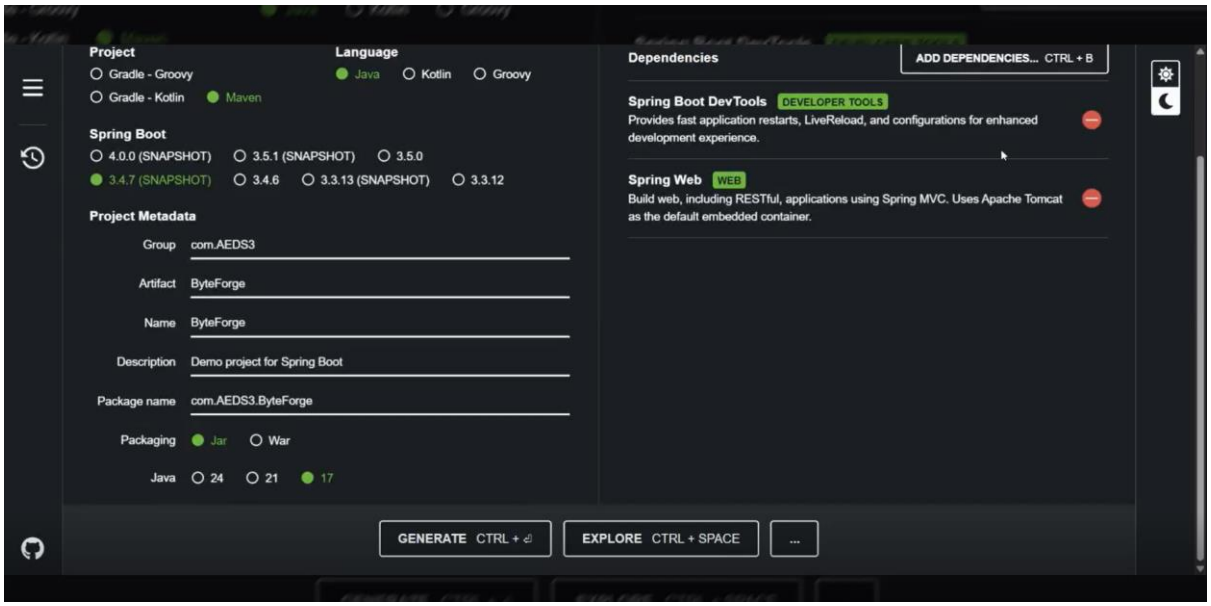
Tutorial CRUD WEB

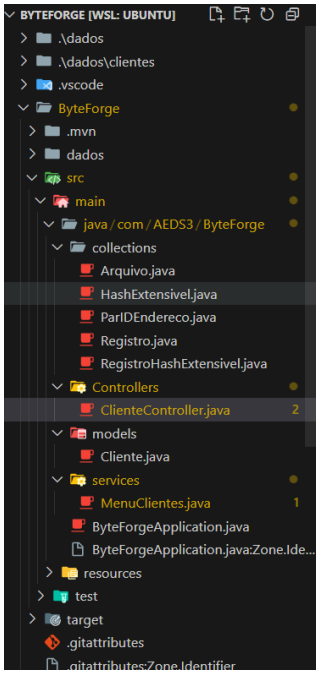
Acesse o site oficial do https://start.spring.io/ e configure o projeto com as seguintes dependências, conforme os requisitos da aplicação. A definição do nome do projeto fica a critério do desenvolvedor.



pós configurar o projeto no Spring Initializr, clique em **"Generate"** para que a plataforma gere um arquivo compactado (`.zip`) contendo a estrutura inicial da aplicação Spring Boot.

Esse arquivo deverá ser extraído em um diretório apropriado de trabalho. Em seguida, o desenvolvedor deverá **adicionar manualmente o código referente às operações CRUD previamente implementadas na disciplina de AEDs III**, adaptando-o à estrutura proposta pelo Spring Boot, de modo a integrá-lo adequadamente com as camadas de controle (`controller`), serviço (`service`) e persistência de dados, conforme os padrões recomendados pela arquitetura RESTful.

A pasta denominada `dados` deverá ser posicionada na raiz do projeto. Recomenda-se a criação de uma subpasta específica destinada ao armazenamento das estruturas de dados utilizadas. É importante ressaltar a necessidade de atenção às importações de arquivos, pois algumas IDEs realizam esse processo automaticamente, enquanto em outras será necessário proceder manualmente com os ajustes.



A classe Menu, anteriormente responsável pela interação com o usuário, será reestruturada para atuar como uma **classe de serviço**. Para isso, deve-se adicionar a anotação `@Service` sobre a declaração da classe, o que permitirá ao Spring Boot reconhecer sua finalidade no contexto da aplicação. Além disso, o construtor da classe deverá ser modificado para se adequar ao mecanismo de injeção de dependência fornecido pelo framework, conforme demonstrado no exemplo abaixo:

```java
@Service
public class MenuClientes {

    Arquivo<Cliente> arqClientes;
    private static Scanner console = new Scanner(System.in);

    @PostConstruct
    public void init() {
        try {
            arqClientes = new Arquivo<>(na:"clientes", Cliente.class.getConstructor());
        } catch (Exception e) {
            throw new RuntimeException("Erro ao inicializar Arquivo<Cliente>", e);
        }
    }
```

Posteriormente, deve-se criar um diretório denominado `controller`, cuja responsabilidade será o gerenciamento das requisições HTTP enviadas à API. A classe definida neste pacote deverá conter a anotação `@RestController`, indicando seu papel como componente controlador. Além disso, será especificado o endpoint principal por meio da anotação `@RequestMapping`. Dentro desta classe, a dependência do serviço será injetada com o uso da anotação `@Autowired`, conforme

ilustrado a seguir:

```java
@RestController
@RequestMapping("/clientes")
public class ClienteController {

    private final MenuClientes menuClientes;

    @Autowired
    public ClienteController(MenuClientes menuClientes) {
        this.menuClientes = menuClientes;
    }
```

A seguir, será realizada a adaptação do método responsável pela adição de novos clientes. Esse método deverá receber um objeto do tipo `Cliente`, persistir a entidade na base de dados, e retornar o mesmo objeto como confirmação da operação:

```java
public Cliente incluirCliente(Cliente cliente) {
    try {
        arqClientes.create(cliente);
        System.out.println("Cliente incluído com sucesso.");
        return cliente; // Retorna o cliente incluído
    } catch (Exception e) {
        System.out.println("Erro do sistema. Não foi possível incluir o cliente!");
        return null;
    }
}
```

No controlador, será declarado um método de requisição do tipo POST, o qual receberá uma entidade `Cliente` via corpo da requisição no formato JSON, repassando-a à classe de serviço responsável pelo processamento.

A partir desse ponto, os demais métodos da aplicação também deverão ser reformulados para eliminar qualquer forma de interação via terminal (como `Scanner` e `System.out.println`). Isso permitirá uma integração mais apropriada com a API REST e facilitará a realização de chamadas via ferramentas de requisição HTTP.

```java
@PostMapping("/adicionar")
public Cliente adicionar(@RequestBody Cliente entity) {
    try {
        menuClientes.incluirCliente(entity);
    } catch (Exception e) {
        throw new RuntimeException("Erro ao adicionar cliente: " + e.getMessage(), e);
    }
    return entity;
}
```

Vamos agora refumular os demais codigos, para tirar qualquer interação com o usuario e facilitar na chamada da nossa api:

```java
public List<Cliente> listarClientes() {
    List<Cliente> clientes = new ArrayList<>();
    try {
        clientes = arqClientes.readAll(); // Lê todos os clientes do arquivo
        if (clientes.isEmpty()) {
            System.out.println("Nenhum cliente cadastrado.");
        } else {
            System.out.println("\nLista de Clientes:");
            for (Cliente cliente : clientes) {
                mostraCliente(cliente); // Exibe os detalhes de cada cliente
            }
        }
    } catch (Exception e) {
        System.out.println("Erro ao listar clientes: " + e.getMessage());
    }
    return clientes;
}
```

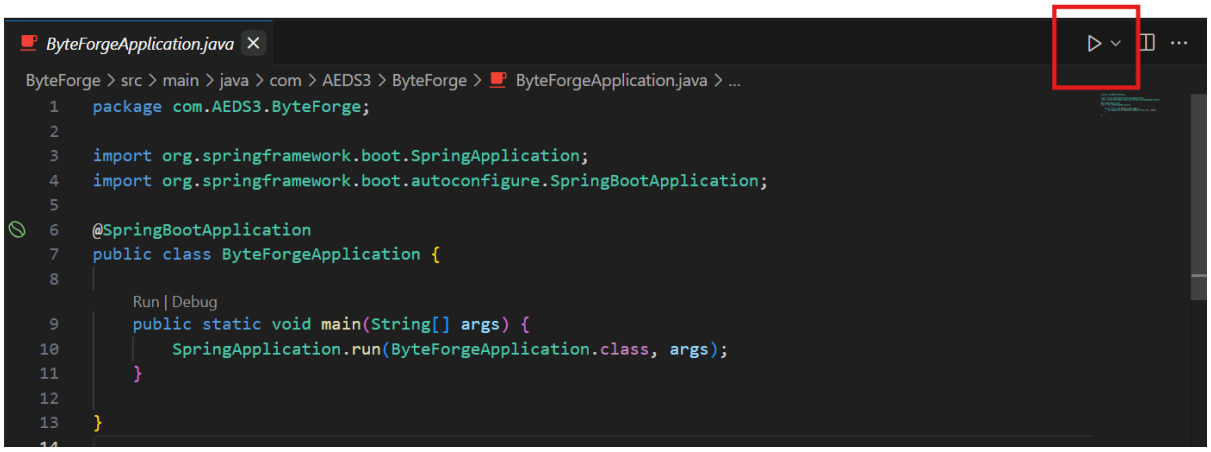
```java
public boolean alterarCliente(int id, Cliente clienteNovo) {
    if (id > 0) {
        try {
            // Tenta ler o cliente com o ID fornecido
            Cliente cliente = arqClientes.read(id);
            if (cliente != null) {
                System.out.println("Cliente encontrado:");
                mostraCliente(cliente);  // Exibe os dados do cliente para confirmação
                    boolean alterado = arqClientes.update(clienteNovo);
                        if (alterado) {
                            return true; // Retorna true se a alteração foi bem-sucedida
                        } else {
                            return false; // Retorna false se a alteração falhou
                        }
                    }
            else {
                System.out.println("Cliente não encontrado.");
                return false; // Retorna false se o cliente não foi encontrado
            }
        } catch (Exception e) {
            System.out.println("Erro do sistema. Não foi possível alterar o cliente!");
            e.printStackTrace();
        }
    }
    return false; // Retorna false se o ID for inválido
}
```

```java
public boolean excluirCliente(int id) {
    if (id > 0) {
        try {
            // Tenta ler o cliente com o ID fornecido
            Cliente cliente = arqClientes.read(id);
            if (cliente != null) {
                System.out.println("Cliente encontrado:");
                mostraCliente(cliente); // Exibe os dados do cliente para confirmação

                    boolean excluido = arqClientes.delete(id); // Chama o método de exclusão no arquivo
                    if (excluido) {
                        System.out.println("Cliente excluído com sucesso.");
                        return true; // Retorna true se a exclusão foi bem-sucedida
                    } else {
                        System.out.println("Erro ao excluir o cliente.");
                        return false; // Retorna false se a exclusão falhou
                    }
            } else {
                System.out.println("Cliente não encontrado.");
            }
        } catch (Exception e) {
            System.out.println("Erro do sistema. Não foi possível excluir o cliente!");
            e.printStackTrace();
        }
    }
    return false; // Retorna false se o ID for inválido ou se ocorrer um erro
}
```

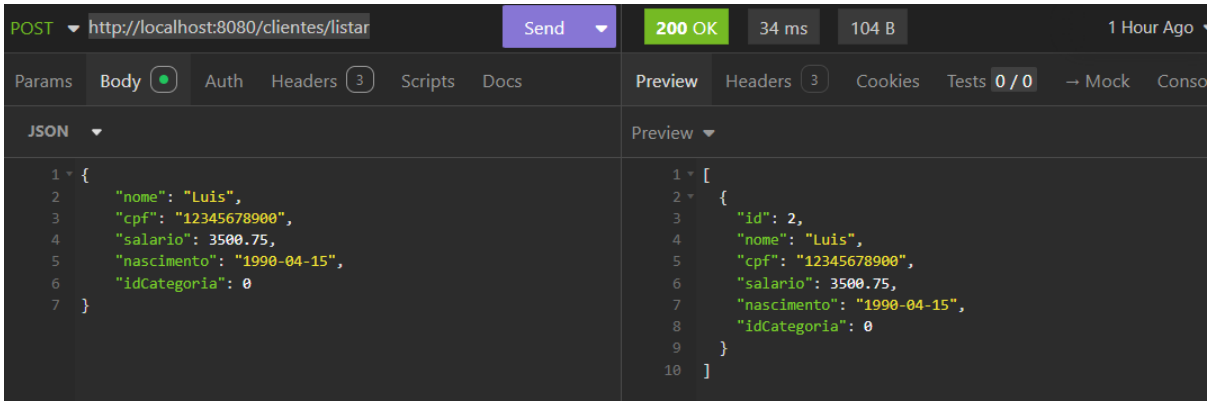
OUTPUT DEBUG CONSOLE TERMINAL PORTS

```java
@GetMapping("/listar")
public List<Cliente> listarClientes() {
    return menuClientes.listarClientes();
}

@PostMapping("/adicionar")
public Cliente adicionar(@RequestBody Cliente entity) {
    try {
        menuClientes.incluirCliente(entity);
    } catch (Exception e) {
        throw new RuntimeException("Erro ao adicionar cliente: " + e.getMessage(), e);
    }
    return entity;
}

@DeleteMapping("/remover")
public ResponseEntity<String> remover(@RequestParam int id) {
    try {
        if(menuClientes.excluirCliente(id)){
            return new ResponseEntity<>("Cliente removido com sucesso.", HttpStatus.OK);
        }else {
            return new ResponseEntity<>("Cliente não encontrado.", HttpStatus.NOT_FOUND);
        }
    } catch (Exception e) {
        return new ResponseEntity<>("Erro ao remover cliente: " + e.getMessage(), HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

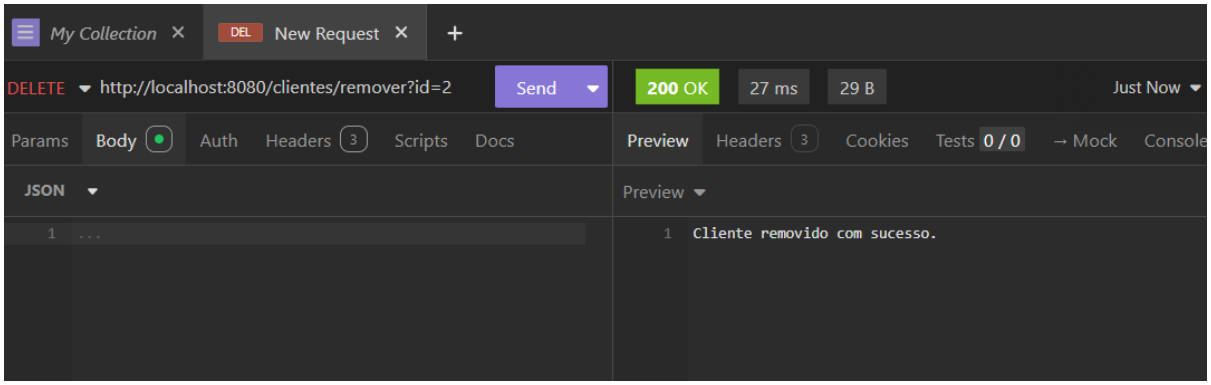
Para executar e testar a aplicação, deve-se iniciar a classe principal do projeto (`@SpringBootApplication`). O servidor local será iniciado, por padrão, na porta `8080`, exceto se esta já estiver ocupada por outro serviço.



Para realizar testes nas rotas da API, recomenda-se o uso de ferramentas como **Insomnia**, **Postman** ou similares, que permitem simular requisições HTTP de forma prática.



Ao efetuar uma requisição POST, por exemplo, a instância da entidade `Cliente` será devidamente persistida na base de dados. Ressalta-se que, conforme os princípios das APIs RESTful, todas as requisições do tipo POST devem conter o corpo no formato **JSON**. Já as requisições de exclusão (DELETE) deverão receber o identificador do recurso como parâmetro na URL, conforme ilustrado no exemplo abaixo:



Pronto, só fazer seu frontend chamando os endpoints necessários, exemplo:

```
const API_BASE = 'http://localhost:8080/clientes';

async function loadClients() {
  try {
    const response = await fetch(`${API_BASE}/listar`);
    if (!response.ok) throw new Error('Erro ao carregar clientes');
    clients = await response.json();
  } catch (error) {
    console.error(error);
    clients = [];
  }
}
```

Qualquer duvida, segue o link do repositorio:
https://github.com/LuisHenriqueDC301/ByteForge