BackEndI
Profesor: Christian Ariel Díaz

Módulo 3. Framework para el desarrollo ágil de aplicaciones

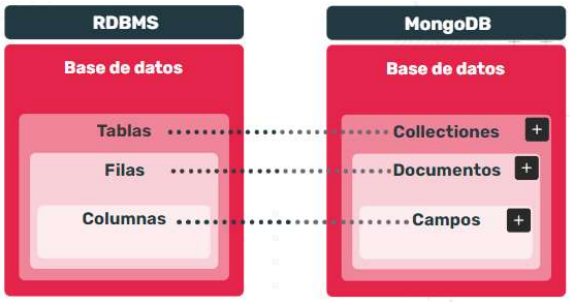
Sesión 38. Bases de Datos No Relacionales: MongoDB
Junio 29 de 2022

Introducción a MongoDB

MongoDB es un sistema open source de base de datos NoSQL orientado a documentos. Almacena los datos en una representación binaria llamada BSON (Binary JSON). La codificación BSON extiende la representación JSON para incluir tipos adicionales como int, long, date, etc.

Los documentos BSON contienen uno o más campos, y cada campo contiene un valor de un tipo de datos específico, que incluye matrices, datos binarios y subdocumentos. Están estrechamente alineados con la estructura de los objetos en los lenguajes de programación.

Elementos de MongoDB

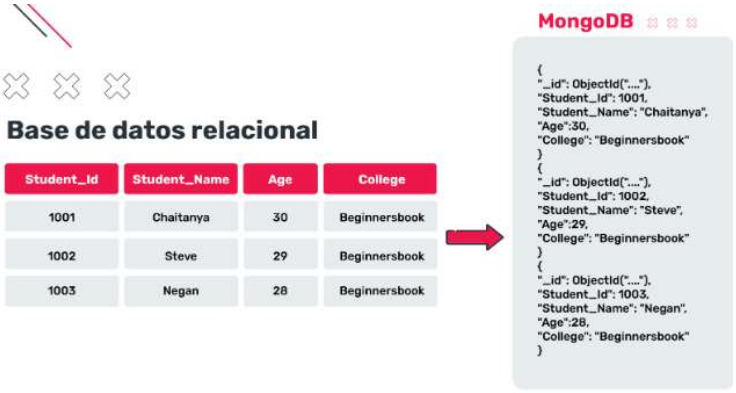


Colecciones:
Una colección es como una tabla y contiene documentos que son equivalentes a los registros.

Documentos:
Un documento es mapeado por un ibjeto en la aplicación.

Campos:
Un documento tiene campos que son pares clave:valor.

.



Anotaciones Spring Mongo Data

@Document
Usada en clases, le indica al MongoDriver que puede tomar objetos de esa clase como Mongo Documents en una colección particular. El nombre de la colección se deduce del nombre de la clase, pero si tienen diferentes nombres es necesario agregar el de la Collection que se va a asociar. Es similar a la anotación @Entity utilizada en JPA.

@Field
Instruye al framework de qué manera persistir los atributos de las clases en Mongo. Podemos utilizar el atributo "name" para especificar el nombre que tiene ese campo en el Document de MongoDB.

```
@Document(collection = "airplanes")
public class Aircraft {
        private String id;
        private String model;
        @Field(name="seats") private int nbSeats;
}
```

@Id
Todo document en una Collection Mongo debe tener un campo de id. Es único y está indexado. Es similar a una clave primaria en la BD relacionales.

@Transient
Excluir un atributo para que no sea persistido.

```
@Document
public class Aircraft {
        @Id private String id;
        private String model;
        @Transient private int nbSeats;
}
```

@Indexed
Si queremos ejecutar queries con filtros más rápidamente. Si, por ejemplo, vamos a realizar muchas queries sobre el field model es bueno indexarlo. Puede especificarse la dirección del index, y si es único.

```
@Document
public class Aircraft {
	@Id private String id;
	@Indexed(direction = IndexDirection.ASCENDING, unique = false)
	private String model;
	private int nbSeats;
}
```

@DbRef
Actúa parecido a un join de una BD relacional. Se une un document a otro de una Collection diferente.

```
@Document
public class Manufacturer {
	@Id private String id;
	private String name;
}

@Document
public class Aircraft {
	@Id private String id;
	private String model;
	@DbRef
	private Manufacturer man;
}
```

MongoDB desde Spring

Crear proyecto con las siguientes dependencias (con SpringBoot Initializer):

```
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>
</dependency>

<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>
</dependency>
```

Agregar el archivo application.properties en la carpeta resources:

```
#mongodb
spring.data.mongodb.host=localhost
spring.data.mongodb.port=27017
spring.data.mongodb.database=mongoexample

// Clase Book
@Document(collection = "books")
public class Book{
	@Id
	private String id;
	private String author;
	@Field(name = "book")
	private String bookTitle;
}

// BookRepository
@repository
public interface BookRepository extends MongoRepository<Book,String> {
}

// BookService
@Service
public class BookService{
	private final BookRepository bookRepository;
	public BookService(BookRepository bookRepository){
		this.bookRepository = bookRepository;
	}
	public List<Book> findAllBooks();
		return bookRepository.findAll(); // Revisar
}

// Controller
@RestController
@RequestMapping(value = "/mongoexample")
public class BookController{
	private final BookService bookService;
	public BookController(BookService bookService){
		this.bookService = bookService;
	}
	@GetMapping(value = "/books")
	public List<Book> getAllBooks();
		return bookService.findAllBooks();  // Revisar
}
```

Queries especiales

Si se desea realizar una query que filtre por un parámetro, como por ejemplo findBooksByAuthor(), se debe pasar el parámetro con el mismo nombre que posee el field en la BD:

```
@Repository
public interface BookRepository extends MongoRepository <Book,String>{
	List<Book> findBooksByAuthor(String author);
}
```

Ejercicio

Una casa de apuestas muy importante quiere agregar a su web una sección en donde se muestre el fixture de diferentes torneos y los resultados en tiempo real. Ya que el acceso a esta información será de forma masiva y recurrente, se utilizará una base de datos no relacional, en este caso MongoDB.

Un torneo está compuesto por:

- ID.
- Nombre.
- País.

Requerimientos:

- Crear una base de datos local llamada “apuestas”.
- Configurar la conexión en application.properties.
- Agregar dependencias necesarias al proyecto.
- Agregar las clases necesarias (Model, Controller, Service, Repository).

La API de los torneos deberá permitir:
- Agregar torneos.
- Listar todos los torneos.

Sincrónico
https://github.com/juan351/clinicaOdontologica

Ejemplo Integrador:
https://github.com/marielagcw/integradormarielagcw

Sesión 39. Integradora 12.
Junio 30 de 2022

Sesión 40. Sistemas Distribuidos
Julio 1 de 2022

Microservicios
Son un enfoque arquitectónico y organizativo en donde el software está compuesto por pequeños servicios independientes que se comunican a través de una API.

Características:

- Autónomos
Cada servicio dentro de una arquitectura de microservicios se puede desarrollar, implementar, operar y escalar sin afectar el funcionamiento de los otros servicios.

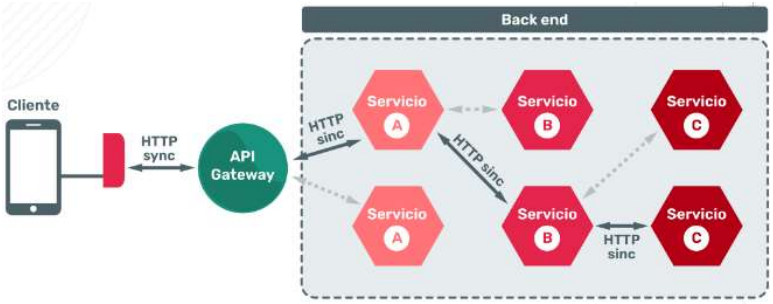
- Especializados
Cada servicio está diseñado para un conjunto de capacidades y se enfoca en resolver un problema específico.

Ventajas
- Agilidad
- Escalabilidad
- Implementación sencilla
- Versatilidad
- Resistencia
- Mantenimiento

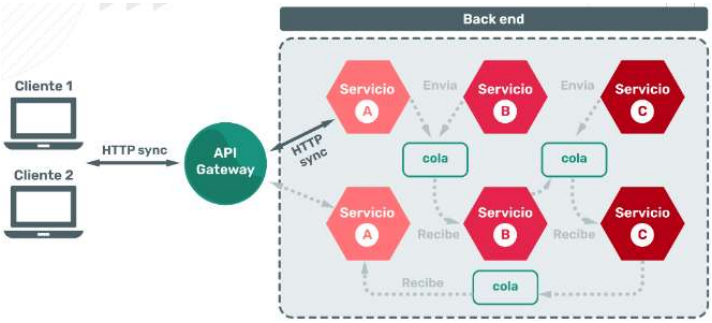
Desventajas
- Mayor consumo de memoria
- Inversión de tiempo inicial
- Complejidad en la gestión
- Dificultad en la realización de pruebas

Comunicación entre microservicios

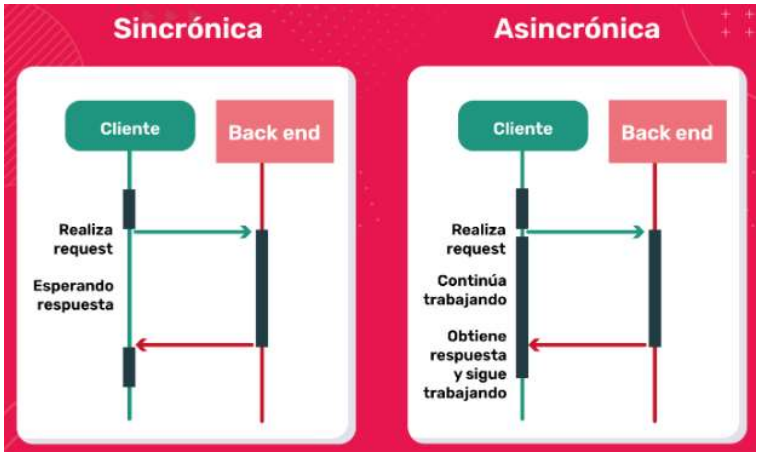
Comunicación Síncrona
En este tipo de comunicación se requiere una dirección de servicio de origen predefinida, hacia dónde enviar la Request, y ambos (remitente y destinatario de la llamada) deben estar en funcionamiento en este momento. El cliente únicamente puede continuar su tarea en el momento que recibe una Response del servidor. El enfoque de Request/Response suele utilizar el protocolo HTTP e incluye REST, GraphQL y gRPC.



Comunicación Asíncrona
En este tipo de comunicación se envía un mensaje a una cola o agente de mensajes. El mensaje se pone en cola si el servicio de recepción está inactivo y continúa más tarde cuando está activo. El remitente del mensaje no espera ninguna respuesta. Los protocolos asincrónicos como MQTT, STOMP, AMQP son manejados por plataformas como Apache Kafka Stream, RabbitMQ.



.



¿Qué elegir?
La comunicación Rest/HTTP funciona para patrones de Request/Response síncronos, para arquitecturas orientadas a servicios (SOA) y APIs expuestas al público.

Algunas desventajas son:

- Baja performance: la Request no obtiene una Response hasta que todas las llamadas internas han terminado esto puede resultar en tiempos de respuesta más lentos. También puede bajar si hay muchas llamadas HTTP.
- Pérdida de autonomía: si los microservicios se conectan a través de HTTP y dependen de la respuesta de otro, no pueden ser totalmente autónomos.
- Manejo de fallas complejo: si hay una cadena de llamadas HTTP y un microservicio intermedio falla, toda la cadena falla. Para esto se utilizan los retries y los circuit breakers.

Suele recomendarse, para la comunicación interna entre microservicios, un patrón asíncrono para disminuir la cantidad de llamadas en cadena, e independizarse del ciclo de Request/Response.

Sesión 41. Manejo de Excepciones
Julio 4 de 2022

Excepciones

Son los errores que se pueden producir en nuestras aplicaciones Java en tiempo de ejecución, es decir, aquellos que se generan cuando estamos ejecutando la aplicación y llevan a que esta finalice abruptamente.

En Java podemos atrapar excepciones y así evitar que nuestro programa se detenga cuando ocurran. Para ello necesitamos encerrar nuestro código en un bloque Try-catch.

```
try {
	double resultado = numerador / denominador;
} catch(ArithmeticException exception) {
	System.out.println("Error: la operación no es válida");
} finally {
	System.out.println("Este mensaje siempre se mostrará, sin importar si
	hay o no error");
}
```

Lanzar excepciones

```
public class Pelicula {

	//…
	public void setRating(double rating) {
		if (rating <= 0) {
			throw new RuntimeException("El rating debe ser positivo");
		this.rating = rating;
	}
}
```

Para evitar que el programa termine su ejecución se utiliza el bloque try-catch:

```
try {
	Pelicula miPeli = new Pelicula();
	miPeli.setRating(-10);
} catch(RuntimeException e) {
	System.out.println(e.getMessage());
}
```

Excepciones personalizadas

```
// Definir una clase que extiende a Exception
public class ExcepcionIntervalo extends Exception {
	public ExcepcionIntervalo(String msg) {
		super(msg);
	}
}

// Utilizar la excepción personalizada
public void setRating(double rating) throws ExcepcionIntervalo {
	if (rating <= 0) {
		throw new ExcepcionIntervalo("El rating debe ser positivo");
	this.rating = rating;
}

// Capturar la excepción
try {
	Pelicula miPeli = new Pelicula();
	miPeli.setRating(-10);
} catch(ExcepcionIntervalo e) {
	System.out.println(e.getMessage());
}
```

Manejo de Excepciones con Spring Boot

@ExceptionHandler
La primera opción es utilizar esta anotación para manejar las excepciones a nivel de controller, es decir, dentro de nuestro controller implementamos un método que se ejecutará en caso de producir un error. Podemos especificar el estado de respuesta para una excepción específica, junto con la definición de la excepción con la anotación @ResponseStatus.

Paso 1
Si necesitamos capturar una excepcion propia creada por nosotros debemos primero crear nuestra propia Excepcion.

```
@ResponseStatus(value = HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends Exception{
	public ResourceNotFoundException(String mensaje){
		super(mensaje);
	}
}
```

Paso 2
Desde algún método del controller podría lanzarse una excepción del tipo ResourceNotFoundException:

```
@RestController
public class MiController{
	@RequestMapping(value = "/files/{id}")
	public String getFile(@PathVariable("id") long id){
		if(id <= 0){
			String mensajeError = "No se encuentra ningun archivo con
			id" + id;
			throw new ResourceNotFoundException(mensajeError);
		}else{
			// ...
		}
	}
}
```

Paso 3
Luego, capturamos la excepcion con @ExceptionHandler:

```
@RestController
public class MiController{


	@ExceptionHandler(Exception.class)
	private ResponseEntity<?> exception(ResourceNotFoundException ex,
	WebRequest request)
}
```

Es buena práctica loguear los errores con log4j:

```
log.error(ex);
return new ResponseEntity<>("Error manejado por exception Handler",
HttpStatus.NOT_FOUND);
```

Desventaja: El método anotado @ExceptionHandlersolo está activo para ese controlador en particular, no globalmente para toda laaplicación.

Excepciones globales

@ControllerAdvice
La anotación @ControllerAdvice nos permite juntar nuestros múltiples @ExceptionHandlers dispersos en un solo componente global de manejo de errores:

- Nos da un control total sobre el cuerpo de la respuesta (Response body), así como sobre el código de estado (response status).
- Proporciona mapeo de varias excepciones al mismo método, para ser manejadas juntas.
- Además podemos usar ResponseEntity para el response.

Ejemplo:

```
@ControllerAdvice
public class GlobalExceptionHandler {

	@ExceptionHandler(ResourceNotFoundException.class)
	public ResponseEntity<?>
	resourceNotFoundException(ResourceNotFoundException ex,
	WebRequest request) {
		return new ResponseEntity<>("Error manejado por Exception
		Handler",
		HttpStatus.NOT_FOUND);
	}

	@ExceptionHandler(Exception.class)
	public ResponseEntity<?> globleExceptionHandler(Exception ex,
	WebRequest request) {
		return new ResponseEntity<>("Error manejado por Exception
		Handler",
		HttpStatus.INTERNAL_SERVER_ERROR);
	}
}
```

Excepciones a nivel de método
Podemos usar la clase ResponseStatusException directamente en un método en particular de un controller.

```
@GetMapping(value = "/{id}")
public Persona findById(@PathVariable("id") Long id, HttpServletResponse
response){
	try {
		Persona persona = service.findOne(id);
		return persona;
	}
	catch (MyResourceNotFoundException ex){
		throw new ResourceNotFoundException(HttpStatus.NOT_FOUND,
		"Persona Not Found", ex);
	}
}
```

Ejercicio
Proyecto paquetería - Parte I
Seguimos trabajando en el proyecto que comenzamos en la clase 39.
Para esta etapa vas a tratar los posibles errores de tu aplicación. Se pide:

- Crear una Excepción llamada BadRequestException.
- Lanzar esta excepción desde el servicio de paquetes, en el caso de que se intente registrar un paquete con un código que ya posee otro paquete.
- Utilizando @ExceptionHandler, crear un método en el controller de los paquetes, para procesar esta excepción lanzada desde el servicio de paquetes.
- Retornar un ResponseEntity con código 400 (BAD REQUEST)
- Por último, debés probar desde Postman el correcto funcionamiento.

Status Codes:

Respuestas Informativas:
100: Continue
101: Switching Protocol
102: Processing
103: Early Hints

Respuestas Satisfactorias:
200: Ok
201: Created
202: Accepted
203: Non-Authoritative Information
204: No Content
205: Reset Content
206: Partial Content
207: Multi-Status
208: Multi-Status
226: IM Used

Redirecciones
300: Multiple Choice
301: Moved Permanently
302: Found
303: See Other
304: Not Modified
305: Use Proxy
306: Unused
307: Temporary Redirect
308: Permanent Redirect

Errores de Cliente:
400: Bad Request
401: Unauthorized
402: Payment Required
403: Forbidden
404: Not found
405: Method Not Allowed
406: Not Acceptable
407: Proxy Authentication Required
408: Request Timeout

409: Conflict
410: Gone
411: Length Required
412: Precondition Failed
413: Payload Too Large
414: URI Too Long
415: Unsupported Media Type
416: Requested Range Not Satisfiable
417: Expectation Failed
418: I’m a teapot
421: Misdirect Request
422: Unprocessable Entity
423: Locked
424: Failed Dependency
426: Upgrade Required
428: Precondition Required
429: Too Many Requests
431: Request Header Fields Too Large
451: Unavailable for Legal Reasons

Errores de Servidor
500: Internal Server Error
501: Not Implemented
502: Bad Gateway
503: Service Unavailable
504: Gateway Timeout
505: HTTP Version Not Supported
506: Variant Also Negotiates
507: Insufficient Storage
508: Loop Detected
510: Not extended
511: Network Authentication Required

Sincrónico

- No se requiere FrontEnd. Con Postman es suficiente.
- Playground: Para el miércoles 13 de julio.

https://app.diagrams.net/#G15Hd0SEddFLu3M7CBz4BbQPbrmkF8tJHn

Mejorar perfil en CV y LinkedIn:
https://www.youtube.com/watch?v=-400BGJsix4&ab_channel=TheFullstackDevs

Jackson nested values:
https://www.baeldung.com/jackson-nested-values

Conversiones de fecha y hora:
para convertir LocalDateTime

en el application properties
spring.jackson.serialization.write_dates_as_timestamps=false
spring.jackson.date-format=yyyy-MM-dd HH:mm:ss

y en el modelo de Turno y turnoDto antes de la propiedad , esta anotación:
@JsonFormat(shape = JsonFormat.Shape.STRING, pattern="yyyy-MM-dd HH:mm:ss")

Sesión 42. Integradora 13
Julio 4 de 2022

Sesión 43. Seguridad
Julio 5 de 2022

Autenticación y Autorización

La autenticación significa confirmar la identidad del uuario, es decir, verificar si es quien dice ser. La autorización significa que se le permite el acceso al sistema, es decir, es el proceso donde se verifica a qué tiene acceso.

Configuraciones en Spring Boot

- Autenticación Basic
- Autorización con JDBC
- Autenticación LDAP
- Autenticación JSON Web Token
- Autenticación OAuth

Spring Security con Spring Boot

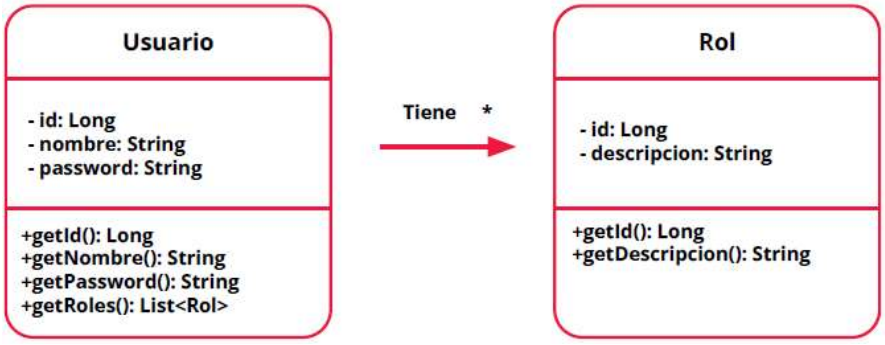
Pasos

1. Agregar la librería, el Starter de Spring Security

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

2. Configurar la administración de roles y users

Para poder tomar decisiones sobre el acceso a los recursos es necesario identificar a los diferentes usuarios y que roles tienen para validar o no si tienen autorización para acceder a los diferentes recurso de la aplicación.
Para ello debemos implementar la interfaz UserDetailsService. Esta interfaz describe un objeto que realiza un acceso a datos con un único método loadUserByUsername que devuelve la información de un usuario a partir de su nombre de usuario.



.

```
@Service
@Transactional
public class UserDetailsServiceImpl implements UserDetailsService {

	@Autowired
	UserRepository userRepository;

	@Override
	public UserDetails loadUserByUsername(String username) throws
	UsernameNotFoundException {

		Usuario appUser = userRepository.findByUsername(username); // Busca
		el usuario por nombre en la BD

		Set<GrantedAuthority> grantList = new HashSet<GrantedAuthority>();
		// Crea la lista de roles/accesos que tiene el usuario

		for (Rol rol: appUser.getRoles()) {
			GrantedAuthority grantedAuthority = new
			SimpleGrantedAuthority(rol.getDescription()); // En la base de
			datos cada cada rol tiene que tener antepuesto "ROLE_". Ej:
			ROLE_USER, ROLE_ADMIN
			grantList.add(grantedAuthority);
		}
		UserDetails user = null;
		user = (UserDetails) new User(username,"{noop}" +
		appUser.getPassword(), grantList); // Crea y retorna el objeto
		UserDetails soportado por Spring Security
		return user;
	}
}
```

3. Configurar la autenticación y seguridad de URL

Usando la anotación @EnableWebSecurity y extendiendo la clase WebSecurityConfigurerAdapter podemos rápidamente configurar y activar la seguridad para los diferentes usuarios que se loguean en nuestra aplicación.

A su vez, @EnableWebSecurity habilita el soporte de seguridad web de Spring Security y también proporciona la integración con Spring MVC y WebSecurityConfigurerAdapter proporciona un conjunto de métodos que se utilizan para habilitar la configuración de seguridad web específica.

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
	@Override
	protected void configure(HttpSecurity http) throws Exception {
		// nuestra propia configuración de seguridad.
	}
}
```

La anotación @Configuration indica que la clase es de configuración y necesita ser cargada durante el inicio del server.

```
// Sobreescribir el método configure
protected void configure(HttpSecurity http) throws Exception {
	http.authorizeRequests()
	.anyRequesy()
	.authenticated()
	.and()
	.httpBasic(); // Activa la protección HTTP básica, normalmente el
	navegador muestra un cuadro donde se nos pedirá introducir nombre de
	usuario y contraseña.
}
```

Ejemplo:

```
protected void configure(HttpSecurity http) throws Exception {
	// Indica que todas las peticiones estarán protegidas, es decir,
	requerimos autenticarnos para poder acceder a cualquier parte del sitio.
	http.authorizeRequests()
	// Un usuario con el rol de USER puede ingresar a la home.
	.antMatchers("/home").hasRole("USER")
	// Un usuario con el rol de ADMIN solo puede ingresar a la página de
ventas.
	.antMatchers("/ventas").hasRole("ADMIN")
	.and().formLogin()
	.and().logout();
}
```

4. Encriptar la contraseña

Las claves de usuario pueden ser codificadas a partir de un algoritmo de encriptación. Una función de encriptación es una función unilateral que toma cualquier texto y lo transforma en un código encriptado que no se puede volver atrás. Spring Security proporciona múltiples implementaciones de codificación de contraseña.

BCryptPasswordEncoder

Instanciando un objeto de la clases BCryptPasswordEncoder, podemos encriptar/hashear un pass, para ellos debemos:

1. Creamos encoder llamando al constructor de BCryptPasswordEncoder con un valor 12:

```
BCryptPasswordEncoder encoder new BCryptPasswordEncoder(12);
```

2. Invocamos al método encode(“pass_para_hashear”), pasando la contraseña que deseamos encriptar, Así es como se ve una contraseña con hash, por ejemplo: encodedPassword: $2a$12$DlfnjD4YgCNbDEtgd/IteOj.jmUZpuz1i4gt51YzetW/iKY2O3bqa

```
String encodedPassword = encoder.encode("UserPassword”);
```

Uso de @PreAuthorize y @PostAuthorize
La anotación @PreAuthorize verifica la expresión dada antes de ingresar al método, para decidir si un usuario en sesión tiene acceso o no a utilizarlo, mientras que la anotación @PostAuthorize la verifica después de la ejecución del método y podría alterar el resultado.

```
@PreAuthorize("hasAnyRole('ROLE_ADMIN','ROLE_USER')")
public User updateUser(User formUser) throws Exception {
	// ...
}
```

Cookies
Una cookie es información que envía el servidor web al cliente (navegador), con información que necesita quedar guardada en la pc de cada usuario. El browser la almacena en archivos de texto plano, de no más de 4kb.

Ventajas
- No consume recursos del servidor.
- Configura usuarios y contraseñas.
- Guarda información de navegación del usuario para mostrarle publicidad.

Permitir cookies en el login: Remember Me & Spring Boot

Spring Security nos permite crear una cookie con la siguiente composición:

- username: identifica el usuario que inició sesión.
- expirationTime: el tiempo de vida de la cookie creada.
- MD5 hash: contiene el valor hash de los dos atributos anteriores, username y expirationTime.

Para poder crear la cookie con Spring Security solo debemos indicar en el método configure lo siguiente:

```
.rememberMe().key("uniqueAndSecret")
```

Además debemos borrar la cookie JSESSIONID cuando cerremos sesión:

```
logout().deleteCookies("JSESSIONID")
```

En nuestro formulario deberíamos agregar un checkbox:

```
<div class="form-check" id="rememberme-group">
	<input type="checkbox" class="form-check-input" name="my-
	remember-me" id="remember-me">
	<label class="form-check-label" for="remember-me"
	style="color:white;">Recordarme</label>
</div>
```

Falta video

Live coding

- Agregar dependencia
- Crear package login y archivo AppUser (entity), con parámetros id, nombre, username, email, password, AppUsuario roles (un enum definido en otra clase)
- Crear clase AppUsuarioRoles como enum

Ejercicio
En esta instancia debemos agregar un login al proyecto, para ello debemos tener la entidad usuario y los siguientes campos:

Id: Long.
nombre: String.
userName: String.
email: String.
password: String.
UsuarioRole: Enum.
¿Qué debemos hacer?

Agregar solo un rol: user, dejar el usuario hardcodeado y crear con spring security un simple login.

Sesión 44. Autenticación de APIs basada en token JWT
Julio 6 de 2022

Encriptación con Hash.
Un algoritmo de hash se encarga de de generar un hash (bloque de caracteres de longitud fija) a partir de una cadena arbitraria de texto.

JSON Web Token
JWT es un estándar de código abierto basado en JSON para crear tokens de acceso que nos permiten securizar las API:

1. El cliente se autentica y garantiza su identidad haciendo una petición al servidor de autenticación. Esta puede ser realizada mediante un usuario y una contraseña, proveedores externos —como Google o Facebook— o mediante otros servicios —por ejemplo, LDAP, Active Directory, base de datos, etc.—.

2. Una vez que el servidor de autenticación garantiza la identidad del cliente, se genera un token de acceso (JWT).

3. El cliente usa ese token para acceder a los recursos protegidos que se publican mediante API.

4. En cada petición, el servidor desencripta el token, comprueba el HASH y valida si el cliente tiene permisos para acceder al recurso haciendo una petición al servidor de autorización.

Estructura del Token
Los tokens están compuestos por tres partes:

Header
Contiene el hash que se usa para encriptar el token. La propiedad alg indica el algoritmo usado para la firma y la propiedad typ define el tipo de token, en nuestro caso, JWT:

```
{
	"alg": "HS256",
	"typ": "JWT"
}
```

Payload
Contiene una serie de atributos clave:valor que se encriptan en el token.

```
{
	"id": "1",
	"username": "sergiodxa"
}
```

Propiedades estándares posibles:

- Creador (iss): Identifica quién creo el JW.
- Razón (sub): Identifica la razón del JWT, se puede usar para limitar su uso a ciertos casos.
- Tiempo de expiración (exp): Una fecha que sirve para verificar si el JWT está vencido y obligar al usuario a volver a autenticarse.
- No antes (nbf): Indica desde qué momento se va a empezar a aceptar un JWT.
- Creado (iat): Indica cuándo fue creado el JWT.

Signature
La firma del JWT se genera usando los anteriores dos campos en base64 y una key secreta —que solo se sepa en los servidores que creen o usen el JWT— para usar un algoritmo de encriptación.

```
key = 'secret'
unsignedToken = base64Encode(header) + ' . ' + base64Encode(payload)
signature = SHA256(key, unsignedToken)
token = unsignedToken + ' . ' + signature
```

Construcción de una API de generación de token

1. Agregar al POM las dependencias de jwt

```
<dependency>
	<groupId>io.jsonwebtoken</groupId>
	<artifactId>jjwt</artifactId>
	<version>0.9.1</version>
</dependency>
```

2. Crear una clase JWTUtil

```
@Component
public class JwtUtil {

	private String SECRET_KEY = "secret";

	public String extractUserName(String token) {
		return extractClaimUsername(token);
	}

	public Date extractExpiration(String token) {
		return extractClaimDate(token);
	}

	public Date extractClaimDate(String token){
		Claims claims = extractAllClaims(token);
```

```
        return claims.getExpiration();
    }

    public String extractClaimUsername(String token){
        Claims claims = extractAllClaims(token);
        return claims.getSubject();
    }

    private Claims extractAllClaims(String token) {
        return Jwts.parser().setSigningKey(SECRET_KEY)
        .parseClaimsJws(token).getBody();
    }

    public String generateToken(UserDetails userDetails) {
        Map<String, Object> claims = new HashMap<>();
        return createToken(claims, userDetails.getUsername());
    }

    private String createToken(Map<String, Object> claims, String subject) {
        // la librería Jwts tiene un método llamado builder que nos permite
        crear un token al que le pasaremos el usuario, le asignaremos una
        fecha de expiración y el algoritmo de encriptación.
        return Jwts.builder().setClaims(claims).setSubject(subject)
        .setIssuedAt(new Date(System.currentTimeMillis()))
        .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60
+60 * 10))
        .signWith(SignatureAlgorithm.HS256, SECRET_KEY).compact();
    }

    public Boolean validateToken(String token, UserDetails userDetails) {
        final String username = extractUserName(token);
        return (username.equals(userDetails.getUsername()) && !
        isTokenExpired(token));
    }

    private boolean isTokenExpired(String token) {
        return extractExpiration(token).before(new Date());
    }
}
```

3. Crear la clase SecurityConfiguration

```
@Configuration
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Autowired
    private MyUserDetailsService myUserDetailsService;
```

```
	@Autowired
	private JwtRequestFilter jwtRequestFilter;

	@Autowired
	private BCryptPasswordEncoder bCryptPasswordEncoder;

	@Override
	protected void configure(AuthenticationManagerBuilder auth) throws
	Exception {
		auth.userDetailsService(myUserDetailsService);
	}

	@Override
	protected void configure(HttpSecurity http) throws Exception {
		http.csrf().disable().authorizeRequests()
		.antMatchers("/authenticate").permitAll().anyRequest()
		.authenticated().and().sessionManagement()
		.sessionCreationPolicy(SessionCreationPolicy.STATELESS);
		http.addFilterBefore(jwtRequestFilter,
		UsernamePasswordAuthenticationFilter.class);
	}

	@Override
	@Bean
	public AuthenticationManager authenticationManagerBean() throws
	Exception {
		return super.authenticationManagerBean();
	}

	@Bean
	public DaoAuthenticationProvider daoAuthenticationProvider() {
		// Nos ayuda a configurar un método de encriptación, podemos ver
		cómo son seteados nuestros bCryptPasswordEncoder y
		myUserDetailsService.
		DaoAuthenticationProvider provider =
		new DaoAuthenticationProvider();
		provider.setPasswordEncoder(bCryptPasswordEncoder);
		provider.setUserDetailsService(myUserDetailsService);
		return provider;
	}
}
```

4. Agregar el JWTRequstFilter

```
@Component
public class JwtRequestFilter extends OncePerRequestFilter {

	@Autowired
	private UserDetailsService userDetailsService;
```

```
	@Autowired
	private JwtUtil jwtUtil;

	@Override
	protected void doFilterInternal(HttpServletRequest httpServletRequest,
	HttpServletResponse httpServletResponse, FilterChain filterChain) throws
	ServletException, IOException {

		final String authorizationHeader =
		httpServletRequest.getHeader("Authorization");

		String username= null;
		String jwt = null;
		if(authorizationHeader != null &&
		authorizationHeader.startsWith("Bearer")) {
			jwt = authorizationHeader.substring(7);
			username = jwtUtil.extractUserName(jwt);
		}

		if(username != null &&
		SecurityContextHolder.getContext()
		.getAuthentication() == null) {

			UserDetails userDetails =
			this.userDetailsService.loadUserByUsername(username);
			if(jwtUtil.validateToken(jwt, userDetails)) {
				UsernamePasswordAuthenticationToken
				usernamePasswordAuthenticationToken = new
				UsernamePasswordAuthenticationToken(userDetails,
				null, userDetails.getAuthorities());

				usernamePasswordAuthenticationToken.setDetails(new
				WebAuthenticationDetailsSource()
				.buildDetails(httpServletRequest));

				SecurityContextHolder.getContext()
				.setAuthentication(usernamePasswordAuthenticationTo
				ken);
			}
		}

		filterChain.doFilter(httpServletRequest, httpServletResponse);
	}
}
```

5. Agregar la clase PasswordEncoder

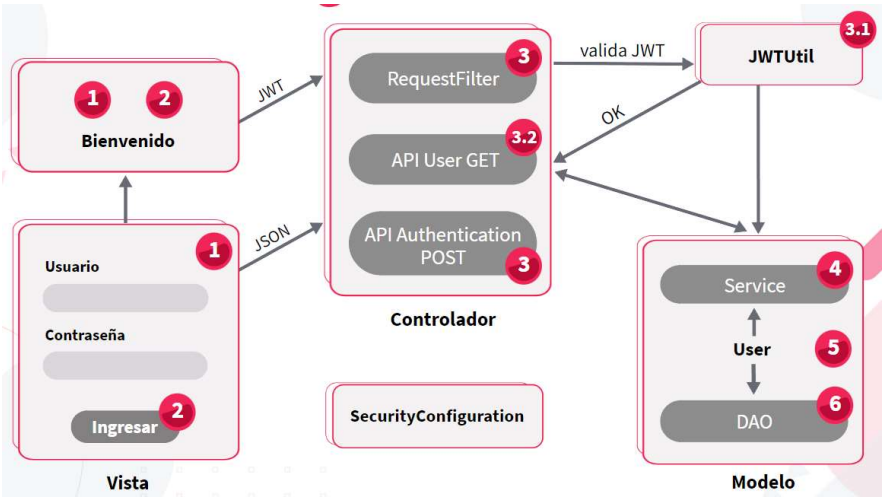
```
@Configuration
```

```
public class PasswordEncoder {
	@Bean
	public BCryptPasswordEncoder bCryptPasswordEncoder() {
		return new BCryptPasswordEncoder();
	}
}
```

6. Crear el Controller

```
@RestController
public class JwtController {

	@Autowired
	private AuthenticationManager authenticationManager;

	@Autowired
	private UserDetailsService userDetailsService;

	@Autowired
	private JwtUtil jwtUtil;

	@RequestMapping(value = "/authenticate", method =
	RequestMethod.POST)
	public ResponseEntity<?> createAuthenticationToken(@RequestBody
	AuthenticationRequest authenticationRequest) throws Exception{

		try {
			authenticationManager.authenticate(new
UsernamePasswordAuthenticationToken(authenticationRequest.getUsername(),
authenticationRequest.getPassword()));

		}catch (BadCredentialsException e) {
			throw new Exception("Incorrect", e);
		}

		final UserDetails userDetails =
userDetailsService.loadUserByUsername(authenticationRequest.getUsername()
);
		final String jwt = jwtUtil.generateToken(userDetails);
		return ResponseEntity.ok(new AuthenticationResponse((jwt)));
	}
}
```

Login con JWT



Los componentes que se necesitan para armar un login con JWT son:

HTML del login
Es la vista, es decir, la pantalla donde ingresaremos el usuario y contraseña a validar.

JavaScript del login
Es el encargado de invocar la API de autenticación que, dado un usuario y password (si estos son válidos), devolverá un token JWT. Este token JWT será guardado del lado cliente en el localStorage, para ser utilizado en la invocación del resto de los servicios que necesitan ese JWT para poder invocarlos.

API de autenticación
Es una API que tenemos que armar, la cual tendrá la lógica de validación de este usuario y contraseña. En nuestro caso, la lógica consistirá en buscar en una base de datos ese usuario y contraseña, y validar que sean iguales.

Service, User, DAO
Son todos los componentes que necesitamos para acceder a la base de datos.

JwtUtil
Es una clase que se encargará de generar el JWT cuando hacemos un login y también de validarlo cuando un servicio que es invocado tiene seguridad por JWT.

RequestFilter
Los filtros son clases que se utilizan para invocarse antes o después de un request o response. En este caso, queremos que ante cualquier request —si el mismo tiene un JWT—, sea validado utilizando la JwtUtil. Para poder programar estas clases debemos heredar de OncePerRequestFilter.

SecurityConfiguration
En esta clase configuraremos las URL que serán exceptuadas para poder ser invocadas sin JWT e instanciaremos el requestFilter.

Sesión 45. Integradora 14.
Julio 7 de 2022

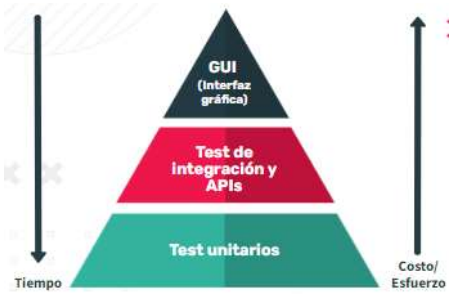
Sincrónico

Sesión 46. Pruebas de Integración
Julio 8 de 2022

Loss test de integración prueban la interacción entre las distintas partes del sistema para verificar que los componentes de una aplicación funcionan correctamente actuando en conjunto. Estos test cubren una mayor área del código, del que a veces no se tiene control —como librerías de terceros—, y comprueban, por ejemplo, que se envió un email, la conexión real con la base de datos, la conexión con otro web service, entre otros.

Por ejemplo, en una arquitectura de microservicios se utilizan típicamente para verificar las interacciones entre las distintas capas de código y componentes externos con los que se está integrando —otros microservicios, base de datos, caches, etc—.

Pirámide ideal de Testing
Introducida por Mike Cohn, propone qué tests se deben hacer según el momento del proyecto para ahorrar tiempos, costos y esfuerzos. El objetivo es poder anticiparse a los errores que se puedan producir en el momento indicado sin que se propaguen hacia arriba.



Beneficios de la prueba de integración:
- Se aegura de que todos los módulos de palicación estén bien integrados y funcionen juntos según lo esperado.
- Detecta problemas y conflictos interconectados para resolverlos antes de generar un problema mayor.
- Valida la funcionalidad, fiabilidad y estabilidad entre diferentes módulos.
- Detecta excepciones ignoradas para mejorar la calidad del código.
- Admite la canalización de CI/CD.

Tests unitarios	Tests de integración
Prueban una clase/unidad en solitario, es decir, testean una unidad de forma aislada.	Prueban los componentes del sistema trabajando juntos, es decir, testean la colaboración de múltiples unidades.
Fáciles de escribir y verificar.	El setup de las pruebas de integración puede ser complicado.
Toda dependencia externa es mockeada o eliminada, de ser necesario, en el escenario de prueba.	Requieren interacción con dependencias externas (por ejemplo, base de datos, hardware, etc.).
Se llevan a cabo desde el inicio del proyecto y luego se puede realizar en cualquier momento.	Deben realizarse después de la prueba unitaria y antes de la prueba del sistema.

Mantenimiento económico.	Mantenimiento costoso.
Los resultados dependen del código Java, solo verifican si cada pequeño fragmento de código está haciendo lo que se pretende que haga.	Los resultados también dependen de sistemas externos. Una prueba de integración fallida puede indicar también que el código continúa siendo correcto, pero el entorno ha cambiado.
Permiten identificar problemas en la funcionalidad de módulos individuales. No exponen errores de integración ni problemas en todo el sistema.	Permiten identificar los errores que surgen cuando diferentes módulos interactúan entre sí. Tienen un alcance más amplio.
Se ejecutan más rápido.	Su ejecución puede demorar mucho más tiempo.

Pruebas de Integración con SpringBoot: MockMVC.

MockMVC: Este framework provee una forma fácil de implementar tests de integración para aplicaciones web. Como alternativa a MockMvc se pueden utilizar frameworks como RestTemplate o Rest-assured. Para comenzar a testear endpoints declarados en un controller utilizando MockMVC, se requiere agregar las dependencias de spring-boot-starter-web y spring-boot-starter-test al archivo pom.xml.

```
// Levantar el contexto de la aplicación Spring
@SpringBootTest
// Inyecta un objeto MockMVC completamente configurado
@AutoConfigureMockMvc
public class HelloWorldIntegrationTest {
	// Inyecta la dependencia requerida
	@Autowired
	private MockMvc mockMvc;


	// Testear un método GET.
	// url: http://localhost:8080/sayHello
	// Salida esperada: { “id”: 1, “message”: “Hello World” }
	@Test
	public void testHelloWorldOutput() throws Exception {
		// perform() ejecuta la petición
		// print() imprime request y response por consola
		// andExpect() verifica que el status sea igual a 200
		// andReturn devuelve el objeto MvcResult completo
		MvcResult mvcResult =
		this.mockMvc.perform(MockMvcRequestBuilders.get("/sayHello"))
		.andDo(print()).andExpect(status().isOk())
		.andExpect(MockMvcResultMatchers.jsonPath("$.message").value("
		Hello World!"))
```

```
                andReturn();
                Assertions.assertEquals("application/json",
                mvcResult.getResponse().getContentType());
        }



        // Testear un método GET con Path Variable
        // url: http://localhost:8080/sayHello/George
        // Salida esperada: { “id”: 1, “message”: “Hello George” }
        @Test
        public void testHelloGeorgeOutput() throws Exception {

        this.mockMvc.perform(MockMvcRequestBuilders.get("/sayHello/{name}",
        "George"))
                .andDo(print()).andExpect(status().isOk())
                .andExpect(content().contentType("application/json"))
                .andExpect(MockMvcResultMatchers.jsonPath("$.message").value("
                Hello George!"));
        }



        // Testear un método GET con QueryParam
        // url: http://localhost:8080/sayHelloWithParam?name=George
        // Salida esperada: { “id”: 1, “message”: “Hello George” }
        @Test
        public void testHelloWithParamGeorgeOutput() throws Exception {

this.mockMvc.perform(MockMvcRequestBuilders.get("/sayHelloWithParam
")
                .param("name", "George"))
                .andDo(print()).andExpect(status().isOk())
                .andExpect(content().contentType("application/json"))
                .andExpect(MockMvcResultMatchers.jsonPath("$.message").value("
                Hello George!"));
        }



        // Testear un método POST
        // url: http://localhost:8080/sayHelloPost
        // Entrada: { “name”: “George” }
        // Salida esperada: { “id”: 1, “message”: “Hello George” }
        @Test
        public void testHelloPostGeorgeOutput() throws Exception {

                NameDTO payloadDTO = new NameDTO("George");
```

```
        // ObjectMapper se utiliza para convertir un objeto de tipo DTO en
        un String con su representación en JSON
        ObjectWriter writer = new ObjectMapper().
        configure(SerializationFeature.WRAP_ROOT_VALUE, false).
        writer().withDefaultPrettyPrinter();

        String payloadJson = writer.writeValueAsString(payloadDTO);

        // contentType especifica el formato del payload de entrada
        // content agrega el paylod en formato JSON al POST request
    this.mockMvc.perform(MockMvcRequestBuilders.post("/sayHelloPost")
        .contentType(MediaType.APPLICATION_JSON)
        .content(payloadJson))
        .andDo(print()).andExpect(status().isOk())
        .andExpect(content().contentType("application/json"))
        .andExpect(MockMvcResultMatchers.jsonPath("$.message").value("
        Hello George!"));
    }


    // Testear un método POST y verificar el contenido completo de la
respuesta
    // url: http://localhost:8080/sayHelloPost
    // Entrada: { "name": "George" }
    // Salida esperada: { "id": 1, "message": "Hello George" }
    @Test
    public void testHelloPostGeorgeOutput() throws Exception {
        NameDTO payloadDTO = new NameDTO("George");
        HelloDTO responseDTO = new HelloDTO(1, "Hello George!");

        // ObjectMapper se utiliza para convertir un objeto de tipo DTO en
        un String con su representación en JSON
        ObjectWriter writer = new ObjectMapper()
        .configure(SerializationFeature.WRAP_ROOT_VALUE, false)
        .writer();

        String payloadJson = writer.writeValueAsString(payloadDTO);
        String responseJson = writer.writeValueAsString(responseDTO);

        MvcResult response =
    this.mockMvc.perform(MockMvcRequestBuilders.post("/sayHelloPost")
        .contentType(MediaType.APPLICATION_JSON)
        .content(payloadJson))
        .andDo(print()).andExpect(status().isOk())
        .andExpect(content().contentType("application/json"))
        .andReturn();

        Assertions.assertEquals(responseJson,
    response.getResponse().getContentAsString());
```

```
    }
```

Anotaciones para tests de Integración

@WebMvcTest: Se utiliza para pruebas MockMVC. Deshabilita la autoconfiguración y permite una configuración determinada, por ejemplo, de Spring Security.
@MockBean: Permite la simulación de Beans.
@InjectMocks: Permite la inyección de Beans.
@ExtendWith: Usualmente se proporciona la extensión SpringExtension.class, inicializa el contexto de testeo Spring.
@ContextConfiguration: Permite cargar una clase de configuración custom.
@WebAppConfiguration: Permite cargar el contexto web de la aplicación.

Ejercicio:
Agregar tests de integración en el proyecto de clínica odontológica:

- Realizar test de integración para el controller de pacientes.
- Utilizando MockMvc, testear la búsqueda de pacientes por id.
- El test debe comprobar que el código de respuesta es 200 y que en el body retorna al paciente con datos.

Recordar cargar un paciente antes de ejecutar el test, para asegurar que haya información en la base de datos. Se pueden ingresar a los pacientes de la misma forma que se hizo en los tests unitarios.

Ayuda extra:
Para ejecutar los test sin que la seguridad de la api lo impida, agregar el parámetro addFilters = false en @AutoConfigureMockMvc

Ejemplo:

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc(addFilters = false)
public class IntegrationPacienteTest {
        //test1

        //test2
}
```

Live Coding:

Ejemplo:

```
@RunWith(SpringRunner.class)
@WebMvcTest(LenguajeController.class)
class IntegracionApplicationTest {

	@Autowired
	private MockMvc mvc;

	@Test
	void buscarTodosLosLenguajeAPI() throws Exception {
		mvc.perform(MockMvcRequestBuilders.get("/lenguajes"))
		.accept(MediaType.APPLICATION_JSON)
		.andDo(MockMvcResultHandlers.print())
		.andExpect(MockMvcResultMatchers.status().isOk());
	}

	@Test
	void BuscarLenguajesTipoAPI() throws Exception {
		mvc.perform(MockMvcRequestBuilders.get("/lenguajes/{tipo}",
		"Java"))
		.accept(MediaType.APPLICATION_JSON)
		.andDo(MockMvcResultHandlers.print())
		.andExpect(MockMvcResultMatchers.status().isOk());
	}

	@Test
	void BuscarLenguajesTipoAPINoExistente() throws Exception {
		mvc.perform(MockMvcRequestBuilders.get("/lenguajes/{tipo}",
		"Pepe"))
		.accept(MediaType.APPLICATION_JSON)
		.andDo(MockMvcResultHandlers.print())
		.andExpect(MockMvcResultMatchers.status().isNotFound());
	}
```

Sesión 47. Taller de Coding sobre nuestro Trabajo integrador
Julio 11 de 2022

Sesión 48. Taller de Coding y entrega
Julio 11 de 2022

Sesión 49. Documentación
Julio 12 de 2022

Swagger
Es un conjunto de herramientas de código abierto creadas en torno a la especificación OpenAPI, que facilita el diseño, construcción, documentación y consumo de API REST.

OpenAPI
Es un estándar para la descripción de APIs. La especificación open API define el formato en el que se deben describir, desarrollar, probar y documentar. Es decir, gracias a este estándar, la documentación de la mayoría de las APIs tendrá el mismo formato.

Swagger con SpringBoot
Swagger genera la documentación automáticamente a partir de los controllers, creando una interfaz web en donde se muestran todos los endpoints de la aplicación y los datos que reciben, y que además permite enviar request a la aplicación. Para utilizar Swagger con Spring Boot, se utilizará la librería SpringDoc.

Pasos:

1. Agregar la dependencia en el POM:

```
<dependency>
	<groupId>org.springdoc</groupId>
	<artifactId>springdoc-openapi-ui</artifactId>
	<version>1.5.2</version>
</dependency>
```

2. Ejecutar la aplicación e ir a la url de Swagger:
http://localhost:8080/swagger-ui.html

3. Ingresar a la Url de la documentación: /api-docs.

Requisitos en los campos:

Es posible agregar requisitos en la documentación de Swagger con anotaciones en las entidades, con @NotNull, @NotBlank, @Size, @Min, and @Max.

```
public class Book {

	private long id;

	@NotBlank
```

```
        @Size(min = 0, max = 20)
        private String title;

        @NotBlank
        @Size(min = 0, max = 30)
        private String author;
}
```

Agregar la descripción a los endpoints:

```
@Operacion(summary = "Registrar un nuevo producto")
@PostMapping
public Producto guardarProducto(@RequestBody Producto p) {
        return productoService.guardar(p);
}
```

Ejercicio
Para el proyecto "Fixture Web'' de la clase 38 realizar lo siguiente:

- Agregar la librería SpringDoc al POM.
- Dirigirnos a la URL configurada para ver la interfaz de Swagger.
- Levantar el proyecto y probar la API desde esa misma interfaz.
- Agregar anotaciones necesarias para indicar que los campos "id" y "nombre", dentro de la entidad torneo, no pueden ser null.

Sesión 50. Despliegue de API en Docker
Julio 13 de 2022

Para crear un ejecutable de una aplicación Java, solo se requiere ejecutar maven install, y se creará un archivo .jar que se podrá hacer funcionar en cualquier ordenador que tenga Java instalado. Sin embargo, esto es ineficiente, ya que puede traer problemas de compatibilidad. Docker ofrece la posibilidad de abstraerse de estos problemas al hacer correr las aplicaciones en diferentes computadoras.

Docker
En la industria del software, se creó el concepto de contenedores para solucionar problemas de construcción, distribución y ejecución de las aplicaciones. Docker e suna plataforma que permite crear, probar e implementar aplicaciones rápidamente.

Ventajas:
- Entregar código con rapidez.
- Estandarizar operaciones.
- Transferir código con facilidad.
- Mejorar el uso de recursos.
- Ahorrar dinero.
- Obtener un objeto que se puede ejecutar de manera fiable en cualquier lugar.

Dockerfile
Los archivos Dockerfile nos permiten crear nuestras imágenes. Podemos almacenarlos en la carpeta principal de nuestro proyecto y es importante que el nombre sea “Dockerfile”.

Ejemplo:

FROM openjdk:8-jdk-alpine // Definimos la versión de Java que usará la imagen para ejecutar nuestra aplicación.
ARG JAR_FILE = mi- proyecto/target/spring-boot-web.jar // Guardamos en una variable la dirección del archivo .jar generado con Maven.
COPY ${JAR_FILE} app.jar // Copiamos el archivo dentro de la imagen, y lo nombramos app.JAR.
ENTRYPOINT ["java","-jar","/app.jar"] // Ejecutamos “java -jar /app.jar” , este comando ejecuta el archivo .jar, por lo tanto, levanta nuestra aplicación dentro del contenedor.

Docker Image
Luego de crear el Dockerfile, abrimos una consola en nuestra computadora y ejecutamos los siguientes comandos:

docker build -t myapp:1.0.

Con este comando, creamos la imagen y, además, le ponemos un tag, en este ejemplo, “myapp:1.0”.

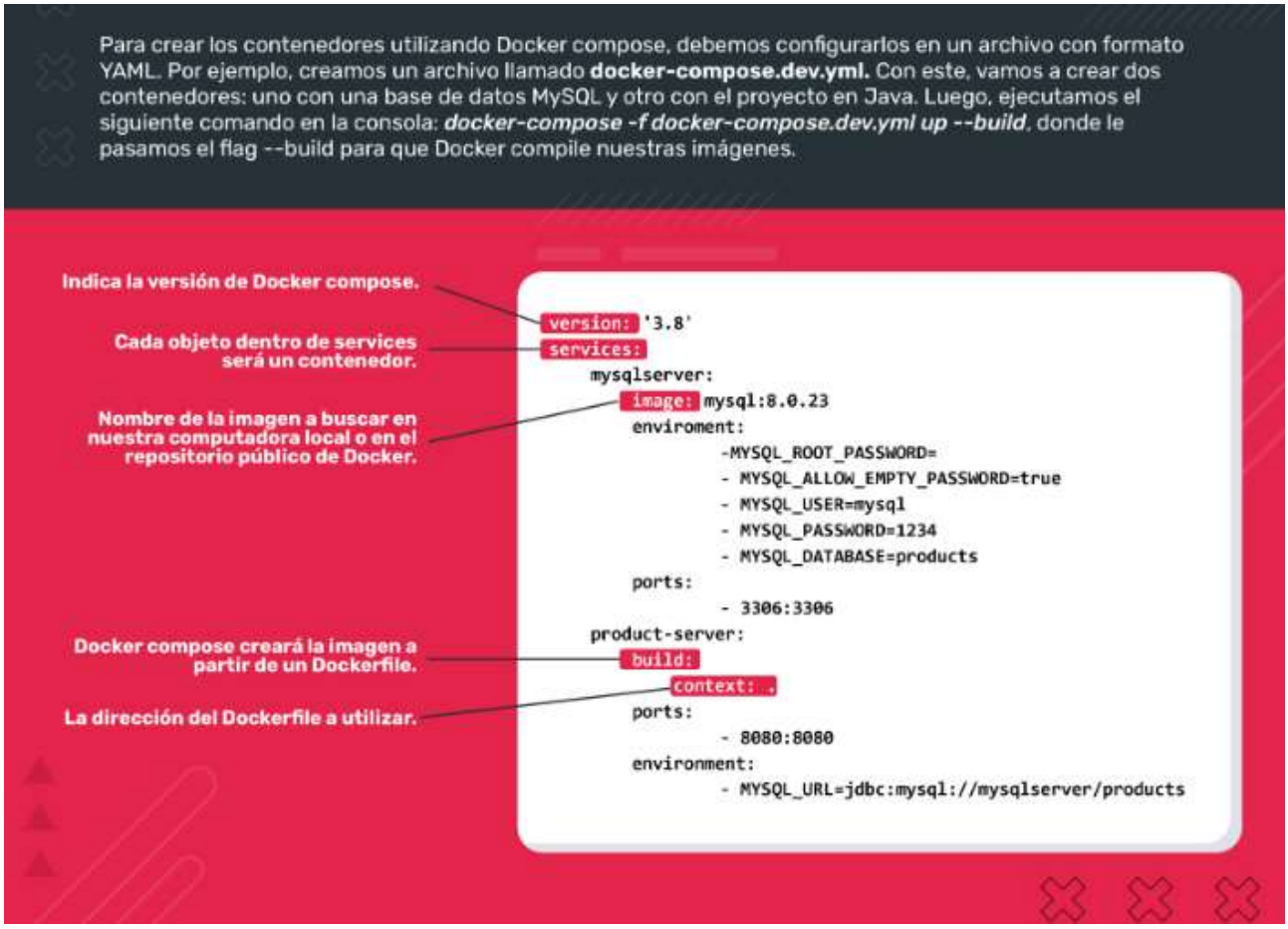
Docker container
Una vez creada la imagen, podemos ejecutarla dentro de un contenedor:

docker run -p 8080:8080 myapp:1.0.

En este caso, le indicamos que tiene que ejecutar un contenedor con la imagen que tenga el tag “myapp:1.0” y que exponga el puerto 8080 del contenedor y que lo relacione con el puerto 8080 de nuestra computadora. De esta manera, si vamos a http://localhost:8080, podremos ver la aplicación.

Docker Compose
Los comandos de Docker y los Dockerfiles son muy útiles cuando queremos crear contenedores individuales y no necesitamos que tengan comunicación con el resto de los contenedores. Sin embargo, si a la hora de desplegar nuestra aplicación necesitamos más de un contenedor —por ejemplo un contenedor para la aplicación en Java y otro contenedor para la base de datos o muchos más contenedores si trabajamos con una arquitectura de microservicios— esto provoca que se vuelva todo mucho más difícil de gestionar a medida que escala nuestro sistema.

Para resolver esto, Docker proporciona Docker Compose, esta herramienta la utilizaremos cuando tenemos que administrar múltiples contenedores, ya que nos permitirá, por ejemplo, iniciar todos los contenedores al mismo tiempo, detenerlos, etc., como si estuviéramos manipulando uno solo.



Live Coding

- Crear Dockerfile:

// Dockerfile base
FROM adoptopenjdk/openjdk11:alpine-jre
ARG JAR_FILE=target/paqueteria-0.0.1-SNAPSHOT.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT [“java”, “-jar”, “app.jar”] // equivale al comando “java -jar app.jar”
EXPOSE 8080:8080

- Crear el archivo .jar con Maven, en el panel de Maven en IntelliJ. Alternativamente, se puede hacer en línea de comandos con el comando **mvn install**.

- docker build --tag paqueteria:1.0 . // El punto al final indica que el archivo se encuentra en el directorio actual.

- docker run --publish 8080:8080 paqueteria:1.0

Este comando ejecuta la aplicación. Durante la ejecución, aparece un error, debido a que no esposible hacer conexión a la base de datos. Esto se debe a

que el contenedor no tiene MongoDB. Se requiere crear una imagen de Mongo y vincular los contenedores por medio de docker compose.

Docker-compose.yml:
// En la carpeta raíz del proyecto, la misma donde se encuentra el dockerfile.

```
version: '3.8'
services:
    mongo-db:
        image: mongo // Indica la imagen que debe descargar
        ports:
            -27017:27017
        environment:
            // Las variables de entorno vienen de la documentación de Mongo. No se deben modificar. (hub.docker.com/_/mongo).
            MONGO_INITDB_ROOT_USERNAME: admin
            MONGO_INITDB_ROOT_PASSWORD: adminpass
            MONGO_INITDB_DATABASE: paqueteria
        // La base de datos se crea en el memento en que se crean datos (es el funcionamiento de mongo). Para modificar esto se puede agregar un volumen con un archivo js en el que se crea una colección; es el archivo init-mongo.js. Esto es opcional.
        #volumes:
            #- ./init-mongo.js:/docker-entrypoint-initdb.d/init-mongo.js:ro
    paqueteria:
        // En este caso no se especifica la imagen, sino que se pide a docker-compose que la construya.
        build:
            context: .
        ports:
            - 8080:8080
        environment:
            MONGO_HOST: mongo-db // Es el nombre del servicio
            MONGO_PORT: 27017
            MONGO_DB: paqueteria
            MONGO_USER: admin
            MONGO_PASS: adminpass
```

// Archivo init-mongo.js (no es requerido):
db.createCollection("paquetes")

Para que las variables de entorno sean leídas, deben especificarse en el application.properties:

// Archivo application.properties:
// Al definir las variables de esta manera, la aplicación utiliza las variables que corresponda para cada entorno: en producción, toma los valores especificados en el archivo .yml, pero durante el desarrollo toma los valores por defecto, que son los que aparecen luego de los dos puntos.

spring.data.mongodb.host=${MONGO_HOST:localhost}
spring.data.mongodb.port=${MONGO_HOST:27017}
spring.data.mongodb.database=${MONGO_DB:paquetes}
spring.data.mongodb.username=${MONGO_USER:admin}
spring.data.mongodb.password=${MONGO_PASS:adminpass}

- docker-compose up --build
// Este comando ejecuta el docker-compose

- En caso de que haya un error y se requiere corregir el código, es necesario generar de nuevo el .jar; para ello ejecutar en Maven clean y a continuación install.

- Crear una conexión a mongo, puede ser a través de Robo3T. Se observa que no se han creado paquetes.

- Lanzar una petición, por ej. con Swagger. La aplicación está corriendo correctamente. Al refrescar Robo3T, se puede observar que se creó la base de datos paqueteria, con la colección paquetes.

- El comando docker ps lista los containers disponibles: paqueteria y mongo.

Ejercicio
Trabajar con el proyecto "Fixture web" de la clase 49. Ejecutar la aplicación en Java y la base de datos de Mongo con Docker:

- Crear Dockerfile para construir la imagen de la aplicación en Java.
- Crear Docker-compose para ejecutar la imagen y la imagen de Mongo.
- Por último, probar el correcto funcionamiento con Postman o desde Swagger.

Sesión 51. Integradora 15
Julio 14 de 2022

Sesión 52. Taller de Coding sobre Pruebas de Integración
Julio 15 de 2022

Sesión 53. Estamos llegando al final…
Julio 15 de 2022

Sesión 54. ¡Llegamos al final!
Julio 15 de 2022