Най напред импортвам скелета в application.properties си слагам моите настройки (копирам ги от друг проект), пома трябва да е както трябва

#Data Source Properties  
spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver  
spring.datasource.url=jdbc:mysql://localhost:3306/real\_estate\_agency?useSSL=false&createDatabaseIfNotExist=true  
spring.datasource.username=root  
spring.datasource.password=1234  
  
#JPA Properties  
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL8Dialect  
spring.jpa.properties.hibernate.format\_sql = TRUE  
spring.jpa.hibernate.ddl-auto = update  
spring.jpa.open-in-view=false  
  
###Logging Levels  
# Disable the default loggers  
logging.level.org = WARN  
logging.level.blog = WARN  
  
#Show SQL executed with parameter bindings  
#logging.level.org.hibernate.SQL = DEBUG  
#logging.level.org.hibernate.type.descriptor = TRACE  
  
#Change server port  
#server.port=8000  
# *ToDo: add your settings here*

после минавам и си слагам анотациите на репозиторитата

@Repository  
public interface TownRepository extends JpaRepository<Town, Long> {

И екстендвам това.(Всички неща които трябва да се дописват са маркирани с //ТОДО: и така мога да ги намеря и да ги допиша)

После сървисите, интерфейсите ги има но трябва да им направя имплементациите и да сложа анотация за сървис

@Service  
public class TownServiceImpl implements TownService {

И мога да стартирам приложението и ако всичко е наред в браузъра пиша

Localhost:8080 и трябва да се визуализира приложението.

След това трябва да направя обектите които ще се визуализират в таблици

Като следвам схемата и упътванията. За връзките м/у таблиците гледам в схемата кой от двата обекта има инстанция към свързания и правя връзката там където я има, от схемата също мога да се ориентирам за типа на данните за полетата. Валидацията ще я направя после в ДТО – то като е хубаво в документа да я оцветя в червено за да не забравя.

Едното ентити за пример

@Entity Пиша и двете анотации понеже при xml се бърка  
@Table(name = "agents")  
public class Agent {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private long id;

Долу освен името по изискване се иска името да е уникално и ако не е упоменато че полето е опционално слагам и другото  
 @Column(name = "first\_name",unique = true,nullable = false)  
 private String firstName;  
  
 @Column(name = "last\_name")  
 private String lastName;  
  
 @Column(nullable = false, unique = true)  
 private String email;  
Връзката гледаме от класа в който сме, ние сме в клас агент и един агент е в един град, но в един град има много агенти затова е тази връзка и понеже в схемата няма поле в за агентите в града изграждам връзката само тук.  
 @ManyToOne(optional = false)  
 private Town town;  
  
 public Agent() {}

Празен конструктор и гетъри и сетъри. Като ги направя стартирам приложението и проверявам дали правилно са генерирани всички таблици в базата.

След това трябва да имплементирам сървисите. В които правим четенето от json и xml и вкарване на данните в базата. Като за реда следвам документа за изпита.

Първо започвам с Града:

@Service  
public class TownServiceImpl implements TownService {  
Инициализирам нужните полета долу и вече съм поставил анотацията че е сървиз горе.

private final TownRepository townRepository;

Репозиторито за връзка с базата.

private final Gson gson;

Обект с който превръщаме от и към json файлове

private final Validator validator;

Валидатора който прави проверката спрямо поставените анотации за валидиране на входните данни.

private final ModelMapper modelMapper;

Мапер с който от един клас правим друг.  
  
 @Autowired Анотация над конструктора (да проверя точно какво значи)  
 public TownServiceImpl(TownRepository townRepository) {

this.townRepository = townRepository;  
Подаваме репозиторито а другите ги създаваме   
 this.gson = new GsonBuilder().create();  
 this.validator = Validation.*buildDefaultValidatorFactory*().getValidator();  
 this.modelMapper = new ModelMapper();  
 }  
  
 @Override   
 public boolean areImported() {

Първия метод проверява дали сме импортнали в базата, връща булева която е резултат от проверката -> взимаме броя на обектите в таблицата и го сравняваме с нула.  
 return this.townRepository.count() > 0;  
 }  
  
 @Override  
 public String readTownsFileContent() throws IOException {

Втория метод е за четене, като пътя до файла можем да го изнесем като константа   
 Path path = Path.*of*("src", "main", "resources", "files", "json", "towns.json");

Прочитаме стринга от посочения път.  
 return Files.*readString*(path);  
 }  
  
 @Override  
 public String importTowns() throws IOException {

String json = this.readTownsFileContent();  
Взимаме прочетения файл от предния метод като стринг

ImportTownDTO[] importTownDTO = this.gson.fromJson(json,ImportTownDTO[].class);  
Създаваме ДТО клас с който да валидираме и обработим входните данни. И от json ги преработваме в ДТО и понеже в json данните са много в масив създаваме масив от ДТО

List<String> result = new ArrayList<>();  
Създавам лист в който да запазя текстово съобщение за резултата от импортването на всяко ДТО към клас Град и от там в базата.

for (ImportTownDTO townDTO : importTownDTO) {

С фор цикъл всяко ДТО, може и с стрейм.

Set<ConstraintViolation<ImportTownDTO>> validationErrors =  
 this.validator.validate(townDTO);  
В този сет се записват грешките при валидация на конкретното ДТО ако има

if (validationErrors.isEmpty()){

Optional<Town> optTown =  
 this.townRepository.findByTownName(townDTO.getTownName());

И правим проверка дали е записана грешка от валидирането, ако не е значи ДТО – то е валидно и можем да го мапнем към нашия клас за таблицата, но първо проверяваме дали вече има обект със същото име в базата. Като взимаме опционален обект чрез репозиторито, като търсим по името на ДТО-то в базата   
   
  
 if (optTown.isEmpty()){  
 Town town = this.modelMapper.map(townDTO,Town.class);  
Ако опционалния обект е празен, или не намерен такъв в базата, мапваме ДТО-то към нужния ни клас чрез мапъра, който сме създали в конструктора.

this.townRepository.save(town);  
С репозиторито записваме обекта в базата

String msg = String.*format*("Successfully imported town %s - %d",  
 town.getTownName(), town.getPopulation());

Създаваме правилното съобщение и го добавяме към листа със съобщенията   
  
 result.add(msg);

}else {  
 result.add("Invalid town");

Има същия обект в базата и понеже по задание всеки обект от този тип трябва да е уникален(да не се повтаря град с едно и също име), извеждаме и добавяме правилното съобщение към резултата.  
 }  
 }else {  
 result.add("Invalid town");  
Тук обекта е дал несъответствие при валидация и отново е невалиден   
 }  
  
 }  
 return String.*join*("\n",result);

И връщаме резултата като джойнваме по нов ред(всеки записан резултат ще се изведе на нов ред)   
 }

ДТО – то:

public class ImportTownDTO {  
  
 @Size(min = 2) Анотация за валидация  
 private String townName;  
Давам имена на полетата точно както на главния клас. ПО условие името на града трябва да е > или = 2, тази анотация прави тази проверка.

@Positive Тук популацията трябва да е положително число   
 private int population;

Правим празен конструктор и гетъри   
 public ImportTownDTO() {}  
  
 public String getTownName() {  
 return townName;  
 }  
  
 public int getPopulation() {  
 return population;  
 }  
}

Репозиторито:

@Repository  
public interface TownRepository extends JpaRepository<Town, Long> {  
  
 Optional<Town> findByTownName(String townName);

Дописали сме само метода с който търсим град със същото име в базата като подадения.  
}

След като сме готови рестартираме приложението тук и в браузъра и натискам копчето за импорт на обектите като първо прочита json-на и после извежда съобщение за успешно импортване, ако всичко е както трябва.

По задание следва Агента:

@Service  
public class AgentServiceImpl implements AgentService {  
 private final Path path = Path.*of*("src", "main", "resources", "files", "json", "agents.json");  
Изнесъл съм пътя до файла иначе тук всичко е същото защото и тук входните данни са в json файл. Другото е че трябва да се добави обект от клас Град и съм добавил и репозиторито за града защото ще търся град по името на града подадено в json.   
 private final AgentRepository agentRepository;  
 private final Gson gson;  
 private final Validator validator;  
 private final ModelMapper modelMapper;  
 private final TownRepository townRepository;  
  
 @Autowired  
 public AgentServiceImpl(AgentRepository agentRepository, TownRepository townRepository) {  
 this.agentRepository = agentRepository;  
 this.townRepository = townRepository;  
  
 this.gson = new GsonBuilder().create();  
 this.validator = Validation.*buildDefaultValidatorFactory*().getValidator();  
 this.modelMapper = new ModelMapper();  
 }  
  
 @Override  
 public boolean areImported() {  
 return this.agentRepository.count() > 0 ;  
 }  
Тук двата метода са същите.  
 @Override  
 public String readAgentsFromFile() throws IOException {  
  
 return Files.*readString*(path);  
 }

@Override  
 public String importAgents() throws IOException {  
 String json = this.readAgentsFromFile();  
  
 ImportAgentDTO[] importAgentDTOS =

this.gson.fromJson(json, ImportAgentDTO[].class);  
  
Тук пускам масива в поток  
 return Arrays.*stream*(importAgentDTOS)  
 .map(this::importAgent)  
 .collect(Collectors.*joining*("\n"));  
 }  
И правя импортването в отделен метод  
 private String importAgent(ImportAgentDTO agentDTO) {  
 Set<ConstraintViolation<ImportAgentDTO>> errors =  
 this.validator.validate(agentDTO);  
  
 if (!errors.isEmpty()){  
 return "Invalid agent.";  
 }  
  
 Optional<Agent> optAgent =

this.agentRepository.findByFirstName(agentDTO.getFirstName());  
  
 if (optAgent.isPresent()){  
 return "Invalid agent.";  
 }  
  
 Agent agent = this.modelMapper.map(agentDTO,Agent.class);  
  
 Optional<Town> town = this.townRepository.findByTownName(agentDTO.getTown());  
  
 agent.setTown(town.get());  
  
 this.agentRepository.save(agent);  
  
  
  
 return String.*format*("Successfully imported agent - %s %s"  
 ,agent.getFirstName(), agent.getLastName());  
 }  
}

А сега ДТО-то :

public class ImportAgentDTO {  
Много важно е имената на полетата да съвпадат с имената в json файла.  
 @Size(min = 2)  
 private String firstName;  
  
 @Size(min = 2)  
 private String lastName;  
  
 private String town;  
  
 @Email  
 private String email;  
  
 public ImportAgentDTO() {}

И репозиторито:

@Repository  
public interface AgentRepository extends JpaRepository<Agent, Long> {  
  
 Optional<Agent> findByFirstName(String firstName);  
}

Следва XML файловете за импортване, по задание са апартаментите.

@Service  
public class ApartmentServiceImpl implements ApartmentService {  
 private final Path path =  
 Path.*of*("src", "main", "resources", "files", "xml", "apartments.xml");  
  
 private final ApartmentRepository apartmentRepository;  
 private final TownRepository townRepository;

Долу това е инструмент за преобразуване на един XML файл в джава клас, за обратното се използва Marshaller.   
 private final Unmarshaller unmarshaller;  
 private final Validator validator;  
 private final ModelMapper modelMapper;  
   
  
 public ApartmentServiceImpl(ApartmentRepository apartmentRepository, TownRepository townRepository) throws JAXBException {  
 this.apartmentRepository = apartmentRepository;  
 this.townRepository = townRepository;  
Понеже пак данните се подават в списък правим долното което създава нова инстанция към нашия ДТО в който сме посочили като поле само списък от нужните ни ДТО обекти.   
 JAXBContext context = JAXBContext.*newInstance*(ImportApartmentRootDTO.class);  
 И подаваме тази инстанция към инструмента при създаване

this.unmarshaller = context.createUnmarshaller();

Тук са валидатора и мапъра.  
 this.validator = Validation  
 .*buildDefaultValidatorFactory*()  
 .getValidator();  
  
 this.modelMapper = new ModelMapper();  
 }  
  
 @Override  
 public boolean areImported() {  
 return this.apartmentRepository.count() > 0 ;  
 }  
Тези два метода са същите като в предните два сървиса.  
 @Override  
 public String readApartmentsFromFile() throws IOException {  
 return Files.*readString*(path);  
 }  
  
 @Override  
 public String importApartments() throws IOException, JAXBException {  
Тук създаваме инстанция от нашия клас в който ще вземем списъка от XML

файла, за разлика от json тук ни трябва допълнително ДТО. И за

прочитането не използваме предния метод, а пишем последното

словосъчетание.  
 ImportApartmentRootDTO apartmentDTOs =  
 (ImportApartmentRootDTO) this.unmarshaller  
 .unmarshal(new FileReader(path.toAbsolutePath().toString()));

След като сме взели успешно списъка пускаме поток и извеждаме

импортването в отделен метод.  
 return apartmentDTOs  
 .getApartments()  
 .stream()  
 .map(this::importApartment)  
 .collect(Collectors.*joining*("\n"));  
 }  
  
 private String importApartment(ImportApartmentDTO dto) {  
  
 Set<ConstraintViolation<ImportApartmentDTO>> errors =  
 this.validator.validate(dto);  
Правим сет в който да пазим грешките при валидацията след което правим

проверка дали са се появили такива грешки, и връщаме правилното

съобщение или продължаваме нататък.  
 if (!errors.isEmpty()){  
 return "Invalid apartment";  
 }

След като е преминала първата проверка изваждам града спрямо името на

града в ДТО-то, за да може спрямо този обект(града) да намерим  
 Optional<Town> town = this.townRepository.findByTownName(dto.getTown());  
 опционален обект(апартамент) и освен града ни трябва и площа.  
 Optional<Apartment> optApartment =

this.apartmentRepository.findByTownAndArea(town.get(), dto.getArea());

Ако намерим такъв извеждаме правилното съобщение. (по условие не трябва да има два апартамента с еднаква площ в един и същ град.)  
 if (optApartment.isPresent()){  
 return "Invalid apartment";  
 }

Ако няма такъв мапваме ДТО към нашия клас   
 Apartment apartment = this.modelMapper.map(dto, Apartment.class);  
 и сетваме града в обекта  
 apartment.setTown(town.get());  
 и вкарваме апартамента в базата.  
 this.apartmentRepository.save(apartment);  
  
 и връщаме правилното съобщение.  
 return String.*format*("Successfully imported apartment %s - %.2f",  
 apartment.getApartmentType(), apartment.getArea());  
  
 }

Сега ДТО-то приемащо списъка от XML:

@XmlRootElement(name = "apartments") Анотация че е роот и името на списъка от XML  
@XmlAccessorType(XmlAccessType.*FIELD*) Анотация над името на класа задължителна  
public class ImportApartmentRootDTO {  
  
 @XmlElement(name = "apartment") Анотация над всяко поле в класа и подаваме име  
 private List<ImportApartmentDTO> apartments; на всеки елемент в списъка  
  
 public ImportApartmentRootDTO() {}  
празен конструктор и гетър.  
 public List<ImportApartmentDTO> getApartments() {  
 return apartments;  
 }  
}

ДТО-то което приема данните за мапване към нашия клас.

@XmlAccessorType(XmlAccessType.*FIELD*) Анотация над името на класа задължителна  
public class ImportApartmentDTO {  
  
 @XmlElement Анотация над всяко поле  
 private ApartmentType apartmentType;  
  
 @XmlElement  
 @DecimalMin("40.00") Валидация за дробно число  
 private BigDecimal area;  
  
 @XmlElement  
 private String town;  
  
 public ImportApartmentDTO() {}  
  
 public ApartmentType getApartmentType() {  
 return apartmentType;  
 }  
  
 public BigDecimal getArea() {  
 return area;  
 }  
  
 public String getTown() {  
 return town;  
 }

По принцип валидатора мога да го изкарам в отделен клас в папката Util(полезни). Ето как ще изглежда класа:

|  |
| --- |
| public class MyValidator { |
| private final Validator localValidator; |
| Инициализираме валидатора горе |
| public MyValidator() { |
| this.localValidator = Validation |
| .buildDefaultValidatorFactory() |
| .getValidator(); |
| } |
| В конструктора го създаваме, горе |
| public boolean isValid(Object object) { |
| Set<ConstraintViolation<Object>> errors = |
| this.localValidator.validate(object); |
| И създаваме метод, към него подаваме обекта за валидиране, вътре създаваме сет за евентуалните грешки и даваме на валидатора да валидира обекта. И връщаме истина или лъжа в зависимост дали са се записали грешки или не при валидация. |
| return errors.isEmpty(); |
| } |

Но мисля че в папката config трябва да се направи Bean configuration

|  |
| --- |
| @Configuration |
| public class BeanConfiguration { |
|  |
| @Bean |
| public MyValidator getValidator() { |
| return new MyValidator(); |
| } |
|  |
| @Bean(name = "default") |
| @Primary |
| public ModelMapper getModelMapper() { |
| ModelMapper mapper = new ModelMapper(); |
| return mapper; |
| } |
|  |
| @Bean(name = "withLocalDate") |
| public ModelMapper getAltModelMapper() { |
| ModelMapper result = new ModelMapper(); |
|  |
| result.addConverter((Converter<String, LocalDate>) context -> { |
| DateTimeFormatter format = DateTimeFormatter.ofPattern("yyyy-MM-dd"); |
| return LocalDate.parse(context.getSource(), format); |
| }); |
|  |
| return result; |
| } |
|  |
| @Bean |
| public Gson getGson() { |
| return new GsonBuilder() |
| .setPrettyPrinting() |
| .create(); |
| } |

А в класа :

private final MyValidator myValidator;

Инициализираме нашия валидатор като константа и го подаваме в конструктора.

И после за всяко ДТО от списъка валидираме като от валидатора извикваме метода и подаваме ДТО-то и правим проверка за грешки при валидирането.

Следват Офертите: