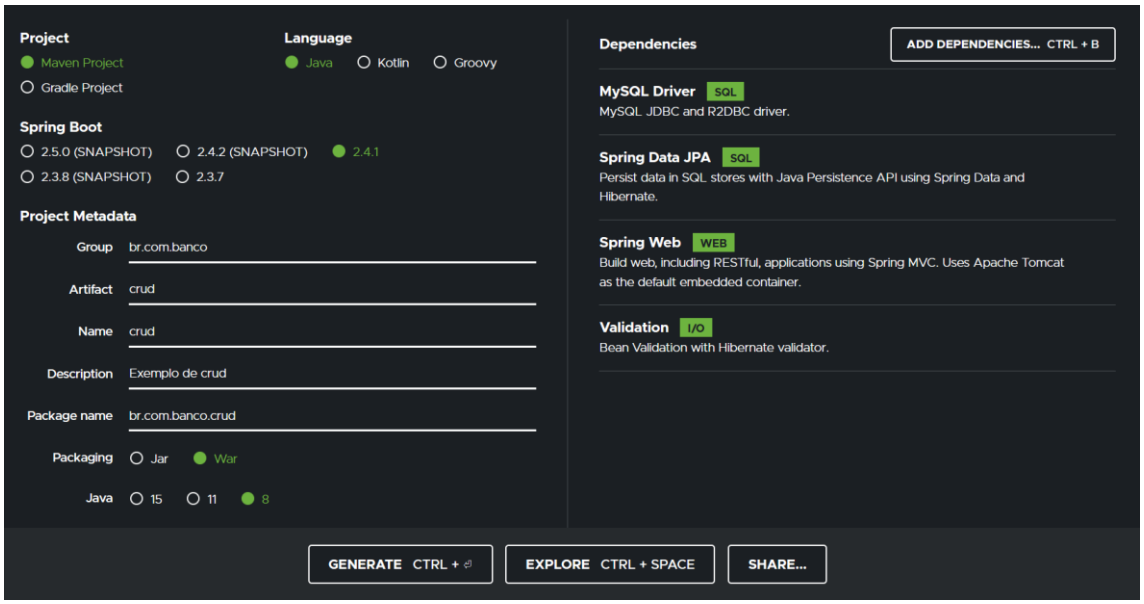
Criando uma API CRUD simples em Rest utilizando JAVA + Spring + Hibernate + MySql + Bean Validation

Nesse projeto, iremos construir uma aplicação CRUD simples utilizando JAVA versão 8 como linguagem de programação, Maven para gerenciamento de dependências, Spring Boot como framework de gerenciamento geral, Spring Web para criação de aplicações REST, Spring Data JPA com Hibernate e MYSQL para persistência de dados e o Bean Validation para validação dos dados. Essas tecnologias serão usadas para facilitar a codificação e por ser as tecnologias mais utilizados atualmente, atingindo um excelente nível de maturidade e robustez tanto para aplicações de pequeno porte até as de grande porte e complexas.

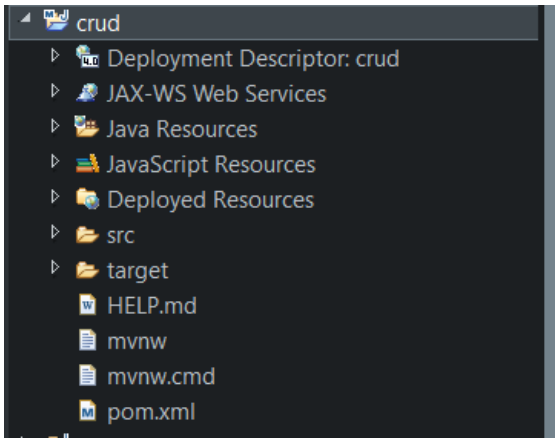
Iremos criar as seguintes classes em nosso projeto:
- **Usuario.java**: Classe Model responsável por criar a entidade, os atributos, getters e setters, além das validações necessárias. A mesma será persistida no Banco de Dados sendo convertida via ORM pelo Spring JPA.
- **UsuarioRepository.java**: Classe responsável por realizar as operações do Banco de Dados. O Spring Data é muito poderoso e já possui toda a lógica de implementação e conversão necessárias se tornando muito simples o controle de nossas entidade e conversão de Objeto para Entidade de Banco de Dados, sendo necessário, apenas, extender alguma de suas interfaces que em nosso caso utilizaremos JPARepository. Toda a lógica de acesso, inserção, deleção e paginação entre outras é abstraída.
- **UsuarioController.java**: Classe Controller do pacote Spring Web onde iremos mapear nossos endpoints, receber as requisições de dados, efetuar a chamada adequada ao Banco de Dados e retornar as respostas adequadas. Como estamos criando uma aplicação REST, o padrão de requisição resposta será via JSON, um padrão criado para integração entre diversos sistemas e aplicações mesmo que utilizem linguagem de programação e infraestrutura diferentes.
- **MyExceptionHandler.java**: Classe responsável pelo tratamento de exceções lançadas pelo nosso sistema, como por exemplo falha de Banco de Dados e falha na validação dos dados. Para manter o padrão de respostas de retorno do Spring, iremos criar uma função createBodyResponse para retornar as mensagens de erro via JSON da mesma forma que o Spring retornaria.

Criação do Projeto via Spring Initializr

Antes de mais nada, precisamos criar um projeto no site Spring Initializr pelo link: https://start.spring.io/. No site selecionamos Project “Maven Project”, Language “Java”, Spring Boot “2.4.1” (devido ser a versão recente mais estável), Packaging “War” e Java “8”. Então preencheremos o Project Metadata conforme nosso projeto e incluiremos as seguintes dependências necessárias:



Após isso, clicamos em GENERATE, baixamos o projeto gerado em formato zip e extraímos para nosso workspace. Em seguida abrimos uma IDE de preferência que no nosso caso será o Eclipse IDE, importamos o projeto clicando em FILE -> IMPORT -> Selecionamos Existing Maven Projects -> NEXT -> Em root directory selecionamos a pasta do projeto que acabamos de extrair -> FINISH. Aguardamos até que o Eclipse reconheça e monte as pastas do nosso projeto, pode demorar um pouco e então o mesmo gera a seguinte estrutura básica:



Application.properties

Vamos definir os dados de nosso Banco de Dados no arquivo application.properties que geralmente fica na pasta main/resources, note que devemos informar a Timezone nessa nossa versão do Driver MySQL, usamos, também, `spring.jpa.hibernate.ddl-auto=update` para que o spring crie as tabelas no Banco de Dados automaticamente e `server.error.include-message=Always` para que as mensagens de exceção personalizadas sejam exibidas via JSON:

```
1 spring.datasource.url=jdbc:mysql://localhost:3306/crud?serverTimezone=America/Sao_Paulo
2 spring.datasource.username=root
3 spring.datasource.password=
4 spring.jpa.hibernate.ddl-auto=update
5
6 server.error.include-message=always
```

Classe Model

Vamos começar criando a nossa classe model referente a Usuario. Para fins de organização vamos criar uma pasta(package) Model e inserir nossa classe Usuario dentro da mesma. Nossa classe vai receber os atributos CPF, Nome, E-mail e Data de nascimento. Devemos anotar nossa classe como @Entity para ser uma classe de persistência no Banco de Dados, anotamos o atributo CPF como @Id para ser a nossa chave primária além de definirmos que CPF e E-mail serão atributos únicos no Banco de Dados para não permitirmos registros duplicados com esses atributos. Definimos as validações utilizando as anotações do Bean Validation e JsonFormat para serialização e desserialização automáticas para resposta da aplicação. Geramos os getters and setters e finalmente nossa classe terá a seguinte estrutura:

```java
@Entity
public class Usuario {
    @Id
    @NotEmpty(message = "CPF é obrigatório")
    @CPF(message = "CPF Inválido")
    @Column(unique = true)
    String cpf;
    @NotEmpty(message = "Nome é obrigatório")
    String nome;
    @NotEmpty(message = "E-mail é obrigatório")
    @Email(message = "E-mail inválido")
    @Column(unique = true)
    String email;
    @DateTimeFormat (pattern = "MM/dd/yyyy")
    @JsonFormat(pattern = "dd/MM/yyyy")
    private LocalDate dataNascimento;

    public String getCpf() {
        return cpf;
    }
    public void setCpf(String cpf) {
        this.cpf = cpf;
    }
    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 LocalDate getDataNascimento() {
        return dataNascimento;
    }
    public void setDataNascimento(LocalDate dataNascimento) {
        this.dataNascimento = dataNascimento;
    }
}
```

Classe Repository

Agora criaremos nossa Classe de acesso ao Banco de Dados, o Spring facilita muito sendo necessário apenas de uma interface que extenda JPARespository:

```java
package br.com.banco.crud.Repository;

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

import br.com.banco.crud.Model.Usuario;

public interface UsuarioRepository extends JpaRepository<Usuario, String>{

}
```

Classe Controller

Nesse momento já podemos criar nosso Controller, que irá receber as requisições e devolver as respostas no formato JSON. Anotamos nossa classe com @RestController e injetamos a dependência do nosso repositório utilizando @AutoWired, criamos os endpoints responsáveis pela criação, alteração, exclusão e lista dos usuários:

```java
@RestController
public class UsuarioController {
    @Autowired
    private UsuarioRepository repository;

    @GetMapping("/usuarios")
    public List<Usuario> Get() {
        return repository.findAll();
    }

    @GetMapping("/usuarios/{cpf}")
    public Usuario GetById(@PathVariable String cpf) {
        return repository.findById(cpf).get();
    }

    @PostMapping("/usuarios")
    public Usuario Post(@Valid @RequestBody(required = false) Usuario usuario) {
            return repository.save(usuario);
    }

    @PutMapping("/usuarios/{cpf}")
    public ResponseEntity<Usuario> Put(@Valid @RequestBody Usuario novoUsuario, @PathVariable String cpf) {
        return repository.findById(cpf)
                .map(usuario -> {
                    usuario.setNome(novoUsuario.getNome());
                    usuario.setEmail(novoUsuario.getEmail());
                    usuario.setDataNascimento(novoUsuario.getDataNascimento());
                    repository.save(usuario);
                    return new ResponseEntity<>(usuario, HttpStatus.OK);
                }).orElseThrow(() -> new IllegalArgumentException());
    }

    @DeleteMapping("/usuarios/{cpf}")
    public void Delete(@PathVariable String cpf) {
        repository.deleteById(cpf);
    }

}
```

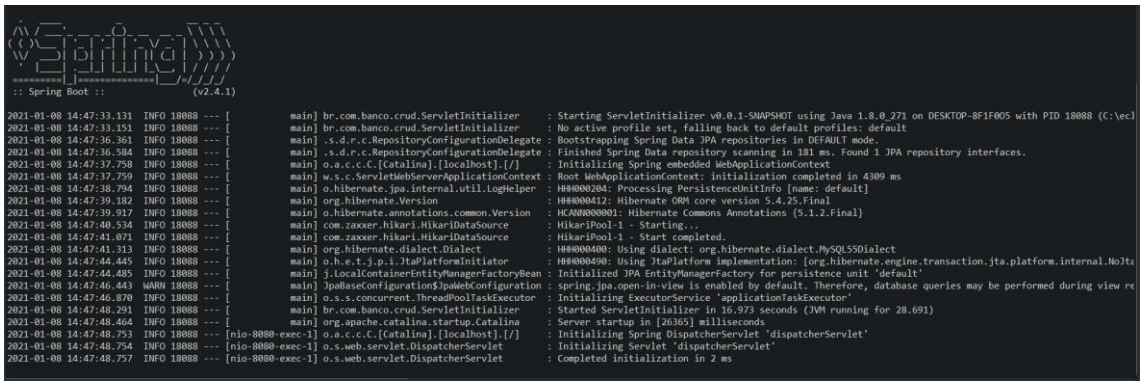
Classe de tratamento de exceções

O framework Spring possui um comportamento padrão de tratamento de exceções, mas em nosso caso, iremos sobrescrever alguns de seus comportamentos para devolver respostas personalizadas para o cliente que consumir a aplicação. Conforme dito anteriormente, criaremos uma função privada na classe (createBodyResponse) para retornar a mensagem no mesmo padrão do Spring. Nossa classe “MyExceptionHandler” irá extender a classe padrão do Spring “ResponseEntityExceptionHandler” e sobrescrever alguns de seus métodos, além tratar algumas exceções não tratadas pela superclasse, ficando da seguinte forma:

```java
@RestControllerAdvice
public class MyExceptionHandler extends ResponseEntityExceptionHandler {

    @Override
    protected ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex,
            HttpHeaders headers, HttpStatus status, WebRequest request) {
        List<String> erros = new ArrayList<String>();
        for(ObjectError oe : ex.getBindingResult().getAllErrors())
            erros.add(oe.getDefaultMessage());
        return new ResponseEntity<>(createBodyResponse(headers, status, erros), headers, status);
    }

    @Override
    protected ResponseEntity<Object> handleHttpMessageNotReadable(HttpMessageNotReadableException ex,
            HttpHeaders headers, HttpStatus status, WebRequest request) {
        return new ResponseEntity<>(createBodyResponse(headers, status,
                ex.getMostSpecificCause().getMessage()), headers, status);
    }

    @Override
    protected ResponseEntity<Object> handleHttpRequestMethodNotSupported(
            HttpRequestMethodNotSupportedException ex,
            HttpHeaders headers, HttpStatus status, WebRequest request) {
        return new ResponseEntity<>(createBodyResponse(headers, status, ex.getMessage()), headers, status);
    }

    private Map<String, Object> createBodyResponse(
            HttpHeaders headers, HttpStatus status, Object erros) {
        Map<String, Object> body = new LinkedHashMap<>();
        body.put("timestamp", new Date());
        body.put("status", status.value());
        body.put("error", status.getReasonPhrase());
        body.put("message", erros);
        body.put("path", ServletUriComponentsBuilder.fromCurrentRequest().toUriString());
        return body;
    }

    @ExceptionHandler(ConstraintViolationException.class)
    public void constraintViolationException(
            HttpServletResponse response, ConstraintViolationException ex) throws IOException {
        response.sendError(HttpStatus.INTERNAL_SERVER_ERROR.value(), "E-mail ou CPF já registrado");
    }

    @ExceptionHandler(IllegalArgumentException.class)
    public void illegalArgumentException(HttpServletResponse response) throws IOException {
        response.sendError(HttpStatus.INTERNAL_SERVER_ERROR.value(), "Falha ao buscar/salvar o usuário");
    }

    @ExceptionHandler({EmptyResultDataAccessException.class, NoSuchElementException.class})
    public void emptyResultDataAccessException(HttpServletResponse response) throws IOException {
        response.sendError(HttpStatus.INTERNAL_SERVER_ERROR.value(), "Usuário não localizado");
    }


}
```

Testando a aplicação

Nossa aplicação já está pronta para funcionar, iremos utilizar o Tomcat como servidor para rodar nossa aplicação. Iremos iniciar diretamente pelo Tomcat sem utilizar nenhuma classe para isso e seu log de saída irá se apresentar da seguinte forma ao iniciar:



Após iniciar, vamos efetuar os testes de nossa aplicação, utilizaremos o Postman Desktop para isso.

Teste HTTP POST

- Utilizando uma data errada:

POST http://localhost:8080/usuarios

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

none form-data x-www-form-urlencoded raw binary GraphQL JSON

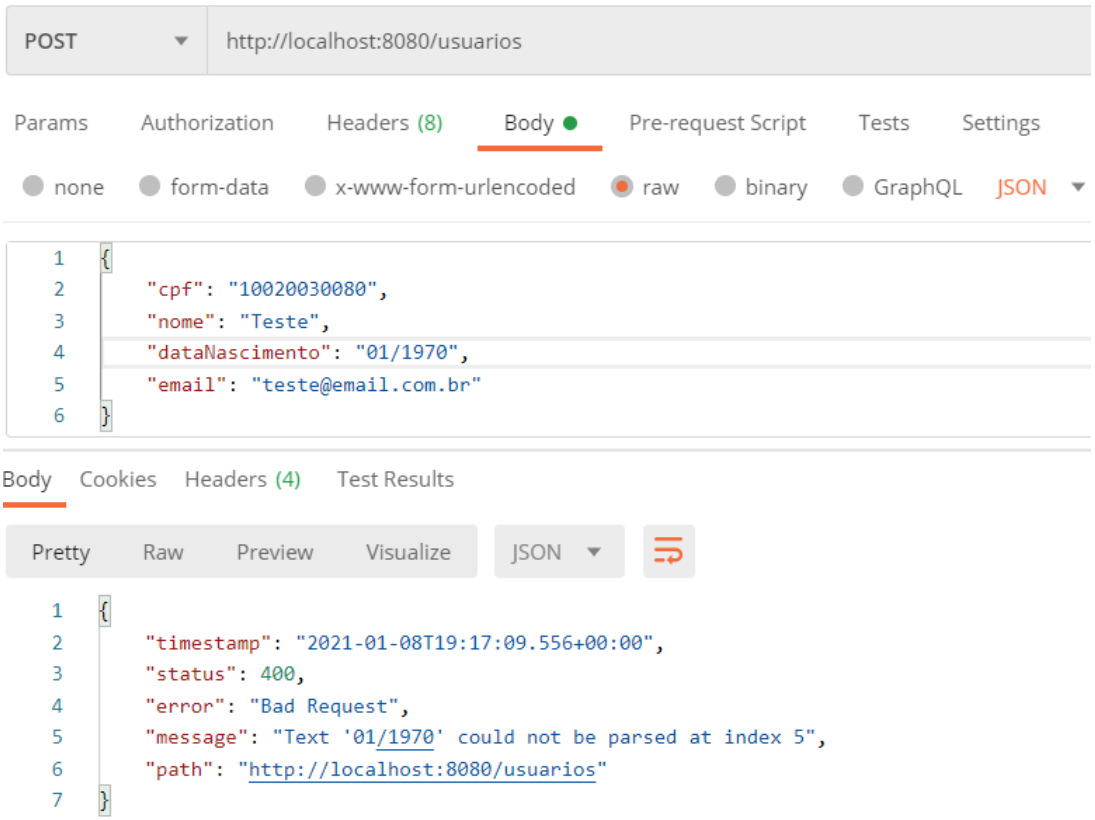
```
{
    "cpf": "10020030080",
    "nome": "Teste",
    "dataNascimento": "41/01/1970",
    "email": "teste@email.com.br"
}
```

Body Cookies Headers (4) Test Results

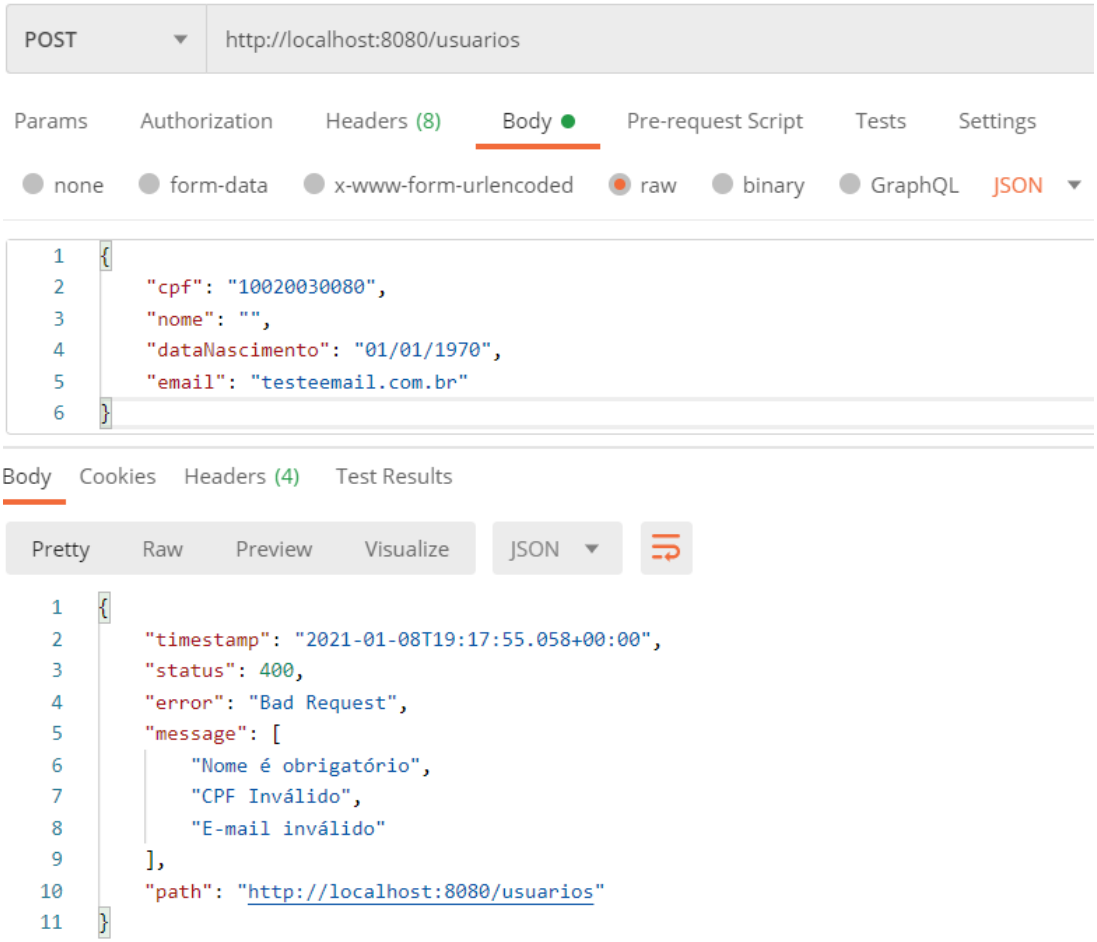
Pretty Raw Preview Visualize JSON

```
{
    "timestamp": "2021-01-08T19:16:17.934+00:00",
    "status": 400,
    "error": "Bad Request",
    "message": "Invalid value for DayOfMonth (valid values 1 - 28/31): 41",
    "path": "http://localhost:8080/usuarios"
}
```

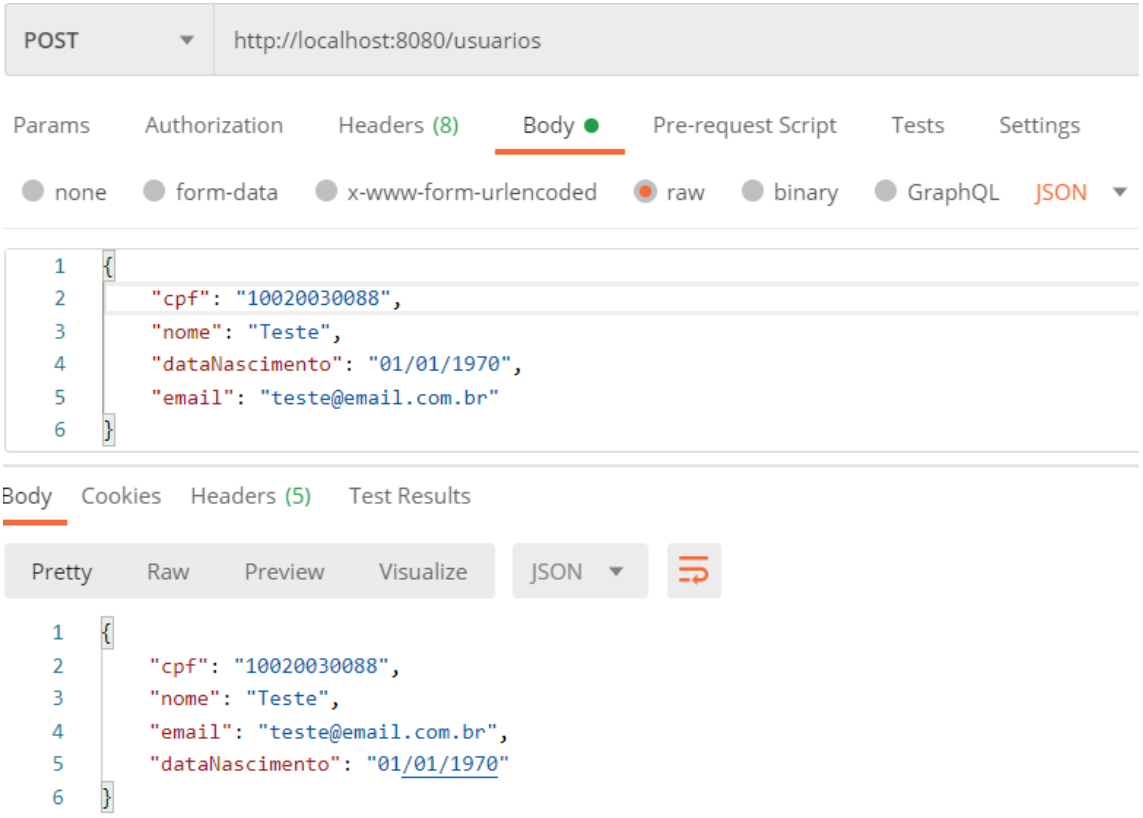
- Utilizando formato de data fora do padrão dd/MM/yyyy:



- Utilizando CPF, Nome e E-mail fora do padrão:



- Inserindo os dados corretamente o mesmo retorna a entidade criada:



Adicionamos mais dois usuários gerados aleatoriamente para testes.

Teste HTTP GET

- Listar todos os usuários:

GET http://localhost:8080/usuarios

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

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON


```
[
    {
        "cpf": "10020030088",
        "nome": "Teste",
        "email": "teste@email.com.br",
        "dataNascimento": "01/01/1970"
    },
    {
        "cpf": "30941165663",
        "nome": "Allana Isabella das Neves",
        "email": "allanaisabelladasneves@email.com.br",
        "dataNascimento": "08/05/1952"
    },
    {
        "cpf": "71307541682",
        "nome": "Oliver Anthony Nunes",
        "email": "oliveranthonynunes@email.com.br",
        "dataNascimento": "23/06/1950"
    }
]
```


- Buscar usuário específico:

GET http://localhost:8080/usuarios/10020030088

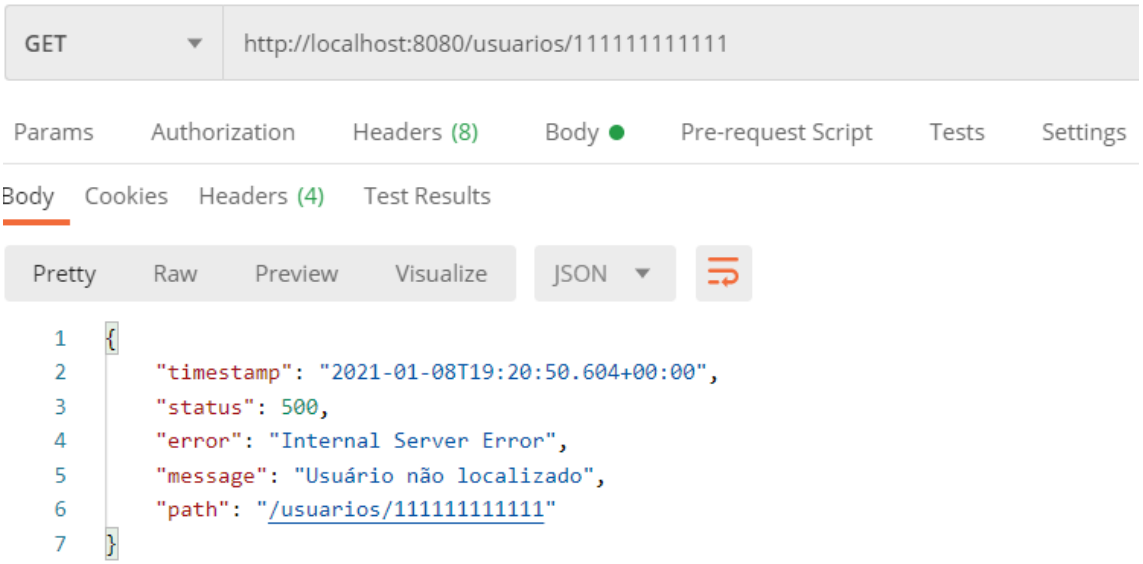
Params Authorization Headers (8) Body Pre-request Script Tests Settings

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

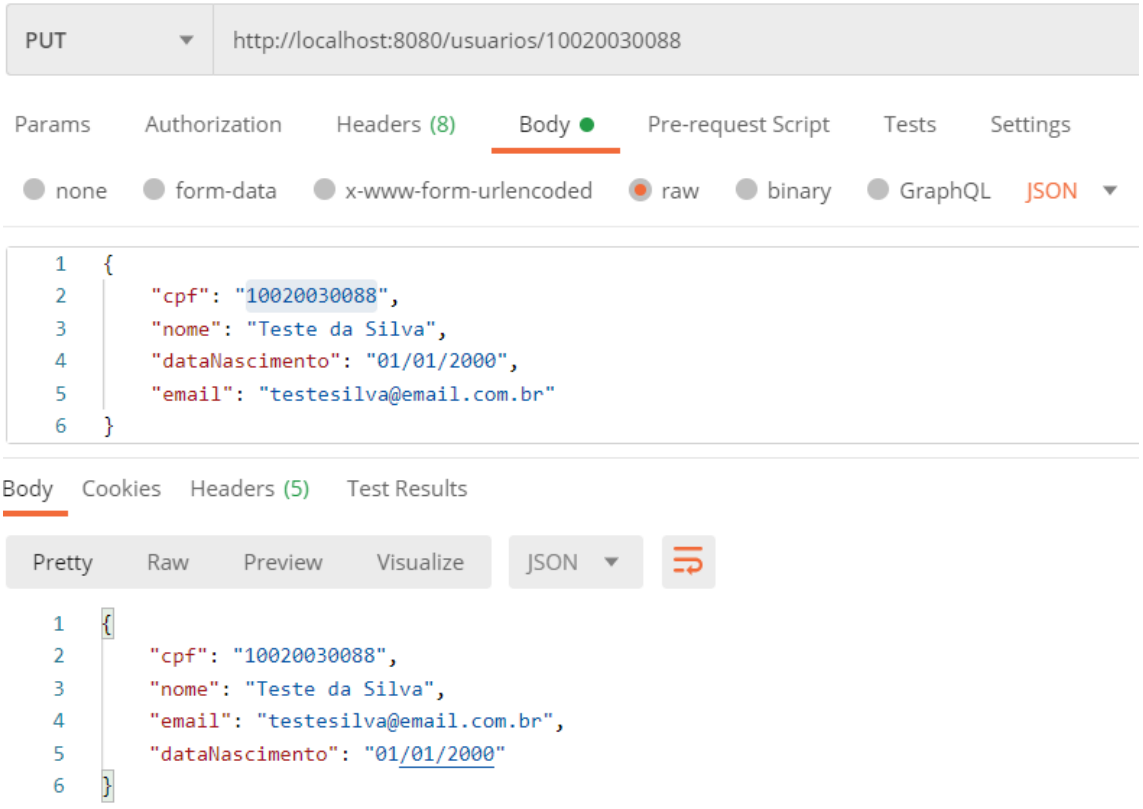

```
{
    "cpf": "10020030088",
    "nome": "Teste",
    "email": "teste@email.com.br",
    "dataNascimento": "01/01/1970"
}
```

- Buscar usuário inexistente:

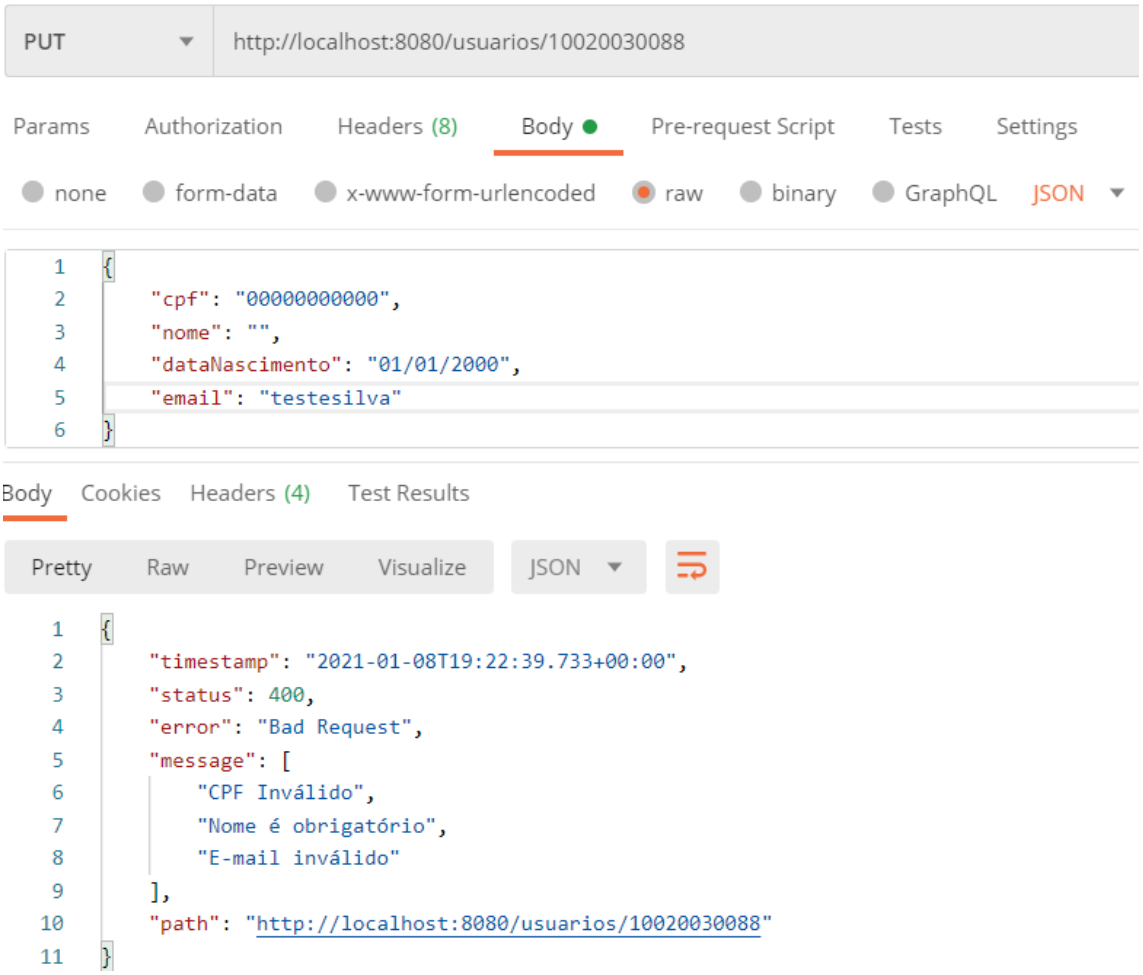


Teste HTTP PUT

- Alterar usuário corretamente:

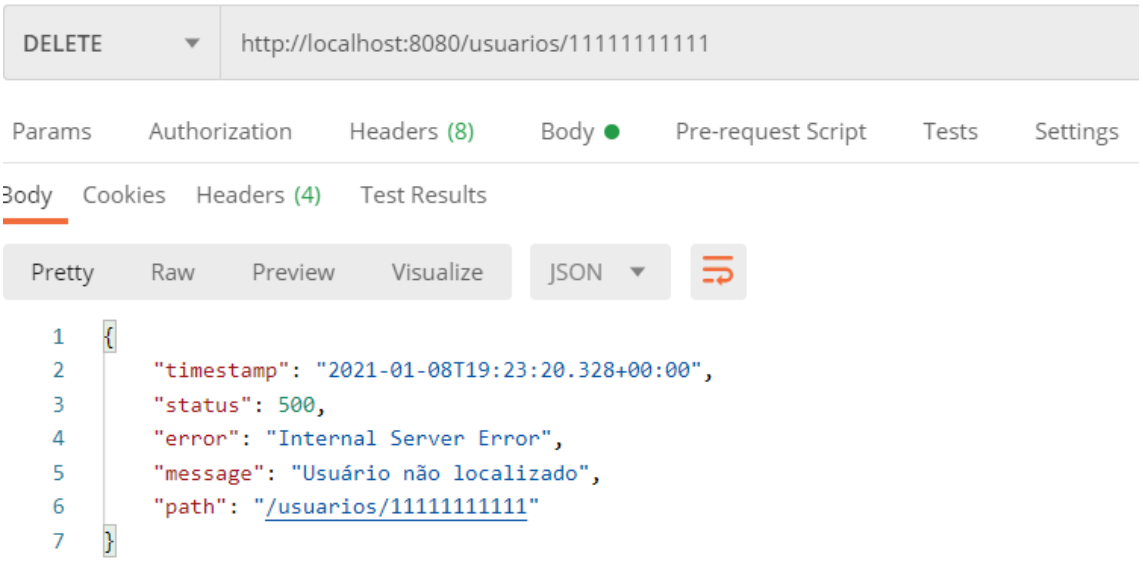


- Utilizando CPF, Nome e E-mail fora do padrão, mesmas validações do POST:

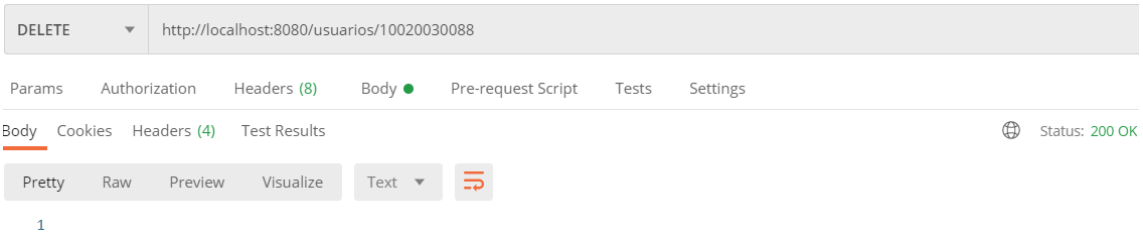


Teste HTTP DELETE

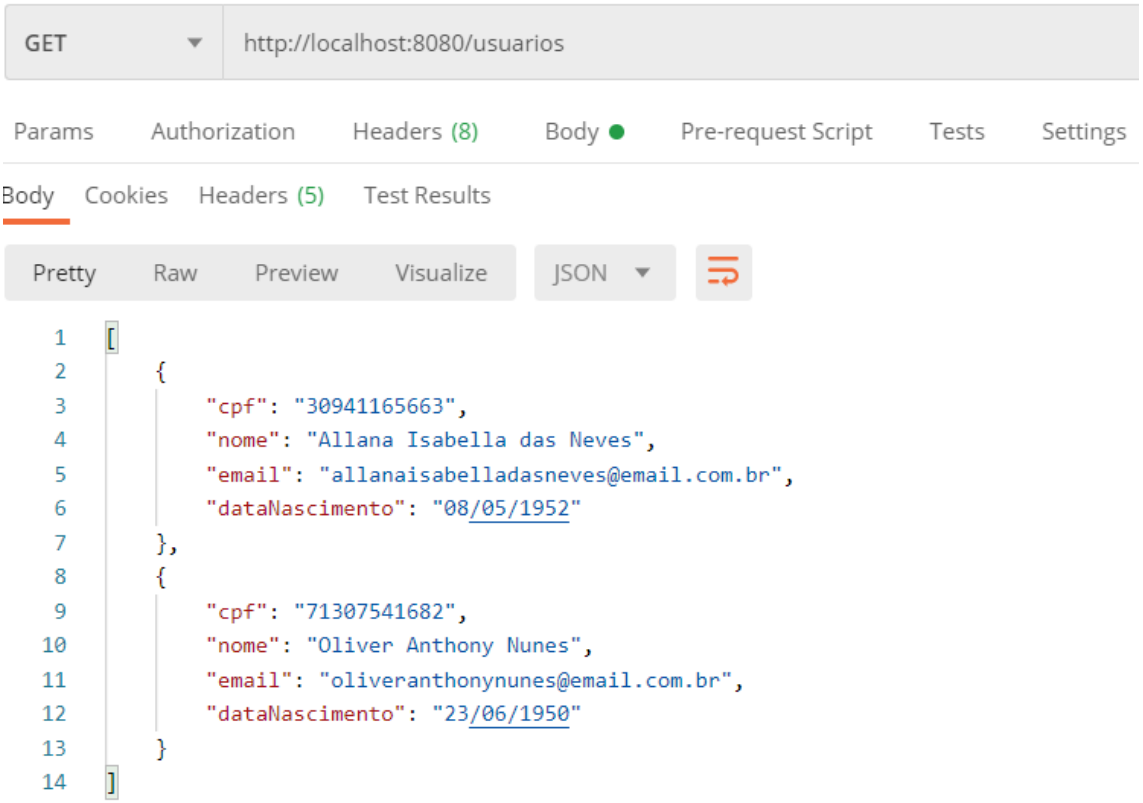
- Utilizando CPF incorreto ou não existente:



- Utilizando CPF existente no Banco de Dados, retorna HTTP Status 200 (Ok):



- Após utilizar o DELETE, verificamos que o usuário realmente foi apagado do Banco de Dados:



- Por último, ao efetuar qualquer requisição com método diferente do mapeado pelo endpoint, será retornado a seguinte mensagem de erro: