# Java Basic CRUD: TODO List

## Problem

You have been tasked to create a simple **TODO List** application. The application should hold **tasks**, which are the main app **entities**.

The functionality of the application should support **creating**, **listing, editing** and **deleting** tasks.

The application should **persist** the data into a **database**.

## Overview

### Requirements

* **Java SDK 11**
* **Spring** framework (**Spring MVC** + **Spring Boot** + **Spring Data**) **ver. 2.1.0.RELEASE**
* **Thymeleaf** view engine
* **JPA** / **Hibernate ORM** + **Spring Data** data access
* **MySQL** database

### Data Model

The Task entity holds **3 properties**:

* id – int
* title – non-empty text
* comments – non-empty text

### Project Skeletons

You will be given the applications’ skeletons, which holds about **90%** of the logic. You’ll be given some **files** (**controllers**, **models**, **views**, etc.). The files will have **partially implemented logic**, so you’ll need to write some code for the application to **function properly**.

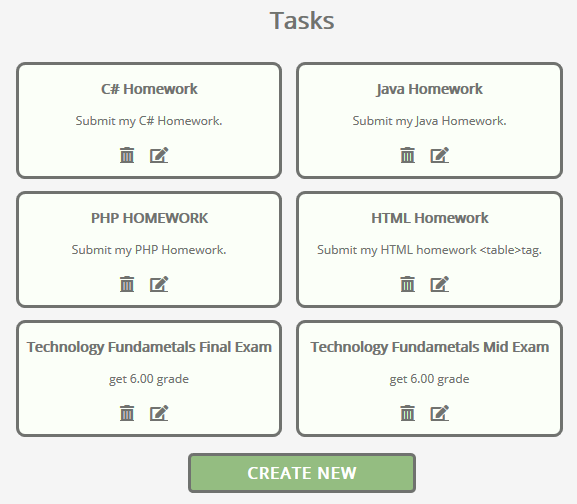
The application’s views will be given to you fully implemented. You only need to include them in your business logic.

Everything that has been given to you inside the skeleton is **correctly implemented** and if you write your code **correctly**, the application should work just fine. You are free to change anything in the Skeleton on your account.

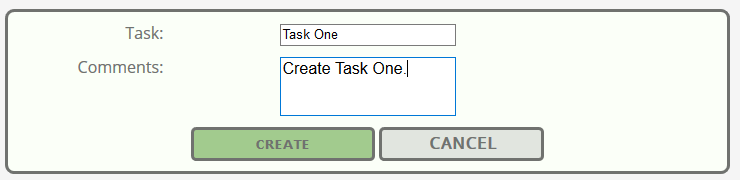
## User Interface

This is the user interface or how the application’s pages should look in their final form (fully implemented). You have several pages, described below:

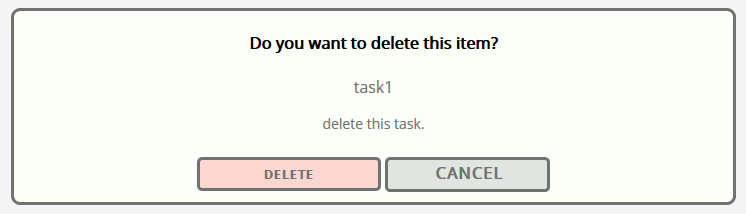
### Index Page



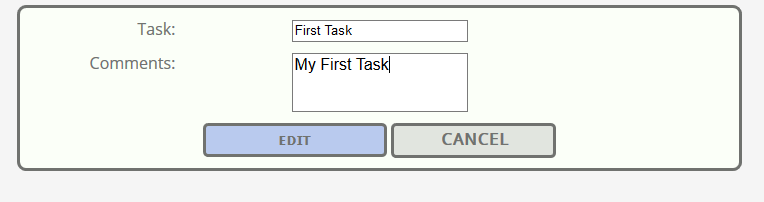
### Create Page



### Delete Page



### Edit Page

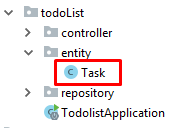


## Preparation

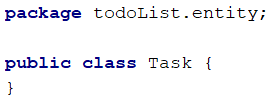
Start XAMPP and MySQL through it. Start Intellij and **import** the skeleton.

## Task Entity

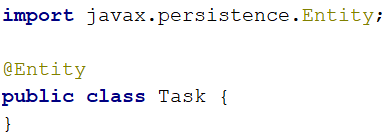
It's time to create our first entity. We are using [Hibernate](http://hibernate.org/orm/) for **ORM**. That means we are going to define our entities with [annotations](https://docs.jboss.org/hibernate/stable/annotations/reference/en/html/entity.html). In the src/main/java/todoList **package** you can see few packages that **define our project**. A **package** is a **folder** **containing** **Java** files. The one we are interested in is the "entity" package. Inside, create a **new** **java class** called "Task":

****

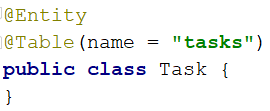
The file should look like this:



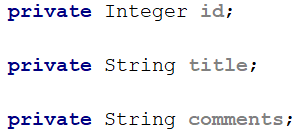
Now we need to tell **Hibernate** that this is an entity:



We need to give our database **proper table name**:



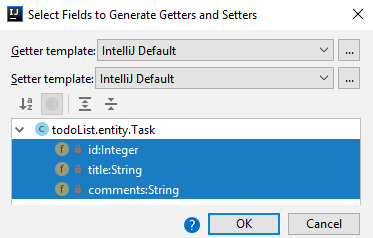
The next important thing is the **table columns**. We need columns for id, title and comments. Create the following private fields:



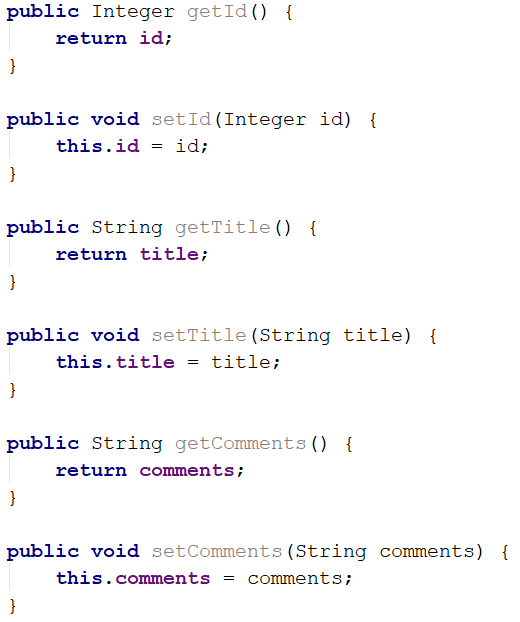
Before we explain each column, let's create [getters and setters](http://java.about.com/od/workingwithobjects/a/accessormutator.htm) for our fields. You should already be familiar with them. If you are curious why are we doing that, you can read more [here](https://www.tutorialspoint.com/java/java_encapsulation.htm). There is a **simple way to create them** in **IntelliJ Idea**. If you press "**Alt + Insert**", you should see that context menu:



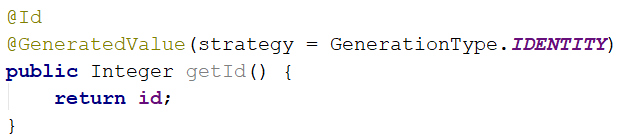
Choosing the "**Getter and Setter**" option will **open new window**. You should select **all private fields** from there:

****

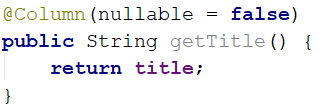
When you **click** "**OK**", you should **receive this code**:

****

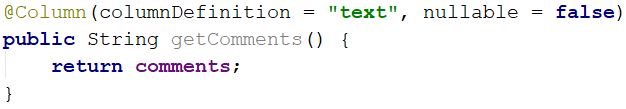
Now we can **explain each column to the database**. We are going to **place our annotations on the getters**. The first one is the id **getter**:

****

The **id** **column** will be the **primary key** in our database and as such we need to use the "@Id" annotation. The "@GeneratedValue" annotation tells **Hibernate** that the database should **generate the values automatically**. The next getter is the title:

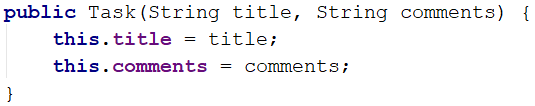
****

The "@Column" annotation gives us many useful features. For this case however, we only want to tell **Hibernate** that **this column can't be empty**. The commentsannotation is more interesting:



Here we are again making the field **required**. By default, fields of type "String" will use the **database type** "VARCHAR(255)". This type is **string** **limited** to **255** **symbols**. We can change the limit, but we can't be sure how long the content of an article will be. That's why we will **change the database type** to "text". The "text" type **doesn’t have limit** on its **length**.

It's the time to **create our constructor**:

****

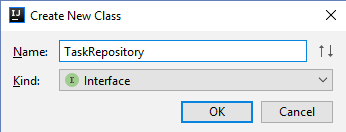
We will **use** **this** **constructor** to **create tasks** easily. However, we need to create another **empty** **constructor** for **Hibernate**:



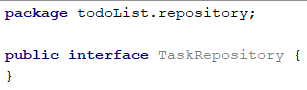
And this is pretty much everything. Our Task entity is ready.

## Task Repository

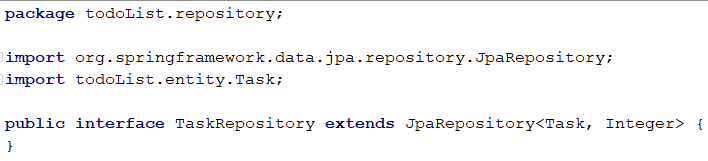
Now, we are not exactly finding a way to get the tasks. There is a way called "**Repositories**". You can imagine that the [repository](http://docs.spring.io/spring-data/data-commons/docs/1.6.1.RELEASE/reference/html/repositories.html) is our **local access** to the **database**. Using **methods** in our **repositories**, we will **get the entities** from our **database** and **use them locally**. Go to the package called "repository" and create TaskRepository:



The important thing is that it will **not be a class**. It will be an [interface](http://tutorials.jenkov.com/java/interfaces.html). The interface is a special type, which **can't contain functional methods**. It can **only** **declare them**. You should have this:



Change it to:



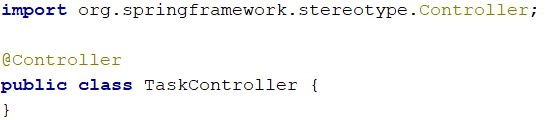
This will give us **some methods** that we are going to use later in our application.

## Task Controller

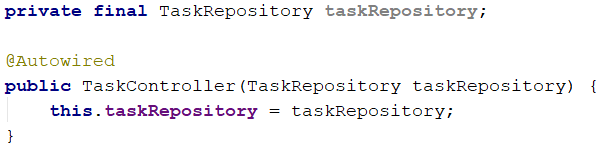
We have finally reached the point in which we can create our **controller**. In the "controller" package create a new class called "TaskController":



Add the following annotation:



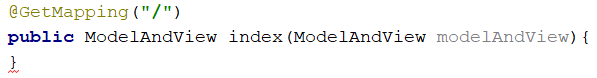
This class will **list**, **create** and **delete** tasks. That means that it will use **routes**. In order to let **Spring,** know that this class will be controller, we need to use the "@Controller" annotation. This annotation also gives us **access** to **requests** and gives us the ability to respond to them. Now, we need to create private fields that will **give** **us** **access** to the **tasks** in the **database**. These fields will be our **repositories**:



In short, **Spring** creates **object** of **each** **type** that we have in our application each time we **start** our **application**. It keeps them in something called [Spring IoC Container](https://www.tutorialspoint.com/spring/spring_ioc_containers.htm). Using the "@Autowired" annotation, we tell **Spring** that **it should initialize** and **configure** our **repositories** **automatically**. We are ready to start creating tasks.

### Listing All Tasks

First, we need to list all articles to the given route. Go to the TaskControllerand create method index.



The "@GetMapping" annotation tells **Spring** that this method **cannot be called** if the user wants to **submit data**. It should be **only used** for **viewing data**.

ModelAndView allows us to pass all the information required by Spring MVC in one return:

Thanks to the **task repository** we can get **all tasks** that are in the **database**.



We should set the view name.



In the layout we want to insert the view for the index page.



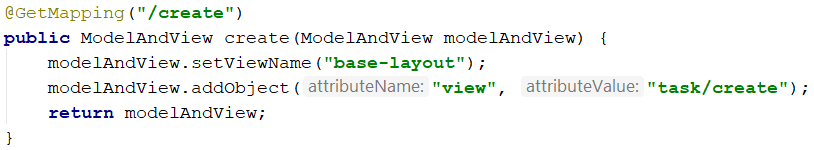
Finally, we should add tasks from the database to the modelAndViewand return it as a result.



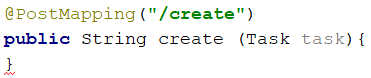
We are ready with the listing all tasks.

### Creating Tasks

First, we need to process the **get request**. As response we need to return the view to the user.



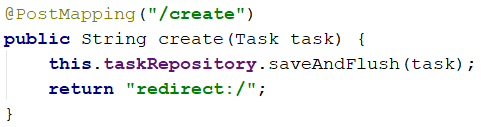
Thanks to the view that we returned the user can send data that we will store into the database. But first, we need to process it.



**Spring** will automatically map the data into task object, if we've created it correctly.

With "@PostMapping" annotation, we told **Spring** that **this method expects data** that it needs to **autofill in** task **object**. The annotation handles "POST" **request** that are usually what the **HTML forms** are using as a "method" of the **request**. In summary, the **other method** **will be called** when the user wants to **create new task** (**render the form**) and **this method** will be called **when he submits the data**.

Finally, we need to the **save the data** in our **database**, using our **task repository** and give a **response** to the user.



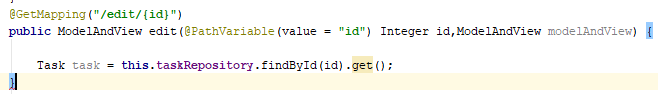
### Editing Tasks

In our TaskController we will create another method called "edit":

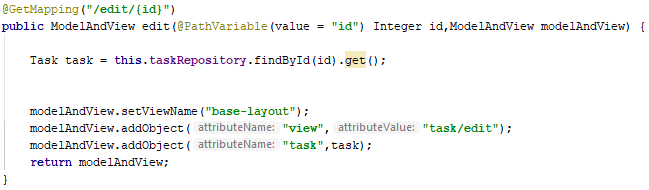


In our route, we declare parameter using curly brackets. Then in our method we use the "@PathVariable" annotation to tell Spring that this parameter should be taken from the URL. We are now free to use it in our method.

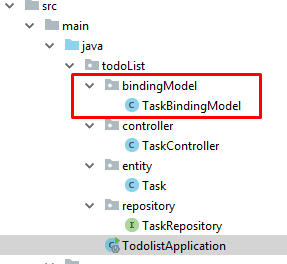
The next thing is to get the task from the database using our repository:



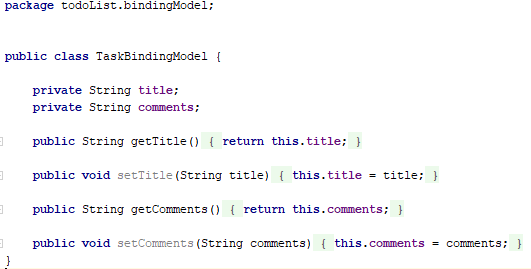
Now we want to send the article and the view to our layout:



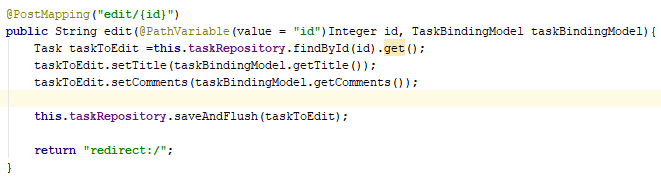
Now, we have to implement the post method for editing, but before this we should create a **BindingModel.** Let`s create a package in our directory **todoList/bindingModel** and create a new Java file called **TaskBindingModel.**



Our **BindingModel** will contains only two fields: **title and comments** with their **getters** and **setters.** It should look like this:

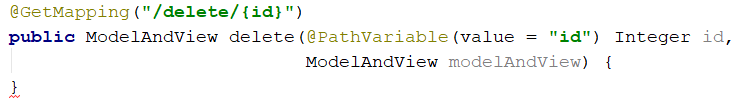


So now we can create our post method:



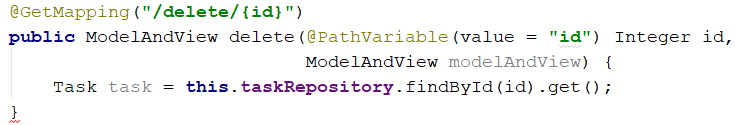
### Deleting Tasks

Here we go again. In our TaskController we will create another method called "delete":

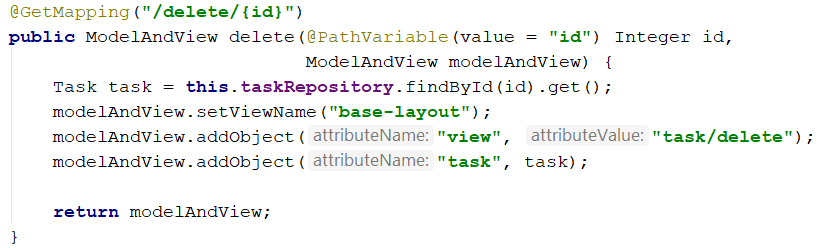


In our route, we declare parameter using curly brackets. Then in our method we use the "@PathVariable" annotation to tell Spring that this parameter should be taken from the URL. We are now free to use it in our method.

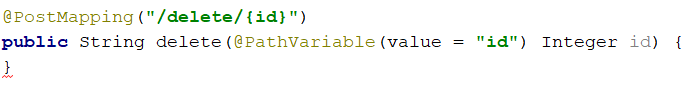
The next thing is to get the task from the database using our repository:



Now we want to send the article and the view to our layout:



Finally, we have to implement the post method for deleting. Create a new method:



We don't need a model, because we can delete an entity by id. We simply need to tell our repository to remove the entity with the given id:



The last step is to give a response to the user.



With that we finished our Java TODO List. Feel free to build on your project even further. ☺