Amy's Programming Corner

Exploring Java one problem at a time

☰ **Menu**

Deep Mapping with ModelMapper

Amy DeGregorio Java, ModelMapper, Spring Boot August 2, 2018December 16, 2018 3 Minutes

Using ModelMapper for Deep Mapping in a Spring Boot Application

In previous posts, I've shown how to write custom converters (https://amydegregorio.com/2018/01/17/using-custom-modelmapper-converters-and-mappings/) and how to skip fields (https://amydegregorio.com/2018/05/23/skipping-fields-with-modelmapper/) while mapping using ModelMapper (http://modelmapper.org/). For those who aren't familiar with it, ModelMapper is a library for handling mapping between two similar classes. A common use case is mapping between data transfer objects (dtos) and entity objects in a web application. In this post, I'm going to demonstrate the deep mapping feature of ModelMapper in a Spring Boot (https://spring.io/projects/spring-boot) application. If you want to follow along with the example, use the Spring Initializr (https://start.spring.io/) to create an empty application. My example uses Gradle. Choose Web, JPA, H2 and Thymeleaf dependencies. Or you can just get the working code from github (https://github.com/amdegregorio/ModelMapper-DeepMappingExample).

Prerequisites:

- Java 8 or Greater
- Favorite IDE
- Basic understanding of Spring

Gradle Dependency:

```
ementation('com.github.jmnarloch:modelmapper-spring-boot-starter:1.1.0')
```

Project Setup

We're going to be using two entity classes Item and Location. The Location class represents a location in a warehouse.

```
@Entity
public class Location {
   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   private Long id;

   private String warehouseName;
   private Integer rowNumber;
   private String binLabel;

   /*Getters and setters below*/
}
```

The Item class represents an imaginary inventory item (Disclaimer: this is not a super realistic example…) that has a Location. *Note: For the sake of making the example work easily, the Location relationship is set to Cascade.ALL. That's not something I'd normally do lightly (if at all).*

```
@Entity
public class Item {
   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   private Long id;

   private String name;
   private String description;
   private Integer quantity;
   @OneToOne(fetch = FetchType.LAZY, cascade = CascadeType.ALL)
   @JoinColumn(name="location_id", nullable=true)
   private Location location;

   /* Getters and setters below */
}
```

We also need our Repository interfaces. For this example, they extend JpaRepository and have no custom methods. Additionally, we need an ItemDto for transferring between our user input form to our entity. For user input, we've flattened out the Location into the Item.

```
public class ItemDto {
   private Long id;
   private String name;
   private String description;
   private Integer quantity;
   private Long warehouseId;
   private String warehouseName;
   private Integer warehouseRowNumber;
   private String warehouseBinLabel;

   /* Getters and Setters below */
}
```

The code that's important for this example is found in the ItemController class.

ItemController class

In the ItemController, we define two custom property maps that do the deep mapping. We need one for each direction. This mapping, sets the fields on the Location entity in the Item entity from the specified Warehouse fields on the entry form for the Item. Using the map() method overrides the default mapping that ModelMapper tries to figure out automatically only for the fields specified.

```
PropertyMap<ItemDto, Item> warehouseMapping = new PropertyMap<ItemDto, I
   protected void configure() {
      map().getLocation().setId(source.getWarehouseId());
      map().getLocation().setWarehouseName(source.getWarehouseName());
      map().getLocation().setRowNumber(source.getWarehouseRowNumber());
      map().getLocation().setBinLabel(source.getWarehouseBinLabel());
   }
};
```

We do the same thing in the other direction to map the fields from the Location object in the Item entity into the flattened form fields, so we can see the warehouse fields in the list.

```
PropertyMap<Item, ItemDto> warehouseFieldMapping = new PropertyMap<Item,
   protected void configure() {
      map().setWarehouseId(source.getLocation().getId());
      map().setWarehouseName(source.getLocation().getWarehouseName());
      map().setWarehouseRowNumber(source.getLocation().getRowNumber());
      map().setWarehouseBinLabel(source.getLocation().getBinLabel());
   }
};
```

Each custom mapping can only be applied once, so we inject the ModelMapper instance into the constructor and add the mappings there.

```
@Autowired
public ItemController(ModelMapper modelMapper) {
   this.modelMapper = modelMapper;
   this.modelMapper.addMappings(warehouseFieldMapping);
   this.modelMapper.addMappings(warehouseMapping);
}
```

ModelMapper is used as it typically would in the actions on the controller. The mappings are automatically applied based on the class types of the objects being mapped.

Advertisement

Amazing Laptops Sale!

Get the best midrange laptop deals now!

This is used for the list. ModelMapper will automatically apply the `warehouseFieldMapping` property map.

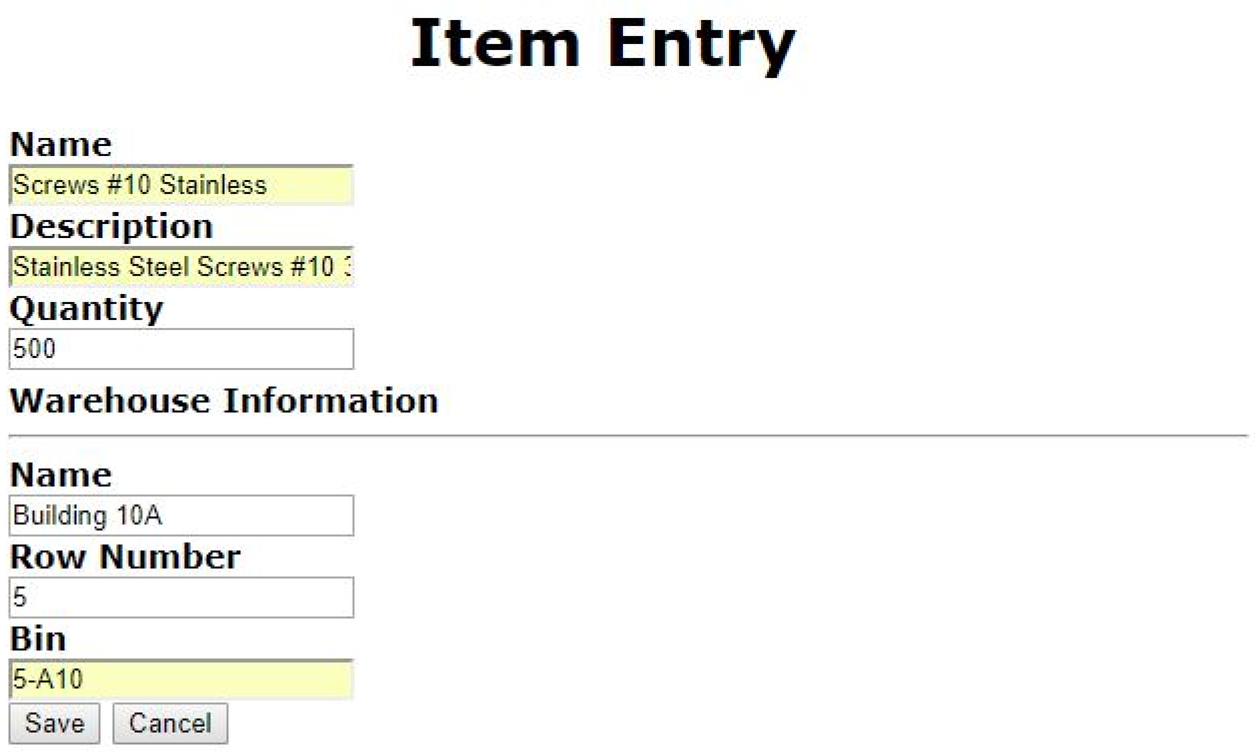
```
List<ItemDto> itemDtos = items.stream().map(item -> modelMapper.map(item
```

This is how it’s used when saving after adding or editing. ModelMapper will automatically apply the `warehouseMapping`.

```
    Item item = modelMapper.map(itemDto, Item.class);
```

Running the App

If we enter an Item like this, the warehouse fields will map to a Location object and get stored in the Location table.



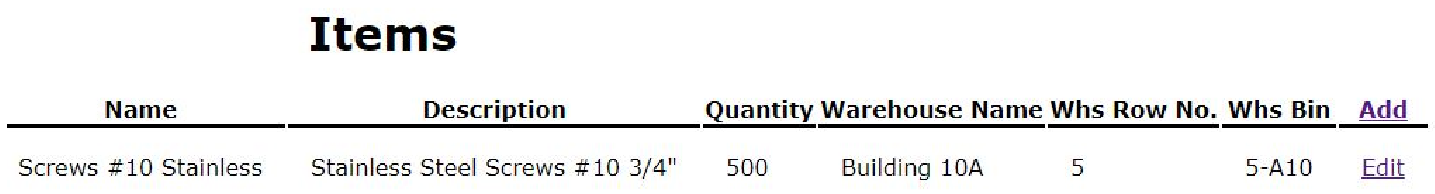
Item Entry

Name	Screws #10 Stainless
Description	Stainless Steel Screws #10 3
Quantity	500

Warehouse Information

Name	Building 10A
Row Number	5
Bin	5-A10

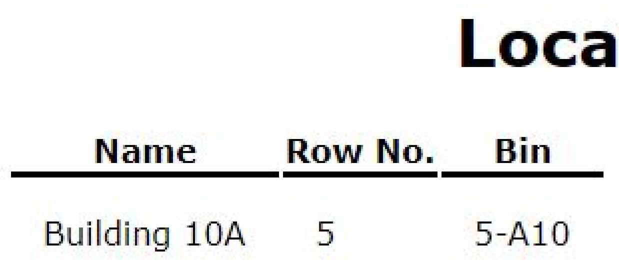
The list of Items will reflect the location fields.



Items

Name	Description	Quantity	Warehouse Name	Whs Row No.	Whs Bin	Add
Screws #10 Stainless	Stainless Steel Screws #10 3/4"	500	Building 10A	5	5-A10	Edit

When we list the locations, we see our Location has been saved into it, having been properly populated on the Item entity.



Locations

Name	Row No.	Bin
Building 10A	5	5-A10

This is a way of using Model Mapper to map from fields on a form into an object that belongs to another object. Please feel free to comment with any questions or observations.

References:

- ModelMapper Deep Mapping Documentation (http://modelmapper.org/user-manual/property-mapping/#deep-mapping)

Related Posts

- Using Custom ModelMapper Converters and Mappings (https://amydegregorio.com/2018/01/17/using-custom-modelmapper-converters-and-mappings/)
- Skipping Fields with ModelMapper (https://amydegregorio.com/2018/05/23/skipping-fields-with-modelmapper/)
- Mapping Children with ModelMapper (https://amydegregorio.com/2018/10/03/mapping-children-with-modelmapper/)
- ModelMapper in Spring Boot No Starter (https://amydegregorio.com/2018/12/16/modelmapper-in-spring-boot-no-starter/)

Tagged:
Java,
ModelMapper,
Spring Boot

Published by Amy DeGregorio

Freelance Java developer View all posts by Amy DeGregorio

5 thoughts on “Deep Mapping with ModelMapper”

Pingback: Mapping Children with ModelMapper – Amy's Programming Corner
Pingback: Using Custom ModelMapper Converters and Mappings – Amy's Programming Corner
Pingback: Skipping Fields with ModelMapper – Amy's Programming Corner
Pingback: Looking Back on 2018 – Amy's Programming Corner

Bogdan says:
January 13, 2020 at 7:00 am
thank you, Amy!

↪ Reply