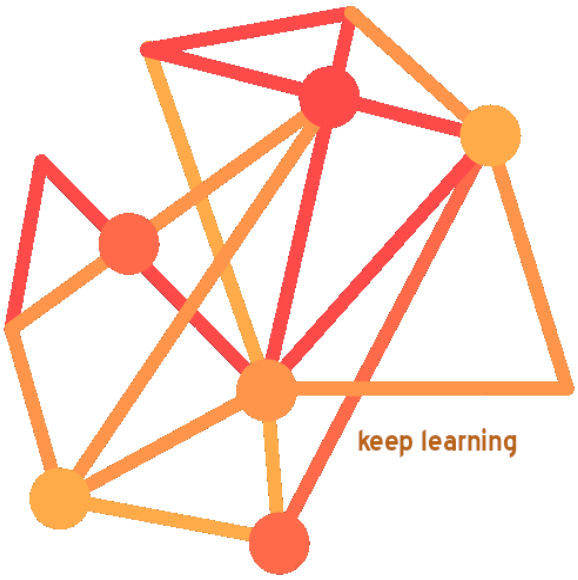
Apis complejas: Arquitectura Hexagonal. Websocket. MongoDB. GraphQL



Juan Carlos Pérez Rodríguez

Sumario

Arquitectura Hexagonal

Introducción

La arquitectura hexagonal es un patrón que está pensado para aplicaciones grandes, complejas. Su objetivo es separar en lo posible e independizar la lógica de nuestra aplicación de cualquier acceso a infraestructura e interacción con el usuario/cliente. Fue creada por Alistair Cockburn

Si pensamos en MVC o arquitectura por capas ya teníamos una aproximación a la idea de separación de nuestras clases de modelo del resto.

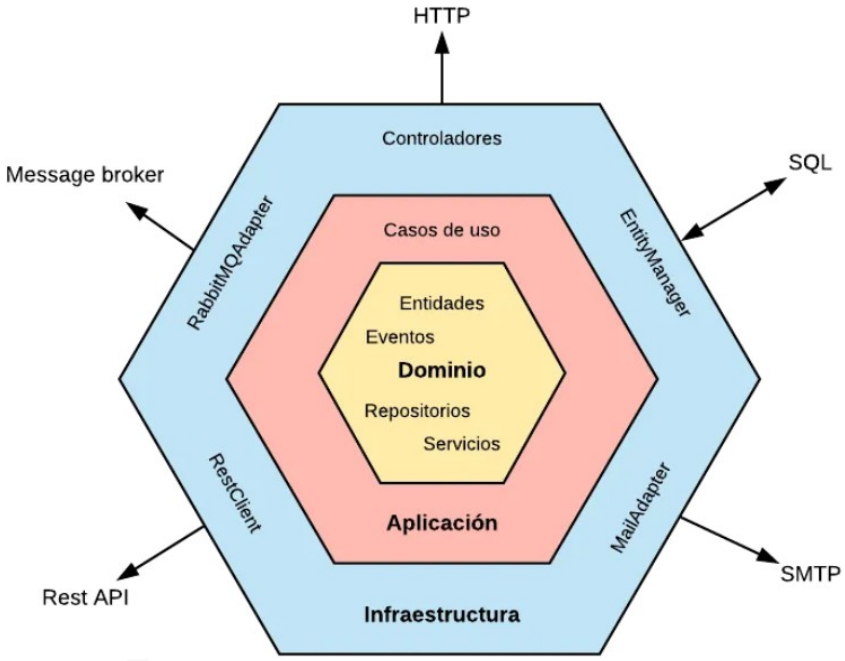
¿ Y por qué es necesario ? ¿ no es suficiente con las otras dos patrones que acabamos de nombrar ?

Si una aplicación es lo suficientemente compleja: acceso a varios SGBD distintos y/o ficheros, vemos que como poco se nos “mezclan” las anotaciones o configuraciones de nuestros objetos de modelo. Observar que si estamos guardando objetos en fichero, puede ser que tengamos que serializarlos y tener un uid de serialización. De alguna forma “manchando” con circunstancias de infraestructura nuestras clases de dominio. Veamos un ejemplo:

```
class DatoClima implements Serializable {
    private static final long serialVersionUID = 1L;


    private String ciudad;
    private String temperatura;


    @Transient
    private ClimaApi apiDeClima;

    public DatoClima(String ciudad, String temperatura, ClimaApi apiDeClima) {
        this.ciudad = ciudad;
        this.temperatura = temperatura;
        this.apiDeClima = apiDeClima;
    }
}
```

Observar que se quiere tener información de los datos del clima de una población. Para nuestro manejo necesitamos acceder a apis externas que tienen sus propias circunstancias (lo que aparece como ClimaApi), datos que no queremos en el fichero (@Transient) y por otro lado tenemos anotaciones como: serializable y el atributo: serialVersionUID que son necesarios por el acceso a la infraestructura.

Imaginemos otro caso en el que queramos tener unas clases que toman sus datos de una base de datos documental y otros datos de una base de datos relacional. En su relación entre ellas a nivel de clases de dominio NO debiera siquiera saberse en que sistema están almacenados y pudiendo intercambiar la infraestructura de nuestras clases de dominio en cualquier momento.

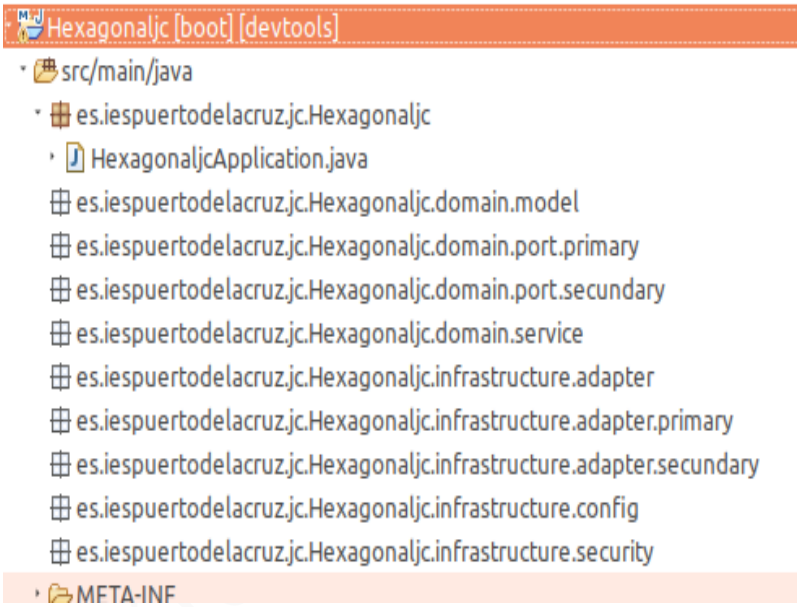


También de cara a lo que nosotros servimos a nuestros clientes hay implicaciones De tal forma que es deseable que las clases de dominio permanezcan igual y estén separadas de nuestros controladores ya sean para servir una api rest, graphql o cualquier otro tipo de servicio

Lo mejor sería que nuestras clases de dominio fueran plenamente agnósticas de nuestra infraestructura. La arquitectura hexagonal tiene esa intención. Se separa con interfaces las clases de dominio de la infraestructura

La arquitectura hexagonal tiene como gran defecto que implica aumentar la complejidad de nuestro proyecto, **no** siendo la mejor opción para proyectos pequeños

Estructura de carpetas



La estructura de carpetas aumenta considerablemente de tamaño.

El código principal de nuestra aplicación, sin efectos de la infraestructura de DDBB ni demás accesos, estará en el paquete: domain (en algunas organizaciones se llama application al mismo paquete)

Dentro de domain tendremos nuestras clases del modelo: domain. model

Así como los servicios (los casos de uso) que se dan con nuestras clases del modelo: domain.service

Finalmente definimos como nos comunicamos con el exterior (los interfaces) en la carpeta: domain.port

Esta separación mediante interfaces establecida en domain.port es la parte crucial de esta arquitectura. Ya que independiza de toda la infraestructura (los controller y los repositorios) a las clases de negocio

Es habitual, cuando trabajamos los casos de uso, hablar de actores primarios y secundarios. Siendo los primarios los que inician la acción y los secundarios los que son llamados

En nuestra arquitectura definiremos en: domain.port.primary la comunicación con los actores principales (endpoint rest, endpoint graphql,…) y en: domain.port.secundary la comunicación con los actores secundarios (bases de datos, etc) . En definitiva en domain.port van

los interfaces (tanto primarios como secundarios) que servirán para comunicar el modelo con el resto. Observar que llamamos: port (domain.port) a esas carpetas donde estarán los interfaces a los que se engancharán los servicios de la base de datos, los servicios que acceden a las apis etc

App de aprendizaje

Una buena forma de aprender es haciendo ejercicios. Vamos a realizar una api para jugar el tres en raya. Haremos varias versiones:

- 1. En una primera versión habrá un jugador contra la máquina, es estado de la partida quedará en una base de datos relacional y no nos preocuparemos de la seguridad (desactivaremos la seguridad en spring)

- 2. Posteriormente pasaremos esa api al modo de separación de paquetes hexagonal

- 3. Agregaremos la parte de usuario y autenticación. Donde la tabla de usuarios estará en relacional

- 4. cambiaremos la partida a una base de datos documental: mongo Y así notaremos lo cómodo e independiente que es la arquitectura hexagonal respecto a la infraestructura que se use

Para modelar la partida se propone:

Clase TresEnRaya con atributos:

Estado (un enum que en la primera versión tendrá únicamente: JUGANDO, FINALIZADA

simboloJug1, simboloJug2 ← son char: 'X' u 'O'

nickJug1, nickJug2 ← son string con el nick de cada jugador. En este versión se fija : nickJug2 a "ordenador"

turno ← una string que contendrá quién tiene el turno de juego (en esta versión inicial es un campo que no se utiliza para la partida ya que hay únicamente un jugador contra la máquina)

ganador ← una string que estará establecida a null toda la partida hasta que finalice y que contendrá el nick del ganador al final o la palabra: “empate”

escenario ← un array doble de char: char[3][3] para almacenar la situación de la partida. Inicialmente cada una de las posiciones del array se rellenan con guiones: ‘-’ y cada apuesta de los contrincantes irá sustituyendo el guión por su símbolo: ‘X’, ‘O’

Para almacenar en la base de datos, se propone una tabla: partidas (id INT,nickjug1 VAR_CHAR, simbolojug1 VAR_CHAR, nickjug2 VAR_CHAR, simbolojug2 VAR_CHAR, estado VAR_CHAR, tablero VAR_CHAR)

El tablero guardará el escenario de la partida (una string con nueve caracteres) . Así por ejemplo:

“O-OXXXX-O”

Es equivalente al escenario (cada tres caracteres es una fila):

O-O
XXX
X-O

que refleja una partida ganada por el jugador1

Del lado del controller (PartidaRESTController) tendremos la ruta base: /api/partidas y los siguientes endpoints:

Get /api/partidas/{id} ← devuelve el estado de la partida con el id dado (la partidadto correspondiente)

Post /api/partidas/{id}/apuestas ← agrega una nueva apuesta del jugador a la partida (La clase Apuesta es un DTO)

Post /api/partidas ← crea una nueva partida. En esta primera versión que no estamos usando una tabla de usuarios, Recibe el nombre del jugador

Get /api/partidas ← devuelve todas las partidas

Práctica 1: Realizar la api descrita con el modelo tradicional por capas

En arquitectura hexagonal se tiene en cuenta el concepto de actores primarios y secundarios de los casos de uso. Así entendemos que los actores primarios (los que inician la acción) serían nuestros REST controller o Graphql controller. Los secundarios serían los demás: todos los accesos a bases de datos (las entities los repositorios, etc) Y las clases de dominio SON INDEPENDIENTES DE TODA LA INFRAESTRUCTURA. Una forma de ver claro que nuestra separación es correcta es fijarse en los import Al finalizar nuestra api veremos que no hay ningún import en las clases que están en el dominio que no sean otras clases del dominio.

Práctica 2: Agregar la estructura de carpetas hexagonal que corresponde (imagen de ejemplo dada antes) e ir poniendo la partida dentro de: domain.model
(NO debe tener ninguna marca de infraestructura en nuestra clase del dominio, nada de @Entity y demás) crear una clase: PartidaEntityMapper que tiene dos métodos:
Partida toDomain(PartidaEntity)
PartidaEntity toPersistence(Partida)

La clase PartidaEntity y el mapper se guardan en: infrastructure.adapter.secundary
Debe continuar funcionando bien la api

Para conseguir que nuestro dominio esté aislado de la infraestructura necesitamos crear interfaces. Uno para separarnos de los controller y otro para separarnos de los repositorios de bases de datos. Así pondremos en: domain.port.primary un interfaz:

```
public interface ITresEnRayaService {
	Partida findById(Integer id);

	Partida save(Partida partida);

	Partida nuevaPartida(String nombreJugador);

}
```

y en domain.port.secundary otro interfaz:

```java
public interface ITresEnRayaRepository {

	Partida findById(Integer id);

	Partida save(Partida partida);

}
```

Vamos a crear un servicio en el dominio, que será inyectado (tendrá puesto: @Service) a los controller y que hace llamadas al interfaz secundario que hemos creado:

```java
@Service
public class TresEnRayaService implements ITresEnRayaService{

	@Autowired ITresEnRayaRepository partidaRepository;

	@Override
	public Partida findById(Integer id) {
		Partida  findById = null;
		if( id != null) {
			findById = partidaRepository.findById(id);
		}
		return findById;
	}

	@Override
	public Partida save(Partida partida) {
		return partidaRepository.save(partida);

	}

	@Override
	public Partida nuevaPartida(String nombreJugador) {
		Partida partida = new Partida(nombreJugador);

		return partidaRepository.save(partida);
	}

}
```

Vemos que hay un @Autowired para el interfaz: ITresEnRayaRepository

Pero ¿ a quién llama ? Crearemos una clase: PartidaEntityService que implemente el interfaz ITresEnRayaRepository y que por tanto, será inyectada en la clase que acabamos de ver.

Un ejemplo de esa clase en infrastructure.adapter.secundary:

```
@Service
public class PartidaEntityService implements ITresEnRayaRepository{

	@Autowired PartidaEntityRepository peRepository;

	@Override
	public Partida findById(Integer id) {
		Partida partida=null;
		if( id != null) {
			Optional<PartidaEntity> opt = peRepository.findById(id);
			if(opt.isPresent()) {
				PartidaEntity partidaEntity = opt.get();
				PartidaEntityMapper mapper = new PartidaEntityMapper();
				partida = mapper.toDomain(partidaEntity);
			}
		}

		return partida;
	}

	@Override
	@Transactional
	public Partida save(Partida partida) {
		Partida resultado = null;
		if(partida != null) {
			PartidaEntityMapper mapper = new PartidaEntityMapper();
		PartidaEntity save = peRepository.save(mapper.toEntity(partida));
			resultado = mapper.toDomain(save);
		}
		return resultado;
	}

}
```

El @Autowired PartidaEntityRepository es el habitual repositorio jpa (lo ubicamos en el mismo paquete):

```
@Repository
public interface PartidaEntityRepository extends JpaRepository<PartidaEntity,
Integer>{

}
```

Ahora en PartidaRESTController habrá un @Autowired para el interfaz que pusimos en domain.port.primary:

```
@Autowired ITresEnRayaService partidasService;
```

> **Práctica 3**: Modificar la api para que esté separado como se ha descrito el dominio del resto de la aplicación

Ahora corresponde poner la seguridad. Lo que tenemos que tener en cuenta es que tenemos que seguir el patrón hexagonal: una clase Usuario en el dominio y un servicio del dominio con sus correspondientes puertos. Luego un UsuarioEntity para JPA el mapper y el repositorio.

Ahora al crear una nueva partida en el restcontroller la información del nombre de jugador podemos tomarla del usuario autenticado:

```
Object p= SecurityContextHolder.getContext().getAuthentication().getPrincipal();
String nombreAutenticado = ((UserDetails)p).getUsername();
```

> **Práctica 4**: Modificar la api para incluir seguridad mediante una tabla en la base de datos relacional de usuarios con roles

Bases de datos Documentales: Mongo

Las bases de datos documentales presentan la ventaja de trasladar mejor las entidades de programación orientada a objetos a la base de datos.

Están pensadas para acceso rápido a la información, pudiendo darse más redundancia en la información almacenada que una base de datos relacional normalizada

Instalación en docker

```
docker pull mongo
```

podemos ver que nos ha quedado instalado observando las imagenes creadas:

```
docker images -a
```

si quisiéramos eliminar la imagen creada de mongo:

```
docker rmi mongo -f
```

arrancar el servicio en el puerto 27017 identificado por contenedor docker: custommongodb

```
docker run -d -p 27017:27017 --name custommongodb mongo:latest
```

parar el servicio:

```
docker stop custommongodb
```

ver los contenedores de docker y su status:

```
docker ps -a
```

borrar el contenedor creado:

```
docker rm custommongodb;
```

Acceder al contenedor docker para usar un terminal:

```
docker exec -it custommongodb bash
```

Una vez dentro accedemos al terminal interactivo:

```
mongosh
```

Práctica 5: Crear un docker con mongo. Arrancarlo y acceder con mongosh al terminal interactivo. Tomar captura de pantalla

Trabajando con MongoDB

Instrucciones básicas:

ver databases:
```
show dbs;
show databases;
```

crear y/o usar una database:
```
use pruebas;
```

ver tablas:
```
show tables;
```

crear una tabla/insertar documentos:
Nos ubicamos en la database
después escribimos db.nombretabla.insert()
con un documento entre paréntesis
En el mismo momento que agregamos se crea la tabla
Ejemplo:
```
use pruebas;
 db.actores.insertOne({nombre: "ana","apellidos": "pl", "edad": 21})
```

ver objetos/buscarlos en una tabla:
```
db.nombretabla.find(query)
```

Ejemplo:
```
db.actores.find();
db.actores.find({"edad":21});
```

Modificar documentos:
usamos la orden updateMany() para modificar datos ya establecidos

Ejemplo:

```
db.actores.updateMany( {edad:{$gt: 20}}, { $set: { edad: 31 }} )
```

$gt es mayor que ($lt menor que, etc) y se está estableciendo la nueva edad: 31

Con esa misma instrucción (updateMany) podemos agregar nuevos campos en la modificación:

```
db.actores.updateMany( {apellidos: "pl" }, { $set: { peso: 50 }} )
```

Borrar documentos:
usamos deleteMany() y le pasamos la condición
así por ejemplo:

```
db.peliculas.deleteMany({});
```

borra todos los documentos de la colección películas

Práctica 6: Crear una base de datos peliculas. Crear 5 actores con las edades respectivas: 20, 30, 34, 40, 58

Disminuir la edad de todos en 15 años (buscar como hacerlo. Pista: **$inc**)
Borrar los menores de edad (edad < 18)
Copia las instrucciones realizadas y toma captura de pantalla del resultado

Relaciones

Si son pocos los objetos, se prescinde de registrar independientemente los objetos de la relación en grado N, quedan "embebidos" en la otra

Ejemplo un albañil y sus aprendices

```
{
        _id: "98172345",
        nombre: "Arqui",
        apellidos: "Medes",
        especializacion: "puentes",
        aprendices:[
                { nombre: "julio", edad: 17
                },
                { nombre: "marco", edad: 19
                }
        ]
}
```

Si ya hablamos de casi miles puede ser demasiada información Ahora se guardarán los objetos de la parte N en su propio objeto y en la parte de la relación: 1 pondremos un listado de ids:

Instituto y alumnos

Instituto:

```
{
	_id: "98172345",
	nombre: "IES Puerto",
	direccion: "c/las cabezas",
	id_alumnos:[
		20072005055,
		20072005059,
		20081005077,
	]
}
```

Alumno:

```
{
	_id: "20072005055",
	nombre: "Ana",
	edad: 21
}
```

Observar que no hemos puesto en Alumno el id de Instituto. Esa opción de agregar tal: id_instituto en Alumno
es viable y buena si hay búsquedas de alumnos que luego precisan de la información
del instituto en el que estén matriculados. Eso sí, implica que una modificación tiene que tocar ambas entidades

Imaginemos que lo habitual es localizar únicamente el nombre del alumno en lugar de todos sus datos
en ese caso puede ser interesante "desnormalizar" y poner tal dato en el array id_alumnos de la entidad Instituto:

```
	id_alumnos:[
		{ id: 20072005055, nombre: "Ana"},
		{ id: 20072005059, nombre: "Mina"},
		{ id: 20081005077, nombre: "Tina"},
	]
```

Finalmente si es mucha más cantidad los implicados en nuestra 1:N reflejamos al estilo de una DDBB relacional, esto es, poniendo el id del objeto del lado 1 en el objeto del lado N

```
Pais:
{
        _id: "1234567"
        nombre: "India",
        tamaño: 3287000000
}

Ciudadano:
{
        _id: "474747"
        nombre: "Asmid",
        id_pais: "1234567"
}
```

La siguiente guía contiene reglas a considerar cuando se esta diseñando un modelo 1:N para una base de datos:

1. Preferir embeber documentos a menos que haya una razón de peso para no hacerlo.

2. Necesitar acceso a los detalles de los documentos embebidos como entidades independientes, es una razón para no embeberlos.

3. Los arreglos no deben crecer ilimitadamente. Si hay más de un par de cientos de documentos en el lado “Muchos”, no se deben embeber; si hay mas de unos pocos miles de documentos en el lado de “Muchos”, es mejor no utilizar un arreglo de referencias de ObjectID. Los arreglos de alta cardinalidad son una razón de peso para no embeber documentos.

4. Considerar la proporción escritura/lectura cuando se desnormaliza. Un campo que en su mayoría será consultado y solo se actualizará rara vez es un buen candidato para la desnormalización: si se desnormaliza un campo que es frecuentemente actualizado entonces el trabajo extra de encontrar y actualizar todas las instancias puede opacar las ventajas obtenidas con la desnormalización.

5. En MongoDB, el modelo de datos depende – completamente – de los patrones de acceso a los datos de la aplicación en particular. Lo recomendable es estructurar los datos para encontrar las formas en que la aplicación hace las peticiones y las
actualizaciones.

Práctica 7: Crear la base de datos: tresenraya y dentro crear una partida en la tabla partidas, jugada por: ana contra el ordenador, estado FINALIZADA y el tablero: "XXX-OO-O-"

Mongo con Spring

Para activar acceso con Mongo en spring basta con especificar en las properties de spring los atributos correspondientes:

spring.data.mongodb.database=tresenraya
spring.data.mongodb.port=27017
spring.data.mongodb.host=localhost
spring.data.mongodb.authentication-database=admin

Ahora en hexagonal, en lugar de hablar de: PartidaEntity hablaremos de PartidaDocument También puede ser útil separar en subcarpetas el contenido de: infrastructure.adapter.secundary para distinguir entre Mongo y Mysql

El equivalente al repositorio JPA es MongoRepository:

```
@Repository
public interface PartidaDocumentRepository extends
MongoRepository<PartidaDocument,String>{

}
```

Las entities, como dijimos antes, serán documents y usan la notación: @Document (especificando con: collection la tabla correspondiente). Por otro lado el id (se anota con @Id) lo establecemos a una String o con BigInteger. Eso nos va a implicar modificar clase Partida del dominio. Se propone poner String y para que siga funcionando la tabla partidas en mysql Hacer que el PartidaEntityMapper transforme la String en un Integer

Ejemplo:

```
@Document(collection="partidas")
public class PartidaDocument {
	@Id
	private String id;
```

El resto es igual (debe haber getter y setter, etc)

Observar que podemos dejar todo lo de mysql (todas las clases entity) pero debemos comentar la anotación: @Service en PartidaEntityService para que no haya problemas, ya que Spring vería que puede inyectar dos servicios idénticos (tanto PartidaEntityService como PartidaDocumentService)

> **Práctica 8**: Modificar la api para que ahora las partidas (únicamente la tabla partidas, la de usuarios se queda como está) las guarde y las recupere de MongoDB.

Websocket

Según wikipedia:

"WebSocket es una tecnología que proporciona un canal de comunicación bidireccional y full-duplex sobre un único socket TCP. Está diseñada para ser implementada en navegadores y servidores web, pero puede utilizarse por cualquier aplicación cliente/servidor.

Debido a que las conexiones TCP comunes sobre puertos diferentes al 80 son habitualmente bloqueadas por los administradores de redes, el uso de esta tecnología proporcionaría una solución a este tipo de limitaciones proveyendo una funcionalidad similar a la apertura de varias conexiones en distintos puertos, pero multiplexando diferentes servicios WebSocket sobre un único puerto TCP"

Si no hay restricción de conexiones crear una conexión mediante socket tradicional no tiene mayores dificultades. Pero al comunicarnos a través de la web hay que poner reglas y filtros por seguridad. Así que este protocolo empieza con una comunicación HTTP (que sí está autorizada) y abre una conexión que se comunica con websocket

Pero una vez abierto nos encontramos con que hay que gestionar las conexiones. Pensemos en un chat con sockets tradicionales: Hay que llevar una gestión de a quién se envía que

Ahí surge: **Stomp** que es un protocolo de texto sobre websocket. Que nos permite hacer esa gestión (hace la gestión con colas de mensajes).

Haremos uso de las siguiente operaciones con Stomp (Entendemos por broker , quién gestiona las colas de mensajes)

- **connect**: Establece conexión con el broker de mensajería.
- **subscribe**: El cliente se suscribe a un destino del broker (una cola o un topic).
- **send**: El cliente envía un mensaje a un destino del broker (cola o topic).
- **disconnect**: Cierra la conexión con el broker de mensajería.

Spring boot ya viene preparado para trabajar con Stomp y websocket con una misma dependencia:

```
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-websocket</artifactId>
		</dependency>
```

Crearemos un fichero de configuración (package config) por ejemplo: WebsocketConfig:

```
@Configuration
@EnableWebSocketMessageBroker
@Order(Ordered.HIGHEST_PRECEDENCE + 99)
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {


	@Autowired JwtService jwtService;
	@Autowired private UsuarioService usuarioservice;
```

```
    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/websocket").setAllowedOriginPatterns("*")
        //.withSockJS()
        ;
    }

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
// aquí a lo que nos suscribimos. Para temas libres: /topic/chatroom etc.
//Para particulares /users/queue/messages
        registry.enableSimpleBroker("/topic","/queue");

//para enviar mensajes generales desde el cliente debe empezar por: /app
        registry.setApplicationDestinationPrefixes("/app");


//para recibir mensajes particulares en el cliente la suscripción
//debe empezar por: /users
        registry.setUserDestinationPrefix("/users");

    }

    @Override
    public void configureClientInboundChannel(ChannelRegistration registration)
{
        registration.interceptors(new ChannelInterceptor() {
            @Override
            public Message<?> preSend(Message<?> message, MessageChannel
channel) {
                StompHeaderAccessor accessor =
                        MessageHeaderAccessor.getAccessor(message,
StompHeaderAccessor.class);
                System.out.println("Headers: "+ accessor);

                assert accessor != null;
                if (StompCommand.CONNECT.equals(accessor.getCommand())) {

                    String authorizationHeader =
accessor.getFirstNativeHeader("Authorization");
                    assert authorizationHeader != null;
                    String token = authorizationHeader.substring(7);

                    String username = jwtService.extractUsername(token);
                    Usuario usuario = usuarioservice.findByName(username);
                    UserDetailsLogin userDetails = new UserDetailsLogin();
                    userDetails.setUsername(username);
                    userDetails.setPassword(usuario.password);
                    userDetails.setRole(usuario.getRol());




                    UsernamePasswordAuthenticationToken
usernamePasswordAuthenticationToken = new
```

```
UsernamePasswordAuthenticationToken(userDetails, null,
userDetails.getAuthorities());

SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenticat
ionToken);

                    accessor.setUser(usernamePasswordAuthenticationToken);
                }

                return message;
            }

        });
    }
}
```

A destacar:

```
public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/websocket").setAllowedOriginPatterns("*")
        //.withSockJS()
        ;
    }
```

Como ya comentamos hay que generar un punto para que comience una petición http (que en general permiten los firewall) y luego se pase a un conexión websocket (Normalmente las conexiones “related” se permiten en los firewall: conexiones que surgen a partir de otras ya establecidas)
También aparece para evitar problemas de cors: `.setAllowedOriginPatterns("*")` Y entre comentarios aparece: .wthSockJS() que se usa para sistemas (navegadores) del lado del cliente que no soporten websocket y lo utilicen como sustituto

Pongamos que las rutas que usará un cliente sean las siguientes:

'ws://localhost:8080/websocket' ← punto inicial para la conexión websocket
"/app/publicmessage" ← ruta stomp para enviar mensajes a todo el mundo
"/app/private" ← ruta stomp para enviar mensajes privados a otros usuarios
'/topic/chatroom' ← ruta stomp para recibir mensajes de todos
'/users/queue/messages' ← ruta stomp para recibir mensajes privados

Observar que después de la primera, todas son rutas que NO son http, se dan ya con la conexión establecida y son rutas del broker para gestionar las colas de mensajes. También vemos que damos rutas stomp diferentes para enviar que para recibir

Ahora veamos la relación de esas rutas con el método `configureMessageBroker()`

```
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.enableSimpleBroker("/topic","/queue");
        registry.setApplicationDestinationPrefixes("/app");
        registry.setUserDestinationPrefix("/users");
    }
```

Vemos que con: `enableSimpleBroker("/topic","/queue");` establecemos colas en: /topic y /queue La primera será para temas de todo un grupo y la segunda para privados

Con `setApplicationDestinationPrefixes("/app");` establecemos el prefijo para que nos envíen mensajes. Por eso existen las rutas en el cliente: /app/publicmessage y /app/private. La parte de publicmessage y private se establece en el controlador

Con `setUserDestinationPrefix("/users")` obligamos que “el buzón” de mensajes a particulares comiencen todos por: /users: `'/users/queue/messages'` es una ruta que se compone de ese prefijo: /users seguida del broker específico que hemos creado (recordar que pusimos broker para /topic y para /queue)

```
public void configureClientInboundChannel(
```

es un método que **NO** NECESITAMOS si nuestra app no usa seguridad para el websocket . Lo que hace es “interceptar” la petición de conexión, lee las cabeceras que vengan y extrae a mano la información del token para validarlo. Si la hace alguien que no tiene un token de autenticación previo su solicitud de conexión al websocket se verá rechazada.
Nota: vemos en el código del interceptor una system.out.println a modo de debugger. Lo quitaremos en produccción

Es lógico pensar, que lo que hace el interceptor ya lo controlamos en la configuración de seguridad de Spring. Pero lo cierto es que el protocolo de websocket no soporta las cabeceras de Authorization así que lo que se hace es desactivar la seguridad para websocket y luego ponemos el interceptor que acabamos de ver. La seguridad nos queda ahora así (fichero: config/SecurityConfiguration.java):

```
@Configuration
@EnableWebSecurity
public class SecurityConfiguration {

      @Autowired  private JwtFilter jwtAuthFilter;

      @Bean
      public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {

            http
                  .cors(cors->cors.disable())
                  .csrf(csrf -> csrf.disable() )
                      .authorizeHttpRequests(auth -> auth
```

```
                        .requestMatchers(HttpMethod.OPTIONS, "/**").permitAll()
                              .requestMatchers(
                              "/", "/swagger-ui.html",
                              "/swagger-ui/**", "/v2/**",
                              "configuration/**",     "/swagger*/**",
                              "/webjars/**", "/api/login",
                              "/api/register", "/v3/**",
                              "/websocket*/**", "/index.html", "/api/v1/**"
                              ).permitAll()

                              .requestMatchers("/api/v3/**").hasRole("ADMIN")
                              .anyRequest().authenticated()
                        )
                        .sessionManagement(sess ->
sess.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                        .addFilterBefore(jwtAuthFilter,
UsernamePasswordAuthenticationFilter.class);

                return http.getOrBuild();
        }
}
```

Observar que el endpoint: /websocket se permite sin estar autenticado, por lo que comentamos antes.

Para terminar con el websocket precisamos un controlador. Supongamos que lo hemos llamado: WebsocketController

```
@Controller
@CrossOrigin
public class WebsocketController {

      @Autowired
      private SimpMessagingTemplate simpMessagingTemplate;

      @Autowired
      private MensajeService mensajeService;


    @MessageMapping("/publicmessage")
    @SendTo("/topic/chatroom")
    public MessageTo sendMessage(@Payload MessageTo chatMessage) {
 //Mensaje m = Mensaje.newPublic(chatMessage.getAuthor(),"chatroom",
chatMessage.getContent());
      //m.setFecha(new java.util.Date());
      //Mensaje save = mensajeService.save(m);
        return chatMessage;
    }
```

```
    @MessageMapping("/private")
     public void sendSpecific(
      @Payload MessageTo msg,
      Principal user,
      @Header("simpSessionId") String sessionId) throws Exception {
      //Mensaje m = Mensaje.newPrivado(msg.getAuthor(),msg.getReceiver(),
msg.getContent());
      //m.setFecha(new java.util.Date());
      //Mensaje save = mensajeService.save(m);

      simpMessagingTemplate.convertAndSendToUser(msg.getReceiver(),
"/queue/messages", msg);
    }
}
```

Entre comentarios aparecen las líneas que pondríamos si quisiéramos guardar en base de datos los mensajes, para generar un histórico de mensajes

Vemos dos endpoints (recordar que son para rutas stomp, no son http) y nos dice que el primero hace que lo que nos llega por /publicmessage lo redirijamos a la cola /topic/chatroom (recordar que desde el cliente eso se traduce en /app/publicmessage para enviarnos el mensaje y recibirlo en /topic/chatroom)

Luego tenemos el endpoint privado que recibe en /private y envía a /queue/messages (desde el cliente son las rutas: /app/private y /users/queue/messages) Observar que recibimos un objeto: Principal que hemos inyectado en el interceptor Ese objeto contiene el nombre del usuario. También tenemos el id de la conexión.

Nota: para mejor control de la seguridad podemos modificar el autor del mensaje recibido por el nombre de usuario que tenemos en el objeto Principal antes de ejecutar: simMessagingTemplate

Finalmente con `simpMessagingTemplate.convertAndSendToUser()` conseguimos que el broker nos envíe a una cola específica para el cliente dado en el destinatario: msg.getReciber() . Nosotros no lo vemos (el broker lo gestiona internamente) pero quedan colas específicas: /users/queue/user-id/messages que el cliente se suscribe automáticamente al hacerlo a: /users/queue/messages

Práctica 9: Modificar la api para que tenga soporte para websocket. Crear una sala común de chat y permitir las comunicaciones particulares. Guardar en base de datos (en mongo) los mensajes para poder recuperar el histórico

Práctica 10: Hacer una nueva versión que permita dos jugadores con comunicación instantánea (websocket)

GraphQL

Introducción

Según wikipedia:

GraphQL es un lenguaje de consulta y manipulación de datos para APIs, y un entorno de ejecución para realizar consultas con datos existentes. GraphQL fue desarrollado internamente por Facebook Proporciona una aproximación para desarrollo **APIs web** y ha sido comparado y contrastado con REST y otras arquitecturas de servicio web. Permite a los clientes definir la estructura de datos requerida, y la misma estructura de datos será regresada por el servidor, impidiendo así que excesivas cantidades de datos sean regresadas, aunque esto tiene implicaciones en cómo de efectivo puede ser el caché web de los resultados de estas consultas. La flexibilidad y riqueza del lenguaje de consulta también añade complejidad que puede no valer la pena para APIs mas sencillas. Consta de un sistema de tipos, lenguaje de consulta y semántica de ejecución, validación estática, e introspección de tipos.

Nombrar Apis web es nombrar HTTP, y dentro de ese tipo, el detalle especial de GraphQL es que hay un único endpoint y se accede mediante POST para todas las consultas, enviando en el cuerpo un JSON:

POST /graphql

y en el body un ejemplo:

```
{
    query: " {  getAllPerson {    name  } }"
}
```

La consulta anterior, parece que trata de obtener el conjunto de personas, pero únicamente su nombre. Importante observar que se lo pasamos como un JSON con un clave valor donde la clave es: query y el valor es la verdadera consulta GraphQL pasada por JSON.stringify()

Los estados de como terminan las consultas son informadas con el JSON que devuelve Por ejemplo, si hay un problema de autorización en el json habrá un mensaje en el JSON devuelto informando. **NO** es habitual enviar una cabecera con un status code de forbidden


```
{
  "errors": [
    {
      "message": "Unauthorized",
      "locations": [
        {
          "line": 1,
          "column": 5
        }
      ],
      "path": [
        "getAllPerson"
      ],
      "extensions": {
        "classification": "UNAUTHORIZED"
      }
    }
  ],
  "data": {
    "getAllPerson": null
  }
}
```


En el anterior JSON de respuesta vemos que se solicitó el método: getAllPerson() pero quién lo hizo no estaba autorizado. La parte de data (lo que sería la respuesta solicitada) devuelve un null y en la parte de “errors” se muestra por qué motivo. Observar que el status code de la consulta http será 200 Esto es, devolverá como que todo está ok. Es en el cuerpo del mensaje donde se informa

Queries

Las consultas que devuelven información (no la crean ni la modifican) son llamadas: **query** y tienen formatos similares al siguiente:

```
query {
  character(id:2){
    id
    name
  }
}
```

La anterior consulta es de la api pública rick y morty: https://rickandmortyapi.com/graphql

Se ve que le pasamos una función: character() en la que se busca el character (personaje) con id 2. Dentro de las llaves se especifica que campos queremos del objeto: Character en este caso se dice que queremos el id y el name.

En la imagen vemos la consulta ejecutada por Rested. Observar que se ha convertido la consulta propiamente a una string para que tenga un formato JSON apropiado

Práctica 11: Ejecutar la consulta anterior desde Rested o Postman Solicitar también que devuelva el campo: género (gender)

Si hacemos lo anterior desde un cliente react, angular,… Es habitual pasar la query por: JSON.stringify para que la consulta POST tenga un cuerpo JSON válido

Para poder establecer del lado del servidor, como puede ser una consulta se usan los tipos de datos que creemos y los tipos: queries. Vamos a ver un ejemplo:

```
type Curso{
   id: Int
   nombre: String
   categoria: String
 }
type Query {
   cursos: [Curso]
   curso(id : Int! ): Curso
 }
```

Al especificar: type Query Estamos informando de que estamos definiendo una consulta. Al especificar: type Curso estamos creando un nuevo tipo de datos para la comunicación (un DTO)

En: `cursos: [Curso]` estamos diciendo que nos pueden hacer una consulta llamada: cursos y que nos devolverá un array de objetos del tipo Curso Ya que se denota un array mediante los corchetes. Así que: [Curso] hace referencia a una lista de Curso

Luego se ha definido una segunda consulta:

```
curso(id: Int!) : Curso
```

Se está estableciendo que la consulta se llama: curso() y recibe un parámetro id de tipo entero. La admiración informa de que es un parámetro obligatorio. Y finalmente dice que devuelve un Curso

Creando las primeras queries en springboot

Elegimos en spring initializr: **Spring for GraphQL** Las dependencias maven más importante que incorpora es:

```
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-graphql</artifactId>
</dependency>
```

En **application/properties** pondremos la siguiente línea:

```
spring.graphql.graphiql.enabled=true
```

La linea anterior nos genera una pagina que nos permite hacer pruebas para nuestras queries y de alguna forma "documentar" nuestra api, algo así como Swagger. La ruta de acceso estará en: http://localhost:8080/graphiql

Ahora hay que crear un Schema. En src/main/resources/graphql crearemos un fichero: schema.graphqls de texto y pondremos los tipos anteriores (type Curso y type Query) Recordar que **type Query** es la forma que nos da graphql para determinar las queries. **type Curso** es únicamente un tipo personalizado. **Spring mapeará la clase java: Curso que hayamos creado con ese tipo**.

Vamos a crear un controlador de prueba:

```
class Curso{
    public Curso() {}
    int id;
    String nombre;
    String categoria;
    public int getId() {
    public void setId(int id) {
    public String getNombre() {
    public void setNombre(String nombre) {
    public String getCategoria() {
    public void setCategoria(String categoria) {
}

@Controller
public class CursoController {

    @QueryMapping
    public List<Curso> cursos() {
        Curso c1 = new Curso();
        c1.id = 1; c1.nombre = "LND"; c1.categoria = "DAM";
        Curso c2 = new Curso();
        c2.id = 2; c2.nombre = "BAE"; c2.categoria = "DAM";
        return Arrays.asList(c1,c2);
    }
}
```

Observamos que aparece una notación: @Controller y una notación nueva: @QueryMapping esta notación enlaza el método con el type Query del Schema. Tenemos que poner el mismo nombre en el método que hayamos puesto en el Schema En este caso aparece el nombre: cursos que es el que definimos en el Schema para devolver un array de cursos: `cursos: [Curso]`

Práctica 12: Crear un proyecto spring boot y agregarle en spring initializr: spring for graphql y springweb Seguir los pasos descritos y comprobar desde postman/rested así como probar la ruta graphiql antes mostrada (el que es al estilo de swagger)

Cuando queremos enlazar un parámetro recibido en la consulta con una etiqueta de una función en un esquema graphql (como en el ejemplo anterior: curso(id: Int!)) usamos **@Argument** Así en el controlador si hemos creado una query:

```
@QueryMapping  curso( @Argument long id ){ … }
```

> **Práctica 13**: Crear en el proyecto anterior un código de prueba para la query que está en el schema: `curso(id : Int! ): Curso` Comprobar que funciona desde graphiql

También podemos usar la anotación @SchemaMapping De hecho QueryMapping lo que hace es hacer un SchemaMapping cuyo padre es "Query". Pero si usamos @SchemaMapping (nos vale tanto para grabar-modificar datos mediante mutaciones como para consultas de datos mediante queries) podemos especificar nombres diferentes. Así por ejemplo para el caso anterior de:

```
type Query {
   cursos: [Curso]
   curso(id : Int! ): Curso
 }
```

Podemos hacer el siguiente @SchemaMapping en el controlador:

```
      @SchemaMapping(typeName="Query", field="cursos")
      public List<Curso> verTodosLosCursos() {}
```

En: **typeName** se especifica el tipo que en nuestro caso es una: Query y le decimos mediante: **field** que tome el campo del esquema llamado: cursos. Así el método: verTodosLosCursos() queda mapeado a la query cursos del esquema

queries: Tipos de datos

Los escalares que soporta graphql:

Int, Float, Boolean, String. Adicionalmente existe: ID que se interpreta como un identificador único (se resuelve como una String)

Podemos establecer que un dato es obligatorio por medio de la admiración: ! Así si ponemos:

busquedaPorId(id: Int!)

Estamos diciendo que el método busquedaPorId recibe un parámetro id obligatorio

Argumentos, input type

Tanto cuando buscamos información, como cuando la guardamos, se puede recibir parámetros enviados desde el cliente. Antes vimos: `curso(id : Int! ): Curso` que se ve que recibe como argumento una etiqueta: id obligatoria de tipo entero.

Los input type vienen a ser DTOs que nos envía el cliente. Normalmente para guardado/actualización de datos (mutaciones) pero también es válido para filtrar queries. El siguiente ejemplo lo muestra:

Schema graphql:

```
type Query{
    libros: [Libro]
    autores: [Autor]
    getLibroBy( filtro: FiltroLibro): [Libro]
}

input FiltroLibro{
    categoria: String
    nombreautor: String
}
```

y el equivalente como DTO en Java: (se omiten getter/setter constructores)

```
class FiltroLibro{
    String categoria;
    String nombreautor;
}
```

La parte nueva es la etiqueta: input FiltroLibro que le dice a graphql que es un tipo de dato de entrada, en este caso se recibe en la consulta

> **Práctica 14**: Continuando con el anterior crear una query que reciba un input type que refleje un subnombre para el curso (que sea un trozo de texto contenido en el nombre) y la categoría

Mutaciones

Con las queries consultamos/obtenemos información de graphql pero si queremos guardar/modificar información usaremos las mutaciones: **type Mutation** en el esquema graphql

El siguiente ejemplo es para agregar una nueva persona:

```
type Mutation {
    agregar(id: Int!, name: String, age: Int): Person!
}
```

Vemos que en el esquema se recibe id, name, age y se crea el nuevo Person (que luego se devuelve también al cliente de la solicitud)

Si usamos en el controlador: @SchemaMapping sería:

```
@SchemaMapping(typeName = "Mutation", field = "agregar")
public Person crearPersona(
      @Argument int id, @Argument String name, @Argument int age
) {}
```

También podríamos haber usado @MutationMapping (equivalente al @QueryMapping que ya hemos visto) y que es un @SchemaMapping que ya se entiende como una mutación: tiene su typeName= “Mutation”

¿ y cómo hacemos la petición desde el cliente ? Seguimos enviando un JSON con un único clave/valor: **“query”: JSON.stringify(mutation{})**

Como ejemplo una mutación desde Rested que crea una nueva persona (sigue el patrón de crearPersona() que acabamos de ver) Observar que el nombre del método enviado es el del esquema: agregar() y luego nos devuelve el id, name y age del objeto creado:

```
{
"query" : " mutation{ agregar(id: 2, name: \"Armando\", age: 31){ id name age } }"
}
```

Práctica 15: Hacer una mutación para los cursos y luego probarla desde Rested/Postman

Seguridad con graphql

Tenemos que tener en cuenta que hay un único endpoint, entonces una parte importante de la seguridad no la podemos aplicar (todos los requestmatcher que usamos en REST) No podemos securizar el endpoint: /graphql de hecho si queremos usar la interfaz gráfica tampoco: /grphiql Así que pondremos en la configuración de seguridad algo parecido a:

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity
public class SecurityConfiguration {

	@Autowired  private JwtFilter jwtAuthFilter;

	@Bean
	public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {

		http
			.cors(cors->cors.disable())
			.csrf(csrf -> csrf.disable() )
				.authorizeHttpRequests(auth -> auth
					.requestMatchers(HttpMethod.OPTIONS,
"/**").permitAll()
					.requestMatchers(
					"/", "/swagger-ui.html",
					"/swagger-ui/**", "/v2/**",
					"configuration/**",     "/swagger*/**",
					"/webjars/**", "/api/login",
					"/api/register", "/v3/**",
					"/websocket*/**", "/index.html", "/api/v1/**",
					"/graphql", "/graphiql"
					).permitAll()

					.requestMatchers
("/api/v3/**").hasRole("ADMIN").anyRequest().authenticated()
					)
				.sessionManagement(sess ->
sess.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
				.addFilterBefore(jwtAuthFilter,
UsernamePasswordAuthenticationFilter.class);

		return http.getOrBuild();
	}
}
```

Las partes nuevas las hemos marcado en amarillo: los puntos /graphql y /graphiql les hemos puesto: permitAll() para que no aplique seguridad sobre ese endpoint que va a soportar todas las peticiones.

Por otro lado hemos agregado: `@EnableMethodSecurity` que nos permitirá poner seguridad a nivel de cada @QueryMapping, @MutationMapping, @SchemaMapping

Ahora en el controller, en las queries que se quieran asegurar. Pondremos:

- si se quiere que sea accesible únicamente a autenticados: isAuthenticated()

```
@PreAuthorize("isAuthenticated()")
@QueryMapping
public nombredelmetodo(){}
```

- si se quiere filtrar por roles: hasAuthority(“nombredelrol”)

```
@PreAuthorize("hasAuthority('ROLE_ADMIN')")
@QueryMapping
public List<Person> getAllPersons() {}
```

Práctica 16: Securizar la api de cursos. Donde las mutaciones únicamente las pueda ejecutar un admin