What is a DTO?

DTO, which stands for Data Transfer Object, is a design pattern conceived to reduce the number of calls when working with remote interfaces. As Martin Fowler defines in his blog, the main reason for using a Data Transfer Object is to batch up what would be multiple remote calls into a single one.

For example, lets say that we were communicating with a RESTful API that exposes our banking account data. In this situation, instead of issuing multiple requests to check the current status and latest transactions of our account, the bank could expose an endpoint that returned a DTO summarizing everything. As one of the most expensive operations in remote applications is the round-trip time between the client and the server, this coarse-grained interface can help improving performance by a great deal.

DTOs and Spring Boot APIs

Another advantage of using DTOs on RESTful APIs written in Java (and on Spring Boot), is that they can help hiding implementation details of domain objects (aka. entities). Exposing entities through endpoints can become a security issue if we do not carefully handle what properties can be changed through what operations.

As an example, let's imagine a Java API that exposes user details and accepts user updates through two endpoints. The first endpoint would handle GET requests and return user data, and the second endpoint would accept PUT requests to update these details. If this application didn't take advantage of DTOs, all the properties of the user would be exposed in the first endpoint (e.g. password) and the second endpoint would have to be very selective on what properties would accept when updating a user (e.g. not everybody can update the roles of a user). To overcome this situation, DTOs can come in handy by exposing only what the first endpoint is intended to expose, and by helping the second endpoint to restrict what it accepts. This characteristic helps us to keep the integrity of the data in our applications.

"DTOs can help us to keep the integrity of data on Java applications."

ModelMapper Introduction

To avoid having to write cumbersome/boilerplate code to map DTOs into entities and vice-versa, we are going to use a library called ModelMapper. The goal of ModelMapper is to make object mapping easy by automatically determining how one object model maps to another. This library is quite powerful and accepts a whole bunch of configurations to streamline the mapping process, but it also favors convention over configuration by providing a default behavior that fits most cases.

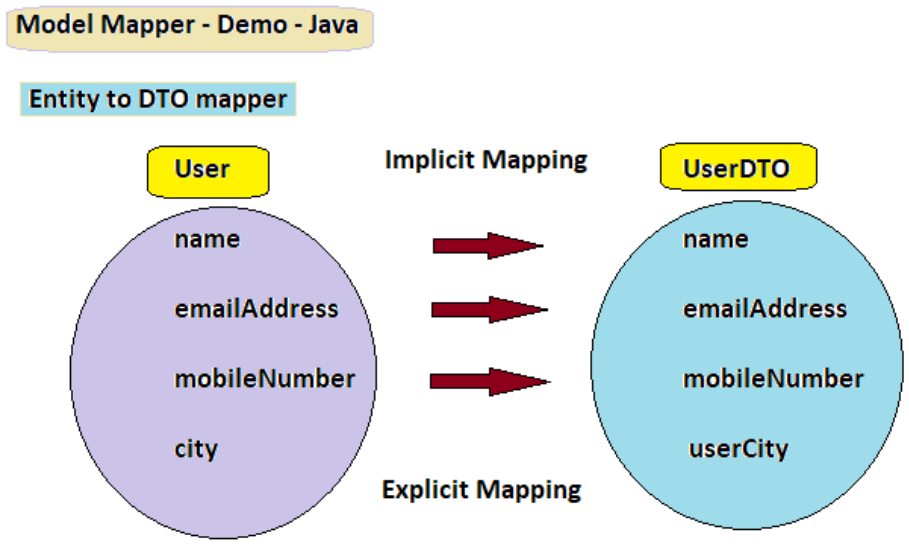
The user manual of this library is well written and can be a valuable resource if time comes where we need to tweak the mapping process.

Modelmapper Tutorial: How to add the ModelMapper to your project ?

If you’re a Maven user just add the modelmapper library as a dependency:

```
<dependency>
   <groupId>org.modelmapper</groupId>
   <artifactId>modelmapper</artifactId>
   <version>1.1.0</version>
</dependency>
```

Let’s try mapping some objects. Consider the following source and destination object models:



User to UserDTO

We can use ModelMapper to implicitly map an user instance to a new UserDTO:

```
ModelMapper modelMapper = new ModelMapper();
UserDTO userDTO = modelMapper.map(user, UserDTO.class);
```

How It Works ?

When the map method is called, the source and destination types are analyzed to determine which properties implicitly match according to a matching strategy and other configuration.

ModelMapper will do its best to determine reasonable matches between properties.

If required we can also do the explicit mapping between properties.(inform the mapper about the properties explicitly)

```
modelMapper.addMappings(new PropertyMap<User, UserDTO>() {
                                protected void configure() {
                                        map().setUserCity(source.getCity());
                                }
                        });
```