**MANIPAL INSTITUTE OF TECHNOLOGY**
**Manipal – 576 104**

**DEPARTMENT OF COMPUTER SCIENCE & ENGG.**

MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
*(A constituent unit of MAHE, Manipal)*

## CERTIFICATE

This is to certify that Ms./Mr. ……...…… Reg. No. …..…… Section: …… Roll No .....has satisfactorily completed the lab exercises prescribed for Big Data Analytics Lab [CSF XXXX] of Second Year B. Tech. Degree at MIT, Manipal, in the academic year 2025-2026.

Date: …….....

| | |
|---|---|
| Signature | Signature |
| Faculty in Charge | Head of the Department |

## CONTENTS

**Course objectives:**

This laboratory course enable students to

- Understand usage HDFS Commands, HIVE Queries and map reduce programs
- Implement recommendation system for a real-world problem
- Understand usage of classification and clustering algorithm on Big Data.
- Understand how graph data can be analysed
- Demonstrate risk estimation using simulation

**Course outcomes:**

At the end of this course, students will gain the

- Ability to use HDFS Commands, query HIVE and develop map reduce programs
- To develop a recommendation system for a real-world problem
- To classify big data by applying algorithms based on supervised and unsupervised techniques.
- To analyse graph data using GraphX.
- To apply Monte-Carlo simulation for estimating risk.

**Evaluation plan**

- Internal Assessment Marks : 60%
  - Continuous Evaluation : 60%
    4*10=40M for (2*10=20M for observation book, + 2*10=20M for Execution and viva)
    20M for midterm evaluation

    The assessment will depend on punctuality, program execution, maintaining the observation note and answering the questions in viva voce.

- End semester assessment of 2 hours duration: 40 %

## INSTRUCTIONS TO THE STUDENTS

**Pre- Lab Session Instructions**

1. Students should carry the Lab Manual Book and the required stationery to every lab session.
2. Be in time and follow the institution dress code.
3. Must Sign in the log register provided.
4. Make sure to occupy the allotted seat and answer the attendance
5. Adhere to the rules and maintain the decorum.
6. Students must come prepared for the lab in advance.

**In- Lab Session Instructions**

- Follow the instructions on the allotted exercises.
- Show the program and results to the instructors on completion of experiments.
- On receiving approval from the instructor, copy the program and results in the Lab record
- Prescribed textbooks and class notes can be kept ready for reference if required.

**General Instructions for the exercises in Lab**

- Implement the given exercise individually and not in a group.
- Observation book should be complete with program, proper input output clearly showing the parallel execution in each process. Plagiarism (copying from others) is strictly prohibited and would invite severe penalty in evaluation.
- The exercises for each week are divided under three sets:
  - Solved example
  - Lab exercises - to be completed during lab hours
  - Additional Exercises - to be completed outside the lab or in the lab to enhance the skill
- In case a student misses a lab class, he/ she must ensure that the experiment is completed during the repetition class with the permission of the faculty concerned but credit will be given only to one day's experiment(s).
- Questions for lab tests and examination are not necessarily limited to the questions in the manual, but may involve some variations and / or combinations of the questions.

**THE STUDENTS SHOULD NOT**

- Bring mobile phones or any other electronic gadgets to the lab.
- Go out of the lab without permission.

**Lab No 1:** **Date:**

# Familiarization of Eclipse IDE and Executing Basic Spring Boot Application

## I. Eclipse IDE KEY SHORTCUTS

### General Shortcuts

- **Ctrl + Shift + R**: Open resource (search for files across the workspace).
- **Ctrl + Shift + T**: Open type (search for classes or interfaces).
- **Ctrl + Shift + F**: Auto-format code according to the defined style.
- **Ctrl + F**: Search in the current file.
- **Ctrl + D**: Delete the current line.
- **Ctrl + Z**: Undo.
- **Ctrl + Y**: Redo.
- **Ctrl + Shift + L**: Display all shortcuts.
- **Ctrl + Space**: Content Assist (code suggestions or autocompletion).

### Navigation

- **F3**: Open declaration (jump to the definition of a variable, method, etc.).
- **Ctrl + O**: Show outline (quick navigation within the current class).
- **Ctrl + E**: Switch editor (list of open editors).
- **Ctrl + Q**: Go to the last edit location.
- **Ctrl + L**: Go to a specific line.
- **Alt + ← / →**: Navigate backward/forward through navigation history.
- **Ctrl + H**: Open search dialog (for text, declarations, etc.).

### Editing

- **Ctrl + Shift + O**: Organize imports (add/remove imports automatically).
- **Alt + Shift + R**: Rename (variable, method, class, etc.).
- **Alt + Shift + L**: Extract local variable (refactor selected code into a local variable).
- **Alt + Shift + M**: Extract method (refactor selected code into a new method).
- **Ctrl + Shift + /**: Add block comment.
- **Ctrl + Shift + **: Remove block comment.
- **Ctrl + /**: Toggle single-line comment.

### Running & Debugging

- **F11**: Debug last launched program.
- **Ctrl + F11**: Run last launched program.
- **F5**: Step into (during debugging).

- **F6**: Step over (during debugging).
- **F7**: Step return (during debugging).
- **Ctrl + Shift + B**: Toggle breakpoint.

## File Management

- **Ctrl + N**: Create a new file.
- **Ctrl + Shift + S**: Save all files.
- **Ctrl + W**: Close current editor.
- **Ctrl + Shift + W**: Close all editors.

## Source Control

- **Ctrl + Shift + G**: Search for references in the workspace.
- **Ctrl + Alt + H**: Open call hierarchy (find where a method is being called).

Setting Up Spring Boot Project

### Step 1: Go to Spring Initializr

- Visit https://start.spring.io/.

### Step 2: Configure the Project

- **Project**: Select either **Maven** or **Gradle** project.
- **Language**: Choose **Java**.
- **Spring Boot Version**: Select the latest stable version of Spring Boot.
- **Group**: Your organization name or domain (e.g., `com.example`).
- **Artifact**: The name of your project (e.g., `myproject`).
- **Name**: The name of your project (e.g., `myproject`).
- **Description**: Write a short description (optional).
- **Package Name**: The root package for your project (e.g., `com.example.myproject`).
- **Packaging**: Select **Jar** (default) or **War** (if deploying to an application server like Tomcat).
- **Java Version**: Select the appropriate Java version (11 or 17 are recommended).

### Step 3: Add Dependencies

Click on the **Add Dependencies** button and select the dependencies you need for your project. Some common ones are:

- **Spring Web**: To create RESTful APIs and web applications.
- **Spring Data JPA**: For database interaction with JPA/Hibernate.
- **H2**: For an in-memory database (optional).
- **MySQL Driver**: If you're using a MySQL database.
- **Spring Boot DevTools**: For easier development with auto-reload.

- **Spring Security**: For adding security (optional).

**Step 4: Generate the Project**

- Click on **Generate** to download the project as a ZIP file.
- Extract the downloaded ZIP to a folder.

**Step 5: Import the Project into Your IDE**

- Open your preferred IDE (e.g., **IntelliJ IDEA**, **Eclipse**, or **VS Code**).
- **In IntelliJ IDEA**:
  - Go to **File > Open** and select the folder where you extracted the Spring Boot project.
- **In Eclipse**:
  - Go to **File > Import > Existing Maven Projects**.
  - Browse to the folder and import it.

**Step 6: Build and Run the Project**

- In IntelliJ IDEA or Eclipse, find the `Application.java` file under the `src/main/java` directory. It is usually named after your project name (e.g., `MyprojectApplication.java`).

```
@SpringBootApplication
public class MyprojectApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyprojectApplication.class, args);
    }
}
```

- Run the `main` method in the `Application.java` class to start the Spring Boot application.

**Step 7: Verify the Application**

- Open a browser and go to `http://localhost:8080/` (default port).
- If you see the error `Whitelabel Error Page`, the application has started successfully (since no endpoints have been configured yet).

**Lab Exercises:**

1. Setup the spring boot application and print the hello world message
2. Setup the spring boot application with hello world message in separate package named controller
3. Test the above application in POSTMAN

**Lab No 2 and 3:** **Date:**

# Data base and Data Structure for Fintech Lab

**Lab Activity:** Create schema, DDL for Create tables, create LDM relation between tables, Execute the DML SQL's
**Objectives:** In this lab sessions the students will be

- Learn the importance of standards to be followed and design the tables.
- Audit fields and its importance
- Introduction to Insert only Paradigm
- Documenting the Logical Data Model, Entities and Attributes as per the given requirement
- Create Schema, Tables with Primary Key and Foreign Key
- Insert data in DB for the tables created
- Perform DML operations on table created

**Output** : The outcome of this Lab will be

- Document LDM - Tables, columns and relationship between tables
- Tables created in MYSQL
- Data in the tables on performing insert operation
- DML Operations on table created - SELECT, UPDATE and DELETE

## Entity Relationship Diagram (ERD)

A data model is a visual representation of data elements and the relationships between them based on real-world objects. Data models reveal and define how data is connected within business processes and support the creation of efficient information systems or applications. For example, in business intelligence, a data model defines what kind of data users can utilize within their analytics.

An Entity Relationship (ER) Diagram or ERDs or ER models in short is a type of flowchart that illustrates how "entities" relate to each other within a system. An **entity–relationship model** (or **ER model**) describes interrelated things of interest in a specific domain of knowledge. ER Diagrams are most often used to design or debug relational databases in the fields of software engineering, business information systems, education and research. They use a defined set of symbols such as rectangles, diamonds, ovals and connecting lines to depict the interconnectedness of entities, relationships and their attributes. They mirror grammatical structure, with entities as nouns and relationships as verbs.
ER Diagrams are composed of entities, relationships and attributes. They also depict cardinality, which defines relationships in terms of numbers.

**Entity:** A definable thing—such as a person, object, concept or event—that can have data stored about it. Think of entities as nouns. Examples: a customer, student, car or product. Typically shown as a rectangle.

**Entity type:** A group of definable things, such as students or customer, whereas the entity would be the specific student or customer. Other examples: Athletes, cars or products.

**Entity set:** Same as an entity type, but defined at a particular point in time, such as customers created in a bank on 1st February 2024. Other examples: Customers who were created in last month, Cars currently registered in Manipal.
A related term is instance, in which the specific person(customer) or car would be an instance of the entity set.

**Entity categories**: Entities are categorized as strong, weak or associative. A strong entity can be defined solely by its own attributes, while a weak entity cannot. An associative entity associates entities (or elements) within an entity set.

**Entity keys:** Refers to an attribute that uniquely defines an entity in an entity set. Entity keys can be super, candidate or primary.

- Super key: A set of attributes (one or more) that together define an entity in an entity set.
- Candidate key: A minimal super key, meaning it has the least possible number of attributes to still be a super key. An entity set may have more than one candidate key.
- Primary key: A candidate key chosen by the database designer to uniquely identify the entity set. Foreign key: Identifies the relationship between entities.

**Entity Relationship:** How entities act upon each other or are associated with each other. Think of relationships as verbs. For example, the named customer might be registered in Manipal branch of a bank and would have provided with Drivers license

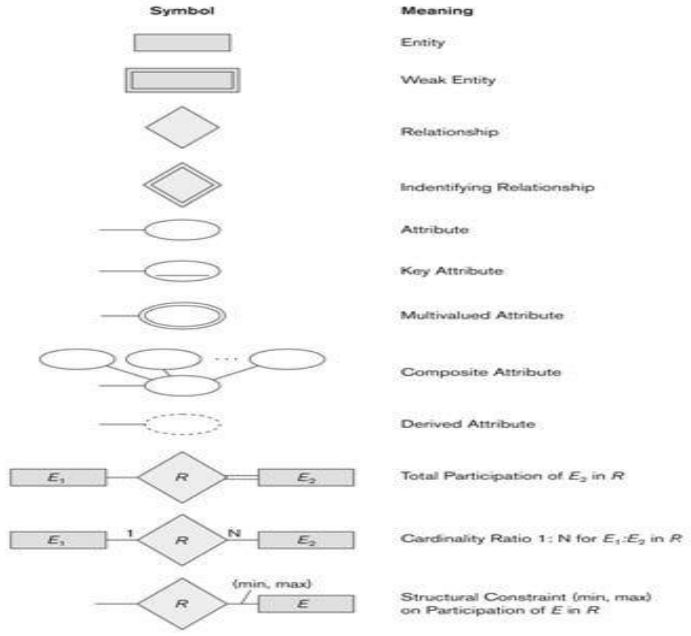
and Aadhar as proof of Identity. The two entities would be the Customer and the Customer Proof of Identity, and the relationship depicted is the act of connecting the two entities in that way. Relationships are typically shown as diamonds or labels directly on the connecting lines.
Recursive relationship: The same entity participates more than once in the relationship.

**Attribute:** A property or characteristic of an entity. Often shown as an oval or circle.

**Descriptive attribute:** A property or characteristic of a relationship (versus of an entity.)
**Attribute categories:** Attributes are categorized as simple, composite, derived, as well as single-value or multi-value.

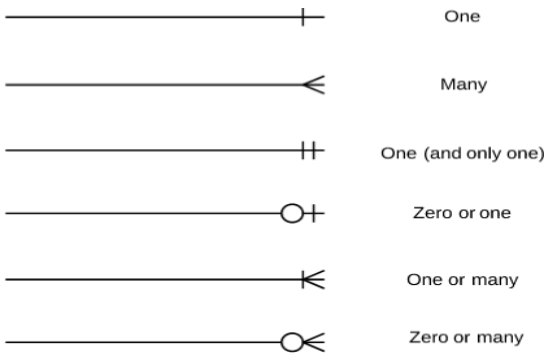
- Simple: Means the attribute value is atomic and can't be further divided, such as a phone number or Aadhar number of a customer.
- Composite: Sub-attributes spring from an attribute. Example Address of customer that can have more than one attribute
- Derived: Attributed is calculated or otherwise derived from another attribute, such as age from a birthdate.
- Multi-value: More than one attribute value is denoted, such as multiple phone numbers for a person.
- Single-value: Just one attribute value. The types can be combined, such as: simple single-value attributes or composite multi-value attributes.

**Symbols and their meaning used in ERD.**




**Cardinality**: Defines the numerical attributes of the relationship between two entities or entity sets. The three main cardinal relationships are:

- One-to-one example would be one customer associated with one Aadhar number.
- One-to-many example (or many-to-one, depending on the relationship direction): One customer having multiple accounts in a bank, but all those accounts have a single line back to that one customer.
- Many-to-many example: Relationship managers in a bank are associated with multiple customers, and customers in turn are associated with multiple relationship managers over the life of customer in that bank.



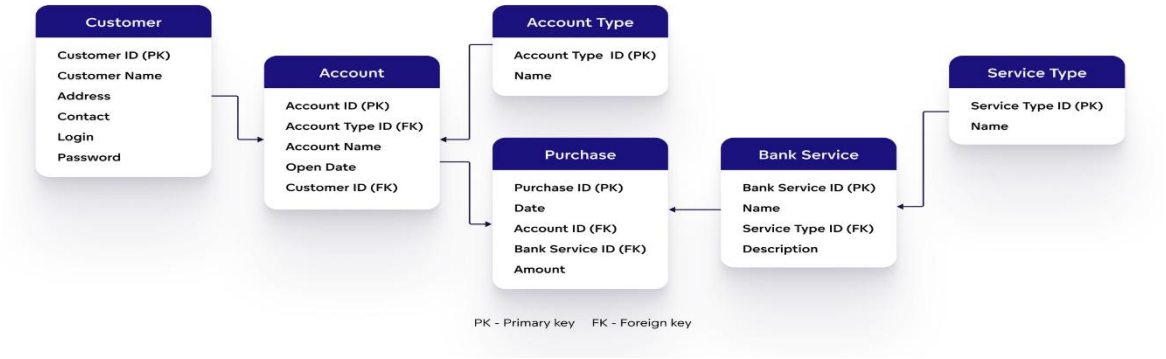

**Cardinality views:** Cardinality can be shown as look-across or same-side, depending on where the symbols are shown.

**Cardinality constraints:** The minimum or maximum numbers that apply to a relationship.

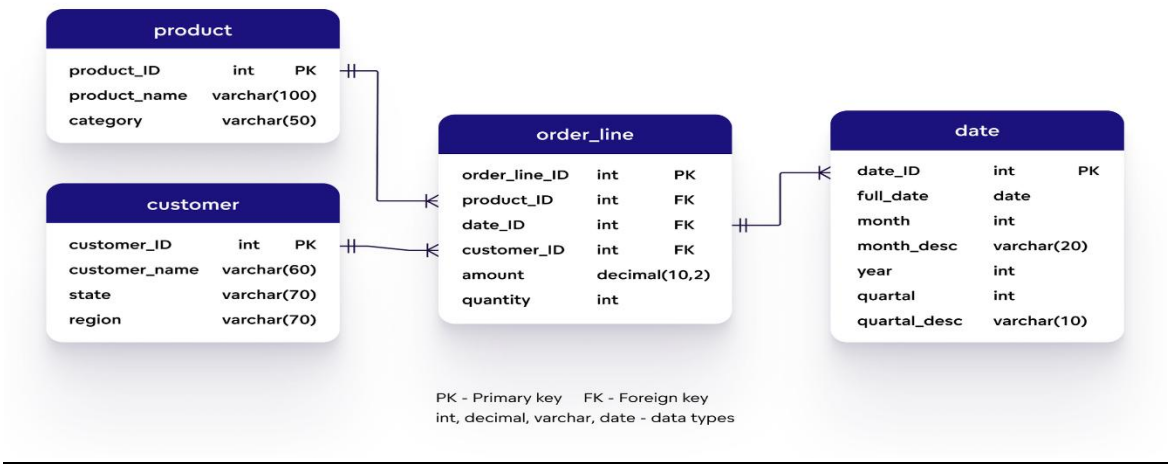
**Conceptual, logical and physical data models**

ER models and data models are typically drawn at up to three levels of detail:

- **Conceptual data model:** The highest-level view containing the least detail. Its value is showing overall scope of the model and portraying the system architecture. For a system of smaller scope, it may not be necessary to draw. Instead, start with the logical model.
- **Logical data model:** Contains more detail than a conceptual model. More detailed operational and transactional entities are now defined. The logical model is independent of the technology in which it will be implemented.
  A logical data model is a data model that provides a detailed, structured description of data elements and the connections between them. It includes all entities — a specific object transferred from the real world (relevant to business) — and the relationships among them. These entities have defined their attributes as their characteristics.
  Logical data models bring together the two most vital basics of application development — business requirements and quality data structure — into a visual representation. Business analysts and data architects are responsible for creating these models. They map relevant business processes and reveal the business requirements in order to create a model which meets company goals. Moreover, they prepare a technical map of rules and structures depending on the scope of the project.
  Example of a LDM in a banking system:

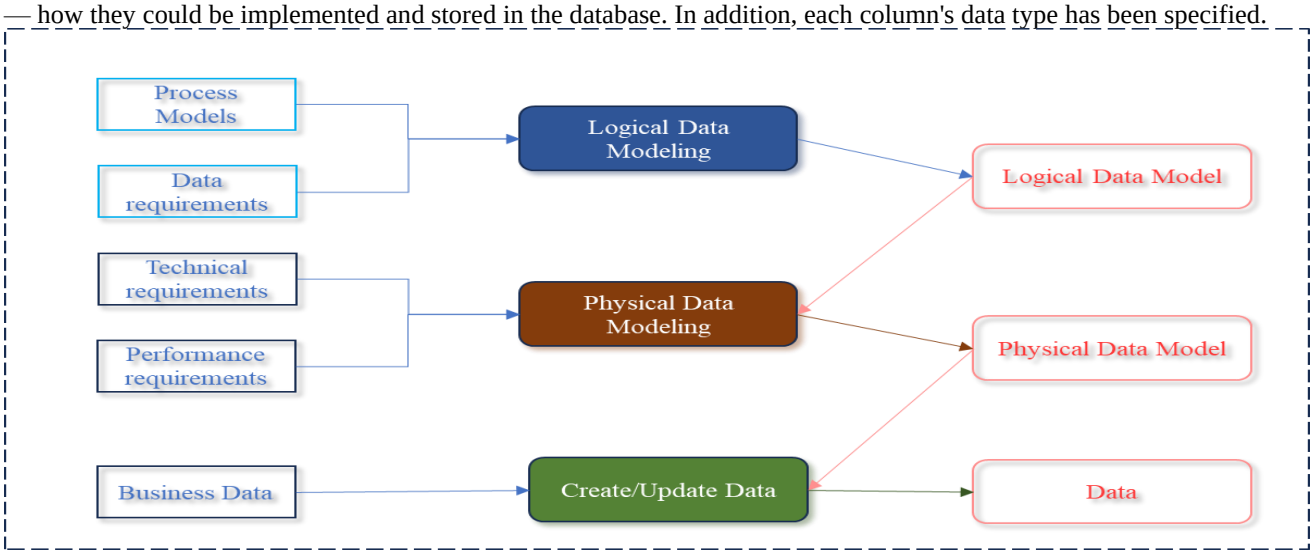



- **Physical data model:** One or more physical model may be developed from each logical model. The physical models must show enough technology detail to produce and implement the actual database.
  A physical data model specifies how the data model will be built in the database. It outlines all table structures, including column name, data types, column constraints, primary key and foreign key with indexes to the relevant table column, relationships between tables, stored procedures, and views.
  The responsibility regarding physical data model creation usually lies with database administrators and developers. Hence this exercise is included for the students to know about the ERD, how to define LDM and then Physical data model.
  Information systems and software applications heavily rely on interactions with physical databases. Physical data models need to be designed and implemented correctly. It is challenging to modify physical data models once data from the existing application has been inserted into databases.




**Converting Logical Data Model to Physical Data Model:**

Entities have been transformed into tables and attributes into table columns. Their names are also translated into technical terms — how they could be implemented and stored in the database. In addition, each column's data type has been specified.




Audit Details in Entities

One of the easiest ways to collect information about activity on a table is to add a set of audit columns on the table. The common audit columns in Fintech consists of

- When was the row created → The timestamps,
  - The time when the process was initiated that is Local time stamp. For systems catering to more than one country this is in UTC.
  - The time the data was inserted in the table that is Host time stamp. For systems catering to more than one country this is in UTC.
  - For systems catering to more than one country then the local timestamps as Acceptance Timestamp and UTC offset which is offset for acceptance timestamp.
- Who created it → the UserID or the ProcessID (in case of batch or other automated processes) that initiated the Process.
- How was it created → The business process or the module that created the data.
- Unique Service Request Identifier (UUID) → To identify the tables which have been updated as a part of the process in a given instance.

**Audit Columns:** These are the Audit Columns that should be present in all the entities. Column Names with "OOO" represent the Entity Code.

| Column Name | Data Type | Description |
|---|---|---|

| | | |
|---|---|---|
| OOO_USER_ID | VARCHAR2(32 CHAR) NOT NULL | User ID – May be the person. User id of the user who initiated the process |
| OOO_WS_ID | VARCHAR2(32 CHAR) NOT NULL | workstation ID/channel ID/IP Address |
| OOO_LOCAL_TS | TIMESTAMP NOT NULL | This is an Operation Timestamp. When the Transaction is initiated. This is in UTC. same timestamp will carry for a transaction. |
| OOO_HOST_TS | TIMESTAMP NOT NULL | This is UTC timestamp when the row got updated in the table. Each row of the table will have diff timestamp. |
| RULE_SYSTEM_ID | VARCHAR2(32 CHAR) NOT NULL | Business process/Module that initiated the process |
| LDBID | NUMBER(6) NOT NULL | The logical database identifier for the financial institution. |
| OOO_ACPT_TS | TIMESTAMP NOT NULL | This is NOT UTC time. this is a local/client system timestamp. This can be passed from the channel also.<br>If not passed, then this will contain the LOCAL_TS values but NOT in UTC. |
| OOO_ACPT_TS_UTC_OFST | CHAR(6 CHAR) NOT NULL | This is offset time-zone for the ACPT_TS. So that ACPT_TS can also convert to UTC. |
| UUID | VARCHAR2(36 CHAR) | Unique Service Request Identifier |

**Standard for defining tables and columns:**

In the software industry there are certain naming standards followed at each level.
In the database the following conventions needs to be followed:

1. Table names must have indicators of the functional area.
2. Usually, abbreviations are used while defining the tables and columns. Standard abbreviation sample is given. One can have their own standards but documenting them for others to understand is very important.
3. Underscores separate the distinct terms that define the Table or Column as per the standard conventions. Tablename and column name can have multiple underscores.

Database%20Standard%20Abbreviations.xl

Note: Oracle Financial Services Analytical Applications (OFSAA) naming standard is given as a sample for reference so that the students are aware that there are certain standards to be followed in the industry. This need not be followed. Student can come up with their own standards, document them in their journals along with the ERD.

Sample for Naming standard: 2 Table and Column Naming Standards

**Converting ERD to Tables and columns:**

Note: In all the tables include the audit columns when defining the table.

| Entity and Attribute Definitions | | |
|---|---|---|
| **Entity** | **Description/Attributes** | **Comments** |

| | | |
|---|---|---|
| Classification types | The main attributes are listed below.<br>**Entity: Customer Classification Table**<br>**Attributes:**<br>• Customer Classification ID<br>• Customer Classification Type<br>• Customer Classification Type Value<br>• Effective Date (Audit Field) | Teach how data can be classified and how new classifications can be added without having to change the DB structures.<br>Defining system data without having Screens/API's to capture them.<br>Use of INSERT SQL scripts.<br>Sample of listing the data for given classification type would be provided as sample.<br>Used to define:<br>* Id types<br>* Name types<br>* Customer Type<br>* Address Types etc. |
| Customer Identification | The main entities other than common ones are listed below.<br>**Entity: Customer Identification**<br>**Attributes:**<br>* Customer Identification ID<br>* Customer Identification Type<br>* Customer Identification Item<br>* Effective Date (Audit Field) | Teach generation of number by using DB sequence for Customer ID<br>Use of internal identifiers and external identifiers.<br>Unique identifier for a given customer - check for duplicates in user entered values.<br>Use of alternate options like phone number for customer identification. |
| Customer Details | The main entities other than common ones are listed below. The detail structure is defined in Table Structure worksheet under<br>**Entity: Customer Details Table**<br>**Attributes:**<br>* Customer Number<br>* Customer Type<br>* Customer Full Name<br>* Customer DOB/DOI<br>* Customer Status<br>* Customer Contact Number<br>* Customer Mobile Number<br>* Customer Email ID<br>* Customer Country of Origination<br>* Effective Date | Teach how data can be stored in different columns as the data of this type are unique and not repetitive.<br>Use of composite Keys.<br>Use Of Foreign Key.<br>Various Data types. |
| Customer Name | The main entities other than common ones are listed below.<br>**Entity: Customer Name components**<br>**Attribute:**<br>* Customer Identifier<br>* Customer Name Component Type – From Classification ID<br>* Customer Name value<br>* Effective Date | This will be provided as API from the code connect which the students can use to store the customer's name components |
| Customer Proof of Identity | The main entities other than common ones are listed below.<br>**Entity**: **Customer Proof of Identity**<br>**Attributes:**<br>* Customer Identifier<br>* Customer Proof of ID Type - From Classification ID<br>* Customer Classification Type Value<br>* Start Date<br>* End Date<br>* Effective Date | This can be part of Mini project<br>Teach how different types of same information can be stored as rows in table than adding them as columns. Validity of data, history of data etc.<br><br>One or more ID Types can be defined.<br>Each ID type and value should be unique.<br>At a given point in time there should not be duplicate values for a given ID type<br>Each ID type should have a start and end date.<br>For the type that are unique for the life of customer the end date should be max date and can never expire.<br>All the ID types should retain the history. |

| | | |
|---|---|---|
| Customer Address | The main entities other than common ones are listed below. The detail structure is defined in Table Structure worksheet under - **Customer Address components**<br>* Customer Identifier<br>* Customer Address Component Type - From Classification ID<br>* Customer Address value<br>* Effective Date | This can be part of Mini project<br>This will be provided as API from the code connect which the students can use to store the customer address compone+C24nts |

**Sample of Customer Classification Table Structure:**

| **Customer Classification** | | | |
|---|---|---|---|
| Entity Name | Short name | Description | |
| Customer Classification | CUST_CL | Used to define different classification types used to design customer data.<br>Example:<br>Customer Types - Individual, Corporate<br>Customer Name Types - First Name, Middle Name, Last Name etc.<br>Customer ID Types - Aadhar, Passport No, Driving License etc.<br>Customer Address Type - First Line, Middle Line, Last Line etc. | |
| **Attribute** | **Short Name** | Data Type / Size | **Description** |
| Customer Classification ID | CSTCL_ID | NUMBER(15) | Classification ID which will be used in other tables to store the value |
| Customer Classification Type | CSTCL_TYP | VARCHAR(100) | Defines the Classification type |
| Customer Classification Type Value | CSTCL_TYP_VALUE | VARCHAR(100) | Defines the Classification Value |
| Effective Date (Audit Field) | CSTCL_EFCTV_DT | DATE | Date from when the data is effective from |
| CRUD Value | CSTCL_CRUD_VALUE | CHAR(1) | Logical status of the data<br>C - Created<br>U - Updated<br>D – Deleted |
| User ID (Audit Field) | CSTCL_USER_ID | VARCHAR(100) | User who created the data |
| Workstation ID (Audit Field) | CSTCL_WS_ID | VARCHAR(100) | Workstation from where the data is created |
| Program ID (Audit Field) | CSTCL_PRGM_ID | VARCHAR(100) | Program that inserted the data |
| Host Timestamp (Audit Field) | CSTCL_HOST_TS | TIMESTAMP | The timestamp when the record is inserted in the DB to be obtained in the program from the host |
| Local Timestamp (Audit Field) | CSTCL_LOCAL_TS | TIMESTAMP | The timestamp of the system that initiates the record i.e the timestamp of the user workstation which can be different from the host timestamp |
| Acceptance Timestamp (Audit Field) | CSTCL_ACPT_TS | TIMESTAMP | The timestamp when the record is accepted in the system |
| Acceptance Timestamp UTC offset (Audit Field) | CSTCL_ACPT_TS_UTC_OFST | TIMESTAMP | The timestamp when the record is inserted in the DB to be obtained in the program from the host |
| Universal Unique Identifier (UUID) | CSTCL_UUID | VARCHAR(100) | Universal Unique Identifier is system generated ID, used to uniquely identify the Process that updated the records in the system. |

**Note:** Students are expected to create a similar table for all the entities defined above.

**Lab Activities**

1. Design an ERD/LDM for the above entities.
2. Define the physical tables in the table format given as sample for Customer Classification with the data type, size and explanation of each field.
   This is to teach the importance of documentation at each level. In the industry it is very important that there is enough documentation provided so that it becomes easy to learn and maintain it.
3. Create Schema and tables SQL's, also create the DDL scripts for table creation

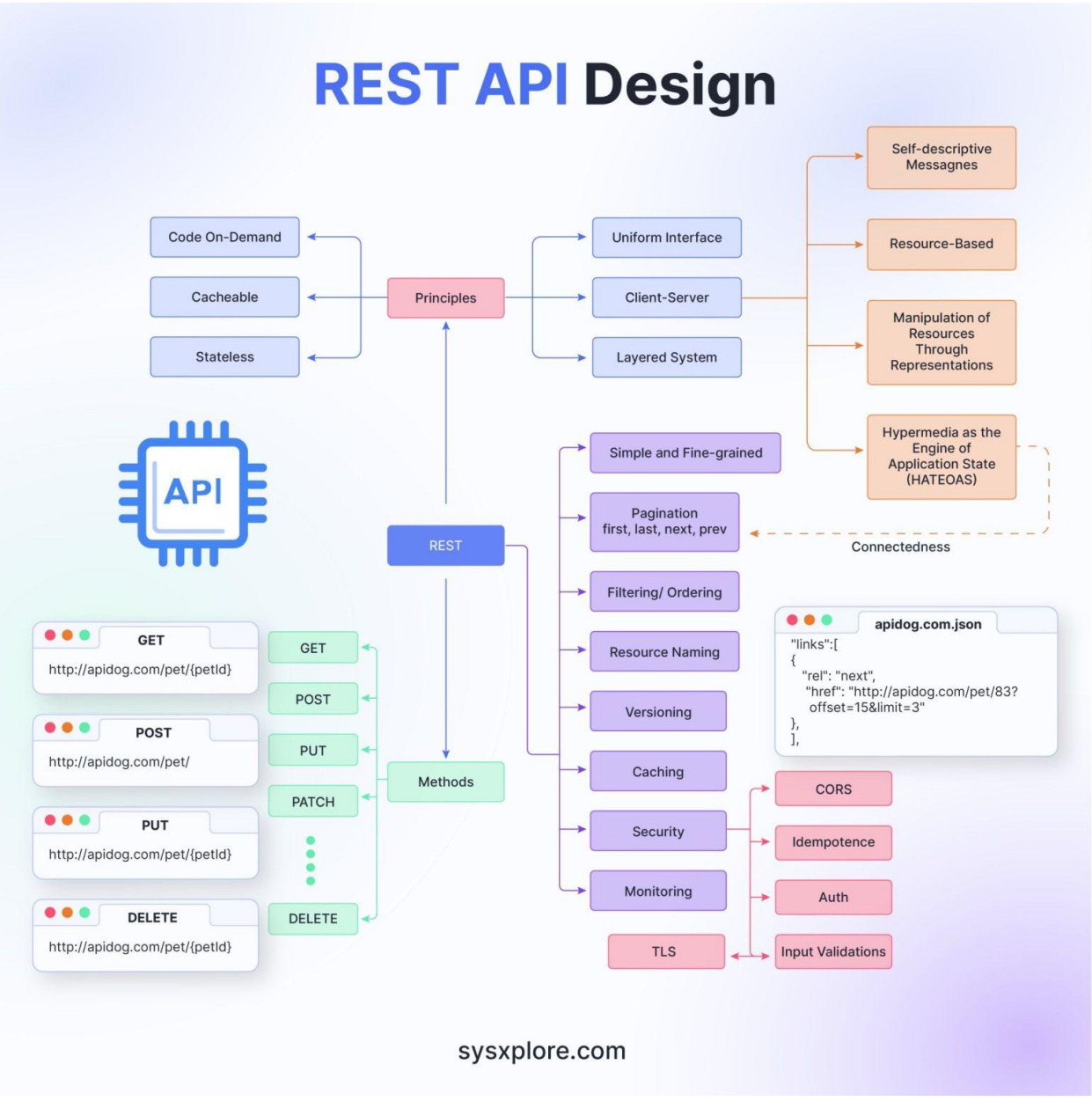
4. DML Operations on table created - SELECT, UPDATE and DELETE

**Lab No 4:** **Date:**

# Writing RestAPI

## REST API Design:

Designing a REST API involves several key principles and best practices to ensure that the API is efficient, easy to use, and maintainable. Here are some fundamental concepts to consider:




1. **Resources and URI's :**

In REST, a resource is a fundamental building block that represents a specific item of data, such as a customer, a product, or an order. Resources are identified by URIs (Uniform Resource Identifiers), which are unique addresses that can be used to access and manipulate the resource.

- Identify Resources: Determine the main entities your API will manage (e.g., customer, products, orders). Use Nouns for URIs: Design URIs using nouns that represent resources.

For example:

- `GET /customer` - Retrieve a list of customers based on filter criteria
- `POST /customer` - Create a new customer
- `GET /customers/{id}` - Retrieve a specific customer
- `PUT /customer/{id}` - Update a specific customer
- `DELETE /customer/{id}` - Delete a specific customer

**2. HTTP Methods:**

For designing REST APIs, Each HTTP request includes a method, sometimes called "HTTP verbs," that provides a lot of context for each call. Here's a look at the most common HTTP methods:

Use appropriate HTTP methods to perform actions:

- GET: Retrieve data
- POST: Create a new resource
- PUT: Update an existing resource
- PATCH: Partially update a resource
- DELETE: Remove a resource

3. **Statelessness:**
   Each request from a client should contain all the information needed to process that request. The server should not store the client context between requests.

4. **Status Codes**
   Use standard HTTP status codes to indicate the outcome of an API request. There are many more HTTP status codes to consider, but the below lists should cater to for most APIs.

- **200**: Successful request, often a GET
- **201**: Successful request after a create, usually a POST
- **204**: Successful request with no content returned, usually a PUT or PATCH
- **301**: Permanently redirect to another endpoint
- **400**: Bad request (client should modify the request)
- **401**: Unauthorized, credentials not recognized
- **403**: Forbidden, credentials accepted but don't have permission
- **404**: Not found, the resource does not exist
- **410**: Gone, the resource previously existed but does not now
- **429**: Too many requests, used for rate limiting and should include retry headers
- **500**: Server error, generic and worth looking at other 500-level errors instead
- **503**: Service unavailable, another where retry headers are useful

**5. Versioning**

- Version your API to manage changes over time. Common practices include:
- Using a version number in the URI (e.g., `/v1/users`)
- Using a version number in request headers

6. **Filtering, Sorting, and Pagination**
   Allow clients to filter results and sort them rather than creating redundant end points. Plan with smart parameters from the start

- **Filtering**: Return only results that match a filter by using field ‘name’ as a parameter for customer search.Use the wildcard characters like “*”(Asterix) For example:

  GET /customer?name=John → Customers with Exact Match
  GET /customer?name=John* →Customer with wildcard filter

- **Pagination:** Implement pagination for large datasets. Don’t overload clients and servers by providing everything. Instead, set a limit and provide prev and next links in the response
  - `GET /customer?age=25&sort=name&page=2&limit=10`

- **Sorting:** Provide a way to sort or some use cases will still require paging through all results to find what’s needed, in which case it is better to sort in some order. Example:

  GET / customer?sort_by=name&order=asc

7. **Documentation**
   Provide clear and comprehensive documentation for the API. Tools like Swagger or Postman can help create interactive API documentation. It would contain all the information a developer needs to integrate with the API

8. **Security**
   Implement authentication (e.g., OAuth, JWT) and authorization to protect the API. Use HTTPS to encrypt data in transit.

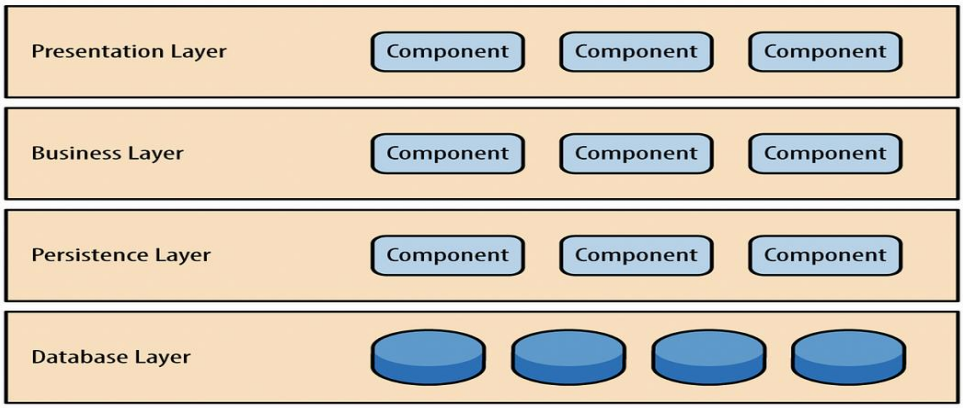
9. **Error Handling**
   Provide meaningful error messages and codes to help clients understand issues.

10. **Testing**
    Test the API thoroughly to ensure it behaves as expected under various conditions.

11. **Cache Data**
    Cache data to Improve Performance. Caching partially eliminates unnecessary trips to the server. Returning data from local memory rather than sending a query for each new request can improve your app’s performance. GET requests are cacheable by default, however, POST requests require you to specify the cache requirements in the header. Caching, however, can lead to stale data on the client’s browser. Here’s how you would deal with this in the POST header.

## Layered Architecture Pattern:

Layered architecture patterns are n-tiered patterns where the components are organized in horizontal layers. This is the traditional method for designing most software and is meant to be self-independent. This means that all the components are interconnected but do not depend on each other. This pattern organizes the system into layers, each with specific responsibilities, such as presentation, business logic, and data access. It promotes separation of concerns and makes the system easier to manage.




There are four layers in this architecture where each layer has a connection between modularity and component within them. From top to bottom, they are:

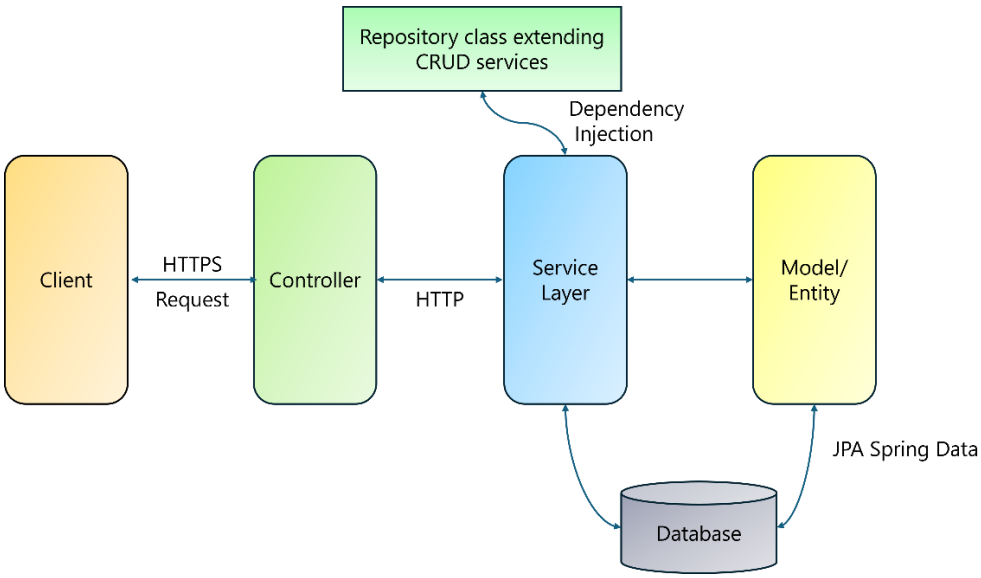
- The presentation layer: It contains all categories related to the presentation layer.
- The business layer: It contains business logic.
- The persistence layer: It's used for handling functions like object-relational mapping
- The database layer: This is where all the data is stored.

In this instance the layers are closed, meaning a request must go through all layers from top to bottom. There are two reasons for this, one being that all 'similar' components are together and the other reason is that it provides layers of isolation.

To elaborate, having 'similar' components together means that everything relevant to a certain layer, stays in that single layer. This allows for a clean separation between types of components and also helps gather similar programming code together in one location. By isolating the layers, they become independent from one another. Thus if, for example, if one want to change the database from an Oracle server to a SQL server, this will cause a big impact on the database layer but that won't impact any other layers. Likewise, suppose that one have a custom written business layer and want to change it for a business rules engine. The change won't affect other layers if there is a well-defined layered architecture.

## Springboot Flow Architecture

To understand the spring boot flow architecture, lets break down how a typical spring boot application works from startup to handling a web request or API call , and its internal components.



## 1. Spring Boot Application Startup

The lifecycle of a Spring Boot application begins when it starts running. This process can be broken into the following key phases:

a. **@SpringBootApplication Annotation**

The Spring Boot application typically starts from a class annotated with @SpringBootApplication. This annotation is a convenience that includes three crucial annotations:

- @SpringBootConfiguration: Defines this class as the configuration class.
- @EnableAutoConfiguration: Enables Spring Boot's automatic configuration, which scans for components and configures them automatically based on the project's classpath (e.g., when spring-boot-starter-web is included, it automatically sets up an embedded web server).
- @ComponentScan: Scans the package and sub-packages where the main class is located for components like @Controller, @Service, and @Repository.

b. **SpringApplication.run()**

When SpringApplication.run() is invoked in the main method, the Spring Boot application:

- Initializes the Spring context.
- Loads configuration properties.
- Scans for Spring components (@Controller, @Service, etc.).
- Configures beans and dependencies via dependency injection.
- Starts the embedded web server (like Tomcat or Jetty) if it's a web application.

This automatic setup significantly reduces the amount of configuration code the developer needs to write.

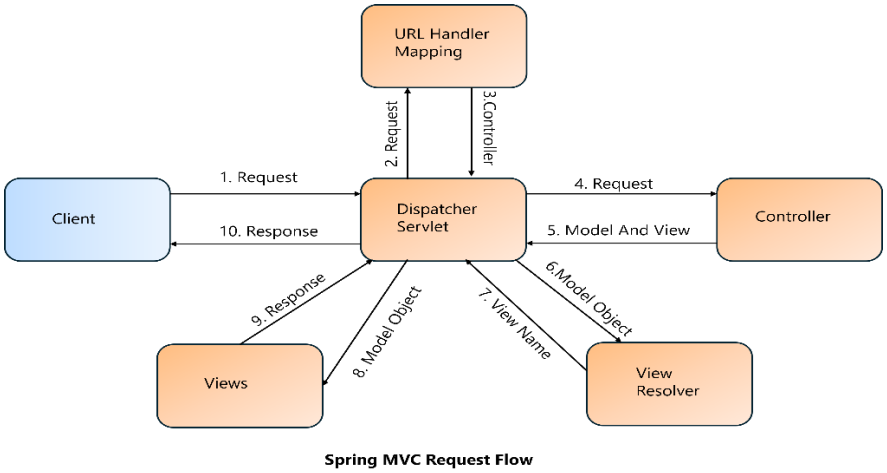
Example:

```
@SpringBootApplication
public class MitDemoApplication {
    public static void main(String[] args) throws IOException {
        SpringApplication.run(MitDemoApplication.class, args);
    }
}
```

## 2. Request Handling Flow (MVC Architecture)

In a Spring Boot web application, handling web requests follows a Model-View-Controller (MVC) architecture. Let's walk through the flow of how a request is processed:

**Below is the flowchart for Spring MVC Request Flow:**




**Spring MVC Request Flow**

**a. Client Request (HTTP Request)**

When a client (usually a browser or a REST API client) sends an HTTP request to the application (e.g., a GET request to /hello), the request enters the application through the embedded web server (Tomcat, Jetty, etc.).

**b. Front Controller (DispatcherServlet)**

At the core of Spring MVC is the DispatcherServlet, which acts as a front controller that intercepts all incoming HTTP requests and dispatches them to the appropriate handlers. It is automatically configured by Spring Boot. DispatcherServlet intercepts the request. It consults the HandlerMapping, which contains mappings of URLs to controllers, and finds the appropriate handler (usually a controller class and its method).

**c. Controller Layer (@RestController or @Controller)**

Once the request reaches the controller, Spring injects the request data (path variables, request parameters, or request bodies) into the corresponding controller method. The controller processes the request, usually with the help of a service layer, and returns a response.

- @Controller: Used for traditional web applications where the controller returns a view (e.g., an HTML page).
- @RestController: A specialization of @Controller where each method returns data directly (like JSON or XML), typically used for RESTful APIs. Please find the example below:

  Example:

  *@RestController*

  *public class CustomerController1 {*

  *@GetMapping("/getCustomerDetails/{id}")*

  *public StandardResponse<CustomerDetailsResponseDTO> getCustomerDetails(@PathVariable Long id)*

  *{*

  *StandardResponse<CustomerDetailsResponseDTO> response = new*

  *StandardResponse<CustomerDetailsResponseDTO>();*

  *response.setResponseOK();*

  *response.setData(customerService.getCustomerDetails(id));*

```
        return response;
        }
}
```

Here, /getCustomerDetails/{id} maps to the getCustomerDetails() method in the CustomerController, which returns customer Details in response.

**d. Service Layer (@Service)**

The service layer contains the business logic. The controller often delegates work to the service layer, which processes the core business logic and interacts with the repository or other services.
Example:

```
@Service
public class CustomerServiceImpl implements CustomerService {
@Override
public CustomerDetailsResponseDTO getCustomerDetails(Long id) {
        CustomerDetails customerDetails = customerDetailsRepository.findBycustomerIdentifier(id);
        CustomerDetailsResponseDTO customerDetailsResponseDTO =
        customerServiceHelper.generateCustomerDetailsObjectForGet(customerDetails);
        return customerDetailsResponseDTO;
        }
}
```

**e. Repository Layer (@Repository)**

If the application interacts with a database, the service layer usually interacts with a repository. The repository layer is where Spring Boot manages data access using interfaces that extend Spring Data repositories like JpaRepository, CrudRepository, etc.
Example:

```
@Repository
public interface CustomerDetailsRepository extends JpaRepository<CustomerDetails, Long> {
   // Custom database queries can be defined here
}
```

The repository interacts with the database (e.g., via JPA, JDBC, or Hibernate), handling CRUD operations (Create, Read, Update, Delete).

**f. Response Handling**

Once the controller receives data from the service (and possibly the repository), it returns a response back to the client.

- In RESTful applications, the response is typically in JSON or XML format.
- In traditional web applications, the response can be an HTML page (using @Controller and view technologies like Thymeleaf).

- In the example above, the *getCustomerDetails* () method returns customer Details which is sent as an HTTP response back to the client.

**3. Spring Boot Auto-Configuration**

One of Spring Boot's most powerful features is auto-configuration. Spring Boot inspects the classpath and configures the application based on the dependencies and beans found. For example: If Spring Web is found in the classpath, Spring Boot automatically configures an embedded web server (like Tomcat) and MVC architecture. It automatically configures a database connection and JPA repositories.This means developers don't need to manually define these components, and Spring Boot provides sensible defaults that can be customized if necessary.

**4. Embedded Web Server**

Spring Boot applications typically run on an embedded web server (like Tomcat or Jetty). This server is started automatically when the application starts, and it listens for HTTP requests on a default port (usually 8080).

**Spring Boot Flow Summary**

Startup: @SpringBootApplication initializes the application.

Request Flow: Client sends an HTTP request, which is routed by the DispatcherServlet to the correct controller.

Controller: Receives the request, interacts with services, and returns a response.

Service Layer: Business logic is executed, often using the repository for data access.

Repository: Interacts with the database and returns the necessary data.

Response: The controller sends the response (JSON, XML, or HTML) back to the client.

**Conclusion**

Spring Boot simplifies the Spring ecosystem with auto-configuration, embedded servers, and convention-over-configuration principles. The core architecture follows the traditional Model-View-Controller (MVC) pattern, but Spring Boot adds modern RESTful handling capabilities, all with minimal setup. The flow from startup to request handling is designed to minimize boilerplate code while retaining the flexibility and power of the Spring framework.

**LAB3-Spring Boot With MySQL**

Objectives

1. **Set up** a MySQL database and configure it for integration with Spring Boot.
2. **Implement** a Spring Boot project with RESTful endpoints interacting with a MySQL database.
3. **Design** a complete Spring Boot application with a custom database schema and CRUD operations.
4. **Develop** additional layers such as Service and Exception Handling to enhance the functionality and reliability of the application.

Spring Boot is a powerful framework for building Java applications, and when combined with MySQL, it is ideal for creating web applications with persistent storage. Here's a detailed explanation and step-by-

step guide to integrate **Spring Boot** with **MySQL**:

---

## 1. Prerequisites

- **JDK** (Java Development Kit): Version 8 or later.
- **MySQL**: Installed and running.
- **Maven** or **Gradle**: For dependency management.
- **Spring Boot**: Basic knowledge.

---

## 2. Steps to Integrate Spring Boot with MySQL

### Step 1: Setup MySQL Database

1. Install MySQL (if not already done).
2. Start MySQL and create a new database:

```
CREATE DATABASE springboot_db;
```

3. Create a MySQL user (optional) and grant privileges:

```
CREATE USER 'PKA'@'localhost' ;
ALTER USER 'PKA'@'localhost' IDENTIFIED WITH mysql_native_password BY
'Password' ;
GRANT ALL PRIVILEGES ON springboot_db.* TO 'PKA'@'localhost';
FLUSH PRIVILEGES;
```

---

### Step 2: Create a Spring Boot Project

1. **Using Spring Initializer** (Recommended):
   - Go to Spring Initializer.
   - Select:
     - Project: Maven or Gradle.
     - Dependencies:
       - Spring Web
       - Spring Data JPA
       - MySQL Driver
   - Generate the project as a **.zip** file and extract it.
2. **Using Your IDE**:
   - Create a new Maven/Gradle project.
   - Add dependencies to the `pom.xml` or `build.gradle` file.

---

### Step 3: Add Dependencies

For **Maven** (`pom.xml`):

```
<dependencies>
    <!-- Spring Boot Starter for Web -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!-- Spring Boot Starter for JPA -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <!-- MySQL Driver -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
    </dependency>
</dependencies>
```

### Step 4: Configure Application Properties

Edit `src/main/resources/application.properties`:

```
# MySQL Configuration
spring.datasource.url=jdbc:mysql://localhost:3306/springboot_db
spring.datasource.username=PKA
spring.datasource.password=Password

# JPA Configuration
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```

---

### Step 5: Create Entity Class

Create a Java class annotated with `@Entity` to map to a database table.

```
package com.example.demo.entity;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
```

```
@Entity
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String email;

    // Getters and Setters
}
```

---

**Step 6: Create Repository Interface**

Define a repository to perform CRUD operations.

```
package com.example.demo.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.example.demo.entity.User;

public interface UserRepository extends JpaRepository<User, Long> {
}
```

---

**Step 7: Create a Service Layer**

Add business logic by creating a service class.

```
package com.example.demo.service;

import java.util.List;
import com.example.demo.entity.User;
import com.example.demo.repository.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    public List<User> getAllUsers() {
        return userRepository.findAll();
    }

    public User saveUser(User user) {
        return userRepository.save(user);
    }
}
```

---

### Step 8: Create a Controller

Expose REST endpoints for interacting with the database.

```
package com.example.demo.controller;

import com.example.demo.entity.User;
import com.example.demo.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/users")
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping
    public List<User> getAllUsers() {
        return userService.getAllUsers();
    }

    @PostMapping
    public User createUser(@RequestBody User user) {
        return userService.saveUser(user);
    }
}
```

---

### Step 9: Run the Application

1. Run the main application class (`DemoApplication.java`):

   ```
   @SpringBootApplication
   public class DemoApplication {
       public static void main(String[] args) {
           SpringApplication.run(DemoApplication.class, args);
       }
   }
   ```

2. Verify that the application starts without errors.

---

### Step 10: Test the Application

1. Use **Postman** or **cURL** to test the API:
   - **GET**: `http://localhost:8080/api/users`
   - **POST**: `http://localhost:8080/api/users`

```
{
    "name": "PKA",
    "email": "PKA@manipal.edu"
}
```

2. Check the MySQL database to confirm data persistence.

**Lab Exercises:**

1. Establish Spring Boot MySQL Connection with CustomerDetail table
2. Establish Spring Boot MySQL Connection with CustomerIdentification table
3. Establish Spring Boot MySQL Connection with CustomerContactInformation table
4. Establish Spring Boot MySQL Connection with CustomerProofOfId table

**Lab No 5 and 6:** **Date:**

# Building a CRUD Application with Spring Boot

## Objective

To understand and implement RESTful APIs in a Spring Boot application using a service layer.

---

## Prerequisites

1. Java Development Kit (JDK) installed.
2. Maven or Gradle installed.
3. An Integrated Development Environment (IDE) like IntelliJ IDEA, Eclipse, or VS Code.
4. Basic understanding of Java, Spring Boot, and REST APIs.

---

## Step 1: Setting Up the Project

1. Go to Spring Initializr.io
2. Configure the project:
    - Project: **Maven**
    - Language: **Java**
    - Dependencies: **Spring Web**, **Spring Data JPA**, **MySQL Database**.
3. Generate the project and unzip it.
4. Open the project in your IDE.

---

## Step 2: Project Structure

The project will have the following structure:

```
src/main/java/com/example/demo/
├── controller/
│   └── ItemController.java
├── entity/
│   └── Item.java
├── repository/
│   └── ItemRepository.java
├── service/
│   ├── ItemService.java
│   └── ItemServiceImpl.java
└── DemoApplication.java
```

---

## Step 3: Writing the Code

### 3.1 Create the `Item` Entity

Represents the database table structure.

```
package com.example.demo.entity;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
public class Item {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String description;

    // Getters and setters
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }
}
```

---

### 3.2 Create the Repository

Handles database interactions.

```
package com.example.demo.repository;

import com.example.demo.entity.Item;
```

```
import org.springframework.data.jpa.repository.JpaRepository;

public interface ItemRepository extends JpaRepository<Item, Long> {
}
```

---

### 3.3 Create the Service Interface

Defines business logic methods.

```
package com.example.demo.service;

import com.example.demo.entity.Item;

import java.util.List;
import java.util.Optional;

public interface ItemService {
    List<Item> getAllItems();
    Optional<Item> getItemById(Long id);
    Item createItem(Item item);
    Item updateItem(Long id, Item itemDetails);
    void deleteItem(Long id);
}
```

---

### 3.4 Implement the Service

Implements the business logic.

```
package com.example.demo.service;

import com.example.demo.entity.Item;
import com.example.demo.repository.ItemRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Optional;

@Service
public class ItemServiceImpl implements ItemService {

    @Autowired
    private ItemRepository itemRepository;

    @Override
    public List<Item> getAllItems() {
        return itemRepository.findAll();
    }

    @Override
    public Optional<Item> getItemById(Long id) {
        return itemRepository.findById(id);
    }
```

```
    @Override
    public Item createItem(Item item) {
        return itemRepository.save(item);
    }

    @Override
    public Item updateItem(Long id, Item itemDetails) {
        return itemRepository.findById(id)
                .map(item -> {
                    item.setName(itemDetails.getName());
                    item.setDescription(itemDetails.getDescription());
                    return itemRepository.save(item);
                })
                .orElseThrow(() -> new RuntimeException("Item not found with id: " +
id));
    }

    @Override
    public void deleteItem(Long id) {
        itemRepository.findById(id).ifPresentOrElse(
                itemRepository::delete,
                () -> {
                    throw new RuntimeException("Item not found with id: " + id);
                }
        );
    }
}
```

---

### 3.5 Create the Controller

Handles HTTP requests and interacts with the service.

```
package com.example.demo.controller;

import com.example.demo.entity.Item;
import com.example.demo.service.ItemService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/items")
public class ItemController {

    @Autowired
    private ItemService itemService;

    @GetMapping
    public List<Item> getAllItems() {
        return itemService.getAllItems();
    }

    @GetMapping("/{id}")
```

```
    public ResponseEntity<Item> getItemById(@PathVariable Long id) {
        return itemService.getItemById(id)
                .map(item -> new ResponseEntity<>(item, HttpStatus.OK))
                .orElse(new ResponseEntity<>(HttpStatus.NOT_FOUND));
    }

    @PostMapping
    public ResponseEntity<Item> createItem(@RequestBody Item item) {
        Item savedItem = itemService.createItem(item);
        return new ResponseEntity<>(savedItem, HttpStatus.CREATED);
    }

    @PutMapping("/{id}")
    public ResponseEntity<Item> updateItem(@PathVariable Long id, @RequestBody Item
itemDetails) {
        try {
            Item updatedItem = itemService.updateItem(id, itemDetails);
            return new ResponseEntity<>(updatedItem, HttpStatus.OK);
        } catch (RuntimeException e) {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteItem(@PathVariable Long id) {
        try {
            itemService.deleteItem(id);
            return new ResponseEntity<>(HttpStatus.NO_CONTENT);
        } catch (RuntimeException e) {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    }
}
```

---

## Step 4: Running the Application

1. Run the application using your IDE or the command

   ```
   mvn spring-boot:run
   ```

2. Access the API using **Postman** or **cURL**.

---

## Step 5: Testing Endpoints

1. **POST**: Add a new item.

   ```
   POST /api/items
   {
       "name": "Laptop",
       "description": "A powerful laptop"
   }
   ```

2. **GET**: Retrieve all items.

```
GET /api/items
```

3. **PUT**: Update an item.

```
PUT /api/items/1
{
    "name": "Updated Laptop",
    "description": "An updated description"
}
```

4. **DELETE**: Delete an item.

```
DELETE /api/items/1
```

Lab Exercises

1. Write a spring boot application for PUT, POST, and GET customer name
2. Write a spring boot application for PUT, POST, and GET customer details
3. Write a spring boot application for PUT and GET customer identification
4. Write a spring boot application for PUT, POST, and GET customer proof of id
5. Write a spring boot application for PUT, POST, and GET customer contact details

**Lab No 7 and 8:** **Date:**

# Build UI Screens for Customer App

## 1. Screen Design UI and UX

- Focuses on creating visually appealing and user-friendly interfaces.
- Ensures intuitive navigation and usability for users.
- Balances aesthetics with functionality to improve user satisfaction.

## 2. Learn About the Landing Page

- The first page users see upon accessing the application or website.
- It sets the tone for the entire application, providing key information.
- Designed to guide users to the next steps or actions, such as logging in or exploring features.

## 3. Learn About Navigation from One Screen to Another

- Refers to the logical and seamless transition between screens in an application.
- Ensures that users can efficiently access desired features or information.
- Maintains consistency in design to avoid user confusion.

## 4. Learn About Displaying Errors

- Involves informing users about issues encountered during interaction.
- Error messages should be clear, concise, and actionable.
- Helps users correct mistakes or understand what went wrong.

## 5. Learn Field-Level Validations and Length Constraints

- Validations are rules applied to input fields to ensure data accuracy.
- Includes checks for correct format, data type, and mandatory fields.
- Length constraints define the maximum and/or minimum number of characters allowed for input.

## 6. Screens Invoking Existing APIs

- UI screens interact with backend APIs to fetch or submit data.
- Ensures that the application is leveraging pre-built functionalities for efficiency.
- Requires proper error handling to manage issues during API calls.

## 7. List Screen Pagination Logic

- Used to divide large datasets into manageable chunks for display.
- Reduces loading time and improves performance by fetching data in smaller subsets.
- Provides users with navigation options to move through pages of data systematically.

**User. Java**

```
package com.MIT.FIS10.entity;

import jakarta.persistence.*;
import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Size;

@Entity
@Table(name = "`user`")

public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank(message = "Name is required")
    @Size(min=3, max = 15, message = "Name must be between 3 and 15 characters")
    private String name;

    @NotBlank(message = "Email is required")
    @Email(message = "Invalid email address")
    private String email;

    // Getters and setters
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
}
```

**UserService.java**

```
package com.MIT.FIS10.service;

import com.MIT.FIS10.entity.User;
import com.MIT.FIS10.exception.ResourceNotFoundException;
import com.MIT.FIS10.repository.UserRepository;

import jakarta.validation.Valid;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.stereotype.Service;

@Service
public class UserService {

   @Autowired
   private UserRepository userRepository;

   public Page<User> getPaginatedUsers(int page, int pageSize) {
      return userRepository.findAll(PageRequest.of(page, pageSize));
   }

         public List<User> getAllUsers() {
                  // TODO Auto-generated method stub
                  return userRepository.findAll();
         }

         public void saveUser(@Valid User user) {
                   userRepository.save(user);

         }
         public void deleteUserById(Long id) {
      if (userRepository.existsById(id)) {
         userRepository.deleteById(id);
      } else {
         throw new ResourceNotFoundException("User not found with ID: " + id);
      }
   }
         // Method to find paginated users
         public Page<User> findPaginated(int page, int size) {
      // Create a Pageable object using PageRequest.of
      PageRequest pageable = PageRequest.of(page - 1, size);  // Page starts at 0, so subtract 1
      Page<User> usersPage = userRepository.findAll(pageable);  // Fetch paginated users
      return (usersPage);  // Wrap it in PageTable
   }
```

```
        public void updateUser(Long id, User user) {
    // Check if the user exists in the database
    User existingUser = userRepository.findById(id).orElseThrow(() -> new ResourceNotFoundException("User not found"));
    // Update user details
    existingUser.setName(user.getName());
    existingUser.setEmail(user.getEmail());
    // Save the updated user
    userRepository.save(existingUser);
  }


        // Method to find a user by id
  public User findById(Long id) {
    return userRepository.findById(id).orElseThrow(() -> new ResourceNotFoundException("User not found"));
  }
}
```

**UserRepository.java**

```
package com.MIT.FIS10.repository;

import com.MIT.FIS10.entity.User;
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long> {
}
```

UserController.java

```
package com.MIT.FIS10.controller;

import com.MIT.FIS10.service.UserService;

import jakarta.validation.Valid;

import com.MIT.FIS10.entity.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
```

```
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping("/")
    public String home() {
        return "home";
    }

    @GetMapping("/users")
    public String getUsers(
            @RequestParam(defaultValue = "1") int page,
            Model model) {
        int pageSize = 2; // Number of users per page
        Page<User> userPage = userService.getPaginatedUsers(page - 1, pageSize);

        model.addAttribute("users", userPage.getContent());
        model.addAttribute("currentPage", page);
        model.addAttribute("totalPages", userPage.getTotalPages());
        model.addAttribute("pageTitle", "User List");
        return "users";
    }


    @GetMapping("/listusers")
    public String listUsers(Model model, @RequestParam(defaultValue = "1") int page) {
        Page<User> usersPage = userService.findPaginated(page, 10);  // 10 users per page
        model.addAttribute("users", usersPage.getContent());
        model.addAttribute("currentPage", page);
        model.addAttribute("totalPages", usersPage.getTotalPages());
        return "userlist";  // Thymeleaf view
    }


    @GetMapping("/new")
    public String showCreateForm(Model model) {
        model.addAttribute("pageTitle", "Add User");
        model.addAttribute("user", new User());
        return "userform";
    }

    @PostMapping("/save")
    public String saveUser(
            @Valid @ModelAttribute("user") User user,
            BindingResult result,
            Model model) {
```

```
        // Check for validation errors
        if (result.hasErrors()) {
            model.addAttribute("pageTitle", "Add User");
            return "userform"; // Return to the form with errors
        }

        // Save the user
        userService.saveUser(user);

        // Redirect to the user list page after successful save
        return "redirect:/users";
    }
    @PostMapping("/deleteUser/{id}")
    public String deleteUser(@PathVariable Long id, Model model) {
        userService.deleteUserById(id);
        model.addAttribute("message", "User deleted successfully!");
        return "redirect:/users";
    }

    @GetMapping("/users/edit/{id}")
    public String showUpdateForm(@PathVariable Long id, Model model) {
        User user = userService.findById(id);  // Fetch user from the database
        model.addAttribute("user", user);  // Add user data to the model
        return "userupdate";  // Return the userform.html page for updating
    }
 // Method to handle the update form submission
    @PostMapping("/users/update/{id}")
    public String updateUser(@PathVariable Long id, @Valid @ModelAttribute("user") User user, BindingResult
result, Model model) {
        if (result.hasErrors()) {
            model.addAttribute("pageTitle", "Update User");
            return "userupdate";  // If there are validation errors, return to the form
        }
        userService.updateUser(id, user);  // Call the service to update the user
        return "redirect:/users";  // Redirect to the list of users after updating
    }
}
```

**ResourceNotFoundException.java**

```
package com.MIT.FIS10.exception;

public class ResourceNotFoundException extends RuntimeException {

    /**
        *
        */
       private static final long serialVersionUID = 1L;

       public ResourceNotFoundException(String message) {
         super(message);
```

```
    }

    public ResourceNotFoundException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

**PageTable.java**

```
package com.MIT.FIS10.service;

import java.util.List;

import org.springframework.data.domain.Page;

public class PageTable<T> {

    private List<T> content;     // List of items on the current page
    private int currentPage;     // Current page number
    private int totalPages;      // Total number of pages
    private long totalItems;     // Total number of items (users)

    // Constructor
    public PageTable(Page<T> page) {
        this.content = page.getContent();  // Get content for the current page
        this.currentPage = page.getNumber() + 1; // Spring data pages are 0-indexed, so add 1 for display
        this.totalPages = page.getTotalPages();
        this.totalItems = page.getTotalElements();
    }

    // Getters and Setters
    public List<T> getContent() {
        return content;
    }

    public void setContent(List<T> content) {
        this.content = content;
    }

    public int getCurrentPage() {
        return currentPage;
    }

    public void setCurrentPage(int currentPage) {
        this.currentPage = currentPage;
    }

    public int getTotalPages() {
```

```
        return totalPages;
    }

    public void setTotalPages(int totalPages) {
        this.totalPages = totalPages;
    }

    public long getTotalItems() {
        return totalItems;
    }

    public void setTotalItems(long totalItems) {
        this.totalItems = totalItems;
    }
}
```

## GlobalExceptionHandler.java

```java
package com.MIT.FIS10.controlleradvice;

import com.MIT.FIS10.exception.ResourceNotFoundException;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;

@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public String handleResourceNotFoundException(ResourceNotFoundException ex, Model model)
{
        model.addAttribute("errorMessage", ex.getMessage());
        return "error"; // You can redirect to a custom error page
    }

    @ExceptionHandler(Exception.class)
    public String handleGeneralException(Exception ex, Model model) {
        model.addAttribute("errorMessage", "An error occurred: " + ex.getMessage());
        return "error";
    }
}
```

## Layout.html

```
<!DOCTYPE html>
```

```
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/web/thymeleaf/layout">
<head>
       <link rel="stylesheet"
th:href="@{https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css}">
       <script th:src="@{https://code.jquery.com/jquery-3.5.1.slim.min.js}"></script>
       <script
th:src="@{https://cdn.jsdelivr.net/npm/bootstrap@4.5.2/dist/js/bootstrap.bundle.min.js}"></sc
ript>

    <meta charset="UTF-8">
    <title layout:title-pattern="${pageTitle} - My App">My App</title>
    <link rel="stylesheet"
th:href="@{https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css}">
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
            <a class="navbar-brand" href="/" th:href="@{/}">My App</a>
            <button class="navbar-toggler" type="button" data-toggle="collapse" data-
target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle
navigation">
                <span class="navbar-toggler-icon"></span>
            </button>
            <div class="collapse navbar-collapse" id="navbarNav">
                <ul class="navbar-nav">
                    <li class="nav-item">
                        <a class="nav-link" th:href="@{/}">Home</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" th:href="@{/users}">Users</a>
                    </li>
                                <li class="nav-item">
                                    <a class="nav-link" th:href="@{/new}">New</a>
                                </li>
                </ul>
            </div>
        </nav>
    </header>
    <main class="container mt-4">
        <section layout:fragment="content"></section>
    </main>
    <footer class="text-center mt-5">
        <p>&copy; 2024 My App</p>
    </footer>
</body>
</html>
```

## Userform.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/web/thymeleaf/layout"
      layout:decorate="layout.html">
<body>
    <section layout:fragment="content">
        <h1>Add User</h1>
        <form th:action="@{/save}" th:object="${user}" method="post">
            <div class="form-group">
                <label for="id">Id</label>
                <input type="text" class="form-control" id="id" name="id" th:field="*{id}"
disabled="true">
                <div class="text-danger" th:if="${#fields.hasErrors('id')}"
th:errors="*{id}"></div>
            </div>
            <div class="form-group">
                <label for="name">Name</label>
                <input type="text" class="form-control" id="name" name="name"
th:field="*{name}">
                <div class="text-danger" th:if="${#fields.hasErrors('name')}"
th:errors="*{name}"></div>
            </div>
            <div class="form-group">
                <label for="email">Email</label>
                <input type="email" class="form-control" id="email" name="email"
th:field="*{email}">
                <div class="text-danger" th:if="${#fields.hasErrors('email')}"
th:errors="*{email}"></div>
            </div>
            <button type="submit" class="btn btn-primary">Save</button>
        </form>

    </section>
</body>
</html>
```

## Users.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org" layout:decorate="layout">
<head>
    <meta charset="UTF-8">
    <title>User List</title>
</head>
<body>
    <div layout:fragment="content">
        <h2>User List</h2>
        <table class="table table-striped">
            <thead>
                <tr>
                    <th>ID</th>
                    <th>Name</th>
```

```
                    <th>Email</th>
                    <th>Actions</th>  <!-- Added a column for actions like Delete -->
                </tr>
            </thead>
            <tbody>
                <tr th:each="user : ${users}">
                    <td th:text="${user.id}"></td>
                    <td th:text="${user.name}"></td>
                    <td th:text="${user.email}"></td>
                    <td>
                                        <a th:href="@{/users/edit/{id}(id=${user.id})}"
class="btn btn-warning">Edit</a>
                        <form th:action="@{/deleteUser/{id}(id=${user.id})}" method="post">
                            <button type="submit" onclick="return confirm('Are you sure you
want to delete this user?')">Delete</button>
                        </form>
                    </td>
                </tr>
            </tbody>
        </table>
        <nav>
            <ul class="pagination">
                <li class="page-item" th:if="${currentPage > 1}">
                    <a class="page-link" th:href="@{'/users?page=' + ${currentPage -
1}}">Previous</a>
                </li>
                <li class="page-item" th:each="i : ${#numbers.sequence(1, totalPages)}"
                    th:classappend="${i == currentPage} ? 'active'">
                    <a class="page-link" th:href="@{'/users?page=' + ${i}}"
th:text="${i}"></a>
                </li>
                <li class="page-item" th:if="${currentPage < totalPages}">
                    <a class="page-link" th:href="@{'/users?page=' + ${currentPage +
1}}">Next</a>
                </li>
            </ul>
        </nav>
    </div>
</body>
</html>
```

## Userupdate.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org" layout:decorate="layout">
<head>
    <meta charset="UTF-8">
    <title th:text="${pageTitle}">User Form</title>
</head>
<body>
    <div layout:fragment="content">
        <h1 th:text="${pageTitle}">Add User</h1>
```

```
        <form th:action="@{/users/update/{id}(id=${user.id})}" th:object="${user}"
method="post">
            <div class="form-group">
                <label for="id">Id</label>
                <input type="text" class="form-control" id="id" name="id" th:field="*{id}"
disabled>
            </div>
            <div class="form-group">
                <label for="name">Name</label>
                <input type="text" class="form-control" id="name" name="name"
th:field="*{name}">
                <div class="text-danger" th:if="${#fields.hasErrors('name')}"
th:errors="*{name}"></div>
            </div>
            <div class="form-group">
                <label for="email">Email</label>
                <input type="email" class="form-control" id="email" name="email"
th:field="*{email}">
                <div class="text-danger" th:if="${#fields.hasErrors('email')}"
th:errors="*{email}"></div>
            </div>
            <button type="submit" class="btn btn-primary">Update</button>
        </form>
    </div>
</body>
</html>
```

## Error.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Error</title>
</head>
<body>
    <h1>Error</h1>
    <p th:text="${errorMessage}"></p>
    <a href="/users">Back to Users List</a>
</body>
</html>
```

### 1. Overview

The combination of **Spring Boot** and **Angular** creates a full-stack web development framework. **Spring Boot** serves as the backend, handling business logic and server-side operations, while **Angular** is used for the frontend, managing the user interface and user experience.

---

### 2. Spring Boot as Backend

- **Definition**: Spring Boot is an extension of the Spring Framework, providing a simplified and efficient way to build production-ready applications.
- **Key Features**:
  - **Auto-configuration**: Automatically configures application components based on dependencies in the project.
  - **Embedded Servers**: Comes with built-in servers like Tomcat or Jetty for running applications without requiring an external setup.
  - **RESTful APIs**: Enables easy creation of REST APIs to handle client requests.
  - **Dependency Management**: Integrates with tools like Maven or Gradle for managing dependencies.
  - **Scalability and Modularity**: Allows developers to build modular and scalable applications.
- **Responsibilities**:
  - Handles business logic and data processing.
  - Manages database interactions using tools like JPA, Hibernate, or JDBC.
  - Provides APIs for the frontend to interact with the server and database.

---

## 3. Angular as Frontend

- **Definition**: Angular is a TypeScript-based framework developed by Google, used for building dynamic, responsive, and interactive web applications.
- **Key Features**:
  - **Component-Based Architecture**: Applications are built as a collection of reusable components.
  - **Two-Way Data Binding**: Synchronizes data between the UI and the model in real time.
  - **Dependency Injection**: Facilitates the management of application services and their dependencies.
  - **Directives**: Enhances HTML with additional functionalities.
  - **Routing**: Supports navigation between views without requiring a full page reload.
  - **Responsive Design**: Integrates well with CSS frameworks to create responsive layouts.
- **Responsibilities**:
  - Renders the user interface and handles user interactions.
  - Communicates with the backend via HTTP requests to fetch or send data.
  - Ensures a seamless and user-friendly experience.

---

## 4. Integration of Spring Boot and Angular

- **Communication**:
  - Spring Boot exposes RESTful APIs using endpoints (e.g., `/api/users`).
  - Angular uses HTTP client modules to send requests to these APIs and receive responses.
- **Separation of Concerns**:
  - Backend (Spring Boot): Focuses on server-side tasks, such as data storage, authentication, and business logic.
  - Frontend (Angular): Handles client-side tasks, such as data presentation and user interaction.
- **Deployment**:
  - Angular builds the frontend as static files (HTML, CSS, and JavaScript).
  - These files can be served from a dedicated web server (like NGINX) or integrated into the Spring Boot application.

---

## 5. Advantages of Using Spring Boot and Angular Together

1. **Decoupled Architecture**:

    - Backend and frontend are independent, enabling parallel development.
    - Each layer can be scaled separately based on requirements.
2. **Rich Ecosystem**:
    - Spring Boot integrates with various libraries for robust backend functionalities.
    - Angular offers a modern framework for building dynamic, SPA (Single Page Applications).
3. **Efficient Communication**:
    - RESTful APIs ensure standardized communication between the client and server.
4. **Enhanced Productivity**:
    - Angular's component-based design and Spring Boot's auto-configuration simplify development.
5. **Flexibility**:
    - Allows developers to choose the best tools or frameworks for each layer of the stack.

---

**6. Use Cases**

This combination is well-suited for:

- Enterprise-grade applications.
- E-commerce platforms.
- Dashboard and analytics systems.
- Single Page Applications (SPAs) requiring dynamic user interfaces.

## User.java

```
package com.pka.demo.model;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String email;

    // Constructors
    public User() {}

    public User(Long id, String name, String email) {
        this.id = id;
        this.name = name;
        this.email = email;
    }
```

```
    // Getters and Setters
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

**UserRepository.java**

```
package com.pka.demo.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.pka.demo.model.User;

public interface UserRepository extends JpaRepository<User, Long> {}
```

**UserController.java**

```
package com.pka.demo.controller;

import com.pka.demo.model.User;
import com.pka.demo.repository.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
```

```
import java.util.List;

@RestController
@RequestMapping("/api/users")
public class UserController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping
    public List<User> getAllUsers() {
        return userRepository.findAll();
    }

    @PostMapping
    public ResponseEntity<User> createUser(@RequestBody User user) {
        User savedUser = userRepository.save(user);
        return ResponseEntity.ok(savedUser);
    }
}
```

**WebConfig.java**

```
package com.pka.demo.config;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.CorsRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**")
                .allowedOrigins("http://localhost:4200")  // Allow requests from Angular frontend
                .allowedMethods("GET", "POST", "PUT", "DELETE");  // Specify allowed HTTP methods
    }
}
```

**Application.Properties**

```
spring.application.name=demo
spring.datasource.url=jdbc:mysql://localhost:3306/FIS
spring.datasource.username=PKA
spring.datasource.password=PKA
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
```

Angular Code

App.component.ts

```
import { Component } from '@angular/core';
import { UserListComponent } from "./user/user-list/user-list.component";
import { CommonModule } from '@angular/common';
import { provideHttpClient } from '@angular/common/http';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  imports: [UserListComponent, CommonModule],
  providers: []
})
export class AppComponent {
  title = 'User Management';
}
```

App.component.html

```
div class="container">
  <h1>{{ title }}</h1>
  <app-user-list></app-user-list>
</div>
```

App.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { provideHttpClient } from '@angular/common/http';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';
import { UserListComponent } from './user/user-list/user-list.component';

@NgModule({
  declarations: [ ],
  imports: [BrowserModule,FormsModule,AppComponent,UserListComponent],
  providers: [provideHttpClient() ],
  bootstrap: [],
})
export class AppModule {}
```

main.ts

```
import { bootstrapApplication } from '@angular/platform-browser';
import { appConfig } from './app/app.config';
import { AppComponent } from './app/app.component';




import { provideHttpClient } from '@angular/common/http';


bootstrapApplication(AppComponent, {
  providers: [
    provideHttpClient(), // Include the HTTP client
  ],
}).catch((err) => console.error(err));
```

User.services.ts

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root',
})
export class UserService {
  private apiUrl = 'http://localhost:8080/api/users'; // Replace with your Spring Boot
API URL

  constructor(private http: HttpClient) {}

  // Fetch all users
  getUsers(): Observable<any[]> {
    return this.http.get<any[]>(this.apiUrl);
  }

  // Fetch a user by ID
  getUserById(id: number): Observable<any> {
    return this.http.get<any>(`${this.apiUrl}/${id}`);
  }

  // Create a new user
  createUser(user: any): Observable<any> {
    return this.http.post<any>(this.apiUrl, user);
```

```
  }

  // Update an existing user
  updateUser(id: number, user: any): Observable<any> {
    return this.http.put<any>(`${this.apiUrl}/${id}`, user);
  }

  // Delete a user
  deleteUser(id: number): Observable<any> {
    return this.http.delete<any>(`${this.apiUrl}/${id}`);
  }
}
```

User-list.component.ts

```
import { Component, OnInit } from '@angular/core';
import { UserService } from '../user.service';
import { User } from '../user.model';
import { FormsModule } from '@angular/forms';
import { CommonModule } from '@angular/common';
@Component({
  selector: 'app-user-list',
  standalone: true,
  imports: [FormsModule,CommonModule],
  templateUrl: './user-list.component.html',
  styleUrls: ['./user-list.component.css']
})
export class UserListComponent implements OnInit {
  users: User[] = [];
  userForm: User = { id: 0, name: '', email: '' };
  isEditing = false;

  constructor(private userService: UserService) {}

  ngOnInit(): void {
    this.loadUsers();
  }

  loadUsers(): void {
    this.userService.getUsers().subscribe(
      (data) => {
        this.users = data;
      },
      (error) => {
        console.error('Error fetching users:', error);
```

```
      // Display an error message to the user, e.g., using a MatSnackBar
    }
  );
}

saveUser(): void {
  if (this.isEditing) {
    this.userService.updateUser(this.userForm).subscribe(
      () => {
        this.loadUsers();
        this.resetForm();
      },
      (error) => {
        console.error('Error updating user:', error);
        // Display an error message to the user
      }
    );
  } else {
    this.userService.addUser(this.userForm).subscribe(
      () => {
        this.loadUsers();
        this.resetForm();
      },
      (error) => {
        console.error('Error adding user:', error);
        // Display an error message to the user
      }
    );
  }
}

editUser(user: User): void {
  this.userForm = { ...user };
  this.isEditing = true;
}

deleteUser(id: number): void {
  this.userService.deleteUser(id).subscribe(
    () => {
      this.loadUsers();
    },
    (error) => {
      console.error('Error deleting user:', error);
      // Display an error message to the user
    }
  );
```

```
  }

  resetForm(): void {
    this.userForm = { id: 0, name: '', email: '' };
    this.isEditing = false;
  }
}
```

User-list.component.css

```
.container {
  margin: 20px auto;
  max-width: 600px;
}

form {
  margin-bottom: 20px;
}

table {
  width: 100%;
  border-collapse: collapse;
}

th, td {
  border: 1px solid #ddd;
  padding: 8px;
}

th {
  background-color: #f4f4f4;
}
```

User-list.component.html

```
<div class="container">
  <h2>User Management</h2>

  <!-- User Form -->
  <form (submit)="saveUser()">
    <div>
      <label for="name">Name:</label>
      <input type="text" id="name" [(ngModel)]="userForm.name" name="name" required />
    </div>
    <div>
```

```
      <label for="email">Email:</label>
      <input type="email" id="email" [(ngModel)]="userForm.email" name="email" required
/>
    </div>
    <button type="submit">{{ isEditing ? 'Update' : 'Add' }} User</button>
    <button type="button" (click)="resetForm()">Cancel</button>
  </form>

  <!-- User List -->
  <table>
    <thead>
      <tr>
        <th>ID</th>
        <th>Name</th>
        <th>Email</th>
        <th>Actions</th>
      </tr>
    </thead>
    <tbody>
      <tr *ngFor="let user of users">
        <td>{{ user.id }}</td>
        <td>{{ user.name }}</td>
        <td>{{ user.email }}</td>
        <td>
          <button (click)="editUser(user)">Edit</button>
          <button (click)="deleteUser(user.id)">Delete</button>
        </td>
      </tr>
    </tbody>
  </table>
</div>
```

Lab Exercise

1. Design UI Screens for
   * Landing Page
   * Customer Detail Screen
   * Customer Name Screen
   * Customer Proof Of Identity
   * Customer Contact Details
   * Customer Address

**Lab No 9:** **Date:**

# Bulk Customer Generation

## 1. Overview

Bulk customer generation refers to the process of creating or importing multiple customer records at once, usually for initializing databases, testing systems, or onboarding large sets of users. This approach is crucial in scenarios where managing large datasets manually would be inefficient.

---

## 2. Objectives

- **Efficiency**: Automates the creation of numerous customer records to save time and reduce manual effort.
- **Consistency**: Ensures that all records adhere to predefined formats and validation rules.
- **Scalability**: Handles large volumes of data without degrading system performance.
- **Integration**: Allows seamless insertion of data into existing databases or systems.

---

## 3. Use Cases

- **Database Seeding**: Populating databases with test or sample customer data during development or testing phases.
- **Customer Onboarding**: Adding records for large groups of customers, such as during organizational migrations.
- **System Testing**: Stress testing systems with bulk data to evaluate performance under high load.
- **Data Import**: Transferring customer data from legacy systems to new platforms.

---

## 4. Methods of Bulk Customer Generation

1. **Programmatic Generation**:
    - Uses scripts or programs to generate customer records with randomly or algorithmically determined attributes.
    - Common tools/languages: Python, Java, or Node.js.
2. **CSV/Excel Upload**:
    - Accepts customer data in standardized file formats like CSV or Excel.
    - Data is parsed and validated before insertion into the system.
3. **Database Queries**:
    - Executes SQL scripts or stored procedures to insert multiple records directly into the database.
4. **API Integration**:
    - Exposes REST or SOAP APIs for external systems to submit bulk customer data programmatically.

---

## 5. Key Components

1. **Customer Data Schema**:
    - Defines the structure and attributes of customer records (e.g., name, email, phone number, address).
    - Includes validation rules like mandatory fields and format constraints.
2. **Data Generation Logic**:

    - Implements algorithms to populate customer attributes (e.g., random name generators, phone number formats).
3. **Validation Mechanism**:
    - Ensures the integrity and correctness of data.
    - Checks for duplicates, required fields, and adherence to business rules.
4. **Batch Processing**:
    - Processes data in chunks or batches to improve performance and prevent memory overload.
    - Handles errors gracefully by logging or skipping problematic records.
5. **Storage and Retrieval**:
    - Stores generated data in relational databases (e.g., MySQL, PostgreSQL) or NoSQL databases (e.g., MongoDB).
    - Supports efficient querying and retrieval.

---

## 6. Challenges

- **Data Validation**:
    - Ensuring that the generated data meets all business rules and avoids duplicates.
- **Performance**:
    - Handling large volumes of data without affecting system speed or stability.
- **Error Handling**:
    - Managing partial failures during batch processing.
- **Security**:
    - Protecting sensitive customer data during and after generation.

---

## 7. Best Practices

1. **Use Realistic Data**:
    - Generate data that resembles actual customer information to ensure meaningful testing.
2. **Batch Size Management**:
    - Adjust batch sizes to balance performance and resource usage.
3. **Validation Layers**:
    - Include multiple layers of validation to catch errors early.
4. **Error Logging**:
    - Implement robust logging mechanisms to track and troubleshoot issues during data generation.
5. **Backup Mechanisms**:
    - Create backups of generated data to prevent data loss or corruption.

---

## 8. Tools and Technologies

- **Programming Languages**: Python (Faker library), Java (Java Faker), JavaScript (Faker.js).
- **Data Formats**: CSV, JSON, XML.
- **Databases**: MySQL, PostgreSQL, MongoDB.
- **APIs**: REST or GraphQL for integration.
- **ETL Tools**: Tools like Apache Nifi, Talend, or Informatica for bulk data processing.

User.java

```
package com.example.demo.entity;

import jakarta.persistence.*;
import lombok.*;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class User {
 public User() {   }
public User(Long userId, String name, String email) {
        this.Id = userId;
        this.name = name;
        this.email = email;
        this.user_id= 0L;
    }
    public Long getUserId() {return Id;}
public void setUserId(Long userId) {this.Id = userId;}
public String getName() {return name;}
public void setName(String name) {this.name = name;}
public String getEmail() {return email;}
public void setEmail(String email) {this.email = email;}
public Long getUser_id() {return user_id;}


public void setUser_id(Long user_id) {this.user_id = user_id;}
@Id
    private Long Id; // Using user ID as a primary key
private Long user_id;
    private String name;
    private String email;
}
```

**UserRepository.java**

```
package com.example.demo.repository;
```

```
import org.springframework.data.jpa.repository.JpaRepository;

import com.example.demo.entity.User;

public interface UserRepository extends JpaRepository<User, Long> {
}
```

UserService.java

```
package com.example.demo.service;

import org.apache.poi.ss.usermodel.*;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.web.multipart.MultipartFile;

import com.example.demo.entity.User;
import com.example.demo.repository.UserRepository;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

@Service
public class UserService {
    private final UserRepository userRepository;
    private final Logger logger = LoggerFactory.getLogger(UserService.class);

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @Transactional
```

```
    public List<String> processFile(MultipartFile file) {
        List<String> errorLogs = new ArrayList<>();

        try (Workbook workbook = new XSSFWorkbook(file.getInputStream())) {
            Sheet sheet = workbook.getSheetAt(0);
            for (int i = 1; i <= sheet.getLastRowNum(); i++) { // Skip header row
                Row row = sheet.getRow(i);

                try {
                    Long userId = (long) row.getCell(0).getNumericCellValue(); // Assuming userId is numeric
                    String name = row.getCell(1).getStringCellValue();
                    String email = row.getCell(2).getStringCellValue();

                    User user = new User(userId, name, email);
                    userRepository.save(user);
                } catch (Exception e) {
                    logger.error("Error processing row {}: {}", i, e.getMessage());
                    errorLogs.add("Row " + i + ": " + e.getMessage());
                }
            }
        } catch (IOException e) {
            logger.error("Error reading file: {}", e.getMessage());
            errorLogs.add("File error: " + e.getMessage());
        }

        return errorLogs;
    }
}
```

UserController.java

```
package com.example.demo.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.multipart.MultipartFile;

import com.example.demo.service.UserService;
```

```java
import java.util.List;

@RestController
@RequestMapping("/users")
public class UserController {
    @Autowired
    private UserService userService;

    @PostMapping("/upload")
    public ResponseEntity<?> uploadFile(@RequestParam("file") MultipartFile file) {
        List<String> errorLogs = userService.processFile(file);

        if (errorLogs.isEmpty()) {
            return ResponseEntity.ok("File processed successfully!");
        } else {
            return ResponseEntity.status(207).body(errorLogs); // Partial success
        }
    }
}
```

GlobalExceptionHandler.java

```
package com.example.demo.exception;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;

@ControllerAdvice
public class GlobalExceptionHandler {
   private final Logger logger = LoggerFactory.getLogger(GlobalExceptionHandler.class);

   @ExceptionHandler(Exception.class)
   public ResponseEntity<String> handleException(Exception e) {
      logger.error("Unhandled exception: {}", e.getMessage());
      return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
            .body("An unexpected error occurred: " + e.getMessage());
   }
```

}

Application.properties

```
spring.application.name=demo
# MySQL Configuration
spring.datasource.url=jdbc:mysql://localhost:3306/FIS?useSSL=false&serverTimezone=UTC
spring.datasource.username=PKA
spring.datasource.password=PKA
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# JPA Settings
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.generate_statistics=true



logging.level.org.springframework=INFO
logging.level.com.yourpackage=DEBUG
logging.file.name=application.log


logging.level.org.springframework.transaction=DEBUG
logging.level.org.springframework.orm.jpa=DEBUG
```

**Exercise**

1. **Generate Bulk customers using \customers api**
2. **Test the api using POST method in POSTMAN**

**Lab No 10:** **Date:**

# Report Generation using Python

**Objective:**

In this lab, the students will be able to understand

* Importance of Reports for Financial institutions and regulations
* Learn about report generation using Python
* Generate report as CSV file

**Prerequisites:**

1. MySQL installed and running
2. Python installed

Reports are vital for financial institutions for several key reasons:

1. **Regulatory Compliance**: Financial institutions must comply with various regulations and standards. Regular reporting ensures they meet these legal requirements, avoiding penalties and maintaining their licenses to operate.
2. **Risk Management**: Reports help in identifying, assessing, and mitigating risks. By analyzing data, institutions can predict potential issues and take proactive measures to manage them.
3. **Decision Making**: Accurate and timely reports provide valuable insights that aid in strategic decision-making. They help management understand the financial health of the institution and make informed choices about investments, lending, and other critical operations.
4. **Transparency and Trust**: Regular reporting fosters transparency, which is crucial for maintaining trust with stakeholders, including customers, investors, and regulators. It demonstrates the institution's commitment to ethical practices and sound financial management.
5. **Performance Monitoring**: Reports allow institutions to track their performance over time. They can compare current data with historical data to identify trends, measure progress, and set future goals.
6. **Operational Efficiency**: Detailed reports can highlight areas where the institution can improve efficiency, reduce costs, and optimize operations. This can lead to better resource allocation and improved profitability.
7. **Investor Relations**: For publicly traded financial institutions, regular reporting is essential to keep investors informed about the institution's financial status and future prospects. This helps in maintaining investor confidence and can impact the institution's stock price.

Here are the steps to connect to a MySQL database using Python, fetch data from the student table, and store it in a CSV file:

1. **Install Required Libraries**:

   You need the mysql-connector-python and pandas libraries. Install them using pip command.
   *pip install mysql-connector-python pandas*
2. **Connect to MySQL Database**:

   Use the mysql.connector library to establish a connection to your MySQL database.

```
import mysql.connector
import pandas as pd

# Establish a connection to the MySQL database
conn = mysql.connector.connect(
    host='your_host_name',
    user='your_username',
    password='your_password',
    database='your_database_name'
)
```

**Note:** Replace your_host_name, your_username, your_password, and your_database_name with your actual MySQL database credentials.

3. **Fetch Data from student Table**:

Execute a SQL query to fetch data from the student table. (Sample table it is)

```
query = "SELECT * FROM student"

# Read data into a pandas DataFrame
df = pd.read_sql(query, conn)

# Close the database connection
conn.close()
```

4. **Store Data in a CSV File**:

Use the pandas library to read the data into a DataFrame and then save it as a CSV file.

```
df.to_csv('student_report.csv', index=False)
print("Data fetched from 'student' table and saved as 'student_report.csv'.")
```

5. Execute the Python script. It will connect to your MySQL database, fetch the data from the student table, and save it as a CSV file named student_report.csv.

**Lab Exercises:**

1. Fetch the customer details based on their language and age group.
2. Calculate the ratio of male to female customers and store the data in a CSV file.
3. Analyse the distribution of languages spoken by customers in different countries and store the data in a CSV file.

**Lab No 11:** **Date:**

# Data Visualisation

Data visualization is the graphical representation of data to help people understand complex data sets through visual elements like charts, graphs, and maps. It makes it easier to identify patterns, trends, and outliers. Here are some common types of data visualizations:

1. Line Charts: Show trends over time.
2. Bar Charts: Compare quantities across different categories.
3. Pie Charts: Display proportions of a whole.

Using the matplotlib library, various types of plots can be created.

**Step 1:**

- Install the library using the command

*pip install pandas matplotlib*

**Step 2:**

- Read the data from the CSV file into a pandas DataFrame.

```
import pandas as pd
import matplotlib.pyplot as plt
df = pd.read_csv('your_data.csv')
```

**Step 3:**

- Draw the chart on data for visualisation.

Bar Chart

```
plt.figure(figsize=(10, 6))
df['your_column'].value_counts().plot(kind='bar')
plt.title('Bar Chart of Your Column')
plt.xlabel('Categories')
plt.ylabel('Frequency')
plt.savefig('bar_chart.png')
plt.show()
```

**Lab Exercise:**

1. Create a bar chart to visualize what are the most common address types (CustomerAddressType) used by customers? (Bar chart showing the count for each address type).
2. How many customer IDs or contact information records have expired based on the `EndDate` field? Visualization: Bar chart showing active vs. expired records.
3. Which languages (`CustomerPreferredLanguage`) are most preferred by customers? Visualization: Horizontal bar chart showing the count of customers for each language.

**Lab No 12:** **Date:**

# Hosting the Application

## 1 - Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article, without having to install anything on your local environment.

To start Azure Cloud Shell:

Expand table

| Option | Example/Link |
| --- | --- |
| Select **Try It** in the upper-right corner of a code or command block. Selecting **Try It** doesn't automatically copy the code or command to Cloud Shell. |   |
| Go to https://shell.azure.com, or select the **Launch Cloud Shell** button to open Cloud Shell in your browser. |   |
| Select the **Cloud Shell** button on the menu bar at the upper right in the Azure portal. |  |

To use Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block (or command block) to copy the code or command.
3. Paste the code or command into the Cloud Shell session by selecting **Ctrl**+**Shift**+**V** on Windows and Linux, or by selecting **Cmd**+**Shift**+**V** on macOS.
4. Select **Enter** to run the code or command.

## 2 - Get the sample app

- Spring Boot
- Embedded Tomcat
- Quarkus

1. Download and extract the default Spring Boot web application template. This repository is cloned for you when you run the Spring CLI command `spring boot new my-webapp`.

   BashCopy
   ```
   git clone https://github.com/rd-1-2022/rest-service my-webapp
   ```

2. Change your working directory to the project folder:

   Azure CLICopy
   Open Cloud Shell
   ```
   cd my-webapp
   ```

## 3 - Configure the Maven plugin

The deployment process to Azure App Service uses your Azure credentials from the Azure CLI automatically. If the Azure CLI isn't installed locally, then the Maven plugin authenticates with OAuth or device sign-in. For more information, see authentication with Maven plugins.

Run the Maven command shown next to configure the deployment. This command helps you to set up the App Service operating system, Java version, and Tomcat version.

Azure CLICopy
Open Cloud Shell
```
mvn com.microsoft.azure:azure-webapp-maven-plugin:2.13.0:config
```

1. For **Create new run configuration**, type **Y**, then **Enter**.
2. For **Define value for OS**, type **2** for Linux, then **Enter**.
3. For **Define value for javaVersion**, type **1** for Java 17, then **Enter**.
4. For **Define value for pricingTier**, type **3** for P1v2, then **Enter**.
5. For **Confirm**, type **Y**, then **Enter**.

   Copy
   ```
   Please confirm webapp properties

   AppName : <generated-app-name>

   ResourceGroup : <generated-app-name>-rg

   Region : centralindia

   PricingTier : F1

   OS : Windows

   Java Version: Java 17
   ```

```
Web server stack: Java SE

Deploy to slot : false

Confirm (Y/N) [Y]: y

[INFO] Saving configuration to pom.

[INFO] ------------------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO] ------------------------------------------------------------------------

[INFO] Total time:  8.139 s

[INFO] Finished at: 2023-07-26T12:42:48Z

[INFO] ------------------------------------------------------------------------
```

After you confirm your choices, the plugin adds the above plugin element and requisite settings to your project's `pom.xml` file that configure your web app to run in Azure App Service.

The relevant portion of the `pom.xml` file should look similar to the following example.

XMLCopy

```xml
<build>
    <plugins>
        <plugin>
            <groupId>com.microsoft.azure</groupId>
            <artifactId>>azure-webapp-maven-plugin</artifactId>
            <version>x.xx.x</version>
            <configuration>
                <schemaVersion>v2</schemaVersion>
                <resourceGroup>your-resourcegroup-name</resourceGroup>
                <appName>your-app-name</appName>
            ...
            </configuration>
        </plugin>
    </plugins>
</build>
```

You can modify the configurations for App Service directly in your `pom.xml`. Some common configurations are listed in the following table:

Expand table

| Property | Required | Description | Version |
|---|---|---|---|
| `<schemaVersion>` | false | Specify the version of the configuration schema. Supported values are: `v1`, `v2`. | 1.5.2 |
| `<subscriptionId>` | false | Specify the subscription ID. | 0.1.0+ |
| `<resourceGroup>` | true | Azure Resource Group for your Web App. | 0.1.0+ |
| `<appName>` | true | The name of your Web App. | 0.1.0+ |
| `<region>` | false | Specifies the region to host your Web App; the default value is **centralus**. All valid | 0.1.0+ |

| Property | Required | Description | Version |
|---|---|---|---|
| | | regions at Supported Regions section. | |
| <pricingTier> | false | The pricing tier for your Web App. The default value is **P1v2** for production workload, while **B2** is the recommended minimum for Java dev/test. For more information, see App Service Pricing | 0.1.0+ |
| <runtime> | false | The runtime environment configuration. For more information, see Configuration Details. | 0.1.0+ |
| <deployment> | false | The deployment configuration. For more information, see Configuration Details. | 0.1.0+ |

For the complete list of configurations, see the plugin reference documentation. All the Azure Maven Plugins share a common set of configurations. For these configurations see Common Configurations. For configurations specific to App Service, see Azure Web App: Configuration Details.

Be careful about the values of <appName> and <resourceGroup>. They're used later.

## 4 - Deploy the app

With all the configuration ready in your pom.xml file, you can deploy your Java app to Azure with one single command.

1. Build the JAR file using the following command:
   - Spring Boot
   - Embedded Tomcat
   - Quarkus

   BashCopy
   ```
   mvn clean package
   ```

2. Deploy to Azure by using the following command:

   BashCopy
   ```
   mvn azure-webapp:deploy
   ```

   If the deployment succeeds, you see the following output:

   OutputCopy
   ```
   [INFO] Successfully deployed the artifact to https://<app-name>.azurewebsites.net

   [INFO] ------------------------------------------------------------------------

   [INFO] BUILD SUCCESS

   [INFO] ------------------------------------------------------------------------

   [INFO] Total time:  02:20 min
   ```

```
[INFO] Finished at: 2023-07-26T12:47:50Z

[INFO] ------------------------------------------------------------------------
```

- Spring Boot
- Embedded Tomcat
- Quarkus

Once deployment is completed, your application is ready at `http://<appName>.azurewebsites.net/`. Open the URL `http://<appName>.azurewebsites.net/greeting` with your local web browser (note the `/greeting` path), and you should see:

```
{
    "id": 2,
    "content": "Hello, World!"
}
```

**Congratulations!** You deployed your first Java app to App Service.

# 5 - Clean up resources

In the preceding steps, you created Azure resources in a resource group. If you don't need the resources in the future, delete the resource group from portal, or by running the following command in the Cloud Shell:

Azure CLICopy
Open Cloud Shell

```
az group delete --name <your resource group name; for example: quarkus-hello-azure-1690375364238-rg> --yes
```

This command might take a minute to run.

Exercise

1. Host your application to azure app service

## References:


1. Craig Walls, Spring in Action (6th Edition), Manning Publications, 2022.
2. Mark Heckler, Spring Boot: Up & Running, O'Reilly Media, 2021.
3. Ferdinand Malcher, Johannes Hoppe, and Danny Koppenhagen, Angular: From Theory to Practice, Rheinwerk Publishing, 2020.
4. Nate Schutta and Craig Walls, Programming with Spring Boot and Angular, Pragmatic Bookshelf, 2019.
5. Minko Gechev, Switching to Angular (2nd Edition), Packt Publishing,
6. https://github.com/PRAKASH-KALINGRAO-AITHAL