivos en el espacio natural.

Ilustración 1: Playa de Las Canteras [4]

Por otro lado, por la seguridad de las personas que acuden a la playa. En el caso de una evacuación de la playa, hay que tener en cuenta el número de personas que puede haber en ella para que la evacuación sea segura para todos los implicados y así evitar accidentes. También debemos considerar el personal de salvamento, dado que hay un límite humano en el número de personas que pueden rescatar en la playa.

Como ejemplo más reciente nos encontramos con el número de personas que puede haber en la playa respetando la distancia de seguridad de las medidas implantadas por la COVID-19. [5]

Estado actual y objetivos iniciales

Estado actual

En los últimos años, se ha visto como los espacios naturales se han convertido en áreas indispensables para conservar los ecosistemas y el medio ambiente. Como dicen Javier Gómez-Limón García y Diego García Ventura en su libro *Capacidad de acogida de uso público en los espacios naturales protegidos – nº 3*: "En los últimos 40 años la superficie protegida a nivel mundial ha aumentado de forma considerable. A mediados de los años 60, del pasado siglo, solo se contaba con un 3% de territorio protegido. En el último informe de la Comisión Mundial de Áreas Protegidas de la UICN (WCPA Strategic Plan 2005-2012) se afirma que la superficie terrestre protegida a nivel mundial, esto supone el 14% de la superficie de nuestro planeta." [2]

Aún con la importancia demostrada de los mismos, el concepto de espacio protegido se comienza a usar a finales del siglo XIX, a raíz de la necesidad de evitar la desaparición de espacios con fauna o flora excepcionales [2], siendo áreas que se diferencian por la necesidad de garantizar beneficios tanto a la sociedad como al ecosistema desde el punto de vista de la gestión.

Ilustración 2: Monumento natural Roque Nublo [6]

En la actualidad, los espacios naturales protegidos españoles reciben cerca de 30 millones de visitas [7]. Gracias a esta gran afluencia de personas, las poblaciones locales que habitan en estos espacios experimentan un desarrollo socioeconómico, aunque también puede generar un impacto en el ecosistema no deseado [8] y, aunque inicialmente los espacios naturales protegidos se crearon para conservar un territorio destinado a una zona de recreación, en la actualidad ha pasado a ser una gran herramienta con el objetivo de la conservación del espacio protegido. [9]

Ilustración 4: Erosión del suelo en el Parque Natural de la Corona Forestal de Tenerife [2]

Ilustración 3: Erosión del suelo en el Parque Regional de la Cuenca Alta del Manzanares. Madrid. [2]

Por ello, es el gestor de estos espacios el que debe analizar los diferentes factores que pueden ocasionar un impacto en el espacio protegido con el fin de preservarlo a la vez que planifica y gestiona las actividades para el uso público.

Estos impactos de los visitantes sobre los espacios pueden ser sobre el suelo, sobre la fauna y sobre el medio acuático. El primero por los impactos en el suelo del paso de los visitantes, que puede llevar a la erosión del suelo, y como puede verse en las ilustraciones anteriores; El segundo por la influencia que tiene el ser humano en los animales del espacio, afectando a la dinámica y el comportamiento de estos; Y el último por el impacto que supone el baño de las personas en el entorno, afectando a la calidad de las aguas y los parámetros vinculados a la misma. [2]

Esta evolución de los espacios naturales y cómo se tratan, así como los impactos que pueden acarrear los visitantes, en la actualidad no se suelen aplicar a las playas. Prueba de ello es que en ningún momento en los últimos años se ha controlado ni el número de personas en la playa ni las actividades que se realizan en ellas, pudiéndose ver playas en las que el tránsito por ellas se vuelve complicado por el gran número de personas que las visitan.

Este es uno de los motivos por los que se ha iniciado el proyecto Investiga las Canteras, mencionado anteriormente, y con el objetivo de estudiar la capacidad de carga de la playa de las Canteras, pero ¿qué es la capacidad de carga?

También conocida como capacidad de acogida, la capacidad de carga tiene como origen la gestión de poblaciones de ganado y de recursos renovables. Como ejemplo, sabemos que una extensión de campo tiene una cierta capacidad de carga ganadera en función de la cantidad de alimento disponible para el ganado. Al añadir más ganado, este pisará sobre el campo, dejando esa zona inservible como alimento y reduciendo así el número de alimento total para todo el ganado. [2]

Ilustración 5: Vacas en un campo [10]

A comienzos de la década de los 80 es cuando se comienza a usar el concepto de capacidad de carga relacionada al turismo, estableciéndose el número máximo de personas que puede haber en un espacio natural debido a los límites de este sin provocar daños irreversibles.

La organización Mundial del Turismo la define como "el número máximo de personas que pueden visitar un destino turístico al mismo tiempo sin poner en peligro el medio físico, económico o sociocultural y causar la disminución en el nivel de satisfacción de los visitantes". [11]

El trabajo desarrollado por Miguel Cifuentes en 1992 [12] supuso la descripción de las bases metodológicas para la aplicación de un procedimiento a la hora de tener un desarrollo turístico sostenible. [2] Estos procedimientos dan lugar a tres niveles de capacidad de carga en un espacio, cada uno de los cuales, tiene en cuenta el anterior nivel para realizar el cálculo:

- Capacidad de carga física: el número máximo de personas que caben sin afectar al ecosistema.
- Capacidad de carga real: la capacidad de carga determinada también por los factores particulares del ecosistema
- Capacidad de carga efectiva o permisible: tiene en cuenta los factores referidos a la administración del área, como por ejemplo el número de socorristas en la playa y las personas que pueden salvar.

En lo que atañe a este proyecto, el número de personas que haya en una playa afecta a la capacidad de preservación de un espacio natural, tanto degradándolo como pudiendo disminuir el nivel de satisfacción de los usuarios de la playa. [2]

En la fase de estudio previo, se realizó una reunión con el doctorando Eloy José del Rosario Rodríguez, que se encontraba realizando el programa de doctorado Oceanografía y Cambio Global, dirigido por la geógrafa Emma Pérez-Chacón Espino, y que era partícipe del proyecto Investiga Las Canteras, en la que explicaría en qué consiste el trabajo que estaba realizando. Este trabajo consistía en el estudio de la capacidad de carga de la playa de Las Canteras y un estudio de la superficie de la misma.

Tras eso, se pasó a una fase de estudio y análisis de los conceptos que se usaron durante el desarrollo de la aplicación, así como de los datos obtenidos de superficie y altura referenciales y de las

zonas en las que se dividiría la playa. Los datos de superficie y alturas referenciales se tratan de mediciones de superficie realizadas en cinco alturas diferentes de la marea, estas son: bajamar, bajamar media, media, pleamar media y pleamar, con unas alturas de -1.434, -0.896, 0, 0.843 y 1.443 respectivamente. Estas alturas son tomadas con el nivel medio del mar como referencia.

Así mismo, la playa también estaría dividida en treinta y dos sectores diferentes que posteriormente se agruparían en cinco zonas. Dichos sectores surgieron de la división de la playa según líneas paralelas a la línea de costa. De la misma forma, la playa también fue dividida mediante líneas perpendiculares teniendo en cuenta las entradas existentes a la misma. Ambas divisiones dieron lugar a los treinta y dos sectores de la playa, mediciones que se tomaron mediante un análisis espacial de la superficie de la playa. En la siguiente tabla se pueden ver las superficies de los diferentes sectores en las cinco alturas referenciales.

sector	bajamar	bajamar media	media	pleamar media	pleamar
1	4.206	3.833	3.457	2.954	2.443
2	6.140	5.704	5.257	4.796	4.316
3	4.831	4.389	3.928	3.434	2.919
4	5.260	4.602	3.918	3.206	2.460
5	6.271	5.647	5.003	4.338	3.644
6	6.606	5.945	5.257	4.541	3.794
7	5.496	4.794	4.079	3.347	2.594
8	7.667	6.687	5.711	4.735	3.751
9	5.491	4.793	4.086	3.367	2.625
10	3.636	3.118	2.598	2.079	1.552
11	2.500	2.041	1.572	1.092	607
12	2.563	2.037	1.514	991	467
13	5.939	4.575	3.227	1.891	574
14	2.763	2.252	1.732	1.206	676
15	1.675	1.482	1.091	738	253
16	52	52	52	50	7
17	2.316	1.753	1.211	692	183
18	4.337	3.547	2.709	1.824	901
19	6.062	5.013	3.939	2.821	1.642
20	5.631	4.443	3.248	2.049	859
21	3.051	2.317	1.575	825	78
22	1.178	849	549	276	20
23	4.663	3.288	2.061	988	95
24	6.680	5.373	4.010	2.541	938
25	9.236	7.582	5.895	4.149	2.352
26	5.201	4.106	2.996	1.870	728
27	3.733	2.859	1.997	1.143	290
28	7.591	5.865	4.134	2.393	639
29	13.597	10.726	7.760	4.683	1.488
30	6.191	4.687	3.181	1.673	175
31	11.517	8.836	6.047	3.134	133
32	6.823	5.414	3.891	2.252	507
Total	**168.903**	**138.609**	**107.685**	**76.078**	**43.710**

Tabla 1: Superficie de los sectores de la playa en cada nivel de marea

Como se ha mencionado anteriormente, estos treinta y dos sectores se han dividido en cinco zonas. Dichas zonas son Jardines del Atlántico, Churruca, Olof Palme, Tomás Miller y Saulo Torón. Estas vienen dadas en los datos por unas coordenadas que hacen referencia al inicio y el fin de cada zona y, según estas coordenadas, es como se dividió los sectores de la playa de las Canteras en las diferentes zonas. Estas divisiones de zonas no son arbitrarias, sino que vienen dadas por el ayuntamiento de Las Palmas de Gran Canaria, y en la aplicación del proyecto se ajustaron los treinta y dos sectores a esas cinco zonas de forma que el resultado quedara coherente con la realidad y no mostrar un gran volumen de datos al usuario en la aplicación final, siendo más sencilla la visualización si los datos están divididos en esas cinco zonas.

Zona	Desde	Hasta
Saulo Torón	28.148735, -15.431398	28.146426, -15.430405
Tomás Miller	28.146426, -15.430405	28.142705, -15.433373
Olof Palme	28.142705, -15.433373	28.136687, -15.438083
Churruca	28.136687, -15.438083	28.133517, -15.442036
Jardines del Atlántico	28.133517, -15.442036	28.130710, -15.449113

Tabla 2: Coordenadas de las zonas de Las Canteras

Zona	Sectores
Jardines del Atlántico	1,2,3,4,5,6
Tomás Miller	7,8,9,10,11,12
Olof Palme	13,14,15,16,17,18
Churruca	19,20,21,22,23,24
Saulo Torón	25,26,27,28,29,30,31,32

Tabla 3: División de sectores en las zonas

Adicionalmente, también se recogen datos de la altura de la marea de la playa de Las Canteras. Estos datos provienen de la web de Puertos del Estado [13]. Se trata de una web que provee al usuario de diversos tipos de información de los mareógrafos de las costas españolas. Los datos que provee son, por ejemplo, el oleaje, el nivel del mar, la salinidad, etc., tanto en tiempo real, como predicciones y un histórico de estos datos, pudiendo mostrarlo en forma de tabla o de gráfico. En el contexto que nos concierne, se han recogido los datos de la altura de la marea actuales y los previstos, obtenidos del Mareógrafo de Las Palmas, situado en el Muelle Elder, dado que es la fuente de datos más cercana a la playa de Las Canteras que está disponible.

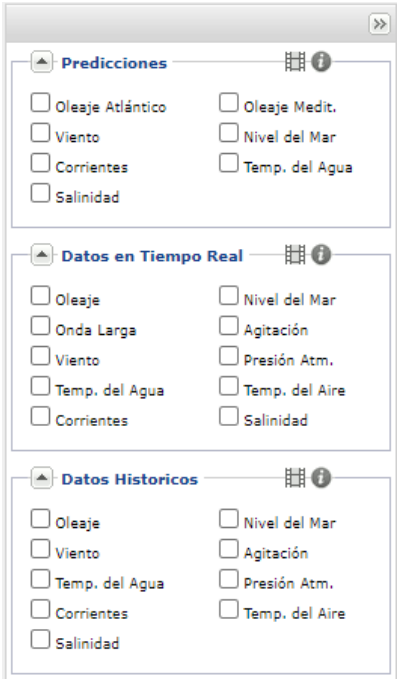


Ilustración 6: Datos provistos posibles por la web de PORTUS [13]

Los datos recogidos en la web se actualizan cada minuto, con una diferencia de alrededor de cinco minutos con la realidad, aunque no tiene en cuenta los cambios de horario que vienen con los solsticios, teniendo permanentemente el horario invernal.

Como se puede observar en la Ilustración 7, la tabla incluye los datos de tres niveles del mar diferentes, estos son: nivel cero REDMAR, nivel medio y nivel cero hidrográfico. El nivel cero REDMAR es el que usa como referencia cada mareógrafo de la REDMAR [14], que es "un proyecto para generar una red de intercambio de experiencias que favorecieran la interrelación diferentes entidades vinculadas al sector pesquero" [15]. El nivel cero hidrográfico toma como referencia la cartografía marina, coincidiendo con el nivel más bajo del agua [14]. Finalmente, el nivel medio, también llamado cero geodésico por el Instituto Geográfico Nacional, en las islas canarias es el que se obtiene a partir de las mediciones de un mareógrafo en un tiempo determinado en una isla. [14].

Mareografo de Las Palmas 2

Fecha (GMT)	Nivel (cero REDMAR) (m)	Nivel (nivel medio) (m)	Nivel (cero Hidrográfico) (m)	Altura Signif. del Oleaje (m)	Onda Larga (m)
15-06-2021 11:07	1.153	-0.453	0.963		
15-06-2021 11:06	1.153	-0.453	0.963		0.0042
15-06-2021 11:05	1.151	-0.455	0.961		0.0059
15-06-2021 11:04	1.145	-0.461	0.955		0.0032
15-06-2021 11:03	1.138	-0.468	0.948		-0.0008
15-06-2021 11:02	1.132	-0.474	0.942		-0.004
15-06-2021 11:01	1.127	-0.479	0.937		-0.0058
15-06-2021 11:00	1.125	-0.481	0.935	0.08	-0.0047
15-06-2021 10:59	1.121	-0.485	0.931		-0.006
15-06-2021 10:58	1.115	-0.491	0.925		-0.0093
15-06-2021 10:57	1.111	-0.495	0.921		-0.0102
15-06-2021 10:56	1.11	-0.496	0.92		-0.0077
15-06-2021 10:55	1.112	-0.494	0.922		-0.0018
15-06-2021 10:54	1.115	-0.491	0.925		0.0048
15-06-2021 10:53	1.111	-0.495	0.921		0.0034

Page 1 of 96

Ilustración 7: Recorte de la tabla de nivel del mar en tiempo real en la web de PORTUS [13]

De los niveles descritos anteriormente, es el último el que se ha tomado como referencia a la hora de hacer los cálculos y se han descartado los otros dos, dado que para el cálculo de los datos referenciales nombrados en la Tabla 1 y en la Tabla 2, este ha sido el nivel usado.

Objetivos iniciales

De esta forma, y como objetivos iniciales, se ha propuesto inicialmente la realización de una funcionalidad software que pueda ser capaz de realizar una estimación de la capacidad de carga de la playa de Las Canteras en un determinado momento, en función de múltiples factores que pueden afectar a esta, teniendo en cuenta el nivel de la marea.

Al ser factores de naturaleza variable, se pretende crear una arquitectura que permita añadir o quitar estos criterios que afecten, en mayor o menor medida, a la capacidad de carga.

Adicionalmente se ha propuesto la realización de una aplicación capaz de calcular y predecir el aforo máximo de diversas zonas predefinidas de Las Canteras. De la misma forma, también calculará el aforo máximo teniendo en cuenta las diversas variables que se mencionaron anteriormente que pueden ser, por ejemplo, las restricciones por la COVID, la presencia de medusas, la preservación del ecosistema, etc. De la misma forma, la capacidad de carga calculada se dividirá en la capacidad de carga física, real y efectiva, descritas anteriormente. Teniendo en cuenta dichos niveles, las diferentes variables que puedan añadirse, por la naturaleza de las mismas, no afectarán a todos los niveles, teniendo que diferenciar a qué nivel afecta y cómo afecta el cambio de nivel de capacidad de carga al siguiente.

También se ha propuesto como objetivo el poder especificarle al modelo software el nivel de la marea en cada momento en relación con el cero marítimo. El nivel del mar afecta directamente a la superficie útil de la playa, ya que cuanto más alta sea la marea, menos superficie habrá, afectando de igual manera a la capacidad de carga de la playa.

Finalmente se ha propuesto la creación de un cuadro de mandos como objetivo. En dicho cuadro de mandos se podrán añadir tareas desde las que se permitirá emitir alertas en caso de que se dé una situación excepcional y que pueda afectar a la playa, como puede ser un derrame de petróleo.

Competencias específicas y aportaciones del trabajo

Competencias específicas relacionadas

- IS01. Durante el desarrollo de la aplicación se ha realizado un desarrollo guiado por pruebas cuando ha sido posible, y se ha seguido una metodología iterativa, realizándose un análisis y diseño de la aplicación, implementando los cambios propuestos y realizando las pruebas de funcionamiento. De esta manera se ha tenido especial cuidado en que los cambios y las nuevas funcionalidades de cada iteración no afecten a lo anterior, a la vez que ofrezca un avance significativo en relación con la iteración anterior.
- IS02. Durante el desarrollo, después de cada iteración realizada, ha habido una reunión para revisar los cambios realizados, así como para proponer nuevos cambios para la siguiente iteración.
- IS03. Se ha podido conectar dos programas software en diferentes entornos y lenguajes y que se comuniquen correctamente para un funcionamiento adecuado de la aplicación final.
- IS06. En la aplicación se puede tener en cuenta el distanciamiento social que ha generado la COVID como criterio para el cálculo de la capacidad de carga de la playa.

Aportaciones del trabajo

En cuanto a las aportaciones del trabajo a nuestro entorno, el programa sirve como solución para tener una referencia del número de personas que hay en la playa. Esto será bastante útil porque el encargado de seguridad podría restringir el acceso a la playa en caso necesario para que el ecosistema de la playa no se vea dañado irreparablemente. Adicionalmente, con las medidas de distanciamiento social, el programa puede servir de guía para controlar que no se salten esas medidas y así manteniendo la seguridad de todos los usuarios de la playa.

Por otro lado, el modelo de cálculo de superficie de la playa y, por consiguiente, de capacidad de carga, se podría llevar a otras playas con facilidad, dado que para hacer el cálculo solo será necesaria la altura de la marea y la superficie referencial de la playa a una altura referencial, que sirve para realizar la interpolación que calculará la superficie actual. Esto es un modelo que no se ha planteado con anterioridad en otros campos y que podría ser innovativo en el cálculo rápido de los datos de la playa.

Resultados

Se ha conseguido desarrollar una aplicación útil y fácilmente usable por el usuario, con la que se puede familiarizar rápidamente sin necesidad de ayuda externa. En ella se recogen los datos de la marea en cada momento y se realiza una estimación de la capacidad de carga de diferentes zonas de la playa de las canteras descritas anteriormente. Esta estimación se realiza para el cálculo de la capacidad de carga física, a partir de la cual se calcula el resto de la misma.

Para la recolección de datos se ha realizado varios scraper, que se conectan con la web de Puertos del Estado [13] para obtener los datos. Estos datos se recogen en crudo, por tanto, el scraper también es el encargado de interpretar estos datos y filtrar los útiles entre todos los recogidos, ya que la web proporciona varios datos a la vez más allá de la altura de la marea, como se verá posteriormente. Esta recolección y filtrado de datos se ha conseguido hacer satisfactoriamente y se ha desarrollado de forma que se puede extrapolar a la recolección de otros datos de la web, como se comentará más adelante en el apartado de desarrollo.

Se obtienen tanto los datos de marea actual como los datos de predicción de la marea. Para el caso de la marea actual, se recoge una media de los datos actuales, que servirán para los datos del histórico. Para éste, se obtienen de los datos recogidos por la aplicación en cada momento, que son guardados en la aplicación cada minuto para luego cada hora guardarlos en memoria, como se explicará más adelante en el desarrollo.

Tanto la capacidad de carga real como la efectiva se ve afectada por las restricciones que pueden ser indicadas mediante un archivo a la aplicación cuando se inicia la ejecución. Estas restricciones, que serán explicadas posteriormente, pueden afectar a la capacidad de carga o no, dependiendo de si el usuario ha marcado el cuadro que puede verse a la izquierda de las restricciones. Las restricciones nuevas pueden añadirse antes del inicio de la aplicación al archivo, cuyo formato es detallado más adelante en el apartado de desarrollo. El software se encarga de leer dicho archivo, interpretar los datos y mostrar las restricciones en la aplicación.

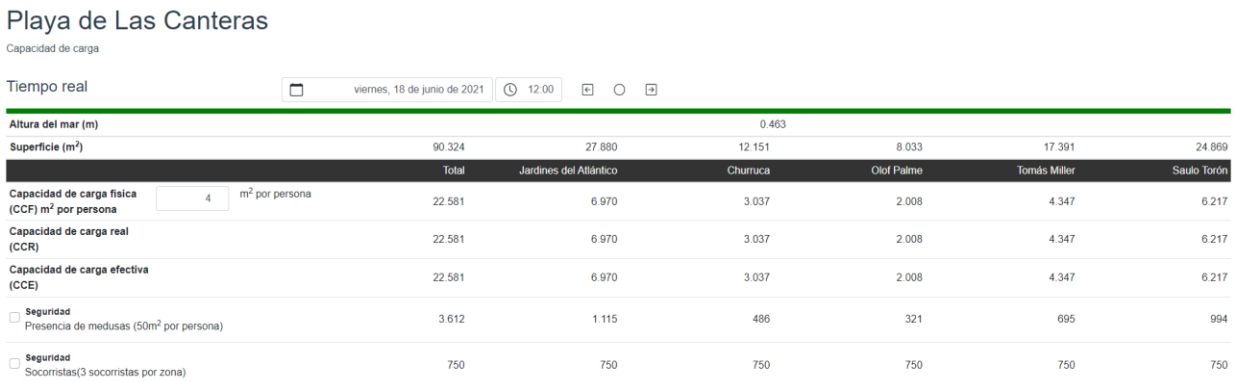
Así mismo, y como se mencionó anteriormente, la capacidad de carga sigue una estructura en la que el valor de cada nivel se ve directamente afectado por el del nivel anterior. De esta forma, si una restricción reduce el valor de la capacidad de carga real, la capacidad de carga efectiva es calculada a partir de ese valor reducido.

Cabe destacar que en el caso de que varias restricciones afecten a un mismo nivel de capacidad de carga, el valor que se utilizará será el más restrictivo, es decir, el valor de capacidad de carga que sea menor.

En cuanto a la capacidad de carga física, y como puede verse en las ilustraciones, al lado de esta se encuentra un cuadro de texto en el que el usuario puede indicar el número de metros cuadrados de los que dispone una persona. Este número se usa para el cálculo de esta capacidad de carga en la aplicación, dado que esto afecta directamente al número de personas que puede haber en la playa. Es en este número en el que se puede introducir una restricción como, por ejemplo, la de la distancia de seguridad de la COVID, dado que afecta a la distancia entre personas en cualquier momento.

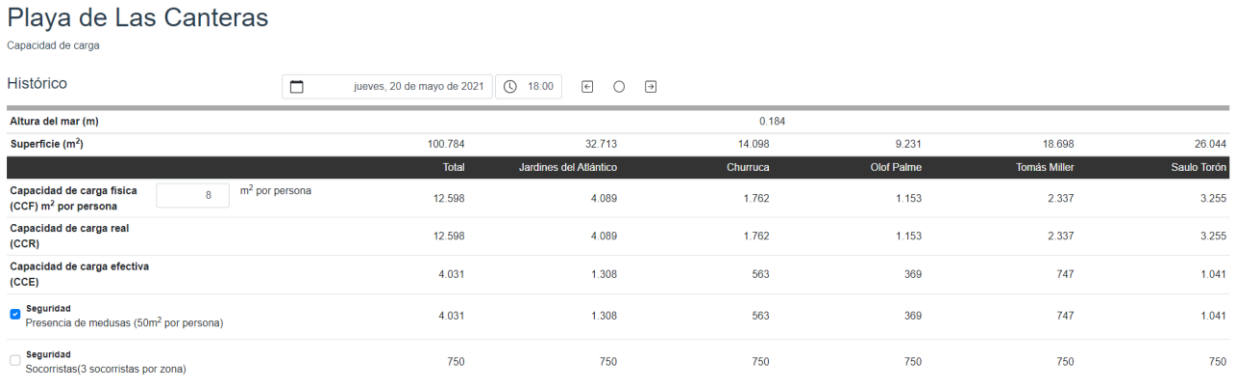
Finalmente, la aplicación, en los tres periodos de tiempo, quedo de la siguiente manera:

Tiempo real



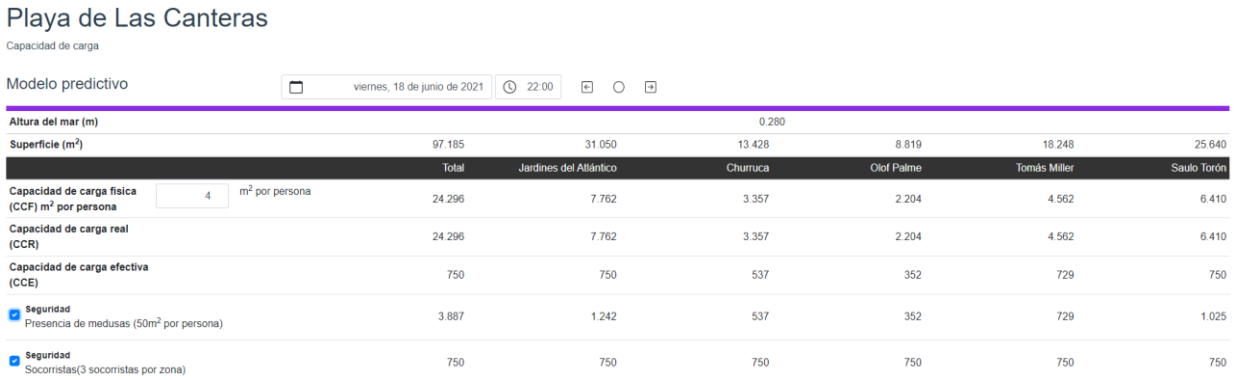
Altura del mar (m)			0.463			
Superficie (m²)	90.324	27.880	12.151	8.033	17.391	24.869
	Total	Jardines del Atlántico	Churruca	Olof Palme	Tomás Miller	Saulo Torón
Capacidad de carga física (CCF) m² por persona (4 m² por persona)	22.581	6.970	3.037	2.008	4.347	6.217
Capacidad de carga real (CCR)	22.581	6.970	3.037	2.008	4.347	6.217
Capacidad de carga efectiva (CCE)	22.581	6.970	3.037	2.008	4.347	6.217
Seguridad Presencia de medusas (50m² por persona)	3.612	1.115	486	321	695	994
Seguridad Socorristas(3 socorristas por zona)	750	750	750	750	750	750

Histórico



Altura del mar (m)			0.184			
Superficie (m²)	100.784	32.713	14.098	9.231	18.698	26.044
	Total	Jardines del Atlántico	Churruca	Olof Palme	Tomás Miller	Saulo Torón
Capacidad de carga física (CCF) m² por persona (8 m² por persona)	12.598	4.089	1.762	1.153	2.337	3.255
Capacidad de carga real (CCR)	12.598	4.089	1.762	1.153	2.337	3.255
Capacidad de carga efectiva (CCE)	4.031	1.308	563	369	747	1.041
Seguridad Presencia de medusas (50m² por persona)	4.031	1.308	563	369	747	1.041
Seguridad Socorristas(3 socorristas por zona)	750	750	750	750	750	750

Predicción



Altura del mar (m)			0.280			
Superficie (m²)	97.185	31.050	13.428	8.819	18.248	25.640
	Total	Jardines del Atlántico	Churruca	Olof Palme	Tomás Miller	Saulo Torón
Capacidad de carga física (CCF) m² por persona (4 m² por persona)	24.296	7.762	3.357	2.204	4.562	6.410
Capacidad de carga real (CCR)	24.296	7.762	3.357	2.204	4.562	6.410
Capacidad de carga efectiva (CCE)	750	750	537	352	729	750
Seguridad Presencia de medusas (50m² por persona)	3.887	1.242	537	352	729	1.025
Seguridad Socorristas(3 socorristas por zona)	750	750	750	750	750	750

En cuanto al cálculo de la superficie, se diseñó un modelo que lo realiza mediante una interpolación a partir de la altura de la marea del momento en el que se encuentre la aplicación, indicado en la primera fila de la tabla de la misma. Con este dato la aplicación compara la altura de la marea con las alturas referenciales mencionadas anteriormente para saber entre qué alturas referenciales se encuentra. Tras esto nos encontramos con tres posibles casos: cuando la altura de la medición es igual a alguna de las alturas referenciales, cuando la medición se encuentra entre dos alturas, y cuando la medición es mayor o menor que las alturas referenciales de los límites.

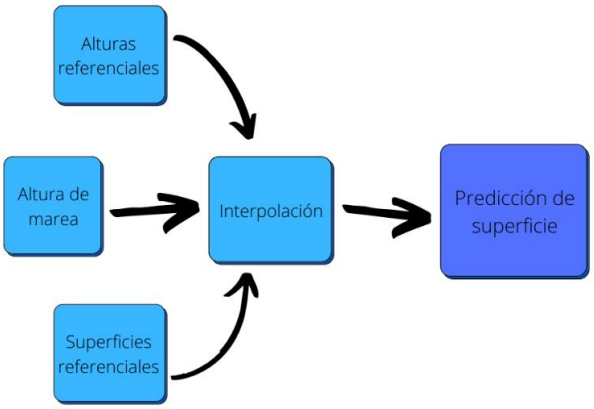
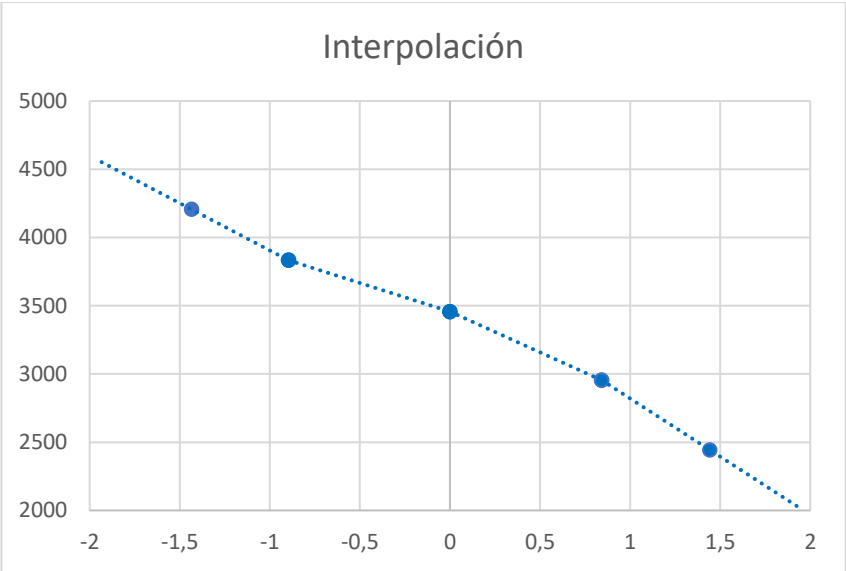


Ilustración 8: Diagrama del modelo

Para el primer caso, para cada sector se toma las superficies referenciales de dichos sectores y es el valor que se usa, dado que, como la altura referencial y la medición son iguales, los valores de la superficie deben ser el mismo.

Para los otros dos casos, con las alturas referenciales se toma la superficie de las alturas en los distintos sectores y, con ello y las alturas, se realiza la interpolación entre los dos puntos para calcular el valor de la superficie en cada sector, como puede verse en la siguiente gráfica. En cuanto al último caso, se toma las dos últimas alturas referenciales, o las dos primeras, y sus superficies para realizar una extrapolación y así estimar la superficie. Por ejemplo, en el caso de que la medición tomada de la altura sea de 1.67 m, este valor es mayor que el valor referencial más alto, el de pleamar. Como puede verse en la siguiente ilustración, los valores tomados son los referenciales para el primer sector de la playa. Al extrapolar entre los dos últimos puntos, podemos tener una estimación del valor de la superficie, dado que el real se acercará al valor que puede verse al seguir la recta interpolada.

Superficie	Altura
4206	-1,434
3833	-0,896
3457	0
2954	0,843
2443	1,443



Desarrollo

Metodología de desarrollo

Para el desarrollo de la aplicación se ha seguido una metodología iterativa, es decir, un proceso de desarrollo basado en un conjunto de tareas agrupadas en pequeñas iteraciones [16], con diferentes iteraciones en las que se ha realizado un análisis y un diseño de la aplicación a realizar, con objetivo de que el usuario reciba valor de la aplicación de manera creciente. Posteriormente, se han implementado las nuevas funcionalidades, así como se han corregido los cambios propuestos y finalmente se han realizado las pruebas pertinentes para comprobar que el programa funcione correctamente y esté listo para la siguiente iteración.

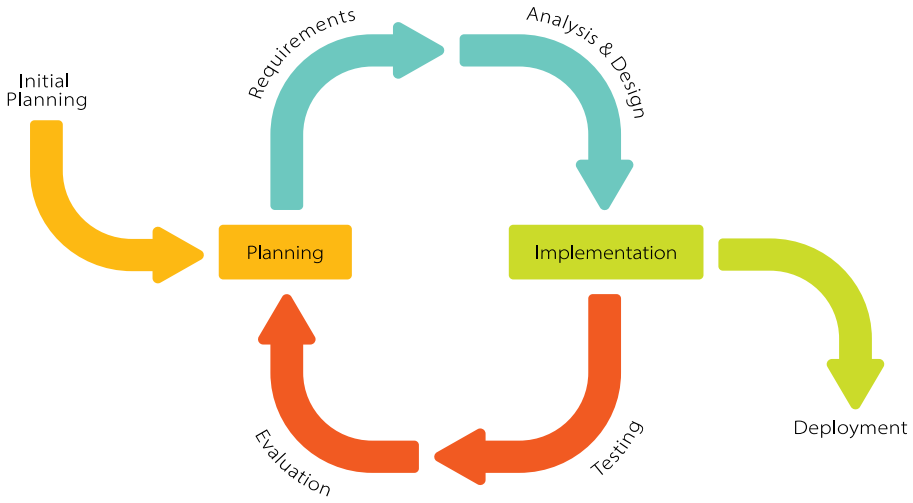


Ilustración 9: Modelo de desarrollo iterativo [17]

Tras una planificación inicial de los objetivos de la aplicación, el desarrollo iterativo se divide en diversas iteraciones en las que se realiza un proceso que trae un avance en el resultado final, buscando que cada iteración agregue más valor al producto final y logrando una aplicación más completa. Antes de cada iteración se realiza una reunión de planificación de los avances que se desarrollarán durante la iteración [16]. Este proceso trae consigo diversas ventajas, tanto por el aspecto incremental, como por el aspecto iterativo del proceso de desarrollo.

Desde el punto de vista incremental, este proceso de desarrollo permite que el usuario pueda ver los avances en la aplicación antes de que la aplicación esté acabada, ya que se basa en aportar valor al usuario en cada iteración. Adicionalmente, cada incremento sirve como prototipo que el usuario puede utilizar, siendo menos probable que haya errores en la aplicación final ya que hay un producto que se va probando y corrigiendo a medida que el desarrollo avanza. [16]

Por otro lado, al ser un desarrollo iterativo, los usuarios, desde una etapa temprana, dan retroalimentación a los desarrolladores dado que se entrega valor tras cada iteración. Adicionalmente este proceso permite dividir la complejidad del proyecto en partes más sencillas que se desarrollarán en cada iteración, aparte de permitir un desarrollo consistente y una entrega a tiempo del producto. Finalmente, el desarrollador aprende en cada iteración que será útil tanto para el desarrollo del resto del proyecto como para nuevos proyectos. [16]

Por esto, se ha decidido optar por el desarrollo iterativo e incremental, no solo por permitir entregar un producto evaluable desde una etapa temprana de la aplicación, sino también por permitir identificar en cada iteración los requisitos del usuario, pudiendo llegar a tener una satisfacción mayor del usuario mientras se ha llevado un desarrollo más flexible que con otros procesos, así como la variabilidad que este proceso permite en cuanto al desarrollo, el tamaño del proyecto, pudiendo usarse tanto en proyectos pequeños como en

proyectos de gran tamaño, en contraste con el desarrollo en cascada, que se trata de un desarrollo que ordena las etapas de desarrollo de tal manera que para que pueda iniciar cada etapa, se debe esperar a la finalización de la etapa anterior, que aun siendo útil en determinadas situaciones, los factores descritos anteriormente han hecho que un desarrollo iterativo fuera la decisión correcta. [18]

Diagrama modular

Se creó un diagrama modular con la siguiente estructura:

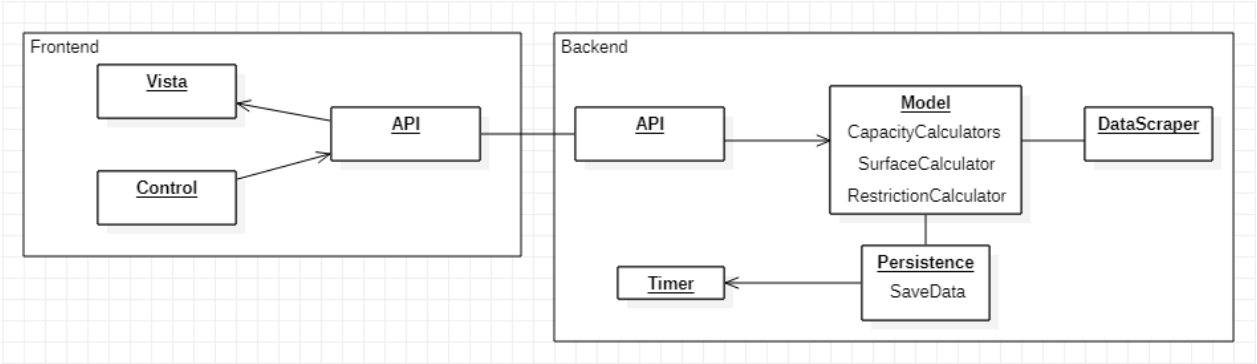


Ilustración 10: Diagrama modular de la aplicación

Por un lado, nos encontramos con la parte del frontend, compuesta por tres módulos principales: la Vista, el Control y la API.

El módulo de la vista es el encargado de tomar los datos que llegan desde el módulo de la API, y mostrarlos en los distintos apartados de la aplicación. Estos cambios son acordes a los cambios especificados por el módulo de control.

El módulo de control es el encargado de interpretar los cambios que el usuario realiza en la aplicación y actuar en consecuencia. Concretamente, en la aplicación será el encargado de realizar los cambios temporales en la aplicación, ya sea mediante los botones o los campos de fecha y hora. También es el encargado de recoger el dato de los metros cuadrados por persona en la capacidad de carga física e indicar a la API este dato, así como las desactivaciones o activaciones de las restricciones.

Para finalizar con el frontend, el módulo de la API es el encargado de comunicarse con el backend, tanto para indicarle los cambios realizados en la aplicación, como para recoger los datos que se van a reflejar en la misma.

Por otro lado, en la parte del backend nos encontramos con cinco módulos: la API, el modelo, el DataScrapper, la persistencia y el temporizador.

Para comenzar, la API es la encargada de comunicarse con el frontend para recoger los datos cambiados por el usuario, y enviar los nuevos datos calculados de superficies y capacidades de carga y la medida de la altura, y recoger el dato de la fecha en la que la aplicación se encuentra.

El módulo del modelo, Model en el diagrama, es en el que se realizan las operaciones de cálculo de superficies, capacidades de carga y restricciones de la aplicación, que serán explicados más adelante a medida que se avance el desarrollo. Junto a este módulo nos encontramos el módulo del DataScraper, encargado de recoger los datos de la altura de la marea de la web de PORTUS [13].

Finalmente nos encontramos los módulos del temporizador y la persistencia, nombrados Timer y Persistence respectivamente. Básicamente, el temporizador tiene en cuenta el tiempo real para actualizar los datos y cuando sea necesario comunicarle al módulo Persistence que guarde estos datos, y finalmente, el módulo Persistence es el encargado de formatear y guardar los datos de la aplicación para su uso posterior. Ambos módulos serán explicados más adelante en la memoria.

Primera iteración

Tras haber finalizado la fase de estudio previo y análisis, el desarrollo de la aplicación se comenzó con la implementación del scraper, que es una técnica en la que un programa software es el encargado de extraer datos o información de un sitio web determinado [18]. En la aplicación, se encargará de recoger los datos nombrados anteriormente de la web de Puertos del Estado [13]. Tras ello, el programa interpretaría la respuesta y la normalizaría para obtener los datos que serían necesarios para la aplicación final, es decir, los datos del nivel de la marea actuales y los previstos.

```
//OK['Wg',
0,
1440,
0,
0,
0,
1,
1.9999999494757503E-4,
12,
1,
13,
1,
1.2419999837875366,
12,
1,
-0.17399999499320984,
12,
1,
1.4320000410079956,
12,
1,
'XoKWZGA',
1452,
1,
10,
6,
2,
1,
-0.006200000178068876,
12,
```

Ilustración 11: Fragmento de la respuesta del nivel del mar en el Mareógrafo de Las Palmas

Como puede observarse en la Ilustración 11, muchos de los datos de la respuesta no son válidos o son identificadores de los diferentes datos que luego se muestran en la web de PORTUS. Inicialmente se usó el nivel cero hidrográfico, que siempre se puede encontrar con la secuencia {1, 13, 1}, siendo esta secuencia la que se usó para determinar los datos que mantiene el scraper, desechando el resto.

Inicialmente el scraper se implementó en Python, realizando tanto las sentencias destinadas a realizar la petición de los datos, como puede verse en la Ilustración 12, como las destinadas a la interpretación y normalización de los datos recogidos, como puede verse en la Ilustración 13.

```
headers = {
    'X-GWT-Module-Base': 'https://portus.puertos.es/Portus_RT/portusgwt/',
    'X-GWT-Permutation': '1C515B081663E4BEA92E2671F2079F9A',
    'Content-Type': 'text/x-gwt-rpc; charset=UTF-8',
}
data = '7|0|6|https://portus.puertos.es/Portus_RT/portusgwt/|4763486D406A92423D6F43998BCFC0EF|es.puertos.portus.main.client' \
       '.service.PortusService|requestData|I|Z|1|2|3|4|4|5|5|5|6|3450|41|0|1|'
response = requests.post('https://portus.puertos.es/Portus_RT/portusgwt/rpc', headers=headers, data=data)
str = response.text
tides = extract_tides(str.split(","))
print(tides)
```

Ilustración 12: Recorte del código de la petición de los datos en Python

```
4   def extract_tides(lines):
5       result = []
6       expect = ["1", "13", "1"]
7       index = 0
8       for line in lines:
9           if index == 3:
10              result.append(round(float(line), 3))
11              index = 0
12          elif line == expect[index]:
13              index += 1
14          else:
15              index = 0
16      return result
```

Ilustración 13: Recorte del código de la interpretación y normalización de los datos

En la Ilustración 13 puede verse, de la línea 37 a la 45 la construcción de la petición de los datos a la web de PORTUS, indicando la web que se usará y la petición en el mareógrafo y los datos requeridos. La respuesta de la petición se pasa a la función *extract_tides*, encargada de extraer el nivel de las mareas, en ella comprueba línea por línea si la sucesión de números descrita anteriormente (1, 13, 1) se encuentra en algunas líneas, identificando el nivel cero hidrográfico de cada medición. Una vez cumplida la secuencia, el dato se redondea a tres decimales, para una mejor interpretación visual, y luego se añade a la lista de valores de marea, en la línea 10.

Adicionalmente, se realizaron los scraper de otras medidas que se consideraron que podían ser útiles para la aplicación, pero finalmente no se les dio uso a las mismas. Estas medidas fueron la temperatura del agua y del aire, la salinidad y el oleaje, y se realizó tanto para las mediciones actuales como para las predicciones.

De esta manera, el usuario ya podía ver los datos de las mareas recogidos de la web de PORTUS, y especificados anteriormente. [13]

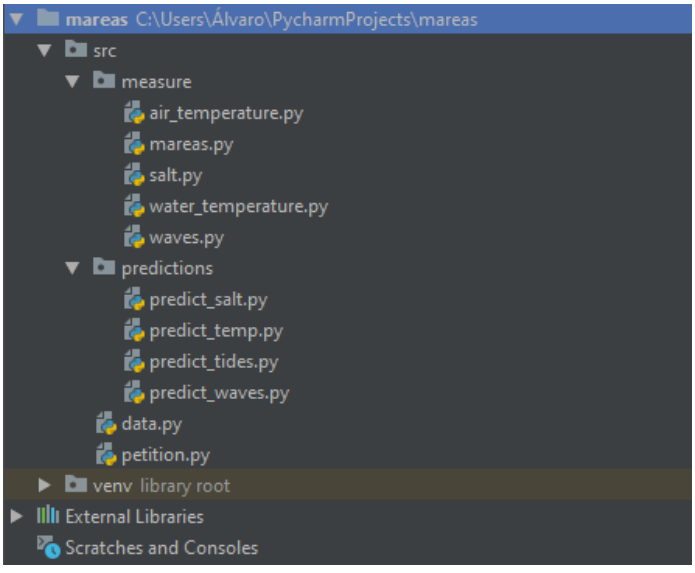


Ilustración 14: Estructura del proyecto en Python

Segunda iteración

Aunque inicialmente, y como se ha podido ver en las ilustraciones anteriores, se había implementado en Python, posteriormente se decidió cambiar de lenguaje a Java, dado que tenía un mayor conocimiento del mismo y la conexión con la interfaz de la aplicación sería más sencilla, así como lo sería implementar el histórico de las mediciones de la marea. La implementación en Python sirvió como una práctica muy útil para familiarizarnos con los datos y los conceptos, así como para la realización del scraper y, gracias a esto, el cambio de lenguaje se pudo hacer rápido. Durante el traslado al nuevo lenguaje, se cambió la secuencia a {1, 12}, que es la que identifica al nivel cero medio, es decir, el que nos interesa para la aplicación. Por consiguiente, hubo que realizar cambios y el primer problema que surgió es que la secuencia sale se puede encontrar varias veces en una medición.

Ilustración 15: Fragmento de la respuesta indicando medición y la secuencia

Como puede observarse en la Ilustración 15, la secuencia aparece hasta cuatro veces en una medición, siendo cada aparición el identificador de un dato diferente de la misma medición, pero la que nos interesa es el valor que va después de la segunda aparición de la secuencia. De esta manera, se realizó la primera aproximación al que sería el scraper final para la aplicación.

```java
    private String extract_tides(String[] lines){
        List<Double> tides = new ArrayList<>();
        String[] expect = {"12", "1"};
        int index = 0, row = 0;
        for (String line: lines) {
            if (index == 2 && row < 4){
                if(row == 1) tides.add((double)Math.round(Double.parseDouble(line) * 1000d) / 1000d);
                index = 0;
                row++;
            }else if(row == 4){
                row = 0;
            }else if(line.equals(expect[index])){
                index++;
            }else{
                index = 0;
            }
        }
        return tides.toString();
    }
```

Ilustración 16: Primera aproximación al scraper de mareas actuales.

Como puede verse en la Ilustración 16, se recorre todas las líneas de la respuesta buscando la secuencia. Cuando encuentra el primer número de la secuencia la variable *index* aumenta en uno. Una vez *index* sea igual a dos significa que ha encontrado la secuencia. Como la secuencia se repite varias veces en la medición, la variable *row* indica cuantas veces ha salido. De esta forma, cuando dicha variable sea igual a uno, la siguiente línea que comprobará será la de la medición que queremos buscar, como se vio anteriormente en la Ilustración 15.

Si la variable *index* llega a dos, como se comentó anteriormente, es que ha llegado a un valor. Tras esto se reinicializa a cero para que pueda seguir con el siguiente valor. De la misma forma, si la variable *row* llega a cuatro significa que ha encontrado la última aparición de la secuencia, de esta forma se reinicializa a cero para poder encontrar la próxima medición.

```
24      private String extract_tides_m(String[] lines){
25          List<Double> tides = new ArrayList<>();
26          String[] expect = {"11", "1"};
27          int index = 0, row = 0;
28          for (String line: lines) {
29              if (index == 2 && row < 3){
30                  if(row == 1) tides.add((double)Math.round(Double.parseDouble(line) * 1000d) / 1000d);
31                  index = 0;
32                  row++;
33              }else if(row == 3){
34                  row = 0;
35              }else if(line.equals(expect[index])){
36                  index++;
37              }else{
38                  index = 0;
39              }
40          }
41          return tides.toString();
42      }
```

Ilustración 17: Primera aproximación al scraper de predicciones de mareas.

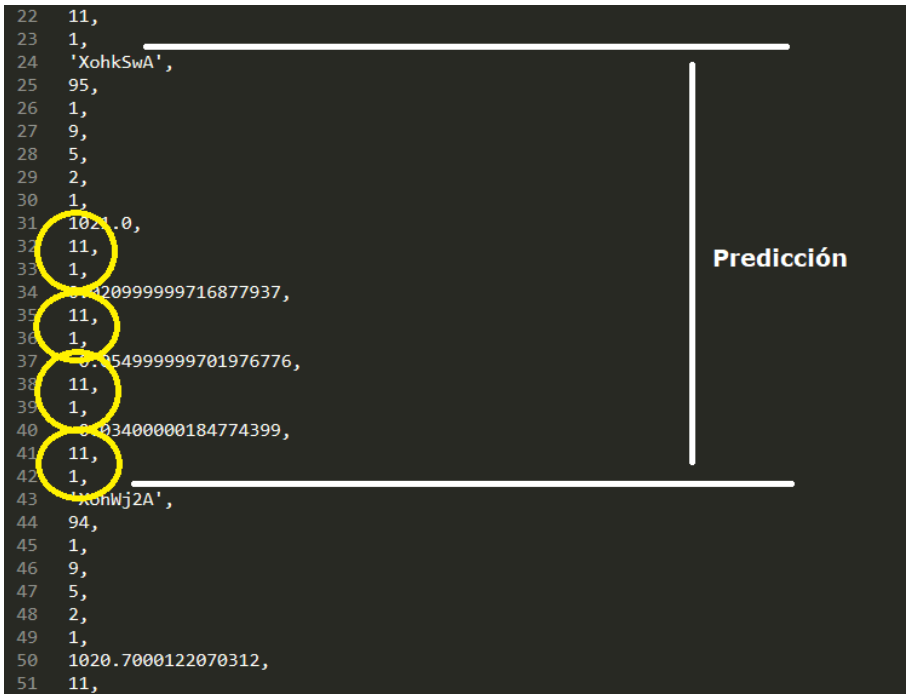


Ilustración 18: Fragmento de la respuesta indicando predicción y la secuencia

Para las predicciones, como puede verse en la Ilustración 18, el código es prácticamente el mismo, lo único que cambia es la secuencia en la que se encuentra la medición que queremos buscar, que en este caso en lugar de ser {12, 1}, pasa a ser {11, 1}.

```
6   public class DataExtractor {
7       public static String extract_data(String[] lines, String[] expect, int maxRow, int targetRow){
8           List<Double> tides = new ArrayList<>();
9           int index = 0, row = 0;
10          for (String line: lines) {
11              if (index == 2 && row < maxRow){
12                  if(row == targetRow)
13                      tides.add((double)Math.round(Double.parseDouble(line) * 1000d) / 1000d);
14                  index = 0;
15                  row++;
16              }else if(row == maxRow){
17                  row = 0;
18              }else if(line.equals(expect[index])){
19                  index++;
20              }else{
21                  index = 0;
22              }
23          }
24          return tides.toString();
25      }
26  }
```

Ilustración 19: Refactorización del método.

Posteriormente, se refactorizó el código, es decir, se reestructuró para que se entendiera mejor y no se repitieran métodos, sin alterar su comportamiento, para que el programa pudiera extraer las mediciones y predicciones desde un mismo método, solo haciendo falta indicar cuántas veces aparece el identificador en la medición, en la Ilustración 19 es la variable maxRow, y cuál de esas apariciones es la que indica la medición que queremos mantener, la variable targetRow.

Inicialmente, la respuesta de los métodos era una ristra de caracteres en los que se podían ver los valores recogidos divididos por comas. Esto fue útil para comprobar el correcto funcionamiento de la recolección, interpretación y normalización de los datos, pero posteriormente esta respuesta se pasó a una lista de los valores, para poder manejarlo mejor y con mayor facilidad en el código.

Para las otras mediciones y predicciones que no se usaron también se realizaron estos métodos, aunque con algunos cambios para adaptarse a las peculiaridades de cada respuesta, como, por ejemplo, para la predicción de salinidad fueron necesarias dos secuencias de números para identificar el valor buscado. Otro ejemplo fue para el oleaje, para ello fue necesario indicar un número máximo de predicciones que recoger, y adicionalmente, se recogieron tres valores por separado de esas predicciones.

Finalmente, quedaron tres métodos para recoger los datos de la web, aunque solo uno de ellos fue utilizado finalmente, dado que los otros dos recogían datos de los scraper que no se pudieron usar en la aplicación final.

```java
public static List<Double> extract_data(String[] lines, String[] expect, int maxRow, int targetRow){
    List<Double> tides = new ArrayList<>();
    int index = 0, row = 0;
    for (String line: lines) {
        if (index == 2 && row < maxRow){
            if(row == targetRow)
                tides.add(parseDouble(line));
            index = 0;
            row++;
        }else if(row == maxRow) row = 0;
        else if(line.equals(expect[index])) index++;
        else index = 0;
    }
    return tides;
}
```

Ilustración 20: Aproximación final del scraper

Una vez finalizado el scraper, y con los datos ya normalizados, se realizó una reunión para combrobar que los datos obtenidos fueran satisfactorios, pero se nombraron aspectos que cambiar para la siguiente iteración, que comentaremos más adelante. Una vez comprobado, se pasó a comenzar el diseño e implementación del software encargado del cálculo de la superficie de las diferentes zonas de la playa de Las Canteras, mencionadas anteriormente, a partir de los datos obtenidos del scraper.

Tercera iteración

Se comenzó implementando los cambios comentados anteriormente, estos son, cambiar el método del scraper para que, en lugar de una lista de los números, devolviera una media de las mediciones de las recogidas. De esta forma, podría usarse el dato para indicar la media en los cálculos y en la aplicación, aunque, como veremos posteriormente, se cambió de nuevo para adaptarse a nuevas funcionalidades.

Posteriormente, se comenzó con el módulo del modelo. Para el cálculo de las superficies de las diferentes zonas se planteó la implementación de una interpolación de los datos obtenidos con los datos referenciales. De esta forma, la aplicación, a partir del dato de la altura de la mar media, comprueba entre qué dos alturas referenciales se encuentra la altura actual, y posteriormente realiza el cálculo de la superficie con los datos de altura y superficie referenciales entre los que se encuentra la altura actual mediante una interpolación. En el caso que el valor de la altura de la marea sea igual al valor referencial, se toma la superficie referencial como dato. Por otro lado, en el caso que la altura se salga de los datos, es decir, sea mayor que el valor de pleamar (cuando la marea se encuentra más alta) o menor que el valor de bajamar (cuando la marea se encuentra más baja), entonces se realiza una extrapolación a partir de los dos puntos mayores (pleamar y pleamar media) o menores (bajamar y bajamar media), dependiendo del caso.

Este cálculo se realiza con todos los sectores mencionados en la Tabla 1 de la playa, para, posteriormente, sumar los datos de esos sectores agrupándolos en las zonas como se ha indicado anteriormente en la Tabla 3.

Como puede verse en la Ilustración 21, inicialmente se planteó calculando la superficie de cada sector por separado, y luego en código habría que sumar los valores de superficie de los sectores. Esto se solucionó posteriormente, y se comentará más adelante.

```
15      public LasCanterasSurfaceModel(int sector) {
16          this.sector = LasCanterasSurfaceModel.sectors.get(sector);
17          this.sectorNum = sector;
18      }
19
20      public double surface(double seaHeight) {
21          return calculateWith(seaHeight);
22      }
23
24      private double calculateWith(double seaHeight) {
25          return interpolation(definedBy(seaHeight)).calculate(seaHeight);
26      }
27
28      private Interpolation interpolation(int index) {
29          index = (index > 0) ? index - 1 : index;
30          return new Interpolation(LasCanterasSurfaceModel.heights[index], LasCanterasSurfaceModel.heights[index + 1], sector[index], sector[index + 1], sectorNum);
31      }
32
33      private int definedBy(double seaHeight) {
34          OptionalInt indexOpt = IntStream.range(0, heights.length)
35                  .filter(i -> seaHeight < heights[i])
36                  .findFirst();
37
38          return (indexOpt.isPresent()) ? indexOpt.getAsInt() : 4;
39      }
```

Ilustración 21:Primera iteración del cálculo de superficie.

```
26      public double calculate(double height) {
27          double div = (y2-y1)/(x2-x1);
28          double result = ((y1+div*(height-x1))*1000d)/1000d;
29          result = (double) Math.round(result * 1000d) / 1000d;
30          saveOnDatalake(height,result);
31          return result;
32      }
```

Ilustración 22: Primera aproximación a la interpolación.

La implementación de la interpolación nombrada anteriormente se realizó siguiendo la herramienta de desarrollo TDD, es decir, Test-driven development, que se basa en realizar casos de pruebas para los requerimientos software del cliente durante el desarrollo, siendo una actividad conjunta a este en el que se prueba repetidamente el código con todos los casos de prueba [19]. Eso se hace siguiendo el siclo "Red-Green-Refactor", en el que primero se realiza un test que falle (Red). Tras fallar se realiza el mínimo cambio para que el test funcione sin que fallen todos los tests anteriores (Green), y finalmente se realiza una refactorización (Refactor), explicada anteriormente.

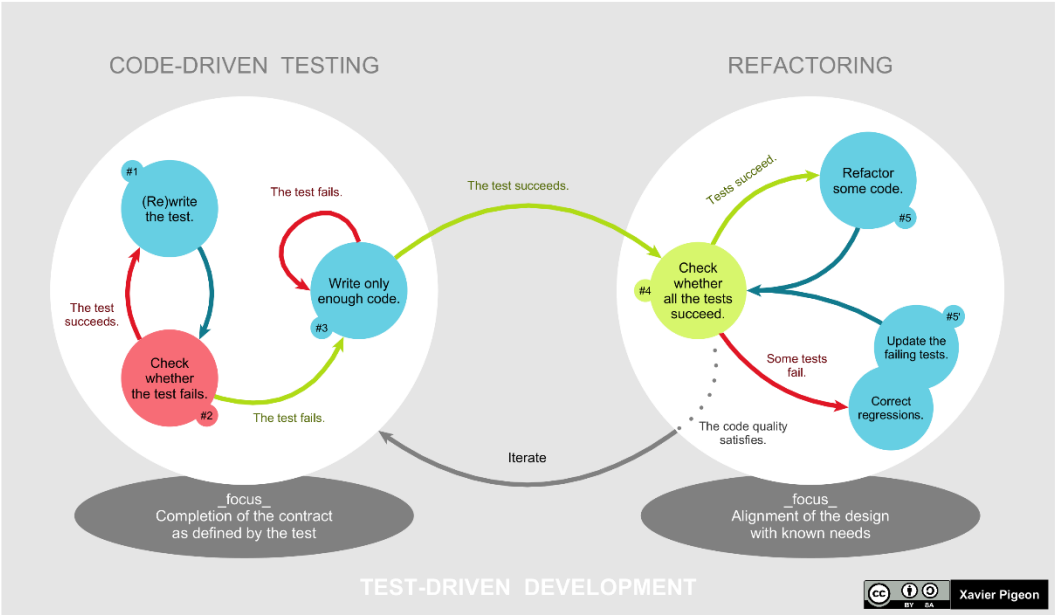


Ilustración 23: Ciclo de TDD [20]

Una parte importante del ciclo es la utilización de repositorios a los que subir los cambios realizados. En cuanto las subidas del código al repositorio, se puede hacer tras cada fase del ciclo, que sería la forma más correcta y segura de hacerlo, o tras cada ciclo. Una vez acabado un ciclo se vuelve a la fase Red hasta completar la implementación y se considere la funcionalidad completada y totalmente probada.

Finalmente, el usuario pudo ver los datos calculados en crudo, ya que todavía no se había comenzado la parte del frontend de la aplicación, pero se ha avanzado en el desarrollo de la aplicación.

Cuarta iteración

En la reunión de comprobación también se planteó la creación de un datalake, para el módulo de persistencia, en el que guardar los datos recabados por la aplicación, así como se planteó la implementación de una funcionalidad, en el módulo del temporizador, que fuera guardando en el datalake los datos del tiempo, el tipo de medición y la marea, así como la salinidad, la temperatura del aire y del agua, la altura del oleaje y el periodo del oleaje, que todavía no se habían desechado de la aplicación. Pero ¿qué es un datalake?

Un datalake es un sistema o repositorio de datos guardados en su formato original, conteniendo una gran cantidad de datos en bruto que se mantienen en él mientras sean necesarios, usado habitualmente con otras herramientas de Big Data [21] [22].

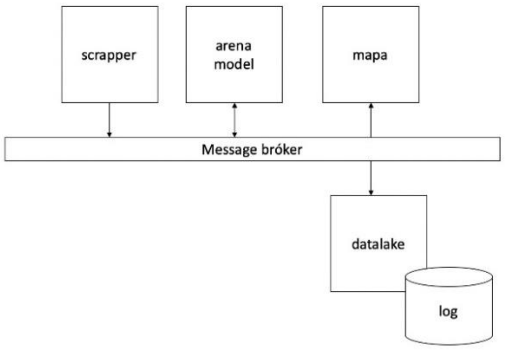


Ilustración 24: Planteamiento inicial de la aplicación.

Tras la implementación del datalake, se diseñó e implementó un temporizador destinado a guardar los datos en el datalake. Cuando el temporizador se activa, guarda en el datalake los datos recabados por la aplicación y mencionados anteriormente. Este timer servirá más adelante para la implementación del histórico de mediciones, y así, poder consultar datos pasados en la aplicación.

De esta forma, se planteó una estructura con un message broker central, es decir, un programa que traduce los mensajes de un lenguaje, que usa una parte de la aplicación, a otro lenguaje usado por otra parte del software [23], al cual llegan los datos desde el scraper, posteriormente lleva esos datos al modelo, que calcula la superficie de los sectores, y los lleva finalmente a el datalake, donde se guardan, y a la interfaz de usuario, que muestra los datos, como se puede ver en la Ilustración 24.

Finalmente, se comenzó a implementar la parte web de la aplicación, y, en un primer momento, se comenzó utilizando el framework Spark [24] para poder crear los endpoint, es decir, los puntos de conexión con el backend para obtener los datos, a los que accedería la aplicación.

```java
public static Route showInterpolation = (Request request, Response response) -> {
    Map<String, Object> model = new HashMap<>();
    int sector = Integer.parseInt(request.queryParams("sectors"));
    double height = Double.parseDouble(request.queryParams("height"));

    LasCanterasSurfaceModel lasCanterasSurfaceModel = new LasCanterasSurfaceModel(sector);
    double result = lasCanterasSurfaceModel.surface(height);

    model.put("sector", sector);
    model.put("height", height);
    model.put("surface", result);

    return new VelocityTemplateEngine().render(
            new ModelAndView(model, "velocity/index.vm"));
};
```

Ilustración 25: End-point para ver los datos de un sector en Spark.

```html
<body>
    <form action="/data">
        <label for="sectors">Sector (between 1 and 32):</label>
        <input type="number" id="sectors" name="sectors" min="1" max="32" value="1"></input>
        <label for="height">Tide height</label>
        <input type="number" id="height" name="height" step=".001"></input>
        <input type="submit"></input>
    </form>

    #if($surface)
        <table style="width:50%">
            <tr align="center">
                <th>Sector</th>
                <th>Height</th>
                <th>Surface</th>
            </tr>
            <tr align="center">
                <td>$sector</td>
                <td>$height</td>
                <td>$surface</td>
            </tr>
        </table>
    #end
</body>
```

Ilustración 26: Documento HTML para el frontend

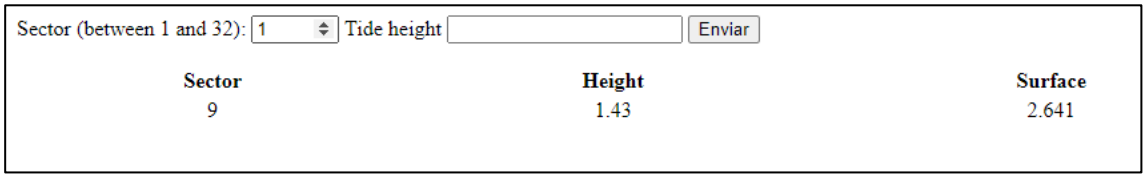


Ilustración 27: Vista inicial.

Inicialmente, y como puede verse en la Ilustración 26 y en la Ilustración 27, para la vista se creó un documento HTML en el que se indicaría el sector en el que se calcula la superficie y la altura de la marea, dado que todavía no se había implementado la conexión de los sistemas.

Tras una reunión para comprobar el correcto funcionamiento de los cambios, se propusieron nuevos cambios en la aplicación. En la reunión también se planteó el primer boceto de lo que sería la interfaz de usuario.

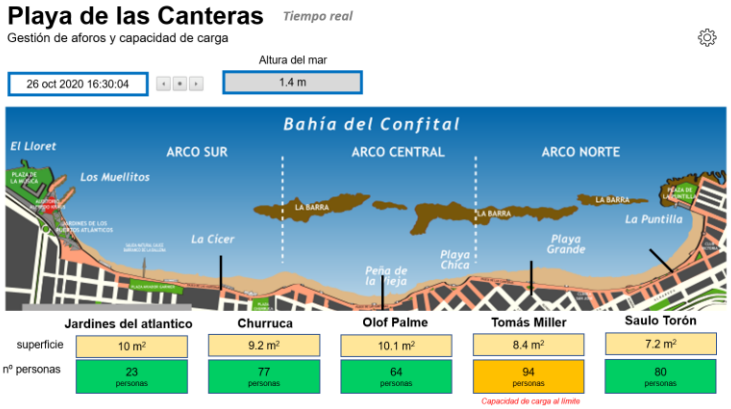


Ilustración 28: Boceto de interfaz de usuario 1

En primer lugar, y como puede verse en la Ilustración 28, se planteó una interfaz que mostrara un mapa de la playa de las Canteras, en el que puede verse la fecha de las mediciones en un cuadro, tres botones para atrasar la fecha una hora, ir a la hora actual y adelantar la fecha una hora, respectivamente, y la altura del mar de la hora relativa a la fecha en un cuadro.

En la parte de debajo del mapa pueden verse cuadros de texto que reflejaran la superficie calculada de cada zona de la playa y, debajo de cada superficie el número de personas de dicha zona.

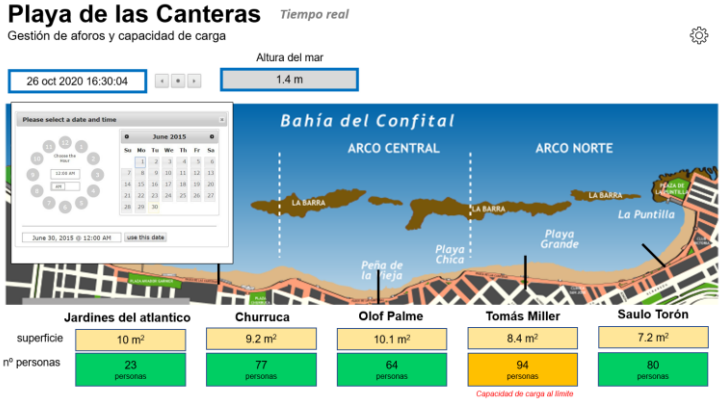


Ilustración 29:Boceto de interfaz de usuario 2

Adicionalmente, y como puede verse en la Ilustración 29, se podrá abrir un desplegable a partir de la fecha en el que se puede elegir la fecha y la hora con mayor facilidad, sin necesidad de navegar con los botones.

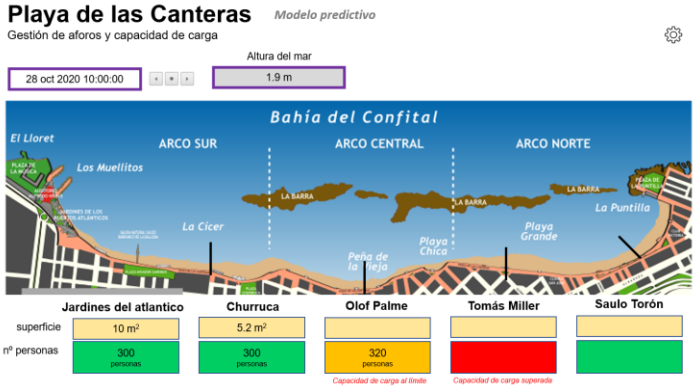


Ilustración 30:Boceto de interfaz de usuario 3

Finalmente, y como se puede ver en las ilustraciones anteriores, cada etapa temporal, es decir, el histórico, el tiempo real y el predictivo, tienen diferentes colores para diferenciarlos. El histórico tiene el color gris, el tiempo real el color azul, y el modelo predictivo un color morado. Adicionalmente, cuando una zona está acercándose al máximo de la capacidad de carga, el cuadro cambia a color amarillo y muestra un mensaje de advertencia, y cuando ha llegado al máximo, o se ha pasado de este, cambia a color rojo y muestra un nuevo mensaje de advertencia. Además, también se propuso un cuadro de configuración en el que el administrador podría indicar el aforo máximo permitido, pero esta idea se desechó posteriormente.

En cuanto a los cambios propuestos, el primero fue la adición al scraper de la recolección de la fecha y hora de las mediciones, datos que se encuentran al final de la respuesta de la web, como puede verse en la Ilustración 31. De esta forma, y como puede verse más adelante en la Ilustración 32, se comprueba si la línea leída es una fecha, y en ese caso la guarda en otra lista aparte, en la que estarían todas las fechas de las mediciones recogidas, en el mismo orden que las mediciones. Se detectó que las fechas estaban en orden contrario al que estaban las mediciones, así que antes de devolver las dos listas juntas, se le da la vuelta a la lista de los datos para tener primero las mediciones que se hicieron antes.

```
1649    "es.puertos.portus.gears.shared.ODate/1232930043",
1650    "15-06-2021 12:00:00",
1651    "es.puertos.portus.gears.shared.ODouble/4190077016",
1652    "15-06-2021 13:00:00",
1653    "15-06-2021 14:00:00",
1654    "15-06-2021 15:00:00",
1655    "15-06-2021 16:00:00",
1656    "15-06-2021 17:00:00",
1657    "15-06-2021 18:00:00",
```

Ilustración 31: Recorte de la respuesta mostrando las fechas

```
8     public static List[] extract_data_tides(String[] lines, String[] expect, int maxRow, int targetRow){
9         List<Double> data = new ArrayList<>();
10        List<String> dates = new ArrayList<>();
11        int index = 0, row = 0;
12        for (String line: lines) {
13            if (isDate(line)) {
14                dates.add(line);
15            }else {
16                if (index == 2 && row < maxRow) {
17                    if (row == 0 && !line.equals("13")) {
18                        index = 0;
19                    } else {
20                        if (row == targetRow)
21                            data.add(parseDouble(line));
22                        index = 0;
23                        row++;
24                    }
25                } else if (row == maxRow) row = 0;
26                else if (line.equals(expect[index])) index++;
27                else {
28                    index = 0;
29                }
30            }
31        }
32        Collections.reverse(data);
33        return new List[]{data,dates};
34    }
```

Ilustración 32: Nueva aproximación al scraper con fechas añadidas.

Adicionalmente, también se propuso cambiar la forma en la que se calcula la superficie de una zona. Se añadiría una clase encargada de sumar los datos calculados de todos los sectores de una zona, posteriormente redondearía el resultado y lo devolvería para continuar el programa.

```
18      public double calculateSurfaces(double seaHeight) {
19          double surface = 0;
20          sectors = zoneSurface.get(zone);
21          for (int i = 0; i < sectors.length; i++) {
22              LasCanterasSurfaceModel lcsm = new LasCanterasSurfaceModel(sectors[i]);
23              surface+=lcsm.surface(seaHeight);
24          }
25          return (double) Math.round(surface * 1000d) / 1000d;
26      }
```

Ilustración 33: Código del cálculo de la superficie de una zona

También se comenzó la implementación de la interfaz descrita anteriormente, perteneciente al módulo de la vista. Esta se implementaría con los framework de Vue.js, que es un framework de JavaScript que facilita la implementación de interfaces de usuario y aplicaciones de una sola página [25], y BootstrapVue, un framework que usa Vue, que permite crear aplicaciones con un diseño web adaptable [26]. Para realizar la conexión con el frontend, esto es, la interfaz de usuario, más sencilla, y tras algunas pruebas y consideraciones, se desechó el uso de Spark y se pasó a usar Spring, un framework de desarrollo de aplicaciones [27], por tener un conocimiento más amplio en él.

De esta manera, se implementó la aplicación, desarrollando los diferentes endpoints, del módulo API, que serían necesarios para la aplicación final, y que usaría el frontend a través de estos, y se implementó la primera aproximación al prototipo propuesto en la reunión, como puede verse en la Ilustración 35, aunque todavía no se habían conectado ambas partes de la aplicación.

```
26      @RequestMapping(path = "/height", method = {RequestMethod.GET})
27      public ResponseEntity getHeight(@RequestParam(required = true) String date) {
28          try {
29              Double height = tidesPersistence.getHeight(date);
30
31              return (height != null) ? buildResponse(HttpStatus.NO_CONTENT, null) :
32                      buildResponse(HttpStatus.OK, "{\"height\": " + height + "}");
33          } catch (Exception e) {
34              return buildResponse(HttpStatus.CONFLICT, "{ \"message\": \"There was a problem, couldn't get height\" }");
35          }
36      }
```

Ilustración 34: Primera aproximación del endpoint de la altura del mar

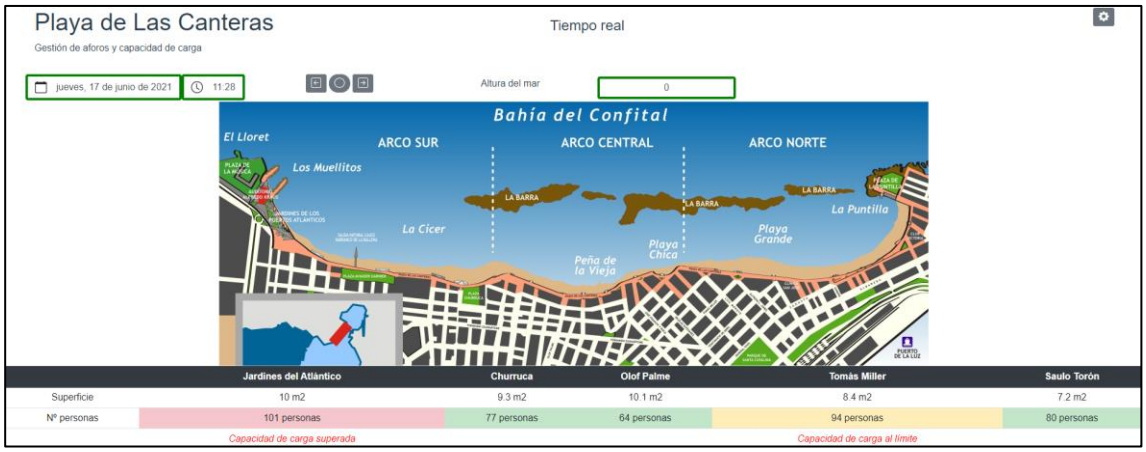


Ilustración 35: Primera aproximación del frontend

Finalmente, se creó una base de datos en la que guardar los datos recabados y calculados por la aplicación. Durante la iteración se presentaron varias versiones de la aplicación, en la que el usuario ha podido ver los datos de la aplicación en un acercamiento a la que sería la aplicación final, comenzando el módulo de vista y el de control en el frontend.

Quinta iteración

Durante la reunión de comprobación se propusieron cambios tanto para el backend como para el frontend. Además, durante el desarrollo de la aplicación surgieron cambios que se realizaron en el software, así como se realizó una refactorización del código.

Principalmente, se cambió el temporizador mencionado anteriormente. Con estos nuevos cambios, el temporizador se activará cada minuto para recoger los datos de la altura del mar y superficie de la playa. Cuando la hora marque las en punto, entonces el programa recogerá del scraper los valores de la última hora, hará la media de esos valores y guardará en el datalake los datos calculados con esa altura media. De esta manera, se ha conseguido tener un histórico que recoge los datos del datalake para mostrarlos.

Se decidió mostrar los datos cada hora, así como cambiar la navegación de los botones del frontend para avanzar o atrasar la hora una hora hacia adelante o hacia atrás respectivamente. Esto fue porque las predicciones usan este formato de las horas, es decir, las predicciones se crean de cada hora durante los próximos 3 días, entonces, para mantener la uniformidad y la coherencia de la aplicación, se decidió cambiar el histórico.

También se finalizó la API, es decir, la interfaz de programación de aplicaciones, que no es más que un conjunto de subrutinas, definiciones y protocolos usados para integrar el software en otras aplicaciones [28], además de realizar la corrección de diferentes errores en la aplicación. Así mismo, se cambió el frontend de forma que ahora, en lugar de mostrar el número de personas que hay en una zona, mostraría el número máximo de personas permitidas en esa zona. Este cambio se dio porque actualmente no hay forma de saber cuántas personas hay en cada zona de la playa. También se cambiaron los colores inicialmente propuestos, aunque el gris para el histórico y el morado para la predicción se mantuvieron, se decidió optar por el verde para reflejar el tiempo real, dado que da una mejor impresión de la etapa temporal a la que se refiere que el azul por el que se optó en un primer momento.

Adicionalmente, se comenzó la implementación de las restricciones a la capacidad de carga mencionadas anteriormente. Inicialmente, tendrían el tipo de restricción, que puede ser General, de Covid o de capacidad de carga, indicando el origen de la restricción, y un valor que indica el número de personas que caben en un metro cuadrado, y que afecta al número de personas máximo que puede haber en una zona. También se decidió eliminar el mapa de la aplicación, ya que se consideró que entorpece la interpretación de los datos y no resulta útil al usuario.

Con esto implementado, se calcularían las diferentes superficies y el número de personas de cada zona y se devolverían al frontend cuando pida los datos de cualquier instante de tiempo, como puede verse en la Ilustración 36.

	Jardines del Atlántico	Churruca	Olof Palme	Tomás Miller	Saulo Torón
Superficie	28424.13	21460.65	11790.918	18250.475	42790.978
Aforo general	2842	2146	1179	1825	4279
Aforo según COVID	1136	858	471	730	1711
Aforo según capacidad de carga	1894	1430	786	1216	2852
Aforo recomendad	1136	858	471	730	1711

Ilustración 36: Captura de la aplicación en la sexta iteración

Como puede observarse, se muestra la superficie de cada una de las zonas y el aforo de esa superficie, y debajo se muestran los diferentes aforos calculados por las restricciones mencionadas anteriormente.

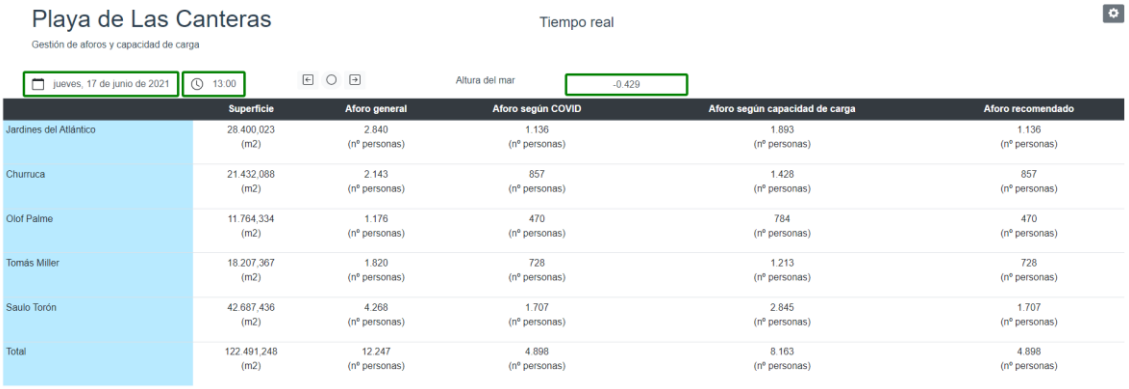
Posteriormente, se decidió cambiar la visualización de la aplicación de forma que las columnas de la tabla pasarían a ser las filas de la tabla, y viceversa, como puede verse en la Ilustración 38, pero este cambio sería revertido más adelante. Esta vez, para resaltar la importancia de las zonas y el total, ahora en cada fila, se decidió añadir un fondo azul a los nombres de éstos. También se añadieron las unidades de las medidas, para que el usuario pudiera entender mejor los datos que está viendo, aparte de añadir una fila para mostrar los totales de los valores calculados, relativos a toda la playa, y se decidió redondear los valores en la tabla para una mejor visualización. En el caso de la altura, se decidió optar por tres decimales, mientras que, para la superficie, al ser un valor muy grande, se optó por no mostrar los decimales, redondeando el valor.

Finalmente, se añadió la conexión de ambas partes de la aplicación, el frontend y el backend. Por una parte, el backend ya lo tenía implementado con la implementación de la API, como se describió anteriormente.

Por otro lado, en el frontend se creó el módulo de la API para que pudiera comunicarse con el backend y conseguir los datos necesarios para la aplicación. Como puede verse en la Ilustración 37, en la línea 12 se indica el endpoint del backend al que se va a conectar, añadiendo los campos necesarios para saber el momento que se quiere calcular. Posteriormente, en la línea 13 es cuando se realiza la llamada al endpoint, para luego devolverlo a la aplicación.

```
3  const API_URL = 'http://localhost:8081/tides'
4
5  const headers = {
6      'Content-Type': 'application/json'
7  }
8
9  export class APITides {
10
11     async getData(date, hour, dataType) {
12         const url = `${API_URL}/` + dataType + `?date=`+ date +'&hour='+ hour;
13         return await axios.get(url, {
14             headers: headers
15         }).then((response) => response).catch( error => { console.log(error); });
16     }
17 }
```

Ilustración 37: API en el frontend



	Superficie	Aforo general	Aforo según COVID	Aforo según capacidad de carga	Aforo recomendado
Jardines del Atlántico	28.400,023 (m2)	2.840 (nº personas)	1.136 (nº personas)	1.893 (nº personas)	1.136 (nº personas)
Churruca	21.432,088 (m2)	2.143 (nº personas)	857 (nº personas)	1.428 (nº personas)	857 (nº personas)
Olof Palme	11.764,334 (m2)	1.176 (nº personas)	470 (nº personas)	784 (nº personas)	470 (nº personas)
Tomás Miller	18.207,367 (m2)	1.820 (nº personas)	728 (nº personas)	1.213 (nº personas)	728 (nº personas)
Saulo Torón	42.687,436 (m2)	4.268 (nº personas)	1.707 (nº personas)	2.845 (nº personas)	1.707 (nº personas)
Total	122.491,248 (m2)	12.247 (nº personas)	4.898 (nº personas)	8.163 (nº personas)	4.898 (nº personas)

Ilustración 38: Aplicación con las filas y columnas intercambiadas

Sexta iteración

En la reunión para comprobar el correcto funcionamiento, revisar y proponer nuevos cambios, se decidió, como se mencionó anteriormente, revertir la posición de las filas y tablas a como estaba anteriormente, con las zonas y los totales como columnas, y los datos calculados en las filas, quedando como se puede ver en la Ilustración 38. Adicionalmente, se cambió el estilo de la aplicación, quitando el fondo azul en las filas y el borde verde en el tiempo y la medición de altura dado que no llegaba a ser agradable para la vista, y podría ser confuso para el usuario. Alternativamente, se añadió una línea, que tendría el color representativo de la etapa en la que estuviera la medición, que separaría la fecha y los botones de la tabla, sirviendo a la vez como inicio a la misma. De esta forma quedaría una mejor visualización, más limpia y clara para el usuario.

Además, la medición de la altura se introdujo en la tabla para que pudiera tener una mejor visualización del dato y la superficie se pasó a la parte alta de la tabla, para tener una separación entre los datos de superficie y de capacidad de carga y número de personas.

Altura del mar			-0.355			
Superficie	119.937,252 (m2)	28.127,477 (nº personas)	21.109,162 (nº personas)	11.463,791 (nº personas)	17.720,008 (nº personas)	41.516,814 (nº personas)
	Total	**Jardines del Atlántico**	**Churruca**	**Olof Palme**	**Tomás Miller**	**Saulo Torón**
Aforo recomendado	4.797,49 (m2)	1.125,099 (nº personas)	844,366 (nº personas)	458,552 (nº personas)	708,8 (nº personas)	1.660,673 (nº personas)
Criterio socal: seguridad Distancia de seguridad 3 m	4.797,49 (m2)	1.125,099 (nº personas)	844,366 (nº personas)	458,552 (nº personas)	708,8 (nº personas)	1.660,673 (nº personas)
Criterio socal: seguridad Vías de evacuación	11.993,725 (m2)	2.812,748 (nº personas)	2.110,916 (nº personas)	1.146,379 (nº personas)	1.772,001 (nº personas)	4.151,681 (nº personas)
Criterio ambiental Conservación de la fauna	7.995,817 (m2)	1.875,165 (nº personas)	1.407,277 (nº personas)	764,253 (nº personas)	1.181,334 (nº personas)	2.767,788 (nº personas)

Ilustración 39: Aplicación revertida al original.

Como puede verse en la ilustración anterior, se cambiaron las restricciones para indicar qué tipo de criterio sigue la restricción y un nombre identificativo de la misma.

En cuanto al código, en la Ilustración 40 puede verse el método que se encarga de llamar a la API dentro del frontend, para luego detectar el periodo de tiempo en el que está la hora, para cambiar el color de la línea, y para actualizar los datos de la tabla con los nuevos datos recabados.

```
updateComponents(date, hour, dataType){
    console.log(date + " - " + hour)

    var apiService = new APITides();

    apiService.getData(date, hour, dataType).then((response) => {
      if (response.status == 200) {
        var color = this.$root.$refs.time.getColor();
        this.$root.$refs.table.setColor(color);
        this.$root.$refs.table.updateTable(response.data);
      }else{
          this.tide_height = 'no-content'
    }}).catch(error => {alert(error);});
}
```

Ilustración 40: Método encargado de llamar a la API del frontend

Adicionalmente, se realizó una refactorización en las restricciones, realizando los cambios en el módulo del modelo, de forma que existiría un archivo en el que se indican las restricciones a utilizar. El formato de este archivo, como puede verse en la Ilustración 41, contiene en cada línea una restricción. Cada línea, separada por ';', se divide de la siguiente manera: en primer lugar, el nombre de la restricción, en segundo el tipo de criterio, en tercer lugar, si está activo inicialmente, eso sirve para un cambio realizado que explicaremos más adelante, y como último dato el tipo de cálculo que se realiza con el dato, junto con el número que indica cómo afecta el dato. El tipo de dato se refiere a cómo afecta el dato a la capacidad de carga, por ejemplo, si es de densidad, el que se ve en el ejemplo, entonces el dato se aplica sobre la capacidad de carga directamente, pero también puede ser un valor absoluto del aforo máximo, este dato sustituiría a la capacidad de carga calculada.

```
Distancia de seguridad 4m;Criterio social:Seguridad;true;DensityCapacityCalculator:0.4
Distancia de seguridad 10m;Criterio social:Seguridad;false;DensityCapacityCalculator:0.01
```

Ilustración 41: Archivo de restricciones.

En cuanto al parámetro que indica si la restricción está activa inicialmente, en la reunión se propuso la adición de unas cajas selectoras, que indican si cada restricción afecta a la capacidad de carga. Como se mencionó anteriormente, la capacidad de carga tiene una estructura de escalera, en la que el valor de una se calcula a partir del anterior. De esta manera, si una restricción afecta a la capacidad de carga real, la capacidad de carga efectiva se verá afectada también, pero la capacidad de carga física seguirá igual. De esta manera, En la Ilustración 42, se puede ver el prototipo creado en la reunión de los nuevos cambios.

Ilustración 42: Prototipo para la séptima iteración

Inicialmente, y como puede verse en la Ilustración 42, solo se añadieron la capacidad de carga física y la capacidad de carga real, siendo las restricciones los metros cuadrados por persona disponibles y cada uno con sus cajas seleccionables para activarlas o desactivarlas y así afectaran a la capacidad de carga real, cambiando acordemente. Posteriormente se añadió también la capacidad de carga efectiva. De la misma manera, se descartó el cuadro de configuración de la parte superior derecha, dado que no se consideró útil su implementación como un elemento que estuviera oculto.

Finalmente, junto con algunos cambios de visualización, como son el tamaño de algunas secciones de la tabla y correcciones a confusiones de los datos, también se cambió el formato de las restricciones, tras una reunión se consideró que el formato, que se puede ver en la Ilustración 43, sería mejor.

```
1  Presencia de medusas(50m<sup>2</sup> por persona);FCCR;Seguridad;true;Density:0.06
2  Socorristas(2 socorristas por zona);FCCE;Seguridad;false;Density:0.05
3  Socorristas(3 socorristas por zona);FCCE;Seguridad;false;Density:0.04
4  Recogida de residuos(500m<sup>2</sup> por papelera);FCCE;Limpieza;false;Density:0.4
```

Ilustración 43: Nuevo formato de las restricciones

Básicamente, y como puede verse, el formato es muy parecido, manteniéndose el nombre identificador, aunque en esta ocasión le acompaña un indicador del dato que se añade. Siguiendo el nombre, se añadió otro campo que indica a qué capacidad de carga afecta, por ejemplo, FCCR significa Factor de Capacidad de Carga Real y FCCE a la efectiva. Posteriormente los datos siguen siendo el criterio, si está activo o no al iniciar la aplicación, y finalmente el dato indicativo de cómo afecta a la capacidad de carga, aunque en esta ocasión el nombre es más sencillo de escribir y entender para el usuario. En este archivo, como se mencionó anteriormente, se pueden añadir múltiples restricciones para que aparezcan en la aplicación. Las existentes en la actualidad se han añadido a modo de ejemplo para que se pueda ver la versatilidad de esta funcionalidad.

Al iniciar la aplicación, el software se encarga de leer las líneas del archivo, como puede verse en la Ilustración 44. Posteriormente, separa la línea leída por el ‘;’ elegido como carácter separador de los datos del archivo y, como puede verse en la línea 32, crea la restricción y crea el objeto de la clase CapacityCalculator, dentro del módulo del modelo, que es la clase encargada de calcular la capacidad de carga de las zonas siguiendo la restricción.

```java
@Override
public List<Restriction> load() {
    List<Restriction> restrictions = new ArrayList<>();
    try(BufferedReader bufferedReader = new BufferedReader(new FileReader(file))){

        for (String l : bufferedReader.lines().collect(Collectors.toList())) {
            if(l.equals("")) continue;
            String[] line = l.split(";");
            String[] capacityCalculator = line[4].split(":");
            restrictions.add(new Restriction(line[0],line[1],line[2],Boolean.parseBoolean(line[3]),
                    Double.parseDouble(capacityCalculator[1]), capacityCalculator[0],
                    new CapacityCalculatorFactory().build(capacityCalculator[0], Double.parseDouble(capacityCalculator[1])));
        }

    }catch (Exception e){
        e.printStackTrace();
        return null;
    }
    return restrictions;
}
```

Ilustración 44: Método de carga de restricciones en la clase FileRestrictionLoader

La creación se hace siguiendo el patrón factory, que es un patrón de diseño creacional que sirve para construir una jerarquía de clases [29]. De esta manera, con el nombre identificativo crea el CapacityCalculator requerido.

```java
public class CapacityCalculatorFactory {

    public CapacityCalculator build(String name, Double value){
        if(name.equals("Density"))
            return new DensityCapacityCalculator(value);
        return null;
    }
}
```

Ilustración 45: Primera aproximación de la CapacityCalculatorFactory

Todos los CapacityCalculator implementan la interfaz homónima. Una interfaz es una clase que indica qué debe hacer la clase que la implementa, pero no indica cómo debe hacerlo [30]. En la interfaz, se ha indicado el método *calculate*, que recibe una lista de las superficies de las áreas y devuelve las superficies calculadas.

Se implementaron los calculadores para las tres distintas capacidades de carga y se añadió el calculador de densidad, que se usa en las restricciones. En la Ilustración 46, se puede ver la implementación del método *calculate*, encargado de calcular la capacidad de carga con la nueva densidad.

```java
@Override
public List<Integer> calculate(List<Integer> areas) {
    return areas.stream().mapToInt(a-> (int) (a*density)).boxed().collect(Collectors.toList());
}
```

Ilustración 46: Método calculate de la clase DensityCapacityCalculator

Adicionalmente, se añadió un cuadro de texto en el que se podría indicar el valor del calculador directamente en la aplicación, así como el factor al que afecta, como puede verse en la Ilustración 47. También se añadió a la tabla la capacidad de carga efectiva, que no estaba presente anteriormente. Finalmente, se decidió pasar el texto indicativo de la etapa de tiempo a la parte izquierda de la pantalla y se colocaron el cuadro de la fecha, la hora y los botones de movimiento en el centro, para que fuera más intuitivo el uso de los mismos. Asimismo, se decidió eliminar las unidades de las mediciones ya que llenaban la pantalla y volvían los datos más difíciles de entender.

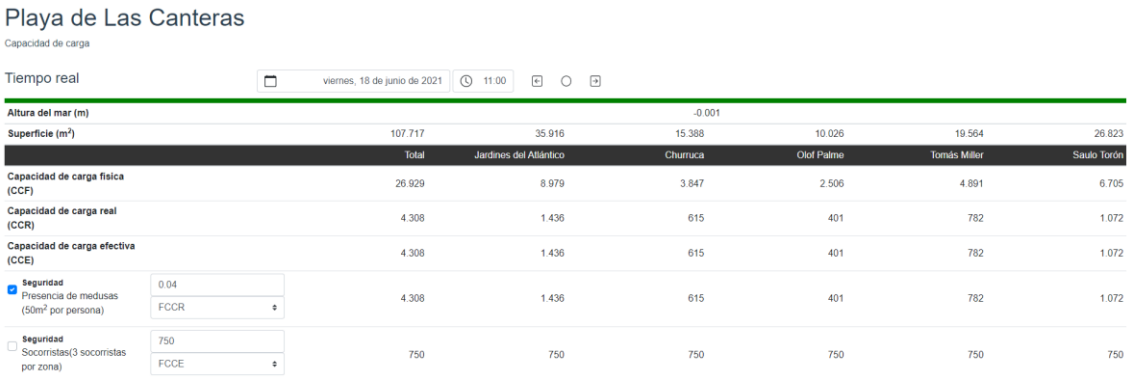


Ilustración 47: Prototipo con cuadros para cambiar la densidad

Si en la ilustación anterior el usuario cambia el valor del factor de capacidad de carga, se abre un cuadro que nos muestra las opciones, al cambiarlo se puede ver en la Ilustración 48, cómo ha dejado de afectar a la capacidad de carga real, pero sigue afectando a la capacidad de carga efectiva. En caso de marcar las dos restricciones, se decidió que la que se aplicaría sería la más restrictiva, en caso de afectar al mismo nivel de capacidad de carga.

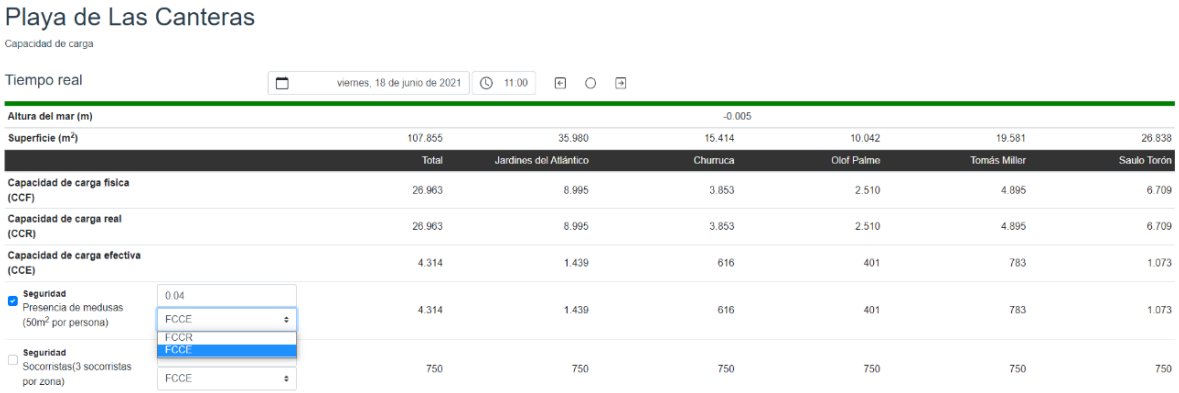


Ilustración 48: Ejemplo de cambio de factor de capacidad de carga

También se añadió un nuevo CapacityCalculator, el cual es AbsoluteCapacityCalculator, en el que se le indica un valor que será el número máximo de personas que puede haber por zonas. Este calculador es diferente al anterior ya que, en lugar de calcular sobre la capacidad de carga ya existente, sustituye el valor por el indicado.

En cuanto al frontend, para las restricciones se creó otra API en la que se añadieron varios métodos destinados a el manejo de las mismas. En la Ilustración 49 pueden verse los dos primeros métodos. El primero, llamado *setRestriction* está destinado a cambiar el estado de una restricción de activa a desactiva, y viceversa, esto se hace mediante las cajas de verificación que se encuentran en la parte izquierda de las restricciones. Por otro lado, el segundo método, *getActiveRestrictions*, está destinado a pedirle a la aplicación las restricciones que estén activas en ese momento, como el nombre indica.

```
11    async setRestriction(restriction) {
12        const url = `${API_URL}/update`;
13        var data = {
14            restrictions: restriction
15        }
16        return await axios.put(url, data, {
17            headers: headers
18        }).then((response) => response).catch( error => { console.log(error); });
19    }
20
21    async getActiveRestrictions() {
22        const url = `${API_URL}/actives`;
23        return await axios.get(url, {
24            headers: headers
25        }).then((response) => response).catch( error => { console.log(error); });
26    }
```

Ilustración 49: Dos primeros métodos de la API de restricciones.

En la Ilustración 50, se pueden ver los otros dos métodos de la API, el primero, getRestrictionsData, es el encargado de pedirle al backend los datos de todas las restricciones existentes en la aplicación, mientras que el segundo método, llamado modifyRestriction, está destinado a cambiar los valores de la restricción cuando el usuario realiza algún cambio en la aplicación.

```
28    async getRestrictionsData() {
29        const url = `${API_URL}/data`;
30        return await axios.get(url, {
31            headers: headers
32        }).then((response) => response).catch( error => { console.log(error); });
33    }
34
35    async modifyRestriction(restriction, index) {
36        const url = `${API_URL}/update/` + index;
37        return await axios.put(url, restriction, {
38            headers: headers
39        }).then((response) => response).catch( error => { console.log(error); });
40    }
```

Ilustración 50: Dos últimos métodos de la API de restricciones.

Séptima iteración

Tras una reunión se decidió cambiar de nuevo la visualización de las restricciones. Finalmente, las restricciones serían estáticas, es decir, no cambiarían su valor durante la ejecución de la aplicación, ni cambiaría a qué capacidad de carga afecta. Esto estaría indicado en el fichero de configuración de las restricciones y no cambiaría a no ser que se modifique dicho fichero.

Por otro lado, se añadió un cuadro de texto al lado de la capacidad de carga física para poder cambiar la densidad de personas por metro cuadrado que se permiten. De esta manera, se puede cambiar la capacidad de carga física en función de las necesidades de la playa y los usuarios, y, posteriormente, se cambió para indicarlo en metros cuadrados por persona, en lugar de ser al revés. Adicionalmente, se añadió en la capacidad de carga física la unidad bajo la que opera el dato, para una mayor facilidad de interpretación.

Más adelante, se detectó un error consistente en que la web de Portus, como se explicó en el capítulo de Estado actual y objetivos iniciales, la página web no tiene en cuenta los cambios de hora para ahorrar en el gasto de luz que vienen con los solsticios, de forma que en ocasiones la hora de los datos está una hora atrasada con la hora de la aplicación, haciendo que no concuerde con los datos de la misma. Este error fue corregido, añadiendo un método (Ilustración 51) que compruebe si la fecha actual se encuentra en la época de ahorro

energético. Como puede verse, en la línea 88 llama al método que comprueba si se encuentra en dicha época, y de ser así corrige la hora para el correcto funcionamiento de la aplicación.

```
86      private String dateToStringForPortus() {
87          int currentHour = currentTime.getHour();
88          if(isDaylightSavingTime()) currentHour--;
89          return formatNumberOfDate(currentTime.getDayOfMonth()) + "-" + formatNumberOfDate(currentTime.getMonthValue())
90                  + "-" + currentTime.getYear() + " " + formatNumberOfDate(currentHour);
91      }
92
93      private boolean isDaylightSavingTime() {
94          ZonedDateTime now = getZonedDateTime();
95          return now.getZone().getRules().isDaylightSavings( now.toInstant() );
96      }
```

Ilustración 51: Método de comprobación del cambio horario

Al finalizar esta iteración se le entregó al usuario un producto útil y que se ajusta a sus necesidades, siguiendo un proceso iterativo e incremental, en el que se fueron arreglando errores y mejorando la aplicación. Esto último fue más notable en las últimas iteraciones, en las que se finalizó la aplicación, arreglando los últimos errores y añadiendo las funcionalidades restantes.

Secciones más relevantes del código

El código quedó de la siguiente manera:

```
19      public Function<Double, Double> calculate() {
20          double div = (y2-y1)/(x2-x1);
21          return x -> (double) Math.round((((y1 + div * (x - x1)) * NUMBER_OF_DECIMALS) / NUMBER_OF_DECIMALS) *
22                  NUMBER_OF_DECIMALS) / NUMBER_OF_DECIMALS;
23      }
```

Ilustración 52: Código final de la interpolación. Clase Interpolation.

Como se puede observar en la Ilustración 52, se cambió el método de forma que ahora se crea una función con los parámetros que se le indican al crear el objeto, y luego en el código se le puede aplicar un valor a esa función.

```
17      public double surface(double seaHeight) { return calculateWith(seaHeight); }
20
21      private double calculateWith(double seaHeight) {
22          return getInterpolation(definedBy(seaHeight)).calculate().apply(seaHeight);
23      }
24
25      private Interpolation getInterpolation(int index) {
26          index = fixIndex(index);
27          return new Interpolation(getHeightFromModel(index), getHeightFromModel( index: index + 1),
28                  sector[index], sector[index + 1]);
29      }
30
31      private int definedBy(double seaHeight) {
32          OptionalInt indexOpt = findReferentialHeightIndex(seaHeight);
33          return (indexOpt.isPresent()) ? indexOpt.getAsInt() : 4;
34      }
```

Ilustración 53: Código final del cálculo de la superficie de un sector. Clase LasCanterasSurfaceModel.

En el código de la Ilustración 53 se puede ver como dada una superficie en un sector determinado, el que se indica al crear el objeto, se crea la interpolación con los valores referenciales de altura y superficie, para posteriormente calcular el valor de la superficie mediante la función explicada anteriormente.

```java
10      public List<Integer> calculateSurfaces(double seaHeight) {
11          return zones.stream()
12                  .map(sectors -> round(calculateSurface(seaHeight, sectors)))
13                  .collect(Collectors.toList());
14      }
15
16      private double calculateSurface(double seaHeight, int[] sectors) {
17          return Arrays.stream(sectors)
18                  .mapToDouble(sector -> calculateSurface(seaHeight, sector))
19                  .sum();
20      }
21
22      private double calculateSurface(double seaHeight, int surface) {
23          return new LasCanterasSurfaceModel(surface).surface(seaHeight);
24      }
```

Ilustración 54: Código final del cálculo de la superficie de las zonas. Clase LasCanterasZonesModel

En la ilustración anterior puede observarse cómo se recorren todas las zonas y se calculan las superficies de todas ellas. El método *calculateSurfaces*, de la línea 10 es el encargado de recorrer las zonas y llamar al método homónimo de la línea 16, que calcula la superficie de una zona sumando la superficie calculada de los sectores de una misma zona.

```java
13  public static double extract_data_tides(String[] lines, String[] code, int maxRow, int targetRow, String date) {
14      List<Double> data = new ArrayList<>();
15      List<String> dates = getTimeStamps(lines);
16
17      int codePosition = 0, row = 0;
18      for (String line : lines) {
19          if (isCodeApproved(codePosition) && !endOfRecord(maxRow, row)) {
20              if (isSpecialValue(row, line)) codePosition = 0;
21              else {
22                  if (isTargetRow(targetRow, row)) data.add(parseDouble(line));
23                  codePosition = 0;
24                  row++;
25              }
26          } else if (endOfRecord(maxRow, row)) row = 0;
27          else if (isPartOfCode(line, code[codePosition])) codePosition++;
28          else codePosition = 0;
29      }
30
31      return round(getAverageForDate(date, data, dates));
32  }
```

Ilustración 55: Código final del extractor de los datos. Clase DataTidesExtractor

En la ilustración anterior puede verse el método encargado de extraer los datos de la respuesta de PORTUS, interpretándolos y normalizarlos para su manejo posterior. Básicamente, como puede observarse en Ilustración 55, el funcionamiento y el código son los mismos que los explicados anteriormente, pero se ha refactorizado el código para una mejor lectura del mismo.

```java
20      public static SeaHeightScrapper getInstance() {
21          if (seaHeightScrapper == null)
22              seaHeightScrapper = new SeaHeightScrapper();
23          return seaHeightScrapper;
24      }
25
26      public void logTide(double tide) {
27          if(currentHoursMeasures.size() == 60 || currentTime.isBefore(getCurrentTime())){
28              currentHoursMeasures.clear();
29          }
30          currentHoursMeasures.add(tide);
31      }
```

Ilustración 56: Parte del código final del timer. Clase SeaHeightScrapper

El código que se puede ver en la Ilustración 56 es el encargado de recoger los valores de la marea que se van calculando, y guardarlos en una lista de valores durante la hora. Cuando sea la hora en punto, borra los datos de la lista, que ya han sido usados y guardados en el datalake.

```
10      public void timeout(){
11          Timer timer = new Timer( isDaemon: true);
12          timer.schedule(new TimerTask(new TideMeasures(), SeaHeightScrapper.getInstance()), delay: 0, period: 60*1000);
13      }
```

Ilustración 57: Código final de la inicialización del timer. Clase TimeoutWriter

En la Ilustración 57 puede observarse, en la línea 12, cómo se inicializa el timer, indicándole la acción que hará cuando se active, y cada cuando se active. En este caso se activa cada minuto.

La clase TimerTask, usada en la anterior ilustración y que se verá más adelante, es una implementación de la clase abstracta ya existente del mismo nombre. Una clase abstracta es una clase en la que se indican los métodos que deben existir en la clase que los extiende, pero no el funcionamiento del método, aunque puede contener métodos con implementaciones. Las clases no pueden extender a varias clases abstractas al mismo tiempo, esto se conoce como herencia [31].

```
28          @Override
29          public void run() {
30              if (timeNotLogged()) currentTime = getCurrentTime();
31              double tide = getTideMeasure();
32              if (hoursHasChanged()) persistMeasures();
33              if(tide != -40.) logTideMeasure(tide);
34          }
```

Ilustración 58: Código final de la clase TimerTask.

En la Ilustración 58 se puede apreciar la implementación final de la clase TimerTask, mencionada anteriormente. Como se puede observar, se recoge el valor de la marea del momento actual, posteriormente comprueba si la hora ha cambiado, para, de ser así, guardar los datos de la última hora y borrarlos, para luego, como se puede ver en la Ilustración 59, guardar en el SeaHeightScrapper el valor de la marea actual en el método persistMeasures, en el que se obtiene la media de las alturas y se guarda en el datalake el dato con la superficie de la media calculada.

```
44      private void logTideMeasure(double tide) {
45          currentTime = getCurrentTime();
46          seaHeightScrapper.logTide(tide);
47      }
48
49      private double getTideMeasure() { return tideMeasures.getMeasures(dateToStringForPortus()); }
52
53      private void persistMeasures() {
54          double meanHeight = getMeanHeight();
55          persistTide(meanHeight, getCanterasSurfaces(meanHeight));
56          currentTime = getCurrentTime();
57      }
```

Ilustración 59: Métodos de guardar las mareas de la última hora y guardar la marea actual, en las líneas 53 y 44 respectivamente.

En cuanto a los calculadores de capacidad de carga, esta parte no ha visto cambios significativos como para mostrar todo el código. Como se mencionó en el apartado de desarrollo, en la séptima iteración (pág. 30), los calculadores sirven para calcular la capacidad de carga a partir de la superficie de las áreas nombradas anteriormente y del valor de la restricción. Los distintos niveles de la capacidad de carga tienen calculadores también, y en la puede verse el calculador de la capacidad de carga real, que filtra las restricciones existentes para saber cuáles afectan a la capacidad de carga, y posteriormente aplica la que menor valor tenga, que será la más restrictiva. Finalmente, se comprueba si el valor nivel superior de capacidad de carga, en este caso, la capacidad de carga física es menor que el actual, dado que, como se mencionó en el apartado de estado actual

(pág. 3), la capacidad de carga sigue una estructura de escalera en la que cada nivel se ve afectado por el valor del anterior.

```
16    public List<Integer> calculate(List<Integer> areas){
17        return restrictions.stream() Stream<Restriction>
18                .filter(Restriction::isActive) Stream<Restriction>
19                .filter(restriction -> restriction.getLimitantTo().equals("FCCR")) Stream<Restriction>
20                .map(Restriction::getCapacityCalculator) Stream<CapacityCalculator>
21                .map(c-> c.calculate(areas)) Stream<List<Integer>>
22                .reduce(new PhysicalCapacityCalculator().calculate(areas), this::min);
23    }
```

Ilustración 60: Código final del calculador de capacidad de carga real. Clase RealCapacityCalculator.

Antes de finalizar el backend explicando los endpoints de la API, primero veremos una parte del código que todos los endpoints comparten. Esto es, las clases encargadas de crear las respuestas que se mandarán cuando se llame a la API.

Para la marea y los valores calculados a partir de ella, como son las superficies y los tres niveles de capacidad de carga, así como las diferentes restricciones se usa la clase *CustomTidesResponseEntity*. Básicamente, los datos que se verán en la tabla inicialmente. En el código de la Ilustración 61, podemos ver cómo se construye en cuerpo de la respuesta, creando una respuesta en formato JSON, que es un formato sencillo orientado al intercambio de datos entre sistemas [32], y devolviendo dicha respuesta ya construida.

```
15  public static ResponseEntity getResponseEntityWhenData(String tide, List<Integer> surface, List<Integer> physicalCapacity,
16                                                          List<Integer> recommendedCapacity, List<Integer> effective,
17                                                          List<Integer> ... args){
18      return getResponse(getBody(tide, surface, physicalCapacity, recommendedCapacity, effective, args));
19  }
20
21  public static ResponseEntity getResponseEntityWhenError() {
22      return buildResponse(HttpStatus.OK, BODY_WHEN_NO_DATA);
23  }
24
25  private static String getBody(String tide, List<Integer> surface, List<Integer> physicalCapacity,
26                                List<Integer> recommendedCapacity, List<Integer> effective,List<Integer> ...args) {
27      String baseBody = "{\"information\":[{\"tide\":\""+ tide +"\"},{\"superficie\":"+
28              surface + "},{\"aforo_fisico\":" + physicalCapacity +
29              "},{\"aforo_recomendado\": "+ recommendedCapacity +"},{\"aforo_efectivo\": " + effective;
30
31      for (int position = 0; position < args.length; position++) {
32          baseBody += "},{\"distancia\":" + args[position];
33      }
34      return baseBody + "]}";
35  }
```

Ilustración 61: Construcción de la respuesta. Clase CustomTidesResponseEntity.

Para conocer los valores de las restricciones y para manejar las restricciones se usa la clase *CustomRestrictionsResponseEntity*, que, como puede observarse en la Ilustración 62, recoge los datos de las restricciones y los devuelve, en el caso de que sean requeridos, adicionalmente, en la Ilustración 63 puede observarse los casos en los que se requieren solamente las restricciones activas, cuando ha habido un error o simplemente cuando se ha cambiado un valor de la restricción, pero no se piden los datos.

```
33  private static String getBody() {
34      double density = new PhysicalCapacityCalculator().getDensity();
35      String baseBody = "{\"CCFDensity\":"+ (1/density) +",\"restrictions\":[";
36
37      ObjectMapper mapper = new ObjectMapper();
38      for (Restriction restriction: RestrictionsCapacity.getInstance().getRestrictionsCapacity()) {
39          try {
40              baseBody += mapper.writeValueAsString(restriction) + ",";
41          } catch (JsonProcessingException e) {
42              e.printStackTrace();
43          }
44      }
45      return baseBody.substring(0, baseBody.length()-1) + "]}";
46  }
```

Ilustración 62: Código de la respuesta con las restricciones. Clase CustomRestrictionsResponseEntity.

```
14      private static final String ERROR_MESSAGE = "{\"message\":\"There was a problem, couldn't get information\"}";
15      private static final String NO_DATA_NO_ERROR = "{}";
16
17      public static ResponseEntity getResponseEntityWithEmptyBody() { return getResponse(NO_DATA_NO_ERROR); }
20
21      public static ResponseEntity getResponseEntityWhenError() {
22          return buildResponse(HttpStatus.CONFLICT, ERROR_MESSAGE);
23      }
24
25      public static ResponseEntity getResponseEntityWithRestrictions(List<Integer> active) {
26          return getResponse( body: "{\"selected\":" + active+"}");
27      }
```

Ilustración 63: Código con el resto de las respuestas de la clase CustomRestrictionsResponseEntity

Ambas clases mencionadas anteriormente son clases hijas de la clase *CustomResponseEntity*, encargada de crear la respuesta y la cabecera de la misma.

Para finalizar con los endpoints, la clase *TidesController* contiene los relativos a los datos mostrados en la aplicación, y contiene tres endpoints, uno para cada periodo de tiempo. Antes de finalizar cada endpoint, los tres llaman a un método encargado de calcular los datos que se han ido explicando durante esta memoria, y que se mostrarán en la aplicación. En primer lugar, para conseguir el histórico, el primer endpoint, al que se accede mediante el enlace *'/tides/historic'* y que se puede observar en la Ilustración 64. En él, se recoge del datalake, explicado en la quinta iteración (pág. 23), indicándole la fecha y hora de la medición objetivo, para luego construir la respuesta y devolverla.

```
29      @RequestMapping(path = "/historic", method = RequestMethod.GET)
30      public ResponseEntity getHistoricInfo(@RequestParam String date, @RequestParam String hour) {
31          try {
32              Double tide = datalakePersistence.getHeight( s: date + " " + hour);
33              return getResponseEntity(tide);
34          } catch (Exception e) {
35              return CustomTidesResponseEntity.getResponseEntityWhenError();
36          }
37      }
```

Ilustración 64: Endpoint del histórico. Clase TidesController.

El segundo endpoint es el que se usa para recabar la información relativa a la marea actual mediante el enlace *'/tides/measure'*. En él, el programa llama a la clase *SeaHeightScrapper*, explicada anteriormente, y recoge la información de la última marea, para posteriormente construir la respuesta.

```
39      @RequestMapping(path = "/measure", method = RequestMethod.GET)
40      public ResponseEntity getMeasureInfo(@RequestParam String date, @RequestParam String hour) {
41          try {
42              Double tide = SeaHeightScrapper.getInstance().getLastTide();
43              return getResponseEntity(tide);
44          } catch (Exception e) {
45              e.printStackTrace();
46              return CustomTidesResponseEntity.getResponseEntityWhenError();
47          }
48      }
```

Ilustración 65: Endpoint de la actualidad. Clase TidesController.

Finalmente, el último endpoint es el relativo a la recolección de las predicciones a través del enlace *'/tides/prediction'*, en él se formatea la fecha a el formato que maneja la web de PORTUS [13], para luego conseguir en las predicciones la medición de esa fecha, tras lo cual se devuelve la respuesta posteriormente.

```java
@RequestMapping(path = "/prediction", method = RequestMethod.GET)
public ResponseEntity getPredictionInfo(@RequestParam String date, @RequestParam String hour) {
    try {
        String[] dateFields = date.split( regex: "-");
        date = dateFields[2] + "-" + dateFields[1] + "-" + dateFields[0];
        Double tide = new TidesPredictions().getMeasures( date: date+" "+hour);
        return getResponseEntity(tide);
    } catch (Exception e) {
        return CustomTidesResponseEntity.getResponseEntityWhenError();
    }
}
```

Ilustración 66: Endpoint de la predicción. Clase TidesController.

Por otro lado, nos encontramos la clase *RestrictionController*, usada para recabar la información de las restricciones y para realizar cambios en las mismas. Esta clase contiene cuatro endpoints; el primero, como puede observarse en la Ilustración 67, es el encargado de conocer qué restricciones están activas en el momento de la llamada, mediante el enlace *'/restrictions/actives'*. En él, se consigue la lista de las restricciones y se filtra, eliminando de la lista las que no están activas y recogiendo los índices en la lista de las que sí lo estén.

```java
@RequestMapping(path = "/actives", method = RequestMethod.GET)
public ResponseEntity getRestriction() {
    try {

        List<Restriction> restrictionsCapacity = RestrictionsCapacity.getInstance().getRestrictionsCapacity();
        List<Integer> indexes = IntStream.range(0, restrictionsCapacity.size()) IntStream
                .boxed() Stream<Integer>
                .filter(r -> restrictionsCapacity.get(r).isActive()) Stream<Integer>
                .collect(Collectors.toList());

        return CustomRestrictionsResponseEntity.getResponseEntityWithRestrictions(indexes);
    } catch (Exception e) {
        e.printStackTrace();
        return CustomRestrictionsResponseEntity.getResponseEntityWhenError();
    }
}
```

Ilustración 67: Endpoint de las restricciones activas. Clase RestrictionController.

En segundo lugar, nos encontramos el endpoint con enlace *'/restrictions/data', dedicado a recoger los datos de las restricciones existentes en la aplicación.*

```java
@RequestMapping(path = "/data", method = RequestMethod.GET)
public ResponseEntity getRestrictions() {
    try {
        return CustomRestrictionsResponseEntity.getResponseEntityWhenData();
    } catch (Exception e) {
        e.printStackTrace();
        return CustomRestrictionsResponseEntity.getResponseEntityWhenError();
    }
}
```

Ilustración 68: Endpoint de todas las restricciones. Clase RestrictionController.

En tercer lugar, en la Ilustración 69 podemos ver el endpoint con enlace *'/restrictions/update'*, destinado a cambiar el estado de una restricción de activo a desactivo, o viceversa. Para ello recoge todas las restricciones existentes, las pone todas como desactivas, y activa las que en el programa constan como tal. Esta última operación puede verse en la Ilustración 70.

```java
@RequestMapping(path = "/update", method = RequestMethod.PUT)
public ResponseEntity setRestriction(@RequestBody String data) {
    try {
        ObjectMapper mapper = new ObjectMapper();
        JsonNode jsonNode = mapper.readTree(data);
        jsonNode = jsonNode.get("restrictions");

        List<Integer> restrictions = StreamSupport
                .stream(jsonNode.spliterator(), parallel: false)
                .map(JsonNode::asInt).collect(Collectors.toList());

        RestrictionsCapacity.getInstance().switchActive(restrictions);
        return CustomRestrictionsResponseEntity.getResponseEntityWithEmptyBody();
    } catch (Exception e) {
        e.printStackTrace();
        return CustomRestrictionsResponseEntity.getResponseEntityWhenError();
    }
}
```

Ilustración 69: Endpoint de cambio a activo. Clase RestrictionController.

```java
public void switchActive(List<Integer> statesIndex) {
    restrictions.forEach(r->r.setActive(false));
    statesIndex.forEach(index->restrictions.get(index).setActive(true));
}
```

Ilustración 70: Operación de cambio a activo, clase RestrictionsCapacity.

En cuarto y último lugar, nos encontramos el endpoint dedicado al cambio de la densidad en la capacidad de carga física, que puede observarse en la Ilustración 71, y con enlace *'/restrictions/update/CCF'*. Para ello, simplemente se crea un calculador de capacidad, y se le indica que cambie la densidad a la deseada, dado que se crea por defecto a la densidad de personas por metro cuadrado normal, sin ninguna restricción.

```java
@RequestMapping(path = "/update/CCF", method = RequestMethod.PUT)
public ResponseEntity setPhysicalDensity(@RequestParam double CCFValue) {
    try {
        new PhysicalCapacityCalculator().changeDensity( densityValue: 1/CCFValue);

        return CustomRestrictionsResponseEntity.getResponseEntityWithEmptyBody();
    } catch (Exception e) {
        e.printStackTrace();
        return CustomRestrictionsResponseEntity.getResponseEntityWhenError();
    }
}
```

Ilustración 71: Endpoint de cambio de densidad. Clase RestrictionController.

Para finalizar, se ha de explicar que el frontend no ha visto cambios significativos más allá de los mostrados con anterioridad como para volver a nombrarlos en este apartado. Lo que más cambios ha visto han sido las hojas de lenguaje de marcas, que indican cómo se ve la aplicación, que se comentarán a continuación.

En primer lugar, tenemos la clase de Vue DataViewer, que es la página de la aplicación que se muestra y se puede ver en la siguiente ilustración. En ella se coloca el selector de tiempo (TimeSelector) que tiene su propia clase y se le indica el método que usa para actualizar la aplicación cuando se modifica el tiempo en el selector. También se coloca la tabla de los datos, que también tiene su propia clase (Table).

```
<template>
  <div id="app" class="mt-1">
    <b-row>
      <b-col cols="11">
        <TimeSelector @updatedata="updateComponents"/>
      </b-col>
    </b-row>
    <b-row>
      <b-col cols="12">
        <Table/>
      </b-col>
    </b-row>
  </div>
</template>
```

Ilustración 72: Clase DataViewer.

La clase TimeSelector se divide divide en dos partes, en primer lugar, se incluye en la clase del selector el texto del nombre de la aplicación, como puede verse en la ilustración que se puede ver más atrásmás adelante. La segunda parte de la clase es la que se refiere al selector en sí. En primer lugar, se añade el selector de fecha y el de hora, con el método para actualizar el tiempo al introducir uno nuevo. Posteriormente, se introducen los tres botones con sus respectivos métodos para avanzar y retroceder en el tiempo y para volver a la actualidad. Esto se puede ver en la Ilustración 74.

```
<b-row>
    <b-col class="text-left">
        <b-row>
            <b-col>
                <h1>Playa de Las Canteras</h1>
                <p>Capacidad de carga</p>
            </b-col>
        </b-row>
    </b-col>
</b-row>
```

Ilustración 73: Primera parte de la clase TimeSelector

```
<b-col class="px-0 mr-1" md="auto" cols="3">
    <b-form-datepicker class="datePicker" v-model="value_date" :max="max" locale="es" @input="checkTime">
</b-col>

<b-col class="px-0" md="auto" cols="3">
    <b-form-timepicker class="hourPicker" v-model="value_time" locale="es" @input="checkTime"></b-form-ti
</b-col>

<b-col class="px-0 selector" cols="1">
    <b-button size="sm" class="ml-1 buttons" variant="light" @click="goBackTime">
        <b-icon icon="arrow-left-square" aria-label="Help"></b-icon>
    </b-button>
    <b-button size="sm" pill class="ml-1 buttons" variant="light" @click="setActualDateAndTime">
        <b-icon icon="circle" aria-label="Help"></b-icon>
    </b-button>
    <b-button size="sm" class="ml-1 buttons" variant="light" @click="advanceTime">
        <b-icon icon="arrow-right-square" aria-label="Help"></b-icon>
    </b-button>
</b-col>
```

Ilustración 74: Segunda parte de la clase TimeSelector

En la clase de la tabla, más larga que la anterior, podemos encontrar la barra de color, el cual queda determinado por el selector de tiempo, la altura del mar y la superficie. Estos datos los recoge de una lista, creada en el constructor, que viene como respuesta desde el backend. El método de actualización, que puede verse en la siguiente ilustración, recoge los datos que llegan del backend en la variable *data*, y lo guarda en la variable information, que es la lista mencionada anteriormente.

```
updateTable(data){
    if(typeof data == 'undefined' || data[0].tide == "no-content") this.content = false
    else{
        this.information = data;
        this.content = true
        this.getActiveRestrictions();
        this.getRestrictionsData();
    }
},
```

Ilustración 75: Método de actualización de la tabla.

Posteriormente, se crea la tabla de los datos, en la que se introduce el modificador de restricciones comentado anteriormente, que también tiene su propia clase (RestrictionModifier), y coloca los datos recogidos en la variable information. Para ello, recorre los datos de information y los va colocando en los cuadros de la tabla, no sin antes formatearlos para que se puedan interpretar bien. La Ilustración 76 es un ejemplo de cómo se realiza lo anteriormente descrito, dado que todas las líneas son iguales, salvo que tan solo esta tiene el modificador de restricciones, y para las restricciones se añadió un botón en la tabla.

```
<b-tr>
    <b-td class="zone align-middle subheader">Capacidad de carga física (CCF) m<sup>2</sup> por persona</b-td>
    <RestrictionModifier v-bind:value="CCFDensity"/>
    <b-td class="header data align-middle" v-for="(value, idx) in information[2].aforo_fisico" :key="idx">
        {{value | formatNumber}}
    </b-td>
</b-tr>
```

Ilustración 76: Ejemplo de una fila de la tabla.

En cuanto a la clase RestrictionModifier, que puede verse en la Ilustración 77,solo se trata de un formulario con el campo de la densidad, que se le pasa al crearlo en la clase Tabla. En caso de que el usuario lo modifique se llama al método de esta clase para hacer la llamada a la API mencionada con anterioridad.

```
<b-row>
  <b-col class="px-0 mx-0">
    <b-form-input
        style="text-align: right"
        v-model="value"
        type="number"
        min="1"
        :state="state"
        required
        @change="modifyCCFDensity"
    ></b-form-input>
  </b-col>
  <b-col>
      m<sup>2</sup> por persona
  </b-col>
</b-row>
```

Ilustración 77: Visualización de la clase RestrictionModifier.

Herramientas

Durante el desarrollo de la aplicación se han usado las siguientes herramientas:

- Python. Lenguaje de programación que se usó inicialmente para la creación de los scraper. [33]
- Pycharm. IDE de Python para desarrolladores. [34]
- Java versión 1.8. Lenguaje de programación usado finalmente para el backend. [35]
- IntelliJ IDEA. IDE para JVM. [36]
- Spark. Framework para la creación de aplicaciones web en Java que se usó en un primer momento. [24]
- Spring. Framework para el desarrollo de aplicaciones en Java que se usó finalmente en la aplicación. [27]
- Javascript. Lenguaje de programación. [37]
- Vue.js. Framewoek de JavaScript para la construcción de interfaces de usuario de una sola página. [25]
- BootstrapVue. Framework que usa Vue, que permite crear aplicaciones con un diseño web adaptable. [26]
- HTML. Lenguaje de etiquetas de hipertexto y componente más básico de la web. Se usa para la estructura base de la aplicación. [38]
- CSS. Lenguaje de estilos usado para describir la presentación de documentos HTML o XML. [39]
- StarUML. Una herramienta para el modelamiento de software basado en los estándares UML (Unified Modeling Language) y MDA (Model Driven Arquitecture). Usado para la realización de diagramas. [40] [41]
- Git. Una herramienta para el control de versiones.

Ajuste a la planificación

Inicialmente, se estimó que la fase del estudio previo y análisis tendría una duración de diez horas, la fase de diseño, desarrollo e implementación doscientas cuarenta horas, y para finalizar la fase de documentación y presentación cincuenta. Finalmente, aunque las horas no son totalmente ajustadas a la realidad, la estimación está cerca de la misma, aunque ha tomado más tiempo del esperado realizar la fase de documentación.

Adicionalmente, la fase de diseño, desarrollo e implementación se separó en cinco iteraciones diferenciadas. Estas iteraciones fueron las que se pueden ver en la Tabla 4.

Iteración 1 – Obtención de datos. Implementar el scrapper de datos
Iteración 2 – Implementación del modelo predictivo
Iteración 3 – Implementación del backend
Iteración 4 – Implementar la vista
Iteración 5 – Implementar la conexión entre la vista y el backend

Tabla 4: Iteraciones de la fase de Diseño / Desarrollo / Implementación:

Pero, finalmente al seguir un desarrollo iterativo, en el que antes de cada iteración se comprueba que la aplicación tenga el funcionamiento deseado y se proponen los objetivos, las iteraciones cambiaron. Asimismo, como se ha podido ver en el apartado de desarrollo (pág. 11), las iteraciones iniciales 2, 3, 4 y 5 comprendieron finalmente varias iteraciones, así como las iteraciones 3, 4 y 5 se mezclaron y dividieron en varias. Esto se hizo para poder ofrecer un prototipo inicial del cual el usuario percibiera valor.

Modificación de los objetivos planteados

Los objetivos planteados originalmente en el documento de propuesta de trabajo de fin de título se pueden ver en la Ilustración 78 y la Ilustración 79.

2. Objetivos (obligatorio): Objetivos/resultados del trabajo
Realizar un software que pueda estimar la capacidad de carga de la playa en función a múltiples factores que puede haber, teniendo en cuenta el nivel de la marea. Otros factores pueden ser la presencia de socorristas en la playa, la presencia de medusas, etc.
Como los factores pueden ser variables, se pretende crear una arquitectura que permita fácilmente añadir o quitar estos criterios que afecten a la capacidad de carga.
Realizar una aplicación que, tomando los datos de mareas, prediga el aforo máximo de diversas zonas de la playa de Las Canteras, así como el aforo máximo teniendo en cuenta diversas variables (restricciones por la COVID, presencia de medusas, preservación del ecosistema…).

La capacidad de carga se puede calcular con diferentes criterios:

1. La capacidad de carga física (CCF), que hace referencia al número máximo de personas que pueden acceder al espacio.
2. La capacidad de carga real (CCR), es decir, el número de personas que pueden acceder al espacio sin afectar al ambiente.
3. Y finalmente la capacidad de carga efectiva (CCE), esto es el número máximo de personas que puede haber cumpliendo con las medidas impuestas en ese espacio. Como ejemplo nos encontramos con la distancia de seguridad implantada como medida por la COVID.

Cada una de las capacidades se calcula a partir de la anterior.

Esto es algo para tener en cuenta de cara a añadir las variables, dado que no afectan todas a la misma.

Ilustración 78: Objetivos planteados inicialmente 1

Como objetivo también nos encontramos el poder especificarle al modelo el nivel de marea en cada momento en relación con el cero marítimo dado que ésta afecta directamente a la superficie de la playa, y en consiguiente, a la capacidad de carga.

Otro objetivo específico es la creación de un cuadro de mando en el que se puedan añadir tareas desde las que se podrán emitir alertas en caso de que se de una situación excepcional, como puede ser un derrame de petróleo, y que pueda afectar a la playa.

Ilustración 79: Objetivos planteados inicialmente 2

Finalmente, el primer objetivo se mantuvo y se logró durante el desarrollo de la aplicación. La aplicación muestra la capacidad de carga de la playa en los diferentes sectores tomando la marea como referencia para realizar una estimación calculada por la aplicación. También se completó el objetivo relativo a el cálculo del aforo teniendo en cuenta diferentes variables. Estas variables finalmente son las restricciones que se muestran en la aplicación, y que se calculan a partir de la capacidad de carga estimada. También se pueden agregar nuevas restricciones añadiendo una nueva línea para dicha restricción y cumplimentándola, siguiendo el formato descrito anteriormente, en la iteración séptima (pág. 30).

En cuanto al segundo objetivo, este quedó modificado desde un primer momento, dado que el usuario ya no es capaz de especificar el nivel de altura de la marea a la aplicación. Actualmente, y en la aplicación final, el nivel de marea lo recoge de la web de PORTUS, y no es necesario que el usuario lo añada manualmente, facilitándole el trabajo.

Finalmente, el último objetivo no pudo realizarse por falta de tiempo, pero podría ser una ampliación del proyecto actual para el futuro.

Conclusiones y trabajo futuro

Conclusiones

Durante el desarrollo de la aplicación se ha tenido la oportunidad de aprender nuevas tecnologías, como son Vue.js, Bootstrap y Spark, que resultaron muy útiles facilitando el trabajo que de otra forma hubiera sido más complicado y hubiera tomado más horas para completarlo.

Asimismo, he podido profundizar mis conocimientos tanto en Java como en JavaScript, así como en el framework Spring, con el que había trabajado anteriormente. De esta manera, he ganado experiencia y conocimientos en estos lenguajes y en el framework mencionado anteriormente tanto durante el desarrollo como el mantenimiento de la aplicación de cara a los arreglos posibles de errores que pudieran surgir. Esto resultó muy útil dado que, semanas después de haber acabado el desarrollo del proyecto, encontré el error, mencionado durante esta memoria, consistente en que PORTUS no tiene en cuenta el cambio de hora que viene con los solsticios y esto no era considerado por la aplicación. Gracias a la experiencia ganada durante el desarrollo, así como haber realizado un desarrollo iterativo, pude encontrar con facilidad donde realizar la corrección de la aplicación para que pudiera funcionar de nuevo.

Gracias al proyecto, he disfrutado de la oportunidad de trabajar en un entorno laboral, con una aplicación con impacto real en nuestra sociedad, dado que los administradores de la playa pueden usar la aplicación como guía para manejar la playa. Como el programa estima la capacidad de carga máxima, el administrador puede llevar un mejor control sobre cuántas personas visitan la playa, resultando útil no solo por el distanciamiento social necesario que ha traído consigo la COVID-19, sino para preservar la playa de las Canteras, un espacio natural del que goza la ciudad de Las Palmas de Gran Canaria y cuyo mantenimiento no se tiene tanto en cuenta como con otros espacios naturales, así como el impacto que puede traer consigo un número excesivo de personas en la playa.

Trabajo futuro

Finalmente, y como proyectos futuros y dada su versatilidad, el programa se podría llevar y ajustar para otras playas de cualquier parte del mundo en el que se tengan los datos necesarios, que son simplemente los mencionados en el apartado de estado actual (pág. 3). En este aspecto la aplicación es muy útil dado que la funcionalidad del cálculo de la capacidad de carga según los valores referenciales es algo que no se ha hecho y que puede extrapolarse a otras playas con sencillez, como se explicó en el apartado de resultados (pág. 11).

Adicionalmente, este proyecto puede ampliarse, pudiendo aumentar el cuadro de mandos para que envíe avisos a los administradores de la playa, ampliación que no pudo realizarse finalmente. También, si se añade una herramienta que ayude a monitorizar el número de personas que están en las zonas de la playa en cada momento, la aplicación podría reflejar esos datos y el administrador podría basarse en ellos para tomar las medidas oportunas en cada caso.

Referencias

[1] J. P. Morales Aymerich, «La capacidad de carga: conceptos y usos,» *CATIE, Turrialba (Costa Rica)*, p. 7, 2015.

[2] J. G.-L. García y D. G. Ventura, Capacidad de acogida de uso público en los espacios naturales protegidos – nº 3, Organismo Autónomo Parques Nacionales, 2014.

[3] Ayuntamiento de Las Palmas de Gran Canaria, «Proyecto: "Investiga Las Canteras",» [En línea]. Available: https://investigalascanteras.iusiani.ulpgc.es/doku.php?id=program:morning.

[4] F. C. González, «File:Playa de las canteras 24 Dec2006 palmas gran canaria.jpg,» 19 Octubre 2020. [En línea]. Available: https://commons.wikimedia.org/w/index.php?title=File:Playa_de_las_canteras_24_Dec2006_palmas_gran_canaria.jpg&oldid=494395188.

[5] Gobierno de Canarias, B.O.C. Nº 42, 2021, Boletín Oficial de Canarias, 02 de marzo.

[6] Roquenublogc, «File:Roque nublo.jpg,» 18 Octubre 2020. [En línea]. Available: https://commons.wikimedia.org/w/index.php?title=File:Roque_nublo.jpg&oldid=492992215.

[7] Fundación Interuniversitaria Fernando González Bernáldez para los Espacios Naturales, Herramientas para la Evaluación de las Áreas Protegidas: Modelo de memoria de gestión, Madrid: Fundación Interuniversitaria Fernando González Bernáldez para los Espacios Naturales., 2010, p. 84.

[8] Fundación Interuniversitaria Fernando González Bernáldez para los Espacios Naturales. , Estándar de calidad en la gestión para la conservación en espacios naturales protegidos. Guía de Aplicación., Madrid: Fundación Fernando González Bernáldez, 2010.

[9] Fundación Fernando González Bernáldez, Evaluación del papel que cumplen los equipamientos de uso público en los espacios naturales protegidos, Madrid: Fundación Fernando González Bernáldez, 2006.

[10] W. C. contributors, «File:Milchkühe rot-weiß.JPG,» Wikimedia Commons, the free media repository., 3 Diciembre 2020. [En línea]. Available: https://commons.wikimedia.org/w/index.php?title=File:Milchk%C3%BChe_rot-wei%C3%9F.JPG&oldid=516433197.

[11] Organización Mundial del Turismo. OMT, Desarrollo turístico sostenible. Guía para planificadores locales. Edición para América Latina y Caribe., Madrid: OMT , 1999.

[12] M. Cifuentes, Determinación de la capacidad de carga turística de áreas protegidas.Serie técnica. Informe técnico 194., Turrialba, Costa Rica: CATIE, 1992.

[13] Gobierno de España, «PORTUS (Puertos del Estado),» Gobierno de España. Ministerio de Fomento, [En línea]. Available: https://portus.puertos.es/Portus_RT/.

[14] Gobierno de España. Ministerio de Fomento, «Niveles de referencia de nivel del mar,» 14 Julio 2011. [En línea]. Available: https://portus.puertos.es/Portus/pdf/referencias/Descripcion_Referencia_NivelDelMar_en.pdf.

[15] Fundación para la Pesca y el Marisqueo, «Fundamar,» Fundación para la Pesca y el Marisqueo, [En línea]. Available: https://www.fundamar.org/proyectos-publicaciones/redmar/.

[16] c. d. Wikipedia, «Desarrollo iterativo y creciente,» Wikipedia, La enciclopedia libre., 14 Marzo 2021. [En línea]. Available: https://es.wikipedia.org/wiki/Desarrollo_iterativo_y_creciente. [Último acceso: 15 Junio 2021].

[17] W. C. contributors, «File:Iterative Process Diagram.svg,» Wikimedia Commons, the free media repository., 5 Mayo 2021. [En línea]. Available: https://commons.wikimedia.org/w/index.php?title=File:Iterative_Process_Diagram.svg&oldid=558109210. [Último acceso: 15 Junio 2021].

[18] W. contributors, «Iterative and incremental development,» Wikipedia, The Free Encyclopedia., 5 Mayo 2024. [En línea]. Available: https://en.wikipedia.org/w/index.php?title=Iterative_and_incremental_development.

[19] W. contributors, «Test-driven development,» Wikipedia, The Free Encyclopedia., 7 Mayo 2021. [En línea]. Available: https://en.wikipedia.org/w/index.php?title=Test-driven_development&oldid=1021946301.

[20] W. C. contributors, «File:TDD Global Lifecycle.png,» Wikimedia Commons, the free media repository., 3 Octubre 2020. [En línea]. Available: https://commons.wikimedia.org/w/index.php?title=File:TDD_Global_Lifecycle.png&oldid=478854937.

[21] PowerData, «PowerData,» [En línea]. Available: https://www.powerdata.es/data-lake.

[22] W. contributors, «Data lake,» Wikipedia, The Free Encyclopedia., 1 Junio 2021. [En línea]. Available: https://en.wikipedia.org/w/index.php?title=Data_lake&oldid=1026268061.

[23] c. d. Wikipedia, «Bróker de mensajería,» Wikipedia, La enciclopedia libre., 13 Enero 2021. [En línea]. Available: https://es.wikipedia.org/w/index.php?title=Br%C3%B3ker_de_mensajer%C3%ADa&oldid=132364199.

[24] P. Wendel y L. Löfdahl, «Spark Framework,» Spark, [En línea]. Available: https://sparkjava.com/.

[25] E. c. d. Vue, «Vue.js,» Patrocinadores de Vue, [En línea]. Available: https://es.vuejs.org/index.html.

[26] E. c. d. BootstrapVue, «BootstrapVue,» Patrocinadores de BootstrapVue, [En línea]. Available: https://bootstrap-vue.org/.

[27] Spring, «Spring | Home,» Spring, [En línea]. Available: https://spring.io/.

[28] Red Hat, «Red Hat,» Red Hat, [En línea]. Available: https://www.redhat.com/es/topics/api/what-are-application-programming-interfaces#:~:text=Una%20API%20es%20un%20conjunto,de%20saber%20c%C3%B3mo%20est%C3%A1n%20implementados..

[29] C. Á. Caules, «arquitecturajava,» [En línea]. Available: https://www.arquitecturajava.com/usando-el-patron-factory/.

[30] Oracle, «Java Documentation,» Oracle, [En línea]. Available: https://docs.oracle.com/javase/tutorial/java/concepts/interface.html.

[31] Equipo Geek, «ifgeekthen,» Everis, [En línea]. Available: https://ifgeekthen.everis.com/es/clases-abstractas-e-interfaces.

[32] JSON.org, «JSON,» [En línea]. Available: https://www.json.org/json-es.html.

[33] Python Software Foundation, «Python,» Python Software Foundation, [En línea]. Available: https://www.python.org/.

[34] Jetbrains, «PyCharm,» Jetbrains, [En línea]. Available: https://www.jetbrains.com/es-es/pycharm/.

[35] ORACLE, «Java 8,» ORACLE, [En línea]. Available: https://www.java.com/es/download/help/java8_es.html.

[36] Jetbrains, «IntelliJ IDEA,» Jetbrains, [En línea]. Available: https://www.jetbrains.com/es-es/idea/.

[37] Pluralsight, «JavaScript.com,» Pluralsight, [En línea]. Available: https://www.javascript.com/.

[38] MDN Web Docs, «HTML: Lenguaje de etiquetas de hipertexto,» Mozilla, [En línea]. Available: https://developer.mozilla.org/es/docs/Web/HTML.

[39] MDN Web Docs, «CSS | MDN,» Mozilla, [En línea]. Available: https://developer.mozilla.org/es/docs/Web/CSS.

[40] S. Zamenfeld, «StarUML una herramienta para modelado,» BrainLabs New IT, [En línea]. Available: https://www.brainlabs.com.ar/novedad/staruml-una-herramienta-para-modelado/.

[41] MKLabs Co.,Ltd., «StarUML,» MKLabs Co.,Ltd., [En línea]. Available: https://staruml.io/.

[42] c. d. Wikipedia, «Web scraping,» Wikipedia, La enciclopedia libre., 30 Diciembre 2020. [En línea]. Available: https://es.wikipedia.org/w/index.php?title=Web_scraping&oldid=132045206.

Enlaces software

- Enlace al repositorio git del backend
- Enlace al repositorio git del frontend