**NLW JOURNEY Parte 1**

[**Plenn.er**](http://Plenn.er)**:**

**Spring Initializr:** ajuda na criação do springBoot

**Repository:** fazem a comunicação com o banco de dados

**API Rest:** armazenam informações do sistema, mas nada da seção do usuário, por exemplo quando que ele logo

**Inicialização do projeto**

Acessar o Spring initializr: <https://start.spring.io/>

**Atifact:** planner → nome do projeto

**Project: Maven →** para as dependências

**Flyway Migration →** controlar versões de banco de dados

**Lombok →** geração de código **Boilerplate**(evitar códigos de copia e cola)

**H2 Database** → fica na memória

**Configurar banco de dados:**

Dentro dos **resources**, devemos entrar também no [**application.properties**](http://application.properties)

Podemos passar vários bancos de dados, por isso declaramos ele(s)

Onde devemos colocar as configurações dentro dela, para que o Spring data JPA se concete com ele(H2 Database), que devmos colocar algumas propriedades nele: 1° spring.datasource.url=jdbc:h2:mem:planner

Onde passamos a url do banco, ele é um jdbc, o banco que vamos usar que é o H2, ele é em memória, e o nome que damos a ela.

2° spring.datasource.driver-class-name=org.h2.Driver

Onde passamos o driver para ele conseguir fazer a conexão de maneira correta

3° spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

Onde indicamos qual é a plataforma do dialeto dele

4° e 5° spring.datasource.password=

`spring.datasource.username=sa`

Onde passamos a senha e o usuário, e essas informações ja vem de default(padrão do H2 database)

6° Ir no main e testar para ver se funciona

**Arquivos Migrations:** são scripts SQL que vão rodar no nosso banco, \*\*\*\*conceito que vai armazenar o versionamento dos esquemas do nosso banco de dados, ou seja, apresenta uma mudança no meu estado atual do banco de dados. Por exemplo:

* criar uma tabela nova
* alterar uma tabela, remover campo, ou adicionar campo
* instalação de driver
* inserção em massa, de dados default

Ele também salva no banco de dados, verifica se ja foi executado o script se não for ele executa, caso contrário ele pula

**OBS:** O versionamento tem que ser MAIÚSCULO. Exemplo: V1\_\_create-table-trips.sql

**RANDOM\_UIID() =** criam ids aleatórios, pre preenche o id com o valor, onde passamos tambem o tipo que é UUID

**Entidades:** representação em forma de objeto que esta dentro do meu banco, usa então o @Entity, para indicar que aquela classe é uma entidade que faz objetificação ao banco de dados

Para identificar uma chave primaria usamos duas **anotações,** POR EXEMPLO: @Id → mostra que é uma chave primaria @GeneratedValue(strategy = \*\*GenerationType\*\*.\*\*\*AUTO\*\*\*) → fala que vai ser gerado automaticamente private UUID id;

@Column(nullable = false) não precisa passar o nome, pois ele é o mesmo nome que tem no banco, ja reconhece, nome único

private String destination;

**Para criarmos nosso Repository POR EXEMPLO:**

public interface TripsRepository extends JpaRepository<Trip, UUID> {}

Criamos uma classe que estende JpaRepository, passando o tipo e qual o id;

**Para criarmos um post, devemos criar um controller**

Passamos duas anotações:

@RestController → Inclui as anotações *@Controller* e *@ResponseBody* e, como resultado, simplifica a implementação do controlador:

@RequestMapping("/trips") → ele fica escutando uma chamada, onde quando fizer uma requisição passando o **/trips** ele irá entrar na classe e executar o que estiver dentro

**Objetos do tipo record:** são objetos que não podem ser alterados assim que foram criados, eles também geram os getter e setter automaticamente

no **@RequestBody →** passamos uma classe que ele deve pegar, então criamos uma nova classe onde vamos mapear cada unidade. POR EXEMPLO:

public record TripRequestPayload (String destination, String starts\_at, String ends\_at, List<String> emails\_to\_invite, String owener\_email, String owner\_name){}

**@Autowired →** injeção de dependência

@GetMapping("/{id}") → eu pego através da url do banco, o id criado, e passo os detalhes dele

**Optional →** pode ou não, essa é a lógica

**Dentro do meu controller**

@GetMapping("/{id}")public ResponseEntity<Trip> getTripDetails(@PathVariable UUID id){ Optional<Trip> trip = this.repository.findById(id); return trip.map(ResponseEntity::ok).orElseGet(() -> ResponseEntity.notFound().build());} Ele tenta encontrar um id dentro do meu trip, caso ele encontre ele monta uma reposta do tipo ResponseEntity no meu body, com status ok, caso ele não encontre ele vai mostrar com status não encontrado

@PutMapping → é usado para modificar/atualizar um recurso onde o cliente envia dados que atualizam todo o recurso.