Arquitectura de microservicios en un sistema de reserva de hoteles

Marc Espinosa Ibañez

Resumen— En este artículo se presenta el proceso realizado para construir un sistema de reserva de hoteles hecho con la arquitectura de microservicios. La finalidad es crear un sistema bien implementado para poder comparar las características de éste con las que suele tener una aplicación monolítica. El resultado de este proyecto, determina que ninguna de las dos estructuras es mejor una ante la otra, ya que la necesidad que tenga el cliente con la aplicación va a determinar cual escoger y cual descartar. En el caso de ser una aplicación muy pequeña sin perspectiva de un crecimiento en el futuro ni modificaciones, la elección más óptima es una estructura monolítica. Por el contrario, si la previsión es añadir más funcionalidades y adaptaciones con otros sistemas, la decisión es la arquitectura en microservicios. El motivo es que con la primera arquitectura los costes en todos los sentidos son bajos y se implementa de una forma más rápida a corto plazo, y con la segunda pasa justamente lo contrario. Por tanto, la major manera de sacar el máximo rendimiento a los microservicios es en grandes aplicaciones, ya que todo esfuerzo y coste realizado desde el principio va a permitir traducirlo directamente en beneficios en un futuro.

Palabras clave— microservicios, endpoint, servicio, modulo, monolítico, arquitectura.

Abstract— The next article presents the process followed for the construction of a booking hotel system made with the microservices architecture. The final goal is to achieve a well-implemented system that will allow us to compare its physiognomies with the ones that use to have a simple monolithic system. To conclude, the final result of this project determines that any structure is useful; thus, both will depend on the necessity of the customer. On the one hand, if we are asked for a small application without growing perspective and any modifications, the best option is a monolithic architecture. On the other hand, if we forecast to add more features and updates with other adaptations, we are choosing the microservices structure.The main reason is about general costs. While we have low cost and faster implementation in the first architecture, we have different properties in the second structure. All in all, the best way to achieve the maximum profit by using microservices is in big applications due to the high future earnings that we will get after the effort expended.

Keywords— microservices, enpoint, service, module, monolithic, architecture.

1 INTRODUCCIÓN

La informática comenzó en enero del año 1975 con la primera computadora personal llamada Altair 8800[1]. A día de hoy, los ordenadores se han convertido en equipos modulares, constituidos por diferentes componentes, que permiten a una parte de usuarios desarrollar con ellos aplicaciones y programas y a la otra usar dichas aplicaciones para un uso personal, corporativo o estudiantil.

Este marco evoluciona continuamente permitiendo la creatividad de muchos desarrolladores para aplicar soluciones a ciertas necesidades de negocio o de interés común. Una de los puntos más importantes del desarrollo es la arquitectura y los patrones para un funcionamiento y rendimiento óptimos [2].

Un taller de arquitectos de Venecia en mayo de 2011, usó el término “microservicio” a una arquitectura que habían explorado y que posteriormente decidieron llamar “microservicios” ya que era, a su parecer, lo más apropiado. James Lewis y Fred George presentaron algunas ideas en un estudio de caso, pero fue Adrian Cockcroft (en Netflix) el pionero en el estilo a escala web junto con Joe Walnes, Dan North, Evan Bottcher y Graham Tackley [3]. El Dr. Peter Rodegers introdujo el término “Micro-Web-Services” en la conferencia Web Services Edge en 2005 [3].

Esta arquitectura consiste en fragmentar la aplicación o sistema en diferentes módulos, llamados servicios. Dichos servicios actúan de forma independiente unos con otros y se nutren entre ellos en caso de necesitar-

- *E-mail de contaco: marc.espinosai@e-campus.uab.cat.*
- *Mención realizada: Enginyeria del Software.*
- *Trabajo tutorizado por: Katerine Diaz (Computer Science)*
- *Curso 2018/2019*

lo. Por el contrario, tenemos la arquitectura monolítica donde todo el proyecto contiene todo, cosa que provoca que los cambios afecten a la mayoría de los elementos.
Este proyecto pretende implementar un sistema de reserva de hoteles haciendo uso de una arquitectura en microservicios con las mejores prácticas, para poder ejemplificar de manera real como son los sistemas con esta estructura. La finalidad de todo esto es determinar si hacer uso de esta estructura es en todos los casos la mejor opción y conseguir un sistema versátil, ya que puede estar hecho en diferentes lenguajes (pese a que únicamente se haga en Java), que sea más fácilmente escalable en comparación con una aplicación monolítica y que su mantenimiento sea menos costoso, ya que las modificaciones no tienen por qué afectar.

1.1 Definiciones, acrónimos y abreviaciones

Esta sección presenta un listado de definiciones, acrónimos y abreviaciones que se utilizan a lo largo del documento:

1. **Microservicios**: unidades funcionales concretas e independientes, que trabajan juntas para ofrecer la funcionalidad general de una aplicación [4].
2. **AWS**: es un conjunto de herramientas y servicios de cloud computing de Amazon [5].
3. **Escalabilidad**: Aumento de la capacidad de trabajo sin comprometer el funcionamiento y la calidad del sistema [6].
4. **Elasticidad**: Es la capacidad de ampliar o reducir rápidamente los recursos informáticos de procesamiento, memoria y almacenamiento para satisfacer demandas variables sin tener que preocuparse por planear y preparar la capacidad para períodos de uso máximo [7].
5. **Endpoint:** El punto final del servicio web describe el punto de contacto para un servicio [8].

2 Estado del arte

Existen grandes empresas que ofrecen una gran cantidad de servicios y presentan una arquitectura monolítica. Esto supone un problema a la hora de escalar, ya que es muy costoso porque se tiene que hacer como un todo. Es por eso, que compañías como Netflix, que dispone de una gran cantidad de recursos, ha migrado sus sistemas a microservicios en la nube con AWS. El proceso ha tardado 7 años, pero las ventajas obtenidas han sido los siguientes [9]:

- Aumento de la disponibilidad a nivel global.
- Gran elasticidad que ha permitido añadir servidores virtuales y petabytes de espacio de almacenamiento en minutos.
- Incremento en la escalabilidad.
- Reducción de costes por streaming en la nube menores que en el centro de datos.
- Más fácilmente escalable debido al cambio de una aplicación monolítica a cientos de microservicios.

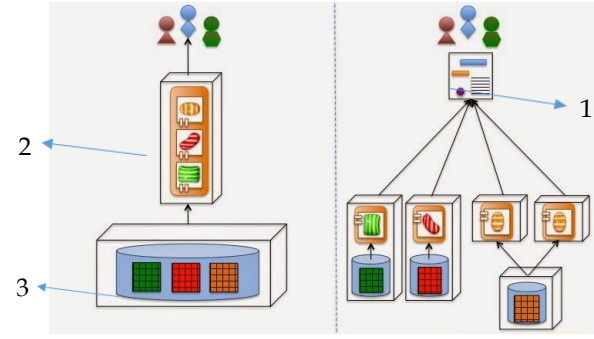


Figura 1: Sistema monolítico vs sistema de microservicios [2]

En la *figura 1*, la imagen de la izquierda (estructura monolítica) y derecha (arquitectura en microservicios) están divididas en tres partes:

1. Interfaz del usuario.
2. Funcionalidades de la aplicación. Se encarga de realizar las acciones que hace el usuario a partir de la interfaz y conecta con la base de datos en caso de necesitarlo.
3. Base de datos. Sistema de almacenamiento de datos.

3 Objetivos

3.1 Objetivo general

El objetivo general de este proyecto es crear un sistema de reserva de hoteles con una arquitectura de microservicios. El sistema estará formados por tres modulos que serán proyectos independientes: *HotelsBooking*, *Login* y *Ratings*. Tendrá test unitarios y una interfaz con Swagger para poder probar los diferentes servicios implementados.

3.2 Objetivos específicos

Los objetivos específicos del proyecto son los siguientes, ordenados por orden de prioridad descendiente:

1. Crear el sistema en dos meses.
2. Generar el sistema separado en módulos.
3. Implementar servicios independientes en cada módulo.
4. Conseguir un sistema más fácilmente escalable que una aplicación monolítica.
5. Crear una aplicación más limpia y fácil de mantener.
6. Implementar test en los servicios.
7. Añadir la interfaz Swagger para interactuar con los servicios.

4 FASES DEL PROYECTO

4.1 PLANIFICACIÓN

En la *figura 2* se muestran el cronograma de la realización de este proyecto:

Fases	Febrero	Marzo	Abril	Mayo	Junio	Julio
Reunión inicial	X					
Preparar informe inicial	X X					
Entrega Informe Inicial		X				
Diseño de la base de datos		X X				
Implementación de booking		X X				
Entrega informe de progreso I			X			
Implementación de booking			X X			
Implementación del resto de módulos				X X X		
Test			X	X X X		
Documentación			X X	X X X		
Entrega informe de progreso II					X	
Preparar demo					X X	
Propuesta informe final					X	
Propuesta de presentación						X
Entrega del dossier						X
Entrega del póster						X
Defensa del TFG						X

Figura 2: Cronograma del proyecto

4.2 FASES

Durante el desarrollo del proyecto se puede observar las siguientes fases:

1. **Diseño de la base de datos:** Básicamente se realizará un diagrama UML con las clases y atributos necesarios para el buen funcionamiento de la aplicación.
2. **Implementación de la base de datos:** Con el diagrama UML se tendrán que crear los scripts SQL para que se ejecuten al poner en marcha la aplicación.
3. **Implementación del módulo "*Booking*":** Se implementará este módulo con las funcionalidades básicas de reservar, cancelar, modificar y eliminar reserva.
4. **Implementación del módulo "*Login*":** Se implementará el módulo de login de un usuario.
5. **Implementación del módulo "*Ratings*":** Valoraciones que hará el usuario una vez haya acabado su reserva.
6. **Test unitarios:** Se realizarán los test unitarios de cada módulo independiente.
7. **Swagger UI:** Se implementará y probará el funcionamiento de la aplicación con Swagger UI que permite testear los enpoints y ver el output de cada uno de ellos.

En la *figura 3* se muestra la apliación resultante una vez realizadas todas las fases:

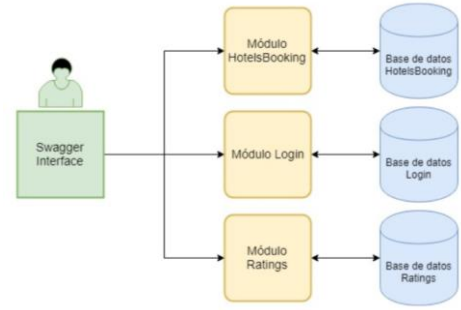


Figura 3: Estructura de la aplicación

5 REQUISITOS

5.1 Requisitos funcionales

En la siguiente tabla se muestran los requisitos funcionales:

ID	Requisitos funcionales	Prioridad
RF1	Un usuario tiene que poder registrarse	Baja
RF2	El usuario debe poder hacer una reserva.	Alta
RF3	El usuario debe poder cancelar una reserva.	Alta
RF4	El usuario debe poder modificar una reserva.	Media
RF5	El usuario debe poder buscar hoteles aplicando algún filtro.	Media
RF6	7. Se deber crear una base de datos particular por cada módulo implementado.	Alta
RF7	8. Se debe implementar test	Baja

5.2 Requisitos no funcionales

Los requisitos no funcionales son:

ID	Requisitos no funcionales	Prioridad
RF1	Los endpoints deben poder ser probados con la API de Swagger.	Alta
RF2	Cada módulo debe ser independiente uno del otro	Alta
RF3	El sistema debe comunacarse con una base de datos MySQL	Media
RF4	La aplicación debe funcionar en Windows	Alta
RF5	Las peticiones de hagan en menos de 1 segundo	Baja
RF6	Los módulos deben estar programados en Java	Media
RF7	En caso de que un módulo caiga, el resto no debe de dejar de funcionar	Media

6 CONFIGURACIÓN

La configuración de cada uno de los módulos se define en el archivo *pom.xml*. Cada proyecto Maven se configura añadiendo las dependencias con las diferentes API que se van a utilizar. En primer lugar, la configuración se ha realizado con "*https://start.spring.io/*", que es un generador rápido de proyectos de Spring donde se pueden elegir las dependencias (según las que se necesiten). Los tres diseños contienen las siguientes dependencias:

- **mysql-connector-java**: Conectar la aplicación con la base de datos MySQL.
- **junit-dep**: Se utiliza para realizar pruebas unitarias en Java**.**
- **spring-boot-starter-tomcat**: Proporciona valores por defecto útiles de Maven. También ofrece una sección de administración de depndencias.
- **spring-boot-starter-test**: Utilizado para probar aplicaciones Spring Boot con bibliotecas como JUnit, Hamcrest y Mockito.
- **spring-boot-devtools**: Ofrece características adicionales de tiempo de desarrollo.
- **spring-boot-starter-web**: Utiliza Tomcat como el contenedor embebido predeterminado.
- **Modelmapper:** Facilita el mapeo de objetos.
- **spring-boot-starter-data-jpa**: Facilita la implementación fácil de repositorios basados en JPA.

```
# Server
server.port = 8090

# Application
spring.application.name = HotelsBooking

# Datasource
spring.datasource.url=jdbc:mysql://localhost:3306/hotelsbooking?
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.

# JPA
spring.jpa.database = MYSQL
spring.jpa.show-sql = true
```

Figura 4: Archivo properties del proyecto HotelsBooking

A continuación en la *figura 4* se puede visualizar la configuración del archivo *application.properties* que contiene cada módulo del sistema. En este archivo se configuran la diferentes *properties* con los valores específicos:

En el apartado "*Server*" comentado en el código de la *ilustración 4*, se especifica el puerto en el que Spring levantará el proyecto con la *property*: "*server.port"*. En "*Applications*" se define el nombre de la aplicación y en "*Datasource*" se establecen los campos necesarios para realizar la conexión a la base de datos. Por último, en "*JPA*" se define el tipo de base de datos y "*spring.jpa.show-sql = true*" que permite ver las consultas SQL generadas automáticamente y el orden que se ejecutan con fines de depuración.

5 BASE DE DATOS

Las bases de datos MySQL se han realizado junto con los scripts respectivos. Los diseños son lo siguientes:

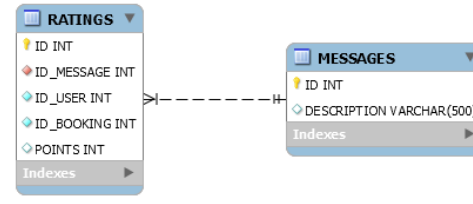


Figura 5: diagrama de la base de datos ratings

En la *figura* 5, se muestra el modelado UML de la base de datos de "*Ratings*", que contiene toda la información relacionada con las valoraciones de una reserva. Las entidades son las siguientes:

- **Ratings**: Corresponde a las valoraciones del usuario sobre una reserva una vez realizada (la puntuación es de 1-5). Los atributos son un mensaje, el usuario que realiza la puntuación, la reserva sobre la que se hace y los puntos.
- **Messages**: Entidad que contiene el mensaje de una valoración (puede haber o no).

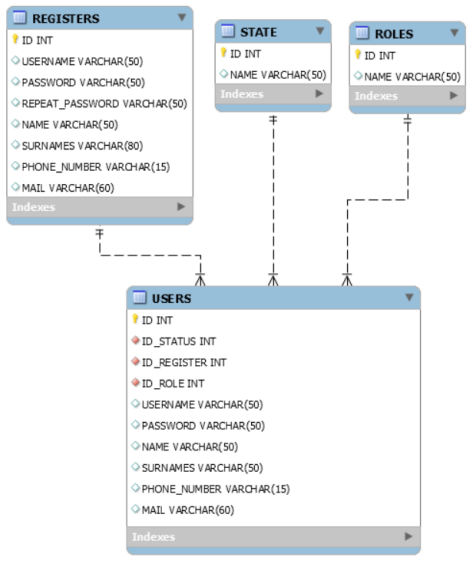


Figura 6: diagrama de la base de datos login

En la *figura 6* se muestra el modelado UML de la base de datos de "*Login*", que contiene toda la información relacionada con el usuario. En este diseño tenemos las siguientes entidades:

- **Users**: Es la tabla principal de esta base de datos. Contiene los datos que necesitamos del usuario como el estado (activado o deshabilitado), su registro asociado, el rol, nombre de usuario, contraseña, nombre, apellidos, número de teléfono y correo.

- **Registers**: Corresponde al registro de un usuario. En esta entidad guardaremos los campos de nombre de usuario, la contraseña, la contraseña repetida, el nombre, apellidos, número de teléfono y correo.

- **Roles**: Corresponde al rol que tendrá el usuario. En el caso de esta aplicación no habrá distinción de rol pero está hecha pensando en un futuro.

- **State**: Hace referencia al estado en el que se encuentra un usuario. Si está habilitado o deshabilitado. En esta tabla guardamos el nombre y la descripción de cada uno de los estados.

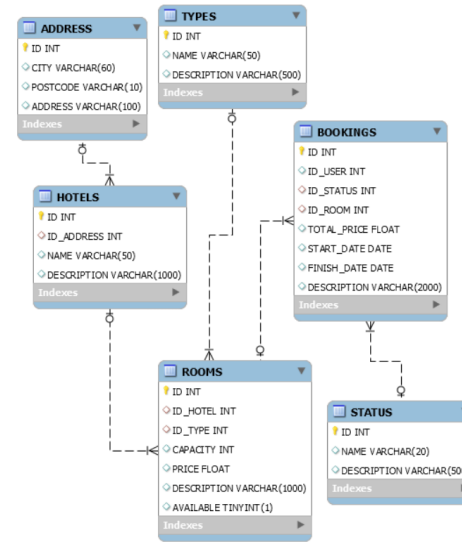


Figura 7: diagrama de la base de datos hotelsbooking

Por último, en la *figura 7*, se muestra el modelado UML de la base de datos de "*HotelsBooking*", que contiene toda la información relacionada con las reservas:
Como se puede observar en este diseño tenemos las siguientes entidades:

- **Rooms**: Entidad que define una habitación, a la cual le corresponden atributos como capacidad, precio, descripción y disponible (valor booleano) que determinara si la habitación está disponible o no. Además, tenemos como claves foráneas la id del hotel y tipo de habitación (normal, especial…).

- **Bookings**: Corresponde a una reserva y contiene los datos como precio total, fecha de inicio y fin, estado y descripción. Además también se guarda como clave foránea el id del usuario (vendrá dado por el módulo de login), estado de la reserva y la habitación a la que corresponde.

- **Hotels**: Entidad que define un hotel. Está formada por un nombre una dirección y una dirección como clave foránea.

- **Address**: Entidad que corresponde a una dirección (los hoteles estarán distribuidos en distintos sitios). Los atributos que la forman son la ciudad, el código postal y una dirección.

- **Type**: Entidad que define el tipo de habitación que es. Formado por el nombre y una descripción.

- **Status**: Define el estado en el que está la reserva. Guarda el nombre del estado (Reservado, cancelado y disponible) y la descripción.

6 MAPEO

El mapeo de las con Hibernate, una solución para aplicaciones Java que permite realizar el mapeo de objetos sobre una base de datos relacional mediante anotaciones [10]. Para realizar este proceso, se especifican los campos (para la obtención de datos) junto con las relaciones.

En las *figuras 8* y *9* se muestra el mapeo de las entidades "*Booking*" y "*Type*" como ejemplo, ya que se ha realizado con todas las entidades de las diferentes bases de datos:

```java
package com.project.tfg.HotelsBooking.model.entities;

import java.util.List;

@Entity
@Table (name = "TYPES")
public class Type {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column (name = "ID", unique = true, nullable = false)
    private int id;

    @Column (name = "NAME", length = 50)
    private String name;

    @Column (name = "DESCRIPTION", length = 500)
    private String description;

    @OneToMany (mappedBy = "type", cascade = CascadeType.ALL, fetch = Fet
    List <Room> rooms;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
```

Figura 8: Mapeo de la clase "Type"

```java
@Entity
@Table (name = "BOOKINGS")
public class Booking {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column (name = "ID", unique = true, nullable = false)
    private int id;

    @Column (name = "TOTAL_PRICE")
    private float totalPrice;

    @Column (name = "START_DATE")
    private Date startedDate;

    @Column (name = "FINISH_DATE")
    private Date finishDate;

    @Column (name = "STATE")
    private int state;

    @Column (name = "DESCRIPTION", length = 2000)
    private String description;

    @Column (name = "ID_USER")
    int idUser;

    @ManyToOne (fetch = FetchType.LAZY)
    @JoinColumn (name = "ID_ROOM")
    private Room room;

    @ManyToOne (fetch = FetchType.LAZY)
    @JoinColumn (name = "ID_STATUS")
    private Status status;

    public int getId() {
        return id;
    }
```

Figura 9: Mapeo de la clase "Booking"

Por otro lado, se han creado los repositorios de cada clase con JPA, que será la manera de hacer las queries a base de datos. JPA ya establece unos patrones predeterminados que podemos ver en su documentación de cómo obtener los datos. En caso de querer hacer una query muy concreta, también se pueden realizar unas específicas.

En las *figuras 10* y *11* se pueden observar las clases de tipo interfaz que conectan la clase Java con la entidad MySQL. De esta manera se realizada la conexión para la obtención de datos de la base de datos:

```java
package com.project.tfg.HotelsBooking.repository;

import java.util.List;

@Repository
public interface BookingRepository extends JpaRepository <Booking, Integer>{

    List <Booking> findByRoom (Room room);
    Booking findByRoomAndStartDateBetween(Room room, Date startDate, Date finishDat
    Booking findById (int id);

}
```

Figuras 10: Repositorio JPA de la clase "Booking"

```java
package com.project.tfg.HotelsBooking.repository;

import java.util.List;

@Repository
public interface RoomRepository extends JpaRepository <Room, Intege

    Room findById (int id);
    List <Room> findByAvailable (int available);
}
```

Figuras 11: Repositorio JPA de la clase "Room"

Siguiendo la documentación de JPA, se han creado las queries necesarias para cumplir con el funcionamiento correcto de los servicios, Aquí se puede ver el ejemplo de "***BookingRepository***":

- ***findByRoom***: Se obtiene un listado de reservas que se han hecho de una habitación.

- ***findByRoomAndStartDateBetween***: Se obtiene una reserva específica de una habitación con una fecha inicio y fin establecida.

- ***findById***: Se obtiene una reserva a partir de un id.

7 SERVICIOS E IMPLEMENTACIONES

Se han creado las interfaces de los servicios junto con las implementaciones. En la *figura 12* se puede ver la declaración de los servicios y en la *figura 13* la implementación de los mismos.

```
package com.project.tfg.HotelsBooking.service;

import java.util.Date;

public interface BookingService {

    public BookingRestOut book (BookingRestIn bookingRest) throws HotelsBookingExcep
    public BookingRestOut cancelBook (BookingRestIn bookingRest) throws HotelsBookin
    public BookingRestOut cancelBook (int id) throws HotelsBookingExceptions;
    public BookingRestOut modifyBook (BookingRestIn bookingRest) throws HotelsBookin
    public void getAvailableRooms (String hotel, Date date) throws HotelsBookingExce

}
```

Figura 12: servicios de BookingService

```
    @Override
    public BookingRestOut modifyBook (BookingRestIn bookingRest) throws HotelsBook

        Booking booking = this.bookingRepository.findById(bookingRest.getId());

        if (booking != null) {

            booking.setStartDate(bookingRest.getStartDate());
            booking.setFinishDate(bookingRest.getFinishDate());
            Room room = roomService.getRoomById(booking.getRoom().getId());
            booking.setTotalPrice(countDays(bookingRest)*room.getPrice());
            this.bookingRepository.save(booking);

        } else {
            throw new HotelsBookingExceptions(ExceptionConstants.ERROR + Exception
        }

        return modelMapper.map(booking, BookingRestOut.class);

    }
```

Ilustración 13: Implementación del servicio "modifyBook" de "BookingService"

En la implementación del servicio "book" se usa un objeto ***ModelMapper*** que se encarga de cargar las entidades que provienen de base de datos a un objeto (también se puede hacer viceversa), siempre y cuando las variables de este objeto tengan el mismo nombre que el nombre de la variable en la entidad. De esta manera se ahorra el uso de los *get* y los *set* de los objetos y el código queda más limpio.
En la arquitectura de microservicios este es el patrón a seguir a la hora de crear un servicio, primero se declaran en la interfaz y después se implementa con los métodos que se consideren necesarios.

En la *ilustración 11*, se hace uso de la anotación ***@Autowired***, la cual inyecta las clases y nos permite hacer uso de ellas:

- ***BookingRepository***. Es el repositorio correspondiente a este servicio y por eso se inyecta de forma directa (para poder hacer uso de los métodos necesarios para acceder a los datos de la base de dates).

- ***RoomService*** y ***StatusService***. Son los servicios de las clases *Status* y *Room.*

Cabe destacar, que cada servicio tiene su propio repositorio y es solo el propio servicio el que puede acceder a su repositorio. En caso de necesitar algún dato de otro repositorio, se creará un método en el otro servicio, ya que ese servicio en concreto si tendrá acceso al repositorio, para obtener ese dato y enviárselo al servicio que lo necesite. Este el motivo por el cual se ha realizado un ***@Autowired*** del servicio y no del repositorio.

8 CONTROLADORES

Una vez creado el servicio y la implementación, se crea el controlador que será el encargado de dar visibilidad a frontend sobre las funcionalidades o servicios que proporciona la API. En la *figura 14,* que se muestra a continuación se pueden observar los distintos controladores del servicio *BookingService*:

```
@Override
@ResponseStatus (HttpStatus.OK)
@PostMapping (path = "/cancelBook", produces = MediaType.APPLICATION_JSON_VALUE)
public HotelsBookingResponse<BookingRestOut> cancelBook(@RequestBody @Validated

    return new HotelsBookingResponse<>(CommonConstants.STATUS_OK_SUCCESS, Commo
            CommonConstants.MESSAGE_OK_BOOKING_CANCELLED, bookingService.cancelB
}

@Override
@ResponseStatus (HttpStatus.OK)
@PostMapping (path = "/cancelBookForId", produces = MediaType.APPLICATION_JSON_V
public HotelsBookingResponse<BookingRestOut> cancelBook(int id) throws HotelsBoo

    return new HotelsBookingResponse<>(CommonConstants.STATUS_OK_SUCCESS, Commo
            CommonConstants.MESSAGE_OK_BOOKING_CANCELLED, bookingService.cancelB
}

@Override
@ResponseStatus (HttpStatus.OK)
@PostMapping (path = "/modifyBook", produces = MediaType.APPLICATION_JSON_VALUE)
public HotelsBookingResponse<BookingRestOut> modifyBook(@RequestBody @Validated
    return new HotelsBookingResponse<>(CommonConstants.STATUS_OK_SUCCESS, Commo
            CommonConstants.MESSAGE_OK_BOOKING_MODIFIED, bookingService.modifyBo
}
```

Figura 14: Controladores del servicio "BookingService"

Siguiendo el patrón realizado con los servicios, (primero crear la interfaz y posteriormente la clase que los mplementa), se ha creado *BookingController* donde se encuentra la declaración de los controladores y *BookingControllerImpl* donde se puede ver el funcionamiento.

Como se puede observar en la imagen anterior, los controladores reciben objetos Rest (formato JSON), que contienen la información que necesitará el servicio para realizar la función para la que ha sido diseñado.

Para hacer el código de una forma limpia se puede

observar en la imagen anterior que todos los métodos devuelven un *HotelsBookingResponse.* En la *figura 15* se muestra su contenido:

```
public class HotelsBookingResponse<T> implements Serializable {

    private static final long serialVersionUID = 8745142610645755815L;

    private String status;
    private String code;
    private String message;
    private T data;

    public HotelsBookingResponse(String status, String code, String mes
        super();
        this.status = status;
        this.code = code;
        this.message = message;
        this.data = data;
    }

    public String getStatus() {
        return status;
    }
    public void setStatus(String status) {
        this.status = status;
    }
    public String getCode() {
        return code;
    }
    public void setCode(String code) {
        this.code = code;
    }
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
```

Figura 15: Clase genérica HotelsBookingResponse "BookingService"

En la imagen se pueden observar las variables ***status***, ***code*** y ***message*** que contendrán diferentes valores, según si se ha ejecutado el servicio de manera correcta o ha habido algún inconveniente. La variable ***data*** contiene la información de backend que se quiere transmitir. Está hecho para que se le pueda pasar cualquier objeto, de esta manera podemos devolver diferentes Rest según el servicio. El servicio de *Booking* devolverá un *BookingRest*, el de *Status* un *StatusRest*… De esta manera se evita tener que declarar una clase para cada servicio y tener código redundante.

8 SWAGGER

8.1 Configuración

En la *figura 16* se visualiza la configuración necesaria para crear los endpoints haciendo uso de la API de Swagger:

```
@Configuration
@EnableSwagger2
public class SwaggerConfig {

    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
                .groupName("HotelsBooking").apiInfo(apiInfo())
                .select()
                .apis(RequestHandlerSelectors.any())
                .paths(regex(".*" + RestConstants.APPLICATION_NAME + "/.*"))
                .build();
    }

    private ApiInfo apiInfo() {
        return new ApiInfoBuilder()
                .title("HotelsBooking")
                .description("Hotels Booking Application")
                .version("v1")
                .build();
    }
}
```

Figura 16: Contenido de la clase SwaggerConfig "BookingService"

Al estar situada la clase "*HotelsBookingApplication"* fuera de cualquier paquete, cuando se ejecuta el sistema, se escanean todos los paquetes y sus contenidos. Con la anotación de ***@Configuration*** se le está comunicando a Spring que esta es una clase de configuración.

8.2 Uso

Una vez configurado el swagger y los módulos están implementados, se ejecuta la aplicación, en la *figura 17* se muestra el resultado con los endpoints del módulo "HotelsBooking":

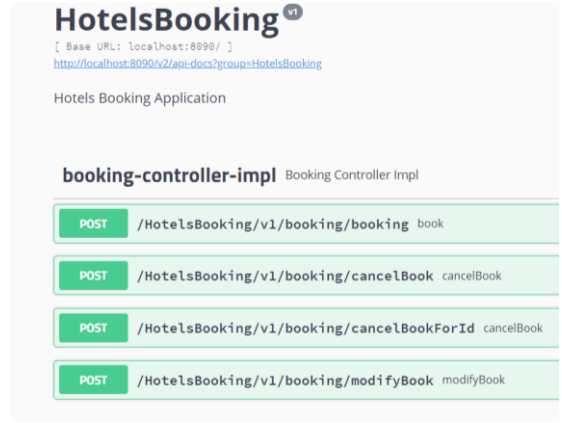


Figura 17: Interfaz Swagger del módulo "HotelsBooking"

Se pueden observar en la ilustración anterior los diferentes controladores que llaman a los respectivos servicios. Con esta interfaz, se pueden probar los servicios implementados y ver los resultados. En las siguientes imagenes, *figuras 18* y *19*, se hará una demostración realizando una reserva, la petición y la respuesta respectivamente:

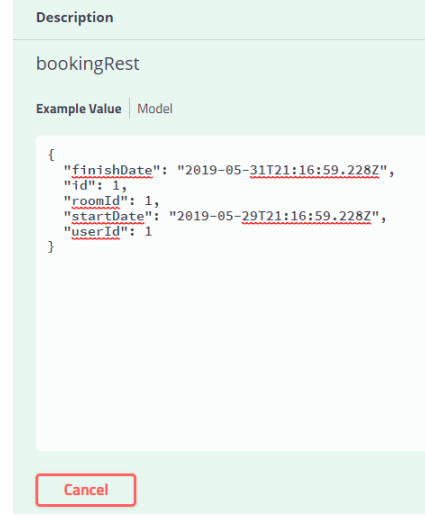


Figura 18: Petición de reserva

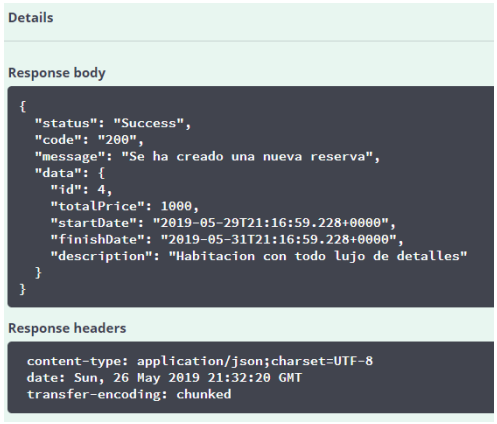
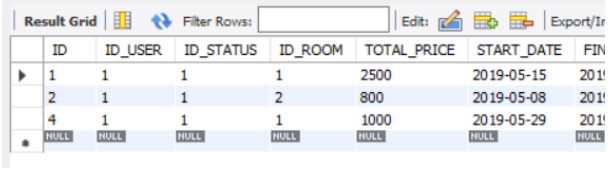


Figura 19: Respuesta de la petición de reserva

En este caso la reserva se ha realizado correctamente. Como se puede observar en la figura 19, se devuelve un código 200 y en la variable *data* del JSON la reserva generada junto con el precio que tiene. A continuación, se comprueba en la base de datos que realmente se haya creado un nuevo registro, en la *figura 20* se muestra el resultado:



ID	ID_USER	ID_STATUS	ID_ROOM	TOTAL_PRICE	START_DATE	FIN
1	1	1	1	2500	2019-05-15	201
2	1	1	2	800	2019-05-08	201
4	1	1	1	1000	2019-05-29	201
NULL	NULL	NULL	NULL	NULL	NULL	NULL

Figura 20: Registro de la reserva realizada

9 PROBLEMAS ENCONTRADOS

Los problemas más importantes que he encontrado han sido sobre todo a nivel de configuración. He estado varios días para configurar el Swagger debido a que no me funcionaba, el problema fue que el archivo de ejecución de la aplicación lo tenía dentro de un paquete, por tanto, al levantarse la aplicación y hacerse el escaneado de paquetes no lo detectaba. La solución fue ponerlo fuera (que es como se tiene que hacer siempre). También he descubierto otra posibilidad, Spring tiene una anotación (***@Configuration***) que permite especificar el path del paquete donde se encuentra los archivos de configuración. Esta segunda alternativa no es tan buena como la primera ya que se carga la cabecera del main, ya que de igual manera se especifica dónde está el paquete de configuración se pueden concretar muchos más paths. De esta forma haría que se quedase un código muy cargado cuando Spring lo puede hacer automáticamente.

Otros problemas relacionados con la configuración han sido las anotaciones. Alguna vez me he dejado alguna, pero gracias a los Loggs que salen por consola cuando se levanta la aplicación he conseguido detectar el lugar.

A la hora de hacer el mapeo de la entidades, he tenido errores en hacer alguna relación que he conseguido solventar mirando ejemplos en internet.

Otros problemas surgidos han sido las peticiones hechas a base de datos que según como construyera el método no me dejaba realizarlas. Mirando detalladamente la documentación de Hibernate y probando en una base de datos test encontré los patrones a seguir según la información que quisiera obtener en ese momento.

10 CONCLUSIONES

En este punto del trabajo podemos sacar diversas conclusiones de la arquitectura en microservicios.
La primera conclusión es un gran incremento en la dificultad tanto en el diseño de la solución como en la implementación en comparación con una aplicación monolítica. Los motivos son simples, al tener todo en distintos módulos y tantas capas, tiene que hacerse una sincronización correcta para que cada una reciba y envíe los datos que necesite en el formato correcto. Por el contrario, es necesario destacar que dicha complejidad una vez implementado se reduce. Esto es gracias a las separaciones hechas que permite crear todo lo que se necesite de manera muy simple, que permite ahorrar tiempo al no tener que buscar donde está todo (porque sigue un patrón) y el flujo de la aplicación.
En consecuencia de todo esto, la segunda conclusión es la facilidad para hacer modificaciones y la claridad del código. En el caso de haberse realizado este proyecto con una estructura monolítica, el resultado serían clases interminables con muchas llamadas entre ellas. El hilo de ejecución hubiese sido muy difícil de detectar, así como los errores.
Ante todas estas ventajas que ofrecen los microservicios, existe la pregunta de si es la mejor opción para cualquier aplicación. En mi opinión la respuesta sería negativa, ya que todo lo bueno que te puedan proporcionar los microservicios depende mucho de lo que se necesite, del futuro de la aplicación (si se expandirá) y del coste. En caso de necesitar una aplicación simple y que se conozca que con el tiempo no va a cambiar mucho, sin dudarlo optaría por una aplicación monolítica, ya que el tiempo, coste y esfuerzo que implica es mucho menor. En el caso opuesto de querer hacer una aplicación que con el paso del tiempo se vayan añadiendo diferentes funcionalidades y componentes, optaría por los microservicios, ya que el esfuerzo, tiempo y coste adicional en un principio se vería amortizado en el futuro.
Respecto a los objetivos establecidos para este proyecto se han conseguido cumplir casi todos. La aplicación se ha realizado en el periodo establecido, esta separado en módulos totalmente independientes cada único con sus servicios. En consecuencia, la aplica-

ción es mas fácilmente escalable ya que al ser cada proyecto independiente, se puede escalar según la demanda, es decir, los más usados escalaran más que los que no. La aplicación es más limpia que hubiese sido en una aplicación monolítica debido a la separación realizada de las distintas capas. Finalmente, han faltado los test unitarios pese a que se han probado los servicios.

AGRADECIMIENTOS

Me gustaría agradecer en primer lugar a Katerine Díaz, ya que ella ha sido la que me ha marcado entrega a entrega como hacer las cosas para obtener los mejores resultados. Por otro lado, agradecer al equipo en el que trabajo de la compañía Everis, los cuales me propusieron la idea de hacer una aplicación enfocada en esta tecnología y ampliar mis conocimientos.

REFERENCES

[1] Main Page. Wikipedia https://ca.wikipedia.org/wiki/Inform%C3%A0tica#1975 [Consulta: 05-02-2019]

[2] Blog. Altran. https://blog.altran.es/altran-smart-society/modulos-o-microservicios/ [Consulta: 22-12-2017]

[3] Blog. Daniel Pariente: http://blog.danielpariente.com/microservicios/ [Consulta: 22-05-2018]

[4] Blog. MdCloud. https://blog.mdcloud.es/que-son-los-microservicios-definicion-caracteristicas-y-retos/ [Consulta: 16-05-2018]

[5] tic.PORTAL. https://www.ticportal.es/temas/cloud-computing/amazon-web-services [Consulta: 2019]

[6] Aboutespanol. https://www.aboutespanol.com/que-es-escalabilidad-157635 [Consulta: 09-08-2016]

[7] Azure. Microsoft. https://azure.microsoft.com/es-es/overview/what-is-elastic-computing/ [Consulta: 01-02-2019]

[8] techlandia. Apps y Software: https://techlandia.com/definicion-del-servicio-web-endpoint-info_500146/ [Consulta: 2019]

[9] Netflix. Completada la migración de Netflix a la nube. https://media.netflix.com/es_es/company-blog/completing-the-netflix-cloud-migration [Consulta: 11-02-2016]

[10] Wikipedia. Hibernate. https://ca.wikipedia.org/wiki/Hibernate [Consulta: 23-02-2019]