

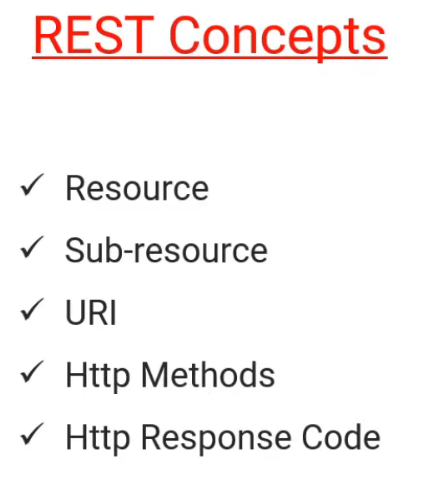
Stateless: refers that the springboot application will not store any state(data) on the server. It’s job is to fetch and process the data and return to the client.

Cacheable: If some data is fetched regularly, it is possible to cache it and the front-end/client can access it from the cache.

Layered System: 3-tier application

Uniform Interface: No matter which user/client requests for data. It’ll be returned in same uniform format.

Code on demand: The server would send the data to client that would be running on client application, when requested.

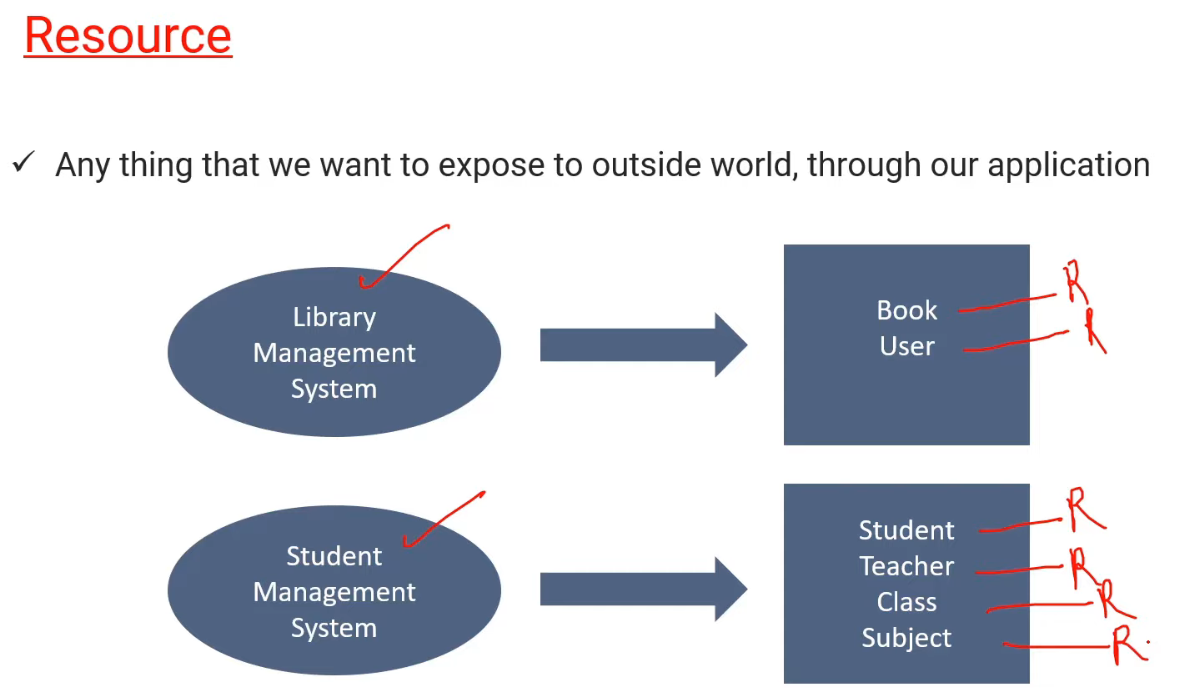


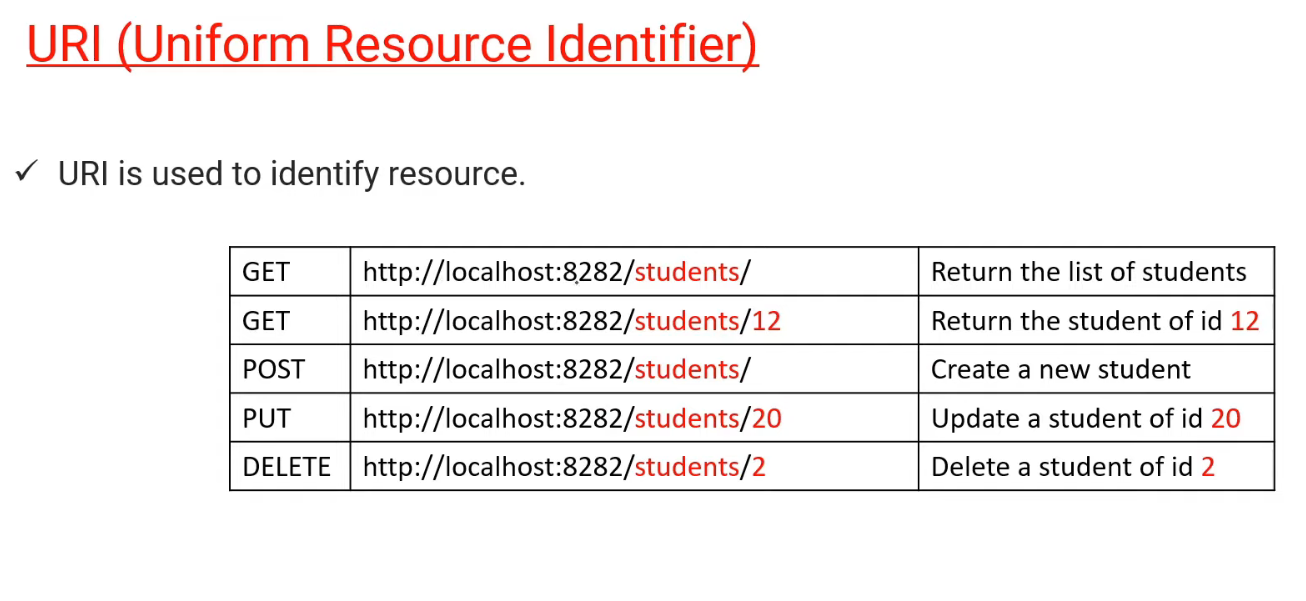
Resource: Entities or Models that we need to manage i.e. create, read, update or delete.

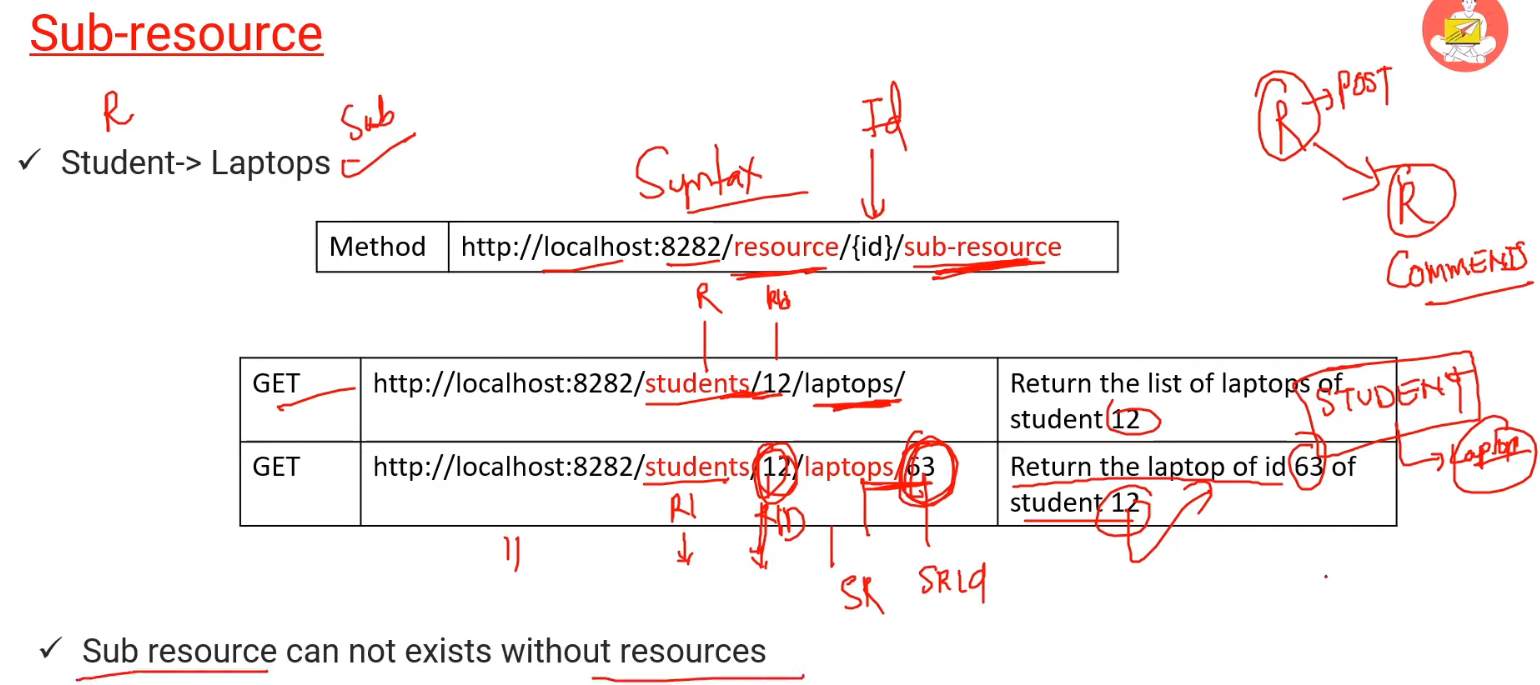
Sub Resource: Any Resource dependent on the existence of another Resource is called the sub resource.

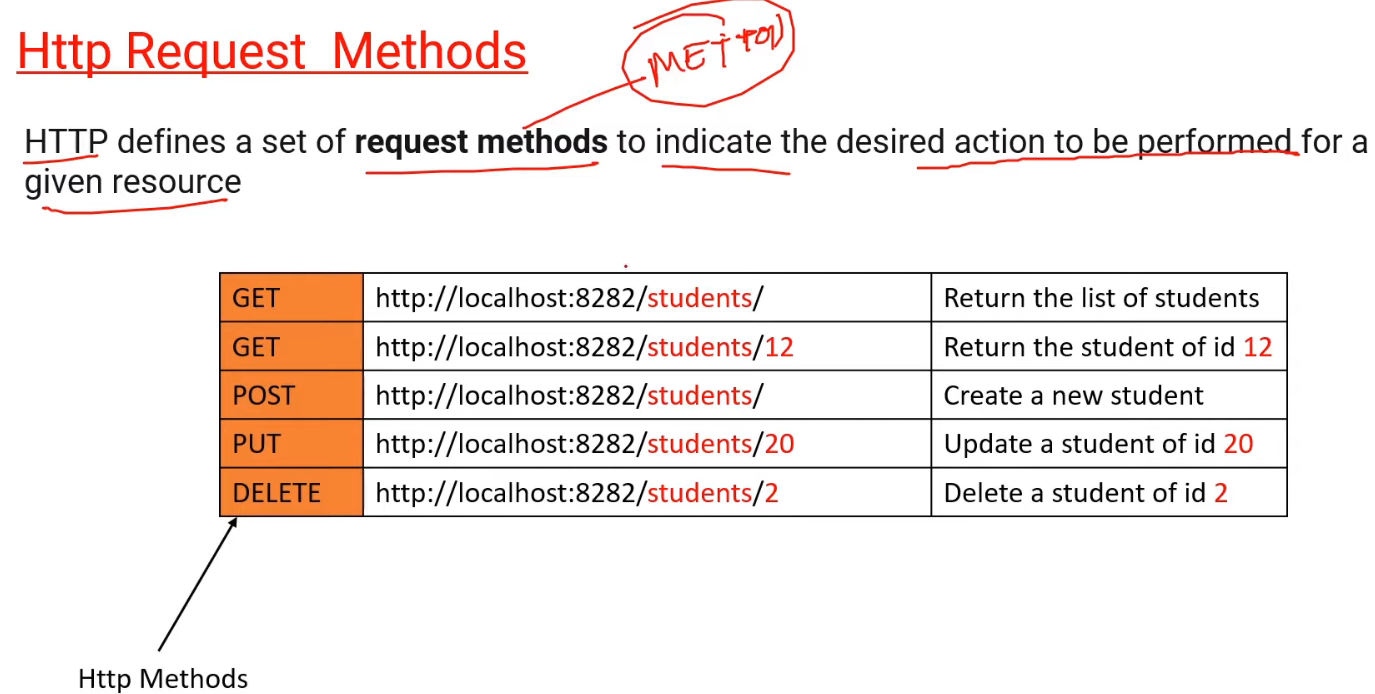
URI: Uniform Resource Identifier – the path on which we will carry our CRUD tasks.

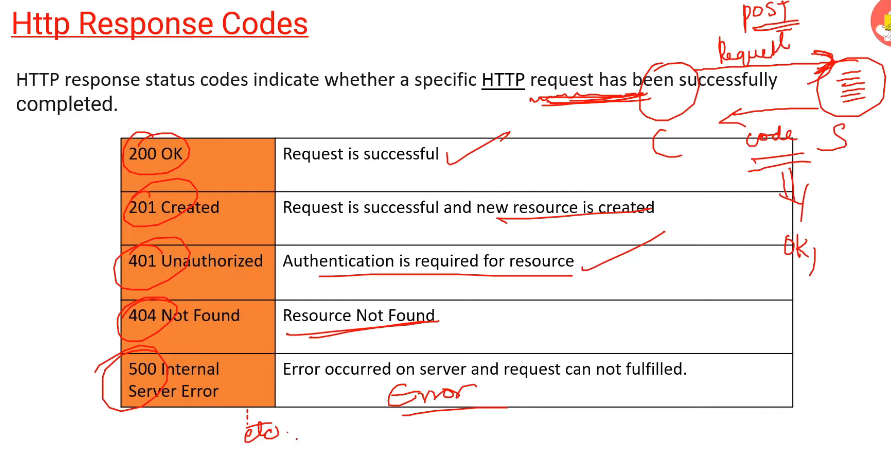
HTTP methods: These are the methods that help us understand what tasks we need to carry out for the particular Request GET, POST, PATCH, PUT, DELETE.



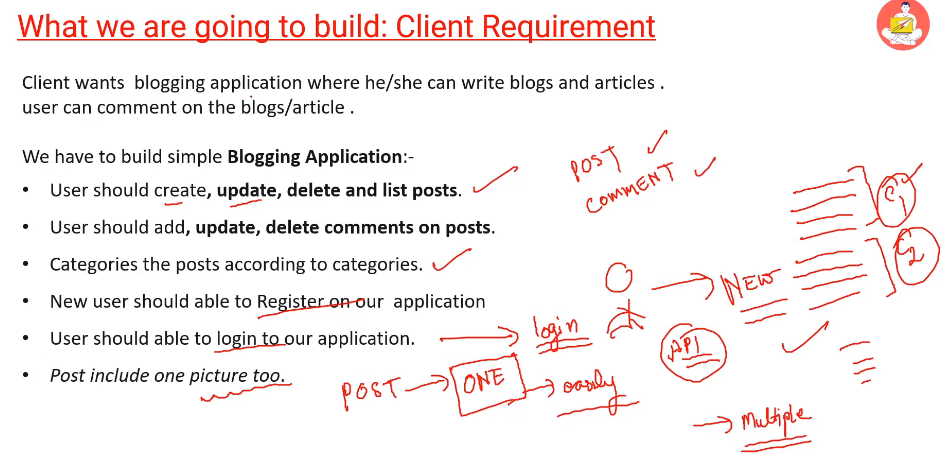


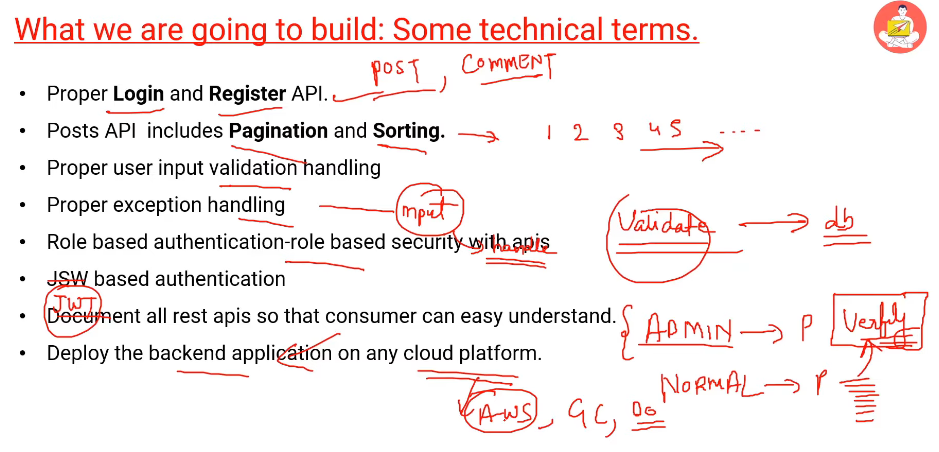


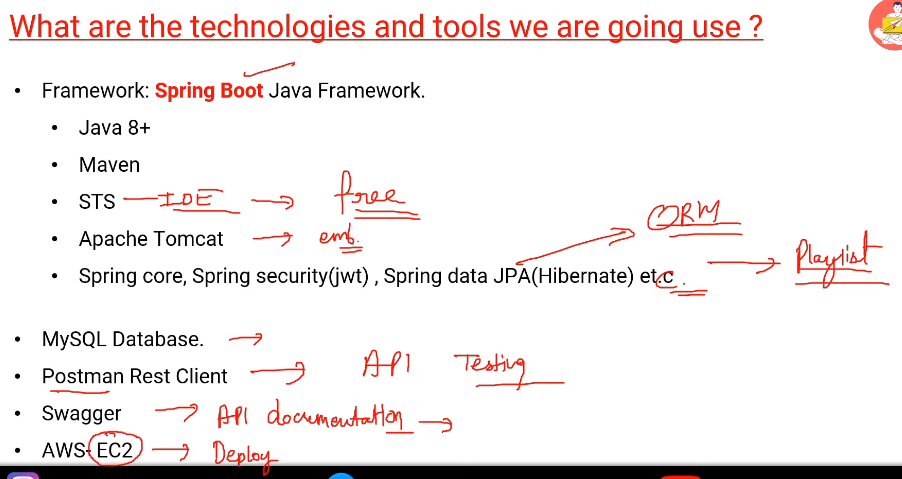


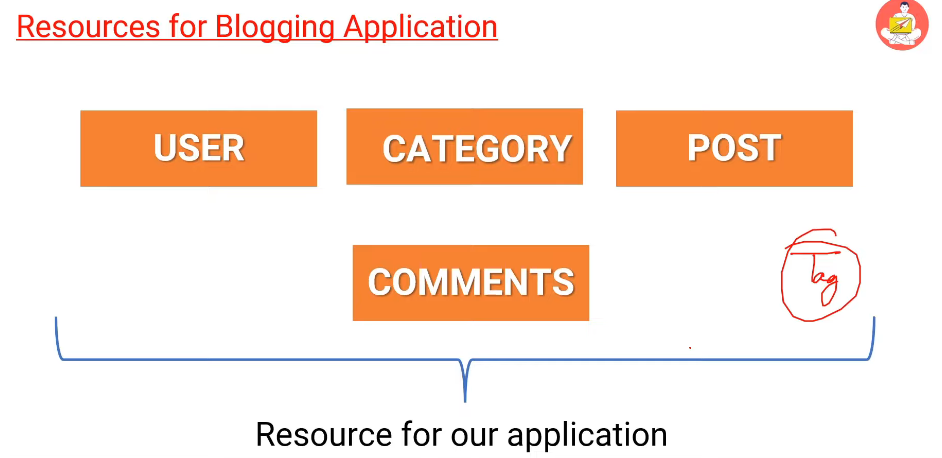


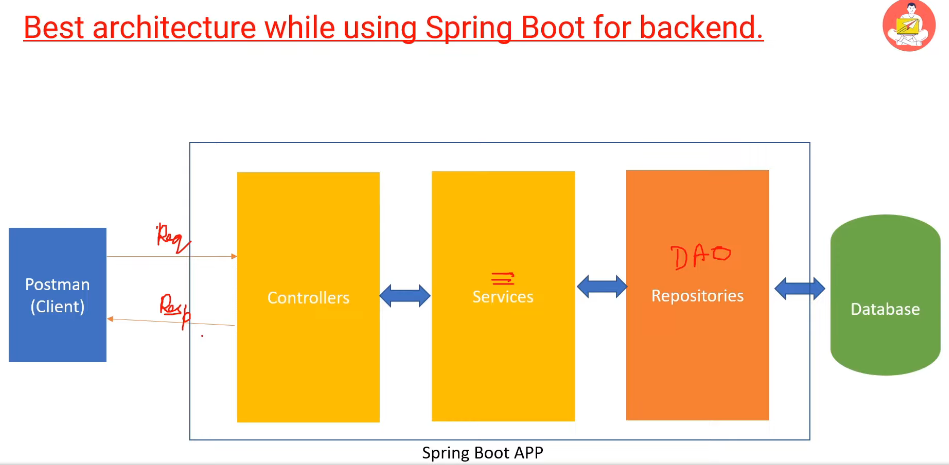
<https://cheatography.com/kstep/cheat-sheets/http-status-codes/>











server.port=9090  
  
#db configuration  
  
spring.datasource.url=jdbc:mysql://localhost:3306/blog\_app\_apis  
spring.datasource.username=root  
spring.datasource.password=  
  
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect  
# Options available are create, create-update, update, validate  
spring.jpa.hibernate.ddl-auto=update

create: Will create but destroy previous session

create-drop: Will create at start and will destroy at the end of the session

update: Will create if necessary

validate:

Exception Handling in Spring Boot Rest

@RestControllerAdvice  
public class GlobalExceptionHandler {  
  
 @ExceptionHandler(ResourceNotFoundException.class)  
 @ResponseStatus(HttpStatus.*NOT\_FOUND*)  
 public ErrorResponse resourceNotFoundExceptionHandler(ResourceNotFoundException ex, WebRequest request){  
 String message = ex.getMessage();  
 ErrorResponse response = new ErrorResponse(HttpStatus.*NOT\_FOUND*.value(), new Date(),message,request.getDescription(false));  
 return response;  
 }  
}

<https://www.bezkoder.com/spring-boot-restcontrolleradvice/>

@RestControllerAdvice: This annotation helps to is special version of @Component annotation. It keeps an eye on the “controllers” defined in the project for any occurrence of an Exception and if any exception occurs it redirects to the corresponding class it is annotated to.

The @ExceptionHandler annotation is used to inform, on which exception we need to act, and the following method executes to handle that exception.

We use @ResponseStatus along with it to send some error status code like 404 and not 20x because something went wrong. In this case we are mapping the Exception to ErrorResponse dto/payload and sending it.

Also similar to @Controller + @ResponseBody = @RestController, @ControllerAdvice +@ResponseBody = @RestControllerAdvice.

We might want our error message in following format, but it can vary:

public class ErrorMessage {

private int statusCode;

private Date timestamp;

private String message;

private String description;

}

**ModelMapper to map the objects of1 model/entity with another model.**

While we were mapping User to UserDto object, we wrote the code for manually transferring the values or state from User object to Dto object and vice-versa.

We might add more properties to our existing entities and even new entities to our project, so it would be troublesome to organise and map these, to and fro, with Dto objects.

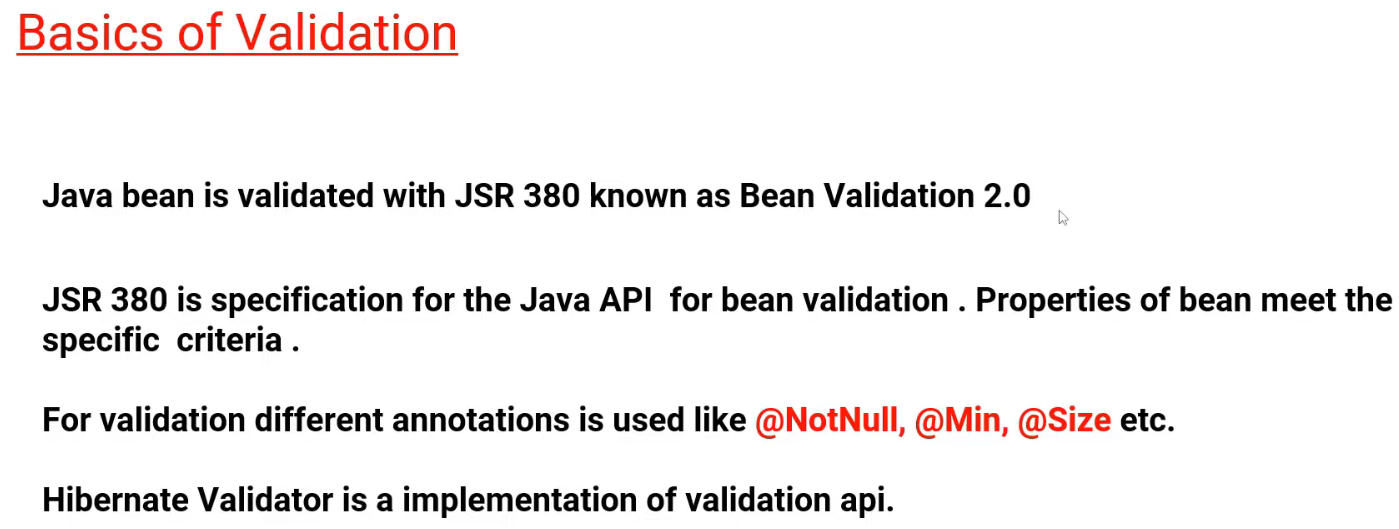
So we can use some pre-existing libraries like ModelMapper to manage or carry out this for us. There are many libraries like ModelMapper available but we will be using ModelMapper.

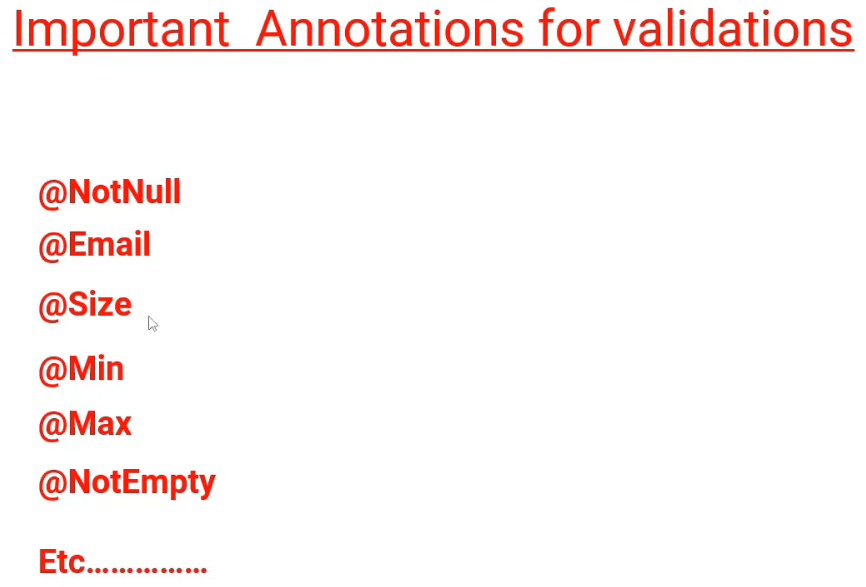
We need to first create a Bean or declare a Bean for ModelMapper, which we will do by creating a configuration class (Or declare it in main SpringBootApplication).

public User dtoToUserMapper(UserDto userDto){  
 User user = modelMapper.map(userDto,User.class);  
  
// user.setId(userDto.getId());  
// user.setName(userDto.getName());  
// user.setEmail(userDto.getEmail());  
// user.setAbout(userDto.getAbout());  
// user.setPassword(userDto.getPassword());  
 return user;  
 }

Validating the data getting stored through our REST API. We will implement it in the User module and follow similarly whenever needed.

Validating our Data using Bean Validator.





<!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-validation -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-validation</artifactId>

<version>3.1.0</version>

</dependency>

public class UserDto {  
 private int id;  
 @NotEmpty(message = "name: Field cannot be empty")  
 private String name;  
 @Email(message = "Email is invalid")  
 private String email;  
 @NotEmpty(message = "Password cannot be empty")  
 @Min(value = 8, message = "Password should have at least 8 characters")  
 @Max(value = 20,message = "Password should have at max 20 characters")  
 private String password;  
 @NotNull(message = "Invalid entry for field: about")  
 private String about;  
}

Also we have to add one more annotation in the Argument of our controller @Valid

@PutMapping("/{userId}")  
@ResponseStatus(HttpStatus.*OK*)  
public UserDto updateUser(@PathVariable int userId, **@Valid** @RequestBody UserDto userDto){  
 UserDto savedUserDto = userService.updateUser(userDto,userId);  
 return savedUserDto;  
}

Handling invalid data submitted to us.

Since we are using Bean Validator, if any incorrect data is submitted it will throw “MethodArgumentNotValidException” and return a status code of 400, i.e. Bad request.

Now in this exception message it will have all the arguments or fields along with the message for which the data received is invalid. So we will handle this exception in our GlobalExceptionHandler like we did for our previous exceptions. We will organise the invalid fields and their messages in Key-Value pair and create a response accordingly and return it.

We can directly send the Map or Key-Value pairs, but here we converted it to a string and sent it as message ErrorResponse payload.

@ExceptionHandler(MethodArgumentNotValidException.class)  
@ResponseStatus(HttpStatus.*BAD\_REQUEST*)  
public ErrorResponse methodArgumentNotValidExceptionHandler(MethodArgumentNotValidException ex, WebRequest request){  
  
 Map<String,String> resp = new HashMap<>();  
  
 ex.getBindingResult().getAllErrors().forEach((error)->{  
 String fieldName = ((FieldError)error).getField();  
 String message = error.getDefaultMessage();  
 resp.put(fieldName,message);  
 });  
  
 ex.getFieldErrors().stream().forEach(System.*out*::println);  
 String message = ex.getMessage();  
 ErrorResponse response = new ErrorResponse(HttpStatus.*BAD\_REQUEST*.value(), new Date(),resp.toString(),request.getDescription(false));  
 return response;  
}

<https://stackoverflow.com/questions/2990799/difference-between-fetchtype-lazy-and-eager-in-java-persistence-api>