

modelmapper

Simple, Intelligent, Object Mapping.

FRAMEWORKS DE
MAPEAMENTO
PARA JAVA

Integrantes:

Adriano Carvalho

Álvaro Claro

Clodoaldo Barbosa

Jaílson Ribeiro

Ricardo Barcelar

Zenildo Crisóstomo



“[...] Quem trabalha com software sabe, as coisas começam simples, e aí o negócio vai se transformando em uma bola de neve gigantesca, é código que não acaba mais [...]”

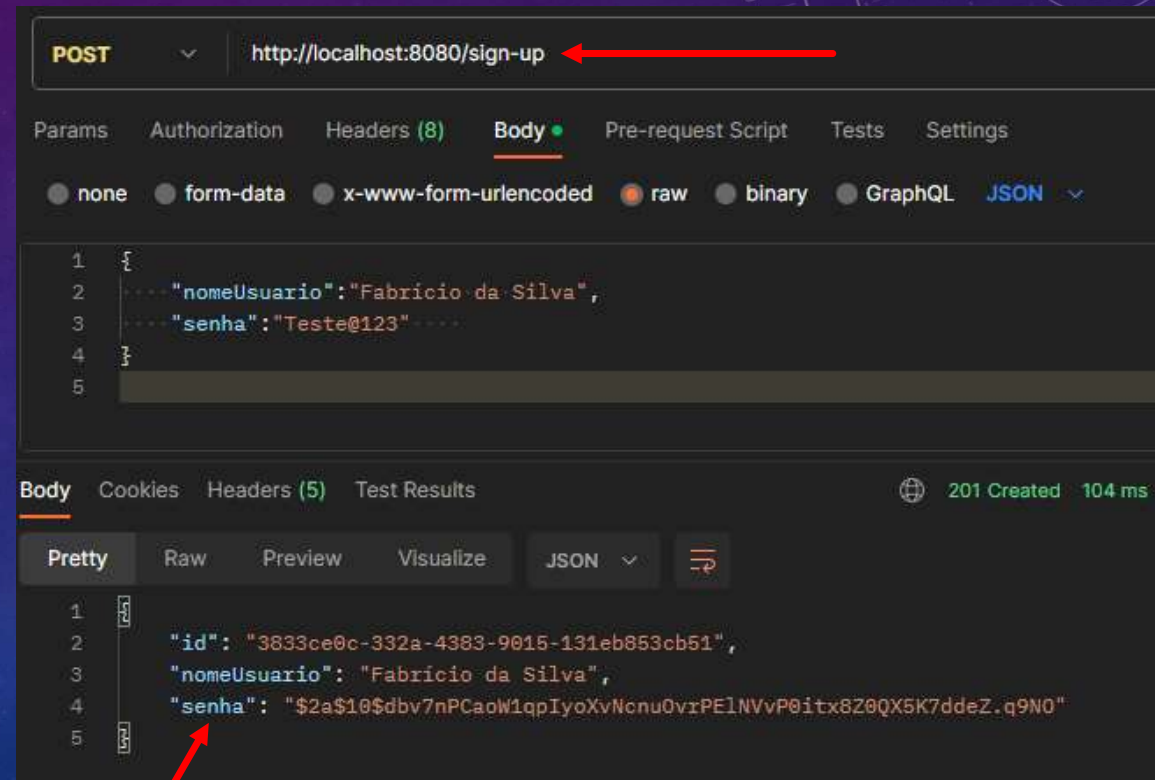
Claudia Gomes, 5º Andar

INTRODUÇÃO

- Criando soluções usando o paradigma de Orientação a Objetos, em algum momento após um processamento surgirá a necessidade de retornar dados
- Esses dados existem na forma de atributos contidos em entidades, que são objetos instanciados de classes, modeladas de acordo com o domínio do problema computacional a ser resolvido
- A troca dos dados entre esses objetos, bem como a resposta a um pedido externo, como o acesso a um endpoint que faz parte de uma API RESTFull, geralmente, não podem ou não devem ter todos os seus atributos visíveis ou expostos

NA PRÁTICA

- Uma entidade Usuario, possuindo 3 atributos básicos: id, nome do usuário e senha
- Num eventual cadastro de usuário pela internet, deve ser devolvido os dados cadastrados, porém a senha não deve aparecer



NA PRÁTICA

- Código de cadastro de usuário (/sign-up) devolvendo a entidade usuario com todos os seus atributos

```
@RestController
@RequestMapping("/sign-up")
public class UsuariosController {

    @Autowired
    private UsuariosRepository usuariosRepository;

    public UsuariosController(UsuariosRepository usuariosRepository) {
        this.usuariosRepository = usuariosRepository;
    }


    @PostMapping
    public ResponseEntity<Usuario> criar(@RequestBody @Valid Usuario usuario) {

        BCryptPasswordEncoder bcrypt = new BCryptPasswordEncoder();
        String senhaCriptografada = bcrypt.encode(usuario.getSenha());

        usuario.setId(UUID.randomUUID().toString());
        usuario.setSenha(senhaCriptografada);

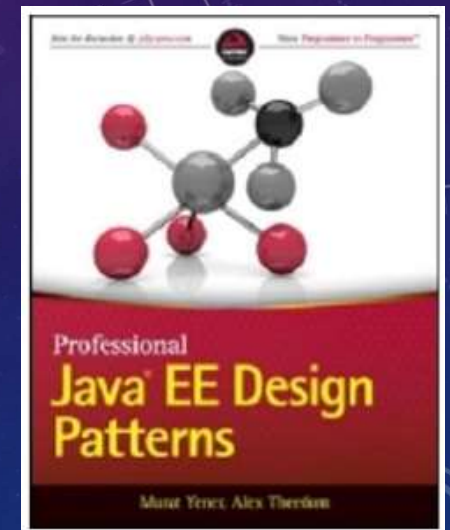
        usuariosRepository.save(usuario);

        return ResponseEntity.status(201).body(usuario);
    }
}
```



O PADRÃO DATA TRANSFER OBJECT (DTO)

- DTO ou Objeto de Transferência de Dados
- *O DTO carrega os dados recuperados de ou persistidos em um banco de dados através das camadas lógicas*
- *Algumas vezes nem todos os dados recuperados de um banco de dados são necessários na camada web ou em qualquer que seja a camada que necessite usar os dados*



Professional Java EE Design Patterns
Murat Yener, Alex Theedom
(página 155)

MAPEAMENTO DE DADOS

- Para resolver o problema citado anteriormente passou-se a utilizar o padrão DTO
- Na prática: a solução consiste em mapear (copiar) os dados do objeto completo para um outro objeto contendo somente os atributos que devem ser devolvidos ou expostos durante um processamento
- Para ajudar a identificar essas classes mapeadoras, convencionou-se que as mesmas tenham o termo DTO em seu nome, seja como prefixo ou sufixo

MAPEAMENTO DE DADOS – NA PRÁTICA

Código de cadastro de usuário (/sign-up) devolvendo a entidade usuario somente os atributos necessários usando DTO

```
@RestController
@RequestMapping("/sign-up")
public class UsuariosController {

    @Autowired
    private UsuariosRepository usuariosRepository;

    public UsuariosController(UsuariosRepository usuariosRepository) {
        this.usuariosRepository = usuariosRepository;
    }

    @PostMapping
    public ResponseEntity<UsuarioDto> criar(@RequestBody @Valid Usuario usuario) {

        BCryptPasswordEncoder bcrypt = new BCryptPasswordEncoder();
        String senhaCriptografada = bcrypt.encode(usuario.getSenha());

        usuario.setId(UUID.randomUUID().toString());
        usuario.setSenha(senhaCriptografada);

        usuariosRepository.save(usuario);

        UsuarioDto usuarioDto = new UsuarioDto(usuario.getId(), usuario.getNomeUsuario());

        return ResponseEntity.status(201).body(usuarioDto);
    }
}
```

```
public class UsuarioDto {

    private String id;
    private String nomeUsuario;

    public UsuarioDto(String id, String nomeUsuario) {
        this.id = id;
        this.nomeUsuario = nomeUsuario;
    }

    public String getId() {
        return id;
    }

    public String getNomeUsuario() {
        return nomeUsuario;
    }

}
```

Body Cookies Headers (5) Test Results 201 Created

Pretty Raw Preview Visualize JSON

```
1
2  "id": "c2ef7a7a-000c-4e1e-84c8-b3220b3e3546",
3  "nomeUsuario": "Fabrício da Silva"
4
```


PROBLEMA: ENTIDADES GRANDES

- A transferência dos dados de um objeto para outro é feita por meio de **getters** e **setters**, ou ainda, via **constructor** durante o processo de instanciação
- Desta forma em aplicações simples onde temos poucos atributos é tranquilo fazer mapeamento de dados com DTOs
- Mas em algumas aplicações, onde existem 2 ou mais entidades, e cada uma contendo mais de 10 atributos, é necessário se pensar em alguma outra solução

PROBLEMA: ENTIDADES GRANDES – NA PRÁTICA

- Caso da vida real: entidade Pessoa contendo 7 atributos:
 - Id
 - Nome
 - CPF
 - RG
 - Sobrenome
 - Nascimento
 - Sexo
- Emitir uma lista de pessoas cadastradas devendo possuir somente 3 atributos:
 - Id
 - Nome
 - CPF

PROBLEMA: ENTIDADES GRANDES – NA PRÁTICA

- Usando getters/setters

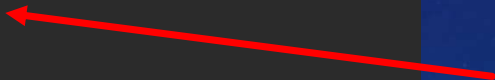
```
@RestController
@RequestMapping("/pessoas")
public class PessoasController {

    @Autowired
    private PessoasRepository pessoasRepository;

    @GetMapping
    public ResponseEntity<List<PessoaDto>> listar() {
        List<Pessoa> listaPessoas = pessoasRepository.findAll();

        List<PessoaDto> listaPessoasDto = listaPessoas.stream().map(pessoa -> {
            PessoaDto pessoaDto = new PessoaDto();
            pessoaDto.setId(pessoa.getId());
            pessoaDto.setNome(pessoa.getNome());
            pessoaDto.setCpf(pessoa.getCpf());
            return pessoaDto;
        }).toList();

        return ResponseEntity.status(200).body(listaPessoasDto);
    }
}
```



PROBLEMA: ENTIDADES GRANDES – NA PRÁTICA

- Usando construtor


```
@RestController
@RequestMapping("/pessoas")
public class PessoasController {

    @Autowired
    private PessoasRepository pessoasRepository;

    @GetMapping
    public ResponseEntity<List<PessoaDto>> listar() {
        List<Pessoa> listaPessoas = pessoasRepository.findAll();

        List<PessoaDto> listaPessoasDto = listaPessoas.stream().map(pessoa ->
            new PessoaDto(pessoa.getId(), pessoa.getNome(), pessoa.getCpf())).toList();

        return ResponseEntity.status(200).body(listaPessoasDto);
    }
}
```



PROBLEMA: ENTIDADES GRANDES – NA PRÁTICA

- Seja com getters e setters ou com construtores, a quantidade de código aumenta de acordo com a quantidade de atributos que precisa ser enviada/recebida, sejam internas entre objetos, seja externa atendendo a pedidos de aplicações

SOLUÇÃO

- Surge então **Frameworks de Mapeamento (mapping frameworks)**
- Na atualidade existem vários, porém abordaremos 2:

modelmapper
Simple, Intelligent, Object Mapping.



MAPSTRUCT

- Aplicações multi-camada com frequência necessitam mapear dados entre diferentes objetos do tipo models (como exemplo mapeamento entre Entidades e DTOs)
- Escrever código de mapeamento é tedioso e propenso a erros, MapStruct simplifica este trabalho por automatizar esse trabalho tanto quanto possível
- Em resumo: É um gerador de código que simplifica bastante as implementações de mapeamento de dados entre as classes Java

MAPSTRUCT

- O código gerado para fazer o mapeamento usa métodos de invocação simples e desta forma é rápido, com tipo seguro e fácil de entender
- É um processador de anotações que é plugado no compilador Java e pode ser usado nos builds de linha de comando (Maven, Gradle, etc) bem como de dentro da sua IDE preferida
- Em contraste com outros frameworks de mapeamento, MapStruct gera classes de mapeamento em tempo de compilação que assegura uma alta performance, permitindo ao desenvolvedor um rápido feedback e checagem de erros

MAPSTRUCT – USO DURANTE IMPLEMENTAÇÃO

```
@Mapper(componentModel = "spring")
public interface PessoaMapper {

    PessoaDto toDto(Pessoa pessoa);

    Pessoa toEntity(PessoaDto pessoaDto);

}
```

```
@RestController
@RequestMapping("/pessoas")
public class PessoasController {


    @Autowired
    private PessoasRepository pessoasRepository;

    @Autowired
    private PessoaMapper pessoaMapper;

    @GetMapping
    public ResponseEntity<List<PessoaDto>> listar() {
        List<Pessoa> listaPessoas = pessoasRepository.findAll();

        List<PessoaDto> listaPessoasDto = listaPessoas.stream().map(pessoa ->
            pessoaMapper.toDto(pessoa)
        ).toList();

        return ResponseEntity.status(200).body(listaPessoasDto);
    }
}
```



MAPSTRUCT – USO DURANTE IMPLEMENTAÇÃO

- Ao lado o código gerado na pasta target pelo processador do MapStruct da interface PessoaMapper, vista no slide anterior

```
@Component
public class PessoaMapperImpl implements PessoaMapper {
    public PessoaMapperImpl() {
    }

    public PessoaDto toDto(Pessoa pessoa) {
        if (pessoa == null) {
            return null;
        } else {
            PessoaDto pessoaDto = new PessoaDto();
            pessoaDto.setId(pessoa.getId());
            pessoaDto.setNome(pessoa.getNome());
            pessoaDto.setCpf(pessoa.getCpf());
            return pessoaDto;
        }
    }

    public Pessoa toEntity(PessoaDto pessoaDto) {
        if (pessoaDto == null) {
            return null;
        } else {
            Pessoa pessoa = new Pessoa();
            pessoa.setId(pessoaDto.getId());
            pessoa.setNome(pessoaDto.getNome());
            pessoa.setCpf(pessoaDto.getCpf());
            return pessoa;
        }
    }
}
```

MAPSTRUCT

- Ele ainda consegue auxiliar no processo de redefinição dos nomes dos atributos mapeados
- Também auxilia em processos de conversão, caso algum atributo precise, como é o caso das datas internacionais, por exemplo: entra uma data *mm-dd-yyyy* precisa guardar *dd-mm-yyyy*

MODELMAPPER

- Definição
- Argumentos/parâmetros

MODELMAPPER

- Exemplos de como usar

PERFORMANCE

- Apresentar aqui a performance entre os 2 frameworks via benchmark apresentado em

<https://www.baeldung.com/java-performance-mapping-frameworks>

CONSIDERAÇÕES FINAIS

- Algum parecer final sobre o assunto

Fim

Obrigado!