**ModelMapper Что такое и зачем он нужен?**

ModelМapper просто использовать в начале проекта без особых знаний. Попытаюсь разобрать основные моменты работы с фреймворком.

1. [Как начать использовать ModelMapper?](https://tproger.ru/articles/chto-takoe-modelmapper-i-zachem-on-nuzhen#part1)
2. [Маппинг отдельных полей](https://tproger.ru/articles/chto-takoe-modelmapper-i-zachem-on-nuzhen#part2)
3. [Маппинг и наследование](https://tproger.ru/articles/chto-takoe-modelmapper-i-zachem-on-nuzhen#part3)
4. [Заключение](https://tproger.ru/articles/chto-takoe-modelmapper-i-zachem-on-nuzhen#part4)

В ходе разработки любого приложения программист сталкивается с необходимостью работать с моделями различных объектов, созданных для разных целей. И соответственно, с необходимостью конвертировать их между собой. Если ваш проект на начальном этапе развития, то можно, конечно, использовать рукописные конверторы. Но рано или поздно проект станет больше, и вы столкнётесь с необходимостью использовать уже готовое решение для конвертации моделей.

Одним из таких решений является МodelMapper. Его очень просто использовать в самом начале проекта без особых знаний. Попытаюсь разобрать основные моменты использования фреймворка.

**Как начать использовать ModelMapper?**

Сначала добавляем его в зависимости. Если вы используете maven, то:

<dependency>

<groupId>org.modelmapper</groupId>

<artifactId>modelmapper</artifactId>

<version>2.3.0</version>

</dependency>

Если gradle то:

implementation group: ‘org.modelmapper’, name: ‘modelmapper’, version: ‘2.3.0’

Предположим, у нас простой проект с базой данных и RestAPI и нам необходимо превратить entity в dto и обратно. На этапе прототипа проекта могут полностью совпадать, и в таком простейшем примере нам вообще не нужно ничего дополнительного писать. МodelMapper всё сделает за нас.

В примере, представленном ниже, я буду использовать аннотации Lombok, чтобы было проще =)

**Наши entity:**

package org.example.entity;

import lombok.Builder;

import lombok.Data;

import lombok.NoArgsConstructor;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.Id;

import javax.persistence.ManyToOne;

@Data

@Entity

@NoArgsConstructor

public class BookEntity {

    @Id

    @GeneratedValue

    private Long id;

    private String bookName;

    @ManyToOne

    private AuthorEntity author;

    private Integer pages;

    private String index;

    @Builder

    public BookEntity(Long id, String bookName, AuthorEntity author, Integer pages, String index) {

        this.id = id;

        this.bookName = bookName;

        this.author = author;

        this.pages = pages;

        this.index = index;

    }

}

import lombok.Builder;

import lombok.Data;

import lombok.NoArgsConstructor;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.Id;

@Entity

@Data

@NoArgsConstructor

public class AuthorEntity {

    @Id

    @GeneratedValue

    private Long id;

    private String name;

    @Builder

    public AuthorEntity(Long id, String name) {

        this.id = id;

        this.name = name;

    }

}

**Наша dto:**

package org.example.dto;

import lombok.Data;

@Data

public class BookDto {

    private String bookName;

    private String authorName;

    private Integer pages;

    private String index;

}

**Для того, чтобы начать маппить entity в dto нам достаточно написать вот такой простой конвертор:**

package org.example.convertor;

import org.example.dto.BookDto;

import org.example.entity.BookEntity;

import org.modelmapper.ModelMapper;

import org.springframework.stereotype.Component;

@Component

public class BookConvertor {

    private final ModelMapper modelMapper;

     public BookConvertor() {

        this.modelMapper = new ModelMapper();

    }

     public BookDto convertToDto(BookEntity entity){

        return modelMapper.map(entity,BookDto.class);

    }

}

**А для наполнения базы использовать следующие entity:**

AuthorEntity authorEntity1 = AuthorEntity.builder().name("Чарльз Диккенс").build();

AuthorEntity authorEntity2 = AuthorEntity.builder().name("Джейн Остин ").build();

AuthorEntity authorEntity3 = AuthorEntity.builder().name("Иоганн Вольфганг фон Гёте").build();

BookEntity bookEntity1 = BookEntity.builder()

        .bookName("Приключения Оливера Твиста")

        .author(authorEntity1)

        .pages(220)

        .index("ISBN: 978-5-91921-226-3")

        .build();

BookEntity bookEntity2 = BookEntity.builder()

        .bookName("Гордость и предубеждение")

        .author(authorEntity2)

        .pages(400)

        .index("ISBN: 978-5-699-52151-7")

        .build();

BookEntity bookEntity3 = BookEntity.builder()

        .bookName("Фауст")

        .author(authorEntity3)

        .pages(270)

        .index("ISBN: 5-699-07346-9")

        .build();

**И наш контроллер с одним единственным методом:**

package org.example.controller;

import lombok.RequiredArgsConstructor;

import org.example.convertor.BookConvertor;

import org.example.dto.BookDto;

import org.example.repositories.BookRepository;

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

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

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

import java.util.List;

import java.util.stream.Collectors;

@RestController

@RequestMapping("/book")

@RequiredArgsConstructor

public class BookController {

    private final BookRepository bookRepository;

    private final BookConvertor bookConvertor;

    @GetMapping

    public List findAllBooks(){

        return bookRepository.findAll()

                .stream()

                .map(bookConvertor::convertToDto)

                .collect(Collectors.toList());

    }

}

**После выполнения запроса http://localhost:8080/book мы получаем следующий ответ:**

[

{

"bookName": "Приключения Оливера Твиста",

"authorName": "Чарльз Диккенс",

"pages": 220,

"index": "ISBN: 978-5-91921-226-3"

},

{

"bookName": "Гордость и предубеждение",

"authorName": "Джейн Остин",

"pages": 400,

"index": "ISBN: 978-5-699-52151-7"

},

{

"bookName": "Фауст",

"authorName": "Иоганн Вольфганг фон Гёте",

"pages": 270,

"index": "ISBN: 5-699-07346-9"

}

]

МodelMapper по названию полей сам догадывается, что на что нужно маппить. Это очень удобно, если у вас есть множество моделек, которые в целом похожи друг на друга. Весь процесс можно разбить на две части: распознание и связь полей, а также перенос значений.

* По умолчанию ModelMapper ищет поля, помеченные как public, и использует JavaBeans Naming Convention, чтобы определить какие проперти соответствуют друг другу. Каждый шаг распознавания поля и связи с полем модели назначения можно настроить;
* AccessLevel — имеет следующие значения PUBLIC, PROTECTED, PACKAGE\_PRIVATE, PRIVATE;
* NamingConvention — имеет следующие значения JAVABEANS\_ACCESSOR, JAVABEANS\_MUTATOR, NONE JAVABEANS\_ACCESSOR ищет гетторы а JAVABEANS\_MUTATOR ищет сеттеры, так-же можно создать свой NamingConvention dspdfd NamingConvention.builder();
* NameTokenizers — имеет следующие значения CAMEL\_CASE, UNDERSCORE эта опция используется для глубокого маппинга, пример маппинга имени автора выше;
* MatchingStrategies — может быть STANDARD, LOOSE, STRICT по умолчанию стоит STANDARD.

Если описать работу маппера простыми словами, то: он сканирует поля в соответствии с AccessLevel, парсит их и бьёт на токены, сравнивая эти токены он пытается понять подходит ли поле для маппинга. Стратегии настраивают степень точности:

* STRICT — все токены должны быть в одном порядке, а также все токены модели источника должны совпадать с токенами модели получателя;
* STANDARD — порядок токенов может не совпадать, все токены цели должны совпадать и только один токен источника должен совпадать.;
* LOOSE — порядок токенов может не совпадать, только один токен модели источника и получателя должен совпадать.

**Пример настройки значений для ModelMapper:**

public BookConvertor() {

    this.modelMapper = new ModelMapper();

    Configuration configuration = modelMapper.getConfiguration();

    configuration.setFieldAccessLevel(Configuration.AccessLevel.PUBLIC);

    configuration.setSourceNamingConvention(NamingConventions.JAVABEANS\_ACCESSOR);

    configuration.setDestinationNamingConvention(NamingConventions.JAVABEANS\_MUTATOR);

    configuration.setSourceNameTokenizer(NameTokenizers.CAMEL\_CASE);

    configuration.setDestinationNameTokenizer(NameTokenizers.CAMEL\_CASE);

    configuration.setMatchingStrategy(MatchingStrategies.STANDARD);

}

Это далеко не все настройки МodelMapper, больше настроек можно посмотреть в классе InheritingConfiguration.

**Маппинг отдельных полей**

Для начинающего специалиста, показанного выше, вполне достаточно. Но для серьёзного приложения нужен больший контроль над маппингом определённых полей. Также было бы удобно маппить вложенные сущности. В этом разделе мы рассмотрим, как нам с этим поможет МodelMapper.

Давайте немного усложним наш маппинг. Предположим, что нам в нашем поле index не нужна подстрока ISBN:. Как нам изменить условия маппинга, чтобы для одного поля мы удаляли эту подстроку?

**Можно использовать Converter<S,D> :**

package org.example.convertor;

import org.example.dto.BookDto;

import org.example.entity.BookEntity;

import org.modelmapper.Converter;

import org.modelmapper.ModelMapper;

import org.springframework.stereotype.Component;

@Component

public class BookConvertor {

    private final ModelMapper modelMapper;

    private Converter<String, String> isbnRemover = (src) -> src.getSource().replaceAll("ISBN: ", "");

    public BookConvertor() {

        this.modelMapper = new ModelMapper();

        modelMapper.createTypeMap(BookEntity.class, BookDto.class)

                .addMappings(mapper -> mapper.using(isbnRemover).map(BookEntity::getIndex, BookDto::setIndex));

    }

    public BookDto convertToDto(BookEntity entity) {

        return modelMapper.map(entity, BookDto.class);

    }

}

В данном примере мы создали TypeMap для двух наших объектов и указали поле, для которого мы хотим использовать этот конвертер.

**Теперь наш запрос возвращает следующее:**

[

{

"bookName": "Приключения Оливера Твиста",

"authorName": "Чарльз Диккенс",

"pages": 220,

"index": "978-5-91921-226-3"

},

{

"bookName": "Гордость и предубеждение",

"authorName": "Джейн Остин",

"pages": 400,

"index": "978-5-699-52151-7"

},

{

"bookName": "Фауст",

"authorName": "Иоганн Вольфганг фон Гёте",

"pages": 270,

"index": "5-699-07346-9"

}

]

Также конвертер можно добавить и для всего МodelMapper, если написать modelMapper.addConverter(isbnRemover); . Тогда он будет применён для всех конвертаций String->String.

А также можно создать конвертер на весь TypeMap: modelMapper.createTypeMap(BookEntity.class, BookDto.class).setConverter(ctx->BookDto.builder().build());, но тогда будет возвращён пустой объект BookDto.

Мы научились модифицировать правила конвертации отдельных полей и целых объектов. С этим уже можно полноценно работать. Но бывают случаи, когда нам не нужно модифицировать значение, а необходимо просто связать два названных по-разному поля.

**Добавим в наши объекты поля: comment в BookEntity и review в BookDto и модифицируем наш BookConverter:**

package org.example.convertor;

import org.example.dto.BookDto;

import org.example.entity.BookEntity;

import org.modelmapper.Converter;

import org.modelmapper.ModelMapper;

import org.springframework.stereotype.Component;

@Component

public class BookConvertor {

    private final ModelMapper modelMapper;

    private Converter<String, String> isbnRemover = (src) -> src.getSource().replaceAll("ISBN: ", "");

    public BookConvertor() {

        this.modelMapper = new ModelMapper();

       modelMapper.createTypeMap(BookEntity.class, BookDto.class)

                .addMapping(BookEntity::getComment,BookDto::setReview)

                .addMappings(mapper -> mapper.using(isbnRemover).map(BookEntity::getIndex, BookDto::setIndex));

    }

    public BookDto convertToDto(BookEntity entity) {

        return modelMapper.map(entity, BookDto.class);

    }

}

**И тогда запрос будет выглядеть вот так:**

[

{

"bookName": "Приключения Оливера Твиста",

"authorName": "Чарльз Диккенс",

"pages": 220,

"index": "978-5-91921-226-3",

"review": "Отличный приключенчиский роман"

},

{

"bookName": "Гордость и предубеждение",

"authorName": "Джейн Остин ",

"pages": 400,

"index": "978-5-699-52151-7",

"review": "Занудная история про богатеев в Америке"

},

{

"bookName": "Фауст",

"authorName": "Иоганн Вольфганг фон Гёте",

"pages": 270,

"index": "5-699-07346-9",

"review": "Пища для ума"

}

]

Теперь при маппинге отдельных полей у нас будет меньше мороки.

А что, если нам нужно маппить ещё и вложенную сущность? Для этого мы снова модифицируем BookDto и добавляем туда поле author вместо authorName. А также создаём класс AuthorDto, содержащий только поле name.

**И наш BookConverter теперь будет выглядеть следующим образом:**

package org.example.convertor;

import org.example.dto.AuthorDto;

import org.example.dto.BookDto;

import org.example.entity.AuthorEntity;

import org.example.entity.BookEntity;

import org.modelmapper.Converter;

import org.modelmapper.ModelMapper;

import org.springframework.stereotype.Component;

@Component

public class BookConvertor {

    private final ModelMapper modelMapper;

    private Converter<String, String> isbnRemover = (src) -> src.getSource().replaceAll("ISBN: ", "");

    public BookConvertor() {

        this.modelMapper = new ModelMapper();

        modelMapper.createTypeMap(BookEntity.class, BookDto.class)

                .addMapping(BookEntity::getComment, BookDto::setReview)

                .addMappings(mapper -> mapper.using(isbnRemover).map(BookEntity::getIndex, BookDto::setIndex));

        modelMapper.createTypeMap(AuthorEntity.class, AuthorDto.class);

    }

    public BookDto convertToDto(BookEntity entity) {

        return modelMapper.map(entity, BookDto.class);

    }

}

**А в ответе на запрос книг получаем:**

[

{

"bookName": "Приключения Оливера Твиста",

"pages": 220,

"index": "978-5-91921-226-3",

"review": "Отличный приключенчиский роман",

"author": {

"name": "Чарльз Диккенс"

}

},

{

"bookName": "Гордость и предубеждение",

"pages": 400,

"index": "978-5-699-52151-7"

"review": "Занудная история про богатеев в Америке",

"author": {

"name": "Чарльз Диккенс"

}

},

{

"bookName": "Фауст",

"pages": 270,

"index": "5-699-07346-9",

"review": "Пища для ума",

"author": {

"name": "Чарльз Диккенс"

}

}

]

**Маппинг и наследование**

Самое просто мы разобрали ранее. Теперь давайте посмотрим, как же ModelМapper работает с наследованием. Для этого мы изменим модель наших данных, добавив наследников для книг.

**Теперь наша модель будет выглядеть так:**

@Data

@Entity

@Inheritance(strategy = InheritanceType.SINGLE\_TABLE)

@NoArgsConstructor

public class BookEntity {

    @Id

    @GeneratedValue

    private Long id;

    private String bookName;

    @ManyToOne

    private AuthorEntity author;

    private String index;

    private String comment;

    public BookEntity(Long id, String bookName, AuthorEntity author, String index, String comment) {

        this.id = id;

        this.bookName = bookName;

        this.author = author;

        this.index = index;

        this.comment = comment;

    }

}

@Entity

@Data

@NoArgsConstructor

public class HardCoverBookEntity extends BookEntity {

    private Integer pages;

    @Builder

    public HardCoverBookEntity(Long id, String bookName, AuthorEntity author, String index, String comment, Integer pages) {

        super(id, bookName, author, index, comment);

        this.pages = pages;

    }

}

@Entity

@Data

@NoArgsConstructor

public class AudioBookEntity extends BookEntity{

    private Integer playLength;

    private String reader;

    @Builder()

    public AudioBookEntity(Long id, String bookName, AuthorEntity author, String index, String comment, Integer playLength, String reader) {

        super(id, bookName, author, index, comment);

        this.playLength = playLength;

        this.reader = reader;

    }

}

**А наши начальные данные так:**

BookEntity bookEntity1 = HardCoverBookEntity.builder()

        .bookName("Приключения Оливера Твиста")

        .author(authorEntity1)

        .pages(220)

        .comment("Отличный приключенчиский роман")

        .index("ISBN: 978-5-91921-226-3")

        .build();

BookEntity bookEntity2 = HardCoverBookEntity.builder()

        .bookName("Гордость и предубеждение")

        .author(authorEntity2)

        .pages(400)

        .comment("Занудная история про богатеев в Америке")

        .index("ISBN: 978-5-699-52151-7")

        .build();

BookEntity bookEntity3 = AudioBookEntity.builder()

        .bookName("Фауст")

        .author(authorEntity3)

        .playLength(873)

        .comment("Пища для ума")

        .index("ISBN: 5-699-07346-9")

        .reader("Илья Прудовский")

        .build();

**И вот так мы поменяем наш конвертор:**

@Component

public class BookConvertor {

    private final ModelMapper modelMapper;

    private final Map<Type, Type> typeMap = Map.of(

            HardCoverBookEntity.class, HardCoverBookDto.class,

            AudioBookEntity.class, AudioBookDto.class);

    private Converter<String, String> isbnRemover = (src) -> src.getSource().replaceAll("ISBN: ", "");

    private Converter<Integer, String> playTimeConverter = (src) -> src.getSource() + " минут";

    public BookConvertor() {

        this.modelMapper = new ModelMapper();

        TypeMap<BookEntity, BookDto> baseTypeMap = modelMapper.createTypeMap(BookEntity.class, BookDto.class);

        modelMapper.createTypeMap(AuthorEntity.class, AuthorDto.class);

        baseTypeMap

                .addMapping(BookEntity::getComment, BookDto::setReview)

                .addMappings(mapper -> mapper.using(isbnRemover).map(BookEntity::getIndex, BookDto::setIndex));

        baseTypeMap

                .include(AudioBookEntity.class, AudioBookDto.class)

                .include(HardCoverBookEntity.class, HardCoverBookDto.class);

        modelMapper.typeMap(AudioBookEntity.class, AudioBookDto.class)

                .addMappings(mapper -> mapper.using(playTimeConverter).map(AudioBookEntity::getPlayLength, AudioBookDto::setPlayTime))

                .addMapping(AudioBookEntity::getReader, AudioBookDto::setReader);

        modelMapper.typeMap(HardCoverBookEntity.class, HardCoverBookDto.class)

                .addMapping(HardCoverBookEntity::getPages, HardCoverBookDto::setPages);

    }

    public BookDto convertToDto(BookEntity entity) {

        return modelMapper.map(entity, typeMap.get(entity.getClass()));

    }

}

Для того, чтобы ModelМapper понял, что AudioBookEntity и HardCoverBookEntity — это наследники BookEntity, мы должны к TypeMap<BookEntity,BookDto> вызвать include и добавить маппинги. Но, к сожалению, для внутренних маппингов нам надо будет указывать вручную маппинг всех полей, как показано в примере. Эта особенность может стать проблемой если у вас на проекте примитивные базовые классы и развитая иерархия наследования классов.

**В ответе на запрос теперь мы получаем:**

[

{

"bookName": "Приключения Оливера Твиста",

"index": "978-5-91921-226-3",

"review": "Отличный приключенчиский роман",

"author": {

"name": "Чарльз Диккенс"

},

"pages": 220

},

{

"bookName": "Гордость и предубеждение",

"index": "978-5-699-52151-7",

"review": "Занудная история про богатеев в Америке",

"author": {

"name": "Джейн Остин"

},

"pages": 400

},

{

"bookName": "Фауст",

"index": "5-699-07346-9",

"review": "Пища для ума",

"author": {

"name": "Иоганн Вольфганг фон Гёте"

},

"playTime": "873 минут",

"reader": "Илья Прудовский"

}

]

**Заключение**

ModelМapper — это удобный фреймворк, который можно использовать как на старте вашего проекта, так и на более поздних этапах. Но у него, как и у любого инструмента, есть свои слабые стороны и ограничения, о которых стоит знать и которые стоит учитывать.