# Passos realizados na aula de Spring

## Aula 01

Validação do ambiente

- Java
- Eclipse IDE
- Maven

Criar um projeto no Spring Initializr
Acessar a página: https://start.spring.io
Criar um projeto e fazer o download.
Maven, Java 11, adicionar as dependências: spring-boot-devtools
Dezipar e importar no Eclipse.

Executar o projeto
Executar a classe com o método main.
Ver o log.
Adicionar a dependência do Starter Web (consultar no site do Spring Initializr)
Acessar a URL: http://localhost:8080
Interromper a execução

## Aula 02

Criando um serviço Rest
Implementar um RestController com RequestMapping("/hello") com um método para responder um GetMapping (“/hello/v1”).

```
@RestController
@RequestMapping("/hello")
public class HelloController {

	@GetMapping("/v1")
	public String hello() {
		return "Hello v1!";
	}
```

Executar o projeto.
Acessar a URL no navegador correspondente ao serviço criado.

Criar um método hello v2 para testar outra URL:

Acessar a URL no navegador correspondente ao serviço criado.

Fazer o endpoint Rest responder JSON:
Criar a entidade Message com 2 atributos (code e message)

```
public class Message {
```

```
	private int code;
	private String message;

	//criar o construtor com os 2 parâmetros
	//criar os sets e gets
```

Criar um método na classe controller para retornar um objeto de mensagem (v3).

```
	@GetMapping("/v3")
	public Message helloMessage() {
		return new Message(1, "Hello from v3", MessagePriority.LOW);
	}
```

Acessar a URL no navegador correspondente ao serviço criado.

# Aula 03

Salvando dados no Banco de dados H2
Incluir a dependência do starter-data-jpa

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

Habilitando o H2 no pom.xml

```
	<dependency>
		<groupId>com.h2database</groupId>
		<artifactId>h2</artifactId>
		<scope>runtime</scope>
	</dependency>
```

Configurando o H2 no arquivo application.properties:

```
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console

#spring.datasource.url=jdbc:h2:file:~/test
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.username=sa
spring.datasource.password=
spring.datasource.driver-class-name=org.h2.Driver

spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

Colocar as anotações do JPA na classe Message:

```
@Entity
public class Message implements Serializable {

	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private int code;
	private String message;

	public Message() {}
	public Message(int code, String message) {
		super();
		this.code = code;
		this.message = message;
	}
	//implementar o hashCode e equals
```

# Aula 04

Salvar dados no H2:
- Criar o MessageRepository:

```
@Repository
public interface MessageRepository extends JpaRepository<Message, Integer>{
}
```

- Injetar o repositório na classe application (main):

```
	@Autowired
	private MessageRepository msgRepo;
```

- Fazer a classe Application implements CommandLineRunner
- criar o método run na classe de Application

```
public void run(String... args) throws Exception {
```

- Instanciar objetos Message
- Persistir os objetos “message’.

```
	Message msg = new Message(1, "Hello from v4");
	msgRepo.save(msg);
```

Executar e acessar o H2 via browser e consultar os dados na tabela.

# Aula 05

Implementar um método no controller para retornar uma mensagem pelo id:

```
@GetMapping("/{id}")
	public Message getMessageById(@PathVariable Integer id) {
		Optional<Message> obj = msgRepo.findById(id);
		if(obj.isPresent()) {
			return obj.get();
```

```
            }
            return null;
      }
```
Testar o endpoint no Postman e analisar como o endpoint responde quando o objeto não é encontrado.

Criar uma exceção personalizada:
```
public class ObjectNotFoundException extends RuntimeException {
      public ObjectNotFoundException(String message) {
            super(message);
      }
}
```

Lançar a exceção caso o objeto não for encontrado no método getMessageById no controller.
```
            if(obj.isPresent()) {
                  return obj.get();
            }
            throw new ObjectNotFoundException("Mensagem não encontrada: " +
id);
      }
```

Analisar como o endpoint responde quando ocorre erro (id not found, etc.)

Melhorar o tratamento de erros do endpoint:
```
//imutável - imutable
public class StandardError {

      private String message;
      private Integer status;
      private Date date;

      public StandardError(String message, Integer status, Date date) {
            super();
            this.message = message;
            this.status = status;
            this.date = date;
      }
      //implementar apenas os gets
```

Implementar o Exception Handler (controller advice):

```
@ControllerAdvice
public class ExceptionHandler {

      @ExceptionHandler(ObjectNotFoundException.class)
      public ResponseEntity<StandardError>
```

```
handleObjectNotFound(ObjectNotFoundException ex, HttpServletRequest req) {

		StandardError error = new StandardError(
				ex.getMessage(),
				HttpStatus.NOT_FOUND.value(),
				new Date());

		return ResponseEntity.status(HttpStatus.NOT_FOUND).body(error);
	}
}
```

Melhorar o endpoint para pegar um objeto pelo ID para usar lambda function junto com o Optional.

Acessar o endpoint via Postman e analisar os resultados.

# Aula 06

Separando a estrutura de dados interna (banco ou outras classes) do sistema da estrutura de dados utilizada no JSON:
Criar a classe:

```
public class MessageDTO {

	private Integer code;
	private String name;

	public MessageDTO(Message message) {
		this.code = message.getCode();
		this.name = message.getName();
	}
	//criar sets and gets
```

Atualizar o controller:

```
	@GetMapping()
	@ResponseStatus(HttpStatus.OK)
	public List<MessageDTO> getAllMessages() {
		return msgRepo.findAll()
			.stream()
			.map(m → new MessageDTO(m))
			.collect(Collectors.toList());
	}

	@GetMapping("/{id}")
	public MessageDTO getMessageById(@PathVariable Integer id) {
		Optional<Message> obj = msgRepo.findById(id);

		Message msg = obj.orElseThrow(() -> new
```

```
ObjectNotFoundException("Mensagem não encontrada: " + id));
            return new  MessageDTO(msg);
      }
```

Acessar o endpoint via Postman e analisar os resultados.

Criar um Service e transferir o acesso ao banco para ele:

```
@Service
public class MessageService {

      @Autowired
      private MessageRepository repo;

      public List<Message> findAll() {}
      public Message findById(Integer id) {}
```

Atualizar o controller:

```
      @Autowired
      private MessageService service;

      @GetMapping("/{id}")
            public ResponseEntity<MessageDTO> find(@PathVariable Integer id) {
            Message obj = service.findById(id);
            return ResponseEntity.ok().body(new MessageDTO(obj));
      }
```

Acessar o endpoint via Postman e analisar os resultados.

# Aula 07

Implementar o **POST** para salvar o registro no banco:

No Service:

```
      public void createMessage(Message message) {}

      public Message toMessage (MessageDTO message) {
            return new Message( message .getId(), message.getName());
      }
```

No controller:

```
      @PostMapping("")
      @ResponseStatus(HttpStatus.CREATED)
      public void createMessage(@RequestBody MessageDTO messageDTO) {
            service.createMessage(service.toMessage(messageDTO));
      }
```

Acessar o endpoint via Postman e analisar os resultados.

Implementar o **PUT** para atualizar o registro no banco:
No Service:

```
	public void updateMessage(Message message, Integer id) {
		if (id == null || message == null || id.equals(message .getId())) {
			throw new MessageException("Invalid message id.");
		}
		Message existingObj = findById(id);
		updateData(existingObj, message);
		repo.save(message);
	}
	private void updateData( Message existingObj, Message obj) {
		existingObj.setName(obj.getName());
	}
```

No Controller:

```
	@PutMapping("/{id}")
	@ResponseStatus(HttpStatus.NO_CONTENT)
	public void updateMessage(@RequestBody Message dto, @PathVariable
Integer id) {
		service.updateMessage(service.toMessage(dto), id);
	}
```

Acessar o endpoint via Postman e analisar os resultados.

# Aula 08

Atualizar a lógica do UPDATE e tratar a exceção.

Implementar o **DELETE** para atualizar o registro no banco:

No Service:

```
public void deleteMessage(Integer id) {
	if (id == null) {
		throw new MessageException("Message id can not be null.");
	}
	Message obj = findById(id);
	try {
		repo.delete(obj);
	} catch (DataIntegrityViolationException e) {
		throw new MessageException("Can not delete a Message with
dependencies/constraints.");
```

```
    }
}
```

No Controller:

```
    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void deleteMessage(@PathVariable Integer id) {
        service.deleteMessage(id);
    }
```

Acessar o endpoint via Postman e analisar os resultados.

# Aula 09

Fazendo a validação dos objetos que vem do cliente:
Incluir a dependência no pom.xml:

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

Incluir as anotações de validação no atributo da classe:

```
public class MessageDTO {

    private Integer id;

    @NotNull(message = "Name can not be null.")
    @NotEmpty(message = "Name can not be empty.")
    @Size(min = 5, max = 80, message = "The size must be between 5 and 80.")
    private String name;
```

Exercício:
Criar um atributo na classe Message assim:

```
private Integer priority;
```

Fazer as modificações necessárias para incluir este atributo no projeto.
Alterar a Entity, DTO, Service, e main

Incluir também:

```
private Date createdAt;
```

no DTO:

```
@JsonFormat(pattern = "dd/MM/yyyy HH:mm:ss")
private LocalDateTime dateCreated;
```

**Extra:**

Criar um ValidationError:

```
public class ValidationError extends StandardError {

    @Getter
    private List<FieldMessage> errors = new ArrayList<>();

    public ValidationError(Integer status, String message, Long timestamp) {
        super(status, message, timestamp);
    }

    public void addError(String fieldName, String message){
        this.errors.add(new FieldMessage(fieldName, message));
    }
}
```

No controller advice:

```
    @ExceptionHandler(MethodArgumentNotValidException.class)
    @ResponseBody
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    public ValidationError objectNotFound(MethodArgumentNotValidException e,
HttpServletRequest request) {

        ValidationError error = new
ValidationError(HttpStatus.BAD_REQUEST.value(), "Validation error.",
System.currentTimeMillis());
        e.getBindingResult().getFieldErrors().stream()
                .forEach(err -> error.addError(err.getField(),
err.getDefaultMessage()));
        return error;
    }
```

Incluir a anotação @Valid no parâmetro nos métodos create e update do controller.

# Aula 10

Criar o Enum

```
public enum MessagePriority {
      LOW(10), MEDIUN(20), HIGH(30);
      private int code;
```

```
	private MessagePriority(int code) {
		this.code = code;
	}
	/**
	 * Coverte de Integer para MessagePriority
	 */
	public static MessagePriority getPriority(Integer code) {
		//implementar do jeito que quiser
		return Stream.of(MessagePriority.values())
			.filter(t -> t.getCode() == code)
			.findFirst()
			.orElseThrow(() -> new InvalidDataException("Prioridade
inválida: " + code));
	}
	public int getCode() {
		return code;
	}
}
```

Criar o atributo na classe Message (atualizar o construtor e criar set e get):
`private MessagePriority priority;`

Atualizar todos os lugares afetados por esta alteração.
Atualizar também os métodos que convertem de Entity para DTO e vice versa.
Criar a classe MessagePriorityConverter para fazer a conversão de número para enum:

```
public class MessagePriorityConverter implements
AttributeConverter<MessagePriority, Integer>{

	@Override
	public Integer convertToDatabaseColumn(MessagePriority attribute) {
		return attribute.getCode();
	}
	@Override
	public MessagePriority convertToEntityAttribute(Integer dbData) {
		return MessagePriority.getPriority(dbData);
	}
}
```