Lombok and DTO

Introduction

In this document, we will explore the Lombok, what Lombok is, and its use. Additionally, we will see what DTO uses and needs, along with a sample implementation. This document provides additional information to guide you in exploring more topics.

What Is Lombok?

Lombok is a popular Java library that simplifies and reduces boilerplate code in Java applications by providing annotations that generate common code constructs like getters, setters, constructors, and more during the compile time. Here are some critical points about Lombok:

- **Reducing Boilerplate Code**: Lombok helps developers reduce the repetitive and boilerplate code they must write. In Java, classes often require getter and setter methods, constructors, `equals()`, `hashCode()`, and `toString()` methods. Lombok allows you to annotate your classes and fields to automatically generate these methods, saving you from writing them manually.

- **Annotations**: Lombok provides a set of annotations, such as ***`@Getter``***, ***`@Setter`***, ***`@NoArgsConstructor`***, ***`@AllArgsConstructor`***, ***`@EqualsAndHashCode`***, and ***`@ToString`***, among others. You apply these annotations to your classes and fields to indicate which code should be generated.

- **Compile-Time Code Generation**: Lombok operates at the compile-time stage, meaning the code is generated when you build your project using a Java compiler. This generated code becomes part of your compiled classes and is not present in the source code, making your source code more concise and readable.

- **Readability and Maintainability**: By reducing boilerplate code, Lombok can make your Java code more concise and easier to read. It also reduces the chances of introducing errors in the manually written getter, setter, or other methods.

- **Integration**: Lombok can easily be integrated into popular Java development environments like Eclipse, IntelliJ IDEA, and Maven. Many build tools and IDEs have plugins supporting Lombok, making it convenient.

- **Compatibility**: While Lombok simplifies code and enhances readability, it's essential to be aware that it modifies your bytecode during compilation. This means that the generated code may not be visible in your source files, which can confuse developers unfamiliar with Lombok.

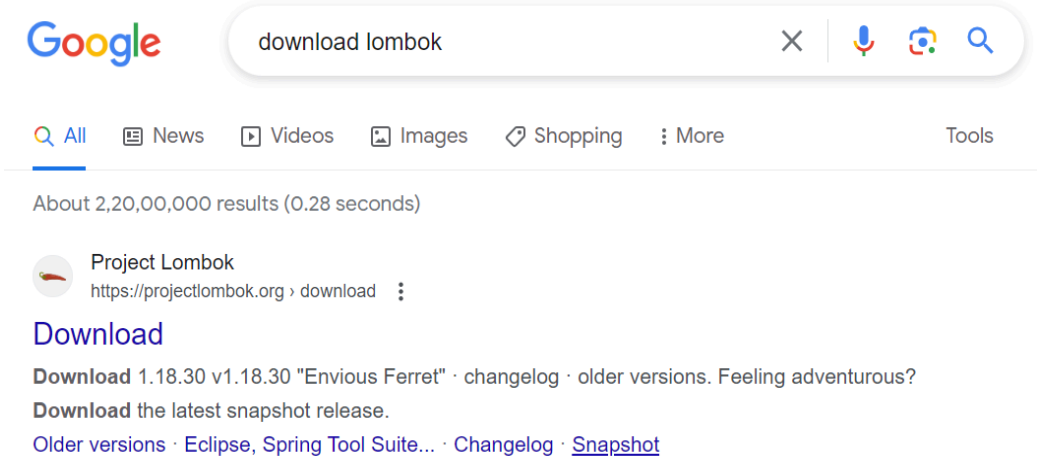
- **Customisation**: Lombok provides customisation options through annotations, allowing you to fine-tune the generated code as needed. For example, you can

exclude specific fields from generating getters and setters using the `@Getter` and `@Setter` annotations.

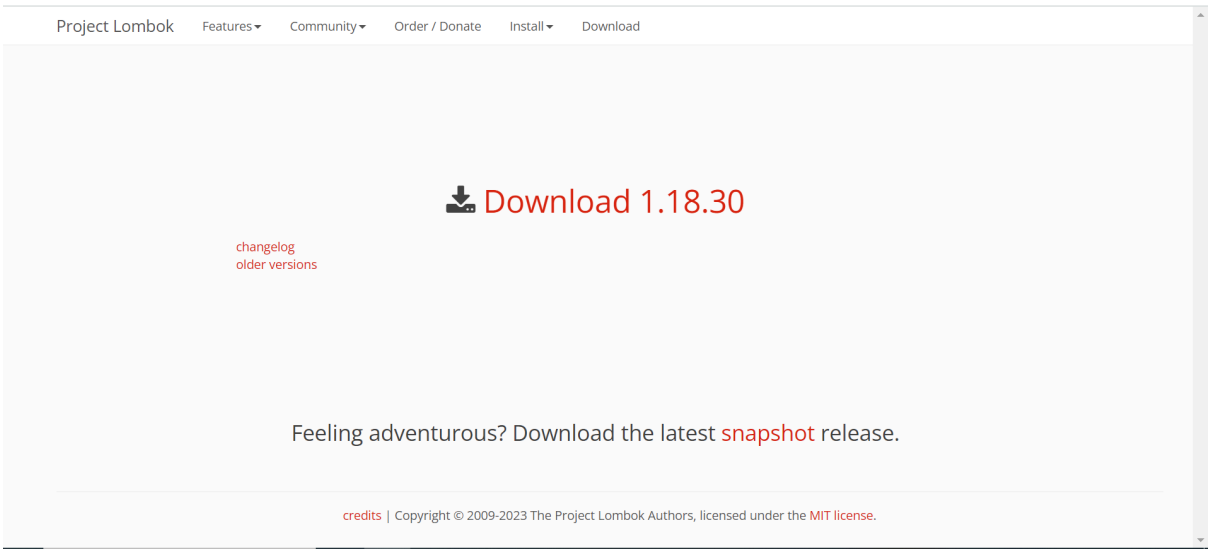
Lombok is a valuable tool for Java developers looking to reduce boilerplate code, enhance code readability, and increase productivity by automatically generating commonly used methods. However, it's crucial to understand how Lombok works and use it judiciously to maintain code clarity and ensure that all team members are comfortable with its usage.

1. Installing Lombok

- Just Search ***"download Lombok"*** on the Google search engine and open the first result link:



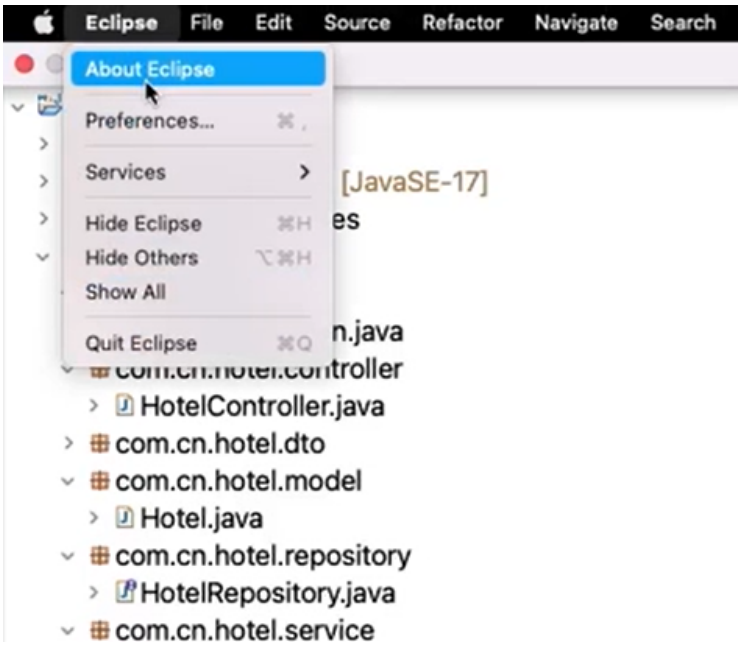
- Click on download



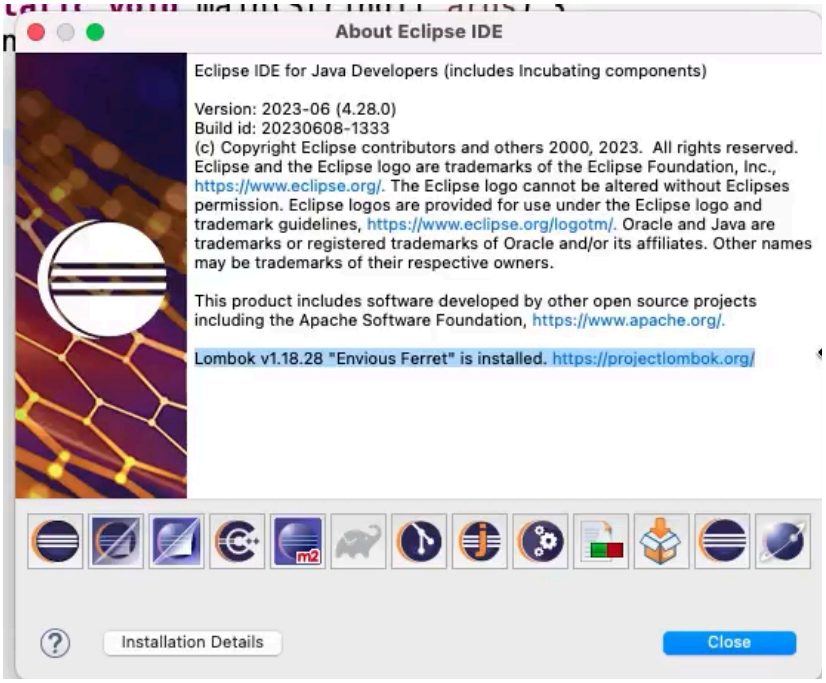
- After clicking on the link, a jar file for Lombok will be downloaded.

- Click on the downloaded Lombok.jar file
- If installed IDEs paths are not showing automatically then you have to click on Specify a location to provide a path of IDEs

- After the installation, you can confirm it by opening your project in Eclipse IDE and clicking on **'about eclipse'.**



  - An 'About Eclipse IDE' window will pop up specifying the successful installation of Lombok in your Eclipse IDE.



- Add the following Lombok dependency in thepom.xml file.

```
<dependency>
     <groupId>org.projectlombok</groupId>
     <artifactId>lombok</artifactId>
     <optional>true</optional>
   </dependency>
```

2. What is a DTO (Data Transfer Object)?

A ***Data Transfer Object (DTO)*** is a design pattern commonly used in software development to transfer data between different layers, components, or modules of an application. DTOs are lightweight objects designed solely to carry data and do not contain any business logic or behaviour. Here are some critical points about DTOs:

- **Data Transfer**: DTOs are primarily used for transferring data between different parts of an application. This could be between a client and a server, between different layers of an application (e.g., the presentation layer and the data access layer), or between various components or services.

- **No Behavior**: DTOs are essentially data containers, and they do not have any behaviour or methods associated with them. They typically consist of fields (attributes), getters, and setters to access and modify those fields.

- **Encapsulation**: DTOs encapsulate a specific set of data attributes that must be exchanged between components. They provide a structured way to package data, making it easier to manage and pass around.

- **Reducing Network Calls**: In distributed systems or client-server architectures, DTOs can help reduce the number of network calls by bundling multiple pieces of data into a single DTO object. This can improve performance and reduce latency.

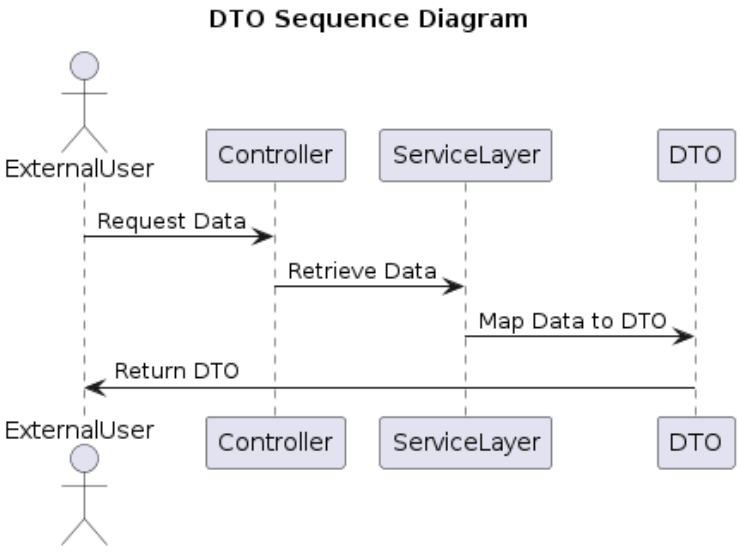
- **Mapping**: In many cases, DTOs are used to map data between different data models or structures. For example, they can convert data from a database entity to a format suitable for presentation on a user interface.

- **Security**: DTOs can also be used to control the data exposed to clients. They allow you to exclude sensitive or unnecessary data from being sent to the client, enhancing security.

- **Versioning**: In evolving systems, DTOs can be versioned to handle changes in the data structure without breaking existing clients. New fields can be added to newer versions of the DTO while maintaining compatibility with older versions.

- **Immutability**: In some cases, DTOs are designed as immutable objects, meaning their state cannot be modified once created. This ensures that the data they carry remains consistent.

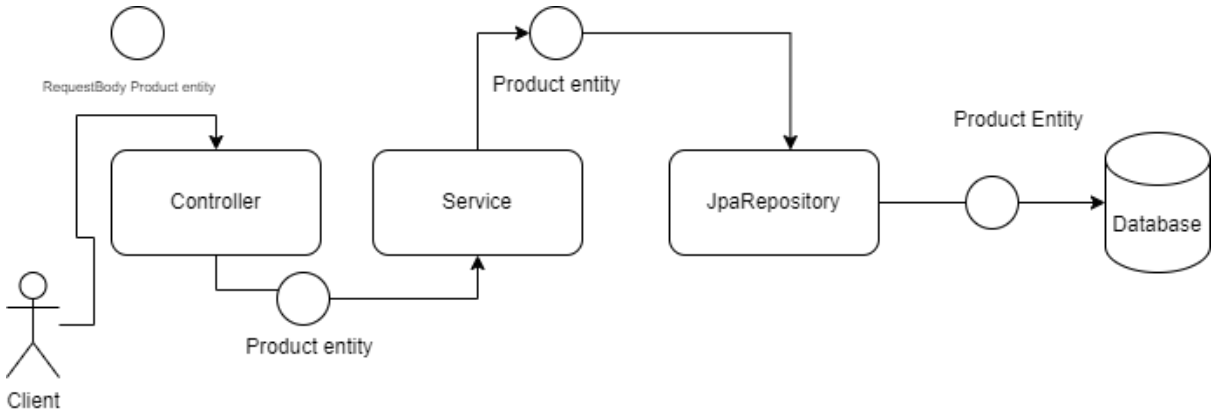
DTOs are a valuable tool for maintaining the separation of concerns and ensuring efficient data transfer between different parts of an application. They help improve code readability and maintainability by clearly defining the data exchange structure. However, it's crucial to use DTOs judiciously and not create overly complex or redundant DTOs that can lead to increased maintenance overhead.



(Example of how DTO is Used Source: https://blog.stackademic.com/)

3. Example of DTO Implementation:

The diagram below shows the flow between layers of ***Product Application.***



As you can see, we are using the Product entity information through the different layers of the application.

We will see a step-by-step implementation of DTO in a sample Product Application.

1. Consider a Product Entity class:

```
@Entity
@Table(name = "Product")
public class Product{
   @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer password;
    private String name;
    private double price;
    private String category;
}
```

ProductRequestDTO Class

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

public class ProductRequestDTO {
    private String name;
    private double price;
    private String category;
    //getters and setters
}
```

In this code:

- We created a new DTO(Data Transfer Object) class (**ProductRequestDTO**) with only attributes name, price and category.
- Creating the **ProductRequestDTO** class is to act as a DTO, which will act as an intermediary for transferring data between different parts of your application and help separate concerns between different layers.
- We have encapsulated the DTO class with the relevant information required of the Entity class(i.e. Name, price and category) by removing the 'id' attribute.
- We will now use this **ProductRequestDTO** class as the DTO for data transfer through different application layers.

2. The Product Application contains the following **ProductService** class.

ProductService Class

```
@Service
public class ProductService {

    private final ProductRepository productRepository;


    public Product createProduct(Product product) {

        return productRepository.save(product);
    }
}
```

- In this code, we use the **Product** Entity as the argument for the *createProduct* method.
- We will now make the following change in the **ProductService** class.

```
@Service
public class ProductService {

    private final ProductRepository productRepository;
    public Product createProduct(ProductRequestDTO productRequest) {
        Product product = new Product();
        product.setName(productRequest.getName());
        product.setPrice(productRequest.getPrice());
        product.setCategory(productRequest.getCategory());

        return productRepository.save(product);
    }

}
```

In this code:

- As you can see here, we have replaced the argument from **Product to ProductRequestDTO**.

- The **createProduct** method creates a new **Product** entity object and maps the data from the **ProductRequestDTO** object.
- Finally, we are passing the entity object *product* in the "*save*" method of **ProductRepository,** which will eventually save the data in the database.
- Thus, we have achieved the implementation of DTO in the *ProductService* class.

3. The ProductController class:

```
@RestController
@RequestMapping("/product")
public class ProductController{

@Autowired
ProductService productService;

  @PostMapping("/create")
  public void createProduct(@RequestBody Product product){
  productService.createProduct(product);
  }
}
```

We will now make the changes accordingly in the ProductController class as well.

- Here, we will replace the **Product** object argument for the createProduct method with the **ProductRequestDTO** object.

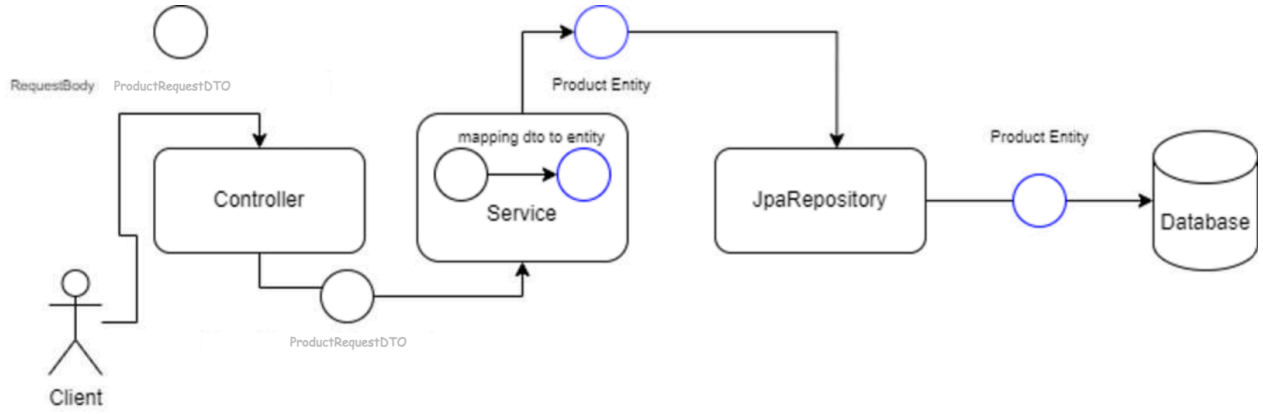
- This DTO object(*productRequest*) will be passed as the parameter to the *createProduct* of the **ProductService** class.

- The purpose of doing this is to avoid unnecessary exposure of the **Product** entity class. Instead, we will use the **ProductRequestDTO** class object to transfer only relevant data between the methods.

```
@RestController
@RequestMapping("/product")
public class ProductController{
@Autowired
ProductService productService;

  @PostMapping("/create")
  public void createProduct(@RequestBody ProductRequestDTO productRequest){
     productService.createProduct(productRequest);
  }
}
```

After making the following changes, the flow between the application layers looks like this.



Let's look at another example for having a better understanding of the DTO implementation.

2. Consider a sample Employee Management Application.

The Employee Entity class:

```
@Entity
@Table(name = "Employee")
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String firstName;
    private String lastName;
    private int age;
    private String department;
}
```

EmployeeRequest DTO Class:

```
public class EmployeeRequestDTO {
    private String firstName;
    private String lastName;
    private int age;
    private String department;
}
```

In this code:

- We have created a new DTO (Data Transfer Object) class named **EmployeeRequestDTO** with attributes *firstName, lastName, age, and department.*
- The purpose of creating the EmployeeRequestDTO class is to act as a DTO, serving as an intermediary for transferring data between different parts of the application.

This separation of concerns between different layers of the application promotes clean architecture.

- The DTO class encapsulates the relevant information required for the Employee entity class, which includes attributes such as *firstName, lastName, age, and department*. The *id* attribute is intentionally omitted from the DTO as it's typically generated by the database.

- We will use the **EmployeeRequestDTO** class to facilitate data transfer across various layers of the application.

The Employee Management Application contains the following **EmployeeService** class:

```
@Service
public class EmployeeService {
    private final EmployeeRepository employeeRepository;

    public Employee updateEmployee(Employee employee) {

        return employeeRepository.save(employee);
    }
}
```

After the DTO implementation, it would look like this:

```
@Service
public class EmployeeService {
    private final EmployeeRepository employeeRepository;

    public Employee updateEmployee(EmployeeRequestDTO employeeRequest) {
        Employee employee = new Employee();
        employee.setFirstName(employeeRequest.getFirstName());
        employee.setLastName(employeeRequest.getLastName());
        employee.setAge(employeeRequest.getAge());
        employee.setDepartment(employeeRequest.getDepartment());

        return employeeRepository.save(employee);
    }
}
```

In this code:

To implement DTO functionality, we made a change in the EmployeeService class:

Here, we have replaced the argument type from **Employee** to **EmployeeRequestDTO**. The *createEmployee* method creates a new **Employee** entity object and maps the data from the **EmployeeRequestDTO** object. Finally, we save the entity object using the *employeeRepository.save(employee)* method, persisting the data in the database.

The **EmployeeController** class:

```
@RestController
@RequestMapping("/employee")
public class EmployeeController {

    @Autowired
    EmployeeService employeeService;

    @PostMapping("/update")
    public void updateEmployee(@RequestBody Employee employee) {
        employeeService.updateEmployee(employee);
    }
}
```

After the DTO implementation, it would look like this

```
@RestController
@RequestMapping("/employee")
public class EmployeeController {

    @Autowired
    EmployeeService employeeService;

    @PostMapping("/create")
    public void createEmployee(@RequestBody EmployeeRequestDTO
employeeRequest) {
        employeeService.updateEmployee(employeeRequest);
    }
}
```

In the EmployeeController class, we made the following changes:

We replaced the **Employee** object argument for the *updateEmployee* method with the **EmployeeRequestDTO** object (employeeRequest).

This DTO object is then passed as a parameter to the *updateEmployee* method of the EmployeeService class.

By making these changes, we ensure that only relevant data is transferred between methods and layers of the application, avoiding unnecessary exposure of the Employee entity class.

This practice follows the principle of encapsulation and maintains a clean separation of concerns between DTOs and entities.

4. Conclusion

- Project Lombok is a powerful Java library that helps reduce boilerplate code by generating methods such as getters, setters, constructors, and more through annotations. This results in cleaner and more concise code, enhancing code readability and maintainability.
- DTOs (Data Transfer Objects) are design patterns used for transferring data between different layers or components of an application. They are lightweight, contain only data without any business logic, and aid in separating data representation from application logic.

5. Reference

- A Deep Dive into the DTO Pattern with Spring Boot and MySQL
- Lombok
- DTO