

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 [...]”

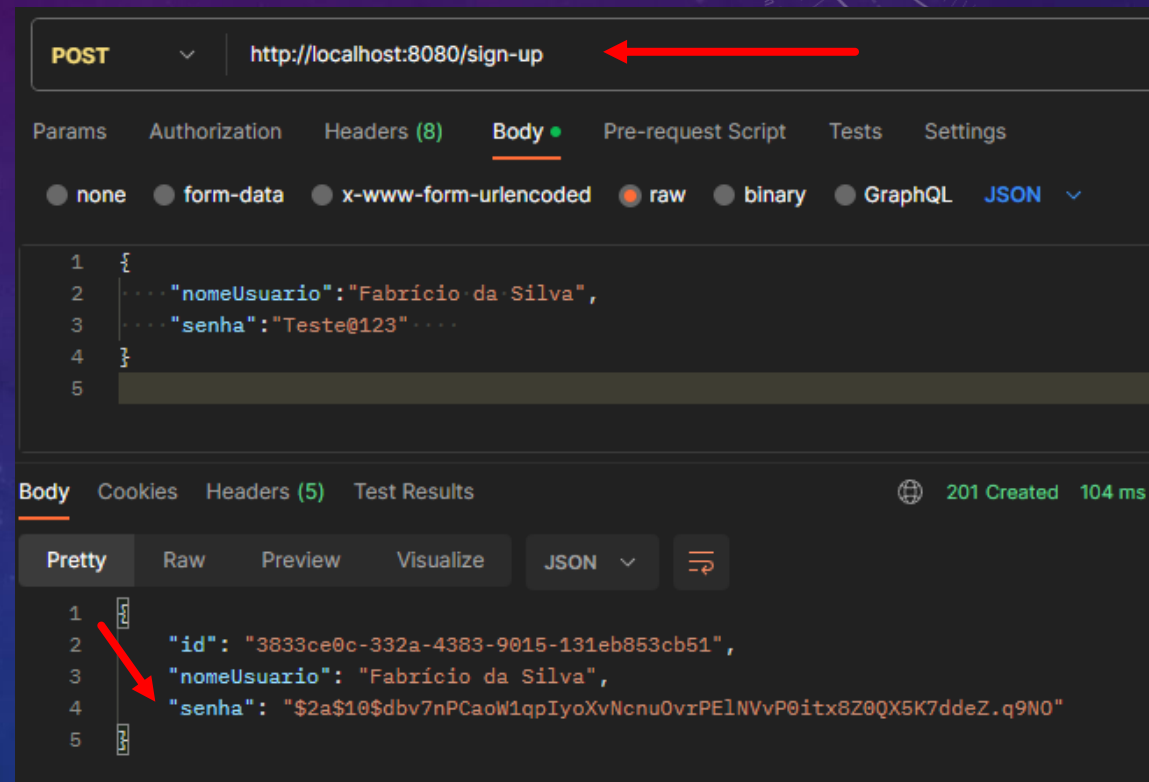
Claudia Gomes, 5º Andar

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);

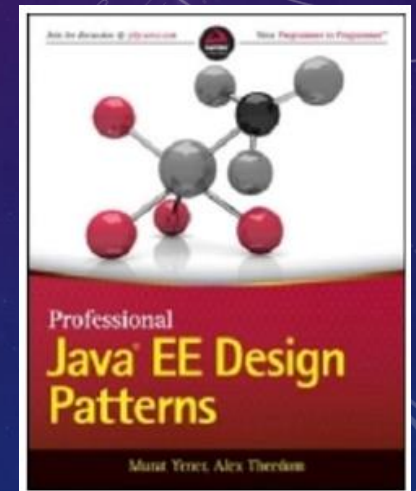
        usuariosRepository.save(usuario);

        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 (os dados expostos) pode-se 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

- Classe da Entidade Usuario, bem como o seu DTO

```
@Entity
@Table(name = "usuarios")
public class Usuario {

    @Id
    private String id;

    @NotEmpty
    @Column(name = "nome_usuario")
    private String nomeUsuario;

    @NotEmpty
    private String senha;
```

```
public class UsuarioDto {

    private String id;
    private String nomeUsuario;
```


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
 - Data Nascimento

PROBLEMA: ENTIDADES GRANDES – NA PRÁTICA

- Classe da Entidade Pessoa, bem como o seu DTO

```
@Entity
@Table(name = "pessoas")
public class Pessoa {
    @Id
    private Long id;
    private String nome;
    private Long cpf;
    private String rg;
    private String sobrenome;
    private LocalDate nascimento;
    private ESexo sexo;

    @Enumerated(EnumType.STRING)
    @Column(name = "enum_sexo", nullable = false)
    private ESexo enumSexo;
```

```
public class PessoaDto {
    private Long id;
    private String nome;
    private String cpf;
    private String nascimento;
```

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().toString());
            pessoaDto.setNascimento(pessoa.getNascimento().toString());
            return pessoaDto;
        }).toList();

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

```
public class PessoaDto {
    private Long id;
    private String nome;
    private String cpf;
    private String nascimento;
```

```
{
  "id": 1,
  "nome": "Fabrício",
  "cpf": "12312312387",
  "nascimento": "1955-08-08"
},
{
  "id": 2,
  "nome": "Mariana",
  "cpf": "6543404043",
  "nascimento": "1980-03-03"
}
```

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().toString(), pessoa.getNascimento().toString())
        ).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
- Vida real: projetos com mais de 20 atributos a serem devolvidos pelo backend

SOLUÇÃO

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

modelmapper

Simple, Intelligent, Object Mapping.





MAPSTRUCT

- Aplicações com frequência necessitam mapear dados entre diferentes objetos, 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: MapStruct 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 – INSTALAÇÃO NO MAVEN (POM.XML)

```
<properties>
  <org.mapstruct.version>
    1.5.5.Final
  </org.mapstruct.version>
</properties>
```

```
<dependency>
<groupId>org.mapstruct</groupId>
<artifactId>mapstruct</artifactId>
<version>${org.mapstruct.version}</version>
</dependency>
```

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.8.1</version>
  <configuration>
    <source>17</source>
    <target>17</target>
    <annotationProcessorPaths>
      <path>
        <groupId>org.mapstruct</groupId>
        <artifactId>mapstruct-processor</artifactId>
        <version>${org.mapstruct.version}</version>
      </path>
      <!-- other annotation processors -->
    </annotationProcessorPaths>
  </configuration>
</plugin>
```

MAPSTRUCT – INSTALAÇÃO NO GRADLE (BUILD.GRADLE)

```
dependencies {  
  
    implementation 'org.mapstruct:mapstruct:1.5.5.Final'  
  
    annotationProcessor 'org.mapstruct:mapstruct-processor:1.5.5.Final'  
}
```

MAPSTRUCT – IMPLEMENTAÇÃO

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

    PessoaDto toDto(Pessoa pessoa);

    Pessoa toEntity(PessoaDto pessoaDto);

}
```

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

    @Autowired
    private PessoasRepository pessoasRepository;

    @Autowired
    private PessoaMapper pessoaMapper;

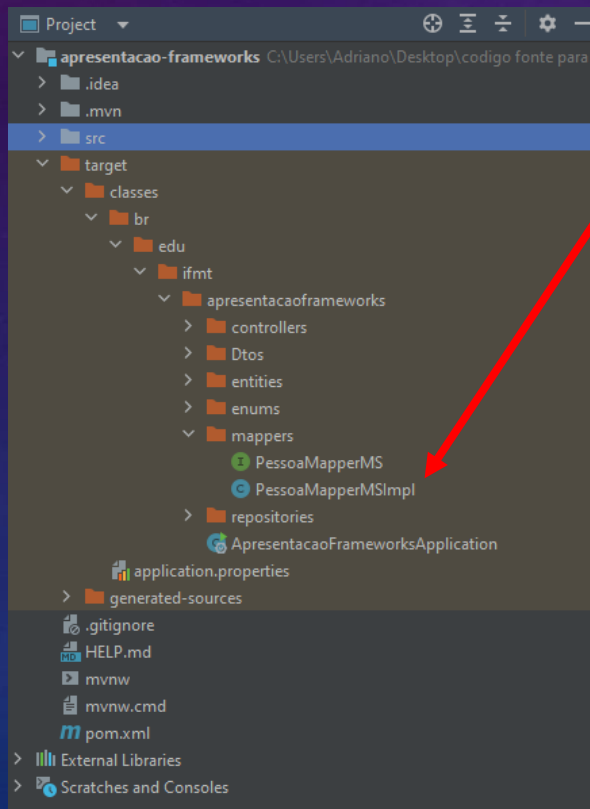
    @GetMapping
    public ResponseEntity<List<PessoaDto>> listar() {
        List<Pessoa> listaPessoas = pessoasRepository.findAll();

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

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

MAPSTRUCT – IMPLEMENTAÇÃO

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



```
@Component
public class PessoaMapperMSImpl implements PessoaMapperMS {
    public PessoaMapperMSImpl() {
    }

    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

- Se os atributos da classe origem (source) e destino (target) possuem nomes iguais, o mapeamento é feito de forma automática, basta chamar o mapeador e passar a entidade:

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

    PessoaDto toDto(Pessoa pessoa);

    Pessoa toEntity(PessoaDto pessoaDto);

}
```

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

    @Autowired
    private PessoasRepository pessoasRepository;

    @Autowired
    private PessoaMapperMS pessoaMapperMS;

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

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

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


MAPSTRUCT


- Se os atributos possuem nomes diferentes basta usar a anotação **@Mapping** para indicar quem são os atributos origem/destino (source/target):

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

    @Mapping(target = "sobrenome", source = "ultimoNome")
    PessoaDto toDto(Pessoa pessoa);

    Pessoa toEntity(PessoaDto pessoaDto);

}
```

A red V-shaped arrow diagram is positioned below the code. The left arm of the 'V' points from the 'ultimoNome' attribute in the source parameter of the '@Mapping' annotation to the 'toDto' method. The right arm points from the 'sobrenome' attribute in the target parameter of the '@Mapping' annotation to the 'toDto' method, illustrating the mapping from source to target.

MAPSTRUCT

- Também auxilia em processos de conversão, caso algum atributo precise, como é o caso das datas que no banco de dados, por exemplo são armazenadas em *yyyy-mm-dd* e precisam ser convertidas para mostrar *dd/mm/yyyy*

MAPSTRUCT

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

    @Mapping(target = "nascimento", dateFormat = "dd/MM/yyyy")
    PessoaDto toDto(Pessoa pessoa);

    Pessoa toEntity(PessoaDto pessoaDto);

}
```

```
{
  "id": 1,
  "nome": "Fabricio",
  "cpf": 12312312387,
  "nascimento": "08/08/1955"
},
{
  "id": 2,
  "nome": "Mariana",
  "cpf": 6543404043,
  "nascimento": "03/03/1980"
}
```

MAPSTRUCT

- Permite ainda dar nome a um método e chama-lo para fazer uma formatação customizada antes de fazer o mapeamento, como por exemplo a máscara do CPF

MAPSTRUCT

```
{
  "id": 1,
  "nome": "Fabrício",
  "cpf": "12312312387",
  "nascimento": "08/08/1955"
},
{
  "id": 2,
  "nome": "Mariana",
  "cpf": "6543404043",
  "nascimento": "03/03/1980"
}
```

```
{
  "id": 1,
  "nome": "Fabrício",
  "cpf": "123.123.123-87",
  "nascimento": "08/08/1955"
},
{
  "id": 2,
  "nome": "Mariana",
  "cpf": "065.434.040-43",
  "nascimento": "03/03/1980"
}
```

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

```
    @Mapping(target = "nascimento", dateFormat = "dd/MM/yyyy")
    @Mapping(target = "cpf", qualifiedByName = "formatarCpf")
    PessoaDto toDto(Pessoa pessoa);
```

```
    Pessoa toEntity(PessoaDto pessoaDto);
```

```
    @Named("formatarCpf")
```

```
    default String formatarCpf(Long cpf) {
        if (cpf == null) {
            return null;
        }
    }
```

```
        StringBuilder sBuilder = new StringBuilder(padLeftZeros(String.valueOf(cpf), 11));
```

```
        sBuilder.insert(3, ".");
        sBuilder.insert(7, ".");
        sBuilder.insert(11, "-");
```

```
        return sBuilder.toString();
    }
```

```
    default String padLeftZeros(String inputString, int length) {
        if (inputString.length() >= length) {
            return inputString;
        }
    }
```

```
        StringBuilder sb = new StringBuilder();
```

```
        while (sb.length() < length - inputString.length()) {
            sb.append('0');
        }
```

```
        sb.append(inputString);
```

```
        return sb.toString();
    }
```

```
}
```


modelmapper

Simple, Intelligent, Object Mapping.

MODELMAPPER

- Projetado para auxiliar de forma fácil no mapeamento de objetos
- Automaticamente determina como um objeto mapeia para o outro objeto, baseado em convenções, do mesmo jeito que um humano faria
- Analisa seu objeto model para determinar inteligentemente como os dados devem ser mapeados
- Não existe necessidade de mapeamento manual

MODELMAPPER

- Usa convenções para determinar como as propriedades e valores são mapeados um para o outro, sendo ainda que o usuário pode criar suas próprias convenções, ou usar uma das convenções fornecidas
- Para ser o mais automático possível, possui 3 tipos de estratégias de atribuição/combinção: **Standard**, **Loose** e **Strict**
- As operação que não são automáticas, como mudanças de formatação, precisam ser mapeadas, ou seja, ensinar o mapeador a fazer o trabalho
- Para ensinar o mapeador existem conceitos que precisam ser usados como:
 - **Converter**: conversores (métodos) que irão auxiliar no mapeamento
 - **TypeMap**: tipos de dados usados nos mapeamentos
 - **PropertyMap**: propriedade/atributo a ser usado nos mapeamentos

MODELMAPPER – INSTALAÇÃO NO MAVEN (POM.XML)

```
<dependency>  
  <groupId>org.modelmapper</groupId>  
  <artifactId>modelmapper</artifactId>  
  <version>3.2.0</version>  
</dependency>
```

MODELMAPPER – INSTALAÇÃO NO GRADLE (BUILD.GRADLE)

```
implementation group: 'org.modelmapper', name: 'modelmapper', version: '3.2.0'
```


MODELMAPPER

- Exemplos de como usar objeto a objeto:

```
ModelMapper modelMapper = new ModelMapper();  
  
modelMapper.map(pessoa, PessoaDto.class);
```

- Exemplos de como usar em lista de objetos:

```
ModelMapper modelMapper = new ModelMapper();  
  
List<PessoaDto> listaPessoasDto = listaPessoas.stream().map(  
    pessoa -> modelMapper.map(pessoa, PessoaDto.class)  
).toList();
```

MODELMAPPER

- O código ao lado mostra a configuração do mapeador
- Mostra também que o mapeador é chamado de dentro de uma iteração, executando o conversor sobre cada linha da tabela

```
{
  {
    "id": 1,
    "nome": "Fabrício",
    "cpf": "12312312387",
    "nascimento": "08/08/1955"
  },
  {
    "id": 2,
    "nome": "Mariana",
    "cpf": "6543404043",
    "nascimento": "03/03/1980"
  }
}
```

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

    @Autowired
    private PessoasRepository pessoasRepository;

    @Autowired
    private PessoaMapperMS pessoaMapperMS;

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

        ModelMapper modelMapper = new ModelMapper();

        Converter<LocalDate, String> toStringDate = new AbstractConverter<LocalDate, String>() {
            @Override
            protected String convert(LocalDate source) {

                if (source == null) {
                    return null;
                }

                DateTimeFormatter format = DateTimeFormatter.ofPattern("dd/MM/yyyy");
                return source.format(format);
            }
        };

        modelMapper.addConverter(toStringDate);
        modelMapper.getTypeMap(LocalDate.class, String.class).setConverter(toStringDate);

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

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

COMPARAÇÕES

The background is a dark blue gradient with faint, light blue technical diagrams. In the top right, there is a large circular gauge with concentric circles and radial markings, resembling a speedometer or a scale. In the bottom right, there is a smaller circular diagram with concentric circles and arrows indicating a clockwise direction. In the bottom left, there is another circular diagram with concentric circles and arrows indicating a counter-clockwise direction.

PRINCIPAIS DIFERENÇAS

modelmapper

Simple, Intelligent, Object Mapping.

- Mapeamento por convenção
- Mapeamento por Profundidade (atributos em níveis)
- Mapeamento condicional
- Atributos a serem ignorados podem ser configurados
- Conversores Embutidos
- Estratégias de Atribuição
- Permite fazer mapeamento por expressão (semelhante a manual)



- Mapeamento por atributos iguais
- Mapeamento Manual, em caso de nomes de atributos diferentes
- Mapeamento condicional
- Mapeamento por expressão
- Necessário criar uma interface para realizar o mapeamento

DADOS DO GITHUB

modelmapper

Simple, Intelligent, Object Mapping.

- Versão: 3.2.0 (16/10/2023)
- Estrelas: 2.2k
- Forks: 334
- Primeiro Commit: 18/07/2011
- Último Commit: 15/10/2023
- Qualidade do Código: L3*



- Versão: 1.5.5 (23/04/2023)
- Estrelas: 6.5k
- Forks: 889
- Primeiro Commit: 02/06/2013
- Último Commit: 04/11/2023
- Qualidade do Código: L4*

*Ranqueamento da Qualidade do Código são calculados e fornecidos por [Lumnify](https://lumnify.com/grades/?rel=libhunt-cmp), sendo que vária de L1 a L5, sendo o L5 o mais alto.

<https://lumnify.com/grades/?rel=libhunt-cmp>

PERFORMANCE

A preocupação aqui é quando se tem aplicações grandes, com muitas entidades e DTOs, fazendo persistências, recebendo e enviando objetos JSONs, podendo impactar na performance da aplicação

modelmapper

Simple, Intelligent, Object Mapping.

- Tempo Médio (ms): 0.002
- Operações por seg: 379
- Tempo Proc. Oper. Única: 8.788ms



- Tempo Médio (ms): 0.00001
- Operações por seg: 58101
- Tempo Proc. Oper. Única: 1.904ms

*Foi usado nesse teste de performance o JMH (Java Microbenchmark Harness), para maiores informações e acesso ao código fonte dos testes:

<https://www.baeldung.com/java-microbenchmark-harness>
<https://github.com/eugenp/tutorials/tree/master/performance-tests>

CONSIDERAÇÕES FINAIS

- Existem outras soluções: ORIKA, AUTOMAPPER, DOZER, SELMA, JMAPPER
- ModelMapper tem vulnerabilidades, e exige instalação do Dozer como dependência
- Dozer está inativo, e recomenda-se no site do Dozer usar o ModelMapper
- MapStruct é simples e performático
- ModelMapper é muito poderoso (canivete suíço para resolver problemas), porém a performance é baixa, sem falar na curva de aprendizagem/uso eficaz que é muito acentuada

REFERÊNCIAS

- <https://mapstruct.org/>
- <https://modelmapper.org/>
- <https://www.baeldung.com/java-microbenchmark-harness>

Fim

Obrigado!