

Desenvolvimento de uma API REST com Spring Boot e JPA-Hibernate para gerenciar quadrinhos de usuários

Que projeto legal, não é mesmo? Pelo título já deu vontade de começar a desenvolver, mas antes disso vamos compreender alguns conceitos necessários para compreensão do nosso projeto. A principal ferramenta deste projeto é o **Spring Boot** que é um Framework Java para a construção de aplicações, principalmente por quem adota uma arquitetura orientada a microsserviços, com objetivo de facilitar o processo em aplicações Java e, conseqüentemente, proporcionar mais agilidade durante o desenvolvimento de *software*. Assim, a escolha do **Spring Boot** facilita a utilização de tecnologias de acesso a dados, bancos de dados relacionais e não relacionais, serviços de dados baseados em nuvem, dentre outros.

Por vários anos, um dos maiores problemas de utilizar a abordagem orientada a objetos era a comunicação com o banco de dados relacional. No entanto, devido ao fato de o **Spring Boot** possuir vários módulos, com configurações rápidas, você consegue, por exemplo, disponibilizar uma aplicação baseada no Spring MVC, utilizando o **Hibernate+ JPA**, para a construção da camada de persistência na abstração do banco de dados. Ficou curioso para compreender mais sobre esse Framework e especificação?! Nesse tutorial explicarei a vocês o passo a passo da implementação de uma *Application Programming Interface* (API) para efetuar o gerenciamento de quadrinhos (*comics*) de usuários. O código desse projeto está disponível no GitHub no seguinte endereço:

<https://github.com/slmelo82/crud-spring-comics>

Para o desenvolvimento desse projeto vamos aprender várias ferramentas importantes descritas a seguir.

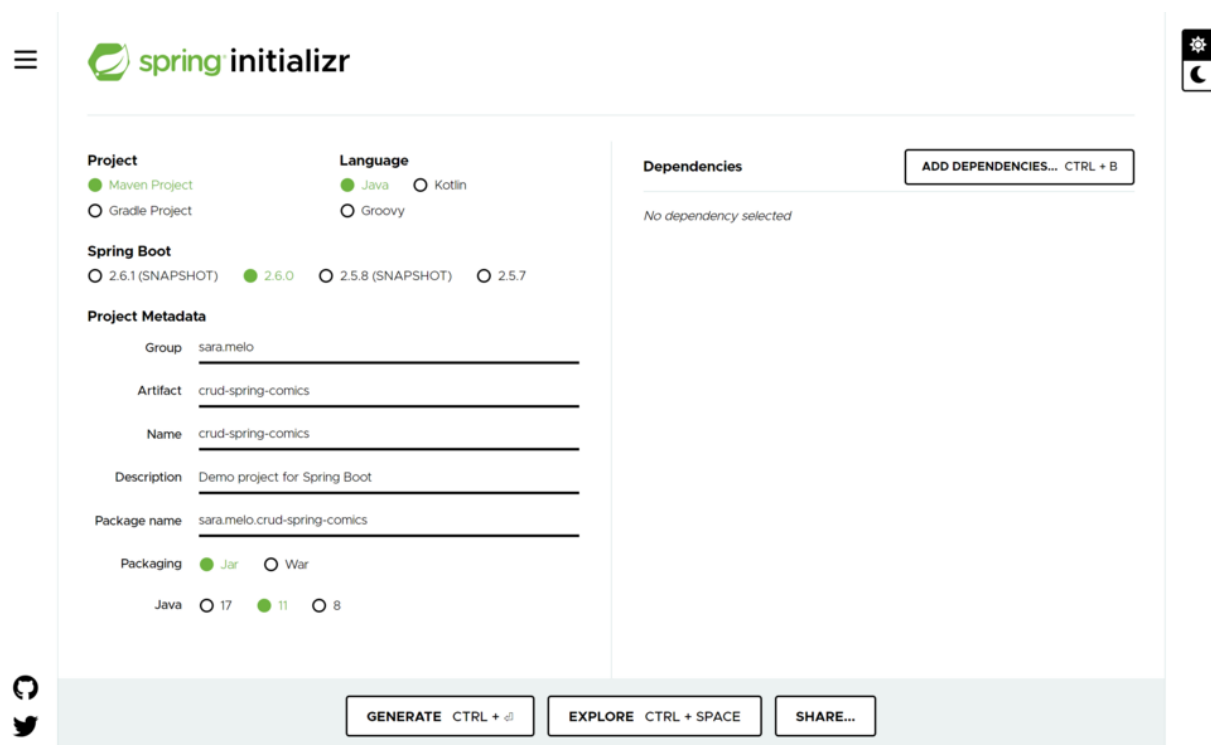
1. **O Spring Boot:** o *framework* para desenvolver a aplicação;
2. **Maven:** como gerenciador de dependências;
3. **Banco de Dados H2:** um Banco de Dados (BD) em memória do Java que possibilita testar e fazer as operações de um cadastro completo de uma entidade, ou seja, o CRUD (create, retrieve, update, delete);
4. **Postman:** para testar as requisições;

O que você precisa ter instalado?

- Um editor de texto ou IDE de sua preferência;
- JDK 1.11 ou superior;
- [Postman](#)

Criação do projeto

Agora que já temos uma visão geral vamos colocar a mão na massa! Para isto, primeiramente, precisamos criar um projeto. Existem diversas maneiras de se criar um projeto Spring Boot Java, nesse tutorial o projeto será criado diretamente no site do [spring.io](#). Nosso projeto será do tipo Maven na Linguagem Java na versão 11, com o Spring Boot na versão 2.6.0 basta selecionar essas opções e, em seguida, preencher os campos relacionados ao projeto conforme mostra a Figura abaixo.



The screenshot shows the Spring Initializr web interface. The form is divided into several sections:

- Project:** ☒ Maven Project, ☐ Gradle Project
- Language:** ☒ Java, ☐ Kotlin, ☐ Groovy
- Spring Boot:** ☐ 2.6.1 (SNAPSHOT), ☒ 2.6.0, ☐ 2.5.8 (SNAPSHOT), ☐ 2.5.7
- Project Metadata:**
 - Group: sara.melo
 - Artifact: crud-spring-comics
 - Name: crud-spring-comics
 - Description: Demo project for Spring Boot
 - Package name: sara.melo.crud-spring-comics
- Packaging:** ☒ Jar, ☐ War
- Java:** ☐ 17, ☒ 11, ☐ 8
- Dependencies:** No dependency selected. Button: ADD DEPENDENCIES... CTRL + B

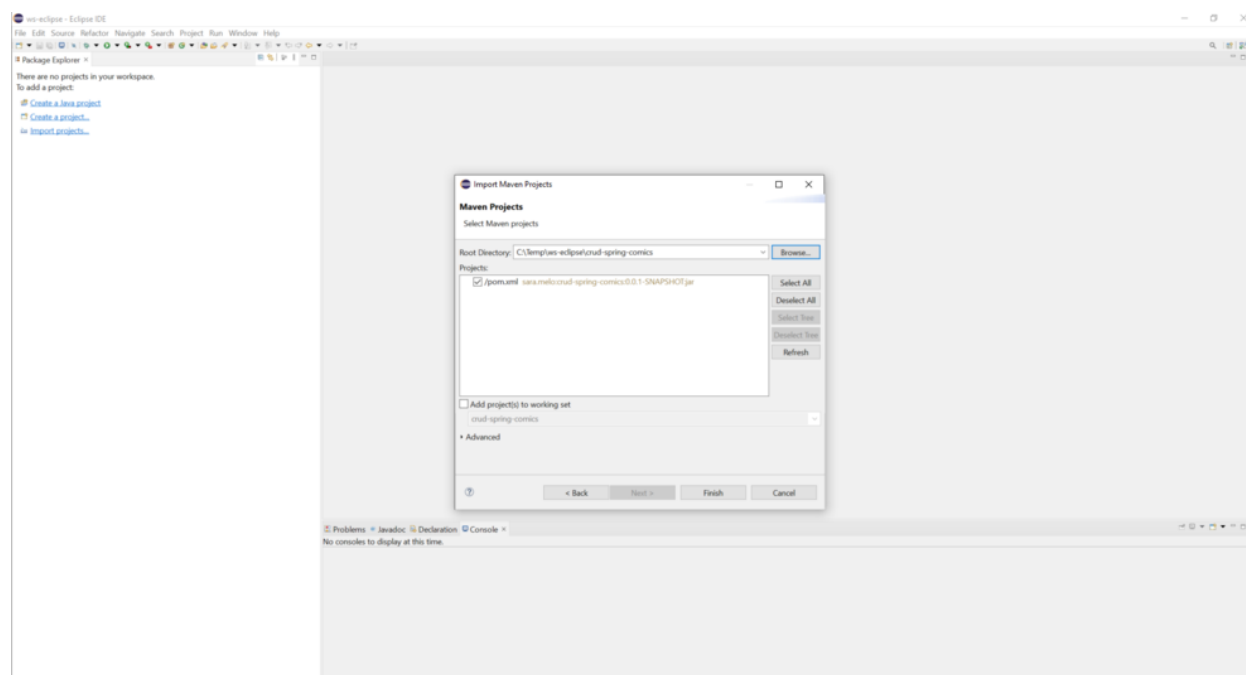
At the bottom, there are three buttons: GENERATE CTRL + G, EXPLORE CTRL + SPACE, and SHARE... There are also social media icons (GitHub, Twitter) on the left and a settings icon on the right.

Calma que ainda não acabou, além disso, eu adicionei as dependências básicas relacionadas ao projeto clicando no botão "**Add Dependencies**" e selecionei: **Spring Web** (para construção da aplicação web incluindo o REST e MVC) **Spring Data JPA** (Para persistir os dados) **H2 Database** (para utilizar o banco de dados em memória para testes), **Spring Boot DevTools** (para fornecer recursos adicionais), **Validation** (Para

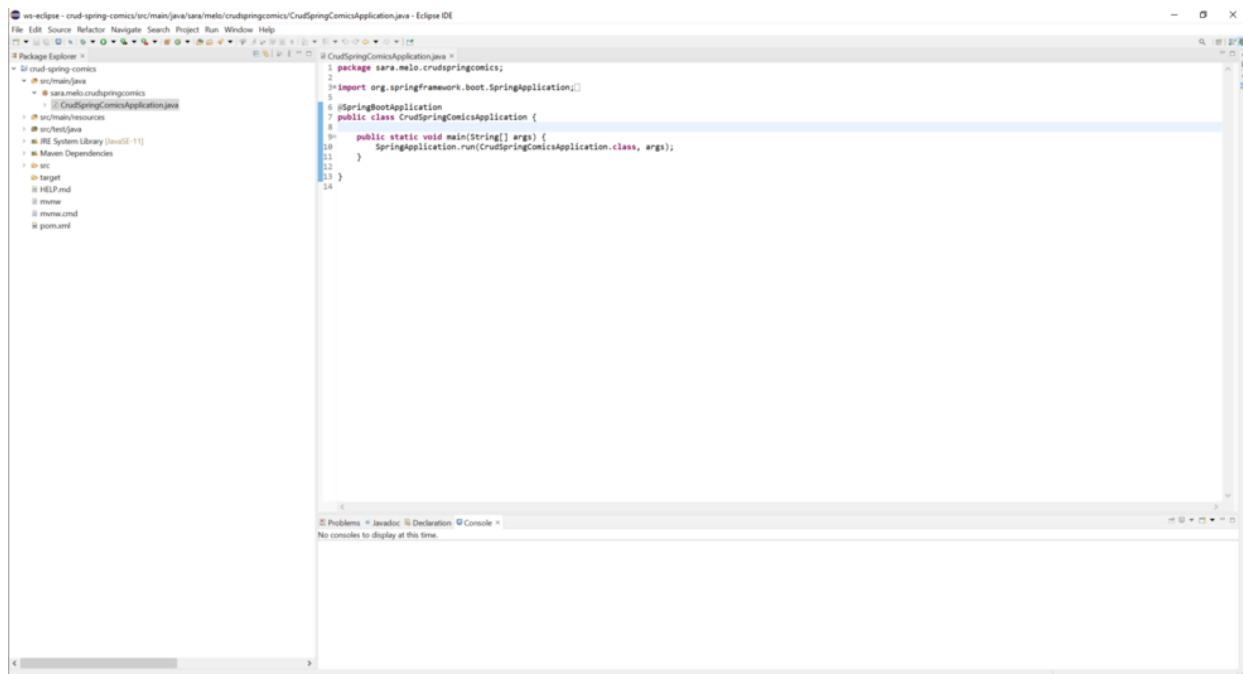
inicializar a validação do Java com o validação do Hibernate) e **OpenFeign** (para implementar clients HTTPs em Java de forma fácil e prática, não sendo necessário escrever nenhum código para chamar o serviço, a não ser uma definição de interface). Desse modo, a figura abaixo exhibe todas as seleções e dependências necessárias para execução do nosso projeto.

The screenshot shows the Spring Initializr web interface. On the left, under 'Project', 'Maven Project' is selected. Under 'Language', 'Java' is selected. Under 'Spring Boot', version '2.6.1' is selected. The 'Project Metadata' section includes fields for Group (sara.melo), Artifact (crud-spring-comics), Name (crud-spring-comics), Description (Demo project for Spring Boot), and Package name (sara.melo.crud-spring-comics). Packaging is set to 'Jar' and Java version is '11'. On the right, the 'Dependencies' section lists several dependencies: 'Spring Web' (WEB), 'Spring Data JPA' (SQL), 'H2 Database' (SQL), 'Spring Boot DevTools' (DEVELOPER TOOLS), 'Validation' (I/O), and 'OpenFeign' (SPRING CLOUD ROUTING). Each dependency has a brief description.

Após o *download* das dependências, crie uma *workspace* no eclipse e descompacte o .zip nela. A seguir, é só importar o projeto no eclipse (como Maven Project) e temos algo parecido com isso:



Ao finalizar a importação (Finish) terminamos a etapa de criação do projeto e agora temos nosso projeto importado na IDE-Eclipse como exemplo a seguir.



Observa-se na Figura acima que temos a estrutura básica de projeto do Spring Boot, ao expandir a pasta **src/main/java** temos o pacote do projeto e, expandindo novamente, tem-se a classe principal do projeto que tem o método main do Java, como também o primeiro comando da sua aplicação Spring Boot rodar. Próximo passo é testar o projeto.

Testando o projeto

Para testar o projeto, eu criei a classe **UserController** como um controlador de usuário responsável por controlar as requisições, ou seja, indicar quem deve receber as requisições e para quem deve responde-las. Em seguida, inserimos a anotação **@RestController**. Após o controlador criado, vamos mapear a rota **"/users"** para o controlador. Para fazer isso, basta adicionar a anotação **@RequestMapping(value = "/users")**, ficando dessa forma:

O que é a anotação @RestController no Spring?

No Framework Spring, um Controller é uma classe responsável pela preparação de um modelo de Map com dados a serem exibidos pela view e pela escolha da view correta. Basicamente ele é o responsável por controlar as requisições indicando quem deve receber as requisições para quem deve responde-las.

```
package sara.melo.crudspringcomics.controllers;
```

```
import org.springframework.web.bind.annotation.GetMapping;
```

```
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping(value = "/users")
public class UserController {

    @GetMapping
    public String getHello() {
        return "Opa, estou no ar!!!";
    }
}
```

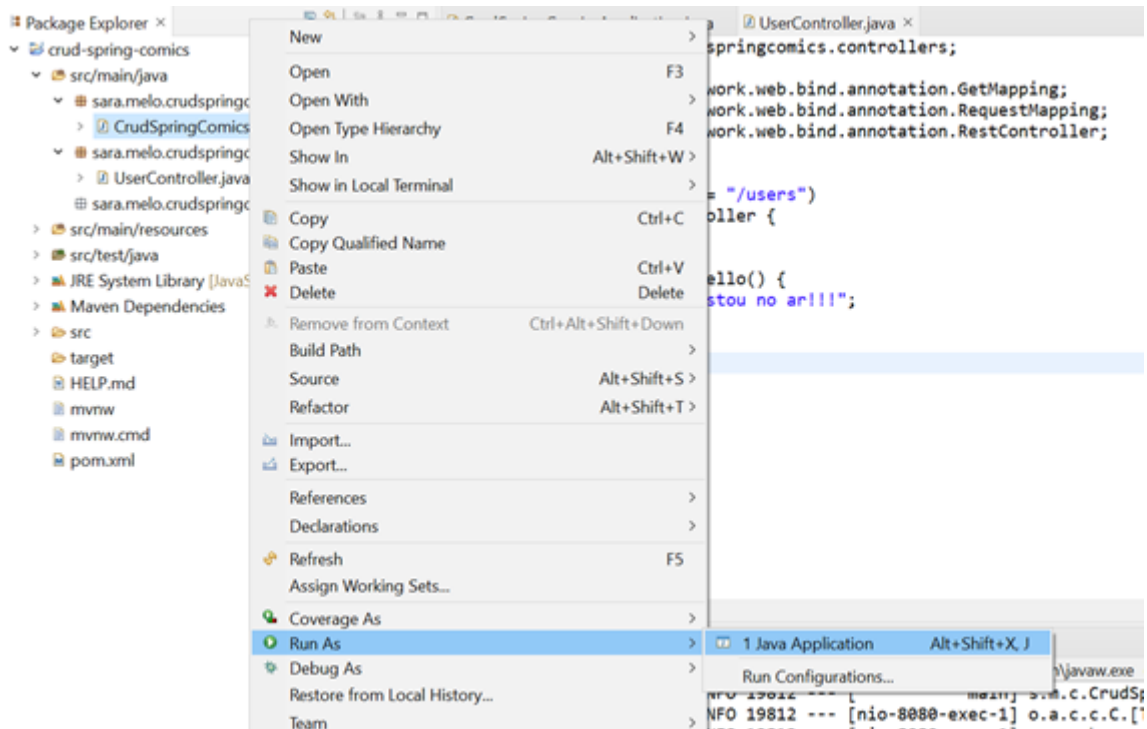
Ao observar o código acima nota-se que adicionamos a anotação **@GetMapping** para definir que o método `getHello()` é uma solicitação HTTP do formato GET para verificar o funcionamento da API.

Quais tipos de métodos de solicitação HTTP e anotações posso utilizar no Spring?

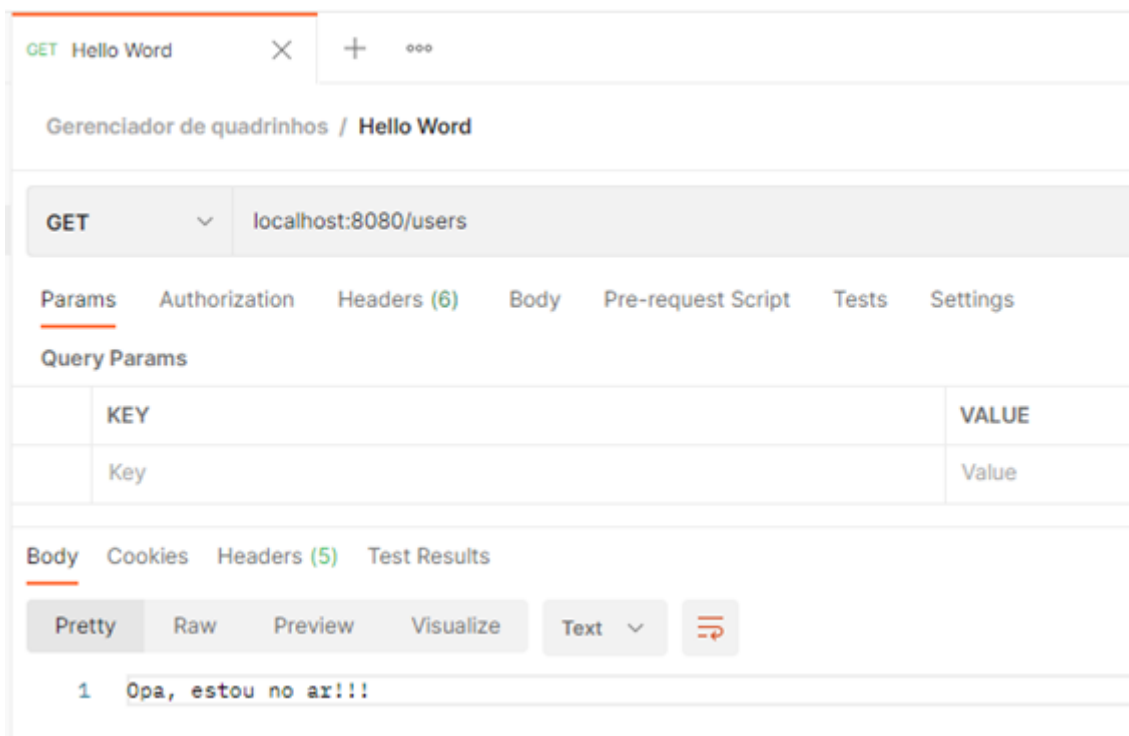
Atualmente, o Spring suporta cinco tipos de anotações embutidas para lidar com diferentes tipos de métodos de solicitação HTTP de entrada que são GET, POST, PUT, DELETE e PATCH. Essas anotações são:

- **@GetMapping** - shortcut for `@RequestMapping(method = RequestMethod.GET)`
- **@PostMapping** - shortcut for `@RequestMapping(method = RequestMethod.POST)`
- **@PutMapping** - shortcut for `@RequestMapping(method = RequestMethod.PUT)`
- **@DeleteMapping** - shortcut for `@RequestMapping(method = RequestMethod.DELETE)`
- **@PatchMapping** - shortcut for `@RequestMapping(method = RequestMethod.PATCH)`

Assim, basta inicializar a API, rodando a classe `CrudSpringComicsApplication` como Java Application.



Para testarmos nossa API, utilizamos o POSTMAN para consultar a serviço <http://localhost:8080/users> com METHOD GET, segue exemplo de sucesso:



Prontinho, nossa API está no ar, observa-se que como não fiz nenhuma configuração a API, então ela subiu na porta padrão (8080).

Construção do cadastro de usuários

A primeira etapa para construção do nosso projeto é criar uma classe, denominada **User** para representar o usuário e, logo após implementar os atributos básicos. Esta classe possui os seguintes atributos:

- Nome
- E-mail
- CPF
- Data de nascimento

As restrições da entidade devem ser respeitadas:

- O Nome é obrigatório
- O E-mail é obrigatório
- O CPF é obrigatório
- A data de nascimento é obrigatória
- O e-mail deve ser um e-mail válido (ex: formato usuario@host.com...)
- O CPF deve ser válido
- A data de nascimento deve estar no formato dd/mm/aaaa
- O e-mail deve ser único (apenas um usuário cadastrado pode possuir um determinado endereço e-mail)
- CPF deve ser único (apenas um usuário cadastrado pode possuir um determinado CPF)

Com posse dessas informações, crie a classe **User** conforme o seguinte código:

```
package sara.melo.crudspringcomics.models;
```

```
public class User {  
  
    private Integer id;  
    private String nome;  
    private String email;  
    private String cpf;  
    private Date nascimento;  
  
    public Integer getId() {  
        return id;  
    }  
    public void setId(Integer id) {
```

```

        this.id = id;
    }
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getCpf() {
        return cpf;
    }
    public void setCpf(String cpf) {
        this.cpf = cpf;
    }
    public Date getNascimento() {
        return nascimento;
    }
    public void setNascimento(Date nascimento) {
        this.nascimento = nascimento;
    }
}

```

Com a classe “User” e seus GETTERS e SETTERS criados, aqui temos algumas anotações do JPA, sendo a primeira delas uma **Entity** a qual indicará que esta classe será uma entidade do nosso banco de dados. A seguir, temos as anotações **Id** e **GeneratedValue**, sendo a primeira notação é responsável por especificar qual atributo será a nossa chave primária e, a segunda anotação serve para gerar os valores automaticamente, você consegue definir o tipo de estratégia de incremento, que no caso optei por AUTO.

```

package sara.melo.crudspringcomics.models;

import java.util.Date;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.validation.constraints.Email;
import javax.validation.constraints.NotBlank;

```



```

import javax.validation.constraints.NotNull;

import sara.melo.crudspringcomics.controllers.validation.Cpf;

@Entity
@Table(name="users",
uniqueConstraints={@UniqueConstraint(columnNames = {"cpf" , "email"})})
public class User {

    @GeneratedValue(strategy = GenerationType.AUTO)
    @Id
    private Integer id;

    @NotBlank(message = "O Nome é obrigatório!")
    private String nome;

    @NotBlank(message = "O E-mail é obrigatório!")
    @Email(message = "Por favor, informe um e-mail válido!")
    private String email;

    @Cpf
    @NotBlank(message = "O CPF é obrigatório!")
    private String cpf;

    @NotNull(message = "A data de nascimento é obrigatória!")
    private Date nascimento;

    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getCpf() {

```

```

        return cpf;
    }
    public void setCpf(String cpf) {
        this.cpf = cpf;
    }
    public Date getNascimento() {
        return nascimento;
    }
    public void setNascimento(Date nascimento) {
        this.nascimento = nascimento;
    }
}

```

Respeitando as restrições, os atributos cpf e email devem ter cadastros únicos, para isto basta adicionar seguinte anotação:

```

@Table(name="users",
uniqueConstraints={@UniqueConstraint(columnNames = {"cpf" , "email"})})

```

Para validação (nome, email, cpf), utilizamos a anotação **@NotBlank** com o parâmetro (name="Mensagem para apresentar o erro") e importamos a dependencia:

javax.validation.constraints.NotBlank e a anotação **@NotNull**(message = "A data de nascimento é obrigatória!") para o atributo nascimento, pois ele é do formato Date.

Para validação da entidade email, utilizaremos a anotação **@Email**(message = "Por favor, informe um e-mail válido!") e importamos a dependencia:

javax.validation.constraints.Email.

Para validar a entidade cpf foi criado a anotação **@Cpf** conforme pode ser observado no código a seguir.

```

package sara.melo.crudspringcomics.controllers.validation;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;

@Documented
@Constraint(validatedBy = CpfValidator.class)

```

```

@Target( { ElementType.METHOD, ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
public @interface Cpf {

    String message() default "O CPF deve ser válido!";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};

}

```

Na anotação @Cpf é definida a mensagem que será devolvida para o usuário e também adicionamos @Constraint(validatedBy = CpfValidator.class) que é classe que contém a lógica para executar a validação. Assim, é necessário criar a classe **CpfValidator** que contém a regra de validação:

```

package sara.melo.crudspringcomics.controllers.validation;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class CpfValidator implements ConstraintValidator<Cpf, String>{

    private final int[] PESO_CPF = { 11, 10, 9, 8, 7, 6, 5, 4, 3, 2 };

    @Override
    public boolean isValid(String cpf, ConstraintValidatorContext context)

        String cpfSomenteDigitos = cpf.replaceAll("\\D", "");

        if ((cpfSomenteDigitos == null)
            || (cpfSomenteDigitos.length() != 11)
            || cpfSomenteDigitos.equals("00000000000")
            || cpfSomenteDigitos.equals("11111111111")
            || cpfSomenteDigitos.equals("22222222222")
            || cpfSomenteDigitos.equals("33333333333")
            || cpfSomenteDigitos.equals("44444444444")
            || cpfSomenteDigitos.equals("55555555555")
            || cpfSomenteDigitos.equals("66666666666")
            || cpfSomenteDigitos.equals("77777777777")
            || cpfSomenteDigitos.equals("88888888888")
            || cpfSomenteDigitos.equals("99999999999"))
        {
            return false;
        }

}

```

```

Integer digito1 = calcularDigito(cpfSomenteDigitos.substring(0, 9), PESO_CP

```

```

Integer digito2 = calcularDigito(cpfSomenteDigitos.substring(0, 9)
+ digito1, PESO_CPF);
return cpfSomenteDigitos.equals(cpfSomenteDigitos.substring(0, 9)
+ digito1.toString() + digito2.toString());
    }

    private int calcularDigito(String str, int[] peso) {
        int soma = 0;
        for (int indice = str.length() - 1, digito; indice >= 0; indice--)
            digito = Integer.parseInt(str.substring(indice, indice + 1));
            soma += digito * peso[peso.length - str.length() + indice];
        }
        soma = 11 - soma % 11;
        return soma > 9 ? 0 : soma;
    }
}

```

A primeira coisa a se notar nessa classe é que ela implementa a interface do `javax.validation.ConstraintValidator<A extends Annotation, T>` que recebe uma anotação como parâmetro. E aqui executamos o algoritmo para validarmos se um CPF é válido ou não. Com isso nós podemos adicionar essa anotação ao atributo `cpf`.

```

@Cpf
@NotBlank(message = "O CPF é obrigatório!")
private String cpf;

```

Precisamos também, fazer a validação da entidade **nascimento** que deve estar no formato `dd/mm/aaaa`. Para isso, deveremos utilizar a anotação **@JsonFormat(pattern="dd/MM/yyyy")**.

Agora criaremos um CRUD (criar, ler, atualizar, deletar). Primeiramente, criaremos o repositório **UserRepository** para a entidade `User`, a qual será a interface responsável pelo acesso a dados e estender a interface `JpaRepository`. Exemplo:

```

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import sara.melo.crudspringcomics.models.User;

@Repository
public interface UserRepository extends JpaRepository<User, Integer> {

}

```

Feito isso, voltaremos ao controlador **UserController** que criamos anteriormente para injeção da dependência **UserRepository** para isso, utilizamos a anotação **@Autowired**. Assim sendo, é possível criar, alterar, remover e listar usuários do banco de dados.

```
@RestController
@RequestMapping(value = "/users")
public class UserController {

    @Autowired
    private UserRepository userRepository;

}
```

Agora criaremos os métodos **save**, **getAll**, **getById**, **deleteById** e **update** que formam o CRUD de Usuários. Importamos a anotação **@ResponseBody** em todos os métodos para definir o corpo, o status e os cabeçalhos de uma resposta HTTP. Também podemos observar que utilizamos a anotação **@Valid** nos métodos **save** e **update** para que ocorra as validações dos dados de entrada que definimos em nossa entidade **User**.

```
@RestController
@RequestMapping(value = "/users")
public class UserController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping
    //Retorna uma lista de todos os usuários
    public ResponseEntity<List<User>> getAll() {
        List<User> users = new ArrayList<>(); // Cria uma lista
        // Busca todos os usuários no banco de dados
        users = userRepository.findAll();
        // Retorna status 201 com a lista de todos usuários no banco
        return new ResponseEntity<>(users, HttpStatus.OK);
    }

    @GetMapping(path =("/{id}")
    //Retorna um unico usuário, de acordo com o id
    public ResponseEntity<Optional<User>> getById(@PathVariable Integer id)
        Optional<User> user;
        try {
            // Busca o usuário no banco de dados
            user = userRepository.findById(id);
            if (user.isEmpty()) {
```

```

        // Não encontrou o usuário, retorna status 404;
        return new ResponseEntity<Optional<User>>(HttpStatus.NOT_FOUND)
    } else {
        // Retorna status 200 com os dados do usuário em questão
        return new ResponseEntity<Optional<User>>(user, HttpStatus.OK);
    }

    } catch (NoSuchElementException nsee) {
        // Não encontrou o usuário, retorna status 400;
        return new ResponseEntity<Optional<User>>(HttpStatus.BAD_REQUEST);
    }
}

@PostMapping
public ResponseEntity<User> save(@RequestBody @Valid User user) {
    // Salva usuário no banco de dados
    userRepository.save(user);
    // Retorna status 201 com os dados do usuário criado;
    return new ResponseEntity<>(user, HttpStatus.CREATED);
}

@DeleteMapping(path =("/{id}")
//Deleta um unico usuário, de acordo com o id
public ResponseEntity<Optional<User>> deleteById(@PathVariable Integer id)
    try {
        // Deleta o usuário no banco de dados
        userRepository.deleteById(id);
        // Retorna status 200 , usuário deletado com sucesso
        return new ResponseEntity<>(HttpStatus.OK);
    } catch (NoSuchElementException nsee) {
        // Não encontrou o usuário, retorna status 400;
        return new ResponseEntity<Optional<User>>(HttpStatus.BAD_REQUEST);
    }
}

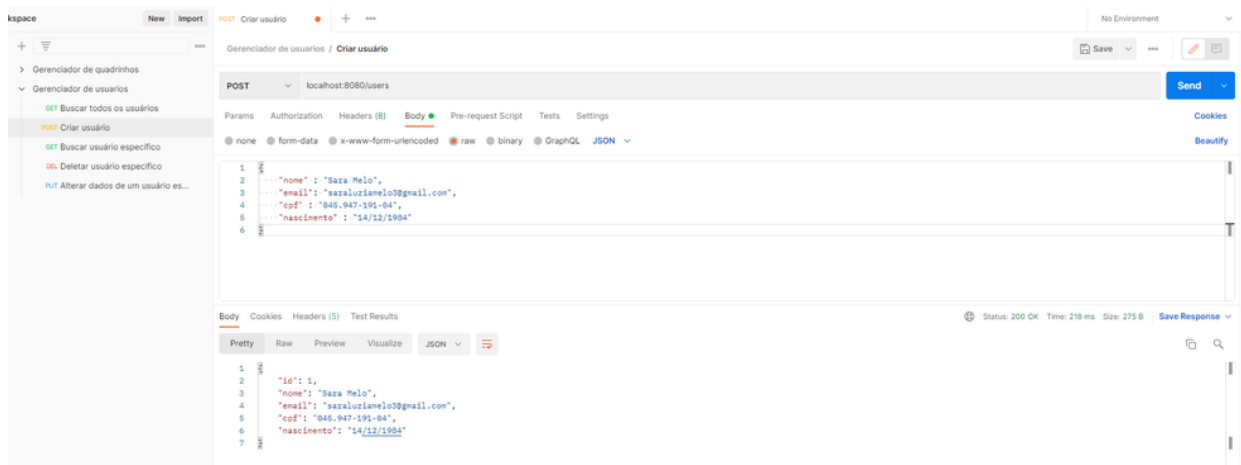
@PutMapping(path =("/{id}")
//Atualiza os dados de um usuário; Se não encontrar, retorna erro 404
public ResponseEntity<User> update(@PathVariable Integer id,
    @RequestBody @Valid User newUser) {
    return userRepository.findById(id).map(user -> {
        user.setNome(newUser.getNome());
        user.setEmail(newUser.getEmail());
        user.setNascimento(newUser.getNascimento());
        user.setCpf(newUser.getCpf());
        User userUpdated = userRepository.save(user);
        return new ResponseEntity<>(userUpdated, HttpStatus.CREATED);
    }).orElse(ResponseEntity.notFound().build());
}

```

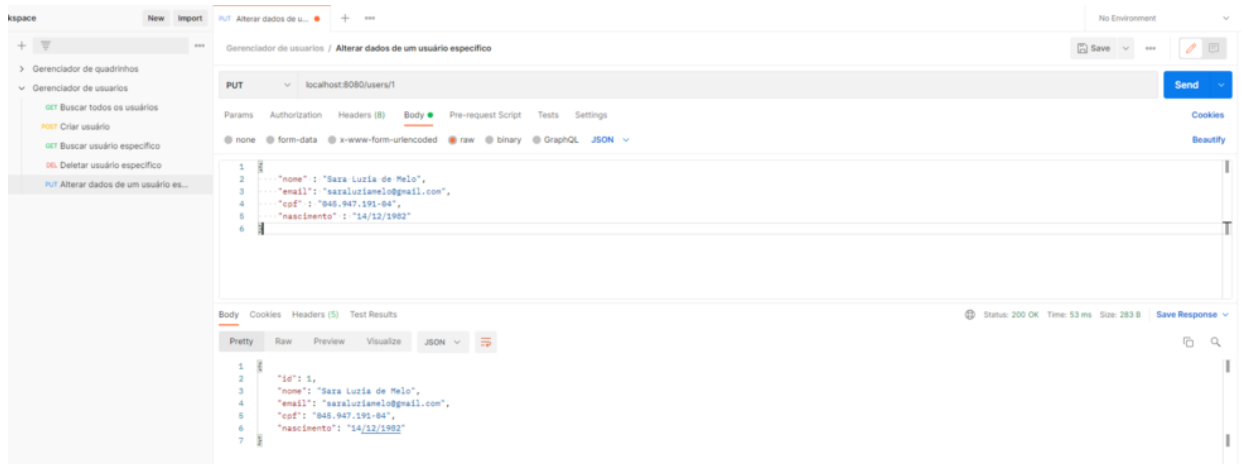
```
}  
  
}
```

Agora utilizaremos o Postman para fazer requisições para testar o nosso CRUD.

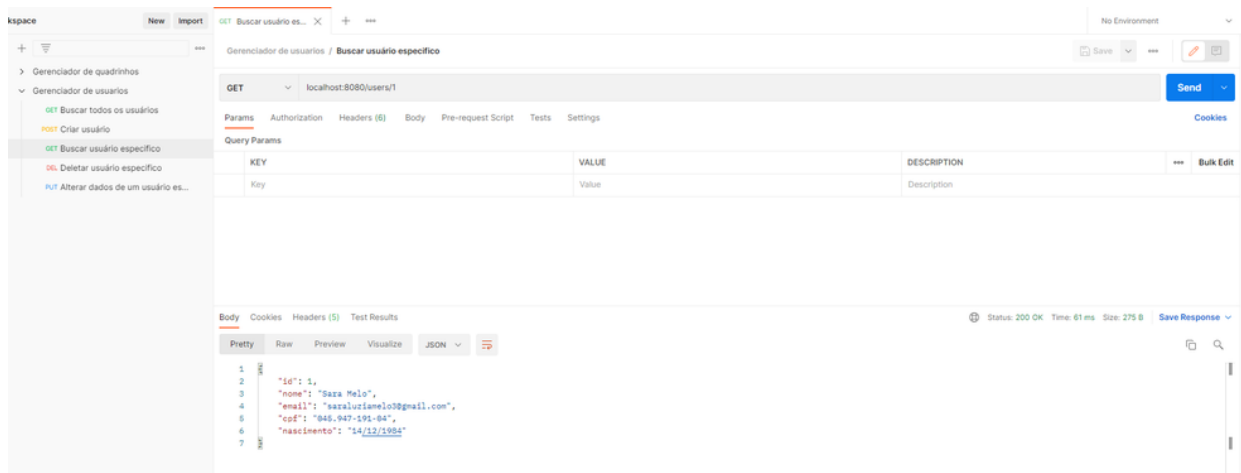
- Criando um usuário



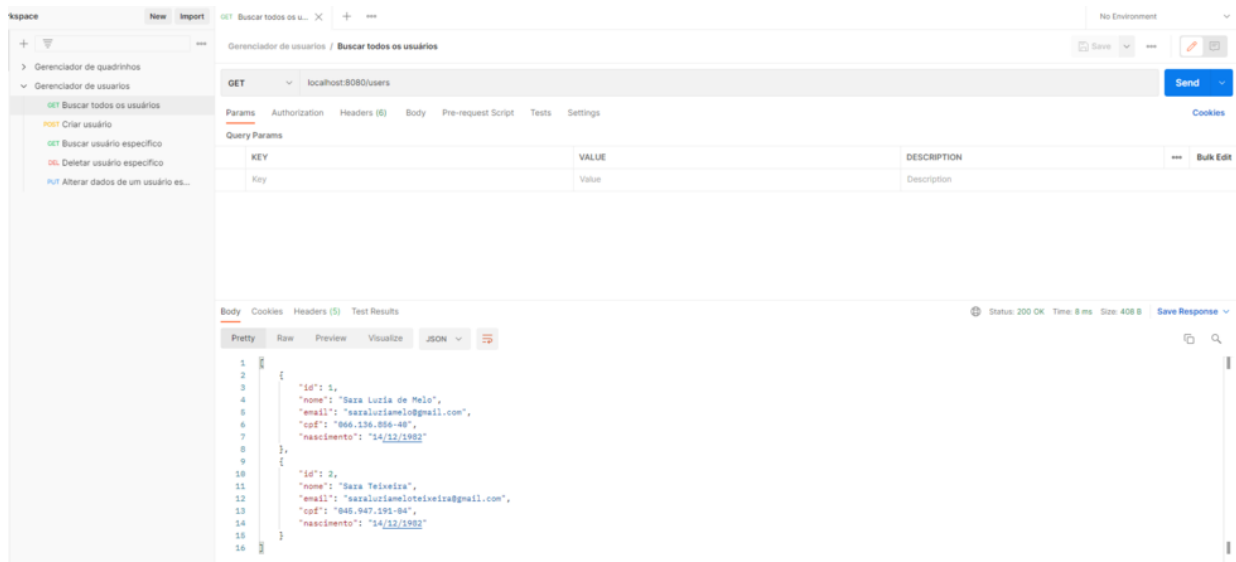
- Atualizando os dados de um usuário



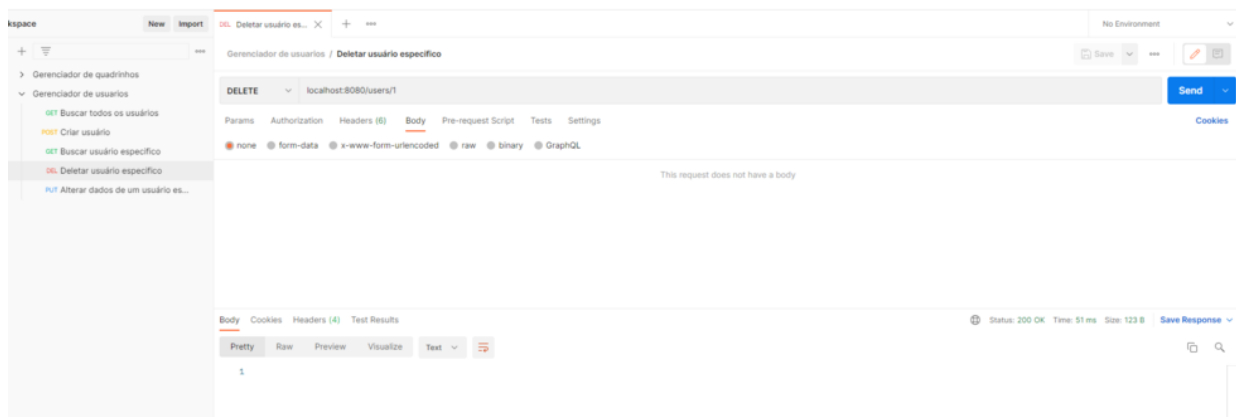
- Buscando um unico usuário, de acordo com o id (1)



- Consulta da lista de usuários cadastrados na aplicação



- Deletando um usuário



Construção do cadastro de quadrinhos (Comics)

Agora vamos criar uma classe **Comic**, a qual será representar um quadrinho. Para o cadastro de quadrinhos, deve-se consumir a API da MARVEL (disponível em <https://developer.marvel.com/>) para buscar os dados de um determinado quadrinho, baseado no comicId informado no momento do cadastro. Para este cadastro, o quadrinho deve possuir os seguintes atributos:

- ComicId
- Título
- Preço

- Autores
- ISBN
- Descrição

As restrições devem ser respeitadas

- O ComicId é obrigatório
- O Titulo é obrigatório
- O Preço é obrigatório
- As Autores são obrigatórios
- O ISBN é obrigatório e deve ser registro único;

Por meio dessas informações, crie a classe **Comic** conforme o seguinte código:

```
@Entity
@Table(name="comics",
uniqueConstraints={@UniqueConstraint(columnNames = {"ISBN"})})
public class Comic {

    @GeneratedValue(strategy = GenerationType.AUTO)
    @Id
    private Integer id;

    @NotNull(message = "O comicId é obrigatório!")
    private Integer comicId;

    @NotBlank(message = "O titulo é obrigatório!")
    private String titulo;

    @NotNull(message = "O preco é obrigatório!")
    private Double preco;

    @NotBlank(message = "Os autores são obrigatórios!")
    private String autores;

    @NotBlank(message = "Os ISBN e obrigatório!")
    private String ISBN;

    private String descricao;

    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
}
```

```

    }
    public Integer getComicId() {
        return comicId;
    }
    public void setComicId(Integer comicId) {
        this.comicId = comicId;
    }
    public String getTitulo() {
        return titulo;
    }
    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }
    public Double getPreco() {
        return preco;
    }
    public void setPreco(Double preco) {
        this.preco = preco;
    }
    public String getAutores() {
        return autores;
    }
    public void setAutores(String autores) {
        this.autores = autores;
    }
    public String getISBN() {
        return ISBN;
    }
    public void setISBN(String iISBN) {
        ISBN = iISBN;
    }
    public String getDescricao() {
        return descricao;
    }
    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }
}

```

Com o model criado, vamos criar nosso CRUD. Para isso, vamos criar o repositório **ComicsRepository** para a entidade Comics e estendemos a interface JpaRepository. Exemplo:

```

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

```

```
import sara.melo.crudspringcomics.models.Comics;

@Repository
public interface ComicsRepository extends JpaRepository<Comics, Integer> {

}
```

Agora criaremos o controlador **ComicController** e injetamos a dependência **ComicRepository**, para que seja possível criar, alterar, remover e listar os quadrinhos no banco de dados.

```
@RestController
@RequestMapping(value = "/comics")
public class ComicController {

    @Autowired
    private ComicRepository comicRepository;

}
```

Agora criaremos os métodos **save**, **getAll**, **getById**, **deleteById** e **update** que formam o CRUD de quadrinhos.

```
@RestController
@RequestMapping(value = "/comics")
public class ComicController {

    @Autowired
    private ComicRepository comicRepository;

    @GetMapping
    //Retorna uma lista de todos os quadrinhos
    public ResponseEntity<List<Comic>> getAll() {
        // Cria uma lista
        List<Comic> comics = new ArrayList<>();
        // Busca todos os quadrinhos no banco de dados
        comics = comicRepository.findAll();
        // Retorna status 200 com a lista de todos quadrinhos no banco
        return new ResponseEntity<>(comics, HttpStatus.OK);
    }

    @GetMapping(path="/{id}")
    //Retorna um unico quadrinho, de acordo com o id
    public ResponseEntity<Comic> getById(@PathVariable Integer id) {
```

```

Comic comic;
try {
    // Busca o quadrinho no banco de dados
    comic = comicRepository.findById(id);
    if(comic.isEmpty()) {
        // Retorna status 200 com os dados do quadrinho em questão
        return new ResponseEntity<Comic>(comic, HttpStatus.OK);
    } else {
        // Não encontrou o quadrinho, retorna status 404;
        return new ResponseEntity<Comic>(HttpStatus.NOT_FOUND);
    }
} catch (NoSuchElementException nsee) {
    // Não encontrou o quadrinho, retorna status 404;
    return new ResponseEntity<Comic>(HttpStatus.NOT_FOUND);
}
}

```

```

@PostMapping
public ResponseEntity<Comic> save(@RequestBody @Valid Comic comic) {
    // Salva quadrinho no banco de dados
    comicRepository.save(comic);
    // Retorna status 201 com os dados do quadrinho criado;
    return new ResponseEntity<>(comic, HttpStatus.CREATED);
}

```

```

@DeleteMapping(path="/{id}")
//Deleta um unico quadrinho, de acordo com o id
public ResponseEntity<Comic> deleteById(@PathVariable Integer id) {
    try {
        // Deleta o quadrinho no banco de dados
        comicRepository.deleteById(id);
        // Retorna status 200 , quadrinho deletado com sucesso
        return new ResponseEntity<>(HttpStatus.OK);
    } catch (NoSuchElementException nsee) {
        // Não encontrou o quadrinho, retorna status 404;
        return new ResponseEntity<Comic>(HttpStatus.NOT_FOUND);
    }
}
}

```

```

@PutMapping(path="/{id}")
//Atualiza os dados de um quadrinho; Se não encontrar, retorna erro 404
public ResponseEntity<Comic> update(@PathVariable Integer id,
@RequestBody @Valid Comic newComic) {
    return comicRepository.findById(id).map(comic -> {
        comic.setComicId(newComic.getComicId());
        comic.setTitulo(newComic.getTitulo());
        comic.setPreco(newComic.getPreco());
    });
}

```

```

        comic.setAutores(newComic.getAutores());
        comic.setISBN(newComic.getISBN());

        if(newComic.getDescricao() != null) {
            comic.setDescricao(newComic.getDescricao());
        }

        Comic comicUpdated = comicRepository.save(comic);
        return new ResponseEntity<>(comicUpdated, HttpStatus.CREATED);
    }).orElse(ResponseEntity.notFound().build());
}
}

```

Pronto! CRUD criado!

Podemos observar que um dos requisitos é consumir a API da MARVEL (disponível em <https://developer.marvel.com/>) para buscar os dados de um determinado quadrinho, baseado no **comicId**.

Geralmente, quando queremos consumir dados a partir de um Webservice, precisamos de um cliente http que seja capaz de fazer a requisição para nós. O problema, é que a maioria das vezes é muito complicado e trabalhoso fazermos isso, sem falar que o código que escrevemos para cada requisição pode se tornar repetitivo. O **Feign** foi criado justamente com o objetivo de reduzir a complexidade para consumir esses serviços.

Para facilitar nossa vida, o Spring incorporou o **Feign** em sua stack de cloud, simplificando ainda mais a configuração e integração com nossa aplicação. Dessa forma, podemos reaproveitar as anotações que utilizamos para criar Webservices, como `GetMapping`, `PathVariable`, etc.

Como definimos o Feign como uma dependência no início do projeto, basta habilitá-lo na aplicação, na classe main do projeto.

```

@EnableFeignClients
@SpringBootApplication
public class CrudSpringComicsApplication {
    public static void main(String[] args) {
        SpringApplication.run(CrudSpringComicsApplication.class, args);
    }
}

```

Para consumo da API da Marvel é necessário estar logado , para isso é necessário a chave publica e privada que são geradas no momento em que você cria a conta. Para gerar suas chaves clique [aqui](#).

Segundo a [documentação da MARVEL](#), o método que busca o quadrinho de acordo com o comicId é o seguinte:

```
GET https://gateway.marvel.com/v1/public/comics/{comicId}?
apikey={apikey}&hash={hash}&ts={timestamp}
```

Consumindo esse método no Postman, ele devolve um JSON com a seguinte estrutura:

```
{
  "code": 200,
  "status": "Ok",
  "copyright": "© 2021 MARVEL",
  "attributionText": "Data provided by Marvel. © 2021 MARVEL",
  "attributionHTML": "<a href=\"http://marvel.com\">Data provided by Marvel. © 2021 MARVEL</a>",
  "etag": "640d6dbe2a2f05adbc7f385d23addd2ffd7d0c37",
  "data": {
    "offset": 0,
    "limit": 20,
    "total": 1,
    "count": 1,
    "results": [
      {
        "id": 15808,
        "digitalId": 0,
        "title": "Ultimate Spider-Man (2000) #110 (Mark Bagley Variant)",
        "issueNumber": 110,
        "variantDescription": "Mark Bagley Variant",
        "description": "#N/A",
        "modified": "2010-11-15T14:32:28-0500",
        "isbn": "",
        "upc": "5960605031-11021",
        "diamondCode": "",
        "ean": "",
        "issn": "",
        "format": "Comic",
        "pageCount": 0,
        "textObjects": [
          {
            "type": "issue_solicit_text",
            "language": "en-us",
            "text": "\"ULTIMATE KNIGHTS\"\\r<br>The mind-stunning climax to
```

```
    }
  ],
  "resourceURI": "http://gateway.marvel.com/v1/public/comics/15808",
  "urls": [
    {
      "type": "detail",
      "url": "http://marvel.com/comics/issue/15808/ultimate_spider-ma
    },
    {
      "type": "purchase",
      "url": "http://comicstore.marvel.com/Ultimate-Spider-Man-110/di
    }
  ],
  "series": {
    "resourceURI": "http://gateway.marvel.com/v1/public/series/466",
    "name": "Ultimate Spider-Man (2000 - 2009)"
  },
  "variants": [
    {
      "resourceURI": "http://gateway.marvel.com/v1/public/comics/1580
      "name": "Ultimate Spider-Man (2000) #110"
    },
    {
      "resourceURI": "http://gateway.marvel.com/v1/public/comics/2190
      "name": "Ultimate Spider-Man (2000) #110 (Zombie Variant)"
    }
  ],
  "collections": [
    {
      "resourceURI": "http://gateway.marvel.com/v1/public/comics/1662
      "name": "Ultimate Spider-Man Vol. 18: Ultimate Knights (Trade P
    }
  ],
  "collectedIssues": [],
  "dates": [
    {
      "type": "onsaleDate",
      "date": "2029-12-31T00:00:00-0500"
    },
    {
      "type": "focDate",
      "date": "-0001-11-30T00:00:00-0500"
    }
  ],
  "prices": [
    {
      "type": "printPrice",
```

```
    "price": 2.99
  }
],
"thumbnail": {
  "path": "http://i.annihil.us/u/prod/marvel/i/mg/c/e0/4bc4947ea8f4"
  "extension": "jpg"
},
"images": [
  {
    "path": "http://i.annihil.us/u/prod/marvel/i/mg/c/e0/4bc4947ea8"
    "extension": "jpg"
  }
],
"creators": {
  "available": 3,
  "collectionURI": "http://gateway.marvel.com/v1/public/comics/1580"
  "items": [
    {
      "resourceURI": "http://gateway.marvel.com/v1/public/creators/"
      "name": "Mark Bagley",
      "role": "penciller (cover)"
    },
    {
      "resourceURI": "http://gateway.marvel.com/v1/public/creators/"
      "name": "Richard Isanove",
      "role": "penciller (cover)"
    },
    {
      "resourceURI": "http://gateway.marvel.com/v1/public/creators/"
      "name": "Brian Michael Bendis",
      "role": "writer"
    }
  ],
  "returned": 3
},
"characters": {
  "available": 1,
  "collectionURI": "http://gateway.marvel.com/v1/public/comics/1580"
  "items": [
    {
      "resourceURI": "http://gateway.marvel.com/v1/public/character"
      "name": "Spider-Man (Ultimate)"
    }
  ],
  "returned": 1
},
"stories": {
```



```

        "available": 2,
        "collectionURI": "http://gateway.marvel.com/v1/public/comics/1580",
        "items": [
            {
                "resourceURI": "http://gateway.marvel.com/v1/public/stories/3",
                "name": "5 of 5 - Ultimate Knights",
                "type": "cover"
            },
            {
                "resourceURI": "http://gateway.marvel.com/v1/public/stories/3",
                "name": "5 of 5 - Ultimate Knights",
                "type": "interiorStory"
            }
        ],
        "returned": 2
    },
    "events": {
        "available": 0,
        "collectionURI": "http://gateway.marvel.com/v1/public/comics/1580",
        "items": [],
        "returned": 0
    }
}
]
}
}

```

Legal, mas essa estrutura tem mais coisas do que eu quero então vamos dar uma “enxugada” nela. Então, criamos a classe **MarvelComicResponse** para representar o retorno dessa requisição:

```

package sara.melo.crudspringcomics.models;

import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

public class MarvelComicResponse {
    public int code;
    public String status;
    public Data data;

    public int getCode() {
        return code;
    }
}

```

```

    }
    public void setCode(int code) {
        this.code = code;
    }
    public String getStatus() {
        return status;
    }
    public void setStatus(String status) {
        this.status = status;
    }
    public Data getData() {
        return data;
    }
    public void setData(Data data) {
        this.data = data;
    }
}

public class Data {
    public int total;

    ArrayList<Results> results = new ArrayList<Results>();
    public ArrayList<Results> getResults() {
        return results;
    }
    public void setResults(ArrayList<Results> results) {
        this.results = results;
    }
    public int getTotal() {
        return total;
    }
    public void setTotal(int total) {
        this.total = total;
    }
}

public static class Results {
    public int id;
    public String title;
    public String description;
    public String isbn;
    public String issn;
    public List<Price> prices;
    public Creators creators;

    public int getId() {
        return id;
    }
}

```

```

    public void setId(int id) {
        this.id = id;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
    public String getIsbn() {
        return isbn;
    }
    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }
    public String getIssn() {
        return issn;
    }
    public void setIssn(String issn) {
        this.issn = issn;
    }
    public List<Price> getPrices() {
        return prices;
    }
    public void setPrices(List<Price> prices) {
        this.prices = prices;
    }
    public Creators getCreators() {
        return creators;
    }
    public void setCreators(Creators creators) {
        this.creators = creators;
    }
}

public static class Price {
    public String type;
    public double price;

    public String getType() {

```

```

        return type;
    }
    public void setType(String type) {
        this.type = type;
    }
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        this.price = price;
    }
}

```

```

public static class Item {
    public String name;
    public String role;
    public String type;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getRole() {
        return role;
    }
    public void setRole(String role) {
        this.role = role;
    }
    public String getType() {
        return type;
    }
    public void setType(String type) {
        this.type = type;
    }
}

```

```

public static class Creators {
    public int available;
    public String collectionURI;
    public List<Item> items;
    public int returned;

    public int getAvailable() {
        return available;
    }
}

```

```

    public void setAvailable(int available) {
        this.available = available;
    }
    public String getCollectionURI() {
        return collectionURI;
    }
    public void setCollectionURI(String collectionURI) {
        this.collectionURI = collectionURI;
    }
    public List<Item> getItems() {
        return items;
    }
    public void setItems(List<Item> items) {
        this.items = items;
    }
    public int getReturned() {
        return returned;
    }
    public void setReturned(int returned) {
        this.returned = returned;
    }

    @Override
    public String toString() {
        String creators = "";

        for (Item item : items) {
            creators += item.name + ", ";
        }

        return creators;
    }
}

```

Pronto. Agora criaremos a interface **MarvelClient** e utilizaremos a dependencia **Feign** para consumir a API da Marvel. Isso mesmo, criamos apenas uma interface e ela faz toda a “mágica” de chamar o serviço e fazer o parser dos dados. Veja como fica a interface:

```

@FeignClient(name = "marvelClient", url = "https://gateway.marvel.com")
public interface MarvelClient {

    @RequestMapping(method = RequestMethod.GET,
        value = "/v1/public/comics/{comicId}?apikey={apikey}&hash={hash}&ts={timestamp}")

```

```

        MarvelComicResponse getComic(@PathVariable("comicId") Integer comicId,
        @PathVariable("apikey") String apikey, @PathVariable("hash") String has
        @PathVariable("timestamp") Long timestamp);
    }

```

Nota-se que ao importamos a anotação **@FeignClient**, deveremos passar a url base da API, que é <https://gateway.marvel.com> .

De acordo com a documentação da MARVEL, em toda requisição deveremos também passar os seguintes parametros: **apikey**, **hash** e **timestamp**. Onde:

- apikey é a chave publica;
- hash é combinação MD5(timestamp + chave privada + chave publica)
- ts é a data atual em milisegundos;

Ensinaremos agora como importar a interface **MarvelClient** em nosso **ComicController** para podemos realizar requisições HTTP para MARVEL.

Abra o nosso **ComicController** e crie duas variáveis como nome de privateKey e publicKey do tipo final String para armazenarmos nossas chaves geradas no site da MARVEL.

Importe o client **MarvelClient** da seguinte forma. Com isso podemos chamar nosso client:

```

public class ComicController {

    private final String privateKey = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
    private final String publicKey = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";

    @Autowired
    private ComicRepository comicRepository;

    @Autowired
    private MarvelClient marvelClient;

    ...

```

Como queremos buscar o quadrinho pelo id, vamos criar o getComicById que recebe o comicId e retorna a resposta da API.

Lembra que no nosso client **MarvelClient** criamos o método getComic que espera 4 parametros para fazer a interface com a API? É aqui que passamos essas informações:

```

private MarvelComicResponse getComicById(Integer comicId) {
    Long timestamp = System.currentTimeMillis();

    MarvelComicResponse comic = this.marvelClient.getComic(comicId,
        this.publicKey, this.MD5(timestamp + this.privateKey + this.publicKey
        timestamp));
    return comic;
}

```

A MARVEL espera em todas requisições um hash MD5 do timestamp + chave publica + chave privada, então criamos o seguinte método para gerar o hash MD5 e o passamos para nosso marvelClient:

```

// GERA O MD5 HASH
private String MD5(String md5) {
    try {
        MessageDigest md = MessageDigest.getInstance("MD5");
        byte[] array = md.digest(md5.getBytes());
        StringBuffer sb = new StringBuffer();
        for (int i = 0; i < array.length; ++i) {
            sb.append(Integer.toHexString((array[i] & 0xFF) | 0x100).substring(1, 3));
        }
        return sb.toString();
    } catch (NoSuchAlgorithmException e) {
    }
    return null;
}

```

Como queremos cadastrar um quadrinho e devemos buscar os dados de um desse quadrinho, baseado no comicId informado no momento do cadastro.

Para isso, devemos alterar o método **save** criado anteriormente para a seguinte forma:

```

@PostMapping
public ResponseEntity save(@RequestBody Comic comic) {
    try {
        // REALIZA A CONSULTA NA API DA MARVEL, PASSANDO O comicId
        MarvelComicResponse comicResponse = this.getComicById(comic.getComicId());

        //Busca o primeiro resultado da busca
        Results comicEncontrado = comicResponse.getData().getResults().get(0);

        // No catalogo de quadrinhos, existem poucos quadrinhos com ISBN.
        // Então verificamos se esse banco e vazio
        // Se for vazio, retorna uma exceção de conflito
    }
}

```

```

    if(comicEncontrado.getIsbn().isEmpty()
    || comicEncontrado.getIsbn().isBlank()
    ) {
        return ResponseEntity
            .status(HttpStatus.CONFLICT)
            .body("O Quadrinho com o comicId" + comic.getComicId() + " não t
                ISBN.");
    } else {
        // SE TIVER O ISBN, PEGAMOS AS INFORMAÇÕES DO QUADRINHO E
        // SALVAMOS EM NOSSO BANCO DE DADOS

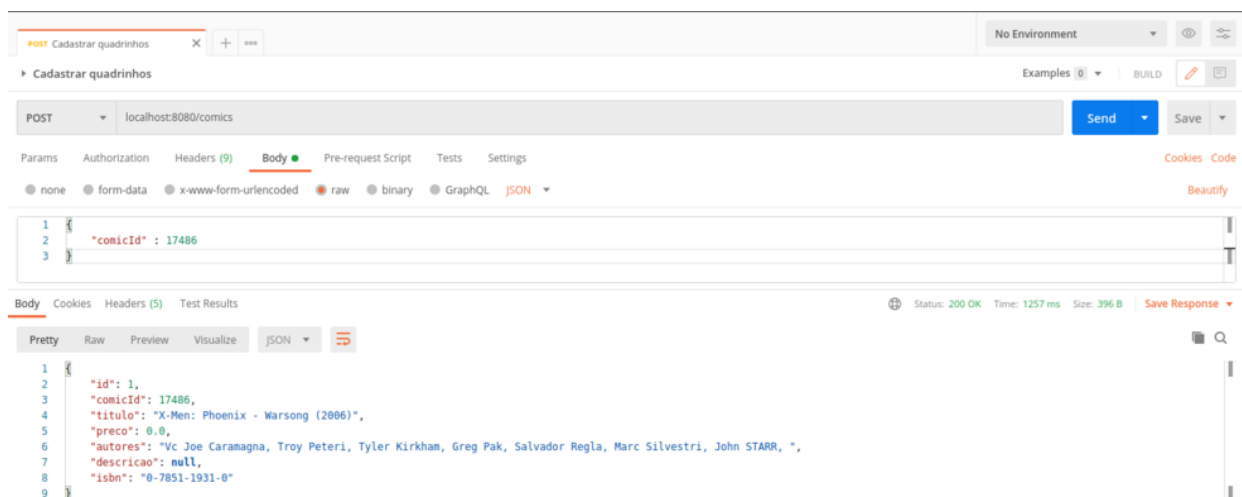
        Comic comicNew = new Comic();
        comicNew.setComicId(comicEncontrado.getId());
        comicNew.setAutores(comicEncontrado.getCreators().toString());
        comicNew.setISBN(comicEncontrado.getIsbn());
        comicNew.setTitulo(comicEncontrado.getTitle());
        comicNew.setPreco(comicEncontrado.getPrices().get(0).getPrice());

        // Salva quadrinho no banco de dados
        comicRepository.save(comicNew);
        return new ResponseEntity(comicNew, HttpStatus.CREATED);
    }

} catch (Exception e) {
    return ResponseEntity
        .status(HttpStatus.CONFLICT)
        .body(e.getMessage());
}
}

```

Testando nosso cadastrado, deveremos passar no body da requisição somente o atributo **comicId** que o restante das informações será consumida da API e o quadrinho será gravado no banco de dados.



Criando relacionamento entre Usuários e Quadrinhos

Como devemos criar um cadastro de quadrinhos para um determinado usuário, e as próximo passo é criar os relacionamentos entre as entidades **Users** e **Comics**.

Observando esse relacionamento, um usuário pode ter vários quadrinhos e um quadrinho deve ser único e pode ter vários usuários. Essa relação é muitos para muitos.

Para definir esse relacionamento no JPA, referenciamos cada entidade e adicionamos a anotação **@ManyToMany** em nossas entidades. Note que utilizamos a propriedade `mappedBy` da anotação `ManyToMany` para informar que o `User` é o dono do relacionamento.

```
@Entity
@Table(name="users",
uniqueConstraints={@UniqueConstraint(columnNames = {"cpf" , "email"})})
public class User {
    ...
    @ManyToMany(mappedBy="users")
    @JsonIgnore
    private Set<Comic> comics = new HashSet<>();
    public Set<Comic> getComics() {
        return comics;
    }
    public void setComics(Set<Comic> comics) {
        this.comics = comics;
    }
    ...
}
```

A entidade `Comic` possui um relacionamento de muitos-para-muitos com a entidade `User`, para definir esta associação utilizamos a anotação **@ManyToMany**.

Para informar que vamos utilizar a tabela `comics_users` para realizar a associação entre `Comic` e `User` utilizamos a anotação **@JoinTable**.

Para informar que a coluna `comic_fk` da tabela `comics_users` é a coluna chave estrangeira para a tabela `Comics` e utilizamos a anotação `inverseJoinColumn` e para informar que a coluna `user_fk` da tabela `comics_users` é a chave estrangeira para a tabela `User` utilizamos a anotação `joinColumns`.

```

@Entity
@Table(name="comics",
uniqueConstraints={@UniqueConstraint(columnNames = {"ISBN"})})
public class Comic {
    ...
    @ManyToMany(cascade = CascadeType.ALL)
    @JoinTable(name="comics_users",
        joinColumns = {@JoinColumn(name = "comic_fk")},
        inverseJoinColumns = {@JoinColumn(name = "user_fk")}
    )
    private Set<User> users = new HashSet<>();

    public Set<User> getUsers() {
        return users;
    }
    public void setUsers(Set<User> users) {
        this.users = users;
    }
    ...
}

```

Pronto, nosso relacionamento no banco de dados está criado.

Precisaremos agora criar o método do tipo **PUT** com o nome de **linkComicToUser** que vincula um quadrinho a um determinado usuário no controlador **ComicController**.

Segue o código de exemplo:

```

@PutMapping(path="/{comicId}/user/{userId}")
//Vincula um quadrinho a um determinado usuário.
public ResponseEntity linkComicToUser(@PathVariable Integer comicId,
@PathVariable Integer userId) {

    if(comicRepository.findById(comicId).isEmpty()) {
        return ResponseEntity
            .status(HttpStatus.NOT_FOUND)
            .body("O Quadrinho com o comicId " + comicId + " não foi encontrado!")
    }

    if(userRepository.findById(userId).isEmpty()) {
        return ResponseEntity
            .status(HttpStatus.NOT_FOUND)
            .body("O Usuário com o userId" + userId + " não foi encontrado!");
    }

    try {
        Comic comic = comicRepository.findById(comicId).get();
    }
}

```

```

        User user = userRepository.findById(userId).get();

        comic.getUsers().add(user);

        Comic comicUpdated = comicRepository.save(comic);

        return new ResponseEntity<>(comicUpdated, HttpStatus.CREATED);
    } catch (Exception e) {
        return ResponseEntity
            .status(HttpStatus.BAD_REQUEST)
            .body(e.getMessage());
    }
}

```

Observando o código acima, mapeamos esse método para a seguinte rota: **/{{comicId}}/user/{{userId}}** e realizamos também a injeção de dependência do **UserRepository** para buscar o usuário por id.

Inicialmente, buscamos o Comic e o User por id. Caso encontre, utilizamos a referência do Comic e o método `getUsers().add(user)` para vincular o usuário a este quadrinho, e por fim, salvar essas informações. Caso contrário, retorne o erro com status **NOT_FOUND**. Segue o exemplo do teste da API para vincular um quadrinho a um usuário.

Vincular um quadrinho a um usuario

Examples 0 BUILD

PUT localhost:8080/comics/11/user/1 Send Save

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies Code

Query Params

KEY	VALUE	DESCRIPTION	*** Bulk Edit
Key	Value	Description	

Body Cookies Headers (5) Test Results Status: 201 Created Time: 19 ms Size: 562 B Save Response

Pretty Raw Preview Visualize JSON

```

1  {
2    "id": 11,
3    "comicId": 41530,
4    "titulo": "Ant-Man: Astonishing Origins (Trade Paperback)",
5    "preco": 17.99,
6    "autores": "Tom DeFalco, Nick Spencer, Horacio Domingues, Ramon Rosanas, Julian Totino Tedesco, Jeff Youngquist, ",
7    "descricao": null,
8    "desconto": null,
9    "users": [
10     {
11       "id": 1,
12       "nome": "Sara Melo",
13       "email": "saraluziamelo2@gmail.com",
14       "cpf": "045.947.191-04",
15       "nascimento": "14/12/1982"
16     },
17     {
18       "id": 2,
19       "nome": "João Silva",
20       "email": "joaosilva@gmail.com",
21       "cpf": "123.456.789-01",
22       "nascimento": "01/01/1990"
23     }
24   ],
25   "isbn": "978-0-7851-6390-9"
26 }

```

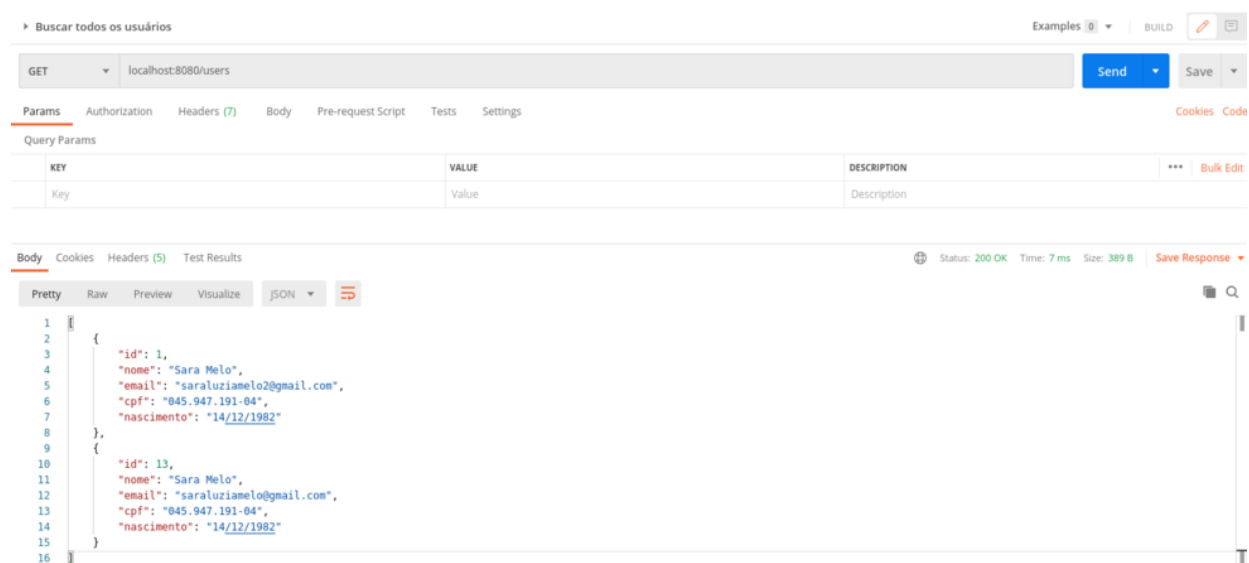
Consultando a lista de usuários cadastrados na aplicação

Conforme implementamos acima o método **getAll** em nosso **UserController** para listar os usuários na aplicação, sem retornar a lista de quadrinhos de cada usuário, adicionei a anotação **@JsonIgnore** no relacionamento com a entidade Comic conforme código abaixo.

```
@ManyToMany(mappedBy="users")
@JsonIgnore
private Set<Comic> comics = new HashSet<>();

@GetMapping
//Retorna uma lista de todos os usuários
public ResponseEntity<List<User>> getAll() {
    // Cria uma lista
    List<User> users = new ArrayList<>();
    // Busca todos os usuários no banco de dados
    users = userRepository.findAll();
    // Retorna status 201 com a lista de todos usuários no banco
    return new ResponseEntity<>(users, HttpStatus.OK);
}
```

Assim sendo, segue o exemplo da consulta.



The screenshot displays a REST client interface with a GET request to `localhost:8080/users` successfully executed. The response is a JSON array of two user objects. The status is 200 OK, and the response size is 389 B.

Request: GET localhost:8080/users

Response Body (JSON):

```
[
  {
    "id": 1,
    "nome": "Sara Melo",
    "email": "saraluziamelo2@gmail.com",
    "cpf": "045.947.191-04",
    "nascimento": "14/12/1982"
  },
  {
    "id": 13,
    "nome": "Sara Melo",
    "email": "saraluziamelo@gmail.com",
    "cpf": "045.947.191-04",
    "nascimento": "14/12/1982"
  }
]
```

Consultando a lista de quadrinhos de um determinado usuário

Para esta listagem, criamos o método com o formato GET **getByUserId** em nosso **ComicController** com o mapeamento para a rota **/byUser/{id}**, onde esperamos receber via parâmetro o id do usuário.

```
@GetMapping(path="/byUser/{id}")
//Retorna um unico quadrinho, de acordo com o id
public ResponseEntity<Set<Comic>> getByUserId(@PathVariable Integer id) {
    Set<Comic> comics;

    try {
        comics = comicRepository.findAllByUserId(id);
        if(!comics.isEmpty()) {
            // Retorna status 200 com os dados do quadrinho em questão
            return new ResponseEntity(comics, HttpStatus.OK);
        } else {
            // Não encontrou o quadrinho, retorna status 404;
            return new ResponseEntity(HttpStatus.NOT_FOUND);
        }
    } catch (NoSuchElementException nsee) {
        // Não encontrou o quadrinho, retorna status 404;
        return new ResponseEntity(HttpStatus.NOT_FOUND);
    }
}
```

Além disso, defini uma consulta personalizada em nosso **ComicRepository**, assim, criamos o método **findAllByUserId** onde utilizamos a anotação **@Query** para buscar todos os quadrinhos de um determinado usuário.

```
@Repository
public interface ComicRepository extends JpaRepository<Comic, Integer> {

    @Query("SELECT new Comic(t.id, t.comicId, t.titulo, t.preco,
        t.autores, t.ISBN, t.descricao) from Comic t join t.users
        u where u.id = :id")
    Set<Comic> findAllByUserId(@Param("id") Integer id);
}
```

Segue o exemplo da listagem de todos os quadrinhos para o usuário **13**:

The screenshot shows a REST client interface with the following details:

- Request:** GET `localhost:8080/comics/byUser/13`
- Response Status:** 200 OK, Time: 6 ms, Size: 421 B
- Response Body (JSON):**

```
{
  "id": 16,
  "comicId": 41530,
  "titulo": "Ant-Man: Astonishing Origins (Trade Paperback)",
  "preco": 17.99,
  "autores": "Tom DeFalco, Nick Spencer, Horacio Domingues, Ramon Rosanas, Julian Totino Tedesco, Jeff Youngquist, ",
  "descricao": null,
  "isbn": "978-0-7851-6390-9"
}
```

Prontinho pessoal, espero que tenha compreendido como desenvolver um projeto utilizando o SpringBoot.

No nosso próximo tutorial vamos inserir algumas regras de negócio para verificar se cada quadrinho de um determinado usuário possui desconto dependendo do dia da semana. Até lá!!!