Prototipo demostrativo de un juguete interactivo enfocado a la adquisición de hábitos de comportamiento.

Universidad Politécnica de Cartagena

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

Trabajo Fin de Grado

AUTOR: Juan José Conesa Hernández

DIRECTOR: Juan Ángel Pastor Franco

Septiembre 2019

Autor	Juan José Conesa Hernández
E-mail del autor	jjch.nc@gmail.com
Director	Juan Ángel Pastor Franco
E-mail del director	juanangel.pastor@upct.es
Codirector(es)	
Titulo del TFG	Prototipo demostrativo de un juguete interactivo enfocado a la adquisición de hábitos de comportamiento.
Descriptores	Flutter, Dart, REST

Resumen:

En este trabajo se ha diseñado e implementado un prototipo de aplicación conectada a la nube, pensada para un dispositivo móvil integrado a un juguete, de forma que esta sirva de asistente personal infantil y con el objetivo de desarrollar hábitos saludables en el uso de las nuevas tecnologías. Además, se ha implementado una aplicación web desde la cual los padres podrán configurar las tareas que aparecerán en el dispositivo móvil. Para esta labor se han empleado tecnologías de IBM Cloud junto con el SDK Flutter de Google para el desarrollo de aplicación móviles.

Titulación	Grado en Ingeniería Telemática
Intensificación	
Departamento	Tecnologías de la Información y las Comunicaciones
Fecha	Septiembre 2019

INDICE

INDICE DE TABLAS

INDICE DE FIGURAS

CAPÍTULO 1: INTRODUCCIÓN

1.1 ESTRUCTURA DE LA MEMORIA

Capitulo 1 *Introducción*: Se presenta la idea o planteamiento general de la realización de este proyecto, así como una breve explicación de los objetivos a alcanzar.

Capitulo 2 *Arquitectura general*: Se describe la estructura general del proyecto y se comentan algunos de los requisitos que se consideran necesarios para el desarrollo del mismo.

Capitulo 3 *Funcionamiento de la aplicación*: Se presentan de manera gráfica el flujo de trabajo de las aplicaciones desarrolladas con objetivo de clarificar la comprensión de la implementación.

Capitulo 4 *Tecnologías empleadas*: Se describen en profundidad el conjunto de tecnologías empleadas en el proyecto.

Capitulo 5 *Implementación de la aplicación*: Se realiza un análisis profundo sobre la estructura interna de las aplicaciones asociadas a este proyecto y su funcionamiento.

Capitulo 6 *Conclusiones y líneas futuras*: Se valoran de forma crítica las tecnologías empleadas durante el desarrollo y se aportan algunas ideas sobre posibles futuros trabajos.

1.2 PLANTEAMIENTO GENERAL

En la actualidad, la tecnología alcanza a cada generación a una edad más temprana. Todo padre puede llegar a plantearse qué método es el más adecuado a la hora de enseñar a sus hijos nuevos conocimientos sobre el mundo tecnológico de tal manera que esto pueda fomentar un desarrollo positivo, y a su vez se evite abuso de prácticas que puedan generar malos hábitos. Esto ha generado la necesidad de realizar estudios sobre cómo puede afectar al desarrollo cognitivo de los más jóvenes

Basándonos en esta idea inicial, este proyecto pretende presentar un posible prototipo de una aplicación para un dispositivo móvil que pueda ser integrado en un juguete como herramienta para que los padres puedan ayudar al desarrollo de sus hijos dentro de un entorno controlado por ellos.

Esta aplicación consistirá en un organizador de tareas diarias que un niño/a pueda visualizar y que generará avisos sonoros a una hora establecida por los padres en la descripción de la tarea.

Esto a su vez implica que los padres deben disponer de algún medio con el cual puedan introducir tareas en el dispositivo incluso si, llegado el momento, no tienen acceso físico al mismo, por lo que es necesaria la implementación de algún tipo de plataforma desde la cual los padres puedan lleva a cabo este proceso.

1.3 OBJETIVOS

Desarrollo de un prototipo de aplicación para un dispositivo móvil, conectado a la nube, capaz de ser integrado en un juguete infantil que sirva como asistente personal con el fin de fomentar un desarrollo positivo el sentido de responsabilidad y organización temporal del usuario. En la implementación de este prototipo se hará uso de distintos tipos de tecnologías que permitan un desarrollo fluido a la hora de incluir futuras ampliaciones sobre el mismo.

1.4 ANTECEDENTES

Antes de comenzar con la implementación de este proyecto se considero que era necesario llevar a cabo una investigación previa sobre posibles productos que pudiesen existir mercado considerados como juguetes inteligentes o *SmartToys*, **centrándonos en aquellos orientados en fomentar algún tipo de desarrollo positivo en los niños** como por ejemplo el sentimiento de responsabilidad, la adquisición de de pautas y horarios o la mejora en sus capacidades para relacionarse con otros niños.

Durante este proceso de investigación la referencia más interesante que se encontró fue la del proyecto CYBERCLOUD4TOYS del AIJU *(Instituto Tecnológico especializado en juguete, producto infantil y ocio) subvencionado por el Instituto Valenciano de Competitividad Empresarial*, iniciado el año 2018 en el que, con la colaboraron de distintas empresas jugueteras, se desarrollaron una serie de aplicaciones demostrables asociadas a distintos juguetes. Según el informe del proyecto CYBERCLOUD4TOYS su objetivo es proporcionar un espacio seguro (usando técnicas de autentificación de usuario y protocolos de cifrado de datos) donde se puedan obtener datos sobre pautas de comportamiento y conductas infantiles que permitan realizar estudios en un futuro con objetivo de mejorar las técnicas de enseñanza para niños.

Algunos de los programas creados en el proyecto CYBERCLOUD4TOYS son los siguientes:

- Una aplicación para tablet asociada al juguete Baby Susú de la empresa BERJUAN S.L para el cuidado de una muñeca que mostrará las necesidades típicas de un bebe, que irán apareciendo en la pantalla del dispositivo a lo largo del tiempo. Esta aplicación fomenta un sentido de responsabilidad y refuerza en concepto de organización temporal.

- Una aplicación para smartphone o tablet asociada al producto Batería Golden Drums de la empresa CLAUDIO REIG S.L. en la que el niño haciendo uso de la batería compite contra un contrincante virtual, teniendo que recrear patrones musicales que irán apareciendo en el dispositivo. El motivo de incluir esta referencia se debe a que se considera que esta aplicación puede mejorar las capacidades motoras del niño/a y su sentido de la competitividad.

- Aplicación de realidad aumentada para Smartphone o tablet asociada al producto Combis Sticker-32 piezas de la empresa Game Movil S.L. Una vez que este juguete tipo puzle este montado, cuando

sea enfocado con la cámara del dispositivo generará una animación a modo de cuento sencillo mediante la app, fomentando el desarrollo de la imaginación del niño.

Por otro lado, ya fuera del marco del proyecto antes mencionado, se han encontrado otros ejemplos de otros productos, fruto de distintos kickstarter, de notable interés como por ejemplo:

- *Dino* de la empresa CogniToys, un muñeco con forma de dinosaurio, conectado a internet, orientado al aprendizaje con el que los niños podían mantener conversaciones medio fluidas pudiendo hacer preguntas que el juguete contestaría, crear cuentos e incluso capaz de contar bromas infantiles.

- Aunque no se trate de un juguete nos ha parecido interesante introducir el reloj organizador *Octupus* de la empresa *Joy* creado con objetivo de enseñar buenos hábitos y el concepto de organización temporal donde los padres pueden a través de una aplicación móvil introducir tareas que generaran alertas en el reloj a las horas indicadas.

CAPÍTULO 2: ARQUITECTURA GENERAL

Aunque los detalles sobre la implementación de las aplicaciones que componen este proyecto son descritos con mayor profundidad en capítulos posteriores se ha considerado conveniente plantear aquí una descripción introductoria de la estructura general con la que se ha trabajado para fundamentar el empleo de determinadas tecnologías.

En la Figura 1 podemos ver un diagrama simple sobre el cual se comenzó a plantear este proyecto.

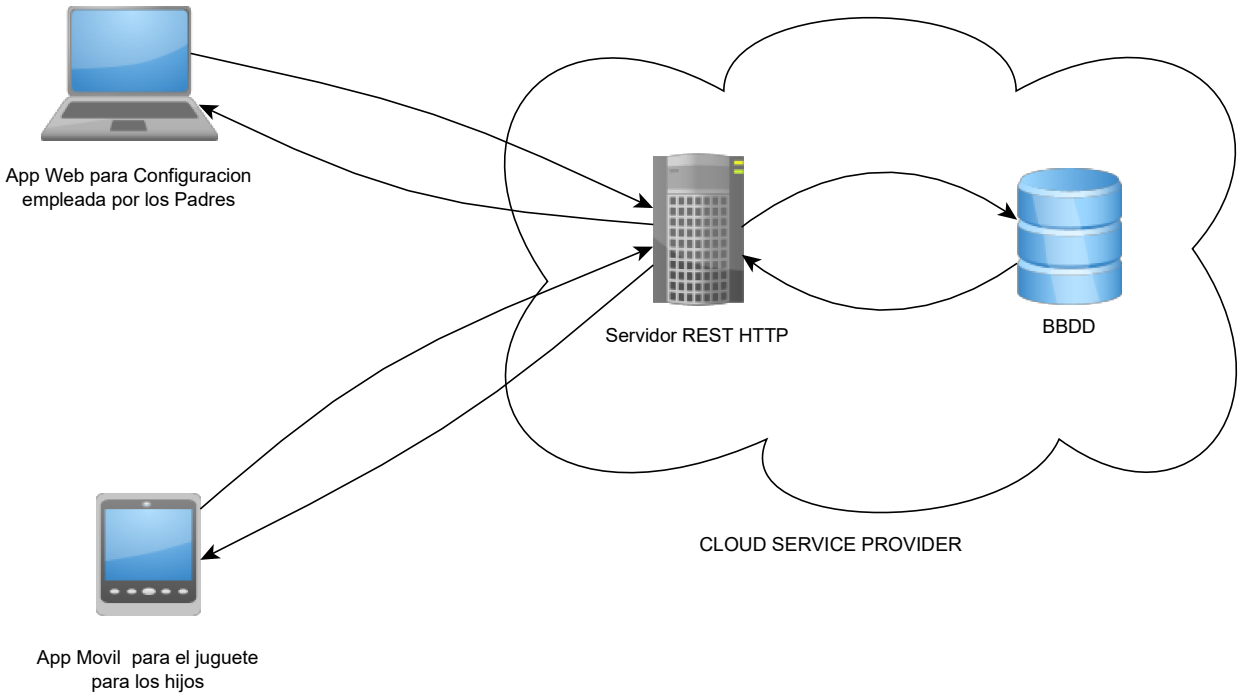

Figura 1: Arquitectura general del proyecto

Como ya se comento en el capitulo introductorio, nuestro objetivo principal es el desarrollo de prototipo de una aplicación para un dispositivo móvil (el cual podría ser integrado en un juguete) que sirva como organizador personal infantil, mostrando un listado de tareas a realizar a determinadas horas y que emita notificaciones sonoras en caso de llegar la hora de llevar a cabo dichas tareas.

Una de la primeras necesidades que se nos platea es cómo introducir los parámetros de configuración de dichas tareas y dónde almacenar esta información para poder recuperarla en caso en el que el dispositivo se apague; además, al tratarse de un prototipo, también es importante tener presente que nuestra implementación debe estar abierta a futuras mejoras o a la inclusión de nuevos servicios.
Bajo esta premisa la solución que se propone es la siguiente:

(1) Haciendo uso de un proveedor de servicios en la nube se desplegara un servidor HTTP con arquitectura REST junto con una base de datos sencilla donde se almacenará el listado de tareas. Una necesidad importante que el proveedor en la nube que escojamos debe de satisfacer es la inclusión de algún medio/producto que aporte un cierto nivel de seguridad en la comunicación entre el servidor y la base de datos.

(2) Desarrollo de nuestra aplicación móvil que obtendrá o actualizará la lista de tareas del usuario mediante la realización de peticiones HTTP al servidor. Otro requisito que debemos satisfacer es la implementación de un método con el que podamos identificar usuarios y el conjunto de tareas asociado a cada uno de ellos.

(3) Desarrollo de una aplicación web sencilla, que será suministrada por nuestro servidor HTTP en la nube, desde la que los padres podrán ver el listado de tareas que han asignado a sus hijos, añadir configurar nuevas tareas o eliminar tareas del listado. Si bien esta aplicación web debería incluir algún método para identificar usuarios, se ha valorado que esta función queda fuera del alcance del proyecto, de manera que su construcción se hará pensando que el usuario ya ha iniciado sesión.

Una vez explicado el esquema general de nuestra aplicación nos encontramos en situación de poder *introducir* las tecnologías que han sido empleadas para la realización de cada una de las partes comentadas, junto con una breve justificación sobre qué nos ha llevado a usar cada una de ellas.

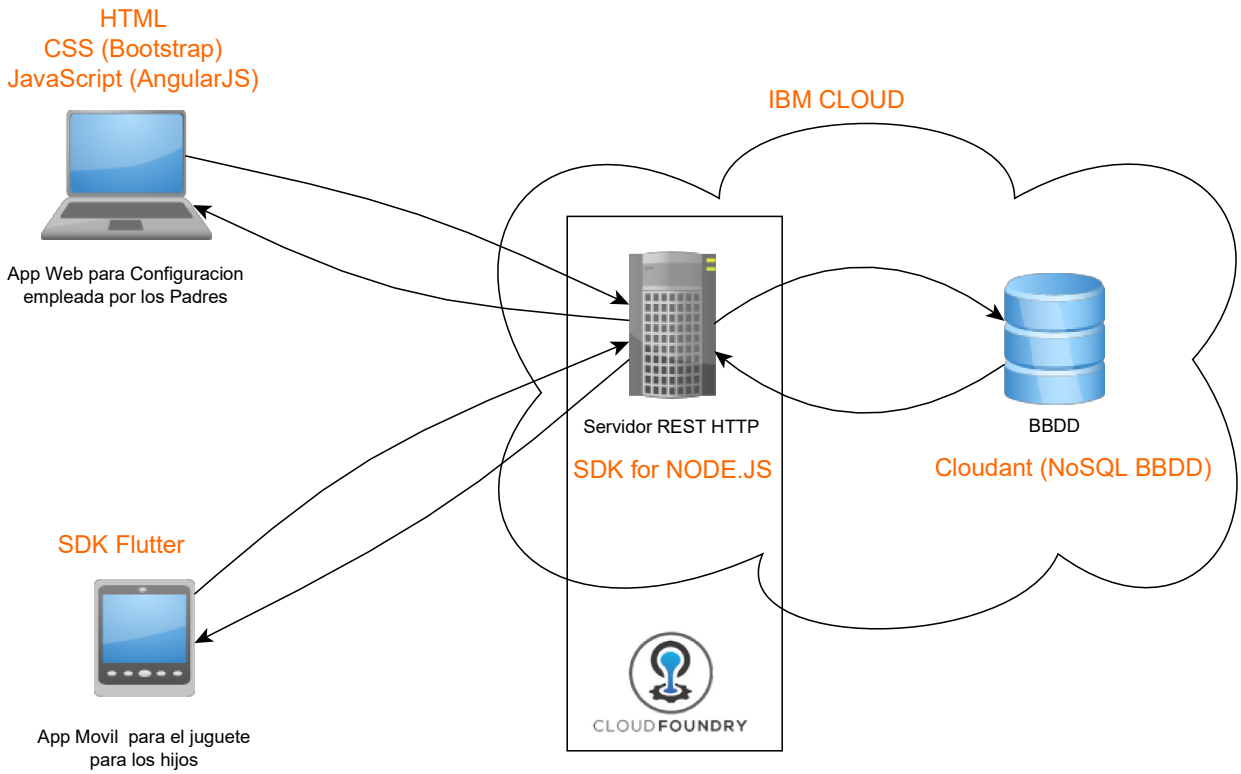

Figura 2: Esquema de tecnologías usadas

Tras un cierto proceso de investigacion sobre las soluciones y productos ofertados por los distintos proveedores en la nube se decidio hacer utilizar de **IBM CLOUD**, en concreto de los productos que aparecen en la Figura 3.

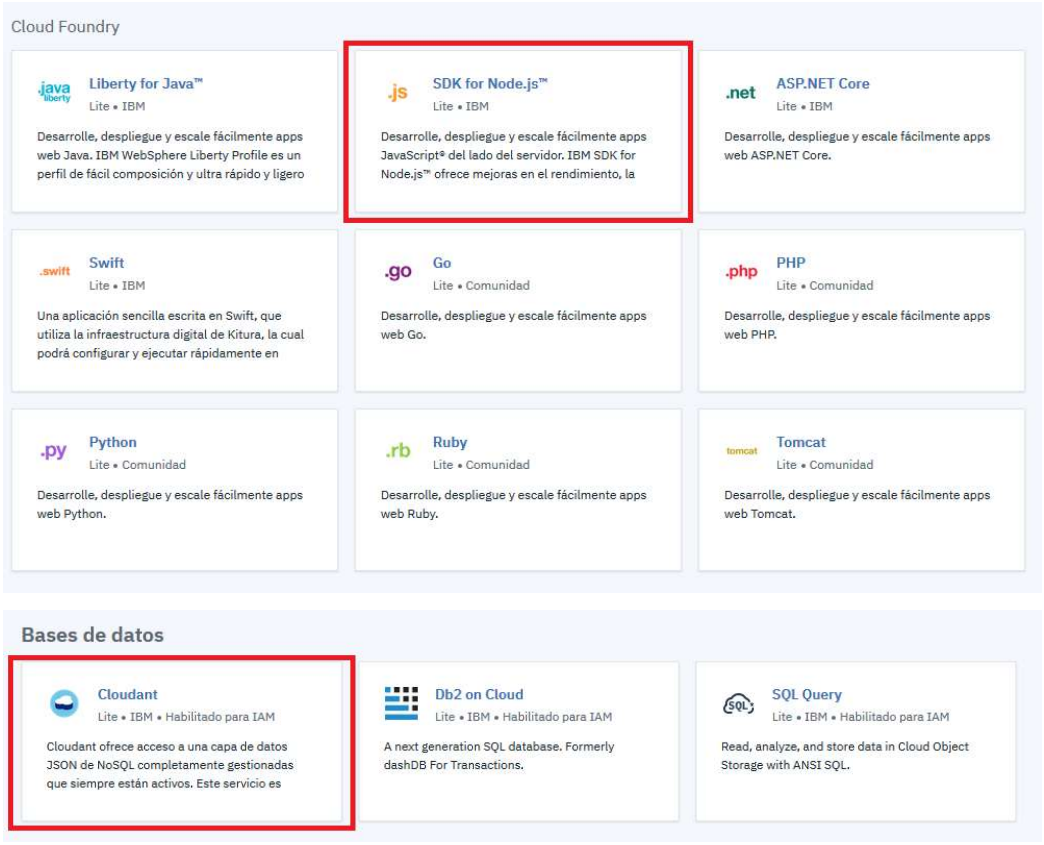

Figura 3: Productos de IBM Cloud empleados

Si bien otros proveedores también suministraban servicios con características similares, el entorno de trabajo y la documentación suministrados por **IBM** permitieron un despliegue casi inmediato de una aplicación web sencilla empleando una instancia del producto *SDK for Node.js; a*demás este entorno soporta entrega continua permitiendo una programación ágil y mucho más fluida.

Para el desarrollo del servidor se decidió usar *Node.js* debido a que dispone de un framework llamado *Express* que permitía por un lado suministrar archivos estáticos (HTML, CSS, JavaScript,..) para poder crear el portal de configuración que será empleado por los padres y por otro generar un servidor REST básico para la atención de peticiones HTTP desde distintos clientes, cumpliendo así dos de los puntos planteados en la estructura inicial.

Se optó por usar una base de datos *NoSQL* empleando el producto *Cloudant* debido a que durante nuestro desarrollo el esquema de los datos a almacenar era susceptible a cambios en función de las necesidades que surgiesen en los clientes.

Por último para nuestra aplicación móvil decidió emplear un framework de código abierto que es relativamente joven creado por Google llamado *Flutter* que además de permitir el desarrollo de aplicaciones que funcionaran indistintamente tanto Android como iOS, presenta una amplia librería de clases y funciones para la generación de interfaces de usuario y una curva de aprendizaje relativamente eficiente.

Todas las tecnologías que han sido comentadas y algunos conceptos relacionados con las mismas serán descritas con mayor profundidad en un capitulo posterior.

CAPÍTULO 3: FUNCIONAMIENTO DE LA APLICACIÓN

Con el objetivo de facilitar la comprensión de los detalles de implementación de las aplicaciones desarrolladas se considera necesario aportar una explicación grafica sobre el funcionamiento de estas en conjunto. En primer lugar, en la Figura 4 podemos ver el aspecto de la aplicación web desde la que los padres podrán configurar las tareas que aparecerán en el dispositivo móvil de los hijos.

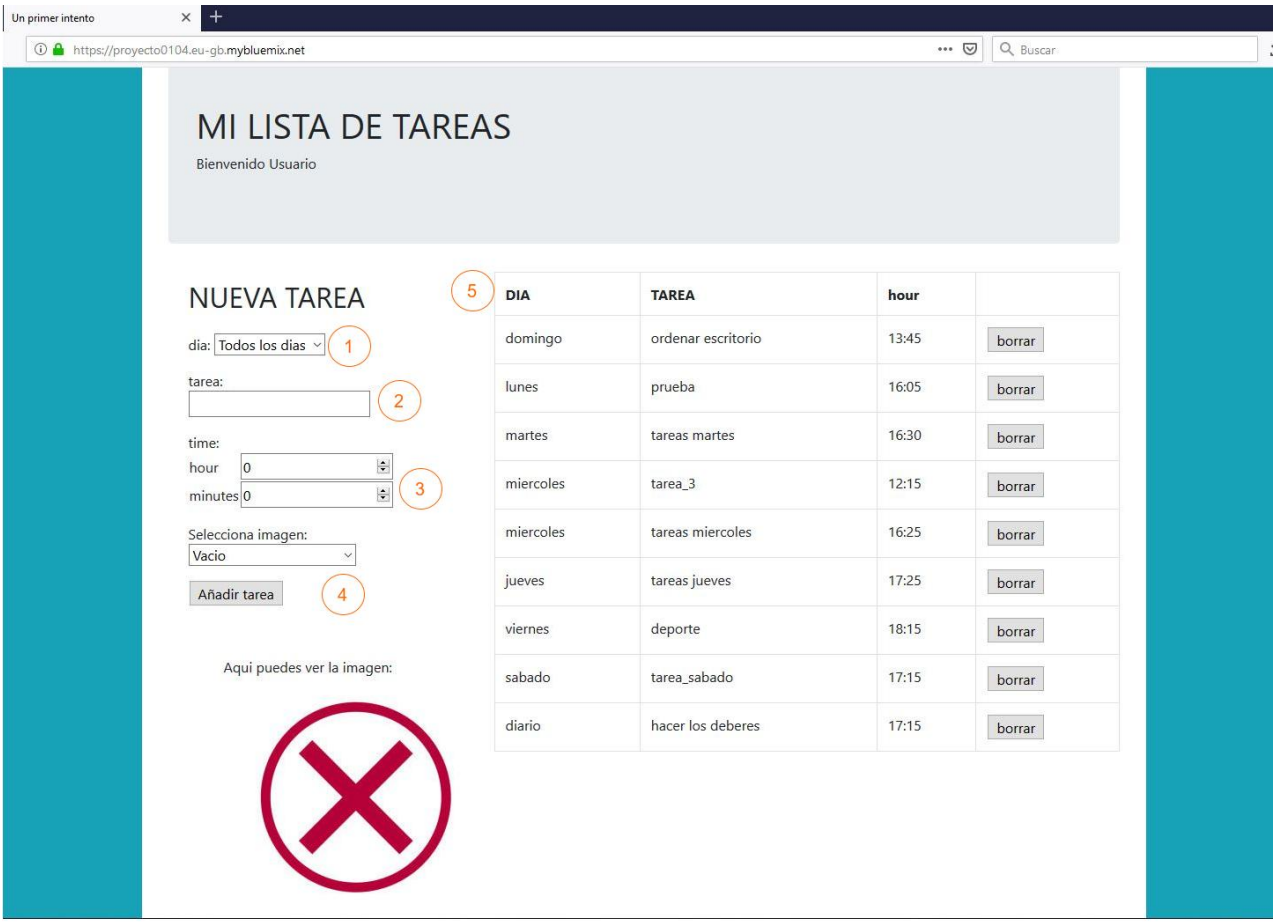

Figura 4: Prototipo de aplicación web para configuración de tareas

Como ya se comentó previamente, partimos del supuesto de que la aplicación web dispone de apartados para registro/identificación de usuarios que no han sido implementados debido a que quedan fuera de las competencias del proyecto. Desde estos apartados los padres deben generar un usuario y contraseña que quedarán almacenados en la base de datos alojada en la nube, y haciendo uso de estos datos podrán acceder a la sección mostrada en la Figura 4. En nuestro caso haremos uso de un usuario de pruebas creado directamente en la base de datos.

Una vez identificados podrán seleccionar ciertos parámetros de cada tarea: (1) el día al que asignar la tarea, (2) una descripción sencilla, (3) hora a realizar la tarea y (4) una imagen asociada a la tarea;

además, es posible ver todo el conjunto de tareas en una tabla (5) y se pueden eliminar las tareas que sean innecesarias.

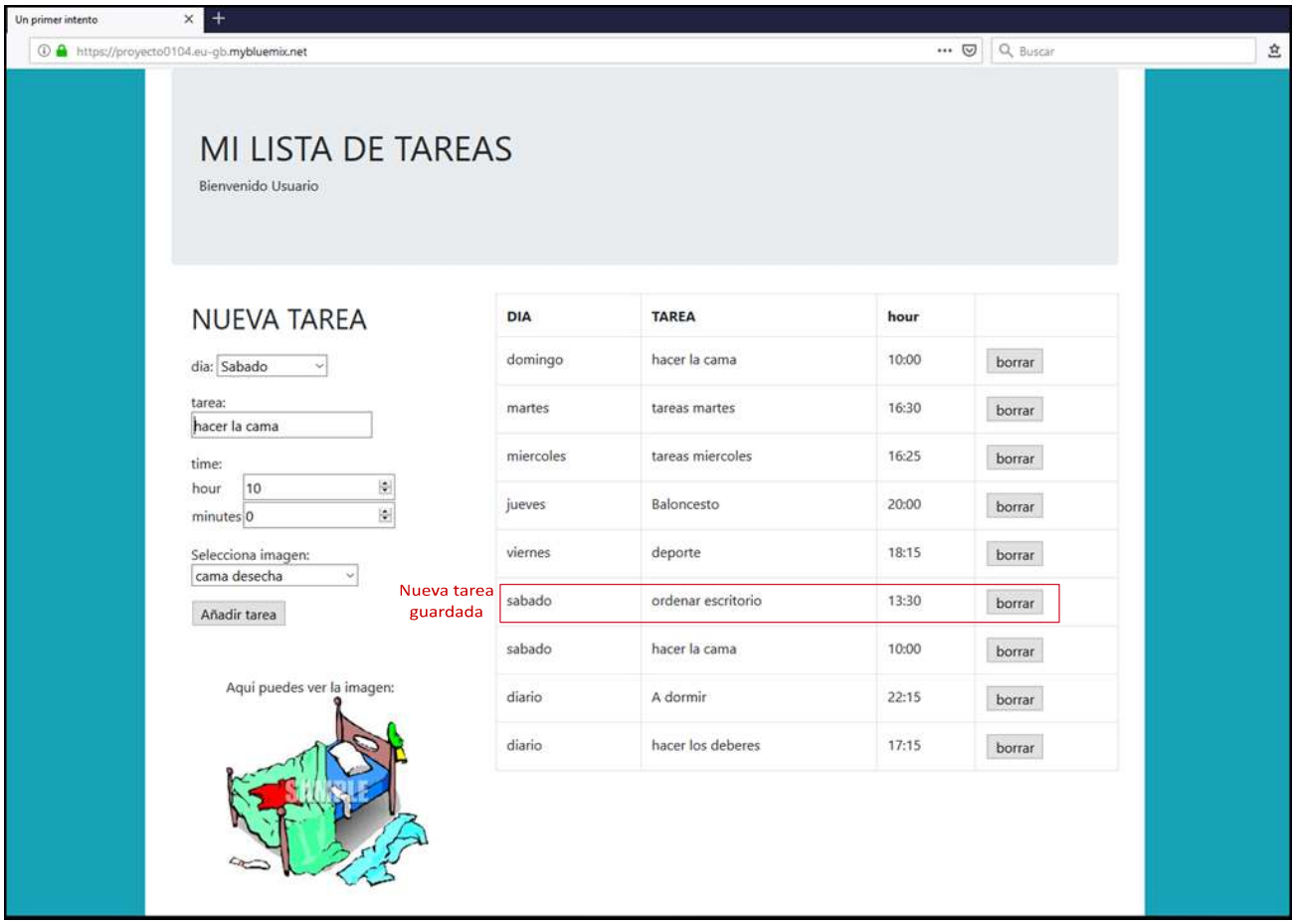

Figura 5: Nueva tarea introducida

Por otro lado, en Figura 6 podemos ver el menú inicial de nuestro prototipo de aplicación móvil. A iniciar esta aplicación la primera pantalla que nos aparece será un menú de identificación de usuario. Cuando el usuario de la aplicación realice este proceso en caso de introducir algún dato erróneo el mensaje de bienvenida cambiará alertándole de que existe algún error como vemos en la Figura 7 y si, por el contrario, los datos son correctos aparecerá un botón, que vemos en la Figura 8, que permitirá acceder al listado de tareas asignadas para este usuario en el día actual.

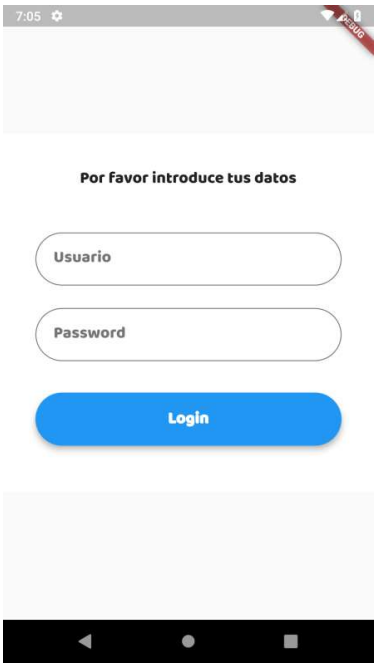

Figura 6: Pantalla de inicio de la aplicación móvil

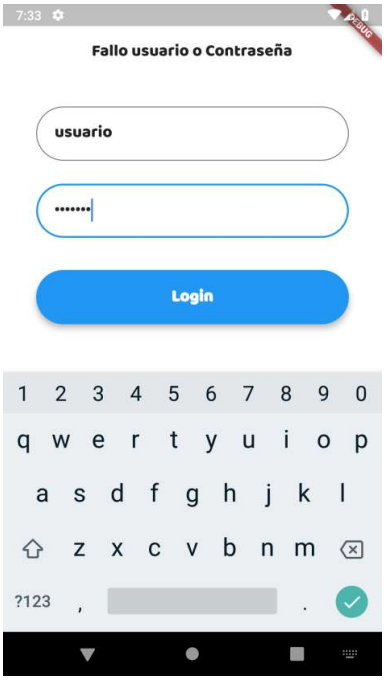

Figura 7: Mensaje de advertencia ante identificación incorrectos

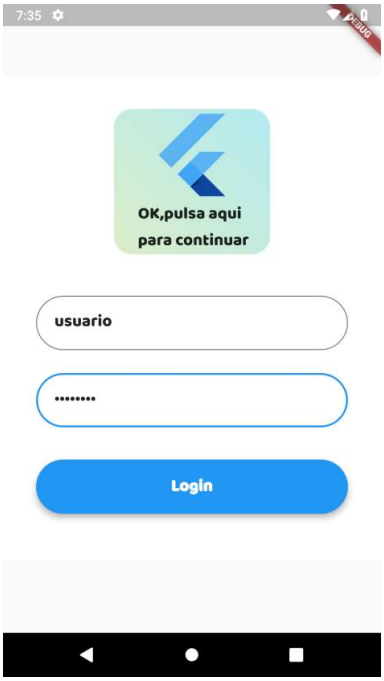

Figura 8: Botón para continuar tras identificación correcta

En el listado de tareas se ha incluido la opción de resaltar aquellas que ha completado haciendo doble clic sobre ellas, o desmarcarlas mediante una pulsación prolongada, con el pensamiento de que los niños interactúen con la aplicación. En las Figura 9 podemos ver el aspecto estas dos situaciones.

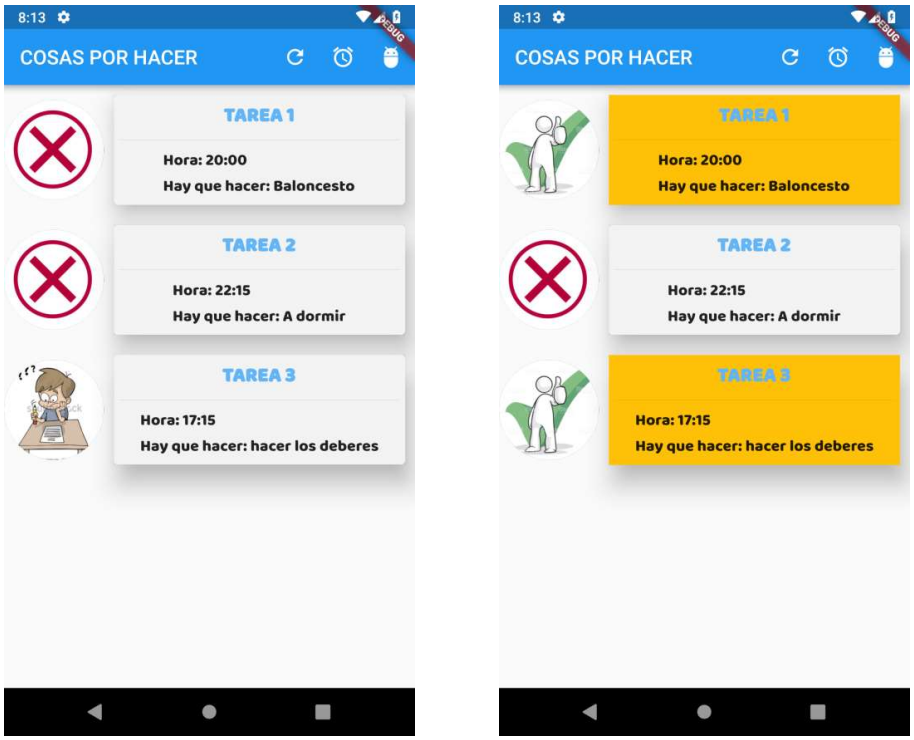

Figura 9: Opción de marcado de tareas

A continuación comentaremos otras de las funciones que se ha incluido en la aplicación. En primer lugar, la lista de tareas que se muestra en pantalla está programada para actualizarse de manera automática tras un cierto intervalo de tiempo con el objetivo de disponer del conjunto de datos más actualizado. También existe la posibilidad de actualizarla a voluntad pulsando el icono de la barra superior. Es posible consultar a qué hora fue la última actualización automática de la lista pulsando el botón lo que generara el mensaje podemos ver en la Figura 10.

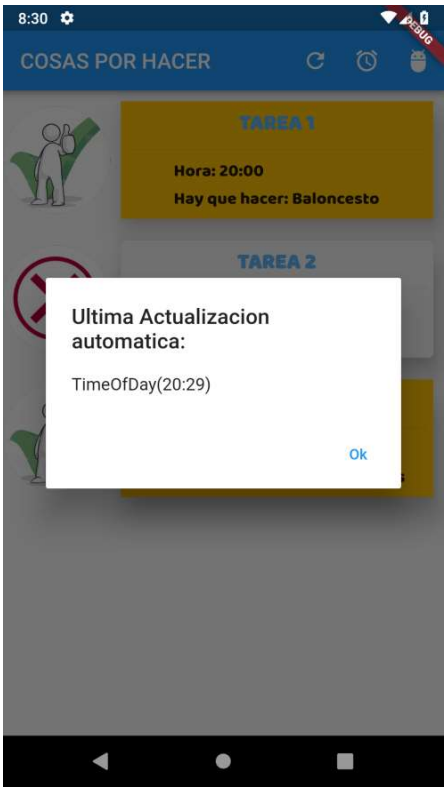

Figura 10: Alerta última actualización automática

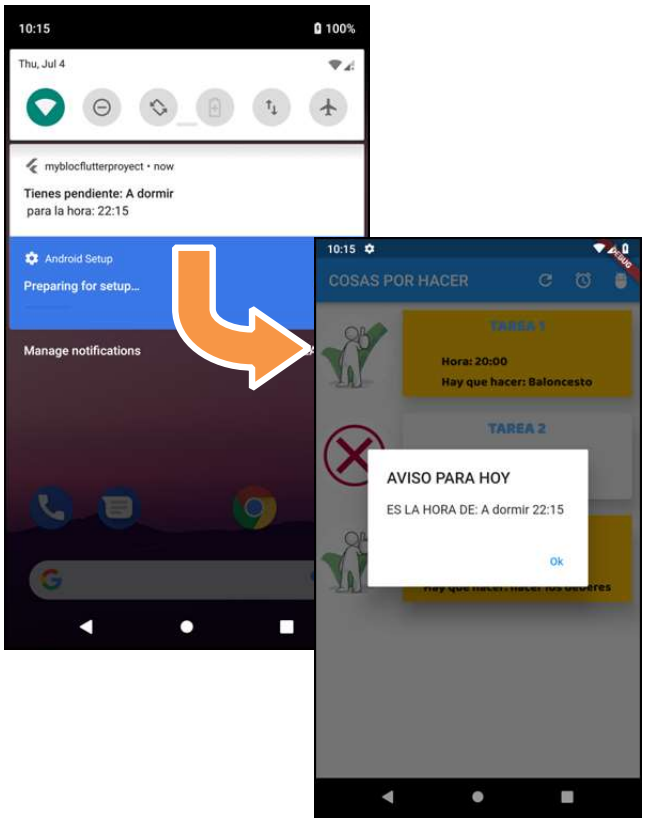

Figura 11: Comportamiento tras pulsar sobre un aviso

Otra función incluida en la aplicación es la generación de avisos sonoros y visuales (solo en dispositivos Android) para las tareas que aparecen en la lista, aunque solo para aquellas cuya hora aun no haya llegado. Cuando un usuario pulse sobre la notificación de una tarea, esta nos llevará a la aplicación para poder marcar la tarea como completada, la Figura 11 muestra este proceso.

Por último, se puede hacer uso del botón  para ver cuántas notificaciones quedan pendientes mediante un mensaje de alerta en pantalla como podemos ver en la Figura 12.

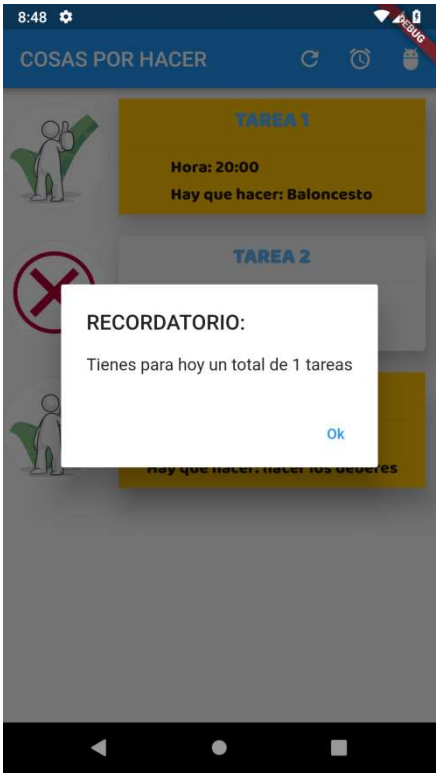

Figura 12: Alerta de avisos pendientes

CAPÍTULO 4: TECNOLOGÍAS EMPLEADAS

A continuación aportaremos una definición más detallada de las tecnologías empleadas en el proyecto así como algunos conceptos de importancia que están relacionados con las mismas. Comenzaremos con los datos referentes a los servicios que hemos empleado en la nube.

CLOUD FOUNDRY

Entorno con el que trabajan las aplicaciones en la nube de IBM. Se trata de una plataforma como servicio (PaaS) multi-nube de código abierto que fue en un principio desarrollada por VMware y que en la actualidad pertenece a la organización Cloud Foundry Foundation.

Una PaaS es un entorno de desarrollo e implementación completo en la nube, con recursos que permiten entregar todo, desde aplicaciones sencillas hasta aplicaciones empresariales sofisticadas habilitadas para la nube. PaaS incluye infraestructura (servidores, almacenamiento y redes), pero también incluye middleware, herramientas de desarrollo, servicios de inteligencia empresarial (BI), sistemas de administración de bases de datos, etc.

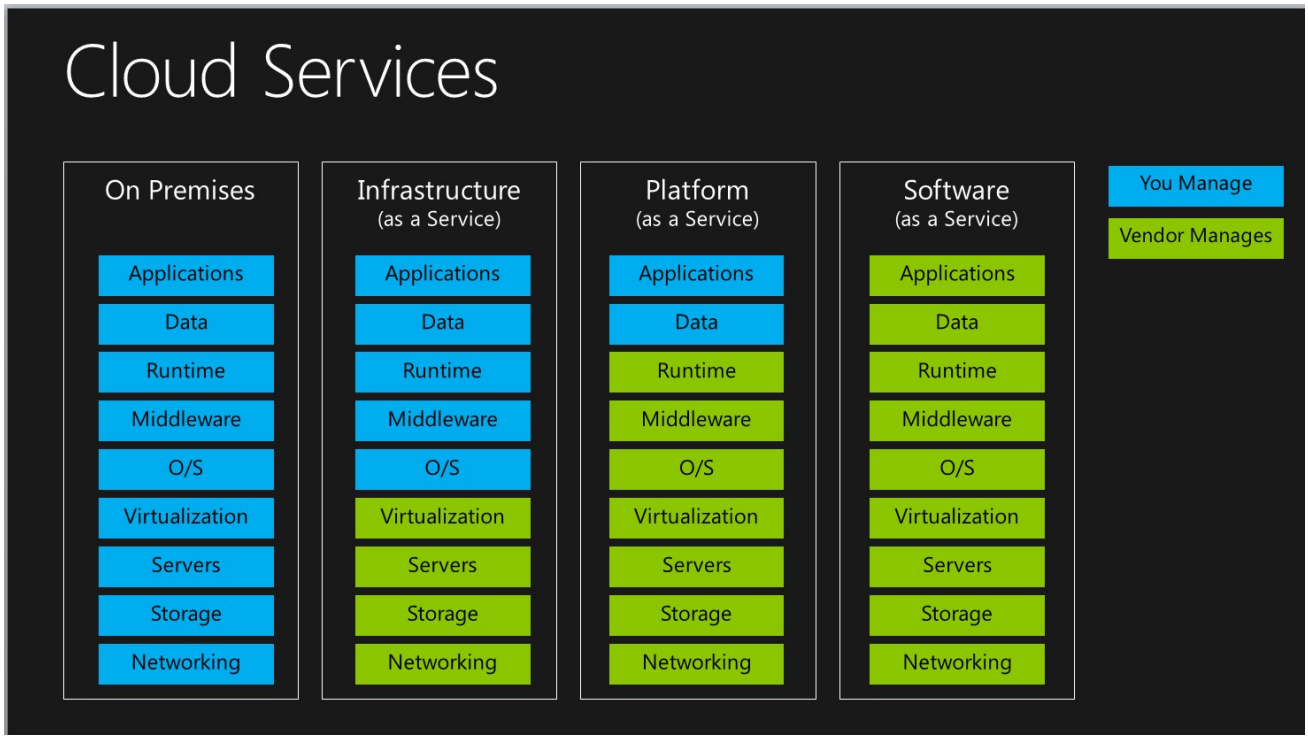

Figura 13: Esquema de servicios PaaS, SaaS, e IaaS

Este entorno permite entrega continua[1] ya que soporta el todo el ciclo de vida de desarrollo de las aplicaciones desde las fases iníciales de desarrollo pasando por las fases de testeo hasta el despliegue

SDK FOR NODE.JS

Este producto de IBM nos permite trabajar en *Node.js* que es entorno de programación JavaScript de código abierto y multiplataforma para el desarrollo, despliegue y escalado de aplicaciones del lado del servidor.

Se ha hecho uso de *Node.js* junto con una de una librería llamada *Express* para crear un **servidor HTTP** que proporciona un API REST (**RE**presentational **S**tate **T**ransfer) y una página web creada usando HTML, framework Bootstrap y AngularJS.

REST es un estilo de arquitectura software que define un conjunto de limitaciones que pueden ser usadas para crear servicios Web, estos servicios proporcionan interoperabilidad entres distintos sistemas en Internet. REST se definía originalmente según una serie de principios fundamentales, aunque en la actualidad se hace referencia a él en un sentido más amplio para describir cualquier interfaz entre sistemas que utilice HTTP para obtener datos o realizar operaciones sobre los datos en cualquier formato (por ej. JSON).

HTML lenguaje de marcas para la elaboración de páginas web, este es usado junto el framework front-end **Bootstrap** que contiene plantillas de diseño con tipografía, formularios, botones, cuadros, menús de navegación y otros elementos de diseño basado en HTML y CSS, así como extensiones de JavaScript adicionales.

AngularJS framework JavaScript de código abierto que se utiliza para crear y mantener aplicaciones web de una sola ventana cuyo objetivo es aumentar las aplicaciones basadas en el navegador con capacidad de *Modelo-Vista-Controlador*.

El patrón **Modelo-Vista-Controlador** aplica el principio de separación de preocupaciones que consiste en dividir una aplicación en distintas partes, cada una tratando una preocupación diferente. El *modelo* el encarga de los datos, el *controlador* se encarga de recibir las órdenes del usuario y de

[1] Enfoque de la ingeniería del software en que los equipos de desarrollo producen software en ciclos cortos, asegurando que el software puede ser liberado en cualquier momento, de forma confiable. Su objetivo es la construcción, prueba y su liberación de software de manera más rápida y más frecuente.

solicitar los datos al modelo y de comunicárselos a la vista y, por último, la *vista* es la representación visual de los datos.

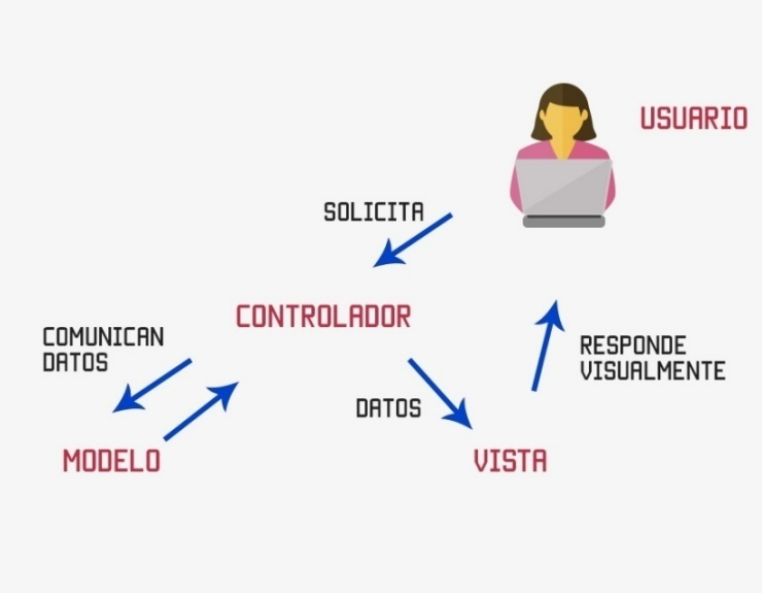

Figura 14: Esquema Modelo-Vista-Controlador

CLOUDANT

Este producto de IBM es una base de datos JSON, basada en Apache CouchDB, que se suministra como un servicio completamente gestionado y altamente disponible. Aprovecha los esquemas de almacenamiento de "Documentos" JSON auto-descriptivos que facilitan un desarrollo de aplicaciones ágil y flexible. Para este proyecto en concreto la emplearemos para almacenar la información de las tareas así como la información de usuario.

El almacenamiento de datos de documentos se representa en notación de objetos JSON y se caracteriza porque cada documento es una estructura de datos compleja. Los documentos pueden contener estructuras subdivididas de varios tipos de datos o incluso otros objetos. Los usuarios de una base de datos de documentos pueden consultar estas estructuras complejas, recuperar o actualizar secciones del documento (o el documento entero) sin necesidad de bloquear la base de datos. Los documentos se almacenan y se recuperan mediante una clave primaria que es exclusiva de cada documento (De forma similar a una clave en un almacenamiento de pares clave-valor).

JSON (JavaScript Object Notation) es un formato ligero de intercambio de datos. Leerlo y escribirlo es simple para humanos, mientras que para las máquinas es simple interpretarlo y generarlo. Está basado en un subconjunto del Lenguaje de Programación JavaScript Standard ECMA-262 3ª Edición. JSON es un formato de texto que es completamente independiente del lenguaje pero utiliza

convenciones que son ampliamente conocidas por los programadores de la familia de lenguajes C, incluyendo C, C++, C#, Java, JavaScript, Perl, Python, y muchos otros. Estas propiedades hacen que JSON sea un lenguaje ideal para el intercambio de datos.

A continuación podemos centrarnos en comentar qué tecnologías se ha decidido emplear para la realización de nuestra aplicación para el dispositivo móvil.

SDK FLUTTER

SDK de código abierto creado por Google que para el desarrollo de aplicaciones móviles tanto para Android como para iOS. Además de ser multiplataforma, Flutter es compatible con otros lenguajes de programación ya que integra código Java sobre Android, y ObjetiveC y Swift en iOS. Por otro lado, permite un desarrollo rápido de aplicaciones debido a la posibilidad de realizar modificaciones sobre las mismas que pueden ser comprobadas al instante gracias a su capacidad de realizar Hot Reload.

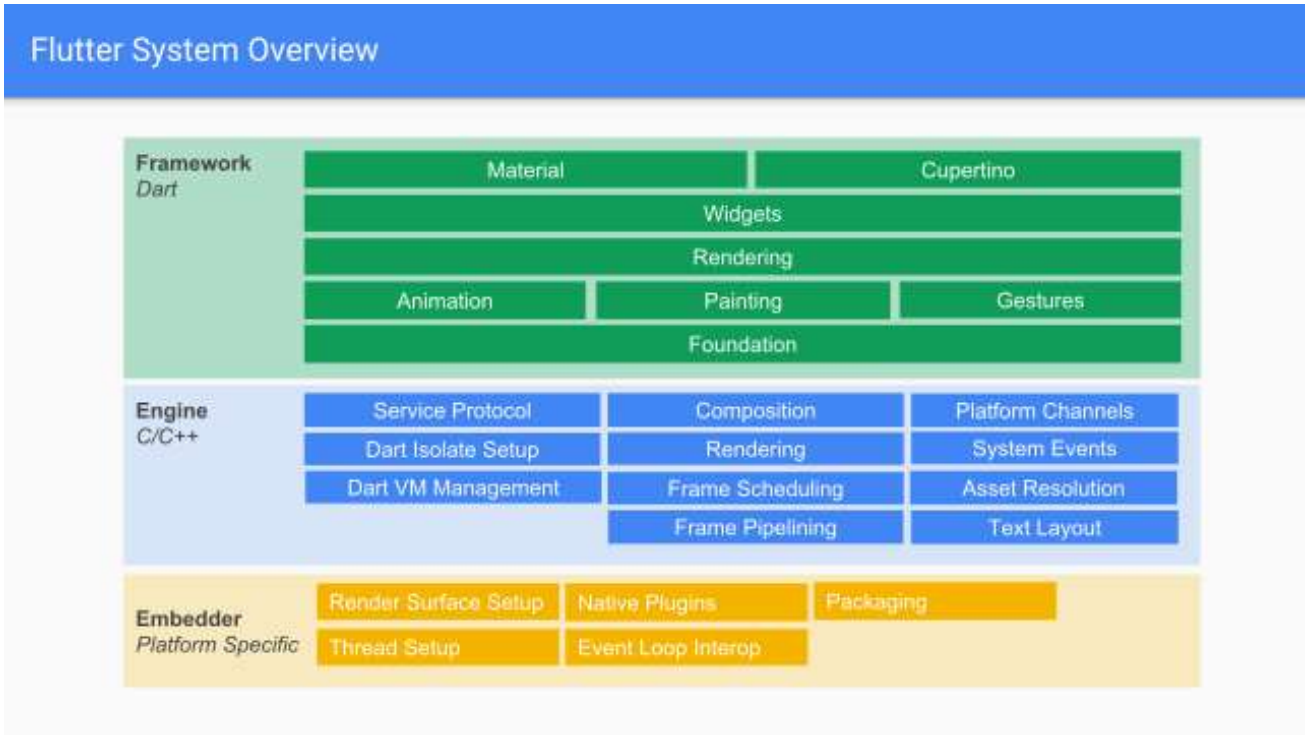

Figura 15: Visión general del sistema *Flutter*

Flutter permite el diseño de interfaces de usuario expresivas y flexibles mediante el uso de widgets, todos los componentes gráficos desde textos hasta formas o animaciones son creados haciendo uso de estos.

Además de la gran cantidad de herramientas que proporciona el SDK también hay disponible un repositorio donde distintos desarrolladores colaboran para crear paquetes que permiten añadir más funcionalidades o simplificar determinados procesos en las aplicaciones.

DART

Las aplicaciones Flutter están escritas en el lenguaje de programación **Dart** y utilizan muchas de las funciones más avanzadas dentro del mismo.
Dart un lenguaje de programación optimizado para el cliente para aplicaciones rápidas en múltiples plataformas, está desarrollado por Google y se utiliza para crear aplicaciones móviles, de escritorio, back-end y web. Dart es un lenguaje de código abierto, estructurado y flexible, orientado a objetos, basado en clases, con herencia simple y soporte de interfaces, clases abstractas y tipado opcional de datos.

CAPÍTULO 5: IMPLEMENTACIÓN DE LA APLICACIÓN

Una vez especificado el funcionamiento y definidas las tecnologías que forman parte de este desarrollo nos encontramos en condiciones de describir los detalles de implementación de cada una de las secciones que lo componen. En primer lugar, se hablará del servidor HTTP REST y de la estructura de los documentos almacenados en la base de datos, tras lo cual se explicará la implementación de la aplicación móvil y por último se describirá la implementación de aplicación web.

5.1 Estructura del servidor y la base de datos

En primer lugar, haciendo uso de la documentación suministrada por IBM se despliegan las instancias de los productos *SDK for Node.js* y *Cloudant*. Después, para poder usar la base de datos como un servicio de nuestro servidor, se conectan ambos productos y se genera un archivo de credenciales de servicio que se emplea para la identificación y el control de acceso que son necesarias para que nuestro servidor pueda realizar consultas a la base de datos (las credenciales de servicio son muy valiosas. Si alguna persona o aplicación tiene acceso a las credenciales, puede hacer lo que quiera con la instancia de servicio).

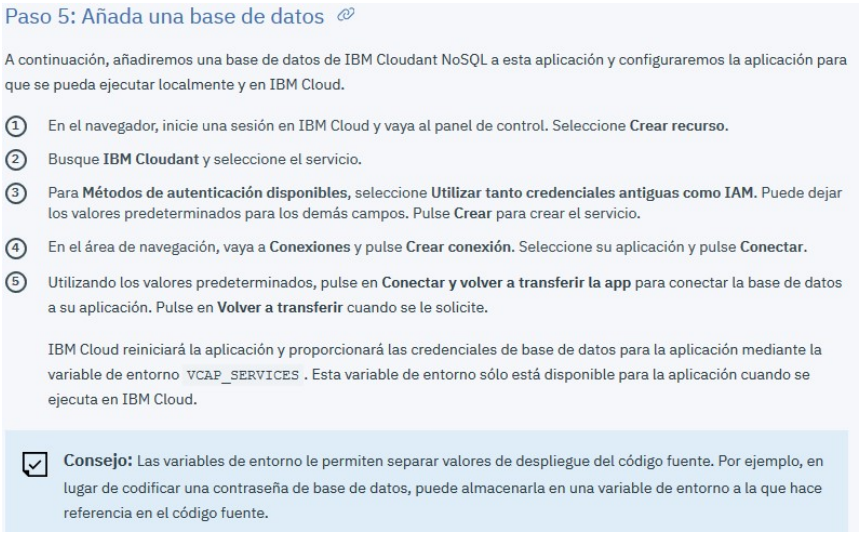

Figura 16: Adición de base de datos con credenciales de autenticación

Una vez llevado a cabo este proceso nos es posible desplegar el servidor HTTP en nuestro espacio en la nube usando las herramientas y una aplicación de ejemplo suministradas por IBM. El script inicial aportado por IBM hace uso del framework *Express* para *Node.js* para generar un servidor sencillo

que se ha usado como esqueleto inicial sobre el que trabajar. La Figura 17 contiene un fragmento del de dicho script inicial.

```
var Express = require("Express");

var app = Express();

app.use(Express.static(__dirname + '/views'));/*PARA SERVIR CLIENTE WEB
(index.html,css,js)*/

var port = process.env.PORT || 3000  /*PUERTO DE ESCUCHA*/

app.listen(port, function () {

  console.log("listening in port" + port);

});
```

Figura 17: Servidor básico usando el framework Express

Express permite también suministrar los archivos estáticos que se encuentren dentro del directorio '/views' haciendo uso de la función de middleware *express.static.* Además es posible configurar el puerto de escucha en función del entorno en el que nos encontremos haciendo uso de '*process.env.PORT*'.

A continuación se enumeran las librerías de *Node.js* que se ha incluido en el servidor, junto con *Express*, y la labor que desempeña cada una de ellas:

- *body*-parser: middleware empleado para el servidor sea capaz de interpretar el contenido de los mensajes de las peticiones HTTP recibidas.
- *@cloudant/cloudant*: librería que permite la conexión con la base de datos.
- *cfenv:* Paquete que aporta funciones para obtener variables de entorno de una app de Cloud Foundry. Se emplea en el script creado para que el servidor pueda obtener las credenciales del servicio *Cloudant* y así generar consultas hacia la base de datos.
- *Bcryptjs*: Paquete empleado para la cifrado de las contraseñas de usuario y su posterior comprobación cuando el usuario de la aplicación móvil se identifique.

El siguiente paso era definir los distintos *endpoints* que recibirían peticiones HTTP con la estructura que vemos en la Figura 18, donde cada uno de ellos tendrá asociado una función *callback* que atenderá las peticiones que se reciban siempre que estas sean del tipo adecuado. La siguiente Tabla 1 contiene la descripción de cada uno de los *endpoint* implementados, qué datos ha de contener la petición recibida por ellos y qué información incluirá la respuesta del servidor ante peticiones correctas.

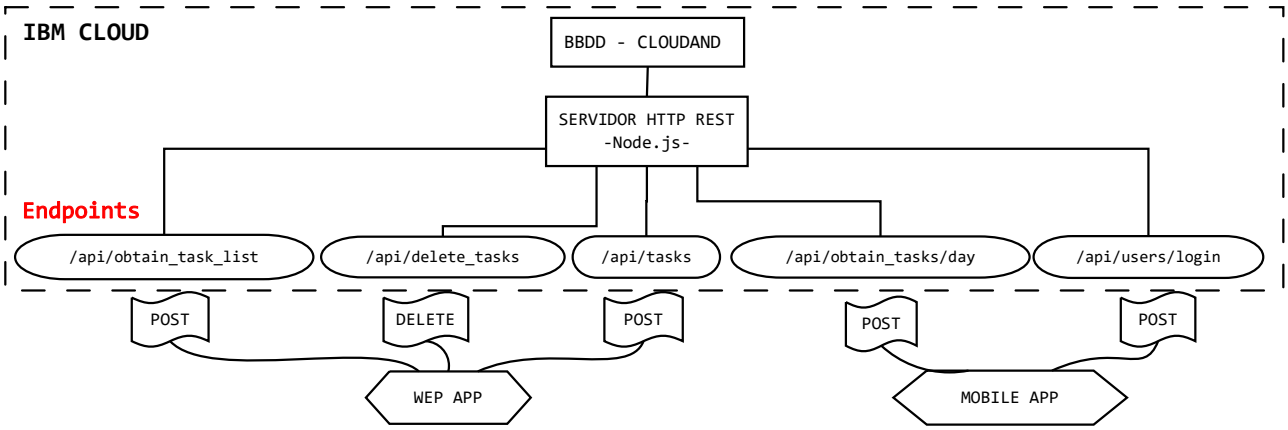

Figura 18: Estructura general del servidor

ENDPOINTS	ENTRADA	DESCRIPCIÓN	SALIDA
/api/obtain_task_list	`{"uuid":int}`	Devuelve un **array** con todas las tareas asociadas a un ID de usuario.	`{"_id":String, "_rev":String, "type":"task", "day":int, "name":String, "hour":String, "done":bool, "uuid":int, "photo":String}`
/api/delete_tasks	`{"_id":String, "_rev":String}`	Elimina una tarea y devuelve un mensaje de confirmación	`"SUCESSFUL - IN DELETE TASK"`
/api/tasks	`{"type":"task", "day":int, "name":String, "hour":String, "done":false, "uuid":int, "photo":String}`	Introduce una nueva tarea en la base de datos. y devuelve un mensaje de confirmación	`"SUCESSFUL - IN POST TASK"`
/api/ obtain_task/day	`{"uuid":int}`	Devuelve **array** con las tareas del día en que nos encontramos para un ID de usuario.	`{"_id":String, "_rev":String, "type":"task", "day":int, "name":String, "hour":String, "done":bool, "uuid":int, "photo":String}`
/api/users/login	`{"name":String, "passw":String}`	Devuelve el ID de asociado al usuario y contraseña enviados en la petición.	`int`

Tabla 1: Descripción endpoints del servidor

Es importante destacar que las respuestas generadas por el servidor contendrán un código de estado HTTP, además de los datos descritos en la tabla anterior, con el pensamiento de permitir a los clientes variar su comportamiento en función de este y así adaptar su funcionamiento según el contenido de la respuesta.

Con idea de ilustrar cómo se ha diseñado la gestion de peticiones recibidas por los *endpoints*, se incluye la Figura 19 en la que vemos un ejemplo de uno de ellos, en concreto el que gestiona la inclusion de nuevas tareas en nuestra base de datos.

```
//**POST - PARA INTRODUCIR NUEVA  TAREA*/

app.post("/api/tasks", function (request, response) {
  var data = request.body;

  if (!mydb) {
    console.log("No database.");
    response.status(500).send('No database');
    return;
  }
  if (data) {
    mydb.insert(data, function (err, body, header) {
      if (err) {
        console.log('[mydb.insert] ', err.message);
        response.status(500).send(err.message);
        return;
      }
      else {
        response.status(200).send("SUCESSFUL - IN POST TASK");
      }
    });
  } else {
    response.status(400).send("Bad Request - IN POST TASK");
  }
});
```

Figura 19: Ejemplo de *endpoint* implementado

Cuando una petición HTTP llega al servidor, la función asociada el *endpoint* llevará a cabo una serie de comprobaciones y si todo es correcto se enviará una respuesta que incluirá un codigo 200 junto con los datos solicitados o un mensaje de confirmación. Por el contrario, si existen problemas, se enviara un codigo de error y un texto de alerta en el contenido de las respuesta.

Comentar que la variable `mydb` del ejemplo contiene la informacion necesaria para la conexion con la base de datos y ha sido generada usando el ejemplo suministrado por la documentacion de IBM que se comentó al inicio de esta sección.

Una vez explicada la estructura general del servidor podemos pasar a describir cómo se han estructurado los documentos de las base de datos *Cloudant*:

- Por un lado tendremos los documentos de tipo *profile* que almacenarán los datos de un usuario padre. Cada uno de estos documentos tendrá asignado un identificador único que debería ser asignado por el servidor (p.ej. durante el registro de usuario) y qué servirá para identificar que tareas pertenecen a cada usuario. Comentar que la contraseña almacenada en este tipo de documentos ha sido cifrada haciendo uso de la librería *Bcryptjs* previamente mencionada.


```
{
"_id": "df1470a6a2ad375d0306506304e782dd",
"_rev": "2-450cb5b9c0908c63651e0a2e684c2b27",
"type": "profile",
"name": "usuario",
"passw": "$2a$10$IMxbfR7xtWLu4IRCR8VOGeiqn73m6cAaLKr67/aBXC5ZoOrVD0cDa",
"uuid": 1122334455
}
```


Figura 20: Estructura documentos tipo "profile" de la base de datos

- Por otro lado, estarán los documentos de tipo *task* que almacenarán los datos de configuración de la tarea elegidos por los padres en la aplicación web junto con el identificador de ese usuario (p.ej. obtenido durante el hipotético proceso de identificación de usuario en la aplicación web de nuestro supuesto). Esta información será usada para la generación del listado de tareas en la aplicación móvil.


```
{
  "_id": "a4675cba9be46fdce92828949cc8842c",
  "_rev": "1-8b9ae1859c443c38cb235acc2a985c76",
  "type": "task",
  "day": 5,
  "name": "Cosas para el viernes",
  "hour": "12:35",
  "done": false,
  "uuid": 1122334455,
  "photo": "assets/images/messy_desktop.png"
 }
```


Figura 21: Estructura documentos tipo "task" de la base de datos

5.2 Detalles técnicos e implementación de la aplicación móvil

El *SDK Flutter* empleado en la aplicación móvil posee una amplia cantidad de clases, funciones y librerías creadas en colaboración otros desarrolladores. Esto puede generar cierta confusión durante la explicación de la implementación a la hora de entender qué elementos forman parte de nuestro desarrollo y cuales son propios del SDK; en consecuencia, con objetivo de facilitar la comprensión de la implementación en las siguientes secciones se ha decidido incluir una serie de notas aclaratorias sobre algunos elementos de importancia del SDK y también se incluirán ejemplos básicos de aplicaciones para que sirvan como referencia.

En primer lugar, se introducirán las *clases/funciones,* junto con algunos conceptos de importancia, de *Flutter* que han sido utilizados durante el desarrollo de la aplicación y que se mencionarán durante la explicación de la implementación.

5.2.1 Conceptos previos sobre la aplicación implementada con Flutter

WIDGETS

La idea principal sobre la que *Flutter* trabaja son los Widgets, siendo estos la **clase central** dentro de la jerarquía de clases del SDK. Un widget es una descripción inmutable de parte de una interfaz de usuario. Al programar la UI de una aplicación definimos un *árbol de widgets* encapsulados unos dentro de otros y cuando ejecutamos la aplicación esta recorre ese árbol buscando la función *build* de cada widget. A esto es lo que los programadores de Google refieren como generar el *árbol de elementos*. Usaremos como referencia el ejemplo de la Figura 22 para explicar este proceso:

```dart
import 'package:flutter/material.dart';

void main() {
 runApp(new DogApp());
}

class DogApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'My Dog App',
     home: Scaffold(
       appBar: AppBar(
         title: Text('Yellow Lab'),
       ),
       body: Center(
         child: Column(
           mainAxisAlignment: MainAxisAlignment.center,
           children: [
         DogName('Rocky'),
             SizedBox(height: 8.0),
         DogName('Spot'),
             SizedBox(height: 8.0),
         DogName('Fido'),
           ],
         ),
       ),
     ),
   );
 }
}
```

```dart
class DogName extends StatelessWidget {
 final String name;

const DogName(this.name);

 @override
 Widget build(BuildContext context) {
   return DecoratedBox(
     decoration: BoxDecoration(color: Colors.lightBlueAccent),
     child: Padding(
       padding: const EdgeInsets.all(8.0),
       child: Text(name),
     ),
   );
 }
}
```

Figura 22: Diseño de un widget y empleo del mismo en una aplicación

1º Definimos una clase sencilla *DogName* (que por herencia es un *Widget*).

2º Su constructor necesita recibir un String *name*.

3º Su método *build* nos devuelve un Widget *DecoratedBox* que en su interior tiene un Widget *Text*.

4º Generamos el *main* desde donde especificamos la aplicación que vamos a ejecutar

5º Generamos el *widget* principal de nuestra aplicación.

6º El método *build* devolverá un widget de tipo *MaterialApp* que envolverá en su interior distintos widgets, uno de ellos es el que previamente fue creado.

Los widgets *DecoratedBox*, *Scaffold*, *AppBar*, *Text*, *Center* y *Column* son unos pocos de los muchos que podemos encontrar en el catalogo de *Flutter*. En la Figura 24 podemos ver el resultado final y en la Figura 23 podemos ver el árbol de elementos que se generará tras el inicio de la aplicación.
Asociadas dos clases de gran importancia a la hora de generar aplicaciones en *Flutter* que heredan de la clase *Widget*, hablamos de las clases *StatelessWidget* y *StatefulWidget*.

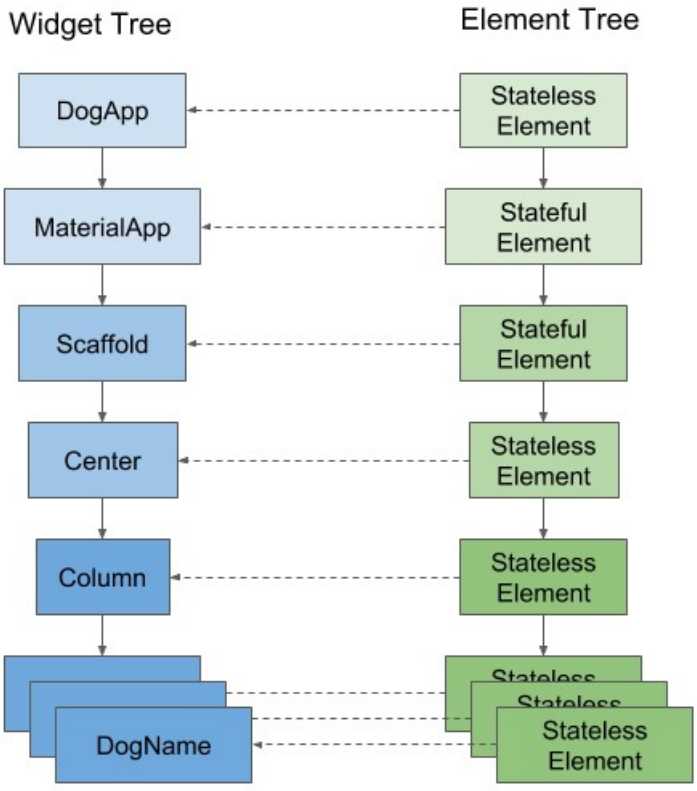

Figura 23: Árbol de widget generado en el ejemplo

Figura 24: Aplicación generada por el ejemplo

Como se vio en el ejemplo anterior, los *StatelessWidget* son empleados para generar elementos de la interfaz que no presentan un tipo de estado o caracteristica cambiante, es decir, aquellos que que no dependen de información de entrada generada por un usuario, siendo solo dependientes de la

informacion suministrada durante su creación. Elementos como botones de retorno o botones de cierre son buenos ejemplos de *StatelessWidget*.

Por otro lado tenemos los *StatefulWidget* que se emplean cuando es necesario actualizar la interfaz en funcion de cambios que se puedan producir con el paso del tiempo, esto es considerado por el framework como un "estado cambiante o mutable". La instancias de este tipo de widgets son inmutables pero el estado de los mismo se almacena en un objeto *State* creado por el metodo `createState`, que es llamado cada vez que el widget sea insertado en el árbol. Algunos ejemplos de este tipo de widgets serian campos para introducir texto, elementos que muestran animaciones, selectores de datos (p.ej. selector de años) o elementos asociados a un temporizador. La Figura 25 presenta la estructura básica de un *StatefulWidget,* donde podemos ver los métodos de mayor importancia a la hora de gestionar el estado.

```
              // CREACION DE UN STATEFULWIDGET
class SimpleCounter extends StatefulWidget {
  final String title;
  SimpleCounter({Key key, this.title}) : super(key: key);

  @override
  SimpleCounterState createState() => SimpleCounterState();
}

              // CREACION CLASE STATE QUE GESTIONA EL ESTADO DEL WIDGET
class SimpleCounterState extends State<SimpleCounter> {
              // método para definir el estado inicial del widget
  @override
  void initState() {
    super.initState();
              // Otras instrucciones para ejecutar al inicio
  }
// método llamado cuando el widget se elimina del arbol de widgets de forma permanente
  @override
  void dispose() {
              /* Aquí se suelen eliminar/cerrar elementos/procesos que ocupen memoria*/
    super.dispose();
  }
// Ejemplo de método que afecta al estado de un widget. Sera llamado en el método build
  void _doSomeChange {
    setState(() {
              // operaciones que afectan al estado de un widget
              // p.ej. actualizar una variable contador que aparece en pantalla
    });
  }
              //Método llamado durante la construcción del elemento asociado al widget
  @override
  Widget build(BuildContext context) {
    return widget(

       child: // … mas widgets

    );
}
```

Figura 25: Ejemplo de *StatefulWidget*

ELEMENTOS DE PROGRAMACIÓN ASÍNCRONA

Debido a la naturaleza de la aplicación programada, teniendo en cuenta que esta realiza peticiones HTTP que pueden tardar un cierto tiempo en ser contestadas, es necesario llevar a cabo operaciones asíncronas. Para ello el lenguaje con el que se encuentra programado *Flutter* aporta varias herramientas de entre las cuales se han empleado:

- Objetos *Future <T>:* representa el resultado de tipo *T* de una operación asíncrona.
- Palabras clave *async*: para expresar que el método/función creada es de tipo asíncrono.
- Palabra clave *await*: partícula que solo funciona en funciones asíncronas con objetivo suspender la ejecución hasta obtener los resultados de una operación futura.

Podemos ver como se usan estos elementos en el código de la Figura 26

```
import 'dart:async';

main() {
  printDailyNewsDigest();
  printWinningLotteryNumbers();
  printWeatherForecast();

}

Future<void> printDailyNewsDigest() async {          // función 2
  var newsDigest = await gatherNewsReports();
  print(newsDigest);
}
printWinningLotteryNumbers() {                       // función 3
  print('2 : Winning lotto numbers: [23, 63, 87, 26, 2]');
}
printWeatherForecast() {                             // función 4
  print("3 : Tomorrow's forecast: 70F, sunny.");
}

const info = '<First functon delayed of 5 seconds>';
const delay = Duration(seconds: 5);

// Imaginemos una funcion larga que tarda 5 segundos en ejecutarse
Future<String> gatherNewsReports() => Future.delayed(delay, () => info);
```

Figura 26: Ejemplo de función asíncrona

En el ejemplo la función `main()` llama a la *función 2* que usa la partícula *await* para llamar a `gatherNewsReports()` que devolverá un `Future<String>` incompleto en principio. Como la *función 2* es asíncrona y esta a la espera de recibir un `Future<String>` la ejecución del `main()` continua llamando a las *funciones 3 y 4*. Cuando el *main*() acaba la función asíncrona continua su ejecución completando el `Future<String>` e imprimiendo por pantalla el *String* resultante de la operación.

LIBRERIAS ADICIONALES EMPLEADAS

Como ya se comento en la introducción de este capítulo, el lenguaje con el que se ha desarrollado la aplicación móvil dispone de muchas librerías creadas por distintos desarrolladores, algunas de las cuales añaden nuevas funcionalidades donde otras solo intentan ampliar las capacidades de clases que ya existen en el lenguaje. En la aplicación móvil, se ha decidido hacer uso de las dos librerías que se presentan a continuación:

Rxdart

Librería para programación funcional basada en *ReactiveX*, que busca añadir una capa extra de funciones para facilitar el uso de objetos *Stream* y *StreamController* propios de *Dart*, y que se ha usado para gestionar, de manera más sencilla, el intercambio de datos entre las clases que "crean" la interfaz de usuario y las clases que controlan la lógica de la aplicación.

De esta librería se han usado las clases:

- *Observable*: Clase que encapsula la clase *Stream* y busca simplificar la lectura de datos por parte de distintos subscriptores de un stream (p.ej. varios widgets).

- *PublishSubject*: clase que permite gestionar/controlar un stream para lo que incluye métodos que permiten añadir datos al mismo, realizar verificaciones en función de eventos (p.ej. errores o tareas completadas), leer datos, etc.

```
PublishSubject<int> subject = new PublishSubject<int>();

/*este listener todos los int añadidos t: 1, 2, 3, ...*/

subject.stream.listen(print);// StreamSubscription 1
subject.add(1);
subject.add(2);

/*este otro listener imprimirá solo 3 pues esta después de su
inicialización*/

subject.stream.listen(print); //StreamSubscription 2
subject.add(3);
```

Figura 27: Declaración de objeto PublishSubject

- *BehaviorSubject*: similar a la clase anterior con la particularidad que recuerda el ultimo valor almacenado en el stream.

```
BehaviorSubject<int> subject = new BehaviorSubject<int>();

subject.stream.listen(print); // imprime 1,2,3
subject.add(1);
subject.add(2);
subject.add(3);
// imprime 3 pues es el ultimo valor que recuerda
subject.stream.listen(print);
```

Figura 28: Declaración de objeto BehaviorSubject

Flutter_local_notificactions

Esta librería se ha utilizado para la generación de notificaciones a horas determinadas, es compatible con API Android 16 o superior (Android 4.1 en adelante) e iOS 8.0 o superior. Pese a que *Flutter* es multiplataforma, para poder usar este paquete en Android es necesario modificar el archivo *AndroidManifest.xml* añadiendo las líneas que aparecen en la Figura 29, debido a que la programación de notificaciones afecta a la clase `AlarmManager` especifica de de ese sistema operativo.

```
<receiver android:name="com.dexterous.flutterlocalnotifications.ScheduledNotificationBootReceiver">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"></action>
    </intent-filter>
</receiver>

<uses-permission android:name="android.permission.VIBRATE" />
```

Figura 29: Datos a modificar en AndroidManifest.xml

El siguiente paso, ya en el script de la aplicación que estemos desarrollando, sería crear e inicializar un objeto *FlutterLocalNotificationsPlugin*, como vemos a continuación, en el script que vayamos a hacer uso de esta función.

```
flutterLocalNotificationsPlugin = FlutterLocalNotificationsPlugin();

var inAndorid = new AndroidInitializationSettings('@mipmap/ic_launcher');
var inIOS = new IOSInitializationSettings();
var initializationSettings = new InitializationSettings(inAndorid, inIOS);
flutterLocalNotificationsPlugin.initialize(initializationSettings,onSelectNotification:Notify);
```

Figura 30: Creación y configuración del objeto FlutterLocalNotificationsPlugin

La opción `onSelectNotification` no permitirá definir que comportamiento tendrá la aplicación cuando toquemos sobre la notificación, en este caso de ejemplo se generará un mensaje de alerta mediante el método **`Notify`** que vemos en la Figura 31.

```
Future Notify(String payload) async {
       debugPrint("payload : $payload");
       showDialog(
          context: context,
          builder: (_) => new AlertDialog(
                title: Text('TILULO'),
                content: Text('$payload'),
              ));
}
```

Figura 31: Función básica para generar diálogos de alerta

Con esto la aplicación estaría preparada para mostrar notificaciones, siendo nuestro siguiente paso la generación de las mismas.

Para establecer cada notificación es necesario especificar, como mínimo, los siguientes parámetros:

- Identificador para la notificación
- Fecha del aviso
- Titulo y contenido de la notificación
- Canal de la notificación: en dispositivos *Android* 8.0+ es necesario que todas las notificaciones estén asociadas a un canal para que los usuarios tengan la posibilidad de inhabilitar canales específicos sin afectar a todos los avisos generados por una aplicación.

```
//fecha del aviso
var scheduledTime = new DateTime.now().add(new Duration(minutes: 10));
//detalles del canal necesario para dispositivos Android 8.0+
var androidPlatformChannelSpecifics = new AndroidNotificationDetails(
    'your other channel id',
    'your other channel name',
    'your other channel description');
//detalles del canal necesario para dispositivos iOS
var iOSPlatformChannelSpecifics = new IOSNotificationDetails();
// objeto que encapsula los detalles de ambas plataformas
NotificationDetails platformChannelSpecifics = new NotificationDetails(
        androidPlatformChannelSpecifics, iOSPlatformChannelSpecifics);

await flutterLocalNotificationsPlugin.schedule(
       0, //id de la notificacion
       'scheduled title', //TITULO
       'scheduled body',  //PAYLOAD
        scheduledTime,    //FECHA
        platformChannelSpecifics);
```

Figura 32: Proceso de creación de una notificación sonora estándar

ESQUEMA DE CLASES

Por último, tras haber introducido todos los conceptos que aparecerán a continuación, podemos proceder a detallar la implementación de la aplicación que estará dividida en dos secciones, una para explicar el funcionamiento del menú de identificación de usuario y otra que describirá como se genera la lista de tareas y las notificaciones. Además, presentamos en la Figura 33 un esquema en el que se enumeran las clases que han sido creadas durante la elaboración de la aplicación junto con una descripción sencilla de su labor.

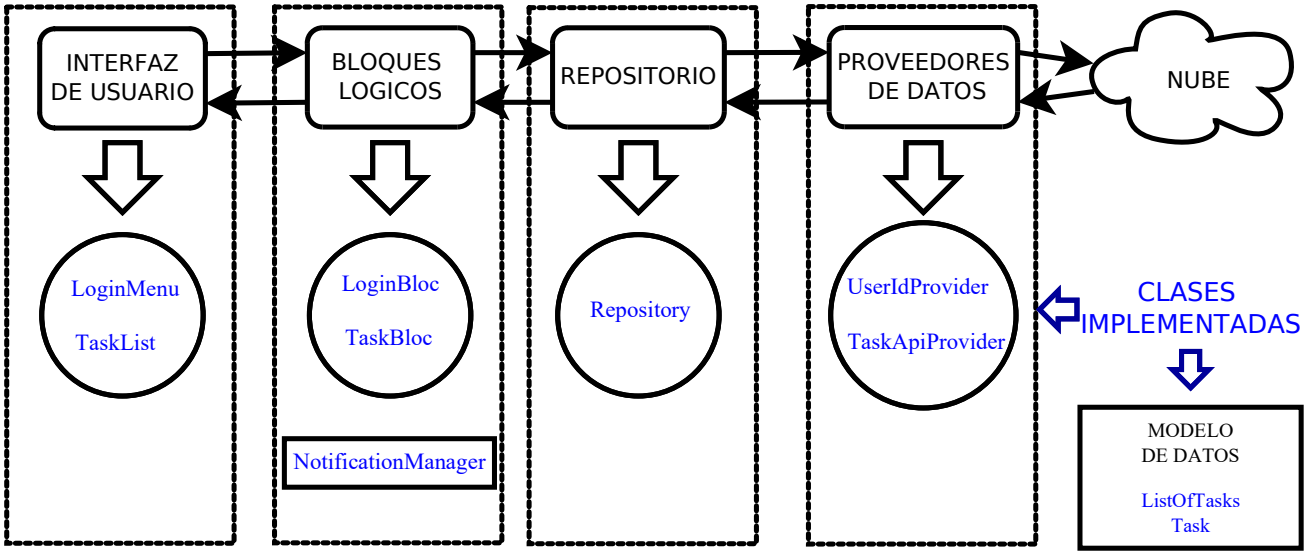

Figura 33: Diagrama y clasificación de las clases implementadas

5.2.2 Inicio de la aplicación y menú inicial de identificación

Para el inicio de nuestra aplicación se han creado los scripts *main.dart* y *app.dart* tal como se puede ver a continuación en la Figura 34.

```
main.dart
import 'package:flutter/material.dart';
import
'package:myblocflutterproyect/src/app.dart';

void main() => runApp(MyApp());
```

```
app.dart
import 'package:flutter/material.dart';
import
package:myblocflutterproyect/src/ui/login_menu.dart';
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      theme: ThemeData.light(),
      home: LoginMenu(),
    );
  }
}
```

Figura 34: Script de inicio de la aplicación móvil

Al igual que en el ejemplo inicial, los menús de identificador de usuario y lista de tareas estarán contenidos en un widget *MaterialApp* y el siguiente widget en el que ambas secciones estarán contenidas será un *Scaffold*. La clase *Scaffold* posee varias propiedades que permiten personalizar la aplicación haciendo uso de widgets como por ejemplo barras flotantes o de navegación, cajones y elementos desplegables entre otros. Este elemento, por lo general, aparece en todas la aplicaciones de *Flutter*.

Para la creación del menú de identificación que vemos en la Figura 35 se ha creado el script *login_menu.dart* donde se ha definido la clase *LoginMenu*. Esta clase extiende de *StatefulWidget* y por lo tanto realizara una llamada a la función *createState* para crear una instancia de la clase *LoginMenuState* que gestionará su estado.

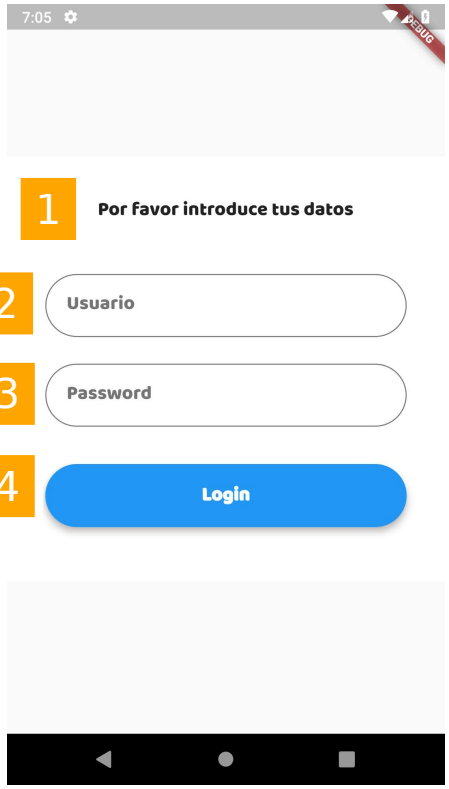

Figura 35: Pantalla inicial de la aplicación móvil con elementos etiquetados

```dart
class LoginMenu extends StatefulWidget {
  @override
  _LoginMenuState createState() => _LoginMenuState();
}
```

```dart
class _LoginMenuState extends State<LoginMenu> {
//… otras operaciones
  Widget build(BuildContext context) {
  //… otras operaciones
   return Scaffold(
    body: Center(
     child: SingleChildScrollView(
      child: Container(
       color: Colors.white,
        child: Padding(
         padding: const EdgeInsets.all(36.0),
         child: Column(
          crossAxisAlignment:CrossAxisAlignment.center,
          mainAxisAlignment:MainAxisAlignment.center,
          children: <Widget>[
              checkedId,                // 1
              SizedBox(height: 45.0),
              userField,                // 2
              SizedBox(height: 25.0),
              passwordField,            // 3
              SizedBox(height: 35.0),
              loginButon,               // 4
              SizedBox(height: 15.0),
       //...
```

Figura 36: Fragmento del script *login_menu.dart*

Debido al cierto grado de complejidad de cada uno de los elementos que aparecen en la Figura 35 ha sido necesario estructurar el método *build* de este widget de una forma distinta. Los elementos que aparecen numerados en la Figura 35 se han declarado en distintas variables que después han sido encapsuladas dentro del widget que será devuelto por el método.

En la Figura 36 vemos la jerarquía de widgets empleada: Partimos de un widget `Center` que permite centrar en la pantalla su elemento hijo `SingleChildScrollView`, este a su vez permite deslizar su hijo y así evitar que errores al exceder los límites de la pantalla, tras el cual tenemos un widget `Padding` que permite añadir un relleno alrededor de su hijo, y un widget `Column` que nos permite declarar una lista de widget que estarán posicionados de manera vertical. Los widget `SizedBox` permiten generar cajas de con unas determinadas dimensiones y posee distintas funciones (p.ej. crear botones personalizados, secciones de texto en distintas posiciones,…), en nuestro caso ha sido empleado para generar un espacio en blanco entre los componentes de la interfaz.

```
final passwordField = TextField(
  onChanged: loginBloc.changedPassw,
  obscureText: true,
  style: style,
  decoration: InputDecoration(
      contentPadding: EdgeInsets.fromLTRB(20.0, 15.0, 20.0, 15.0),
      hintText: "Password",
      border:
          OutlineInputBorder(borderRadius: BorderRadius.circular(32.0))), // InputDecoration
); // TextField
```

```
final userField = TextField(
  onChanged: loginBloc.changedUser,
  obscureText: false,
  style: style,
  decoration: InputDecoration(
      contentPadding: EdgeInsets.fromLTRB(20.0, 15.0, 20.0, 15.0),
      hintText: "Usuario",
      border:
          OutlineInputBorder(borderRadius: BorderRadius.circular(32.0))), // InputDecoration
); // TextField
```

Figura 37: Implementación de los campos de usuario y contraseña

Describamos primero los dos widget *TextField* que generan dos campos donde poder introducir usuario y la contraseña. Estos emplean su propiedad *decoration* junto con la clase *InputDecoration* para afectar su diseño. Cuando un usuario escribe sus datos en los campos usuario y contraseña, debido a la propiedad *onChanged* de estos campos, cada vez que cambien su contenido será pasado como parámetro entrada a los métodos *changedUser* y *changedPassw* de una instancia de la clase *Loginbloc*. Las llamadas a estas métodos, a su vez, generarán una llamada a al método *add*() de los

objetos *userController* y *passController* de la clase *Loginbloc*, siendo ambos de ellos de tipo *BehaviorSubject<String>*(), almacenado el texto de los campos usuario y contraseña en los streams gestionados por estos objetos, para poder ser leídos cuando los necesitemos.

```
final _userController = BehaviorSubject<String>();
final _passwController = BehaviorSubject<String>();
```

```
Function(String) get changedUser => _userController.sink.add;
Function(String) get changedPassw => _passwController.sink.add;
```

Figura 38: Funciones asociadas a los campos de usuario y contraseña

La clase *LoginBloc* se ha creado con el objetivo de obtener datos y gestionar los eventos de la interfaz de usuario creada por *LoginMenu* y a su vez informar a esta de los resultados de dichos eventos para que pueda "cambiar/reconstruirse" en función de los estos.

El siguiente elemento de la interfaz de usuario que analizaremos es el asociado a la variable `checkedId` que está marcado con la etiqueta uno en la Figura 35 de la interfaz de usuario. Este elemento emplea un tipo de widget llamado *StreamBuilder,* el cual construye su contenido en función de la última instantánea de datos contenida en el stream que este asociado a él en su propiedad *stream*. En nuestro caso, se ha definido una función *completeLogin*(`int uuid`) que nos devolverá un widget distinto en función de los datos leídos del stream suministrado por el método *streamForObtainId* de nuestra instancia de L*oginBloc*.

```
final checkedId = StreamBuilder(
  stream: loginBloc.streamForObtainId,
  initialData: ManagementCodes.initialStart,
  builder: (BuildContext context, AsyncSnapshot<int> snapshot) {

  if (snapshot.hasError) return Text(snapshot.error);
    if (snapshot.hasData) {
      return _completeLogin(snapshot.data);
    } else {
      return CircularProgressIndicator();
    }
  },
);
```

```
//...Definidos en LoginBloc
final _idController = BehaviorSubject<int>();

Observable<int> get streamForObtainId => _idController.stream;
```

Figura 39: Implementación widget *StreamBuilder* para menú de identificación

Cuando un usuario pulsando el botón de Login se desencadena un proceso del que habláramos a continuación, pero es importante destacar que cuando ese proceso finalice el contenido del stream devuelto por *streamForObtainId* (que está asociado al objeto *idController*) es susceptible de cambiar y por lo tanto afectará a este widget previo. La función *completeLogin*(`int uuid`), que usa este widget, contiene una estructura switch-case que evalúa el parámetro de entrada y devolverá uno de los siguientes widget:

- Un widget T*ext* con posibles mensajes de información como vemos en la Figura 41.
- Un widget *GestureDetector*, cuya implementación vemos en la Figura 40, que crea un contendor capaz de reconocer distintos tipos de gestos en función de las propiedades que se incluyan en él. Haciendo uso de este widget se ha creado el "botón" que nos envía a la pantalla con la lista de tareas.

```
return GestureDetector(
  onTap: () => Navigator.pushReplacement(context,
      MaterialPageRoute(builder: (BuildContext context) => TaskList())),
  child: Container(
    width: 170,
    decoration: BoxDecoration(
      borderRadius: BorderRadius.circular(20.0),
      gradient: new LinearGradient(
          colors: [Colors.lightGreen[100], Colors.cyan[100]],
          begin: Alignment.bottomLeft,
          end: Alignment.topRight), // LinearGradient
    ), // BoxDecoration
    child: Column(
      children: <Widget>[
        FlutterLogo(size: 100.0),
        Text(' OK,pulsa aqui \n para continuar', style: style),
      ], // <Widget>[]
    ), // Column
  ), // Container
); // GestureDetector
```

Figura 40: Implementación del botón de continuar

```
case ManagementCodes.initialStart:
  return Text(
    'Por favor introduce tus datos',
    style: style,
  );
  break;
case ManagementCodes.incorrectData:
  return Text(
    'Fallo usuario o Contraseña',
    style: style,
  );
  break;
case ManagementCodes.httpServerError:
  return Text(
    'Fallo de conexion, por favor intentelo pasados unos minutos',
    style: style,
  );
  break;
```

Figura 41: Implementación mensajes de error durante la identificación

Asociadas a este *GestureDetector* podemos ver la propiedad *onTap* junto con un *callback* vinculado a esta que nos permitirá cargar la clase *TaskList* del script *task_list.dart* generando en pantalla la transición a la lista de tareas.

```
final loginButon = Material(
  elevation: 5.0,
  borderRadius: BorderRadius.circular(30.0),
  color: Colors.blue[500],
  child: MaterialButton(
    minWidth: MediaQuery.of(context).size.width,
    padding: EdgeInsets.fromLTRB(20.0, 15.0, 20.0, 15.0),
    onPressed: () {
      loginBloc.fechtUserId();
    },
    child: Text("Login",
        textAlign: TextAlign.center,
        style: style.copyWith(
            color: Colors.white, fontWeight: FontWeight.bold)),
  ), // MaterialButton
); // Material
```

Figura 42: Implementación botón "Login"

Tras haber explicado los elementos que aparecen en pantalla podemos hablar del proceso que se producirá cuando el usuario pulse el botón Login, cuya implementación podemos ver en la Figura 42, cuando esto ocurre se primero que se produce es una llamada al método *fechtUserId* de *loginBloc*.

El método *fechtUserId* leerá el texto contenido de los streams asociados a *userController* y *passController* y pasara el valor de estos como parámetro en una llamada al método asíncrono *fechtUserId* de la variable *repo* (instancia de *Repository*). Esta llamada empleará la instrucción *await* esperando hasta obtener el resultado de la operación y lo almacenará en la variable *obtainedUuidFromHttpCall* y lo añadirá al stream controlado por el objeto *idController*.

```
void fechtUserId() async {
  String user = _userController.value;
  String passw = _passwController.value;

  if (_userController.hasValue && _passwController.hasValue) {
    _idController.sink.add(null);
    _obtainedUuidFromHttpCall = await _repo.fechtUserId(user, passw);
    print(_obtainedUuidFromHttpCall);
    _idController.sink.add(_obtainedUuidFromHttpCall);
  } else {
    print(_userController.hasValue && _passwController.hasValue);
  }
}
```

Figura 43: Implementación del método *fechtuserId* de la clase *LoginBloc*

Nota: Recordemos que el stream relacionado con *idController* es el leído por el *StreamBuilder* marcado como uno en la Figura 36 (implementación en la Figura 39).

La clase *Repository*, contenida en el script *repository.dart,* mencionada antes se crea con pensamiento de disponer una clase que englobe las llamadas a todos los métodos de todas aquellas clases que suministren datos a la aplicación desde distintas fuentes. En el caso actual, como vemos

en la Figura 44, una llamada al método *fechtUserId* de una instancia de *Repository* causa, a su vez, una llamada al método *fechtUserId* de la clase *UserIdProvider* contenida en el script *user_id_provider.dart*.

```
class Repository {
  final TaskApiProvider _taskApiProvider= TaskApiProvider();
  final UserIdProvider _userIdProvider= UserIdProvider();

  Future <ListOfTasks> fetchAllTask(int uuid) =>_taskApiProvider.fetchTaskList(uuid);
  Future <int> fechtUserId(String user, String password) => _userIdProvider.fechtUserId(user, password);
}
```

Figura 44: Implementación clase *Repository*

Será la clase *UserIdProvider* la que llevará a cabo una petición HTTP de tipo POST al *endpoint /api/users/login* de nuestro servidor REST adjuntando en el cuerpo de la petición los valores de usuario y contraseña que fueron obtenido previamente por la clases anteriores. En la Figura 45 podemos ver la estructura del método *fechtUserId* de la clase *UserIdProvider*.

```
Future<int> fechtUserId(String user, String password) async {
  String url = "https://              .eu-gb.mybluemix.net/api/users/login";

  var content = {'user': user, 'passw': password};

  try {
    final response = await _client.post(url, body: content);

    debugPrint('user_id_ provider -> Code: ${response.statusCode}');

    switch (response.statusCode) {
      case 200:
        int uuid = json.decode(response.body) == null
            ? 0
            : json.decode(response.body);
        return uuid;
        break;
      case 401:
        debugPrint('user_id_ provider::= ${response.body}');
        return ManagementCodes.incorrectData;
        break;
      default:
        debugPrint('user_id_ provider::= ${response.body}');
        return ManagementCodes.httpServerError;
        break;
    }
  } catch (e) {
    debugPrint('user_id_ provider exception -> $e');
    return ManagementCodes.httpServerError;
  }
```

Figura 45: Implementación del método *fechtUserId* de la clase *UserIdProvider*

Dependiendo del código de estado incluido en la respuesta del servidor, el método devolverá el identificador del usuario registrado o un identificador de error que recorrerá el camino inverso hasta llegar a *loginBloc* donde se almacenará en la variable *obtainedUuidFromHttpCall* (que puede ser recuperada mediante una llamada a un método con el mismo nombre) y se añadirá al stream controlado por el objeto *idController*. Al haber un valor nuevo en el stream asociado a *idController*, el widget *StreamBuilder* se reconstruirá haciendo uso de la función *completeLogin*(`int uuid`).

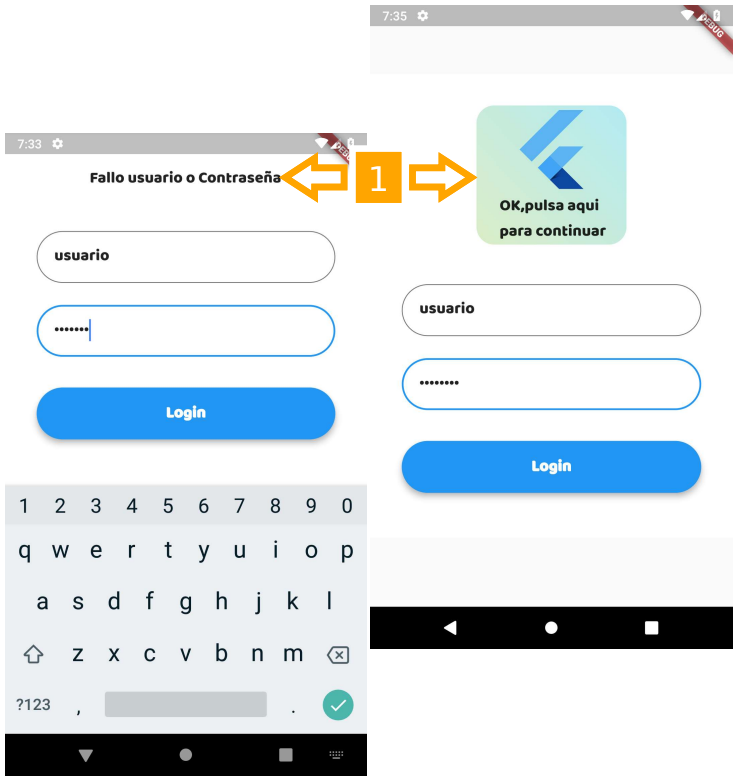

Figura 46: Pantalla inicial tras procesos de identificación correcto y fallido

El diagrama de la Figura 47 busca mostrar el proceso explicado hasta ahora de una forma más gráfica. Como vemos en este diagrama, durante toda esta cadena de eventos se trabaja con funciones que han de esperar una respuesta por parte de un servidor HTTP, esto ha hecho necesario crear funciones que emplean la instrucción *async* y que devuelven objetos *Future<T>* (en concreto *Future<int>* para el procedimiento descrito).

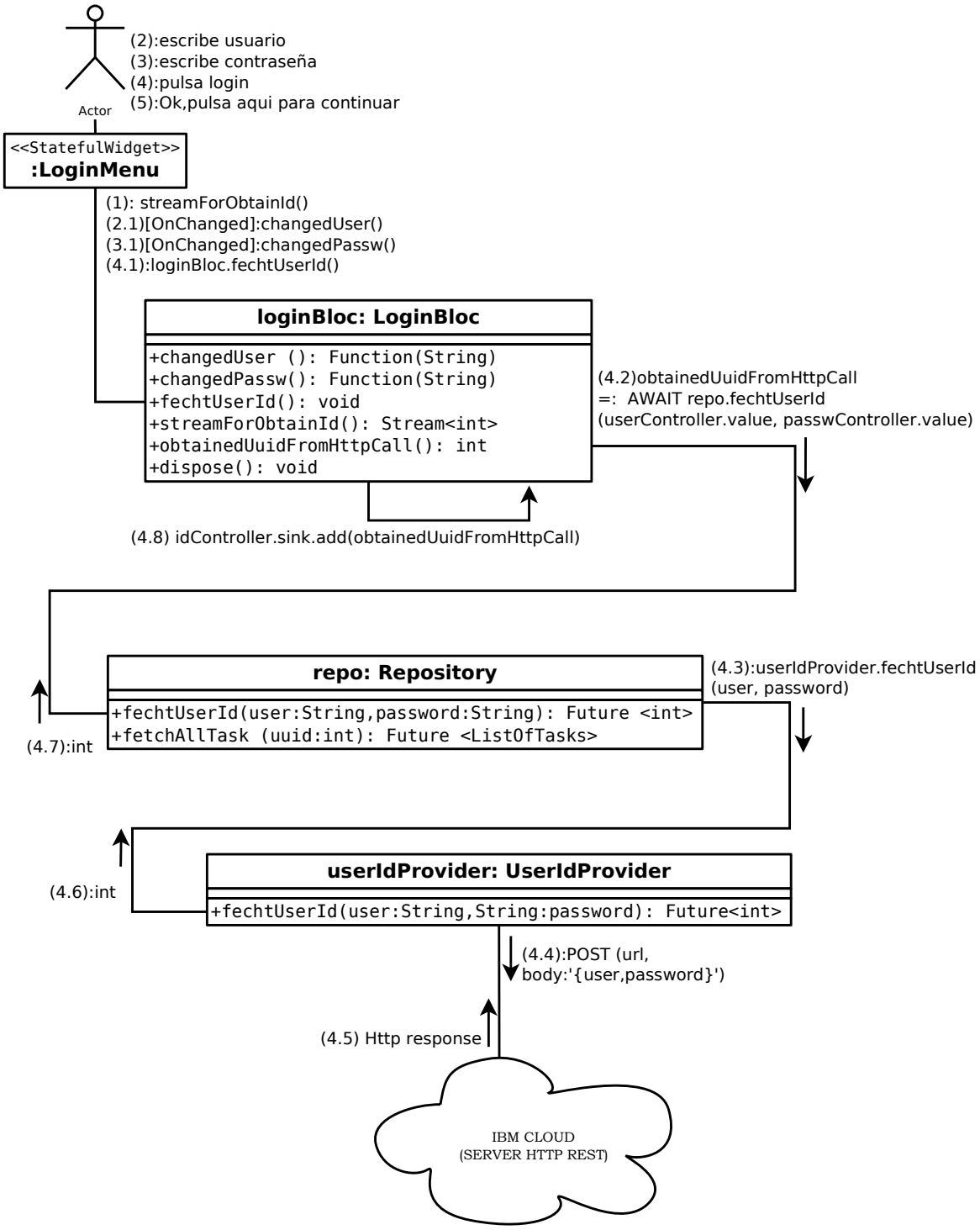

Figura 47: Diagrama de colaboración del proceso de identificación completo

La Figura 48 presenta el diagrama de secuencia que muestra el proceso de comunicación entre la aplicación móvil y el servidor.

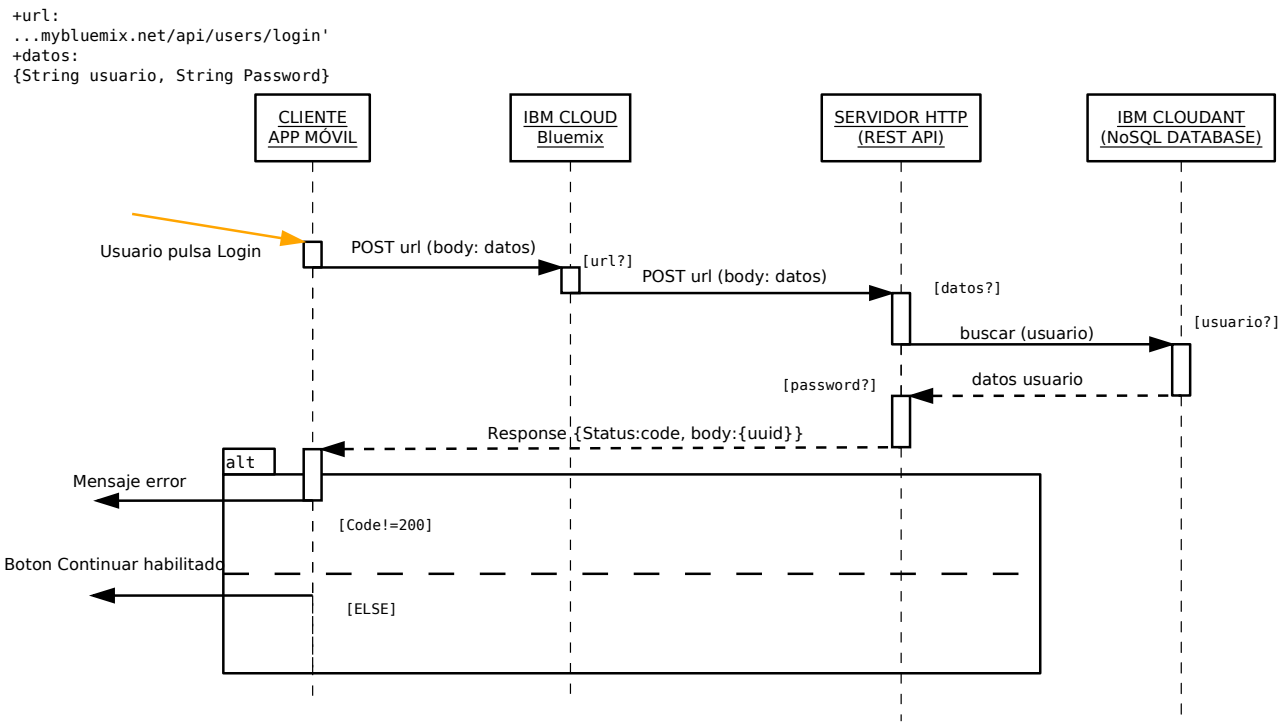

Figura 48: Diagrama de secuencia del proceso de identificación de usuario

5.2.3 Menú de lista de tareas y generación de notificaciones

Tras haber finalizado el proceso de identificación se habilitara el botón de continuar permitiendo al usuario acceder a la lista de tareas gracias al *callback* asociado a la propiedad `onTap` del mismo. Esta función emplea el método estático `pushReplacement` de la clase *Navigator* de *Flutter*.

```
onTap: () => Navigator.pushReplacement(context,
             MaterialPageRoute(builder: (BuildContext context) => TaskList())),
```

Figura 49: Método estático *pushReplacement* de la clase *Navigator* para salto entre pantallas

Siguiendo un esquema similar al ya explicado, la aplicación creará la siguiente pantalla/pagina haciendo uso de la clase *TaskList* implementada en el script con nombre *task_list.dart*. La lógica detrás de ella será gestionada usando una instancia de la clase *TaskBloc* (de nombre *taskBloc*) programada en el script *task_bloc.dart*.

```
class TaskList extends StatefulWidget {
  @override
  State<StatefulWidget> createState() {
    return TaskListState();
  }
}
class TaskListState extends State<TaskList> {
//… }
```

Figura 50: Declaración de la clase *TaskList*

La clase *TaskList* hereda de *StatefulWidget* y, por lo tanto, también hará uso del método *createState*() para generar un objeto *TaskListState* que gestione su estado. Este objeto contendrá tres variables de interés:

- Una instancia de la clase *FlutterLocalNotificationsPlugin*, para la configuración de las notificaciones, con nombre `flutterLocalNotificationsPlugin`.
- Una instancia de la clase *Timer* de *Dart* llamada `periodicGetDataTimer`.
- Una lista que contendrá los identificadores de las tareas que el usuario marque como completadas con nombre `doneTaskIds`.

Estas variables serán necesarias para distintas operaciones que serán descritas posteriormente.

```
@override
void initState() {
   super.initState();
//1º
     // NOTIFICATION PLUGIN CONFIGURATION
     flutterLocalNotificationsPlugin = new FlutterLocalNotificationsPlugin();
     var inAndorid = new AndroidInitializationSettings('@mipmap/ic_launcher');
     var inIOS = new IOSInitializationSettings();
     var initializationSettings = new InitializationSettings(inAndorid, inIOS);
     flutterLocalNotificationsPlugin.initialize(initializationSettings,
         onSelectNotification: _selectNotification);
//2º
     // OBTAIN INITIAL DATA IN THE STREAM AND GENERATE NOTIFICATIONS
     _updateDataStream();
     //taskBloc.fechtAllTask(flutterLocalNotificationsPlugin, doneTaskIds);
//3º GENERATE TIMER FOR PERIODIC DATA UPDATE
    periodicGetDataTimer = new Timer.periodic(Duration(minutes: 5), (Timer t) {
      if ((DateTime.now()).hour == 0 && doneTaskIds.isNotEmpty) {
        doneTaskIds.clear();
      }
      lastUpdate = TimeOfDay.now().toString();
      _updateDataStream();
    });
}
```

Figura 51: Método *initState* de la clase *TaskList*

Cuando un *StatefulWidget* es creado por primera vez, este ejecutara, por defecto, su función *initState*() para definir cuál es su estado inicial, ocurriendo lo mismo con el objeto *TaskListState* que ejecutará este método, que hereda de la clase padre, antes de llamar al método *build* widget. En nuestro caso es necesario que, en este punto de la ejecución de la aplicación, se lleven a cabo otras operaciones de configuración, por lo que se ha hecho uso de la constante *@override* para modificar

el método *initState*() e incluir dichas operaciones tal como vemos en la Figura 51. Analicemos a continuación estas operaciones que se producen antes de que se genere el widget:

1º Se inicializan las opciones de configuración del objeto *flutterLocalNotificationsPlugin* que será empleado para generar las notificaciones y se especifica qué función se ejecutara cuando toquemos sobre una notificación que aparezca en pantalla. Para esta última operación se ha creado la función asincrona *selectNotification* que realiza un llamada a la función *genericDialogAlert*.

```
Future _selectNotification(String payload) async {
    if (payload != null) {
      debugPrint('notification payload: ' + payload);
    }
    genericDialogAlert('AVISO PARA HOY', 'ES LA HORA DE: $payload');
  }
```

```
Future genericDialogAlert(String title, String content) {
  return showDialog(
      context: context,
      barrierDismissible: false, // user must tap button!
      builder: (_) => new AlertDialog(
            title: Text(title),
            content: Text(content),
            actions: <Widget>[
              FlatButton(
                child: Text('Ok'),
                onPressed: () {
                  Navigator.of(context).pop();
                },
              ),
            ],
          ));
}
```

Figura 52: Funciones para la creación de avisos de alerta o notificación

La función *genericDialogAlert* se ha creado para generar mensajes de alerta con los datos pasados como parámetro haciendo uso de la función *showDialog* definida en *Flutter*, que permite mostrar un cuadros de dialogo sobre al actual contenido que se encuentre en pantalla.

2º Ejecutamos la función *updateDataStream* que realizará una llamada al método asíncrono *fechtAllTask* de nuestra instancia de *TaskBloc* pasando como parámetros el objeto *flutterLocalNotificationsPlugin* ya configurado y la lista *doneTaskIds* con los identificadores de las tareas marcadas como completadas.

```
void _updateDataStream() {
    taskBloc.fechtAllTask(flutterLocalNotificationsPlugin, doneTaskIds);
}
```

Figura 53: Función *updateDataStream* para la obtención de datos de la lista de tareas

Este método lleva a cabo las labores de obtención de los datos necesarios para generar la lista de tareas y la generación de las notificaciones mientras que la aplicación continua construyendo la interfaz de usuario. En un punto posterior veremos con mayor profundidad cómo se realiza este procedimiento.

3º Se inicializará la variable *periodicGetDataTimer* que periódicamente ejecutará el *callback* que contiene una llamada al método *updateDataStream* permitiéndonos actualizar la información de la lista de tareas y vaciar la lista *doneTaskIds* cuando cambiemos de día. Para comprobar el correcto funcionamiento de este *timer,* cada vez que este temporizador se ejecute almacenaremos la hora de su activación en el *String lastUpdate* que usaremos más adelante para generar un aviso en pantalla.
Tras finalizar el método *initState* se ejecutara el método *build* del widget generando el árbol de elementos. El diagrama de la Figura 54 busca facilitar la comprensión de la jerarquía de elementos creada por el método *build* debido grado de complejidad de esta por la cantidad de widgets empleados en la generación de esta sección.

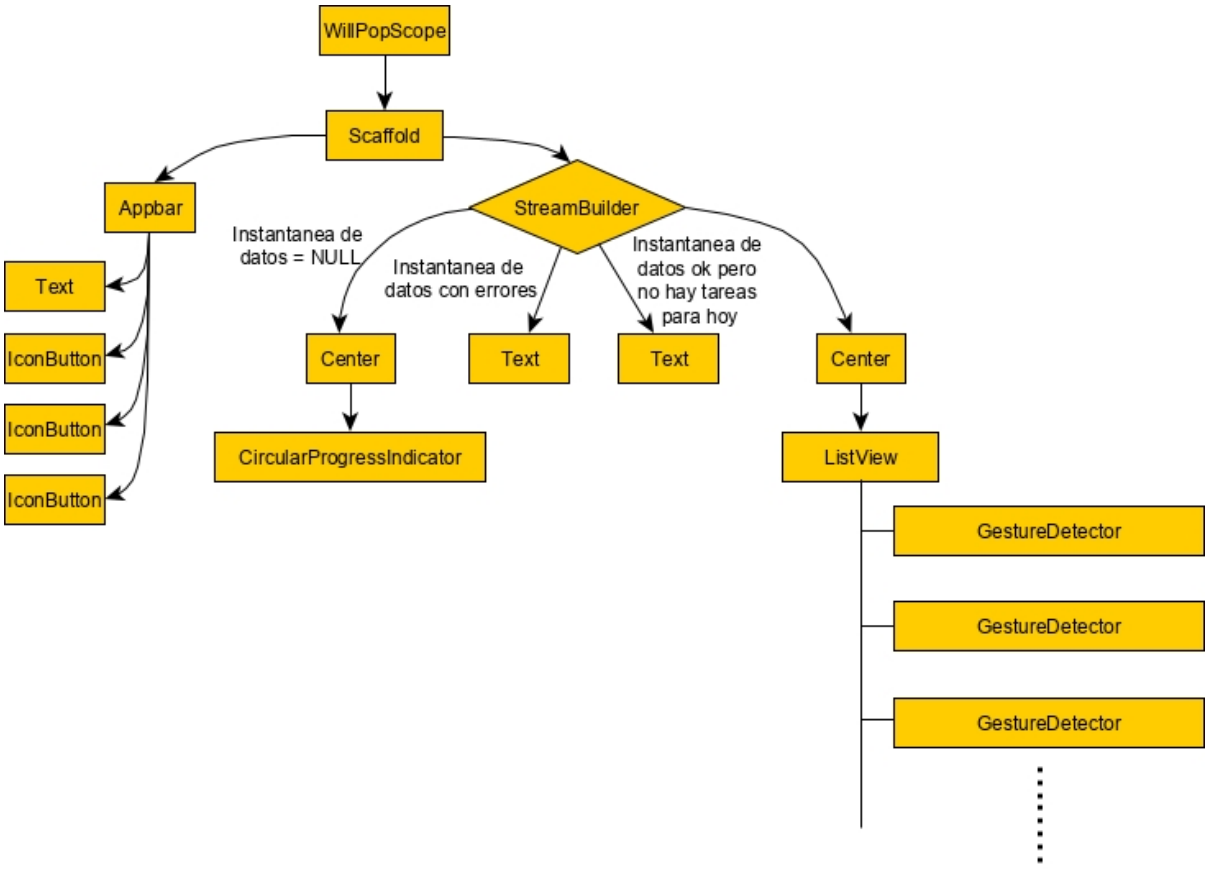

Figura 54: Árbol de widgets generado por la clase *TaskBloc*

Nuestra jerarquía de widgets se empieza generando un widget *WillPopScope* que se usa para cancelar la opción de volver a la ruta anterior. Esto evitara que el usuario pueda accidentalmente acceder al menú anterior.

```
return WillPopScope(

      onWillPop: () => Future.value(false),
      child: Scaffold(
               //...
```

Figura 55: Widget WillPopScope

Después encontraremos un widget *Scaffold* compuesto por dos partes. Por un lado, una barra de opciones con widget *Text,* para el titulo, junto con los iconos que se comentaron en al capítulo cuatro.

```
appBar: AppBar(title: Text("COSAS POR HACER"), actions:
<Widget>[

          IconButton(
            icon: Icon(Icons.refresh),
            tooltip: '',
            onPressed: _updateDataStream,
          ),
          IconButton(
            icon: Icon(Icons.access_alarms),
            tooltip: '',
            onPressed: _checkNotifications,
          ),
          IconButton(
            icon: Icon(Icons.adb),
            tooltip: '',
            onPressed: _checkLastUpdate,
          ),
        ]),
```

Figura 56: Implementación de la barra de opciones

Como podemos ver en la Figura 56 estos iconos tienen asociadas las funciones en su propiedad `onPressed`:

- *updateDataStream*: función creada para actualizar la lista de tareas.
- *checkLastUpdate*: Esta función emplea la variable *lastUpdate* como parámetro de entrada de la función *genericDialogAlert* para generar un aviso que informa de la última hora a la que se actualizo la lista de tareas de forma automática.
- *checkNotifications*: Esta función utiliza el método *pendingNotificationRequests* del objeto *flutterLocalNotificationsPlugin* para obtener el numero de aviso pendientes y usa esta información junto con la función *genericDialogAlert* (Figura 52) para generar un mensaje con la cantidad de tareas pendientes.

Además de la barra de opción, la parte principal del *Scaffold* es construida utilizando un widget *StreamBuilder* que usara la última instantánea de datos obtenidos del stream (controlado por el objeto *taskFetcher*) devuelto por el método *allTask* del objeto *taskBloc* para generar uno de los siguientes elementos:

```dart
//...Definidos en la clase TaskBloc
final _taskFetcher = PublishSubject<ListOfTasks>();

  /* GET TO OBTAIN DE DATA IN THE STREAM*/
  Observable<ListOfTasks> get allTask => _taskFetcher.stream;
```

```dart
  Widget buildListOfTaks(AsyncSnapshot<ListOfTasks> snapshot) {

    return Center(
        child: ListView(
      children: _buildList(snapshot),
    ));
  }
```

```dart
body: StreamBuilder(

    stream: taskBloc.allTask,
    builder: (BuildContext context, AsyncSnapshot<ListOfTasks> snapshot) {
     if (snapshot.hasError) return Text(snapshot.error.toString());                //(4)
      if (snapshot.connectionState == ConnectionState.waiting || !snapshot.hasData)
        return Center(                                                             //(2)
          child: CircularProgressIndicator(),
         );
      if (snapshot.data.results.isEmpty) {
        return Center(                                                             //(3)
          child: Text("No hay nada para hoy",
              style: TextStyle(
                  fontFamily: 'BalooBhai',
                  color: Colors.blue[300],
                  fontSize: 30.0,
                  fontWeight: FontWeight.bold)),
        );
      }
       return buildListOfTaks(snapshot);                                  // ListView  (1)
    },
),
```

Figura 57: Widget StreamBuilder empleado en la generación de la lista de tareas

(1) La lista de tareas que será creada usando un widget *Center* que contendrá un widget *ListView* que, como su nombre indica, sirve para desplegar una lista deslizable.

(2) Un indicador de espera empleando el widget *CircularProgressIndicator* dentro de un widget *Center.*

(3) Un texto informativo, usando un widget *Text* anidado dentro de un widget *Center*, en caso de que no existan tareas para el día actual.

(4) Un texto, empleando también un widget *Text*, cuando la ultima instantánea de datos presente algún tipo de error.

El contenido de este widget puede cambiar cada vez que se ejecuta la función *updateDataStream* debido a que, como ya sabemos, esta llama a la método *fechtAllTask* del objeto *taskBloc* el cual tras realizar las operaciones necesarias para generar las notificaciones de las tareas, actualizará el contenido de stream controlado por el objeto *taskFecher* vinculado a este widget.

ntes de continuar con el análisis de cómo se generan los elementos de la interfaz, es necesario hacer un **inciso** para describir la clase *ListOfTask*, usada en el widget anterior, junto con la clase *Task* ambas contenidas en el script *ítem_model.dart*.

La clase *ListOfTask* ha sido creada con el pensamiento poder trabajar con objetos que encapsulen la información recibida desde el servidor facilitando la gestión de esta información en funciones o métodos y simplificar la generación de la lista de tareas de la interfaz.

```dart
class ListOfTasks {

 List<Task> _results = [];                          //(1)
  ListOfTasks(){                                    //(3)
    print("Non-parameterized constructor of ListOfTasks");
  } //default constructor

  ListOfTasks.fromJson(List<dynamic> parsedJson) {//(4)
    List<Task> temp = [];

    if (parsedJson.length > 0) {
      for (int i = 0; i < parsedJson.length; i++) {
        Task t1 = Task(parsedJson[i]);
        temp.add(t1);
      }
    }
    _results = temp;
  }

  List<dynamic> get results => _results;           //(2)
}
```

```dart
class Task {
  String _id;
  String _rev;
  String _type;
  int _day;
  String _name;
  String _hour;
  bool _done;
  int _uuid;
  String _photo;
//(Constructor)
  Task(element) {
    _id = element['_id'];
    _rev = element['_rev'];
    _type = element['type'];
    _day = element['day'];
    _name = element['name'];
    _hour = element['hour'];
    _done = element['done'];
    _uuid = element['uuid'];
    _photo =element['photo'];
  }
//(Metodos)
  String get id => _id;
  String get rev => _rev;
  String get type => _type;
  int get day => _day;
  String get name => _name;
  String get hour => _hour;
  bool get done => _done;
  int get uuid => _uuid;
  String get photo => _photo;

  set setDone(bool isDone) {
    _done = isDone;
  }
```

Figura 58: Implementación de las clases *Task* y *ListOfTasks*

Esta clase contiene:

(1) Un parámetro llamado *results* que consistirá en una lista de objetos tipo *Task*.

(2) Un método que nos devuelva *results* cuando sea invocado.

(3) Un constructor por defecto.

(4) Un constructor *ListOfTasks.fromJson* que nos permitirá inicializar este objeto haciendo uso de una lista de elementos `List<dynamic>` que pasemos como parámetro de entrada.

Por otro lado tendremos la clase *Task* que nos permite crear objetos que poseen los parámetros necesarios para almacenar la información de las tareas que sea recibida desde el servidor y un conjunto de métodos que nos permiten recuperar dichos parámetros junto con un método (*setDone*) se usara en el proceso de marcar las tareas que han sido completadas.

Una vez que conocemos la estructura de datos con la que trabaja la aplicación, podemos continuar explicando cómo se generan los widgets que forman la lista de tareas cuando disponemos de dichos datos.

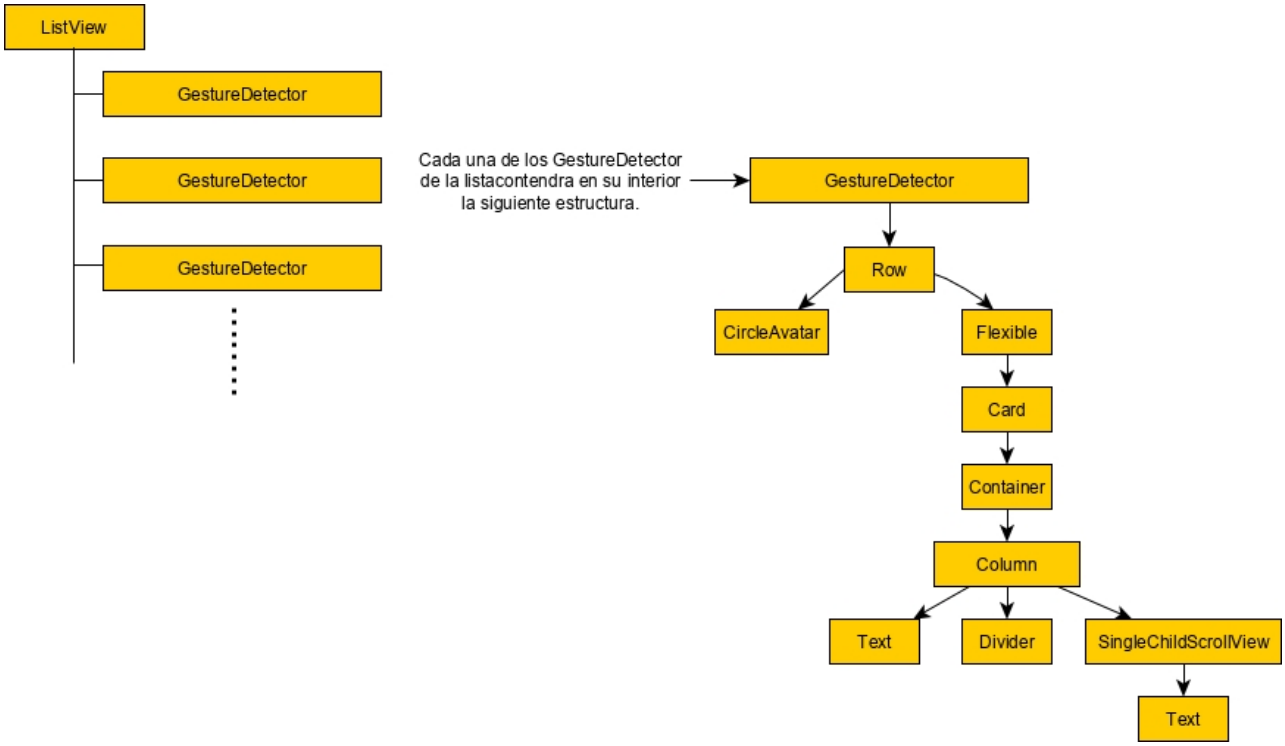

Figura 59: Árbol de wigdets de cada elemento de la lista de tareas

Esta labor se lleva a cabo en la función *buildList* haciendo uso del constructor *List.generate*, proporcionado por *Flutter*, que como resultado final nos devolverá una lista de elementos *GestureDetector* empleando el los datos del contenidos en la propiedad *results* del objeto tipo *ListOfTasks* que es obtenido del parámetro de entrada dicha función.
Dentro de cada *GestureDetector* encontraremos una conjunto de widgets *Row*, *Container*, *Card* y *Column* que se han empleado para personalizar el aspecto de la interfaz usando sus propiedades `margin`, `elevation`, `padding`, `decoration`, `crossAxisAlignment`, `verticalDirection`, etc. De

entre los widgets que afectan al estilo de la interfaz los únicos widgets con una función un poco más especial serian:

- Widget *Flexible*: aporta a su widget hijo la capacidad de expandirse hasta llenar el espacio disponible en el eje principal.
- Widget *SingleChildScrollView*: permite a capacidad deslizar el contenido del widget y es útil cuando dicho contenido (el widget hijo) excede los límites/bordes del espacio asignado al widget padre.

Por otro lado, cada *GestureDetector* obtenido del constructor *List.generate* poseerá un *callback* asociado a sus propiedades `onDoubleTap` y `onLongPress`. En el *callback* de `onDoubleTap` se ejecutará cuando un usuario realice una pulsación doble sobre la tareas, en el se comprobará si el identificador de la tarea pulsada (`results[index].id`) fue previamente incluido en la lista *doneTaskIds* y en caso de no estar se realizan dos operaciones:

1º Se incluirá ese id en la lista *doneTaskIds*.
2º Se cambiará el valor de la propiedad *done,* de este objeto tipo *Task,* a *true* en un *callback* que se encuentra dentro de una llamada al método *setState*. Este método es usado en los *StatefulWidget* y para notificar al framework que se ha producido un cambio en el estado interno de este objeto causando que el contenido de la pantalla se actualice.

```
List<GestureDetector> _buildList(AsyncSnapshot<ListOfTasks> snapshot) {
    int count = snapshot.data.results.length;
    List<Task> results = snapshot.data.results;
    /* .
         .
   */    .
    List<GestureDetector> elementos = List.generate(
        count,
      (index) => GestureDetector(
          onDoubleTap: () {
            if (!doneTaskIds.contains(results[index].id)) {

              doneTaskIds.add(results[index].id);
              setState(() {
                results[index].setDone = true;
              });
            }
          },
          onLongPress: () {
            if (doneTaskIds != null &&
                doneTaskIds.contains(results[index].id)) {

              doneTaskIds.remove(results[index].id);
              setState(() {
                results[index].setDone = false;
              });
            }
          },
           child: Row(// …
```

Figura 60: Función para la generación de elementos de la lista de tareas

El *callback* de `onLongPress` realizara el proceso contrario, cuando se realice una pulsación larga sobre el widget asociado a un objeto *Task* se comprobara si el identificador de ese objeto está en la lista *doneTaskIds,* en caso positivo se sacara de la lista y se cambiara la propiedad *done* de ese objeto a *false* dentro de una llamada al método *setState*.

Las acciones definidas en los elementos *GestureDetector* unido al hecho de que el valor de la propiedad *done* de cada objeto *Task* es verificado cuando se construyen los widget *Card* y *CircleAvatar* dará a la aplicación como resultado la capacidad de resaltar aquellas tareas de la lista que han sido completadas.

```
CircleAvatar(
    backgroundImage: results[index].done          //Condicional
        ? AssetImage('assets/images/well_done.png')
        : AssetImage(results[index].photo),
    radius: deviceVariableRadio),
```

```
Card(

    margin: EdgeInsets.all(10),
    elevation: 20.0,
    child: Container(
    padding: const EdgeInsets.all(5.0),
    decoration: new BoxDecoration(
    color: results[index].done              //Condicional
           ? Colors.amber
           : Color.fromRGBO(0, 0, 0, 0.05),
              ),

    child: Column(//.....
```

Figura 61: Widgets empleados para marcar tareas

Figura 62: Aspecto de una tarea tras ser marcada como completada

Por último, dentro del último elemento *Column* de la jerarquía asociado a cada *Task*, se generará el contenido de las imágenes que vemos en la parte superior haciendo uso de la instrucción $*NobreDeVariable* (o ${*expresión*}) empleada en *Dart* para generar un String a en función del contenido de una variable o expresión asociada, que en nuestro caso serán los valores de las propiedades de cada *Task*.

```
<Widget>[
  Text('TAREA ${index + 1}',
      style: TextStyle(
      fontFamily: 'BalooBhai',
      color: Colors.blue[300],
      fontSize: 20.0,
      fontWeight: FontWeight.bold)
      ),
   Divider(),
   SingleChildScrollView(
     Child: Text('Hora: ${results[index].hour} \nHay que hacer: ${results[index].name}',
            style: TextStyle(
            fontFamily: 'BalooBhai', fontSize: 16.0)),
            ),
],
```

Figura 63: Widget para mostrar la información de una tarea

Una vez que conocemos como se construye la interfaz de usuario, nuestro siguiente paso será analizar estructura la función *fechtAllTask* (del objeto *taskBloc*) y las labores que se llevan a cabo dentro ella para suministrar de datos a la interfaz de usuario y, además, generar las notificaciones de las tareas a determinadas horas.

```
//fragmento de la CLASE TASKBLOC
final Repository _repo = Repository();
final _notificationManager = NotificationManager();
//...

fechtAllTask(FlutterLocalNotificationsPlugin flutterLocalNotificationsPlugin, List<String>
doneTaskIds) async {

  _taskFetcher.sink.add(null);
  ListOfTasks newListOfTasks = await repo.fetchAllTask(loginBloc.obtainedUuidFromHttpCall);


//CHECK IF THE NEW TASKS RECIEVED ARE DONE
  if (newListOfTasks != null) {
    if (newListOfTasks.results.isNotEmpty && doneTaskIds.isNotEmpty) {
      newListOfTasks.results.forEach((task) {
        if (doneTaskIds.contains(task.id)) task.setDone = true;
      });
    }

/*GENERATE SCHELUDED NOTIFICATIONS WITH THE RECIEVED DATA */
    _notificationManager.buildNotifications(flutterLocalNotificationsPlugin, newListOfTasks);


/* ADD RECIEVED DATA TO THE STREAM */
      _taskFetcher.sink.add(newListOfTasks);  /* Nuevos datos a leer por la interfaz*/
    }
  }
```

Figura 64: Implementación del método *fechtAllTask* de la clase *TaskBloc*

Cuando esta función es ejecutada en primer proceso llevado a cabo es la obtención de los datos necesarios para generar la lista de tareas para lo cual ejecuta el método *fetchAllTask* de una instancia clase *Repository*. A este método se le pasará como parámetro de entrada de entrada el identificador de usuario actual, obtenido durante el proceso de registro, que puede ser obtenido mediante una

llamada a la función *obtainedUuidFromHttpCall*() del objeto *loginBloc* explicado en el punto anterior.

```
class Repository {
  final TaskApiPorvider _taskApiPorvider= TaskApiPorvider();
  final UserIdPorvider _userIdProvider= UserIdPorvider();

  Future <ListOfTasks> fetchAllTask(int uuid) =>_taskApiPorvider.fetchTaskList(uuid);
  Future <int> fechtUserId(String user, String password) => _userIdProvider.fechtUserId(user, password);
}
```

El método *fetchAllTask* de la clase *Repository* genera una llamada al método *fetchTaskList* de una instancia de la clase *TaskApiProvider* que esta implementada en el script *task_api_provider.dart* y que será la encargada de realizar la petición HTTP al servidor y gestionar la respuesta obtenida por este.

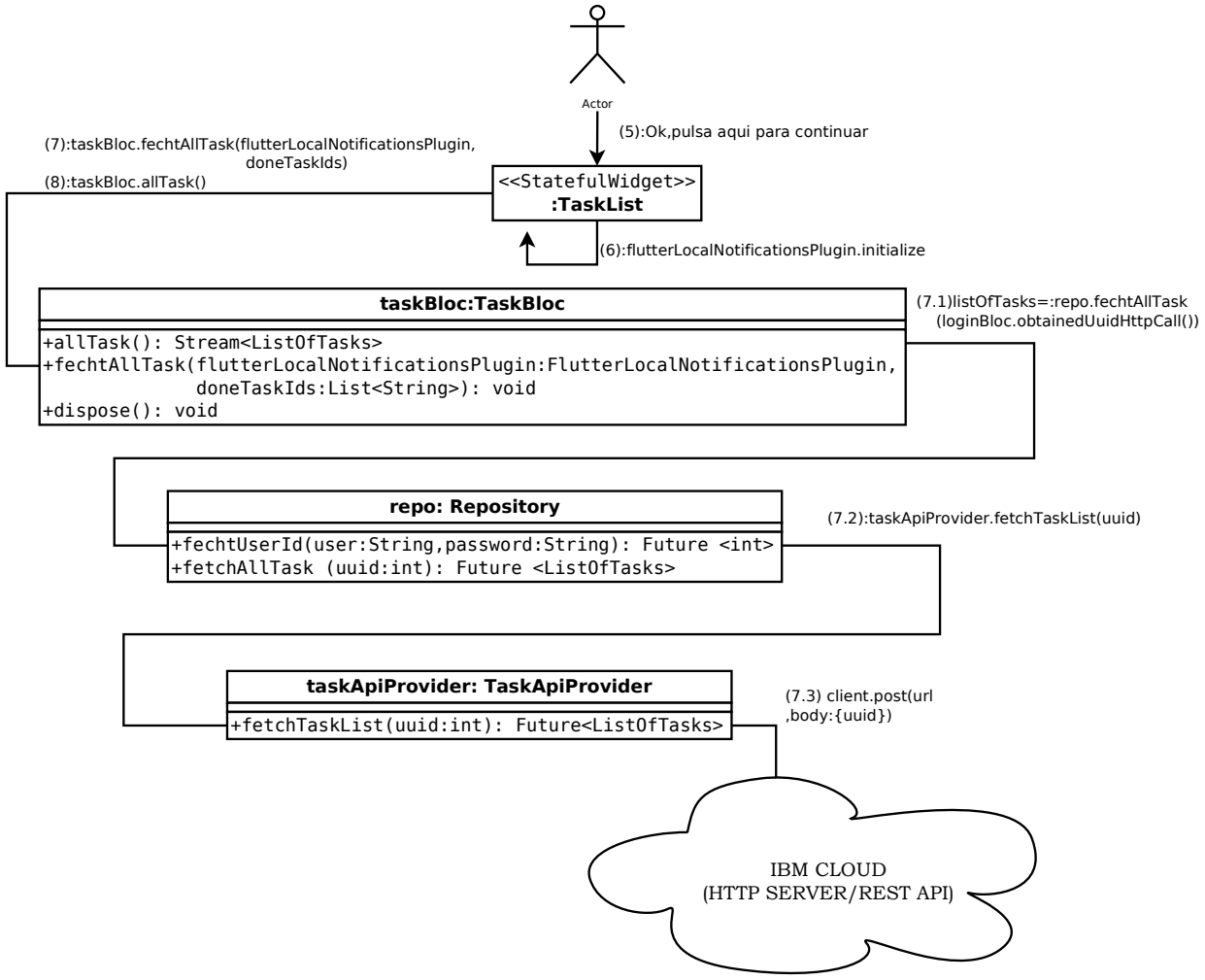

Figura 65: Diagrama de una petición de datos para generar lista de tareas

Una vez que la petición llegue al *endpoint /api/ obtain_task/day* del servidor este realizara una consulta la base de datos obteniendo el conjunto de tareas de ese usuario para el día actual y adjuntará esta información en la respuesta que le enviará al la aplicación móvil.

```dart
import 'package:http/http.dart' show Client;
import 'package:myblocflutterproyect/src/models/item_model.dart';

class TaskApiProvider {
  Client client = Client();

  Future<ListOfTasks> fetchTaskList(int uuid) async {
    String _url = "https://proyecto0104.eu-gb.mybluemix.net/api/obtain_tasks/day";

    var _body={'uuid':'$uuid'};

    try {
      final response = await client.post(_url, body:_body); //  Expection HERE

      if (response.statusCode == 200) {
        return ListOfTasks.fromJson(json.decode(response.body));
      } else {
        debugPrint('code:${response.statusCode} Error: ${response.body}');
        return null;
      }
    } catch (e) {
      debugPrint('catch:');
      debugPrint('$e');
      return null;
    }
  }
}
```

Figura 66: Implementación clase *TaskApiProvider*

Cuando la aplicación reciba la respuesta por parte del servidor verificara el código de estado incluido en ella, si este es correcto empleará la información recibida para crear un objeto de tipo *ListOfTask* usando en constructor *fromJson* de la clase tras usar la función *json.decode*[2] de *Flutter*. El objeto de tipo *ListOfTask* creado será devuelto y llegará hasta el método *fechtAllTask* de *taskBloc* donde se generó la llamada y se almacenara en la variable *newListOfTasks*.

En caso de que de recibir una respuesta del servidor que incluya un código de error el método devolverá *null* que finalmente será recibido por el método *fechtAllTask* de *taskBloc* causando que no se lleven a cabo los siguientes procesos descritos en el método.

Tras esto se verificaran los identificadores de las *Task* contenidas en este objeto comparándolos con los que están almacenados en la lista *doneTaskIds* cambiando la propiedad *done* de aquellos que se encuentran marcados en la interfaz de usuario. En caso de no realizar este proceso la aplicación la interfaz construiría la lista de tareas con todas ellas desmarcadas cada vez que obtengamos un nuevo conjunto de tareas.

El siguiente paso llevado a cabo es la generación de las notificaciones asociadas a cada tarea haciendo uso del método asíncrono *buildNotifications*, de una instancia de la clase N*otificationManager* definida en script *notifications_management.dart*, pasando como parámetros de

[2] Función empleada para obtener un objeto JSON de un String suministrado

entrada la información de las tareas recibida almacenada en la variable *newListOfTasks* y el objeto de *flutterLocalNotificationsPlugin* configurado previamente.

Por último, se añadirá el objeto *newListOfTasks* al stream controlado por el objeto *taskFetcher* que está siendo "escuchado" por el elemento *StreamBuilder* de la interfaz grafica causando que este reaccione al nuevo contenido se reconstruya usando la nueva información recibida.

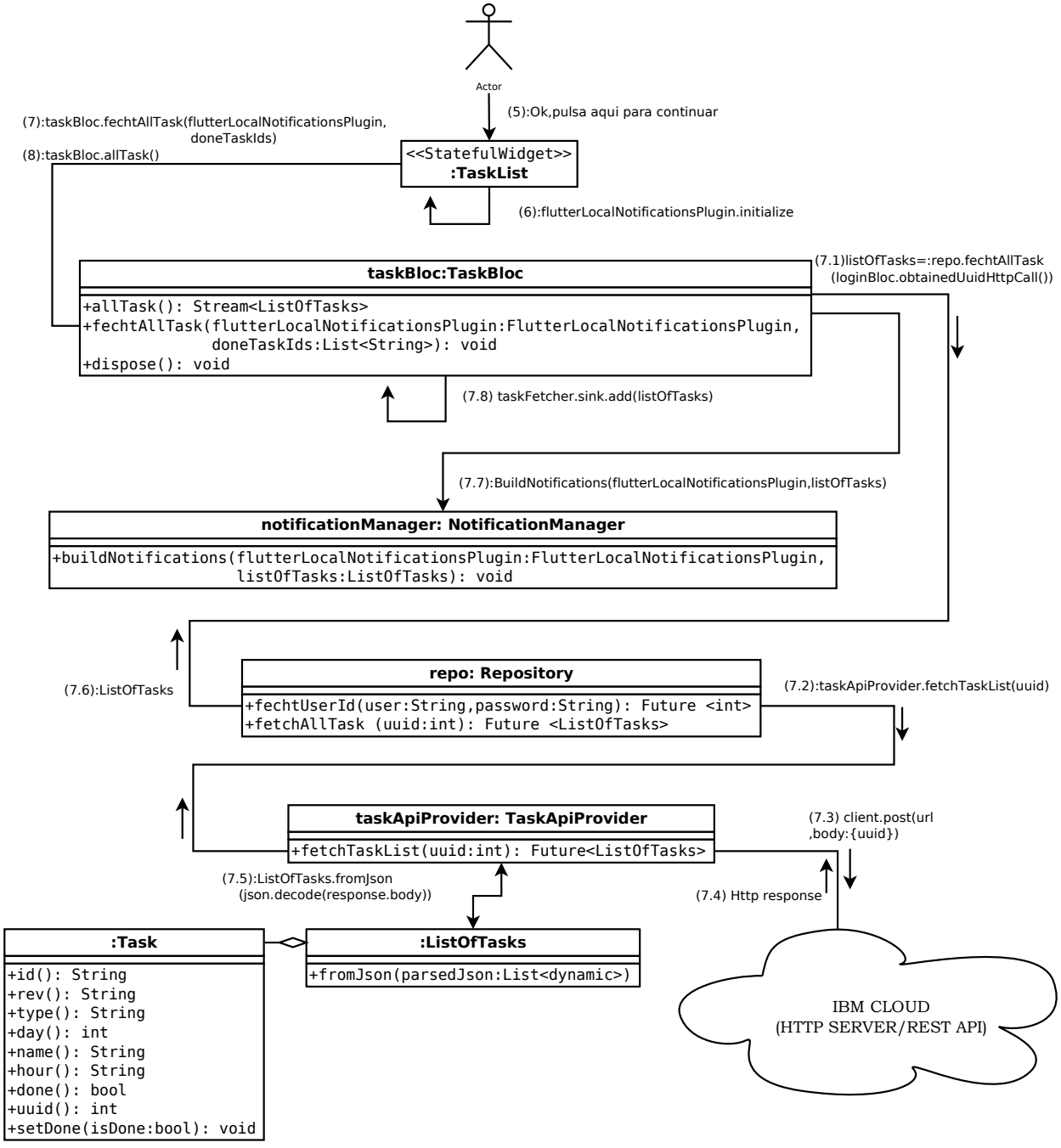

Figura 67: Diagrama del proceso de generación de la lista de tareas en la interfaz

La Figura 67 muestra todo el proceso descrito previamente en el que podemos ver la cadena de eventos desde el momento en el que el usuario accede a este menú tras haber iniciado sesión.

Además, también se adjunta un diagrama con la secuencia temporal y el intercambio de menajes con la nube.

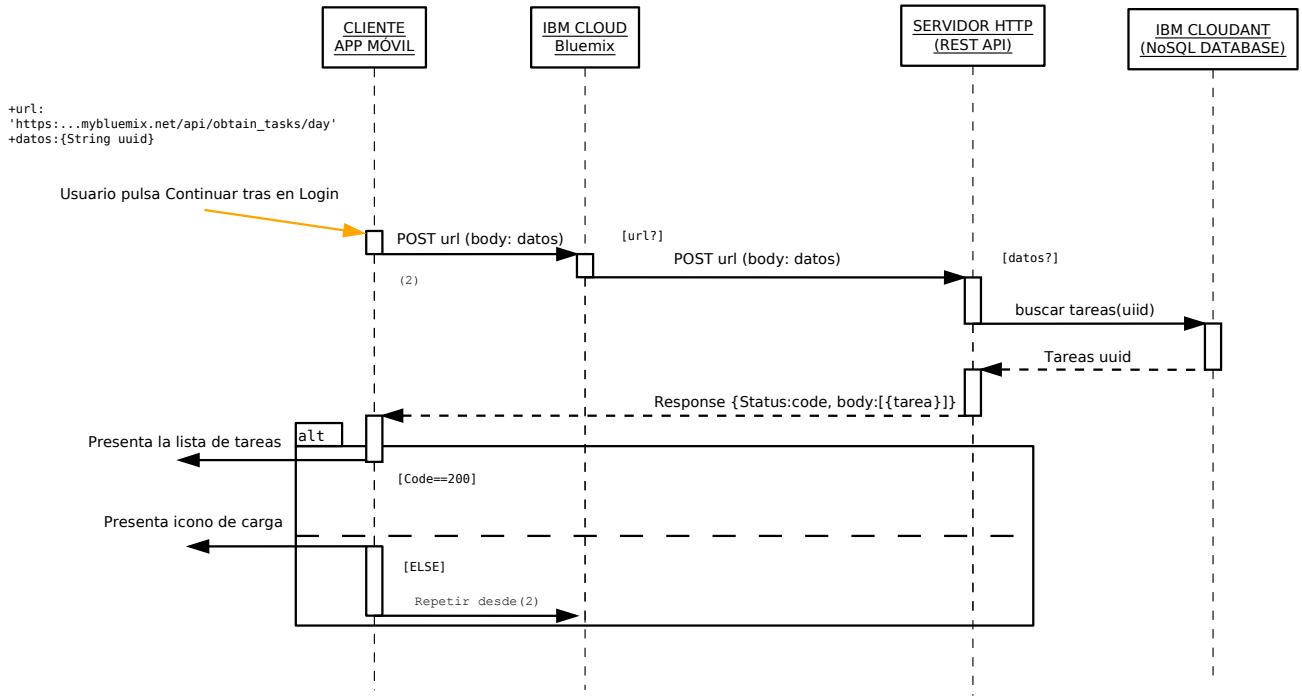

Figura 68: Diagrama secuencia del proceso para obtener los datos de las tareas

Por último queda pendiente por describir el funcionamiento de la clase *NotificationManager* en la cual se ha creado el método asíncrono *buildNotifications* para la generación de avisos.

Cada vez que la aplicación obtiene los datos para crear una nueva lista de tareas, mediante el proceso explicado previamente, se llevará a cabo una llamada al método *buildNotifications* que incluirá como parámetros de entrada un objeto *listOfTasks* y un objeto *flutterLocalNotificationsPlugin*. A continuación el método *buildNotifications* llevará a cabo el siguiente procedimiento:

1º Se verificará si existen notificaciones previas y en caso de ser cierto estas serán canceladas para no generar conflictos con las nuevas notificaciones que se generarán a continuación.

```
var isEmpty =(await flutterLocalNotificationsPlugin.pendingNotificationRequests()).length;

    if (isEmpty != 0) {
      await flutterLocalNotificationsPlugin.cancelAll();
    }
```

Figura 69: Cancelación de notificaciones programadas

2º Se creará un objeto tipo *DateTime* con la fecha y hora actual que se almacena variable con nombre *now*.

3º Para cada *Task* dentro de la lista devuelta por *listOfTasks.results* se crea un objeto tipo *DateTime* con la fecha actual y la hora obtenida del parámetro *hour* de este objeto.

```
for (int i = 0; i < listOfTasks.results.length; i++) {
     var taskToSchelude = listOfTasks.results[i];

     var hhmm = taskToSchelude.hour.split(':');

     var next = new DateTime(now.year, now.month, now.day, int.parse(hhmm[0]),
          int.parse(hhmm[1]), 0);
     if (now.difference(next).isNegative){ //(now-next<0)?

       // ... var androidPlatformChannelSpecifics = ...

    }
```

Figura 70: Bucle necesario para de creación de notificaciones

4º Verificamos si el objeto *DateTime* que se ha creado en el tercer paso hace referencia a un instante temporal posterior al del objeto *now* que fue creado en el segundo paso, en caso de ser así se procederá a generar una notificación para dicha tarea siguiendo el procedimiento explicado en la documentación de la librería *Flutter_local_notificactions* que fue previamente descrita.

```
var androidPlatformChannelSpecifics = new AndroidNotificationDetails(
   'Channel id',
   'Channel name',
   'Channel description',
   importance: Importance.Max,
   priority: Priority.High,
   );
   var iOSPlatformChannelSpecifics = new IOSNotificationDetails();
   var platformChannelSpecifics = new NotificationDetails(
      androidPlatformChannelSpecifics, iOSPlatformChannelSpecifics);

   await flutterLocalNotificationsPlugin.schedule(
       i, //id de la tarea
       'Tienes pendiente: ' + taskToSchelude.name,
       ' para la hora: ' + taskToSchelude.hour,
       next, // DATE TIME OF taskToSchelude
       platformChannelSpecifics,
       payload: taskToSchelude.name+ ' ' +taskToSchelude.hour);//texto a mostrar
```

Figura 71: Proceso de configuración de las notificaciones de la aplicación

5.3 Implementación y estructura de la aplicación web

En esta última sección de este capítulo se aporta una descripción breve de la estructura de la aplicación web creada para que los padres puedan introducir datos en la aplicación móvil. Centráramos nuestra exposición en aspectos relacionados con el funcionamiento de la aplicación y solo se comentarán de manera superficial aquellas características relacionadas con el aspecto o la presentación de la misma.

Como previamente se comento en el capítulo en un capitulo anterior, en la aplicación se han empleado, por un lado, el lenguaje de marcas HTML junto con framework *Bootstrap* para generar la presentación de la misma y, por otro, se ha empleado el framework *AngularJS* para el control de la lógica detrás de la misma. Para poder hacer uso de estos framework el documento *index.html* sobre el que se ha trabajado presenta, dentro de varias etiquetas *script*, referencias a distintos archivos necesarios para el funcionamiento de los mismos que son aportadas en sus respectivas documentaciones.

Figura 72: Diseño de aplicación web empleado Bootstrap

El framework *Boostrap* ua sido usado debido a posee un amplio con numero de etiquetas que pueden ser asociadas al atributo *class* de cada elemento en un documento HTML permitiendo al desarrollador generar con rapidez distintos estilos para la presentación de la aplicación. En la aplicación se ha hecho uso de algunos de dichos identificadores (`row`, `container`, `col-lg-4`, `col-`

`md-12, table, table-bordered, table-responsive-md, p-5,…`) junto con el sistema de rejilla que posee framework para generar un diseño simple en el que los distintos elementos del documento están encapsulados en dos contenedores que a su vez están englobados cada uno en su propia columna como vemos en la Figura 72.

Por otro lado, *AngularJS* facilita el desarrollo de la lógica de negocio que hay detrás de una aplicaciones web debido aporta un conjunto de atributos extendidos usando directivas que permiten asociar el modelo de datos a la vista presentada por el documento HTML mediante el uso de ciertas expresiones.
A continuación se explican las directivas de mayor interés que han sido empleadas en esta aplicación web y se nombraran ciertas funciones que se ha desarrollado para realizar las labores de obtención de la lista de tareas de un usuario, la adición de nuevas tareas y la eliminación de tareas mediante peticiones HTTP al servidor:

- *ng-app*: Empleada para definir una aplicación *AngularJS*. En nuestro caso esta aplicación se encuentra definida dentro de un *script* con el nombre *ngModel.js*.

- *ng-controller:* Permite asociar un objeto controlador definido en la aplicación a un determinado fragmento/elemento de documento HTML. Dicho controlador puede contener funciones y variables que son parte de un determinado objeto del controlador y que estarán disponibles para el elemento HTML asociado. En *AngularJS* este objeto es llamado *scope*.

```
<!—index.html -->
<body ng-app="ngModule" class="bg-info">
    <div class=" container bg-white">
        <header class="container">
            <div class="jumbotron">
                <h1>MI LISTA DE TAREAS</h1>
                <p>Bienvenido Usuario </p>
            </div>
        </header>

        <main class="container " ng-controller=mainCtrl>
            <div class="row">
              <!-- ...
                   ... -->
```

```
// script ngModule.js

var IndexModule = angular.module('ngModule', ['ngSanitize']);

IndexModule.controller('mainCtrl', ['$scope', '$location', '$http', function ($scope,
$location, $http, ) {
//...
```

Figura 73: Directivas ng-app y ng-controller empleadas en la aplicación web

En nuestro caso, se ha definido un único controlador con el nombre *mainController* cuyo objeto *scope* permitirá que el contenedor *main* del documento *index.html* pueda acceder a las variable y funciones definidas dentro del mismo.

- *ng-model*: empleada para asociar el valor de un elemento HTML (input, select, textarea,…) a una determinada variable definida en la aplicación. Ha sido empleada en los elementos HTML del formulario donde se selecciona el día de la tarea, su descripción, la hora de la misma y una imagen asociada a la tarea. Cuando un usuario modifica estos elementos, sus valores serán almacenados en un conjunto de variables del *scope* del controlador (*day*, *name*, *time*, *imageSelected*) que serán empleadas posteriormente por la función *postTask* para enviar la información de nuevas tareas hacia el servidor y que este las almacene en la base de datos.

- *ng-init*: Permite inicializar un dato de la aplicación al inicio de la misma. Esta directiva se ha utilizado para ejecutar una función con nombre *refreshData* del controlador con objetivo de inicializar, al comienzo de la aplicación, un *array tasksList* que contendrá la lista de tareas de un usuario y que es empleado para generar la tabla con todas tareas que vemos en la aplicación.

- *ng-repeat*: Permite crear una instancia de un determinado elemento por cada elemento dentro de una colección.

```
<div class="col-lg-8 col-md-12">

     <div id="listoftasks" ng-init="refreshData()">
          <table class="table table-bordered table-responsive-md">
                 <tr>
                    <th>DIA</th>
                    <th>TAREA</th>
                    <th>hour</th>
                     <th></th>
                 </tr>
                 <tr ng-repeat="eachtask in tasksList">
                     <td>{{eachtask.day }}</td>
                     <td>{{eachtask.name }}</td>
                     <td>{{eachtask.hour }}</td>
                     <td><input type="button" ng-click="borrar(eachtask)" value="borrar"></td>
                  </tr>
           </table>
      </div>
</div>
```

Figura 74: Ejemplo de uso de la directiva ng-repeat

Esta directiva ha sido empleada para la generación de las filas que componen la tabla de tareas, empleando la información contenida en el *array taskList*. Además cada una de estas filas presentará

un botón que permitirá ejecutar la función *borrar* para eliminar este elemento de la lista de tareas almacenado en la base de datos mediante una petición HTTP al servidor. Esta función será explicada junto con la siguiente directiva.

- *ng-click*: directiva que permite ejecutar un determinado comportamiento cuando un elemento es pulsado. Esta directiva ha sido asociada a los botones que se pueden ver en la interfaz y causará que se ejecuten las funciones *postTask* y *borrar* en función del botón pulsado.

Tras explicar el funcionamiento de la directivas empleadas, solo nos queda analizar los detalles de mayor importación de la implementación de las tres funciones mencionas. Las tres funciones mencionadas harán uso del servicio *$http* de *AngularJS* para poder realizar peticiones al servidor.

En primer lugar tenemos la función *refreshData* que, al ser ejecutada, realiza una petición HTTP POST al *endpoint* '*/api/obtain_task_list*' del servidor en la que se incluirá el identificador del usuario. Si esta petición es contestada por el servidor con un código de estado OK, la función ejecutara un *callback* que actualizará el contenido de la variable *tasksList* causando que se refresque la tabla de tareas visible en el documento HTML.

```
$scope.refreshData = function () {

    //... creación de la variable tasksWithID que contine ide de usuario

    $http({
        method: 'POST',
        url: './api/obtain_task_list',
        timeout: 60000,
        data: JSON.stringify(tasksWithID), // tasksWithID:=({"uuid": int})
        headers: {
            'Content-Type': 'application/json'
        }
       }).then(function successCallback(response){

            $scope.tasksList = response.data;
                 //...

       }), function errorCallback(response) {

              //...

       }
}
```

Figura 75: Fragmento de la implementación de la función *refreshData*

La función *postTask* está vinculada mediante la directiva *ng-click* al botón "Añadir Tarea" de la interfaz. Al ejecutar esta función se genera un objeto JavaScript *newTask* que contiene la información de una nueva tarea y a continuación, tras usar la función *JSON.stringify,* se enviará este elemento al *endpoint* '*/api/tasks*' del servidor mediante una petición HTTP POST para que este incluya dicha información en la base de datos. Si durante este proceso no aparece ningún error, el servidor contestara la petición del cliente web con un código de estado OK lo que causara que el cliente realice un *callback* ejecutando la función *refreshData* para actualizar la información de la tabla de tareas.

```
$scope.postTask = function () {
        //... creación de la variable newTask

        $http({
            method: 'POST',
            url: './api/tasks',
            timeout: 60000,
            data: JSON.stringify(newTask),
            headers: {
                'Content-Type': 'application/json'
            }

        }).then(function successCallback(response) {
            //...
            $scope.refreshData();

        }, function errorCallback(response) {
            //...
        });
    }
```

```
Estructura objeto
newTask
{
"type": "task",
"day": int,
"name": String,
"hour": String,
"done": false,
"uuid": int,
"photo": String

}
```

Figura 76: Fragmento de la implementación de la función *postTask*

Por último, la función *borrar* está asociada mediante la directiva ng-click a cada uno de los botones "borrar" que aparecen en la tabla de tareas.

```
$scope.borrar = function (t) {
        //... creación de la variable deletedTask

        $http({
            method: 'DELETE',
            url: './api/delete_tasks',
            data: JSON.stringify(deletedTask),
            headers: {
                'Content-Type': 'application/json'
            }
        }).then(function successCallback(response) {
            //...
            $scope.refreshData();

        }, function errorCallback(response) {
            //...
        });

}
```

```
Estructura objeto
deletedTask
{
"_id":String,

"_rev": String,
}
```

Figura 77: Fragmento de la implementación de la función *borrar*

Cuando uno de los botones “borrar” de la tabla de tareas es pulsado se ejecutara la función asociada y a esta se le pasara como parámetro el elemento asociado a la línea de la tabla cuyo botón ha sido pulsado. Usando los atributos _id y _rev de este elemento, se generará un objeto *deletedTask* que será enviado al *endpoint* ‘*/api/delete_tasks*’ del servidor dentro de una petición HTTP DELETE para que este elimine dicha tarea de la base de datos. Igual que en el caso de la función anterior, si el procedimiento funciona correctamente el servidor enviara un código de estado ok al cliente web lo que causara que se realice un *callback* que llamara a la función *refreshData* que actualizará la información de la tabla de tareas.

CAPÍTULO 6: CONCLUSIONES Y LINEAS FUTURAS

En este capítulo se plantearan algunas nuevas líneas de desarrollo que pueden ser llevadas a cabo empleando como base el prototipo propuesto en este proyecto y se realizará una valoración de las tecnologías empleadas para este prototipo.

6.1 Objetivos alcanzados.

En este proyecto se ha desarrollado un prototipo de aplicación para un dispositivo móvil, conectado a la nube, capaz de ser integrado en un juguete infantil. Tal y como se planteó, la app puede servir de asistente personal para fomentar el desarrollo del sentido de responsabilidad y la organización temporal del usuario.

Para ello, se han revisado diferentes tipos de tecnologías y se han seleccionado las más prometedores desde los puntos de vista de facilidad de uso y funcionalidad (en el siguiente punto se ofrece una visión crítica de las tecnologías utilizadas). Todas ellas, como se planteó en un principio, permiten un desarrollo fluido para incluir futuras ampliaciones. Se puede decir, por tanto, que los objetivos generales, planteados en un principio, se han conseguido de forma satisfactoria.

La parte de selección y uso de tecnologías supuso un esfuerzo considerable, limitando la cantidad de funcionalidad que se ha podido incluir. Uno de los puntos más importantes ha sido entender el diseño de las librerías utilizadas con objeto de poder usarlas. Como contrapartida, estos patrones de diseño aparecen de forma exhaustiva en la memoria proyecto. Aquella funcionalidad que por razones de tiempo y alcance del proyecto ha quedado fuera del trabajo realizado se enumeran en la última sección, trabajos futuros.

6.2 Visión crítica de las tecnologías utilizadas

De entre las tecnologías empleadas en este desarrollo, algunas de ellas poseen ciertos rasgos que pueden suponer un reto e incluso llegar generar algo de confusión a aquellos desarrolladores que comienzan a hacer uso de ellas. En concreto trataremos las tecnologías empleadas en la aplicación móvil, debido a que este proyecto ha requerido un estudio más profundo de las mismas es comparación al resto.

El framework *Flutter* usado para la generación de interfaces móviles permite desplegar la plantilla de un proyecto en cuestión de minutos y sus arquitectura basada en objetos tipo Widget es fácil de comprender, pero por otro lado al ser un producto relativamente joven carece de la profundidad de otras herramienta y de un gran número de ejemplos didácticos que traten labores más concretas, además el conjunto de librerías externas de código libre que posee una variedad limitada. Estos problemas fueron afrontados en este proyecto durante su etapa inicial cuando se intento incluir la función para generar notificaciones de usuario asociadas a las tareas. Otro detalle observado durante este proceso de desarrollo es que las aplicaciones *Flutter* suelen tener un tamaño mucho mayor en comparativa con aquellas implementadas usando otros lenguajes, siendo este rasgo, según se ha investigado, uno de los principales que el equipo de Google intenta optimizar.
Por otro lado, el lenguaje de programación *Dart* también empleado en la aplicación móvil, aunque presenta ciertas similitudes con otros lenguajes como Java o JavaScript, puede llegar a ser difícil de tratar para un desarrollador cuando este intenta encontrar soluciones para problemas específicos debido a que su sintaxis y su enfoque al tratar estos (el cual puede llegar a ser algo inusual).

6.3 Trabajos futuros

Al tratarse de un prototipo el diseño presentado, las aplicaciones que lo componen y su organización, se encuentra ampliamente sujeto a cambios, adición de nuevos servicios o mejoras con idea de optimizar su funcionamiento. Por ello, tras realizar una cierta valoración, a continuación se comentan algunas ideas sobre futuras líneas de trabajo.

La primera ampliación que se ha considerado de gran interés e importancia hace referencia a labores para mejorar la seguridad de todos los procesos en los que se lleva a cabo intercambio de información entre servidor y clientes desarrollados. El objetivo es evitar posibles usos maliciosos de la información de los clientes en caso de que se amplíen las funciones de aplicación y estas contengan información de carácter más sensible.

Por otro lado, debido a que quedaba fuera del alcance del proyecto, la aplicación web carece de secciones para el registro y la identificación de usuario para los padres, la cuales son de vital importancia. Por lo tanto otro posible desarrollo seria la ampliación de la aplicación web como del servidor para llevar a cabo estas funciones.

Otra posible línea de trabajo de interés implicaría la posibilidad de alterar la estructura de aplicaciones planteada o la posibilidad de incluir algún nuevo servicio que sea capaz de informar a los padres en la aplicación web de las tareas que han sido marcadas como completadas por sus hijos en la aplicación móvil.

Un posible futuro trabajo implicaría el estudio sobre nuevos servicios de la nube capaces de dotar a las aplicaciones desarrolladas de un bajo nivel inteligencia artificial. En el caso de IBM sería interesante analizar los servicios Watson aportados por dicho proveedor.

Por último, una posible rama de trabajo trataría sobre la modificación de la estructura de la aplicación móvil desarrollada usando el framework *Flutter* tras haber realizar un análisis en profundidad del grado de optimización de la misma bajo el condicionante de que esta esté integrada en dispositivos con características limitadas, por ejemplo, debido a algún tipo de condición impuesta por el fabricante.

REFERENCIAS

AIJU. (2018). CYBERCLOUD4TOYS Desarrollo de entornos cloud cyberseguros para el uso componentes electrónicos y tecnologías IOT para el desarrollo de juguetes innovadores en el sector [Pdf]. Recuperado de https://www.aiju.info/wp-content/uploads/2019/02/Informe-Web-CYBERCLOUD4TOYS.pdf

AngularJS. (2018). *AngularJS API Docs*. Recuperado de https://docs.angularjs.org/api

Cloud Foundry. (s.f.). En *Wikipedia*. Recuperado el 29 de mayo de 2019, de https://en.wikipedia.org/wiki/Cloud_Foundry

Codecacademy. (s.f). *What is REST? Learn about how to design web services using the REST paradigm*. Recuperado de https://www.codecademy.com/articles/what-is-rest

CogniToys: Internet-connected Smart Toys that Learn and Grow. (s.f.). Recuperado de https://www.kickstarter.com/projects/cognitoys/cognitoys-internet-connected-smart-toys-that-learn

Continuous delivery. (20 de agosto de 2019). En *Wikipedia*. Recuperado el 20 de junio de 2019, de https://en.wikipedia.org/wiki/Continuous_delivery

Dart. (2019). *Dart documentation*. Recuperado de https://dart.dev/guides

Dart Packages. (2019). *rxdart 0.22.2*. Recuperado el 17 de mayo de 2019, de https://pub.dev/packages/rxdart

Dart Packages. (2019). *flutter_local_notifications 0.8.2*. Recuperado de https://pub.dev/packages/flutter_local_notifications

DevOps. (7 de julio de 2017). *Conceptos de MVC (Modelo, Vista, Controlador)*. Recuperado el 30 de junio de 2019, de https://devopstic.wordpress.com/2017/07/07/conceptos-de-mvc-modelo-vista-controlador

Documentación IBM Cloud. (2018). *SDK for Node.js*. Recuperado de https://cloud.ibm.com/docs/runtimes/nodejs?topic=Nodejs-getting-started#getting-started-tutorial

Documentación IBM Cloud. (2019). *IBM Cloudant*. Recuperado de https://cloud.ibm.com/docs/services/Cloudant?topic=cloudant-creating-an-ibm-cloudant-instance-on-ibm-cloud&locale=es#creating-an-ibm-cloudant-instance-on-ibm-cloud

Express - framework para aplicaciones web Node.js. (2019). Recuperado de https://expressjs.com/

Flutter. (2019). *Flutter Documentation*. Recuperado de https://flutter.dev/docs

Flutter. (2019). *The Engine architecture*. Recuperado de https://github.com/flutter/flutter/wiki/The-Engine-architecture

Flutter. (s.f.). Inicio [Canal de YouTube]. Recuperado de https://www.youtube.com/channel/UCwXdFgeE9KYzlDdR7TG9cMw

HTML. (21 de agosto de 2019). En *Wikipedia*. Recuperado el 22 de junio de 2019, de https://es.wikipedia.org/wiki/HTML

IBM. (s.f.). *IaaS, PaaS and SaaS – IBM Cloud service models*. https://www.ibm.com/cloud/learn/iaas-paas-saas

IBM Cloud. (26 de julio de 2018). *Cloud Foundry*. Recuperado de https://cloud.ibm.com/docs/cloud-foundry-public?topic=cloud-foundry-public-about-cf#about-cf

Introducing JSON. (s.f.). Recuperado de https://json.org/

Node.js. (23 de agosto de 2019). En *Wikipedia*. Recuperado el 20 de junio de 2019, de https://es.wikipedia.org/wiki/Node.js

Octopus by Joy, the training wheels for good habits!. (s.f). Recuperado de https://www.kickstarter.com/projects/octopusbyjoy/the-icon-based-watch-that-young-kids-can-read-and

Platform_as_a_service. (08 de agosto de 2019). En *Wikipedia*. Recuperado el 29 de junio de 2019, de https://en.wikipedia.org/wiki/Platform_as_a_service

REST API Tutorial. (s.f.). Recuperado de https://restfulapi.net/