PROYECTO 2020/2021 - SISTEMAS DISTRIBUIDOS.

SISTEMA DE AGENCIA DE VIAJES.

Autor: Luis Girón Juárez.

1. DISEÑO DE LA ARQUITECTURA CONCEPTUAL.

En este primer apartado, trataré de esclarecer y explicitar de una manera general, con cierto nivel de abstracción y sin llegar a incluir ningún tipo de tecnología, la concepción funcional del sistema.

1.1. CARACTERÍSTICAS PRINCIPALES.

Primero de todo, analizaré las diferentes características principales de un sistema distribuido con los requerimientos recogidos. Esto me servirá para luego ser capaz de escoger una arquitectura conceptual objetivamente y con mayor sencillez.

Características	Descripción
Heterogeneidad	En nuestro sistema es menester tratar la heterogeneidad, puesto que vamos a tener que comunicar diferentes sistemas. A saber, entre nuestro sistema de agencia de viajes y el resto de los proveedores y con los clientes que pueden usar diferente hardware, diferente tipo de red, diferente software, etc.
Extensibilidad	A su vez, necesitaremos que nuestro sistema sea capaz de extenderse, por ejemplo, en el caso en que queramos ampliar nuestros servicios software tanto a nivel de módulos en la programación, servicios a ofrecer aparte de los ya definidos o a nivel de aplicación como el de crear una aplicación móvil. Además, si el número de clientes que solicitan servicios aumenta, necesitaremos extender el sistema a nivel hardware con más procesadores, RAM, almacenamiento externo, etc.
Escalabilidad	Tomando la narrativa de la anterior característica, si aumenta el número de clientes que utilizan los servicios, necesitamos que dinámicamente se repartan correctamente nuestros recursos software y hardware en función de la situación actual. Si hay poca gente, restringiremos la capacidad a usar por cliente, liberando parte del sistema, pero si crece el flujo de clientes, habrá que escalar adecuadamente el sistema para proporcionar los servicios correctamente.
Seguridad	Tendremos que hacer frente a: ataques externos DoS; confidencialidad de los datos de los clientes; mantener la disponibilidad del servicio; mantener la integridad de las transferencias.
Concurrencia y Sincronización	Hemos de ofrecer una concurrencia a las visualizaciones de peticiones idénticas de clientes distintos, así como que accedan de manera exclusiva a las compras no permitiendo que dos usuarios compren la misma. También haremos las transacciones atómicamente para asegurar la compra. Así evitaremos inconsistencias en acciones críticas.
Tolerancia a fallos	Es posible que alguna parte del sistema falle en algún momento, pero no por ello tiene que dejar de funcionar todo el sistema. Tener esto en cuenta a la hora de implementar nuestro sistema mejorará la calidad del servicio (QoS). Si un servidor cae, hemos de traspasar la responsabilidad que tenía este para que la cumpla otro ininterrumpidamente.

Transparencia	De localización: Ha de ignorarse la ubicación geográfica del sistema tanto en vistas al cliente como al sistema mismo. Así, no importará dónde reside el sistema. De escalabilidad: El sistema escala para adaptarse a la demanda de servicios. De acceso: Acceder a elementos remotos como si fuera local es indispensable. El cliente ha de interactuar con facilidad a la hora de adquirir los productos necesarios.

1.2. ARQUITECTURA CONCEPTUAL DE FORMA GRÁFICA.

Con las características principales bien definidas, procedo ahora a generar una arquitectura conceptual de forma gráfica que, como primera iteración en la creación del sistema, nos proveerá una visión abstracta pero necesaria para poder abordar el proyecto correctamente.

Para ello, antes de nada, vamos a establecer diferentes puntos que van a intervenir en la arquitectura conceptual.

- **Actores relevantes del sistema:**

- Cliente.
- Administrador.
- Proveedor de coches.
- Proveedor de vuelos.
- Proveedor de hoteles.
- Banco

- **Funcionalidades e interfaces del sistema:**

  - ***Funcionalidades e interfaces de usuario (Front-End):***

- Registro: El cliente nuevo se registrará con los campos correspondientes. Después, si desea acceder inmediatamente a la página, tendrá que identificarse.
- Identificación: El cliente ya registrado se identificará con su cuenta para acceder a la página.
- Buscar ofertas: El cliente buscará las ofertas de lo que desee. Lo buscará por las fechas de inicio y fin.
- Seleccionar ofertas: Seleccionará las diferentes opciones que desee el cliente.
- Reservar: Reservará todo aquello que haya seleccionado. Así, ningún otro cliente podrá acceder a esas ofertas.
- Comprar Reservas: Se comprará todo aquello que haya sido reservado.

  - ***Funcionalidades y servicios del sistema (Back-End):***

- Tramitar registro del cliente: Se tramitará el registro del cliente guardando su información en la base de datos.

- Comprobar identificación del cliente: Accediendo a la base de datos, se comprobará si el cliente existe. Si existe, accederá a la página web. Si no, devolverá un error.
- Mostrar reservas: El cliente podrá seleccionar las reservas que le interesen a partir de la búsqueda.
- Buscar reservas: Se realizará una búsqueda de acuerdo con los parámetros de búsqueda del cliente. Para ello, contactaremos con los proveedores.
- Tramitar compra individual: Una vez seleccionadas la reserva, el cliente podrá tramitar el pedido.
- Tramitar compra conjunta: En caso de ser varias reservas, se tramitará todo en conjunto como una única compra.
- Gestionar pago de compra: En el trámite de la compra se efectuará la transacción de la compra dada su tarjeta de crédito. Se realizará con la entidad bancaría asociada al cliente.
- Conectar con proveedor de hotel: Conectará el sistema con el proveedor de hoteles.
- Conectar con proveedor de vehículo: Conectará el sistema con el proveedor de vehículos.
- Conectar con proveedor de vuelo: Conectará el sistema con el proveedor de vuelos.
- Excluir reserva: En caso de que más de un cliente haya seleccionado una reserva y uno de ellos haya tramitado la compra, el otro usuario será avisado de que esa oferta ha sido ya consumida.
- Ofrecer nueva oferta: En caso de una reserva excluida, se le ofrecerá una oferta nueva al cliente con los mismos parámetros.
- Aceptar transferencia: En la compra, se comprobará que la tarjeta de crédito/débito del cliente es válida para la transacción.
- Compra Interfaz B2B: Dada la entidad bancaria del cliente, se accederá a la interfaz de dicha entidad para garantizar el cobro de las reservas del usuario.
- Comprobar pago: Se comprobará si la tarjeta del cliente tiene suficiente dinero para el pago. En caso contrario, las selecciones serán anuladas junto con las acciones que haya implicado dichas reservas temporales en los proveedores.
- Anulación de las reservas: Se anularán las reservas del cliente que haya seleccionado. Esto sucederá en el caso de que no haya podido tramitar la compra.
- Anulación de reservas temporales en proveedor: Se anularán las reservas temporales en los proveedores que se especifique.
- Enviar ofertas: Envía el proveedor las ofertas al gestor de viajes.
- Comprobar oferta: Comprueba si la oferta seleccionada por el cliente ya ha sido reservada por otro anteriormente.

- Modelos arquitectónicos:

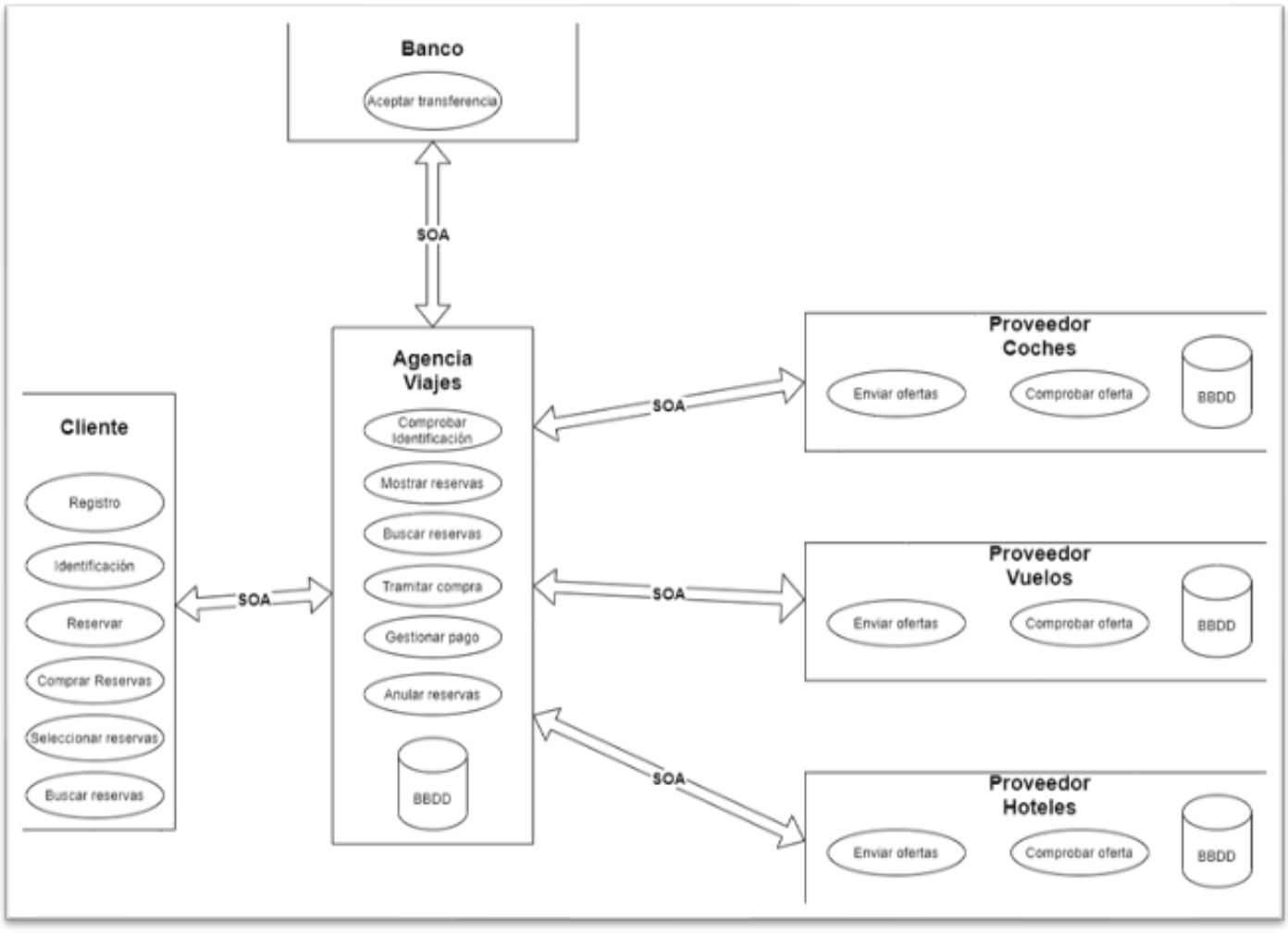
El modelo que voy a escoger para mi sistema es SOA (Service-Oriented Architecture). Es un sistema que no es tan solo una arquitectura, sino que más bien es una filosofía de concebir un sistema, con unos principios y características asociados. En concreto, usaré la arquitectura match-maker.

He escogido este modelo por las siguientes razones y principios que se tienen que cumplir:

- Con el principio de *localización, descubrimiento y publicación* nos servirá para comunicar las partes. Tendrá un intermediario que nos permitirá abstraernos de la publicación y la localización de algún servicio.

- Otro es la *interoperabilidad*, que nos permitirá conectarnos independientemente de la tecnología que use cada una de las partes.
- La *composición*, por otro lado, nos permitirá generar servicios de forma jerárquica uniendo varios servicios para crear uno que los una. Por ejemplo, si tengo los 3 servicios de reservar coche, hotel y vuelo, nos permitirá unir los 3 en 1 con gran facilidad.
- *Autonomía y autocontenidos*: Este principio lo que permite es mantener cada parte de manera independiente y si modifico alguna de las partes del sistema, no se ve afectado. Esto reduce el acoplamiento. Eso sí, cuanta más *composición* menor capacidad de *autonomía* y *autocontenido.*
- La *reusabilidad* es indispensable. Esto nos permitirá que los servicios sean genéricos, reutilizables. Que puedan usarse en diferentes procesos o funcionalidades.
- *Desacoplamiento:* Es la dependencia que hay entre los elementos, de tal manera que si conseguimos esa independencia (desacoplamiento), nos permitirá que al crear nuevos servicios y que se adapte fácilmente al sistema.
- *Contrato bien definido*: Es una estructura de datos que va a permitir al proveedor registrar el contrato y el consumidor poder adquirirlo y ver qué cosas tiene, cómo relacionarme con él, dónde se encuentra, etc.

Con todo, este es a grosso modo y gráficamente el sistema que vamos a realizar:



- Patrones de intercambio de mensajes:

Los patrones de intercambio de mensajes (MEP: Message Exchange Pattern) es una de las partes fundamentales a la hora de comunicar correctamente las diferentes partes del sistema.

En cuanto a la comunicación del *Cliente* con la *Agencia de Viajes*, he decidido que la comunicación sea mediante el patrón Request-response. Esto lo he decidido así ya que generalmente, el cliente va a solicitar(request) diferente información a la agencia que va a tener que mostrarle al cliente (response) y todo de manera asíncrona.

Por otro lado, la comunicación de la *Agencia de Viajes* y los *Proveedores* será mediante el Request-response, donde la agencia pedirá (request) a uno o varios proveedores las diferentes ofertas que tienen conforme a unos criterios bien definidos. Los proveedores entonces se encargarán de devolver las ofertas que cumplan con los criterios (response) de manera síncrona.

En cuanto a la última comunicación. A saber, entre el *Banco* y la *Agencia de viajes,* también será Request-response. Ya que lo que necesitamos aquí es una comunicación síncrona que permita a la agencia solicitar la transacción y al banco responder si la tarjeta del cliente es válida y tiene dinero suficiente para completar la compra.

- Los mensajes y su contenido:

- *Solicitud de registro*: El cliente envía a la agencia los siguientes datos para registrarse: nombre, apellidos, nombre de la cuenta, contraseña.
- *Respuesta de registro*: La agencia le manda un mensaje de aceptación del registro al cliente si todos los campos son correctos y uno de denegación si hay algún campo incorrecto.
- *Solicitud de Identificación*: El cliente, una vez registrado, se identifica y envía su nombre de cuenta y contraseña a la agencia para entrar a la web.
- *Respuesta de identificación*: La agencia acepta la identificación si existe tal cuenta y le deja acceder a la web con los servicios correspondientes. Si no existe la cuenta, le deniega el acceso.
- *Solicitud buscar oferta*: El cliente busca reservas de uno o varios servicios (hotel, vuelo, coche) introduciendo los siguientes datos: Fecha inicio y fin, origen y destino del viaje. Luego esto se lo envía a la agencia una vez rellenado.
- *Respuesta buscar oferta*: La agencia responde con todas las ofertas que tiene y se las muestra al cliente. La oferta contiene diferente información según cada servicio*.
- *Solicitud buscar oferta con proveedor*: La agencia, con los datos del cliente, se comunica con el proveedor para que le envíe las ofertas.
- *Respuesta buscar oferta con proveedor*: El proveedor, dada la información, filtra las ofertas compatibles y las envía a la agencia. Se guarda la información del cliente que ha pedido la oferta temporalmente.
- *Solicitud seleccionar ofertas:* El cliente selecciona las ofertas que le interesen.

- *Respuesta seleccionar ofertas:* La agencia muestra la selección hecha por el cliente. Además, le mostrará los datos a introducir para la compra.
- *Solicitud comprar ofertas:* El cliente introduce los siguientes datos para comprar la oferta: número de la tarjeta, nombre del titular, código de seguridad, domiciliación (país, ciudad, calle, número, piso).
- *Solicitud tramitar compra:* La agencia se pone en contacto con el proveedor para ver si alguna de las ofertas que quiere comprar el cliente ya está siendo observada por otro. Si se da el caso, la agencia le dirá al otro cliente que ya no está disponible.
- *Respuesta tramitar compra:* El proveedor se encarga de dar la información de los clientes que están observando la oferta.
- *Solicitud gestionar pago:* La agencia solicita la transacción con el banco dado los datos del cliente para la compra.
- *Respuesta gestionar pago:* El banco da el OK si es válida la tarjeta y si tiene dinero suficiente. En caso de rechazar el pago, la agencia devolverá al cliente el mensaje correspondiente con el error y lo llevará al escenario anterior de introducir los datos de la tarjeta.
- *Respuesta comprar oferta*: La agencia responde con un mensaje de que la compra ha sido efectuada, finalizando así la compra y lo llevará al menú principal de la página.

*Hotel → Nombre del hotel, lugar, precio por noche.

Coche → Empresa, marca del coche, número de asientos, precio por día, lugar de recogida.

Vuelo → Empresa, aeropuerto de salida y llegada, hora de salida y llegada, duración del vuelo, matrícula.

1.3. ARQUITECTURA TÉCNICA

En este apartado procedo a explicitar y justificar la arquitectura técnica que voy a usar en mi sistema.

Sobre los principios SOA comentados en el apartado anterior, lo implementaré mediante una arquitectura software. A saber, REST (REpresentational State Transfer). En concreto **RESTFul**. RESTFul usa HTTP de normal, empleando sus métodos básicos (GET, PUT, POST, DELETE, etc.).

Mientras que REST es el estilo arquitectónico, RESTFul se podría decir que son los servicios web que se ajustan al primero proporcionando interoperabilidad entre sistemas informáticos en Internet. Estos servicios web RESTFul permiten que los sistemas solicitantes accedan y manipulen representaciones textuales de recursos web mediante el uso de un conjunto uniforme y predefinido de operaciones sin estado.

¿Por qué elegir RESTFul con respecto a alguna otra arquitectura?

- Por su **escalabilidad**: Gracias al desacoplamiento o separación entre cliente y servidor, el producto puede escalar sin que presente muchas dificultades.
- Por su **flexibilidad y portabilidad**: Con el requisito de que los datos de cada una de las peticiones sean enviados de forma correcta, es posible realizar una migración de un servidor a otro o practicar cambios en la base de datos en todo momento. De esta forma el front y el back se pueden alojar en servidores diferentes, lo que supone una enorme ventaja de manejo.
- Por su **independencia:** Debido a la separación entre el cliente y el servidor, el protocolo facilita que los desarrollos de las diferentes partes de un proyecto se puedan dar de manera independiente.

Ventajas sobre SOAP:

RESTFul supera desventajas de SOAP como puede ser la necesidad de que el cliente conozca la semántica de las operaciones. RESTFul, por otro lado, con pocas operaciones permite manejar muchos recursos mientras que SOAP necesita muchas operaciones para ello. Entre las ventajas de RESTFul podemos encontrar las siguientes:

- Sencillez a la hora de construir y adaptar.
- Escaso consumo de recursos.
- Las instancias del proceso se crean de forma explícita.
- El cliente, a partir de la URI inicial, no requiere información de enrutamiento.

Para construir el sistema con RESTFul, vamos a apoyarnos con un lenguaje para describir la API RESTFul. A saber, **RAML** (RESTFul API Modeling Language). Este lenguaje proporciona toda la información necesaria para describir las API RESTFul o prácticamente RESTFul ya que es capaz de describir las API sin todas las descripciones RESTFul. Además, fomenta la reutilización, permite el descubrimiento y el intercambio de patrones y apunta a que surjan mejores prácticas.

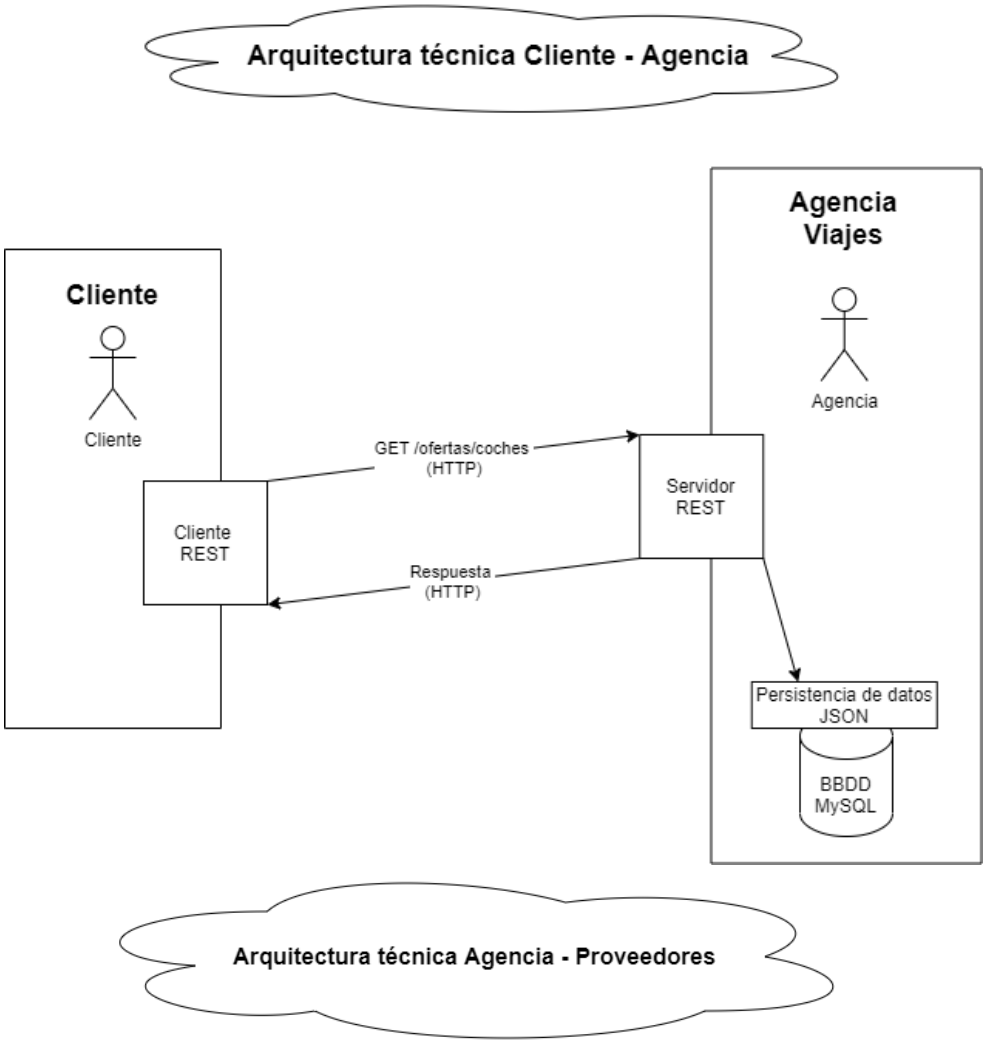
En cuanto al mecanismo de comunicación de texto que vamos a emplear será **JSON**. Hoy día, es el formato más usado mientras que XML está siendo menos empleado. Las razones por las que usar JSON son las siguientes:

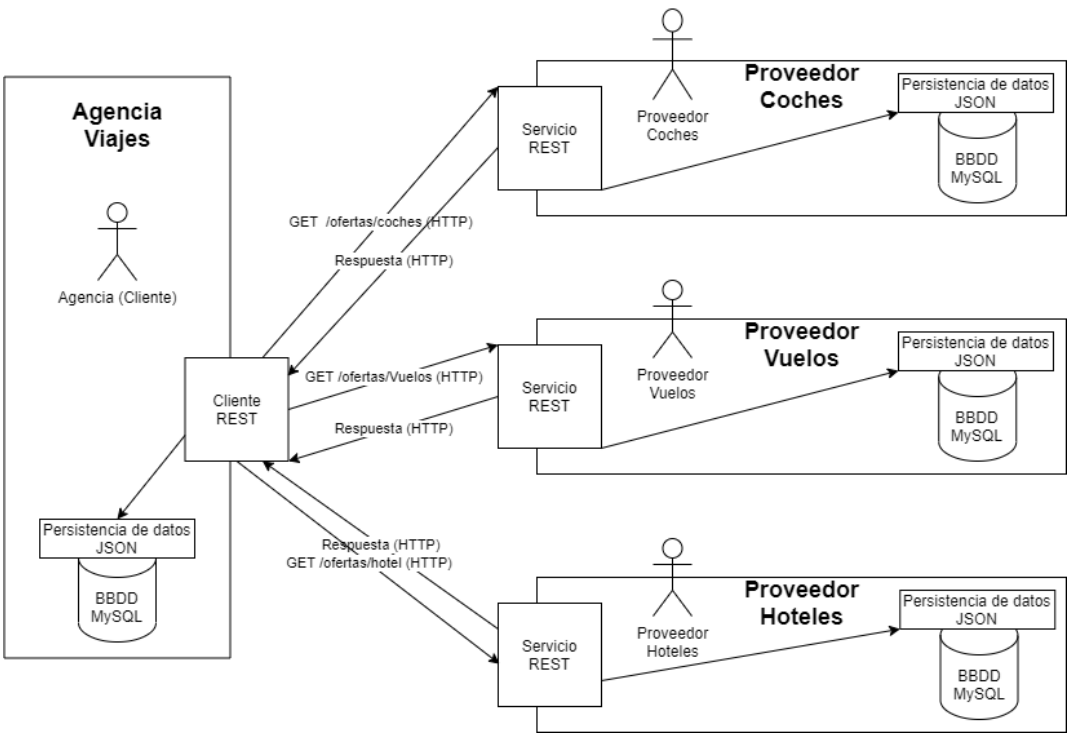
- Tiene un formato sumamente simple.
- La velocidad de procesamiento es alta.
- Los archivos generados son de un tamaño bajo en comparación con XML.

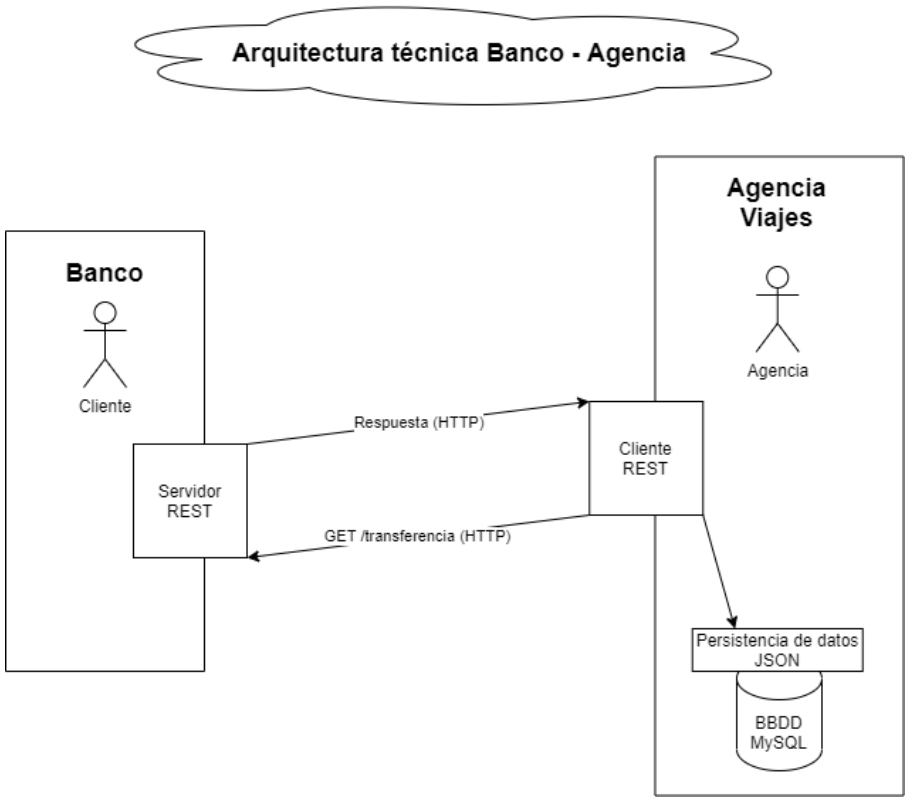
Las **tecnologías** usadas para realizar el proyecto han sido básicamente las expuestas en el famoso **Stack MEAN**: MongoDB para la BD; Express para el BackEnd, Angular para el FrontEnd, Node.js como entorno de ejecución.

La elección de estas tecnologías se basa en que todo funciona con un mismo lenguaje (JavaScript/TypeScript) y su sencillez asociada. Además, la BD guarda los objetos como JS.

- Arquitectura técnica gráficamente:







API'S

- Agencia:

https://documenter.getpostman.com/view/13214132/Tzm8GvrW

- Coches:

https://documenter.getpostman.com/view/13214132/Tzm8Gw5p

- Hoteles:

https://documenter.getpostman.com/view/13214132/Tzm8GwAA

- Vuelos:

https://documenter.getpostman.com/view/13214132/Tzm8GwAD

- Banco:

https://documenter.getpostman.com/view/13214132/Tzm8Gw1N

Para que la aplicación funcione, ya no solo correctamente, sino con unas medidas mínimas de seguridad, es necesario blindar a la aplicación con las medidas, que, a mi juicio, son necesarias. A saber:

- Cifrado del canal:
  Para que la información del sistema no viaje en texto plano, los servidores están implementado con SSL, de tal forma que cuando el cliente acceda a la aplicación, tendrá que realizar una petición con HTTPS. Esto es, HTTP seguro. Para ello, hemos generado un certificado SSL/TLS que nos protegerá, en general, de los hackers que escuchen el canal de transmisión de la información. Cabe mencionar, además, que SSL/TLS funciona, dentro del sistema de capas de la Red, por debajo de HTTP encargándose de cifrar toda la información.

- Cifrado de contraseñas:
  Otro punto que, a mi parecer es indispensable, es que las contraseñas se guarden cifradas. Para ello, lo que hago es que cuando un usuario se registre, antes de guardarlo en la BD, encripto su contraseña. Y cuando inicie sesión, se desencriptará y se comparará la contraseña con la de la BD anteriormente almacenada para ver si coinciden.

  Esto lo hago con una librería de JS llamada 'bcryptjs'. Con ella, primero de todo, genero la contraseña cifrada mediante SALT, que es una cadena de caracteres alfanuméricos pseudoaleatorios, producida tras una función hash. Con ella, lo que hacemos es encriptar la contraseña dada por el usuario. Cuando el usuario inicie sesión, la contraseña que indicó se comparará esta con la contraseña cifrada (e irreversible) gracias a una función de la librería mencionada.

- Autorización:

  Un aspecto relevante también es el de la autorización del usuario a ciertos subdominios de la API. En este sentido, he generado dos roles: user y admin. Para el primero he generado una función que nos permite comprobar que efectivamente el usuario es un usuario registrado. Para ello, utilizamos una autorización basada en el sistema de Bearer Token.

  Para implementarlo, he utilizado la librería 'jsonwebtoken'. Cuando un usuario se registra o inicia sesión, lo que hago es generar un token a partir de su ID y una palabra clave que he definido en el fichero config.js. Así, cuando se registre o inicie sesión, tendrá un token asociado, con duración de 1 día. Este token, le permitirá pasar ciertos middlewares creados como filtros de autorización. En donde verifico si el token pasado existe o no.

Para el admin, aparte de realizar la verificación del token, compruebo si tiene guardado en su BD, en un apartado de Roles, el id del rol de admin. En caso afirmativo, podrá pasar el middleware y acceder al servicio correspondiente.

- intercambio de recursos de origen cruzado (CORS)

  CORS es importante tenerlo en cuenta para nuestra aplicación por los siguientes motivos. Lo primero, porque nuestra agencia se comunica a su vez con otros servidores y, segundo, porque debemos de prohibir que el usuario sea capaz de acceder a ellos sin pasar por la agencia de una manera segura. Implementar CORS nos da la seguridad de que las peticiones al servidor son las que esperamos que el cliente realice y no haga peticiones maliciosas intentando modificar, crear o eliminar datos a placer. Para ello he utilizado la librería 'cors' de JS.

- Logs

  Los logs nos permiten saber qué acciones se están realizando en el servidor. En este sentido, es útil para la seguridad en el caso de que accedan a nuestro servidor. Porque si guardamos estos en un servidor aparte en un fichero, nos dará la información necesaria para saber por dónde se ha colado, qué daños ha realizado, a qué hora lo ha hecho, qué datos ha introducido, etc. Personalmente no he podido realizar hasta tal punto esta implementación, pero sí utilizo una librería que me da la información necesaria. Esta se llama 'morgan'.

- Otras medidas básicas de seguridad

  Además, he tratado de cubrir otros tipos de seguridad (11 en concreto) que cubre la librería 'helmet' por defecto. La mayoría relacionadas con las peticiones.

TRANSACCIONES

Por último, tenemos el módulo de las transacciones. Para ello, he estado barajando diferentes implementaciones. Principal y primeramente pensé en realizarlas mediante el famoso patrón SAGA orquestado.

Patrón SAGA:

El patrón SAGA se podría definir como una secuencia de transacciones locales, en la que, tras cada transacción realizada, se actualiza el tipo de evento que indicará la siguiente transacción local a realizar. En caso de que falle, se ejecutará una lógica de reversión de la transacción de forma que quede como si no se hubiera hecho nada.

Este patrón tiene 2 formas de implementarse. Uno es basado en *orquestación* y otro basado en *coreografía*.

El primero, el de *orquestación*, se podría definir como una transacción que está dirigida por un coordinador (director de orquesta) y se encargará de llamar a cada servicio (músicos de la orquesta) en su momento pertinente.

Las ventajas que ofrecen este tipo de implementación es que permite realizar un seguimiento de la transacción al estar centralizada la ejecución en el coordinador. Además de ello, la complejidad de la transacción en cuanto a su extensibilidad tan solo es lineal. También se evitan las dependencias cíclicas.

Por otro lado, tenemos la implementación SAGA basada en *coreografía*. Aquí los servicios a consumir por la transacción se comportan independientemente, aunque con una sincronicidad determinada (como los bailarines en una coreografía). Así, todos los servicios de la transacción escuchan por unos canales donde se enviarán diferentes eventos. De esta forma, cada servicio sabrá cuándo le toca ejecutarse y cuándo esperar.

Esta segunda implementación es sencilla de realizar cuando los servicios que intervienen son 2 o 3 a lo sumo. Pero su complejidad lógica es bastante elevada en el caso de que tengas que coordinar 10 servicios. Para el cual, será más conveniente usar el patrón mediante *orquestación*.

Mi implementación:

Personalmente, me hubiera gustado implementar el patrón SAGA basado en *orquestación,* pero buscando información, no he encontrado ninguna guía que me permitiese familiarizarme correctamente con la implementación de este. Y como no disponía de mucho tiempo busqué otras alternativas.

Vi, por ejemplo, que MongoDB (la BD que empleo) tiene un sistema de transacciones, pero no está diseñado para transacciones distribuidas.

Así, me decanté por realizar una implementación propia de lo que considero que una transacción requiere.

Una transacción normalmente se basa en satisfacer los principios **ACID**. Y he procurado guiarme con estos principios para implementarlo.

Mi implementación se fundamente en realizar una serie de peticiones de manera **atómica**. Es decir, que o se ejecutan todos o no se ejecuta ninguno. En el caso que una de las llamadas en la transacción por lo que fuera, se revertirán las acciones tomadas manualmente mediante otras llamadas que reviertan las acciones precedentes y posteriormente se mostrará el error.

También satisfago la **consistencia** de la transacción. Pues no existe ninguna acción que pase de un estado *válido* de la BD a otro *inválido*. Esto es así porque realizo una comprobación de que los datos son íntegros y coherentes.

En cuanto al **aislamiento**, no estoy tan seguro de poder afirmar que, en efecto, un par de llamadas concurrentes sobre los mismos datos no puedan generar colisiones inesperadas. Pero ciertamente me parece algo a lo que la amplitud de mi aplicación muy raramente vaya a ser sometida.

Por último, constatar que mi implementación sí cubre el último apartado de los requerimientos **ACID** que es el de la **durabilidad**. En efecto, cualquier transacción que se realice permanecerá almacenada en la BD.

Con todo, me gustaría resaltar que mi implementación está orientada a la implementación que sería realizar el patrón SAGA con *orquestación*. Pues en mi caso que el coordinador de la transacción es la agencia y es esta quien se ocupa de llamar en el momento adecuado a los diferentes servicios de la transacción y de manejar los errores para revertir la operación en caso de fallo.

Las transacciones están implementadas en agenciaController.js

FRONTEND

En cuanto al FrontEnd, como dije anteriormente creado con **Angular**, nos permite generar una GUI completamente desacoplada con el BackEnd.

La **funcionalidad** que incorpora es:

- SignIn,
- SignUp,
- formulario de consulta,
- reservar productos,
- carro para pasar al pago o para deshacer el pedido.
- Parte del admin que permite gestionar los recursos (admin/usuarios, admin/coches, admin/hoteles, admin/vuelos).

El proyecto contiene una separación de **componentes, servicios y modelos**. Además de que incorpora un sistema de seguridad que permite diferenciar los accesos a las páginas que el usuario puede hacer según el rol que incorpore (admin o user).

Otro aspecto por resaltar es que incorpora **Bootstrap** para una mejor implementación.

INFORMACIÓN SOBRE EL PROYECTO

- **GitHub**: https://github.com/lgj19/ProyectoSD
- **Puesta en marcha**: En el README del GitHub.