Introduction

This Document provides a comprehensive overview of the Backend course offered by the PHU ACM Student Chapter, covering everything from the fundamentals of reading and writing to a MySQL database using Java, to building large, scalable applications using vanilla Java. The course emphasizes best practices and incorporates valuable knowledge gained along the way, including the use of the MVC (Model-View-Controller) architecture, layered infrastructure, and designing robust API endpoints to enable effective communication between clients and servers, all aimed at developing maintainable and extensible applications.

Throughout the course, you will work primarily with technologies such as JDBC, MySQL, and core Java, gaining hands-on experience with essential backend development tools. By the end of this course, you will have acquired the skills to build your own backend applications, establish database connections securely, implement API endpoints, and structure your projects following clean code principles and best practices for testing and maintenance.

This course is designed for learners with basic programming knowledge who wish to deepen their understanding of backend development and build scalable, production-ready applications. The PHU ACM Student Chapter hopes that upon completing this course, you will be equipped to create your own applications, host your own services, and share the knowledge you have acquired with others because ***knowledge is not an asset to monopolize, but to be shared***.

Created by: Ibrahim Radwan

Maven

`Maven` is a **build automation tool** primarily used for managing Java projects. It simplifies the process of compiling, testing, and packaging Java applications by handling project dependencies, build processes, and other project-related tasks. When you create a project in IntelliJ IDEA using Maven, it allows you to leverage Maven's capabilities directly within the IDE.

It helps in **Dependency Management** , it manages the external libraries your project depends on. Instead of manually downloading JAR files, you declare your dependencies in a `pom.xml` file, and Maven automatically downloads them from a central repository found here Maven Repository.

Example:

```xml
<dependencies>
    <dependency>
        <groupId>com.mysql</groupId>
        <artifactId>mysql-connector-j</artifactId>
        <version>9.0.0</version>
    </dependency>
</dependencies>
```

It also has **Project Structure** which enforces a standard project structure (known as a Maven "convention"), making it easier to organize and maintain large projects. Maven uses a directory structure like:

- `src/main/java`: Your application's Java source files
- `src/main/resources`: Resources like configuration files
- `src/test/java`: Test code
- `src/test/resources`: Test-related resources

Exception Handling

all of the below use `import java.io.*;`

```
import java.io.*;

public class Main {
    public static void main(String[] args) {
        String path = "/path/to/file.txt";
        FileInputStream input = new FileInputStream(path);
        System.out.println(input.available());
        input.close()
    }
}
```

`input.available()` return the numbers of characters (bytes) inside the file, if the file is empty return 0. If the file doesn't exist it crash.

`input.close()` closes the input stream because it will take a lot of your device resources.

```
public static void main(String[] args) {
    String path = "/path/to/file.txt";
    try {
        FileInputStream input = new FileInputStream(path);
    } catch (IOException ex){
        System.out.println(ex.getMessage());
    }
```

Java doesn't allow you to open files stream just like this, it will force you to use `try-catch` blocks to handle if the file doesn't exist and if any errors squires while opening it.

`IOException ex` stands for **Input/Output Exception**, and it occurs when there is a failure or an issue during input or output operations, such as reading from or writing to a file, network communication, hardware failure,or accessing a resource.

`ex.getMessage()` return the error message which display the cause of the problem

```java
public static void main(String[] args) {
    String path = "/path/to/file.txt";
    try {
        FileInputStream input = new FileInputStream(path);
        FileOutputStream output = new FileOutputStream(path);
        String x = "Name";
        output.write(x.getBytes());
        while (input.available() > 0){
            System.out.print((char) input.read());
        }
    } catch (IOException ex){
        System.out.println(ex.getMessage());
    }
}
```

You convert the data to bytes using `x.getBytes()` because file streams like `FileOutputStream` handle raw byte data. This conversion ensures that the stream can correctly process the data for writing to a file.

`while (input.available() > 0){System.out.print((char) input.read());}` when reading it reads it byte by byte (character by character) which means that it print the Unicode number so the `(char)` covert them to readable characters.

```java
public static void main(String[] args){
    String path = "/path/to/file.txt";
    FileWriter wr;
    try {
        wr = new FileWriter(path);
        wr.write("Name");
        wr.flush();
    } catch (IOException ex){
        System.out.println(ex.getMessage());
    }
}
```

`wr.flush()` Ensure all data is written before closing and no need for close() after it.
this way is same as before but simple.

Key Differences

Feature	FileInputStream/FileOutputStream	FileWriter
Purpose	Reading and writing binary data	Writing character data (text)
Data Type	Byte streams (binary files)	Character streams (text files)
Encoding	Does not handle encoding; writes raw bytes	Handles character encoding automatically
Common Use Cases	Images, audio files, videos, etc.	Text files (e.g., `.txt`, `.csv`, etc.)
Character Encoding Support	No encoding; raw byte handling	Supports character encoding (e.g., UTF-8)

> In some code examples online, you may see `throw exception` used to indicate that an exception has occurred. However, simply throwing an exception only signals the system or console about an error, without handling it. By using `try-catch` blocks, you can manage exceptions directly within the same method, handling multiple types of exceptions as needed. This approach not only makes your code more robust but also aligns with best practices for error handling.

Database Connection

```java
import java.io.*;
import java.util.Properties;

public class Main {
    public static void main(String[] args){
        String path = "/path/to/file.txt";
        FileInputStream file = new FileInputStream(path);
        Properties properties = new Properties();
        properties.load(file);
        String s = properties.getProperty("name");
        System.out.println(s);
    }
}
```

`Properties` class handle data as a set of key-value pairs where both the key and the value treated as string. It is often used to store configure data within web application, database connections, user preferences.....etc. `IT DOESN'T READ JSON FILES` it is for later.

Example:

```
name=myName
age=20
wight=95
```

when it read the file it save it in a form of a hash table

- we are going to use the `Properties` class to connect to a database, so we need a config file the have the 3 of the most important things in this format

```
url=jdbc:mysql://localhost:3306/DatabaseName
username=root
password=
```

- `url`: which is the path for the database and also contains
  1. `jdbc` It is an API in Java that allows you to connect and interact with relational databases. it provides methods to execute SQL queries, retrieve data, and update records from Java applications.
  2. `mysql` Specifies the type of database.
  3. `localhost` means the MySQL server is running on the same machine where the Java application is executing.
  4. `3306` default port number for my sql, if you work with different port you need to specify it.
  5. `DatabaseName` which is the name of the database use.
- `username` This is the username used to authenticate the connection to the database.
- `password` this one is clear -_-, here it blank because it is dummy data in real life if you left it blank and an attacker got the name of the database he can access it without password so be careful `password=your_password_here`.

Starting Connection

```
package Connection;
import java.io.FileInputStream;
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;
public class ConnectionClass {
    private final static String dbFilePath = "/path/to/config";
    // The path should start from the project's root directory
    // Relative to the project's root directory (src/main/resources/config)
        public static Connection getConnection() {
            Connection connection = null;
            try(FileInputStream file= new FileInputStream(dbFilePath)){
                Properties properties = new Properties();
                properties.load(file);

                String url= properties.getProperty("url");
                String username= properties.getProperty("username");
                String password= properties.getProperty("password");

                connection =
DriverManager.getConnection(url,username,password);
                System.out.println("Connection Established");
            }catch (IOException e ){
                System.out.println(e.getMessage());
            }catch (SQLException e){
                System.out.println(e.getMessage());

            return connection;
        }
```

`private final String dbFilePath = "/path/to/config";` this the path shown above it is set to final so no one can temper it.

`public Connection getConnection(){...}` This method is responsible for establishing a connection to the database. It returns an object of type `Connection`, which is part of the Java Database Connectivity (JDBC) API.

`Connection connection = null;` a variable called `connection` is declared but not yet initialized. It will later hold the database connection object. It is declared inside the method allows it to be closed at the end of the method's lifecycle. It's common practice to close database connections when they are no longer needed to release resources.

`try(FileInputStream file= new FileInputStream(dbFilePath)){...}` the file stream has been opened inside the `try` statement because automatically closes the file stream when the block finishes, whether it finishes normally or an exception occurs. If an error acquire when opening the file the block doesn't run at all too.

`static` most of methods here are static because this is a utility class which mean we gonna use it a lot and mostly everywhere in the project so we don't have to create insistence every time we use the class and overflow the memory.

Closing Connection

```
public static void closeConnection(Connection connection){
    try{
        connection.close();
        System.out.println("Connection Closed");
    }catch (SQLException e){
        System.out.println(e.getMessage());
    }
}
```

i think this one is so simple...

Another Way to Load Configuration

```
private static String url = "", username = "", password = "";

public static void setDbInfo() {
    FileInputStream file = null;
    try {
        file = new FileInputStream(dbFilePath);
        Properties properties = new Properties();
        properties.load(file);
        url = properties.getProperty("url");
        username = properties.getProperty("username");
        password = properties.getProperty("password");
    } catch (IOException e) {
        System.out.println(e.getMessage());
    } finally {
        try {
            file.close();
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

- This approach is better because it **saves resources** by reading the configuration file only once, which can be done in a constructor or as a `static` method. This avoids repeatedly reading the file every time `getConnection()` is called, and the loaded credentials can be reused.
- By separating the configuration reading into its own method, it **enhances maintainability** if the reading process needs to change, you can simply modify that method without having to update the logic throughout the entire project.
- It allows handling file-related errors separately from database connection errors, providing better clarity and control over error management.

Here is an example on how to use it:

```java
private final static String dbFilePath = "/path/to/config";

public static Connection getConnection() {
    Connection connection = null;
    try {
        setDbInfo();
        connection = DriverManager.getConnection(url, username, password);
        System.out.println("Connection Established");
    } catch (SQLException e) {
        e.getMessage();
    }
    return connection;
}
```

Why Use Environment Variables?

Environment variables are dynamic values stored outside your code (in the device itself) that your application can access at runtime. They're typically used to configure system settings and store sensitive information like API keys, database URLs, and credentials, without hardcoding them into your source files. These variables are saved inside shared system libraries or the operating system’s environment and can be accessed by other users or processes on the system provided they have the necessary permissions to view or modify them. This centralized storage allows controlled sharing of configuration data across multiple applications or users while maintaining security through permission management.

While using a config file is acceptable during development, in real-world production environments it’s **best practice** to use environment variables for storing sensitive data such as database credentials. Environment variables are stored within the system or server itself and can only be accessed by authorized users, whereas config files risk being accidentally exposed, leaked, or pushed to public repositories or insecure locations.

Simple Example in Java:

Instead of reading from a config file or hardcoding values:

```java
String url = System.getenv("DB_URL");
String username = System.getenv("DB_USERNAME");
String password = System.getenv("DB_PASSWORD");
```

Setting Environment Variables:

On Linux/macOS (bash):

```bash
export DB_URL="jdbc:mysql://localhost:3306/testing"
export DB_USERNAME="root"
export DB_PASSWORD="your_password_here"
```

On Windows (Command Prompt):

```
set DB_URL=jdbc:mysql://localhost:3306/testing
set DB_USERNAME=root
set DB_PASSWORD=your_password_here
```

Using environment variables offers several benefits. They enhance security by keeping passwords and secrets out of source code and version control; improve portability by allowing different configurations across environments (such as development, staging, and production) without changing the code; and promote separation of concerns by keeping configuration independent from application logic, making the code cleaner and easier to maintain. Overall, leveraging environment variables makes your application more secure, flexible, and environment-agnostic, which is essential for modern software development and deployment.

Method Overloading

```
public static void closeConnection(Connection connection) {
    try {
        if (connection != null)
            connection.close();
        System.out.println("Connection Closed");
    } catch (SQLException e) {
        System.out.println(e.getMessage());
    }
}

public static void closeConnection(Connection connection, PreparedStatement
preparedStatement) {
    try {
        if (connection != null)
            connection.close();
        if (preparedStatement != null) ;
            preparedStatement.close();
        System.out.println("Connection Closed");
    } catch (SQLException e) {
        System.out.println(e.getMessage());
    }
}

public static void closeConnection(Connection connection, PreparedStatement
preparedStatement, ResultSet resultSet) {
    try {
        if (connection != null)
            connection.close();
        if (preparedStatement != null) ;
            preparedStatement.close();
        if (resultSet != null)
            resultSet.close();
        System.out.println("Connection Closed");
    } catch (SQLException e) {
        System.out.println(e.getMessage());
    }
}
```

There are 3 methods with the same name but different parameters, this is not a syntax error it is called `Method Overloading`.

Method overloading is a common programming concept supported by many object-oriented languages like Java, C++, and C#. It allows multiple functions or methods to share the same name as long as they differ in the number or type of their parameters. This makes code more readable and flexible, as it lets developers use the same method name to perform similar tasks with different types or amounts of input. Method overloading is part of polymorphism and helps simplify code by avoiding the need for many uniquely named methods that do nearly the same thing.

The three functions provide methods to close different combinations of database resources (`Connection`, `PreparedStatement`, and `ResultSet`). They differ based on the types and number of parameters, offering flexibility depending on which resources are in use. This is a recommended practice to ensure that resources are properly released and not left open, which can lead to memory leaks or connection issues. When you call one of these methods, the program determines which one to execute based on the method name and the matching parameter types.

Differences:

- The first method only closes the `Connection`.
- The second method closes both the `Connection` and `PreparedStatement`, making it suitable for queries that don't return a `ResultSet`.
- The third method closes all three resources: `Connection`, `PreparedStatement`, and `ResultSet`, making it ideal for queries that return results.

CRUD

`CRUD` : (create, read, update, delete) are the operation you can perform on a database and are essential in any DBMS.

```java
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

class Main {
    public static void main(String[] args) {
        Connection conn = ConnectionClass.getConnection();
        PreparedStatement ps = null;
        try {
            ps = conn.prepareStatement("insert into Student values
(null,'TEST','2000-5-5',56)");
            ps.executeUpdate();
        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
        ConnectionClass.closeConnection(conn);
    }
}
```

`PreparedStatement` object in Java allows you to execute parameterized SQL queries securely.

`ps = conn.prepareStatement("...")` represents a precompiled SQL statement that can be executed multiple times with different parameters, the sql query differ depends on your needs, don't be shy customize it as much as you want.

`ps.executeUpdate()` execute the query. It is used because `INSERT`, `UPDATE`, and `DELETE` statements do not return a `ResultSet`. Instead, they return an integer indicating the number of rows affected.

- if you use `ps.executeUpdate()` twice in row it will execute the this query twice, so best practice to call `executeUpdate()` immediately after setting up the `PreparedStatement` and providing any necessary parameters like this:

```
ps = conn.prepareStatement("insert into Student values (null,'TEST','2000-5-
5',56)");
ps.executeUpdate();
ps = conn.prepareStatement("insert into Student values (null,'Test22','2004-
10-26',72)");
ps.executeUpdate();
```

you can query one statement at a time you can't fill it like a buffer and then execute them all at once

---

```
ps = conn.prepareStatement("insert into Student values (null,?,?,?)");
```

- `?` are **placeholders** or **parameters** used in a prepared statement.
- When you execute the prepared statement, you can set the values for these placeholders using methods like `setString()`, `setInt()`, `setDate()`, etc., depending on the data type of the corresponding column in the `Student` table.

```
ps = conn.prepareStatement("insert into Student values (null,?,?,?)");
System.out.print("Enter name: ");
String name = in.nextLine();
ps.setString(1, name);

System.out.print("Enter birthdate (YYYY-MM-DD): ");
ps.setDate(2, Date.valueOf(in.nextLine()));

System.out.print("Enter grade: ");
ps.setFloat(3, in.nextFloat());
in.nextLine();  // clear the buffer after using float

ps.executeUpdate();
```

Here each place holder treated as number, `1` the first placeholder, `2` the second placeholder.... and so on, you can pass variable or object to the set function or just the input statement inside it as appeared above.

Each value must be sent after proper validation and sensitization to insure that no errors happen during the process.

The first value is set to `null` because it is a primary key, the primary key is generated by the database. As i said before the sql query change based in your needs.

The rest of the operations are the same `update`, `delete`, `alter` ...etc all you need is to change the sql queries, but `read` has a different way to do it explained below.

Read

First of all we need one library `import java.sql.ResultSet` because as mentioned before `ps.executeUpdate()` is for `INSERT`, `UPDATE`, and `DELETE` statements and doesn't return data but rather a count of affected rows and don't return any data from the database.

```
ResultSet rs = null;
try {
    ps = conn.prepareStatement("select * from Student");
    rs = ps.executeQuery();
    while (rs.next()) {
        System.out.println(rs.getString("name") + " " +
rs.getDate("birth_date") + " " + rs.getFloat("gpa"));
    }
} catch (SQLException e) {
    System.out.println(e.getMessage());
}
```

`ResultSet rs = null;` initialize `ResultSet` object that return data from the database after using `select` or any other methods. `rs` stands for ResultSet.

`ps.executeQuery()` is the method to use for retrieving data from a `SELECT` statement, as it returns a `ResultSet` containing the data. `ps` stands for PreparedStatement.

`rs.next()` is used to iterate through the rows of the `ResultSet`. It moves the cursor to the next row and returns `true` if there is a valid row, `false` if there are no more rows. It is correctly used here for row-by-row traversal.

`rs.getDate`, `rs.getString`, `rs.getFloat` and a lot more, these methods retrieve the values of the columns from the current row in the `ResultSet`. You can retrieve data by using the column index like this `rs.getString(1)` but it is prone to errors so just the name.

Here is an example using placeholder:

```
try {
    ps = conn.prepareStatement("select birth_date from Student where id=?");
// or
// ps = conn.prepareStatement("select ? from Student where id=5");
    System.out.print("Enter id: ");
    int input_id = in.nextInt();
    in.nextLine();
    ps.setInt(1, input_id);
    rs = ps.executeQuery();
    rs.next();
    System.out.println(rs.getString("birth_date"));
} catch (SQLException e) {
    System.out.println(e.getMessage());
}
```

```
import java.sql.Statement;
.
.
.
ps = con.prepareStatement("insert into users values(null,?,?)",
Statement.RETURN_GENERATED_KEYS);
ps.setString(1, "myName");
ps.setString(2, "myEmail@gmail.com");
ps.executeUpdate();
rs = ps.getGeneratedKeys();
if (rs.next()) {
    int id = rs.getInt(1);
}
```

When the primary key is generated by the database, using `Statement.RETURN_GENERATED_KEYS` allows us to automatically retrieve it right after an `INSERT` operation. This is crucial when we need to cascade the key into related tables as a foreign key, ensuring data integrity and saving the extra step of querying the database to manually retrieve the generated key.

`rs = ps.getGeneratedKeys();` this retrieve the row of the generated key not the key itself.

`if (rs.next()) {int id = rs.getInt(1);}` this ensure that the row exist and now error acquire before getting the key.

Rollback

If you insert a user into the `users` table but, for some reason, the password doesn't get saved in the `passwords` table, you'll end up with a user without a password, which could lead to data inconsistency or corruption. To prevent this, you can use a **transaction** and `rollback` the entire operation if any part fails, ensuring that both the user and the password are saved together or not at all.

This approach is especially useful when dealing with `Many-to-Many`, `One-to-Many`, and `One-to-One` relationships in the database.

It's important to note that transactions are not limited to just `INSERT` operations; they can also apply to `UPDATE`, `DELETE`, or any other database modification. Using `rollback` in these cases ensures data integrity and helps avoid issues down the line.

In simple terms, `rollback` undoes any changes that were made during the transaction if something goes wrong.

```
try {
    conn.setAutoCommit(false);
    ps = conn.prepareStatement("insert into Student values
(null,'test','2000-2-2',95)", Statement.RETURN_GENERATED_KEYS);
    ps.executeUpdate();
    rs = ps.getGeneratedKeys();
    ps = conn.prepareStatement("insert into passwords values
(null,?,'error_test')");
    ps.setInt(1, 4545); // key that doesn't exist
    ps.executeUpdate();
    conn.commit()
} catch (SQLException e1) {
    System.out.println("SQLException: " + e1.getMessage());
    try {
        conn.rollback();
    } catch (SQLException e2) {
        System.out.println("SQLException: " + e2.getMessage());
        System.out.println("Rollback"); // to identify the error
    }
}
```

When `conn.setAutoCommit(false)` is set to false, queries are executed but not saved immediately. Instead, they are held in a transaction until you explicitly call `conn.commit();`, which saves all changes made during the transaction, or `conn.rollback();`, which undoes those changes if an error occurs.

`conn.rollback();` is just ctrl+z sql version.

```
rs.next();
int id = rs.getInt(1);
ps = conn.prepareStatement("insert into passwords values
(null,?,'error_test')");
ps.setInt(1, id);
```

here is how to use it always take care.

Multi Layer Application

MVC (Model-View-Controller): The goal of this pattern is to split large web application into specific sections that all have its own purpose.

- `Controller`: The server routes all client requests to a specific controller, which acts as the middleman between the **Model** and **View**. It handles the request logic but contains minimal code, delegating tasks to other parts of the system.

- `Model`: Upon receiving a request, the controller consults the model, which manages the data logic. The model interacts with the database—validating, saving, updating, or deleting data. The controller never handles data logic directly but relies on the model to do so. The model focuses solely on data management and does not handle request outcomes.
- `View`: After the model responds, the controller uses the view to dynamically render the data into HTML. The view's role is purely to format data, without concern for how it will ultimately be presented. Once ready, the view passes the formatted data back to the controller for delivery to the client.

`Note`: The **View** and **Model** never interact with each other; all data handling is based on the controller's communication.

The point of the MVC is to split the programmer into sections to improve readability, better performance, enhance security, and a lot more

---

Here is an example (just an example for beginner):

```
src/
└── main/
    └── java/
        ├── controller/
        │   └── controller.java
        │
        ├── Model/
        │   ├── DTO/
        │   │   ├── UserReqDTO.java
        │   │   └── UserResDTO.java
        │   │
        │   ├── Entities/
        │   │   ├── Password.java
        │   │   └── User.java
        │   │
        │   ├── Mapper/
        │   │   └── DtoToUser.java
        │   │
        │   └── Repository/
        │       ├── PasswordRep.java
        │       └── UserRep.java
        │
        ├── Service/
        │   └── UserService.java
        │
        └── Utilities/
            └── ConnectionClass.java
```

Model

It contains four major packages `DTO` (Data Transfer Object), `Entities`, `Mapper`, and `Repository`. (google them)

Data Transfer Object (DTO)

A **Data Transfer Object (DTO)** is a lightweight container designed specifically to move only the information needed for a particular operation between different parts of an application. Instead of passing a full domain model (Enitity) with all its fields and relations. You define a DTO with just the properties required by the receiving component. This targeted approach keeps your data payloads small and focused, which in turn helps your application run faster and with less network overhead.

Using DTOs also enhances security by preventing unintentional exposure of sensitive or internal data. When you send only the attributes that an API endpoint or UI layer actually needs, you eliminate the risk of leaking private fields. In practice, you'll create a DTO class (for example, `UserProfileDto`) that includes only the necessary fields like `firstName`, `lastName`, and `email` and map your full entity or model to this DTO before returning it from your service or controller.

DTOs serve as the standard format for both sending and receiving data across application layers or network boundaries, ensuring that each component exchanges only the information it needs.

Example:

```java
package Model.DTO;
import java.sql.Date;
public class UserReqDTO { // Request
    private String name;
    private String email;
    private Date birthday;
    private String password;
}
```

```java
package Model.DTO;
import java.sql.Date;
public class UserResDTO { // Response
    private String name;
    private String email;
    private Date birthday;
}
```

Entities

An **Entity** is a full-fledged object in your application that represents a real-world concept like a `User`, `Product`, or `Order` and usually maps directly to a database table. Unlike a DTO, an entity includes all of its properties, relationships, and any business rules or behaviors you've defined. You use entities to load data from the database, work with it in your service layer, and then save any changes back to the store. By keeping the complete state and logic together, entities ensure that your core domain concepts stay consistent and accurate when sending to or receiving from your database.

```java
package Model.Entities;
public class Password {
    private String password;
    private Integer passwordID;
    private Integer userID;
}
```

```java
package Model.Entities;

import java.sql.Date;

public class User {
    private Long id;
    private String name;
    private String email;
    private String phoneNumber;
    private String address;
    private Date birthday;
    private Date createdAt;
    private boolean isActive;
    private Password password;
}
```

Mapper

A **Mapper** is a simple utility class that converts between your DTOs and your domain entities. In this example, the `DtoToUser` mapper takes a `UserReqDTO` which carries just the data you need for a request and builds a full `User` entity by copying over the name and email, then delegates to a helper method that creates a `Password` entity from the DTO's password field. This keeps the conversion logic in one place, so your service layer can work directly with fully populated entities while keeping the data-transfer concerns cleanly separated.

```java
package Model.Mapper;

import Model.DTO.UserReqDTO;
import Model.Entities.Password;
import Model.Entities.User;

public class DtoToUser {
    public User DtoReqToUser(UserReqDTO request) {
        User user = new User();
        user.setName(request.getName());
        user.setEmail(request.getEmail());
        user.setPassword(DtoReqToPassword(request));
        return user;
    }
    public Password DtoReqToPassword(UserReqDTO request) {
        Password password = new Password();
        password.setPassword(request.getPassword());
        return password;
    }
}
```

Repository

A **Repository** is a layer in your application responsible for handling communication with the database. It contains SQL queries that are used to perform CRUD operations like inserting, retrieving, or checking data—without exposing the details of how the data is stored or accessed. By centralizing database queries in a repository, your application becomes more organized, testable, and easier to maintain.

Repositories act as a bridge between the application and the database, keeping your business logic (like services or controllers) clean and focused on what needs to happen, not how it's done at the data level.

```
package Model.Repository;
public class PasswordRep {
    public static final String InsertPassword = "insert into passwords
values(null,?,?)";
}
```

```
package Model.Repository;

public class UserRep {
    public static final String InsertUser = "insert into users
values(null,?,?,?)";
    public static final String GetAllUsers = "select * from users";
    public static final String GetOneUser = "select * from users where id =
?";
    public static final String CheckEmail = "SELECT TRUE WHERE EXISTS
(SELECT email FROM users WHERE email = ?)";
}
```

In the example above:

- `UserRep` stores SQL statements for operations on the `users` table such as inserting a new user, retrieving all users, fetching a specific user by ID, and checking if an email already exists.
- `PasswordRep` handles insert operations for the `passwords` table.

> It’s recommended to declare these query strings as `public static final` fields so they remain constant and cannot be modified at runtime.

Later, we’ll introduce **Hibernate**, an ORM (Object-Relational Mapper) that sits beneath Spring and automatically translates your entity operations into SQL. By configuring a `LocalSessionFactoryBean` (or a JPA `EntityManagerFactory`), Hibernate inspects your mapped `User`, `Password`, and other entity classes and handles all queries for you. No more handwritten SQL strings. Spring Framework 6.0 requires Hibernate ORM 5.5+ (we recommend 5.6), and Hibernate 6.x can be used as a JPA provider in new projects.

Once Hibernate is set up, you can replace custom repositories with simple interfaces, for example:

```
package Model.Repository;

import org.springframework.data.jpa.repository.JpaRepository;
import Model.Entities.User;

public interface UserRepository extends JpaRepository<User, Long> {
    // JPQL query to find a user by email
    @Query("SELECT u FROM User u WHERE u.email = :email")
    User findByEmail(@Param("email") String email);

    // Native SQL query example
    @Query(
      value = "SELECT * FROM users WHERE name LIKE %:name%",
      nativeQuery = true
    )
    List<User> searchByName(@Param("name") String name);
}
```

Behind the scenes, Hibernate inspects your `User` entity, generates the necessary SQL for inserts, updates, deletes, and queries, and ensures your database schema and Java model stay in sync letting you focus on business logic instead of SQL maintenance.

Parameterized Queries

Using parameterized queries is essential for protecting your application against SQL injection attacks. When you build SQL statements by concatenating user input directly into the query string, malicious actors can craft inputs that alter the intended command potentially exposing or corrupting your data. In contrast, parameterized queries send the SQL text and the user-supplied values separately to the database. The database driver then safely treats those values as pure data, never executing them as part of the command itself. This separation effectively neutralizes attempts to inject harmful SQL code.

Examples:

```
public static final String requestOrder = "insert into orders
values(null,?,?,?)";
```

```
public static final String checkQuantity = "select quantity from
market_storage where product_id=? and market_id = ?";
```

Beyond security, parameterized queries offer significant performance benefits. Most modern database engines cache execution plans for prepared statements: the first time a parameterized query runs, the database parses, optimizes, and plans it, then reuses that plan for subsequent executions with different parameter values. This reuse reduces CPU overhead and improves throughput, especially in high-traffic applications that execute the same query patterns repeatedly. In contrast, dynamically constructed SQL strings typically incur the cost of reparsing and replanning on every execution.

Finally, parameterized queries lead to cleaner, more maintainable code. By keeping the SQL text constant and using placeholders (such as `?` in JDBC or `:email` in JPQL), you avoid cumbersome string-building logic, manual escaping, and ad-hoc formatting. The database driver handles quoting, escaping, and type conversion for you, reducing the likelihood of subtle bugs related to date formats, numeric precision, or special characters. Overall, parameterized queries improve your application's security, performance, and readability.

Service

```
package Service;

import Model.DTO.*;
import Model.Entities.User;
import Model.Mapper.DtoToUser;
import Model.Repository.PasswordRep;
import Model.Repository.UserRep;
import Utilities.ConnectionClass;

import java.sql.Connection;
import java.sql.Statement;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class UserService {

    public void insertUser(UserReqDTO UserDTO) {
        Connection conn = ConnectionClass.getConnection();
        PreparedStatement ps = null;
        ResultSet rs = null;
        DtoToUser dtoToUser = new DtoToUser();
        User user = dtoToUser.DtoReqToUser(UserDTO);
        try {
            conn.setAutoCommit(false);
            ps = conn.prepareStatement(UserRep.InsertUser,
Statement.RETURN_GENERATED_KEYS);
            ps.setString(1, UserDTO.getName());
            ps.setDate(2, UserDTO.getBirthday());
            if (UserService.Checkemail(UserDTO.getEmail())) {
                System.out.println("This email is already used");
                ConnectionClass.closeConnection(conn, ps, rs);
                return;
            } else {
                ps.setString(3, UserDTO.getEmail());
            }
            ps.executeUpdate();
            rs = ps.getGeneratedKeys();
            rs.next();
            ps = conn.prepareStatement(PasswordRep.InsertPassword);
            ps.setInt(1, rs.getInt(1));
            ps.setString(2, UserDTO.getPassword());
            ps.executeUpdate();
            conn.commit();
        } catch (SQLException e1) {
            System.out.println("SQLException: " + e1.getMessage());
            try {
                System.out.println("Data did not save.");
                conn.rollback();
            } catch (SQLException e2) {
                System.out.println("SQLException: " + e2.getMessage());
                System.out.println("Error while rollbacking"); // to
identify the error
            }
        }
        ConnectionClass.closeConnection(conn, ps, rs);
    }

    public static void GetAllUsers() {
        Connection conn = ConnectionClass.getConnection();
        PreparedStatement ps = null;
        ResultSet rs = null;
        try {
            ps = conn.prepareStatement(UserRep.GetAllUsers);
            rs = ps.executeQuery();
            while (rs.next()) {
                System.out.println("ID: " + rs.getInt(1) + " | Name: " +
rs.getString(2) + " | Birth Day: " + rs.getDate(3) + " | Email: " +
rs.getString(4));
            }
        } catch (SQLException e) {
            System.out.println("SQLException: " + e.getMessage());
        }
        ConnectionClass.closeConnection(conn, ps, rs);
    }

    public static void GetUserByID(int id) {
        Connection conn = ConnectionClass.getConnection();
        PreparedStatement ps = null;
        ResultSet rs = null;
        try {
            ps = conn.prepareStatement(UserRep.GetOneUser);
            ps.setInt(1, id);
            rs = ps.executeQuery();
            rs.next();
            System.out.println("Name: " + rs.getString("name") + " | Birth
Day: " + rs.getDate("birthday") + " | Email: " + rs.getString("email"));

        } catch (SQLException e) {
            System.out.println("SQLException: " + e.getMessage());
        }
        ConnectionClass.closeConnection(conn, ps, rs);
    }

    public static boolean Checkemail(String email) {
        Connection conn = ConnectionClass.getConnection();
        PreparedStatement ps = null;
        try {
            ps = conn.prepareStatement(UserRep.CheckEmail);
            ps.setString(1, email);
            boolean ans = ps.executeQuery().next();
            if (ans) {
                ConnectionClass.closeConnection(conn, ps);
                return true;
            }
        } catch (SQLException e) {
            System.out.println("SQLException: " + e.getMessage());
        }
        ConnectionClass.closeConnection(conn, ps);
        return false;
    }
}
```

Connection

This `ConnectionClass` is a utility for managing JDBC connections in a consistent, configurable way. Below is a breakdown of its responsibilities and some key points:

- Configuration Loading
  - Reads database connection parameters (`url`, `username`, `password`) from an external properties file whose path is held in `dbFilePath`.
  - Encapsulates file I/O and `Properties` parsing in the private helper `setDbInfo()`.
- Connection Acquisition

- Exposes a single public method, `getConnection()`, which:
    1. Loads (or reloads) the latest DB credentials via `setDbInfo()`.
    2. Calls `DriverManager.getConnection(...)` to open a new `Connection`.
  - Returns `null` if a `SQLException` occurs (you may choose to rethrow or wrap exceptions for stronger error handling).
- Resource Cleanup
  - Provides three overloads of `closeConnection(...)` to safely close:
    1. `Connection` only
    2. `Connection` + `PreparedStatement`
    3. `Connection` + `PreparedStatement` + `ResultSet`
  - Each method checks for `null` and suppresses (logs) any `SQLException` without throwing.

```java
package Utilities;

import java.io.FileInputStream;
import java.io.IOException;
import java.sql.*;
import java.util.Properties;

public class ConnectionClass {
    private final static String dbFilePath = "/path/to/config";

    public static Connection getConnection() {
        Connection connection = null;
        try {
            setDbInfo();
            connection = DriverManager.getConnection(url, username,
password);
//              System.out.println("Connection Established");
        } catch (SQLException e) {
            e.getMessage();
        }
        return connection;
    }

    private static String url = "", username = "", password = "";

    public static void setDbInfo() {
        FileInputStream file = null;
        try {
            file = new FileInputStream(dbFilePath);
            Properties properties = new Properties();
            properties.load(file);
            url = properties.getProperty("url");
            username = properties.getProperty("username");
            password = properties.getProperty("password");
        } catch (IOException e) {
            System.out.println(e.getMessage());
        } finally {
            try {
                file.close();
            } catch (IOException e) {
                System.out.println(e.getMessage());
            }
        }
    }

    public static void closeConnection(Connection connection) {
        try {
            if (connection != null)
                connection.close();
//              System.out.println("Connection Closed");
        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
    }

    public static void closeConnection(Connection connection,
PreparedStatement preparedStatement) {
        try {
            if (connection != null) {
                connection.close();
            }
            if (preparedStatement != null) {
                preparedStatement.close();
            }
//              System.out.println("Connection Closed");
        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
    }


    public static void closeConnection(Connection connection,
PreparedStatement preparedStatement, ResultSet resultSet) {
        try {
            if (connection != null) {
                connection.close();
            }
            if (preparedStatement != null) {
                preparedStatement.close();
            }
            if (resultSet != null) {
                resultSet.close();
            }
//              System.out.println("Connection Closed");
        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

By centralizing connection setup and teardown in one place, this utility makes your data-access code cleaner, easier to maintain, and less error-prone, also providing:

- Externalized Configuration
  - Database credentials live outside the code, in a `.properties` file.
  - Changing environments (dev/test/prod) only requires updating the config file path or contents, not recompiling.
- Single Responsibility
  - Separation of concerns: one class strictly handles connection logic and cleanup, keeping your DAOs/services free of boilerplate.
- Static Utility Methods
  - All methods and fields are `static`, making it easy to call without instantiating the class.
  - Thread-safe for read-only operations (properties file is reloaded on every connection request, avoiding stale credentials).
- Graceful Cleanup
  - Overloaded `closeConnection(...)` methods handle all common JDBC resources in one place.
  - Null-checks prevent `NullPointerException`, and exceptions during close are caught and logged.
- Error Handling Considerations
  - Currently, SQL and IO exceptions are caught and printed; for production you might:

  1. Rethrow as a custom unchecked exception
  2. Use a logging framework instead of `System.out.println` for better visibility
  3. Fail fast on configuration errors to avoid silent connection failures
- Extensibility
  - You can extend this pattern to include connection pooling (e.g., using HikariCP) by replacing the `DriverManager` call with a `DataSource` lookup.
  - You could also add methods to retrieve a pooled `DataSource` directly for better performance in high-load environments.

Controller

Now we are cooking....

This file defines the main **API endpoints** related to employee operations in the application. It is part of the `Controller` layer in the MVC architecture, responsible for handling incoming HTTP requests and delegating the logic to the `EmployeeService` class.

An **API endpoint** refers to a specific URL or path where the backend application listens for requests from clients. It acts as a digital interface that allows external systems, such as front-end applications or other services, to interact with backend functionality.

The controller is mapped to the base path `/employee` using `@RequestMapping`, and it allows cross-origin requests (`@CrossOrigin`), enabling access from different domains—particularly useful in frontend-backend integrations during development or deployment.

This means you can access these resources by making HTTP requests to URLs such as:

```
http://myDomain.local/market/employee
```

Each defined method in this controller corresponds to a specific action, such as saving a new employee, logging in, or retrieving employee data. These endpoints provide a clean and organized way to interact with employee-related features in the application.

```java
package Controller;

import Model.DTO.UserDto.EmployeeReqDTO;
import Model.DTO.UserDto.forgetpassword;
import Model.DTO.UserDto.LoginDto;
import Service.EmployeeService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping(path = "/employee")
@CrossOrigin()
public class EmployeeController {
    @Autowired
    private EmployeeService employeeService;

    @PostMapping(path = "/save-emp")
    public Object saveEmployee(@RequestBody EmployeeReqDTO employeeReqDTO) {
        return this.employeeService.saveEmployee(employeeReqDTO);
    }

    @PostMapping(path = "/login")
    public Object login(@RequestBody LoginDto loginDto) {
        return this.employeeService.empLogin(loginDto);
    }

    @PostMapping(path = "/Forgot-Password")
    public Object forgotPassword(@RequestBody forgetpassword forGetPassword)
{
        return this.employeeService.forgetPassword(forGetPassword);
    }

    @GetMapping(path = "/get-all-employees")
    public Object getAllEmployees() {
        return this.employeeService.getAllEmployee();
    }

    @GetMapping(path = "/unassigned-employees")
    public Object getUnassignedEmployees() {
        return this.employeeService.getUnassignedEmployees();
    }

}
```

1. `@RestController`: Marks the class as a RESTful web service controller. It combines `@Controller` and `@ResponseBody`, meaning that each method returns a JSON or XML response directly. (More details on REST APIs will be explained later.)
2. `@RequestMapping(path = "/employee")`: Defines the base URL path for all endpoints in this controller. All routes will be prefixed with `/employee`.
3. `@CrossOrigin()`: Enables CORS (Cross-Origin Resource Sharing), allowing resources to be requested from different origins (domains, ports, or protocols). This is useful for allowing frontend apps hosted elsewhere to interact with the API. (This concept will be covered in more detail later.)
4. `@Autowired`: Enables **dependency injection** Spring automatically injects the required instance of `EmployeeService` into this controller, promoting loose coupling and better test-ability.
   - **loose coupling**: design principle where software components have minimal dependencies on each other.

dependency injection

Dependency Injection is a design pattern where an object's dependencies are provided from the outside, rather than the object creating them itself. In other words, dependencies are "injected" into the object. This allows a class to depend on an abstraction or interface rather than a specific implementation, reducing tight coupling.

If you need to change the entire functionality of a dependency, you only need to change the declaration or configuration where the dependency is provided, instead of rewriting the whole class repeatedly.

For example, if a class needs to use a `NotificationService`, you don't create it directly using `new NotificationService()` inside the class. Instead, you inject i. Typically through the constructor or by using annotations like `@Autowired` in Spring.

```
public class UserManager {
    private final NotificationService service;

    public UserManager(NotificationService service) {
        this.service = service;
    }
}
```

Now you can pass in any implementation of `NotificationService`, such as `EmailNotificationService` or `SMSNotificationService`, without modifying the

`UserManager` logic. This pattern improves flexibility, makes testing easier (using mock objects), and aligns with the principles of clean architecture and separation of concerns.

GET Request

A **GET** request is used to retrieve data from a server without sending any data in the request body. It is typically used to request static content like web pages or resources where no user input or credentials are required. Because GET requests do not have a body, they are simple and fast, making them ideal for fetching data or loading pages.

The `@GetMapping` annotation is used to indicate that the method handles HTTP **GET** requests for the specified path. It directs the application to process incoming GET requests and respond accordingly using the associated service logic.

POST Request

A **POST** request is used to send data to the server within the request body. This allows more flexibility, enabling the client to submit information such as login credentials, form data, or any other type of input. POST requests are commonly used on webpages that require user interaction or data submission, such as login forms, registration pages, and data processing APIs.

Similarly, the `@PostMapping` annotation is used to indicate that the method handles HTTP **POST** requests for the given path. It processes incoming POST requests, typically carrying data in the request body, and delegates the handling to the corresponding service.

Endpoints:

1. **POST** `/employee/save-emp`
   - **Purpose:** Create a new employee record.
   - **Request Body:** `EmployeeReqDTO`
   - **Response:** Result of employee creation handled by the service layer.
2. **POST** `/employee/login`
   - **Purpose:** Authenticate an employee using login credentials.
   - **Request Body:** `LoginDto` (includes email/username and password).
   - **Response:** Success or failure response from the login process.
3. **POST** `/employee/Forgot-Password`
   - **Purpose:** Trigger the password recovery process for an employee who forgot their password.
   - **Request Body:** `forgetpassword` (likely includes email or phone number).
   - **Response:** A message or status indicating password reset instructions.
4. **GET** `/employee/get-all-employees`
   - **Purpose:** Retrieve all employees stored in the system.
   - **Response:** A list of all employee records.
5. **GET** `/employee/unassigned-employees`

- **Purpose:** Retrieve employees who are not currently assigned to any task, project, or department.
- **Response:** A filtered list of unassigned employee records.

SOP & CROS

1. Same-Origin Policy (SOP)

The **Same-Origin Policy** (**SOP**) is a fundamental security feature in web browsers that prevents malicious websites from accessing sensitive data from another site. It ensures that web pages can only make requests to the **same origin**, where the origin consists of:

1. **Protocol (e.g.,** `http`, `https`)
2. **Domain (e.g.,** `example.com`)
3. **Port (e.g.,** `80`, `443`)

If any of these differ, the browser considers the request **cross-origin** and enforces SOP restrictions.

1. Protocol Check

- A website running on `https://example.com` tries to request data from different protocol variations.
- If you are logged into `https://bank.com`, a malicious site (`http://evil.com`) should not be able to send unauthorized requests to your bank.

2. Domain Check

- A website running on `https://book.com` tries to request data from another domain.
- If you are logged into `https://email.com`, another site (`https://hacker.com`) should not be able to fetch your private emails from `email.com/api/inbox`.

3. Port Check

- A website running on `https://example.com:443` tries to request data from a different port.

Some web applications run different services on different ports. If SOP didn't exist, a malicious script could interact with services running on internal ports, potentially causing security issues.

Compared URL	Outcome	Reason
`http://www.example.com/dir/page2.html`	Success	Same protocol, host, and port
`http://www.example.com/dir2/other.html`	Success	Same protocol, host, and port
`http://username:password@www.example.com/dir2/other.html`	Success	Same protocol, host, and port
`http://www.example.com:81/dir/other.html`	Failure	Same protocol and host, but different port
`https://www.example.com/dir/other.html`	Failure	Different protocol

Compared URL	Outcome	Reason
`http://en.example.com/dir/other.html`	Failure	Different host (subdomain mismatch: `en.example.com` vs. `www.example.com`)
`http://example.com/dir/other.html`	Failure	Different host (exact match required: `example.com` vs. `www.example.com`)
`http://v2.www.example.com/dir/other.html`	Failure	Different host (subdomain mismatch: `v2.www.example.com` vs. `www.example.com`)
`http://www.example.com:80/dir/other.html`	Depends	Port explicitly stated; behavior depends on browser implementation

Without SOP, a malicious website could:

- Steal your session tokens by making unauthorized API calls.
- Read private data from another site while you're logged in.
- Interact with sensitive web applications (e.g., banking, emails) without your consent.

2. Cross-Origin Resource Sharing (CORS)

Cross-Origin Resource Sharing (CORS) is an extension of the browser's same-origin policy that allows servers to specify which external origins are permitted to access their resources. By default, web pages may only make requests to the exact same scheme, host, and port from which they were served. CORS introduces a controlled way for a server to relax this policy, enabling secure cross-domain communication.

CORS is necessary when you want to allow authorized sharing of resources between different origins. For example, if your web application running at `https://example.com` needs to request data from an API hosted at `https://api.anotherdomain.com`, the browser will block this request by default because the origins differ. Enabling CORS on the API server tells the browser that this cross-origin request is allowed.

This is how CORS Works

here a link that has Good explanation for http headers that i recommend you to read.

1. **Simple Requests**
   For safe HTTP methods (GET, POST, HEAD) with only "simple" headers (`Accept`, `Content-Type: text/plain|multipart/form-data|application/x-www-form-urlencoded`, etc.), the browser attaches an `Origin` header and sends the request directly.

- **Request header:**

```
Origin: https://frontend.example.com
```

- **Response headers (must include):**

```
Access-Control-Allow-Origin: https://frontend.example.com
Access-Control-Expose-Headers: X-My-Custom-Header
```

2. **Preflighted Requests**
   For non-simple methods (PUT, DELETE, PATCH) or custom headers, the browser sends an **OPTIONS** preflight request first to verify permissions.
   - **Preflight request:**

```
OPTIONS /api/resource HTTP/1.1
Origin: https://frontend.example.com
Access-Control-Request-Method: DELETE
Access-Control-Request-Headers: X-Custom-Header
```

   - **Preflight response (must include):**

```
Access-Control-Allow-Origin: https://frontend.example.com
Access-Control-Allow-Methods: GET, POST, DELETE
Access-Control-Allow-Headers: X-Custom-Header, Content-Type
Access-Control-Max-Age: 3600
```

CORS-Related Headers

- **Access-Control-Allow-Origin**
  Specifies which origin(s) may access the resource. Can be a specific origin or `*` (not allowed when using credentials).
- **Access-Control-Allow-Methods**
  Lists HTTP methods permitted for cross-origin requests.
- **Access-Control-Allow-Headers**
  Lists non-simple request headers clients are allowed to send.
- **Access-Control-Expose-Headers**
  Makes certain response headers available to JavaScript (beyond CORS-safelisted).

- **Access-Control-Allow-Credentials**
  `true` if the response to the request can be exposed when the credentials flag is true (cookies, HTTP authentication).
- **Access-Control-Max-Age**
  Indicates how long the results of a preflight request can be cached.

Enabling CORS in Spring Boot

```java
// Controller-level configuration
@RestController
@RequestMapping("/api")
@CrossOrigin(
    origins = "https://frontend.example.com",
    methods = { RequestMethod.GET, RequestMethod.POST, RequestMethod.DELETE
},
    allowedHeaders = { "Content-Type", "X-Custom-Header" },
    allowCredentials = "true",
    maxAge = 3600
)
public class ResourceController {
    @GetMapping("/resource")
    public Resource getResource() { /* ... */ }

    @DeleteMapping("/resource/{id}")
    public void deleteResource(@PathVariable Long id) { /* ... */ }
}
```

Here's what these policies do:

> remember, this is just an example there are many possible headers and policies, and they should be customized based on the specific needs and structure of the website or API you're developing

- `origins = "https://frontend.example.com"`
  Specifies the only allowed origin for incoming cross-origin requests. In this case, only web pages served from `https://frontend.example.com` may call these endpoints. All other domains will be rejected by the browser.
- `methods = { RequestMethod.GET, RequestMethod.POST, RequestMethod.DELETE }`
  Restricts the HTTP methods that the frontend may use. Here, only GET, POST, and DELETE requests will be permitted; any attempts with PUT, PATCH, etc., will be blocked as not allowed.
- `allowedHeaders = { "Content-Type", "X-Custom-Header" }`
  Lists which request headers the browser is allowed to send. The frontend can include `Content-Type` (e.g. `application/json`) or `X-Custom-Header` in its requests; any other custom headers will be stripped or cause the preflight to fail.

- `allowCredentials = "true"`
  Allows the browser to include credentials (cookies, HTTP authentication tokens, or client-side SSL certificates) in cross-origin requests. When set to `true`, the server will send back `Access-Control-Allow-Credentials: true` and the browser will honor cookies or HTTP auth.
- `maxAge = 3600`
  Determines how long (in seconds) the results of a preflight (OPTIONS) request can be cached by the browser. Here, once a valid preflight succeeds, the browser will reuse that approval for the next hour (3,600 seconds) without sending another OPTIONS request.

It is considered best practice to ensure a secure and efficient CORS configuration, always restrict `Access-Control-Allow-Origin` to specific trusted domains rather than using `*`, limit `Access-Control-Allow-Methods` and `Access-Control-Allow-Headers` to only what your application actually needs, and set a reasonable `Access-Control-Max-Age` to cache preflight responses and reduce request overhead. Never combine `Access-Control-Allow-Origin: *` with `Access-Control-Allow-Credentials: true`; enforce HTTPS for all cross-origin traffic to guard against interception; log and monitor both successful and failed CORS requests to detect misconfigurations or abuse; and expose only the minimum necessary response headers via `Access-Control-Expose-Headers`.

SOP vs CORS

Feature	Same-Origin Policy (SOP)	Cross-Origin Resource Sharing (CORS)
Definition	A security policy that restricts web pages from making requests to a different origin.	A mechanism that allows a server to explicitly permit cross-origin requests.
Purpose	Prevents unauthorized access to resources from different origins.	Allows controlled cross-origin requests when needed.
Default Behavior	Blocks cross-origin requests by default.	Allows cross-origin requests only if explicitly permitted by the server.
How It Works	Requests can only be made to the same protocol, domain, and port.	The server includes CORS headers in the response to specify allowed origins.
Exception Handling	Typically, blocked requests result in a **CORS error** in the browser console.	Developers must configure the server to send proper CORS headers.
Common Issues	Prevents APIs from being accessed from different websites, restricting integrations.	Misconfigurations may lead to security vulnerabilities (e.g., open `Access-Control-Allow-Origin: *`).