MAPSTRUCT IN JAVA
SIMPLIFYING DTO MAPPING

Clean, Fast and Type-Safe Mappings for Spring Boot Projects

Created by Emir Totić – Java Backend Engineer

When Manual Mapping Becomes a Mess

When mapping DTOs and Entities manually, we end up writing repetitive, error-prone code.
Every time a field changes, we must update it in multiple places.

Code example:

```
UserDTO dto = new UserDTO();
dto.setName(user.getName());
dto.setEmail(user.getEmail());
dto.setAge(user.getAge());
```

Potential issues with this code:

- Too much boilerplate
- Hard to maintain when models evolve
- Risk of missing or mismatched fields

The Solution: MapStruct

MapStruct automatically generates mapping code at compile time, eliminating repetitive and error-prone manual work. It avoids reflection, runs lightning-fast, ensures full type safety, and keeps your code clean - no boilerplate required.

```
@Mapper
public interface UserMapper {
    UserMapper INSTANCE = Mappers.getMapper(UserMapper.class);
    UserDTO toDto(User user);
}
```

Result: MapStruct generates the implementation behind the scenes and you get a ready-to-use mapper without writing a single line of boilerplate.

Under the Hood

Why MapStruct Is Faster?

- MapStruct generates plain Java code, no reflection at runtime.
- Mapping happens just like you would write it manually, but generated automatically.
- Compared to ModelMapper, there’s zero runtime cost and full compile-time validation.

What MapStruct Generates:

```java
@Generated
public class UserMapperImpl implements UserMapper {

    @Override
    public UserDTO toDto(User user) {
        if (user == null) return null;

        UserDTO dto = new UserDTO();
        dto.setName(user.getName());
        dto.setEmail(user.getEmail());
        dto.setAge(user.getAge());
        return dto;
    }
}
```

Result: performance like manual mapping, but maintainability like magic.

Custom Field Mapping

When our field names don’t match, MapStruct will let us define custom mappings easily using the @Mapping annotation.

```
@Mappings({
    @Mapping(source = "emailAddress", target = "email"),
    @Mapping(source = "birthDate", target = "dateOfBirth")
})
UserDTO toDto(User user);
```

The source field comes from the entity, and target is the DTO field.

This is very clean, type-safe, and easy to maintain - even as our models evolve or grow in complexity. MapStruct ensures every field is mapped explicitly, reducing human error and keeping our codebase consistent over time.

Bidirectional Mapping

MapStruct supports bidirectional object mapping and conversions, making it easy to handle complex models.

```java
@Mapper
public interface OrderMapper {

    @Mapping(source = "customer.name", target = "customerName")
    OrderDTO toDto(Order order);

    @InheritInverseConfiguration
    Order toEntity(OrderDTO dto);
}
```

@InheritInverseConfiguration automatically reuses the same mapping rules in the opposite direction.

One mapper - both directions, zero duplication. It is perfect for mapping entities with nested relationships (like Order → Customer → Address) and DTOs that are used in both read and write operations.

Nested Mapping

MapStruct makes it easy to map nested objects and you can directly access inner fields without manual null checks or helper methods.

```java
@Mapper
public interface OrderMapper {

    @Mapping(source = "customer.address.city", target = "customerCity")
    @Mapping(source = "customer.address.zipCode", target = "postalCode")
    OrderDTO toDto(Order order);
}
```

It automatically navigates through nested objects and maps only the needed fields.

This approach is ideal for complex domain models, MapStruct eliminates deep boilerplate mapping code and significantly improves readability and maintainability.

Integration with Spring

MapStruct easily integrates with the Spring ecosystem, allowing you to inject mappers like any other Spring bean.

```java
@Mapper(componentModel = "spring")
public interface UserMapper {
    UserDTO toDto(User user);
}
```

```java
@Service
public class UserService {
    private final UserMapper userMapper;

    public UserService(UserMapper userMapper) {
        this.userMapper = userMapper;
    }
}
```

By setting *componentModel = "spring"*, MapStruct automatically registers the mapper as a Spring bean.

No manual initialization required. MapStruct works seamlessly with Spring's dependency injection, keeping your architecture clean, modular, and easy to test.

Custom Mapping

MapStruct allows defining custom logic when default field mapping isn’t enough.
You can write custom expressions or delegate the logic to another method.

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

    @Mapping(target = "fullName",
             expression = "java(user.getFirstName() + ' ' + user.getLastName())")
    UserDTO toDto(User user);
}
```

Custom expressions give full control when mapping requires transformations or computed fields.

It is flexible for complex conversions while keeping all transformation logic centralized inside the mapper, preventing unnecessary clutter in service layers.

Mapping Collections

MapStruct automatically maps collections between objects - no loops or manual conversion needed. It handles List, Set, and even Map types out of the box.

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

    ProductDTO toDto(Product product);
    List<ProductDTO> toDtoList(List<Product> products);
}
```

Each element is mapped using the same mapping rules as for single objects.

It automatically maps each element, supports nested mappers for complex types, and greatly simplifies batch data transformations.

Updating Existing Entities with @MappingTarget

MapStruct allows updating existing entities without creating new objects - ideal for partial updates or PATCH operations.

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

    void updateUserFromDto(UserDTO dto, @MappingTarget User entity);
}
```

Instead of returning a new User, this method updates the existing instance directly - keeping unchanged fields intact.

It is perfect for PATCH endpoints - it prevents data loss in persistent entities and keeps the service logic clean and focused.

Advantages

FEATURE	DESCRIPTION
Type-safe mapping	All conversions are validated at compile time
Zero reflection	Ensures excellent runtime performance
Easy maintenance	Eliminates repetitive boilerplate code
Seamless integration	Works perfectly with Lombok and Spring Boot
Clean & professional	Produces readable, production-ready code

Conclusion

MapStruct bridges the gap between clarity and performance, giving developers the speed of automation with the control of handwritten code.

This document is a quick, basic overview of MapStruct — the library offers many more advanced features and customization options beyond what's shown here.

From my perspective, it's an incredibly powerful and reliable tool that I use in my everyday development work to keep codebases clean, type-safe, and maintainable.

Thank you for reading


www.linkedin.com/in/emirtotic


https://emirtotic.github.io/portfolio-site


emirtotic@gmail.com