# Создание проекта User-Product-JPA-H2

Используйте

<https://start.spring.io/>

для создания нового проекта с зависимостями:

- Spring Web

- Spring Data JPA

- H2 Database

- Spring Boot Starter

- Thymeleaf

Запустите на исполнение, убедитесь, что проект работает.

## Ведите проект на github

Создайте репозиторий

Git – Create Repository

Загрузите проект в репозиторй

Git – Commit

Зайдите на GitHub, создайте репозиторий, заберите ссылку на удаленный репозиторий.



Добавьте в настройку

Git – Push – Define Remote – Ok - Push

pom.xml

Откройте файл pom.xml и убедитесь, что у вас есть следующие зависимости:

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-thymeleaf</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

## application.properties

В файл src/main/resources/**application.properties** добавьте следующие настройки:

spring.h2.console.enabled=true

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=password

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

## Создание сущностей

User.java

import jakarta.persistence.\*;

import java.util.Set;

@Entity

@Table(name = "users")

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@OneToMany(mappedBy = "user", cascade = CascadeType.ALL)

private Set<Product> products;

// Getters and Setters

public Long getId() { return id; }

public void setId(Long id) { this.id = id; }

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public Set<Product> getProducts() { return products; }

public void setProducts(Set<Product> products) { this.products = products; }

}

Product.java

import jakarta.persistence.\*;

@Entity

@Table(name = "products")

public class Product {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@ManyToOne

@JoinColumn(name = "user\_id")

private User user;

// Getters and Setters

public Long getId() { return id; }

public void setId(Long id) { this.id = id; }

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public User getUser () { return user; }

public void setUser (User user) { this.user = user; }

}

## Создание репозиториев

Создайте интерфейсы репозиториев.

UserRepository.java

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

public interface UserRepository extends JpaRepository<User, Long> {

}

ProductRepository.java

import com.example.demo.model.Product;

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

public interface ProductRepository extends JpaRepository<Product, Long> {

}

## Создание контроллеров

UserController.java

import com.example.demo.model.User;

import com.example.demo.repository.UserRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

@Controller

@RequestMapping("/users")

public class UserController {

http://localhost:8080/users

@Autowired

private UserRepository userRepository;

@GetMapping

public String getAllUsers(Model model) {

List<User> users = userRepository.findAll();

model.addAttribute("users", users);

return "users"; // Возвращаем имя шаблона

}

@PostMapping

public String createUser (@RequestParam String name) {

User user = new User();

user.setName(name);

userRepository.save(user);

return "redirect:/users"; // Перенаправление после создания

}

@PostMapping("/{id}/delete")

public String deleteUser (@PathVariable Long id) {

userRepository.deleteById(id);

return "redirect:/users"; // Перенаправление после удаления

}

}

ProductController.java

import com.example.demo.model.Product;

import com.example.demo.repository.ProductRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

<http://localhost:8080/products>

@Controller

@RequestMapping("/products")

public class ProductController {

@Autowired

private ProductRepository productRepository;

@GetMapping

public String getAllProducts(Model model) {

List<Product> products = productRepository.findAll();

model.addAttribute("products", products);

return "products"; // Возвращаем имя шаблона

}

@PostMapping

public String createProduct(@RequestParam String name) {

Product product = new Product();

product.setName(name);

productRepository.save(product);

return "redirect:/products"; // Перенаправление после создания

}

@PostMapping("/{id}/delete")

public String deleteProduct(@PathVariable Long id) {

productRepository.deleteById(id);

return "redirect:/products"; // Перенаправление после удаления

}

}

## Создание HTML-шаблонов

users.html

Создайте файл users.html в папке src\main\resources\templates

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

<title>Users</title>

</head>

<body>

<h1>Users</h1>

<form action="" th:action="@{/users}" method="post">

<input type="text" name="name" placeholder="Enter user name" required/>

<button type="submit">Add User</button>

</form>

<ul>

<li th:each="user : ${users}">

<span th:text="${user.name}"></span>

<form th:action="@{/users/{id}(id=${user.id})}" method="post" style="display:inline;">

<input type="hidden" name="\_method" value="delete"/>

<button type="submit">Delete</button>

</form>

</li>

</ul>

</body>

</html>

products.html

Создайте файл products.html в папке templates:

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

<title>Products</title>

</head>

<body>

<h1>Products</h1>

<form action="" th:action="@{/products}" method="post">

<input type="text" name="name" placeholder="Enter product name" required/>

<button type="submit">Add Product</button>

</form>

<ul>

<li th:each="product : ${products}">

<span th:text="${product.name}"></span>

<form th:action="@{/products/{id}(id=${product.id})}" method="post" style="display:inline;">

<input type="hidden" name="\_method" value="delete"/>

<button type="submit">Delete</button>

</form>

</li>

</ul>

</body>

</html>

## Тестирование приложения

Используйте H2 консоль по адресу для просмотра и управления данными.

[http://localhost:8080/h2-console](http://localhost:8080/h2-console%60)

Перейдите для управления пользователями.

<http://localhost:8080/users>

Перейдите для управления продуктами.

<http://localhost:8080/products>

Перейдите, чтобы увидеть информацию о пользователе с ID 1 и связанных с ним продуктах

[http://localhost:8080/users/1/products](http://localhost:8080/users/1/products%60),

# Дополнение - инициализация данных

Чтобы создать одну запись пользователя с тремя связанными записями продукта в базе данных H2 в вашем приложении Spring Boot, вы можете использовать CommandLineRunner. Этот интерфейс позволяет вам выполнить код при запуске приложения, что идеально подходит для инициализации базы данных.

Создайте новый класс, который реализует CommandLineRunner, и добавьте в него код для создания пользователя и трех продуктов, принадлежащих этому пользователю.

import com.example.demo.model.Product;

import com.example.demo.model.User;

import com.example.demo.repository.ProductRepository;

import com.example.demo.repository.UserRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.CommandLineRunner;

import org.springframework.stereotype.Component;

import java.util.HashSet;

@Component

public class DataInitializer implements CommandLineRunner {

@Autowired

private UserRepository userRepository;

@Autowired

private ProductRepository productRepository;

@Override

public void run(String... args) throws Exception {

// Создание пользователя

User user = new User();

user.setName("John Doe");

// Создание продуктов

Product product1 = new Product();

product1.setName("Product 1");

product1.setUser (user);

Product product2 = new Product();

product2.setName("Product 2");

product2.setUser (user);

Product product3 = new Product();

product3.setName("Product 3");

product3.setUser (user);

// Установка продуктов для пользователя

user.setProducts(new HashSet<>());

user.getProducts().add(product1);

user.getProducts().add(product2);

user.getProducts().add(product3);

// Сохранение пользователя и продуктов в базе данных

userRepository.save(user); // Это также сохранит продукты благодаря каскадированию

}

}

## Обновление сущностей (если необходимо)

Убедитесь, что в ваших сущностях User и Product правильно настроены отношения. Например:

User.java

@Entity

@Table(name = "users")

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@OneToMany(mappedBy = "user", cascade = CascadeType.ALL)

private Set<Product> products = new HashSet<>();

// Getters и Setters

}

Product.java

@Entity

@Table(name = "products")

public class Product {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@ManyToOne

@JoinColumn(name = "user\_id")

private User user;

// Getters и Setters

}

## Проверка данных в H2

Вы можете проверить данные в H2 консоли, перейдя по адресу

http://localhost:8080/h2-console.

Используйте следующие параметры для подключения:

- JDBC URL: jdbc:h2:mem:testdb

- User Name: sa

- Password: password

После подключения выполните следующий SQL-запрос, чтобы увидеть данные:

SELECT \* FROM users;

SELECT \* FROM products;

Заключение. Теперь у вас есть один пользователь с тремя продуктами, связанными с ним, в базе данных H2. Если у вас есть дополнительные вопросы или требуется помощь с другими аспектами приложения, дайте знать!

# Заготовка для общей таблицы User-Product

## userProducts.html

Создайте файл userProducts.html в каталоге src/main/resources/templates.

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

<title>User Products</title>

</head>

<body>

<h1>User Products</h1>

<div th:if="${userProducts != null}">

<h2>User: <span th:text="${userProducts.userName}"></span></h2>

<h3>Products:</h3>

<ul>

<li th:each="product : ${userProducts.products}">

<span th:text="${product.name}"></span>

</li>

</ul>

</div>

<div th:if="${userProducts == null}">

<p>User not found.</p>

</div>

</body>

</html>

## Optional

@GetMapping("/{id}/products")

public String getUser (@PathVariable Long id, Model model) {

Optional<User> userOptional = userRepository.findById(id);

if (userOptional.isPresent()) {

User user = userOptional.get();

User userProducts = new User(user.getId(), user.getName(), user.getProducts());

model.addAttribute("userProducts", userProducts);

return "userProducts.html"; // имя HTML-шаблона

}

return "error"; // или другая страница, если пользователь не найден

}

Optional

— это класс в Java, который был введен в версии Java 8 и находится в пакете java.util. Он предназначен для представления значения, которое может быть либо присутствующим, либо отсутствующим. Это полезный способ избежать NullPointerException, который часто возникает при работе с объектами, которые могут быть null.

Optional позволяет явно указать, что значение может отсутствовать. Вместо того чтобы возвращать null, метод может вернуть Optional.empty().

Методы класса Optional:

- isPresent(): возвращает true, если значение присутствует, и false, если отсутствует.

- get(): возвращает значение, если оно присутствует. Если значение отсутствует, выбрасывает NoSuchElementException.

- orElse(T other): возвращает значение, если оно присутствует; в противном случае возвращает other.

- orElseGet(Supplier<? extends T> other): возвращает значение, если оно присутствует; в противном случае вызывает переданный Supplier и возвращает его результат.

- orElseThrow(Supplier<? extends X> exceptionSupplier): возвращает значение, если оно присутствует; в противном случае выбрасывает исключение, создаваемое Supplier.

Пример использования Optional

import java.util.Optional;

public class OptionalExample {

public static void main(String[] args) {

// Пример с Optional

Optional<String> optionalString = Optional.of("Hello, World!");

// Проверка наличия значения

if (optionalString.isPresent()) {

System.out.println(optionalString.get()); // Вывод: Hello, World!

}

// Пример без значения

Optional<String> emptyOptional = Optional.empty();

// Использование orElse

String value = emptyOptional.orElse("Default Value");

System.out.println(value); // Вывод: Default Value

// Использование orElseGet

String valueWithSupplier = emptyOptional.orElseGet(() -> "Generated Value");

System.out.println(valueWithSupplier); // Вывод: Generated Value

// Использование orElseThrow

try {

String valueWithException = emptyOptional.orElseThrow(() -> new RuntimeException("Value not present"));

} catch (RuntimeException e) {

System.out.println(e.getMessage()); // Вывод: Value not present

}

}

}

Преимущества использования Optional

1. Избегание NullPointerException: Использование Optional позволяет избежать распространенной проблемы с null, делая код более безопасным и понятным.

2. Явное указание на возможность отсутствия значения: Optional делает намерения разработчика более ясными, когда он показывает, что метод может не вернуть значение.

3. Улучшение читаемости кода: Методы Optional позволяют использовать функциональный стиль программирования, что может улучшить читаемость и выразительность кода.

Заключение. Optional —мощный инструмент для управления значениями, которые могут отсутствовать, и его использование может значительно улучшить качество и безопасность вашего кода. Однако следует помнить, что Optional не предназначен для использования в качестве замены всех случаев null; он должен использоваться в тех случаях, когда отсутствие значения является нормальным и ожидаемым поведением.

# Обсуждение отношений в БД и JPA

Сущность ([Entity](https://www.ibm.com/support/knowledgecenter/en/SSWSR9_11.6.0/com.ibm.mdmhs.overview.doc/entityconcepts.html)) — это некий объект из реальной жизни (например, машина), который имеет атрибуты (двери, [КОЛЁСА](https://www.google.com.ua/search?q=%D0%BA%D0%BE%D0%BB%D1%91%D1%81%D0%B0&source=lnms&tbm=isch&sa=X&ved=0ahUKEwilw9anwfHdAhUHXSwKHfZTBqEQ_AUIDigB&biw=1920&bih=948#imgrc=7ZYC5HIpJy27dM:), двигатель). DB Entity: в этом случае наша сущность хранится в DB, все просто. Зачем и каким образом мы засунули машину в базу данных — рассмотрим позже.

Взаимосвязи данных таблиц - 4

1. [One-to-One](https://en.wikipedia.org/wiki/One-to-one_(data_model))
2. [One-to-Many](https://en.wikipedia.org/wiki/One-to-many_(data_model))
3. [Many-to-One](https://en.wikipedia.org/wiki/One-to-many_(data_model))
4. [Many-to-Many](https://en.wikipedia.org/wiki/Many-to-many_(data_model))

## One-to-One Relationship

Используем две сущности: Author и Book. У книги может быть автор, может быть несколько авторов, а может и не быть. На этом примере создадим все виды связей. Скрипт, который [создает DB Tables](https://github.com/qThegamEp/ForJavaRushPublication2/blob/OneToOne/src/main/resources/initDB.sql):

DROP TABLE IF EXISTS PUBLIC.BOOK;

CREATE TABLE PUBLIC.BOOK (

ID INTEGER NOT NULL AUTO\_INCREMENT,

NAME VARCHAR(255) NOT NULL,

PRINT\_YEAR INTEGER(4) NOT NULL,

CONSTRAINT BOOK\_PRIMARY\_KEY PRIMARY KEY (ID)

);

DROP TABLE IF EXISTS PUBLIC.AUTHOR;

CREATE TABLE PUBLIC.AUTHOR (

ID INTEGER NOT NULL AUTO\_INCREMENT,

FIRST\_NAME VARCHAR(255) NOT NULL,

SECOND\_NAME VARCHAR(255) NOT NULL,

BOOK\_ID INTEGER NOT NULL UNIQUE,

CONSTRAINT AUTHOR\_PRIMARY\_KEY PRIMARY KEY (ID),

CONSTRAINT BOOK\_FOREIGN\_KEY FOREIGN KEY (BOOK\_ID) REFERENCES BOOK (ID)

);

Напишем [скрипт который заполняет](https://github.com/qThegamEp/ForJavaRushPublication2/blob/OneToOne/src/main/resources/populateDB.sql) нашу DB тестовыми данными:

INSERT INTO PUBLIC.BOOK (NAME, PRINT\_YEAR)

VALUES ('First book', 2010),

('Second book', 2011),

('Third book', 2012);

INSERT INTO PUBLIC.AUTHOR (FIRST\_NAME, SECOND\_NAME, BOOK\_ID)

VALUES ('Pablo', 'Lambado', 1),

('Pazo', 'Zopa', 2),

('Lika', 'Vika', 3);

One-to-One relationship нужно тогда, когда сущность одной таблицы связанная с одной сущностью другой ( или вообще не связанная если NOT NULL убрать у BOOK\_ID). В нашем примере у одной книжки ДОЛЖЕН быть один автор. Как связать Java класс с DB сущностями? Создадим два класса Book и Author.

[Author](https://github.com/qThegamEp/ForJavaRushPublication2/blob/OneToOne/src/main/java/com/qthegamep/forjavarushpublication2/entity/Author.java) класс

@Data

@Entity

@DynamicInsert

@DynamicUpdate

@Table(name = "AUTHOR")

public class Author {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "ID", nullable = false)

private Long id;

@Column(name = "FIRST\_NAME", nullable = false)

private String firstName;

@Column(name = "SECOND\_NAME", nullable = false)

private String secondName;

@OneToOne

@JoinColumn(name = "BOOK\_ID", unique = true, nullable = false)

private Book book;

}

Все поля в классе повторяют атрибуты DB сущности.

[@Data](https://projectlombok.org/features/Data) (из [Lombok'a](https://projectlombok.org/)) говорит, что для каждого поля будет создан геттер и сеттер, будет переопределен equals, hashcode, и сгенерирован toString метод.

[@Entity](https://docs.oracle.com/javaee/7/api/javax/persistence/Entity.html) говорит, что данный класс — сущность и связан с сущностью DB.

[@DynamicInsert](https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/DynamicInsert.html) и [@DynamicUpdate](https://docs.jboss.org/hibernate/orm/5.3/javadocs/org/hibernate/annotations/DynamicUpdate.html) говорят, что будут выполнятся динамические вставки и обновления в DB. Это более глубокие настройки Hibernate, которые пригодятся вам, что бы у вас был ПРАВИЛЬНЫЙ батчинг.

[@Table](https://docs.oracle.com/javaee/7/api/javax/persistence/Table.html)(name = "AUTHOR") связывает класс Book с таблицей DB AUTHOR.

[@Id](https://docs.oracle.com/javaee/6/api/javax/persistence/Id.html) говорит, что данное поле — primary key.

[@GeneratedValue](https://docs.oracle.com/javaee/6/api/javax/persistence/GeneratedValue.html)(strategy = GenerationType.IDENTITY) — стратегия генерации primary key.

[@Column](https://docs.oracle.com/javaee/5/api/javax/persistence/Column.html)(name = "ID", nullable = false) связывает поле с атрибутом DB, и также говорит, что данное поле DB не может быть null. Это также полезно при генерации таблиц из сущностей. Обратный процесс тому, как мы сейчас создаем наш проект, это нужно в тестовых DB для Unit тестов.

[@OneToOne](https://docs.oracle.com/javaee/6/api/javax/persistence/OneToOne.html) говорит, что данное поле является полем отношения One-to-One.

[@JoinColumn](https://docs.oracle.com/javaee/6/api/javax/persistence/JoinColumn.html)(name = "BOOK\_ID", unique = true, nullable = false) — будет создана колонка BOOK\_ID, которая является уникальной и not null.

В классе [Book](https://github.com/qThegamEp/ForJavaRushPublication2/blob/OneToOne/src/main/java/com/qthegamep/forjavarushpublication2/entity/Book.java) также нужно сделать связь One-to-One и указать поле, по которому происходит mapping. @OneToOne(mappedBy = "book") — в данном примере это поле book класса Author. JPA сам их свяжет. С первого взгляда может показаться что тут каша из аннотаций, но на самом деле это очень удобно и с опытом вы удете их ставить, даже не задумываясь.

Author

import lombok.Data;

import org.hibernate.annotations.DynamicInsert;

import org.hibernate.annotations.DynamicUpdate;

import javax.persistence.\*;

@Data

@Entity

@DynamicInsert

@DynamicUpdate

@Table(name = "BOOK")

public class Book {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "ID", nullable = false)

private Long id;

@Column(name = "NAME", nullable = false)

private String name;

@Column(name = "PRINT\_YEAR", nullable = false)

private int printYear;

@OneToOne(mappedBy = "book")

private Author author;

}

## One-to-Many Relationship

[инициализирующий скрипт](https://github.com/qThegamEp/ForJavaRushPublication2/blob/OneToMany/src/main/resources/initDB.sql).

DROP TABLE IF EXISTS PUBLIC.AUTHOR;

CREATE TABLE PUBLIC.AUTHOR (

ID INTEGER NOT NULL AUTO\_INCREMENT,

FIRST\_NAME VARCHAR(255) NOT NULL,

SECOND\_NAME VARCHAR(255) NOT NULL,

CONSTRAINT AUTHOR\_PRIMARY\_KEY PRIMARY KEY (ID),

);

DROP TABLE IF EXISTS PUBLIC.BOOK;

CREATE TABLE PUBLIC.BOOK (

ID INTEGER NOT NULL AUTO\_INCREMENT,

NAME VARCHAR(255) NOT NULL,

PRINT\_YEAR INTEGER(4) NOT NULL,

AUTHOR\_ID INTEGER NOT NULL,

CONSTRAINT BOOK\_PRIMARY\_KEY PRIMARY KEY (ID),

CONSTRAINT AUTHOR\_FOREIGN\_KEY FOREIGN KEY (AUTHOR\_ID) REFERENCES AUTHOR (ID)

);

One-to-Many Relationship —у одного автора может быть несколько книг. Левой сущности соответствует одна или несколько правой. В [Author](https://github.com/qThegamEp/ForJavaRushPublication2/blob/OneToMany/src/main/java/com/qthegamep/forjavarushpublication2/entity/Author.java) класса появляется поле:

@OneToMany(fetch = FetchType.LAZY, mappedBy = "author")

private Set<Book> books;

Оно уже является сетом, так как у нас может быть несколько книг. [@OneToMany](https://docs.oracle.com/javaee/6/api/javax/persistence/OneToMany.html) говорит о типе отношения. FetchType.Lazy говорит, что не нужно нам подгружать весь список книг если это не указанно в запросе. Также следует сказать, что данное поле НЕЛЬЗЯ добавлять в toString, иначе StackOverflowError. Об этом заботится Lombok:

@ToString(exclude = "books")

Код выглядит так

import lombok.Data;

import lombok.ToString;

import org.hibernate.annotations.DynamicInsert;

import org.hibernate.annotations.DynamicUpdate;

import javax.persistence.\*;

import java.util.Set;

/\*\*

\* This class is an entity.

\* There is an no args constructor, getters and setters for all fields, override equals, hashcode and toString methods

\* generated by lombok.

\*/

@Data

@ToString(exclude = "books")

@Entity

@DynamicInsert

@DynamicUpdate

@Table(name = "AUTHOR")

public class Author {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "ID", nullable = false)

private Long id;

@Column(name = "FIRST\_NAME", nullable = false)

private String firstName;

@Column(name = "SECOND\_NAME", nullable = false)

private String secondName;

@OneToMany(fetch = FetchType.LAZY, mappedBy = "author")

private Set<Book> books;

}

В классе [Book](https://github.com/qThegamEp/ForJavaRushPublication2/blob/OneToMany/src/main/java/com/qthegamep/forjavarushpublication2/entity/Book.java) мы делаем обратную связь (Many-to-One):

@ManyToOne(fetch = FetchType.LAZY, cascade = CascadeType.ALL)

@JoinColumn(name = "AUTHOR\_ID", nullable = false)

private Author author;

One-to-Many является зеркальным отображением Many-to-One и наоборот. Следует подчеркнуть, что Hibernate нечего не знает о двунаправленных связях. Для него это две разные связи: одна в одну сторону, другая — в противоположную.

import lombok.Data;

import lombok.ToString;

import org.hibernate.annotations.DynamicInsert;

import org.hibernate.annotations.DynamicUpdate;

import javax.persistence.\*;

import java.util.Set;

@Data

@ToString(exclude = "books")

@Entity

@DynamicInsert

@DynamicUpdate

@Table(name = "AUTHOR")

public class Author {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "ID", nullable = false)

private Long id;

@Column(name = "FIRST\_NAME", nullable = false)

private String firstName;

@Column(name = "SECOND\_NAME", nullable = false)

private String secondName;

@OneToMany(fetch = FetchType.LAZY, mappedBy = "author")

private Set<Book> books;

}

## Many-to-One Relationship

Many-to-One является зеркальным отображением One-to-Many. [Инициализирующий Скрипт](https://github.com/qThegamEp/ForJavaRushPublication2/blob/ManyToOne/src/main/resources/initDB.sql).

DROP TABLE IF EXISTS PUBLIC.BOOK;

CREATE TABLE PUBLIC.BOOK (

ID INTEGER NOT NULL AUTO\_INCREMENT,

NAME VARCHAR(255) NOT NULL,

PRINT\_YEAR INTEGER(4) NOT NULL,

CONSTRAINT BOOK\_PRIMARY\_KEY PRIMARY KEY (ID)

);

DROP TABLE IF EXISTS PUBLIC.AUTHOR;

CREATE TABLE PUBLIC.AUTHOR (

ID INTEGER NOT NULL AUTO\_INCREMENT,

FIRST\_NAME VARCHAR(255) NOT NULL,

SECOND\_NAME VARCHAR(255) NOT NULL,

BOOK\_ID INTEGER NOT NULL,

CONSTRAINT AUTHOR\_PRIMARY\_KEY PRIMARY KEY (ID),

CONSTRAINT BOOK\_FOREIGN\_KEY FOREIGN KEY (BOOK\_ID) REFERENCES BOOK (ID)

);

В классе [Author](https://github.com/qThegamEp/ForJavaRushPublication2/blob/ManyToOne/src/main/java/com/qthegamep/forjavarushpublication2/entity/Author.java) больше нет сета,

import lombok.Data;

import org.hibernate.annotations.DynamicInsert;

import org.hibernate.annotations.DynamicUpdate;

import javax.persistence.\*;

@Data

@Entity

@DynamicInsert

@DynamicUpdate

@Table(name = "AUTHOR")

public class Author {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "ID", nullable = false)

private Long id;

@Column(name = "FIRST\_NAME", nullable = false)

private String firstName;

@Column(name = "SECOND\_NAME", nullable = false)

private String secondName;

@ManyToOne(fetch = FetchType.LAZY, cascade = CascadeType.ALL)

@JoinColumn(name = "BOOK\_ID", nullable = false)

private Author book;

}

так как он переместился в [Book](https://github.com/qThegamEp/ForJavaRushPublication2/blob/ManyToOne/src/main/java/com/qthegamep/forjavarushpublication2/entity/Book.java) класс.

import lombok.Data;

import lombok.ToString;

import org.hibernate.annotations.DynamicInsert;

import org.hibernate.annotations.DynamicUpdate;

import javax.persistence.\*;

import java.util.Set;

@Data

@ToString(exclude = "author")

@Entity

@DynamicInsert

@DynamicUpdate

@Table(name = "BOOK")

public class Book {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "ID", nullable = false)

private Long id;

@Column(name = "NAME", nullable = false)

private String name;

@Column(name = "PRINT\_YEAR", nullable = false)

private int printYear;

@OneToMany(fetch = FetchType.LAZY, mappedBy = "book")

private Set<Author> author;

}

## Many-to-Many Relationship

Отношение создается через дополнительную таблицу. Но данная таблица не является сущностью. У книги может быть много автором, и у автора может быть много книг. Они могут пересекаться.Таблица HAS — это не сущность.

Класс [Author](https://github.com/qThegamEp/ForJavaRushPublication2/blob/ManyToMany/src/main/java/com/qthegamep/forjavarushpublication2/entity/Author.java):

import lombok.Data;

import lombok.ToString;

import org.hibernate.annotations.DynamicInsert;

import org.hibernate.annotations.DynamicUpdate;

import javax.persistence.\*;

import java.util.Set;

@Data

@ToString(exclude = "books")

@Entity

@DynamicInsert

@DynamicUpdate

@Table(name = "AUTHOR")

public class Author {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "ID", nullable = false)

private Long id;

@Column(name = "FIRST\_NAME", nullable = false)

private String firstName;

@Column(name = "SECOND\_NAME", nullable = false)

private String secondName;

@ManyToMany

@JoinTable(name = "HAS",

joinColumns = @JoinColumn(name = "AUTHOR\_ID", referencedColumnName = "ID"),

inverseJoinColumns = @JoinColumn(name = "BOOK\_ID", referencedColumnName = "ID")

)

private Set<Book> books;

}

[@ManyToMany](https://docs.oracle.com/javaee/6/api/javax/persistence/ManyToMany.html) — вид отношения.

[@JoinTable](https://docs.oracle.com/javaee/7/api/javax/persistence/JoinTable.html) — связывает атрибут с дополнительной таблицей HAS. В ней мы указываем два атрибута, которые будут указывать на primary keys двух сущностей.

Класс [Book](https://github.com/qThegamEp/ForJavaRushPublication2/blob/ManyToMany/src/main/java/com/qthegamep/forjavarushpublication2/entity/Book.java):

import lombok.Data;

import lombok.ToString;

import org.hibernate.annotations.DynamicInsert;

import org.hibernate.annotations.DynamicUpdate;

import javax.persistence.\*;

import java.util.Set;

@Data

@ToString(exclude = "authors")

@Entity

@DynamicInsert

@DynamicUpdate

@Table(name = "BOOK")

public class Book {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "ID", nullable = false)

private Long id;

@Column(name = "NAME", nullable = false)

private String name;

@Column(name = "PRINT\_YEAR", nullable = false)

private int printYear;

@ManyToMany(fetch = FetchType.LAZY, mappedBy = "books")

private Set<Author> authors;

}

# Docker

Создайте файлы конфигурации и запустите приложение в Docker.