

Logistic Project

Module

Module

=====

- **User Management**
- **Inventory Management**
- **Order Management**
- **Shipment Management**
- **Tracking and Monitoring**
- **Billing and Invoicing**

Database Table

DataBase Table

-
- **Users**
 - **Roles**
 - **Products**
 - **Inventory**
 - **Orders**
 - **OrderItems**
 - **Shipments**
 - **ShipmentTracking**
 - **Invoices**
 - **Payments**

Users

Columns:

1. `user_id` - INT, Primary Key
2. `username` - VARCHAR(50), Not Null
3. `password` - VARCHAR(50), Not Null
4. `email` - VARCHAR(100), Not Null
5. `role_id` - INT, Foreign Key referencing Roles(`role_id`)

Roles

Columns:

1. `role_id` - INT, Primary Key
2. `role_name` - VARCHAR(50), Not Null

Products

Columns:

1. product_id - INT, Primary Key
2. name - VARCHAR(100), Not Null
3. description - TEXT
4. price - DECIMAL(10, 2), Not Null
5. sku - VARCHAR(50), Unique, Not Null

Inventory

Columns:

1. inventory_id - INT, Primary Key
2. product_id - INT, Foreign Key referencing Products (product_id)
3. quantity - INT, Not Null
4. location - VARCHAR(100), Not Null
5. last_updated - DATETIME, Not Null

Orders

Columns:

1. order_id - INT, Primary Key
2. user_id - INT, Foreign Key referencing Users(user_id)
3. order_date - DATETIME, Not Null
4. status - VARCHAR(50), Not Null
5. total_amount - DECIMAL(10, 2), Not Null

OrderItems

Columns:

1. order_item_id - INT, Primary Key
2. order_id - INT, Foreign Key referencing Orders(order_id)

- der_id)
- 3. product_id - INT, Foreign Key referencing Products (product_id)
 - 4. quantity - INT, Not Null
 - 5. price - DECIMAL(10, 2), Not Null

Shipments

Columns:

- 1. shipment_id - INT, Primary Key
- 2. order_id - INT, Foreign Key referencing Orders (order_id)
- 3. shipment_date - DATETIME
- 4. delivery_date - DATETIME
- 5. status - VARCHAR(50), Not Null
- 6. carrier - VARCHAR(50), Not Null
- 7. tracking_number - VARCHAR(50), Unique, Not Null

ShipmentTracking

Columns:

- 1. tracking_id - INT, Primary Key
- 2. shipment_id - INT, Foreign Key referencing Shipments (shipment_id)
- 3. location - VARCHAR(100), Not Null
- 4. status - VARCHAR(50), Not Null
- 5. timestamp - DATETIME, Not Null

Invoices

Columns:

- 1. invoice_id - INT, Primary Key
- 2. order_id - INT, Foreign Key referencing Orders (order_id)

3. amount - DECIMAL(10, 2), Not Null
4. due_date - DATETIME, Not Null
5. status - VARCHAR(50), Not Null
6. issued_date - DATETIME, Not Null

Payments

Columns:

1. payment_id - INT, Primary Key
2. invoice_id - INT, Foreign Key referencing Invoices (invoice_id)
3. amount - DECIMAL(10, 2), Not Null
4. payment_date - DATETIME, Not Null
5. payment_method - VARCHAR(50), Not Null
6. confirmation_number - VARCHAR(50), Unique, Not Null

```
CREATE TABLE Users (
    user_id INT PRIMARY KEY,
    username VARCHAR(50) NOT NULL,
    password VARCHAR(50) NOT NULL,
    email VARCHAR(100) NOT NULL,
    role_id INT,
    FOREIGN KEY (role_id) REFERENCES Roles(role_id)
```

);

```
CREATE TABLE Roles (
    role_id INT PRIMARY KEY,
    role_name VARCHAR(50) NOT NULL
);
```

```
CREATE TABLE Products (
    product_id INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    description TEXT,
    price DECIMAL(10, 2) NOT NULL,
    sku VARCHAR(50) UNIQUE NOT NULL
);
```

```
CREATE TABLE Inventory (
    inventory_id INT PRIMARY KEY,
    product_id INT,
    quantity INT NOT NULL,
    location VARCHAR(100) NOT NULL,
    last_updated DATETIME NOT NULL,
    FOREIGN KEY (product_id) REFERENCES
Products(product_id)
);
```

```
CREATE TABLE Orders (
    order_id INT PRIMARY KEY,
    user_id INT,
    order_date DATETIME NOT NULL,
    status VARCHAR(50) NOT NULL,
    total_amount DECIMAL(10, 2) NOT NULL,
    FOREIGN KEY (user_id) REFERENCES Users(user_id)
```

);

```
CREATE TABLE OrderItems (
    order_item_id INT PRIMARY KEY,
    order_id INT,
    product_id INT,
    quantity INT NOT NULL,
    price DECIMAL(10, 2) NOT NULL,
    FOREIGN KEY (order_id) REFERENCES
Orders(order_id),
    FOREIGN KEY (product_id) REFERENCES
Products(product_id)
);
```

```
CREATE TABLE Shipments (
    shipment_id INT PRIMARY KEY,
    order_id INT,
    shipment_date DATETIME,
    delivery_date DATETIME,
    status VARCHAR(50) NOT NULL,
    carrier VARCHAR(50) NOT NULL,
    tracking_number VARCHAR(50) UNIQUE NOT NULL,
    FOREIGN KEY (order_id) REFERENCES
Orders(order_id)
);
```

```
CREATE TABLE ShipmentTracking (
    tracking_id INT PRIMARY KEY,
    shipment_id INT,
    location VARCHAR(100) NOT NULL,
    status VARCHAR(50) NOT NULL,
    timestamp DATETIME NOT NULL,
```

```
FOREIGN KEY (shipment_id) REFERENCES  
Shipments(shipment_id)  
);
```

```
CREATE TABLE Invoices (  
    invoice_id INT PRIMARY KEY,  
    order_id INT,  
    amount DECIMAL(10, 2) NOT NULL,  
    due_date DATETIME NOT NULL,  
    status VARCHAR(50) NOT NULL,  
    issued_date DATETIME NOT NULL,  
    FOREIGN KEY (order_id) REFERENCES  
Orders(order_id)  
);
```

```
CREATE TABLE Payments (  
    payment_id INT PRIMARY KEY,  
    invoice_id INT,  
    amount DECIMAL(10, 2) NOT NULL,  
    payment_date DATETIME NOT NULL,  
    payment_method VARCHAR(50) NOT NULL,  
    confirmation_number VARCHAR(50) UNIQUE NOT  
NULL,  
    FOREIGN KEY (invoice_id) REFERENCES  
Invoices(invoice_id)  
);
```

Front end

User-Side Navigation

For regular users who are interacting with the logistics system:

1. User Dashboard

- Overview of recent orders, shipment status, and account information.

2. Order Management

- Create new orders, view order history, and track current orders.

3. Shipment Tracking

- Access tracking information for ongoing shipments related to their orders.

- - - - -

Admin-Side Navigation

For administrators who manage the logistics system and user accounts:

1. Admin Dashboard

- Overview of system metrics, recent activities, and important notifications.

2. User Management

- Manage user accounts, roles, permissions, and view user activity logs.

3. Inventory Management

- Add, update, and manage products in the inventory.

4. Order Management

- View all orders, manage order statuses, and handle order-related issues.

3. Shipment Tracking

- Access tracking information for ongoing shipments related to their orders.

Admin-Side Navigation

For administrators who manage the logistics system and user accounts:

1. Admin Dashboard

- Overview of system metrics, recent activities, and important notifications.

2. User Management

- Manage user accounts, roles, permissions, and view user activity logs.

3. Inventory Management

- Add, update, and manage products in the inventory.

3. Shipment Tracking

- Access tracking information for ongoing shipments related to their orders.

Admin-Side Navigation

For administrators who manage the logistics system and user accounts:

1. Admin Dashboard

- Overview of system metrics, recent activities, and important notifications.

2. User Management

- Manage user accounts, roles, permissions, and view user activity logs.

3. Inventory Management

- Add, update, and manage products in the inventory.

- **UI/UX Design:** Design the user interface to be intuitive and user-friendly, guiding users to easily find and navigate through the application's features.
- **Consistency:** Maintain consistent navigation across the application to provide a seamless user experience. Use clear labels and icons to represent different modules and actions.
- **Accessibility:** Ensure that navigation is accessible across different devices and screen sizes, considering responsive design principles.

Example Navigation Structure

Here's a simplified example of how the navigation structure could be represented:

User-Side Navigation

- Dashboard
- Orders
 - View Orders
 - Track Shipments



- Account Settings
 - Profile
 - Change Password
- Logout

Admin-Side Navigation

- Dashboard
- Users
 - Manage Users
 - Roles & Permissions
- Inventory
 - Manage Products
- Orders
 - View Orders
 - Manage Orders



-
- Shipments
 - Create Shipment
 - Manage Shipments
 - Billing
 - Generate Invoices
 - Manage Payments
 - Logout

Class & methods

User Side

1. User Management

- UserController
 - createUser
 - authenticateUser
 - logout

2. Order Management

- OrderController
 - createOrder
 - addOrderItem
 - getOrder



3. Shipment Management

- ShipmentController
 - getShipment
 - getTrackingInfo

Admin Side

1. User Management

- UserController
 - assignRole

2. Inventory Management

- InventoryController
 - addProduct



- updateInventory
- getProduct

3. Shipment Management

- ShipmentController
 - createShipment
 - updateShipmentStatus

4. Tracking and Monitoring

- TrackingController
 - logTrackingInfo

5. Billing and Invoicing

- BillingController
 - createInvoice



- processPayment
- getInvoice

Database Table

Users

Columns:

1. `user_id` - INT, Primary Key
2. `username` - VARCHAR(50), Not Null
3. `password` - VARCHAR(50), Not Null
4. `email` - VARCHAR(100), Not Null
5. `role_id` - INT, Foreign Key referencing `Roles(role_id)`

Roles

Columns:

1. `role_id` - INT, Primary Key
2. `role_name` - VARCHAR(50), Not Null

Products

Columns:

1. `product_id` - INT, Primary Key
2. `name` - VARCHAR(100), Not Null
3. `description` - TEXT
4. `price` - DECIMAL(10, 2), Not Null
5. `sku` - VARCHAR(50), Unique, Not Null

Inventory

Columns:

1. `inventory_id` - INT, Primary Key
2. `product_id` - INT, Foreign Key referencing `Products(product_id)`
3. `quantity` - INT, Not Null
4. `location` - VARCHAR(100), Not Null
5. `last_updated` - DATETIME, Not Null



Orders

Columns:

1. `order_id` - INT, Primary Key
2. `user_id` - INT, Foreign Key referencing `Users(user_id)`
3. `order_date` - DATETIME, Not Null
4. `status` - VARCHAR(50), Not Null
5. `total_amount` - DECIMAL(10, 2), Not Null

OrderItems

Columns:

1. `order_item_id` - INT, Primary Key
2. `order_id` - INT, Foreign Key referencing `Orders(order_id)`
3. `product_id` - INT, Foreign Key referencing `Products(product_id)`
4. `quantity` - INT, Not Null



-
- 5. `price` - DECIMAL(10, 2), Not Null

Shipments

Columns:

- 1. `shipment_id` - INT, Primary Key
- 2. `order_id` - INT, Foreign Key referencing `Orders(order_id)`
- 3. `shipment_date` - DATETIME
- 4. `delivery_date` - DATETIME
- 5. `status` - VARCHAR(50), Not Null
- 6. `carrier` - VARCHAR(50), Not Null
- 7. `tracking_number` - VARCHAR(50), Unique, Not Null

Shipment Tracking

Columns:



-
- 1. `tracking_id` - INT, Primary Key
 - 2. `shipment_id` - INT, Foreign Key referencing `Shipments(shipment_id)`
 - 3. `location` - VARCHAR(100), Not Null
 - 4. `status` - VARCHAR(50), Not Null
 - 5. `timestamp` - DATETIME, Not Null

Invoices

Columns:

- 1. `invoice_id` - INT, Primary Key
- 2. `order_id` - INT, Foreign Key referencing `Orders(order_id)`
- 3. `amount` - DECIMAL(10, 2), Not Null
- 4. `due_date` - DATETIME, Not Null
- 5. `status` - VARCHAR(50), Not Null
- 6. `issued_date` - DATETIME, Not Null



1. `tracking_id` - INT, Primary Key
2. `shipment_id` - INT, Foreign Key referencing `Shipments(shipment_id)`
3. `location` - VARCHAR(100), Not Null
4. `status` - VARCHAR(50), Not Null
5. `timestamp` - DATETIME, Not Null

Invoices

Columns:

1. `invoice_id` - INT, Primary Key
2. `order_id` - INT, Foreign Key referencing `Orders(order_id)`
3. `amount` - DECIMAL(10, 2), Not Null
4. `due_date` - DATETIME, Not Null
5. `status` - VARCHAR(50), Not Null
6. `issued_date` - DATETIME, Not Null



SELVAMANI

Java Developer



Motivated Junior Software Developer with one year of experience in Java, SQL, and web technologies. Currently working on developing robust applications using Spring Boot and REST APIs. Passionate about improving user experience and contributing to team success through effective collaboration and problem-solving.

selvamani42077@gmail.com

+917826980179

Chennai, India

<https://www.linkedin.com/in/selvamani-p-2965a4315>

WORK EXPERIENCE

Highly motivated and results-driven marketing coordinator with a strong background in developing and implementing effective marketing strategies. Skilled in digital marketing, campaign management, and market research. Proven ability to drive brand awareness, engage target audiences, and generate leads. Possesses exceptional organizational and communication skills, with a keen eye for detail. A creative problem solver who thrives in fast-paced environments and enjoys collaborating with cross-functional teams to achieve business objectives.

Education

Thynk Unlimited
Digital Marketing Specialization
2020-2022

Salford & Co. University
Bachelor of Arts in Marketing
2016-2019

Languages

English - Native
Spanish - C1
Japanese - A2

Skills

- Digital Marketing
- Trade Marketing
- Design Thinking
- Branding
- Social Media Marketing

Professional Experience

Rimberio Agency

Marketing Coordinator | Jan 2020 - Recent

- Assisted in the development and execution of marketing campaigns, including email marketing, social media, and advertising campaigns
- Coordinated and executed marketing events, including trade shows, webinars, and product launches
- Managed and maintained social media profiles, including content creation, scheduling, and engagement

Aldenaire & Partners

Marketing Analyst | Dec 2018 - Jan 2020

- Developed and maintained relationships with key industry partners and associations to drive brand awareness and generate leads
- Conducted market research and analyzed data to identify opportunities for growth and optimization
- Managed and maintained the company's social media profiles, including content creation, scheduling, and engagement

Larana, Inc.

Marketing Intern | Dec 2016 - Dec 2018

- Assisted in the development and execution of marketing campaigns, including social media, email marketing, and content marketing
- Conducted market research and analyzed data to identify opportunities for growth and optimization
- Assisted in the creation of marketing materials such as brochures, flyers, and sales presentations

Personal Experience

- Studied abroad in Spain for one semester, expanding cultural knowledge and gaining proficiency in the Spanish language.
- Completed an internship with a local marketing agency in Madrid, assisting with the development and execution of marketing campaigns for clients in various industries.

Class & methods

User Side

1. User Management
 - UserController
 - createUser
 - authenticateUser
 - logout
2. Order Management
 - OrderController
 - createOrder
 - addOrderItem
 - getOrder
3. Shipment Management
 - ShipmentController
 - getShipment
 - getTrackingInfo

Admin Side

1. User Management
 - UserController
 - assignRole
2. Inventory Management
 - InventoryController
 - addProduct

- updateInventory
- getProduct

3. Shipment Management

- ShipmentController
 - createShipment
 - updateShipmentStatus

4. Tracking and Monitoring

- TrackingController
 - logTrackingInfo

5. Billing and Invoicing

- BillingController
 - createInvoice
 - processPayment
 - getInvoice

User-Side Modules (3 modules)

1. User Management

- Handles user authentication, creation, and basic profile management.

2. Order Management

- Allows users to create orders, add items to orders, and view their order history.

3. Shipment Management

- Provides basic tracking information for shipments related to the user's orders.

Admin-Side Modules (5 modules)

1. User Management

- Admins can manage users, assign roles, and potentially deactivate or delete user accounts.

2. Inventory Management

- Admins can add, update, and manage products in the inventory.

3. Order Management

- Admins have additional capabilities to view and manage orders, such as editing or canceling orders.

4. Shipment Management

- Admins can create shipments, update shipment statuses, and view detailed tracking information.

5. Billing and Invoicing

- Admins can generate invoices, process payments, and manage billing-related tasks.

1. Scalability : Ensure that the architecture and design can scale as the volume of users, orders, and transactions increases.
2. Security : Implement robust security measures to protect user data, transactions, and system integrity.
3. Performance : Optimize performance to handle concurrent user interactions, especially during peak times.
4. Error Handling and Logging : Implement thorough error handling and logging mechanisms to facilitate troubleshooting and maintenance.
5. Compliance : Ensure compliance with relevant regulations and standards, such as data protection laws (e.g., GDPR, CCPA).
6. Testing : Conduct comprehensive testing, including unit testing, integration testing, and user acceptance testing, to ensure reliability and functionality.

Additional Classes or Modules

Depending on the specific requirements and complexity of the logistics project, you may need to consider additional classes or modules, such as:

- Customer Service : Handling customer inquiries, support tickets, and feedback.

- Analytics and Reporting : Generating reports on sales, inventory levels, shipment statuses, etc., to support decision-making.
- Integration with External Systems : Integration with ERP systems, shipping carriers, payment gateways, etc., to streamline operations.
- Notifications and Alerts : Implementing notifications for order updates, shipment tracking, payment reminders, etc.
- Localization : Support for multiple languages, currencies, and regional settings if operating internationally.

TestCase

Test Case

=====

1. Boundary Value Analysis (BVA):

Purpose: Tests the behavior of the system at the boundaries of input ranges to ensure it handles edge cases correctly.

2. Equivalence Partitioning (EP):

Purpose: Divides the input space into equivalence classes and tests representative values from each class to ensure consistent behavior within each class.

3. Decision Table Testing:

Purpose: Systematically tests all possible combinations of inputs and conditions to verify that the system makes correct decisions.

4. State Transition Testing:

Purpose: Tests the transitions between different states of the system to ensure it behaves correctly as it moves from one state to another.

5. Error Guessing:

Purpose: Uses past experience and intuition to identify

potential error-prone areas in the code and designs test cases to specifically target these areas.

6. Control Flow Testing:

Purpose: Tests the control flow of the program by exercising different paths through the code to ensure that all statements and branches are executed and behave as expected.

7. Data Flow Testing:

Purpose: Tests the flow of data through the program to ensure that variables are defined, initialized, and used correctly throughout the code.

8. Mutation Testing:

Purpose: Introduces small changes (mutations) to the code to create faulty versions and then runs tests to check if the tests can detect these mutations, thus ensuring the effectiveness of the test suite.

9. Path Testing:

Purpose: Tests all possible paths through the code to ensure that every path is executed and behaves as expected.

10. Combinatorial Testing:

Purpose: Tests combinations of input values to identify

interactions or dependencies between inputs that may cause unexpected behavior.

11.Cause-Effect Graphing:

Purpose: Identifies the cause (input conditions) and effect (output conditions) relationships in the system, creating a graphical representation to derive test cases that ensure all cause-effect combinations are covered.

12.Orthogonal Array Testing:

Purpose: Utilizes orthogonal arrays to design test cases that efficiently cover combinations of input parameters with a minimal number of test cases, often used for complex systems with multiple variables.

13.Predicate Testing:

Purpose: Focuses on testing the logical predicates (conditions) in the code to ensure they evaluate correctly under various scenarios, especially in conditional statements like `if`, `while`, and `for`.

14.Assertions Testing:

Purpose: Incorporates assertions in the code to automatically check for specific conditions or invariants during execution, helping to catch logic errors at runtime.

15. Pairwise Testing:

Purpose: Tests all possible pairs of input parameters to identify interactions between parameters that might cause defects, providing a balance between thoroughness and efficiency.

16. Smoke Testing:

Purpose: Performs a preliminary test to check the basic functionality of the system to ensure that the most crucial features work correctly before proceeding to more detailed testing.

17. Sanity Testing:

Purpose: Verifies that specific functionality works as expected after minor changes or bug fixes, ensuring that recent modifications haven't introduced new defects.

18. Regression Testing:

Purpose: Retests the existing functionalities of the system to ensure that recent changes or additions haven't adversely affected the existing codebase.

19. Back-to-Back Testing:

Purpose: Compares the outputs of two versions of a system (e.g., old vs. new or different implementations) to ensure consistency and correctness.

20. Random Testing:

Purpose: Generates random inputs to test the system, useful for exploring unexpected scenarios and identifying defects that may not be covered by structured test cases.

21. Ad-hoc Testing:

Purpose: Involves informal and unstructured testing based on the tester's intuition and experience to find defects that might not be identified through formal testing methods.

22. Dynamic Testing:

Purpose: Tests the system during execution to check its runtime behavior and performance, including functional and non-functional aspects.

23. Static Testing:

Purpose: Examines the code and documentation without executing the program, including code reviews, walkthroughs, and inspections to identify potential issues early in the development process.

24. Stress Testing:

Purpose: Tests the system under extreme conditions, such as high load, limited resources, or other stress factors, to ensure it can handle stress without failing.

25.Load Testing:

Purpose: Simulates expected usage scenarios to test the system's performance under normal and peak load conditions, ensuring it meets performance requirements.

26.Performance Testing:

Purpose: Measures the system's response time, throughput, and resource utilization to ensure it meets performance criteria under various conditions.

27.Scalability Testing:

Purpose: Tests the system's ability to scale up or down in terms of performance, handling increased loads, and accommodating growth.

28.Usability Testing:

Purpose: Evaluates the system's user interface and user experience to ensure it is intuitive, easy to use, and meets user expectations.

29.Compatibility Testing:

Purpose: Verifies that the system works correctly across different environments, such as operating systems, browsers, devices, and configurations.

30. Security Testing:

Purpose: Identifies vulnerabilities and ensures that the system protects data and maintains functionality as intended, even in the presence of malicious attacks.

31. Utilizing a combination of these testing methods can help ensure comprehensive coverage and improve the robustness and reliability of your code.

32. Exploratory Testing:

Purpose: Involves simultaneous learning, test design, and test execution to explore the software's behavior and discover defects that might not be identified through scripted testing.

33. Concurrency Testing:

Purpose: Tests the system's behavior under concurrent access and operations, ensuring it handles multiple simultaneous users or processes correctly without issues like deadlocks or race conditions.

34. Volume Testing:

Purpose: Tests the system's ability to handle large volumes of data, ensuring it performs well with extensive datasets.

35.Recovery Testing:

Purpose: Tests the system's ability to recover from crashes, hardware failures, or other catastrophic problems, ensuring it can restore operations and data integrity.

36.Installation Testing:

Purpose: Verifies that the system installs correctly on different platforms and configurations, ensuring a smooth and error-free installation process.

37.Configuration Testing:

Purpose: Tests the system with various configurations to ensure it works correctly under different hardware, software, network, and other environmental conditions.

38.Localization Testing:

Purpose: Ensures the software is correctly adapted for specific languages and regions, including translations, regional settings, and cultural nuances.

39.Internationalization Testing:

Purpose: Verifies that the software can be easily adapted for various languages and regions without requiring significant changes to the code.

40.Compliance Testing:

Purpose: Ensures the software complies with relevant standards, regulations, and guidelines, such as industry standards or legal requirements.

41. End-to-End Testing:

Purpose: Tests the complete workflow of the system from start to finish, ensuring all integrated components work together as expected to achieve the desired outcomes.

42. Alpha Testing:

Purpose: Conducted by the internal team before releasing the product to external users, focusing on identifying bugs and issues in the early stages.

43. Beta Testing:

Purpose: Conducted by external users (beta testers) before the final release to gather feedback and identify issues that might not have been caught during internal testing.

44. User Acceptance Testing (UAT):

Purpose: Conducted by the end users to verify that the system meets their requirements and works as expected in real-world scenarios before going live.

45. A/B Testing:

Purpose: Compares two versions of a software feature to determine which one performs better based on user behavior and feedback.

46. Integration Testing:

Purpose: Tests the interaction between different modules or components to ensure they work together correctly.

47. Component Testing:

Purpose: Tests individual components of the system in isolation to verify their functionality.

48. API Testing:

Purpose: Tests the Application Programming Interfaces (APIs) to ensure they function correctly, handle requests and responses as expected, and meet performance and security requirements.

49. GUI Testing:

Purpose: Tests the graphical user interface to ensure it meets design specifications and provides a good user experience.

50. Accessibility Testing:

Purpose: Ensures the software is accessible to users with disabilities, complying with accessibility standards

such as WCAG (Web Content Accessibility Guidelines).

51. Prototype Testing:

Purpose: Tests early prototypes of the software to gather feedback and identify potential issues before full-scale development.

52. Compliance Testing:

Purpose: Ensures the software adheres to specific regulations, standards, or contractual agreements.

53. Non-functional Testing:

Purpose: Focuses on non-functional aspects such as performance, usability, reliability, and scalability, rather than specific behaviors or functions.

54. Static Analysis:

Purpose: Involves analyzing the code without executing it to identify potential issues such as syntax errors, code smells, and security vulnerabilities.

55. Dynamic Analysis:

Purpose: Involves analyzing the code during execution to monitor its behavior, identify runtime errors, and assess performance.

56.Parallel Testing:

Purpose: Runs two systems in parallel (e.g., old and new versions) to compare outputs and ensure consistency and correctness.

57.Remote Testing:

Purpose: Involves conducting tests remotely, often over the internet, to assess the system's behavior in a distributed environment.

57.Mobile Testing:

Purpose: Tests mobile applications to ensure they work correctly on various devices, operating systems, and network conditions.

58.Cloud Testing:

Purpose: Tests applications deployed in cloud environments to ensure they work correctly, scale effectively, and meet performance and security requirements.

59.Service Virtualization:

Purpose: Simulates components that are not yet available or are difficult to test directly, allowing for earlier and more efficient testing.

60.Continuous Testing:

Purpose: Involves automated testing throughout the development lifecycle to provide immediate feedback on code changes, ensuring ongoing quality.

English 1000 words Target

1000 words

=====

- 1.The -> therinja tha pathi pesum posth
- 2.main -> most important.
- 3.most -> 100% la 75% use panuvoom are soluvom na pathula most beautiful house .
- 4.important -> ni etha pathi pesurayo athanudaya mukiyathuvam evalavu sola use this.
- 5.purpose -> the aim or intention of something.

ஓன்றின் நோக்கம் அல்லது உட்கருத்து;

குறிக்கோள்; இலக்கு.

- 6.aim -> something that you intend to do; a purpose.

செயல் நோக்கம்; குறிக்கோள்; இலட்சியம்.

7. intention -> what somebody intends or means to do;

a plan or purpose. ஒருவர் செய்யக் கருதியுள்ள அல்லது

திட்டமிட்டுள்ள ஓன்று; திட்டம்; உள்ளக் கருத்து; செயல்

நோக்கம்.

8. much -> (used with uncountable nouns, mainly in negative sentences and questions, or after as, how, so, too) a large amount of something.

(எண்ணுதற்கியலா பெயர்ச்சொற்களுடன், குறிப்பாக

எதிர்மறை வாக்கியங்கள் மற்றும் வினாக்களில், அல்லது as,

how, so, too என்பவைக்குப்பின் பயன்படுத்தப்படுவது)

ஒன்றின் மிகுதியான அளவு; ஏராளம்.

9.used -> oru visayatha itha use panni tha panna, apdinu solum pothu use panikalam.

10.with -> in the company of somebody/something; in or to the same place as somebody/something.

ஒருவருடன்/ஒன்றுடன் சேர்ந்து அல்லது கூடி; உடன் சேர்ந்து;

ஒருவர்/ஒன்று இருக்கும் இடத்திலேயே அல்லது இருக்கும் இடத்திற்கே; உடனாக; உடன்கூட்டாக.

11.uncountable -> எண்ண முடியாததும்

12.mainly -> mostly.பெரும்பாலும்; முக்கியமாக;

மொத்தத்தில்.

13. mostly -> in almost every case; almost all the

time.கிட்டத்தட்ட அனைத்து ஆட்கள்/பொருள்கள்

வகையிலும்; ஏறத்தாழ எல்லா நேரத்திலும்.

14. task -> a piece of work that has to be done,

especially an unpleasant or difficult one.செய்யப்பட

வேண்டிய, குறிப்பாக மனம் நாடாத அல்லது இடர்ப்பாடான

பணி; இடுபணி; கடமைப் பொறுப்பு.

15. encounter -> to experience something (a danger, difficulty, etc.).(ஆபத்து, இடர், முதலிய) ஒன்றை

எதிர்கொள்; அனுபவி.

16. ensure -> to make sure that something happens or

is definite. ஒரு செயல் நிகழ்தலை அல்லது ஒன்றன்

நிச்சயத்தன்மையை உறுதிப்படுத்திக்கொள்; ஒன்று உறுதியாக

நிகழுமாறு அல்லது அமையுமாறு ஜயத்துக்கிடமறப்

பார்த்துக்கொள்

17. especially -> more than other things, people,

situations, etc.; particularly. மற்ற பொருள்கள், ஆட்கள்,

சூழ்நிலைகள், முதலியவற்றை விட; குறிப்பாக;

குறிப்பிடத்தக்க முறையில்; தனிப்பட;

தனிமுறையில்

18. framework ->the basic structure of something that

gives it shape and strength. ஒன்றுக்கு உருவமைதியும்

உறுதியும் அளிக்கும் அதன் ஆதாரக் கட்டமைப்பு; உருவரைச்

சட்டம்.

19. inject -> to put a drug under the skin of a person's

or an animal's body with a needle (syringe). ஓராள்

அல்லது ஒரு விலங்குக்கு தோலின் உட்பகுதியில் ஊசி

மூலமாக மருந்து செலுத்து; ஊசி போடு.

20. during -> within the period of time

mentioned. குறிப்பிடப்பட்ட கால அளவில்;

நேரத்தினிடையே.

21. cases-> a particular situation or example of something. குறிப்பிட்ட ஒரு சூழ்நிலை; சூழல்; ஒன்றன் எடுத்துக்காட்டு

22. concise -> giving a lot of information in a few words; brief. மிகுதியான தகவலைக் குறைந்த சொற்களில் தருகிற; சுருக்கமான; பொழிப்பான.

23. reduce -> to make something less or smaller in quantity, price, size, etc. எண்ணிக்கை, விலை, உருவளவு முதலியவற்றைக் குறை; சிறிதாக்கு.

24. boilerplate -> Boilerplate text, or simply boilerplate, is any written text that can be reused in new contexts or applications without significant changes to the original.

25. typically -> in a typical case; that usually happens in this way. எதிர்பார்க்கத்தக்கதாக; வழக்க முறையானதாக.

26. ecosystem -> all the plants and animals in a particular area considered together with their surroundings. குறிப்பிட்ட பகுதியில் காணப்படும் அனைத்துத் தாவரங்கள், விலங்குகள் மற்றும் அவற்றின் சுற்றுச்சூழல்கள் ஆகியவற்றின் முழுமைத் தொகுதி; சுற்றுச்சூழல் அமைவு.

27. primarily -> more than anything else; mainly. வேறுதனையும் விட; முக்கியமாக; முதன்மை நிலையில்.

28. library -> a room or building that contains a collection of books, etc. that can be looked at or borrowed. பாடிப்பதற்குரிய அல்லது கடனாகப் பெற்றுச் செல்வதற்குரிய நூல்கள் முதலியவை திரட்டி வைக்கப்பட்டுள்ள அறை அல்லது கட்டடம்; நூல் நிலையம்; நூலகம்.

29. test -> to try, use or examine something carefully to find out if it is working properly or what it is like. ஒன்றின் சரியான செயல்பாட்டை அல்லது தன்மையைக் கண்டறியும் பொருட்டு அதனைக் கவனமாகச் சோதித்துப் பார், பயன்படுத்திப் பார் அல்லது ஆராய்ந்து பார்.

30. interactions -> **an occasion when two or more people or things communicate with or react to each other**

31. integration -> **the action or process of combining two or more things in an effective way**

32. verify -> to check or state that something is true. ஒன்றின் உண்மைத் தன்மையைச் சரிபார் அல்லது ஒன்று உண்மை என்று கூறு; உண்மையை உறுதி செய்.

33. between -> in the space in the middle of two things, people, places etc. இரண்டு பொருள்கள், நபர்கள், இடங்கள் முதலியவற்றுக்கு மத்தியில் உள்ள இடத்தில்;

இடைப்பட்ட இடத்தில்; இருவர்/இரண்டின் நடுவில்

34. different -> not the same. ஒரே மாதிரி இல்லாமல்

இருக்கிற; வேறுபட்ட; வித்தியாசமான.

35. part -> one of the pieces, areas, periods, things, etc. that together with others forms the whole of

something; some, but not all of something.(பொருள்,

இடம், காலம் முதலியவை வகையில்) ஒரு முழுமையின்

பகுதிகளில் ஒன்று; பகுதி; துண்டு; கூறு; பிரிவு.

36. multiple -> involving many people or things or

having many parts. பலர் அல்லது பல பொருள்கள்

உள்ளடங்கியுள்ள/சேர்ந்தினைந்துள்ள; பல பகுதிகளை/

கூறுகளையுடைய; பல்கூட்டான்; பன்முகமான.

37. components-> one of several parts of which

something is made. ஒரு பொருளின் பல உறுப்புகளில்/

கூறுகளில் ஒன்று; உறுப்பு; அங்கம்; ஆக்கக் கூறு; பகுதி.

38. end-to-end -> "end-to-end" typically means covering a process or system from the starting point to the end point, ensuring that all steps and components in between are included

39. suitable -> It implies that something meets the necessary requirements or criteria and is well-matched to the needs

40. fitting -> right; suitable. சரியான; ஏற்புடைய; தக்க;

பொருத்தமான; தகுதிப்பாடுடைய.

41. unit -> a single thing which is complete in itself, although it can be part of something larger. தன்னளவில் முழு நிறைவான, பெரிதான ஒன்றின் பகுதியாகவும்

இருக்கக்கூடிய, தனித்த ஒரு பொருள்; தனியொருவர்/

தனியொன்று; ஒன்றின் தனிக்கூறு; ஒருமம்

42. entity -> something that exists separately from something else and has its own

identity. மற்றொன்றிலிருந்து தனிவேறாக அமைவதும்,

தனித்துவம் உடையதுமான ஒன்று; தனிப்பண்டுப் பொருள்.

43. individual -> considered separately rather than as part of a group. ஒரு குழுவின் அங்கமாக இன்றி, தனிப்படகருதப்படுகிற; தனியான; தனிப்பட்ட.

44. behaviour -> the way that you act or

behave. செயல்படும் அல்லது நடந்துகொள்ளும் முறை;

ஒழுக்கம்; நடத்தை.

45. together -> with or near each other. ஒருவரோடு

ஒருவராக/ ஒன்றுடன் ஒன்றாக; ஒருவர் பக்கத்தில் ஒருவராக/

ஒன்றின் பக்கத்தில் ஒன்றாக; ஒருசேர; இணைந்து; கூடி.

46. external -> connected with the outside of

something. ஒன்றன் வெளிப்பகுதியைச் சார்ந்த அல்லது

வெளிப்பகுதிக்குரிய.

47. dependency -> the state of being dependent on somebody/something; the state of being unable to live without something, especially a drug. ஒருவரை/ஒன்றைச் சார்ந்திருக்கும் நிலை; ஒன்று இல்லாமல், குறிப்பாக, போதைப் பொருள் இல்லாமல், வாழ முடியாத நிலை.

48. without -> not having or showing something. ஒன்று இல்லாது அல்லது ஒன்றைக் காட்டாது.

49. manage -> to succeed in doing or dealing with something difficult; to be able to do something. கடினமான ஒன்றைச் செய்வதில் அல்லது கையாள்வதில் வெற்றிகாண்; ஒன்றைச் செய்ய இயல்.

50. after -> many times; frequently. அடிக்கடி; பல தடவை; மீண்டும் மீண்டும்.

51. in -> inside or to a position inside a particular area or object. (இட வகையில்) குறிப்பிட்ட இடத்தின் அல்லது பொருளின் உட்பகுதியில் அல்லது உட்பகுதியிலுள்ள ஓரிடத்திற்கு; உள்ளிடத்தில்; உள்ளிடத்திற்கு; உள்ளே; உள்ளாக; உட்புறமாக.

52. practice -> **the act of doing something regularly or repeatedly to improve your skill at doing it**

53. comprehensive -> including everything or nearly everything that is connected with a particular

subject. குறிப்பிட்ட ஒரு பொருள் தொடர்பான

அனைத்தையும் அல்லது சற்றேறக்குறைய அனைத்தையும் உள்ளடக்கியுள்ள; அகல்விரிவான; அகல் பெருந்

தொகுதியான.

54. create -> to cause something new to happen or exist. புதிதாக ஒன்று நிகழச்செய் அல்லது தோன்றச் செய்;

புதிது ஆக்கு; படை..

55. effective -> successfully producing the result that

you want. விரும்பும் முடிவைப் பயன் நிறைவுடன்

உருவாக்குகிற; வெற்றிகரமான நல்விளைவை ஏற்படுத்துகிற

56. provide -> to give something to somebody or make something available for somebody to use; to supply

something. ஒருவருக்கு ஒன்றைக் கொடு; ஒருவருக்கு

ஒன்றைக் கிடைக்கக்கூடியதாக்கு; தருவித்து வழங்கு.

57. structure -> the way that the parts of something

are put together or organized. ஒன்றின் கூறுகள்

ஒருங்கிணைக்கப்பட்டுள்ள அல்லது ஒழுங்கமைவு

செய்யப்பட்டுள்ள முறை; அமைப்பு முறை; கட்டமைப்பு.

58. while -> during the time that; when. குறிப்பிட்ட ஒரு

செயல் நிகழ்வின்போது; அச்சமயத்தில்; அப்பொழுது.

59. assert -> to say something clearly and firmly. தெளிவாகவும் உறுதியாகவும் ஒன்றைக் கூறு;

அழுத்தந்திருத்தமாகக் கூறு.

60. expect -> to think or believe that somebody/something will come or that something will

happen. ஒருவர் வருவார்/ ஒன்று வரும் அல்லது ஒன்று

நிகழும் என நினை அல்லது நம்பு; எதிர்நோக்கு

61. since -> from a particular time in the past until a

later time in the past or until now. கடந்த காலத்தில்

குறிப்பிட்ட வேளையிலிருந்து கடந்த காலத்தில் பிந்திய ஒரு சமயம் வரை அல்லது இப்பொழுது வரை.

62. immerse -> to put something into a liquid so that it is covered. முழுமையாக முழுகுமாறு ஒன்றை ஒரு திரவத்தில் இடு; உள் அமிழ்த்து; மூழ்க்கச்செய்.

63. enthusiasm -> a strong feeling of excitement or interest in something and a desire to become involved in it. ஒன்றன் காரணமாக உண்டாகும் தீவிரமான மன எழுச்சி

அல்லது பற்றார்வம் மற்றும் அதில் ஈடுபாடு கொள்ள

வேண்டுமென்ற விருப்பம்; ஆர்வக் கிளர்ச்சி; உற்சாகம்.

64. invoke -> call on, that method

65. setting -> the position something is in; the place

and time in which something happens. ஒன்று

அமைந்துள்ள நிலை; இருப்பு நிலை; கிடப்பு நிலை; ஒரு நிகழ்வின் இடமும் காலமும்; பின்னணி; சூழல்.

66. code -> a system of words, letters, numbers, etc. that are used instead of the real letters or words to

make a message or an information secret. ஒரு செய்தி

அல்லது தகவலை மறைபொருளாக்க வேண்டி, உண்மையான

எழுத்துகள் அல்லது சொற்களுக்கு மாற்றாகப்

பயன்படுத்தப்படும் சொற்கள், எழுத்துகள், எண்கள்

முதலியவற்றைக் கொண்ட ஒரு முறை; இரகசியச் சொல்/

எழுத்து/இலக்கக் குறியீட்டு முறை; குழுஉக்குறி.

67. of -> belonging to, connected with, or part of

something/somebody. ஒன்றுக்கு/ஒருவருக்கு உரிய; ஒன்று/

ஒருவர் தொடர்பான; ஒன்றன்/ஒருவரது பகுதியான; உடைய;

உள்ள; சார்ந்த; கொண்ட.

68. this -> used for talking about somebody/something

that is close to you in time or space. கால அல்லது இட

வகையில் தனக்கு அண்மையில் உள்ள ஆள் அல்லது பொருள்

பற்றிப் பேசப் பயன்படுத்தப்படுவது: இவர்; இது; இந்த.

69. line -> a long thin mark on the surface of

something or on the ground. ஒன்றின் பரப்பின் மீது அல்லது

தரையின் மீது இடப்படும் நீண்டு மெலிதான குறி; கோடு; வரை.

70. meaning -> the thing or idea that something represents; what somebody is trying to communicate. ஒன்று சுட்டும் பொருள் அல்லது கருத்து; ஒருவர் தெரிவிக்க முயலும் ஒன்று; சொற்பொருள்; பொருள்; கருத்து; விளக்கம்.

71. up -> at or to a high or higher level or position. உயர் மட்டத்தில் அல்லது உயர் நிலையில்; உயர் மட்டத்திற்கு அல்லது உயர் நிலைக்கு; மேலும் உயர் மட்டத்தில் அல்லது உயர் மட்டத்திற்கு.

72. expectation -> the belief that something will happen or come. ஒன்று நிகழும் அல்லது வந்துசேரும் என்ற நம்பிக்கை; எதிர்பார்ப்பு.

73. for -> **used to indicate the place someone or something is going to or toward.** He just left for the office

74. method -> a way of doing something. ஒன்றைச் செய்யும் முறை; வழிமுறை.

75. on -> supported by, fixed to or touching something, especially a surface. ஒன்றினால், குறிப்பாக ஒரு

பரப்பினால், தாங்கப்படுகிற, அதனுடன் பொருத்தப்பட்ட அல்லது அதனைத் தொட்டுக்கொண்டிருக்கிற; மேலே; மேல்; மீது.

76. statement -> something that you say or write, especially formally.குறிப்பாக, முறை சார்ந்து, கூறப்படுவது அல்லது எழுதப்படுவது; அறிவிப்பு வாசகம்; அறிக்கை.
77. specify -> to say or name something clearly or in detail.ஓன்றைத் தெளிவாக அல்லது நுனுக்க விவரங்களுடன் கூறு அல்லது சுட்டிக் குறிப்பிடு; விளக்க விவரமாகக் கூறு.

78. should -> ought to.(ஓருவர் ஓன்றைச் செய்வது அல்லது ஒன்று நிகழ்வது சரியானது அல்லது பொருத்தமானது எனக் கூறப் பயன்படுத்தப்படுவது) இன்னது செய்யப்பட வேண்டும் அல்லது நிகழ வேண்டும்; இன்னது செய்யப்படுவது அல்லது நிகழ்வது விரும்பத்தக்கது அல்லது நலமானது.

Learning Approach

Learning Approach

1. Roadmap for the Skill:

- **Create a Roadmap:** Outline the skill, including key topics and subtopics. Set clear goals and milestones.
- After core topic find and which problem resolve check and remove unwanted module

• Topic Overview:

What: Define the topic. Understand its fundamental concepts.

Why: Understand the importance and relevance of the topic. Identify its applications and significance.

How: Learn how the topic works. Explore its mechanisms, processes, and methodologies.

Advantages: Identify the benefits and strengths of the topic.

Disadvantages: Understand the limitations and challenges associated with the topic.

Problem Resolution: Determine which problems the topic resolves and its practical applications.

• Subtopic Exploration:

Identify Subtopics: Break down the main topic into

smaller, manageable subtopics.

What (Subtopic): Define each subtopic. Understand its fundamental concepts.

Purpose (Subtopic): Learn the purpose and importance of each subtopic.

How to Use (Subtopic): Understand how to apply the subtopic in practical scenarios.

Advantage

DisAdvantage

finaly one time check you know

JavaScript

Introduction

1. why javascript created

->html , css only style purpose no interaction so that reason create javascript , first name live script after change js.

->without js any website doesn't create . eventhough facebook, google

2. javascript supports?

⇒ websites, webapp ,mobile apps, games, AI & machine learning also.

3. why js learn?

-> any front end compulsory js need;

4.how javascript work?

-> browser la mattum tha javascript execute engine available.

-> laterly browser la irrunthu eduthu node la set pananga c++ la after that js demand anathu.

5. browser console js work why?

-> by default js browser la only working.

-> console.log means , web la console la show

jsCodeStart

show output console & alert message

=====

1.alert("hariamni") -> alert message came.

2.console.log("harimani"); -> show output in console tab in browser

variables

=====

1.let -> default value = “undefiend” -> prefer this using

2.const ->

3.var

operator precedence

=====

Please Excuse My Aunt Sally, Remember Expressions Before Xmas, Or Later Learn Theory.

conditional statement

-
- 1.if/else
 - 2.if/else if/else
 - 3.new Date()

switch case

=====

switch(conditon){

 case a>20: -> this is support js but java does not

```
support  
    default  
}
```

Loops

- 1.for loop
- 2.while loop
- 3.do while loop
- 4.for-in loop

```
let persons = {  
    name:"hari",  
    age:20  
};
```

```
{  
    > name, age print not value
```

```
for(let key in persons)  
    console.log(key)-
```

```
console.log(persons[key]) -> value print  
    }
```

```
let colors  
=['red','blue'];
```

```
for(let color in colors)  
{
```

```
console.log(color) -> key only print like index 0,1,2
```

```
console.log(colors[color]) value print  
    }
```

iteration purpose use
use this
5.for-of loop

```
for(let  
person of persons{
```

console.log(person) -> directly value provide so the object

doesn't allow only array

```
}
```

without concatenate call name using ``

```
=====
```

1.let msg = "my name is "+name+", i love
"+interest";

2.let msg = `my name is \${name}, i love \${
interest}`;

OOp

OOP

=====

```
let person = {
    name: "hari",
    age: 21,
    isAlive: true,
    color: ['red', 'blue'],
    address: {
        city: "chennai",
        state: "tamil Nadu"
    },
    //method also called in oops
    greeting: function() {
        let msg = `my name is ${this.name}, i love ${
            interest}`;
        console.log(msg);
    }
}

//call object inside function
```

person.addition(); -> output show
person.addition; -> function code show
person.name -> name value print

short code write Factory function

=====

step 1:

=====

```
function createPerson(){
    let person = {
        name: "hari",
        greeting:function(){
            let msg = `my name
is ${this.name}, i love ${interest}`;
            console.log(msg);
        }
    };
    return person;
}
```

step 2:

=====

```
function createPerson(){
    return {
        name: "hari",
        greeting:function(){
            let msg = `my name
is ${this.name}, i love ${interest}`;
            console.log(msg);
        }
    };
    //return person
}
```

step 3:

=====

```
function createPerson(name){  
    return {  
        name: name, // dynamic value  
        pass  
            greeting:function(){  
                let msg = `my name  
is ${this.name}, i love ${interest}`;  
                console.log(msg);  
            }  
    };  
}
```

step 4:

```
function createPerson(name){  
    return {  
        name, // same parameter so not  
        mention -> name = name;  
        greeting:function(){  
            let msg = `my name  
is ${this.name}, i love ${interest}`;  
            console.log(msg);  
        }  
    };  
}
```

step 5:
=====

```
function createPerson(name){  
    return {  
        name, // same parameter so not  
        mention -> name = name;  
        greeting:function(){  
            let msg = `my name  
is ${this.name}, i love ${interest}`;  
            console.log(msg);  
        }  
    };  
}
```

```
        name, // same parameter so not  
mention -> name = name;  
  
        greeting(){  
            let msg = `my name  
is ${this.name}`;  
            console.log(msg);  
        }  
    };  
}
```

```
let anbu = createPerson("anbu");  
anbu.greeting();
```

step 6:

=====

```
function createPerson(name,age){  
    return {  
        name, // same parameter so not  
mention -> name = name;  
        age,  
  
        greeting(){  
            let msg = `my name  
is ${this.name} , ${this.age}`;  
            console.log(msg);  
        }  
    };  
}  
createPerson('Anbu',24).greeting();
```

constructor function()

=====

function Person(name,age){

 this.name=name;

 this.age =age;

 this.greeting= function(){

 console.log(`my

name is \${this.name}`);

}

//return object -> it is

automatically nadakum

}

new Person('Anbu',24).greeting();

js Dynamic Object

=====

const person ={ -> const using object insted of
any other change error through

 name:"hari"

}

person.name="mani";

person.age=21;

person.sex="male"

```
delete person.name;  
delete person.age;  
console.log(person);
```

constructor -> each object have default constructor

=====

```
let object ={} ;-> new Object();  
let number = 5; -> new Number(5);  
let alph = "hari";-> new String("hari");  
let bool = true; -> new Boolean (true);
```

primitive & reference type

=====\\

```
→ a = 10;  
→ b=a;  
→ a=20; so the output a =20 & b= 10 so the reason is  
primitve store value.
```

```
a ={value:10};
```

```
b=a;
```

```
a.value=20; so the output is a=20 & b=20 also b'z it  
is referce store.
```

advance

for of loop

=====

- it is support for to iterate directly array value only but not for objec.
- we need object iteration for value use this method "Object.keys(user)
- the above method provide keys only.
- we need key and value Object.entries(user) this method like java provide key and value in array.

```
let user = {  
    name:"Anbu",  
    getFullName(){  
        console.log(`my name is ${this.name}`);  
    }  
}
```

```
for( let key of Object.keys(user))  
    console.log(key)-> error throw
```

```
for(let entry of Object.entries(user))  
    console.log(entry); ->  
["name","Anbu"]
```

if()

```
=====
  if("name" in user)
    console.log("yes!");
```

object Cloning

=====

3 ways object cloning

```
let user = {
  name: "hari",
  age: 24
}
```

```
let another = {};
```

1. for(let key in user){
 another[key] = user[key];
}

2. let another = Object.assign({number: 20}, user); -> name, age, number

3. spread operator

```
=====
let another = {...user};
console.log(another);
```

Garbage

Garbage collection

=====
-> garbage collection concept also in js .

-> js engine automatically clear the unwanted object memory.

Math Operation

- =====
- 1.min
 - 2.max
 - 3.ceil
 - 4.round
 - 5.random()
 - 6.clear()
 - 7.PI -> is a property

String Objects create

- =====
- 1. let name = "hari" -> string literals it is not object it is primitive
 - 2. let name = new String("hari") -> it is a String object;

String inbuild methods()

=====

- 1.length -> property

- 2.concat()

- 3.includes() -> if it is available or not like contains

4.startsWith()

5.endsWith()

6. indexOf()

7.match(regex)

8.repeat() -> how many times that will be

repeat

9.replace()

10.split()

11.slice(2) -> print data index from 2 with
start

12. substr()

13.toLowerCase()

14.toUpperCase()

15.trim()

16.trimStart()

17.trimEnd()

Literals

- 1.Object: {} -> object literal
- 2.Boolean: true or false -> boolean literal
- 3.String : ' ', " " -> string literal \n \t \" \"

Template literal -> ` ` -> back tick. prettier extension used to auto formate in vs code editor

=====

\n \t \" \" -> we can use before template literal used to formate
but after ecmo script just back tick used to format

`hari mani “done” ‘hey’
now i am here,` -> same formate output show with space next line also

Date()

====

const date = Date() -> it is convert by default string so that specifically we can't get date,month year .
Date().toString() it is auto convert

To Overcome this proble create Object insted of String

const now = new Date();
- now we can acces all methods

1.getFullYear

- 2.now
- 3.Day
- 4.Date

Array

=====

```
const array = ["banana", "apple", "grapes"];
```

1. array[3] = "orager";

2.push("hell","hari") these two added in end of the array

3.unshift("hari",mani") these two added in front of the array

4.splice(3,0,"hi") index,delet, value this is added in specify index

search

=====

1.indexOf("lemon") -> it return the index value otherwise return -1

2.lastIndexOf()

3.includes() -> checks if it exist or not

4.toString()

5.length

6.at(2) -> indexed element return it is like as indexOf()

more method availe we check and learn new methods

Array inside object check items

=====

- it is like as predicate in java 8;

-> find(predicate<?>) this method have predicate functione

```
let order =[  
    {id :1, nam:"hari", item:"phone"},  
    {id :2, nam:"hari", item:"cell"},  
]
```

```
let orders =order.find(function(order){  
    return order.item === "phone";  
});  
console.log(orders);t
```

js

```
let order = [
    {id :1, nam:"hari", item:"phone"},  

    {id :2, nam:"hari", item:"cell"},  

]
```

```
let orders =order.find(function(order){  

    return order.item === "phone";  

});  

console.log(orders);
```

Event Handling

what is event handling?

Event handling in javascript the process of actions that occur in browser such as user click a button, move to mouse, press key . theese actions are called event handling.

why should we go?

To intractive user and show content dynamically.

How to implemt?

```
<button id="myButton">Click Me!</button>
```

```
document.getElementById("myButton").addEventListener("click", function() {
    alert("Button was clicked!");
});
```

Advantage

- intractivity
- dynamic content
- user experience
- customization and personalization

DisAdvantage

- Overhead adding many eventlistener
- complexity of code
- perform decrease

Problme Resolution

user interaction
real-time updates
asynchronization fetch data from database
control and customization

1.**Mouse Event**

2.Keyboard Events

3.Form Events

4.window Events

5.Drag Events

6.Touch Events

7.Media Events

8.Other Events

Mouse Event

what is mouse event?

Mouse event in web development make websites interactive by responding to actions like click, hover and mouse movements on page elements.

why it is important?

To Interactive user by mouse click, hover user feedback

Navigation

Integration with Other Events

How to implemt?

```
<button id="myButton">Click Me!</button>
```

```
document.getElementById("myButton").addEventListener("click", function() {
    alert("Button was clicked!");
});
```

problem Resolution?

To Interactive user by mouse click, hover user feedback

Navigation

Integration with Other Events

1.click

-> press and release a mouse button.

```
<button id="myButton">Click Me!</button>
```

```
document.getElementById("myButton").addEventListener("click", function() {
    alert("Button was clicked!");
});
```

2. dblclick

-> Quickly click a mouse button twice.

3. mouseover

-> Move the mouse over an element.

```
<button id="myButton">Click Me!</button>
```

```
function buton(){ alert("button was clicked"); }
document.getElementById("buton").addEventListener(
"mouseover",buton); }
```

4. mouseout

-> Move the mouse away from an element.

```
<button id="myButton">Click Me!</button>
```

```
function buton(){ alert("button was clicked"); }
document.getElementById("buton").addEventListener(
"mouseout",buton); }
```

5. mousedown

6. mouseup

7. mousemove

what is mouse event?

why it is important?

How implemt?

problem to sove?

Keyboard Event

- 1.keydown
- 2.keyup
- 3.keypress

Form Event

Window Event

- 1.load
- 2.unload
- 3.resize
- 4.scroll

Drag Event

- 1.dragstart
- 2.drag
- 3.dragover
- 4.drop
- 5.dragend

Touch Event

- 1.touchstart
- 2.touchmove
- 3.touchend
- 4.touchcancel

Media Event

Other Event

- 1.error
- 2.contextmenu
- 3.focusin
- 4.focusout

DOM

Document object model

Document -> file(xml)

Object -> element, tags(<h1>)

Model -> Layout structure

what is dom?

The w3c Document Object Mo

window

window inside dom and Bom available

window.document instead of we can specify
document.title

Bom -> browser object model

nodejs full and full window based not document based

dom and Bom parent window

```
window.open("http://google.com");
window.close();
window.moveTo(500,700);
window.resizeTo(100,300);
window.alert("hello");
window.prompt("what is your name");
window.print("it is print the window");
console.log(console);
```

Bom

Browser Object Model(Bom)

- 1.Screen -> it is used to manipulate in window screen
- 2.Location -> it is used to get host url, protocol etc...
- 3.History -> it is used to forward, backward etc...
- 4.Navigator -> it is used to application related like appname, cookie, vendor so much information and also it is mainly we can research in hacking.

document

document.cookie

document.URL

document.all -> all html tag used in that web page

document.forms

document.links -> how many linkes used in that web page

document.body

document.images

dir() -> directory find

document.write()

document.writeln()

[developer.mozilla.org // Dom documentaion](https://developer.mozilla.org/en-US/docs/Web/API/Document)

Java

List

====

1.ArrayList

=====

- ⇒ indexed based retrieve and replace best choice
- ⇒ show datas in that context only use
- ⇒ data read purpose only

2.HashMap

=====

- ⇒ show data and search then use HashMap is best choice

3.LinkedList

=====

- ⇒ insert and delet for midle LinkedList is the best choice
- ⇒ stack and Queue
- ⇒ FILO & FIFO

4.Stack

=====

- ⇒ only show the data FILO order use

5.Queue

=====

- ⇒ only show the data FIFO order use

6. PriorityQueue

=====

⇒ task priory based also order priority based

7.ArrayDeque

=====

⇒ navigate or undo/redo

Question And Answer Professional Way

Choosing the Best Method:

- **CAR:** Use for straightforward, technical explanations or when highlighting specific contributions and results.
- **STAR:** Use for behavioral questions where you need to demonstrate soft skills, teamwork, or detailed processes.
- **S-CAR:** Use when you need to emphasize problem-solving skills and overcoming specific challenges.

Universal Template Using the B-SCAR Method:

1. Background:

- "Define the concept or term."
- "Explain its relevance or importance."

• Situation:

"Describe a scenario or context where this concept is applied."

"Provide any necessary details to set up your example."

- **Challenge:**

"Outline the specific problem or challenge you faced in this situation."

"Highlight why this was a significant issue to address."

- **Action:**

"Describe the steps you took to address the challenge."

"Explain the tools, technologies, or methods you used."

- **Result:**

"Share the outcome of your actions."

"Quantify the results if possible (e.g., improved performance by X%, reduced errors by Y%)."

Example Answer to "What is a collection?" using the B-SCAR method:

1. **Background:**

"In Java, a collection is a framework that provides an architecture to store and manipulate a group of objects. It is part of the `java.util` package and includes interfaces and classes for various types of collections such as lists, sets, and maps."

- **Situation:**

"At Spire Systems, we needed to process and manage large datasets for our analytics module. Efficient data handling was crucial for ensuring timely and accurate analysis."

- **Challenge:**

"The main challenge was to handle a large volume of data efficiently, ensuring quick access and manipulation without compromising performance. We also needed to ensure that the data remained consistent and free from duplicates."

- **Action:**

"We utilized the Java Collections Framework extensively. `ArrayList` was used for maintaining ordered lists of user inputs, `HashSet` for eliminating duplicates, and `HashMap` for efficient key-value pair management. Additionally, we implemented synchronized collections to ensure thread safety during concurrent data processing."

- **Result:**

"This approach significantly improved our data processing efficiency. We saw a 30% improvement in data retrieval times and a 25% reduction in memory usage due to the elimination of duplicate entries. This not only enhanced the performance of our analytics module but also improved the accuracy and reliability of our data insights."

SelfIntro

Self Intro

I am selvamani. I have completed my bachelor of computer application from shanmuga college. I am having around 2 years of experience in this IT field.

Currently I am working in spire systems as a java developer. I am working in the project titled “Logistic Management Systems” the project is based on goods Management , Purchase management, sales Management, Invoice and payment management etc...

My role is to develop the application based on user requirement as well as I involve in Bugs fixing also.

Technologies used in this project are Java,Java 8, Spring framework, Spring Boot, spring Data JPA, Spring MVC, MySQL, Git, Github, RESTful Web services, Maven, and I used IDEs named “eclipse”, “sts”.

My Over all skills sets are Java, java 8, spring framework, spring boot,spring AOP, spring MVC, Spring Data jpa, spring security, JDBC,Mysql , Html ,css, javascript, servlet , jsp.

Thank you that's from my side .

java8To21

Java 8

=====

1. **Stream API:** Revolutionized data processing with functional programming constructs.
2. **Lambda Expressions:** Simplified anonymous function implementation.
3. **Optional:** Helped reduce `NullPointerExceptions` by providing a more expressive way to handle optional values.
4. **Date and Time API:** Modern and thread-safe replacement for the old date and calendar classes.

Java 9

=====

1. **Module System (Project Jigsaw):** Significant for large applications needing better dependency management.
2. **JShell (REPL):** Great for quick prototyping and learning, though less critical in production environments.

Java 10

=====

1. **Local-Variable Type Inference (var):** Widely adopted for cleaner and more concise code.

Java 11

=====

1. **HTTP Client API:** Heavily used for modern web applications and microservices.
2. **Local-Variable Syntax for Lambda Parameters:** Used for consistency, though not as impactful as the core lambda features from Java 8.

Java 14 and 15

=====

1. **Switch Expressions:** Becoming more common as it simplifies and makes switch statements more powerful.
2. **Text Blocks:** Very popular for handling multi-line strings, improving readability.
3. **Records:** Gaining traction for data classes to reduce boilerplate code.
4. **Pattern Matching for instanceof:** Simplifies code and is becoming widely used.

Java 17

=====

1. **Sealed Classes:** Provides better control over class hierarchies, useful in domain modeling.

Java 19 (Preview) and Beyond

=====

1. **Virtual Threads:** Part of Project Loom, they are expected to transform concurrency handling by providing lightweight threads, making it easier to write scalable concurrent applications.
2. **Structured Concurrency:** Also from Project Loom, aims to simplify error handling in concurrent

applications.

General Trends

- 1. Adoption of Modern HTTP Client API:** For microservices and web applications.
- 2. Use of Functional Programming Constructs:** Enabled by Streams, lambdas, and new concurrency features.
- 3. Immutable Data Classes:** Using Records for data modeling.
- 4. Improved Code Readability and Maintainability**: Through features like text blocks, pattern matching, and local variable type inference.

Industry Adoption

- **Microservices Architecture:** Often leverages the HTTP Client API and modern concurrency features.
- **Data Processing and Analytics:** Uses Streams and functional programming extensively.
- **Enterprise Applications:** Benefit from modularization and enhanced concurrency handling.
- **Cloud-Native Development:** Adopts many of these features for better scalability and maintainability.

java9 and 10

java 9

=====

Module System (Project Jigsaw): project structure
maintain panna module concept eduthutu vanthanga

JShell (REPL): code snippet check panna use
pannikalam.

java 10

=====

var - > it is a veriable datatype.

SpringFramework

1.What is SpringFramework?

i) what is spring framework?

- The Spring Framework is an open-source framework for building enterprise Java applications. It provides comprehensive infrastructure support, making it easier to develop Java applications by providing solutions to many common problems faced in enterprise application development

- Spring is a framework of frameWork.

- it provides comprehensive for develop j2ee web development.

ii why spring framework use?

- The main purpose of spring framework is to simplyfy j2ee development application.

- The below functionaliy we get.

Advantage of Spring framework:

1. Comprehensive Infrastructure Support

- **Dependency Injection (DI)**: Spring's powerful DI mechanism promotes loose coupling and enhances the manageability and testability of applications.
- **Aspect-Oriented Programming (AOP)**: Spring provides robust support for AOP, which allows for the separation of cross-cutting concerns such as logging, transaction management, and security, improving code modularity.

2. Modularity and Reusability

- ◊ **Modular Architecture**: Spring's modular architecture allows developers to use only the components they need, such as Spring MVC for web applications or Spring Data for data access, without being forced to include unnecessary parts.
- ◊ **Reusable Components**: The framework encourages the creation of reusable, loosely-coupled components, which can be easily integrated into different parts of an application or even across different projects.

3. Ease of Integration

- ◊ **Integration with Other Frameworks**: Spring integrates seamlessly with various other frameworks and technologies, such as Hibernate, JPA, and JMS. This

flexibility allows developers to choose the best tools for their specific needs.

◇ **Enterprise Services:** Spring provides support for a wide range of enterprise services like messaging, transaction management, and security, simplifying the development of robust and scalable applications.

4. Declarative Programming

◇ **Annotations:** Spring uses annotations extensively to simplify configuration and reduce boilerplate code. For example, annotations like `@Component`, `@Service`, and `@Repository` help in defining beans and managing their lifecycle.

◇ **XML and Java-based Configuration:** Developers can configure Spring applications using XML, Java-based configuration, or a combination of both, providing flexibility in how they set up their application context.

5. Productivity Enhancements

◇ **Spring Boot:** A subproject of Spring, Spring Boot simplifies the development of new Spring applications with auto-configuration, an embedded server, and production-ready features like metrics and health checks. This leads to faster development and deployment cycles.

◇ **Spring Initializr:** A web-based tool that helps in quickly generating Spring Boot projects with the required dependencies, further speeding up the

development process.

6. Testing Support

- ◊ **Testability:** The loose coupling achieved through DI makes it easier to write unit tests and integration tests. Spring also provides support for testing with utilities and annotations, such as `@SpringBootTest` and `@MockBean`.
- ◊ **Mocking and Stubbing:** Spring's testing support simplifies the use of mock objects and stubs, which is essential for effective unit testing.

7. Scalability and Performance

- ◊ **Efficient Transaction Management:** Spring provides declarative transaction management, which is efficient and reduces the complexity of managing transactions manually.
- ◊ **Lightweight Container:** Spring's lightweight container provides efficient dependency injection and management of beans without significant performance overhead.

8. Community and Ecosystem

- ◊ **Large Community:** Spring has a large, active community, which means abundant resources, documentation, and third-party libraries are available.
- ◊ **Extensive Ecosystem:** Spring has a rich ecosystem of projects and subprojects, such as Spring Cloud for

microservices, Spring Data for data access, and Spring Security for authentication and authorization, which provide comprehensive solutions for various enterprise needs.

Key Features of the Spring Framework

1. **Dependency Injection (DI):**

- The core concept of Spring is Dependency Injection, which promotes loose coupling by allowing the framework to manage the dependencies of objects.
- DI helps in creating more modular and testable code.

• **Aspect-Oriented Programming (AOP):**

- ◊ AOP complements DI by providing a way to modularize cross-cutting concerns like transaction management, logging, and security.
- ◊ AOP allows these concerns to be separated from the business logic.

• **Transaction Management:**

- ◊ Spring provides a consistent abstraction for transaction management that can be used in any environment.
- ◊ It supports both programmatic and declarative transaction management.

- **Model-View-Controller (MVC) Framework:**
 - ◊ Spring's MVC framework is a well-designed, flexible framework for building web applications.
 - ◊ It separates the application logic, business logic, and UI, making the application more manageable and testable.
- **Data Access Framework:**
 - ◊ Spring provides a comprehensive abstraction layer for data access that simplifies the integration with various data access technologies like JDBC, Hibernate, JPA, and more.
 - ◊ It provides transaction management and exception handling for data access operations.
- **Spring Boot:**
 - ◊ Spring Boot is a project within the Spring ecosystem that simplifies the development of new Spring applications.
 - ◊ It offers features like auto-configuration, an embedded server, and production-ready tools.

Dependency Injection

what is Dependency injection?

Dependency: An object that another object depends on. For example, if a class `Car` needs an instance of `Engine`, then `Engine` is a dependency of `Car`.

Injection: The process of passing (injecting) the dependency to a dependent object. This can be done through various methods, such as constructor injection, setter injection, or interface injection.

i) what is dependency injection?

- Dependency Injection (DI) is a design pattern used in software development to achieve Inversion of Control (IoC) between classes and their dependencies.
 - In simpler terms, it means that instead of a class creating its dependencies internally,
 - those dependencies are provided (injected) from the outside.

ii) How does it work?

- **Normal Java:** You have to manually instantiate and inject dependencies, which can be cumbersome as the application grows.

- **Spring:** Uses a container to manage the lifecycle and configuration of beans. Dependencies are automatically injected based on configuration metadata (XML, annotations, or Java configuration).

iii) Advantage of dependency injection?

- This helps to create more modular, testable, and maintainable code
- Spring promoting loose coupling, improved testability, and enhanced flexibility

iv) why use dependency injection?

Dependency Injection (DI) is a design pattern used to achieve Inversion of Control (IoC) between classes and their dependencies.

1. **Loose Coupling**
2. **Improved Testability**

3. Enhanced Flexibility and Maintainability

4. Separation of Concerns

5. Reusable Components

1. Loose Coupling

DI promotes loose coupling between classes and their dependencies. Instead of a class creating its own dependencies, dependencies are injected from the outside. This makes it easier to change and manage dependencies without modifying the class itself.

2. Improved Testability

DI makes it easier to test classes in isolation. By injecting dependencies, you can easily replace real implementations with mock or stub objects during testing, which allows for more effective unit testing.

3. Enhanced Flexibility and Maintainability

DI allows for greater flexibility in how dependencies are provided and configured. It makes it easier to switch between different

implementations of a dependency or change the configuration without modifying the dependent class. This leads to better maintainability of the codebase.

4. Separation of Concerns

DI helps in separating the creation and configuration of dependencies from the business logic of the application. This leads to a cleaner and more modular codebase where each class has a single responsibility.

5. Reusable Components

By decoupling classes from their dependencies, DI allows for the creation of more reusable and interchangeable components. This modularity facilitates code reuse and easier integration of new features.

Issues with Normal Java Class Dependencies

1. Tight Coupling

: The Car class is tightly coupled to the Engine

class. Any changes to the `Engine` implementation will require changes to the `Car` class.

2. Difficult to Test:

Testing the `Car` class in isolation is challenging because you cannot easily replace the `Engine` with a mock implementation.

3. Reduced Flexibility

: Switching to a different implementation of `Engine` requires changes in the `Car` class.

Benefits of Spring-Managed Dependencies

1. Loose Coupling

Coupling: The `Car` class is loosely coupled to the `Engine` interface. It doesn't care about the specific implementation of `Engine`.

2. Improved Testability

Testability: You can easily test the `Car` class by injecting mock implementations of the `Engine` interface using Spring's testing support.

3. Enhanced Flexibility

Flexibility: You can switch between different `Engine` implementations by changing the Spring configuration without modifying the `Car`

class.

4. Centralized

Configuration: Spring allows you to manage and configure dependencies in a centralized manner using configuration files or annotations

- **Normal Java Class Dependencies:**

Dependencies are instantiated and managed within the classes themselves, leading to tight coupling, harder testing, and reduced flexibility.

- **Spring-Managed Dependencies:**

Dependencies are managed by the Spring container, Spring uses Dependency Injection to inject dependencies, which can be configured via annotations or configuration files.

Differences Between Spring and

Normal Java for Dependency Management

1. Configuration and Wiring:

- **Normal Java:** You have to manually instantiate and inject dependencies, which can be cumbersome as the application grows.
- **Spring:** Uses a container to manage the lifecycle and configuration of beans. Dependencies are automatically injected based on configuration metadata (XML, annotations, or Java configuration).

• Dependency Injection Techniques:

- ◊ **Normal Java:** You manually write code for dependency injection, which can be done via constructors, setters, or factories.
- ◊ **Spring:** Provides a powerful and flexible DI mechanism that supports constructor injection, setter injection, and field injection out of the box, often reducing boilerplate code.

• Bean Lifecycle Management:

- ◊ **Normal Java:** You have to manage the lifecycle of objects (initialization, destruction, etc.) manually.
- ◊ **Spring:** Automatically manages the lifecycle of beans, including creation, initialization, destruction, and scope management (singleton, prototype, etc.).

• Aspect-Oriented Programming (AOP):

- ◇ **Normal Java**: Implementing cross-cutting concerns (like logging, transaction management, security) requires manual code changes across various classes.
- ◇ **Spring**: Provides AOP support to cleanly separate cross-cutting concerns from business logic, improving modularity and reducing boilerplate code.

- **Testing Support**:

- ◇ **Normal Java**: Setting up dependencies and managing them for unit tests can be cumbersome.
- ◇ **Spring**: Provides extensive support for integration and unit testing, including context configuration, mock object injection, and test-specific bean definitions.

- **Configuration Flexibility**:

- ◇ **Normal Java**: Configuration is typically done programmatically or via property files, which can be scattered and hard to manage.
- ◇ **Spring**: Supports various configuration styles (XML, Java annotations, Java configuration classes), allowing for centralized and flexible configuration management.

- **Framework Integrations**:

- ◇ **Normal Java**: Integrating various frameworks (like Hibernate, JPA, JMS) requires manual setup and configuration.
- ◇ **Spring**: Offers seamless integration with a wide range of frameworks and libraries, providing pre-configured templates and utilities to simplify

configuration and usage.

- **Infrastructure Management:**
 - ◇ **Normal Java:** Requires manual setup for transactions, security, and other infrastructural concerns.
 - ◇ **Spring:** Provides comprehensive support for managing transactions, security, and other infrastructure needs through declarative configuration

programming paradigm means ?

Yes, that's right! A programming paradigm refers to the overall structure or approach to writing code. It includes the principles, concepts, and patterns that govern how you organize and design your programs. Each paradigm has its own set of rules and conventions for expressing the logic and structure of a program.

Example :

1. **Imperative Programming**: Focuses on describing a sequence of steps to be executed by the computer. It emphasizes changing program state through assignment statements.

2. **Declarative Programming**: Focuses on describing what should be accomplished rather than how to accomplish it. Examples include functional programming and logic programming.

3. **Functional Programming**: Focuses on treating computation as the evaluation of mathematical

functions and avoids changing state and mutable data.

4. Object-Oriented Programming (OOP): Focuses on modeling real-world entities as objects that have attributes (fields) and behaviors (methods), and emphasizes encapsulation, inheritance, and polymorphism.

5. Procedural Programming: Similar to imperative programming but emphasizes procedures or routines to be called rather than the sequence of steps.

6. Aspect-Oriented Programming (AOP): Focuses on modularizing cross-cutting concerns that cut across multiple parts of the application, such as logging, security, and transaction management.

Java8

Java8

Why Use These Return Types?

Entity: Directly interact with the entity in the database. Suitable for straightforward CRUD operations.

Optional<Entity>: Handle potential absence of data gracefully.

List<Entity>: Retrieve multiple entities when you expect more than one result.

Page<Entity>: Efficiently manage large datasets by breaking them into pages. Ideal for pagination.

Slice<Entity>: Fetch portions of data without needing the total count. Good for scenarios like infinite scrolling.

Stream<Entity>: Process large datasets in a memory-efficient way.

Collection<Entity>: More generic than **List**, but serves a similar purpose.

Set<Entity>: Ensure uniqueness of results.

Entity Projection:

Optimize performance by retrieving only the necessary fields.

Entity

-> to handle one object for straightforward.

Optional<Entity>

-> to handle only one object that is object null or not . if it is null to avoid nullPointerException.

Entity Projection:

-> which table column do you want that only we can get.

-> Optimize performance by retrieving only the necessary fields.

List

-> To small datas search or

read operation list is best.

Stream

-> To handle large datas searching or reading operation Stream is best.

-> b'z stream parallel processing to handle data that is simultaneously (threads) processing multi-core process.

Page<Entity>

-> page interface provide no of product or vide show in page, size ecommerce product show like amazon.

-> it is develop by using hasNext(), hasPrevious() method to develop an interface.

-> page 0 and size 10 that is 10 products load only in that particular page.

-> total product and pages count it is handle and store . when you want you go for this.

Slice<Entity>

-> slice Interface provide infinite of

datas loading .like instagram

-> it is also to develop by using
hasNext() , hasPrevious .

-> without need no of page and show no
of datas that we use only for infinit of
scrolling.

Entity Projection

-> To retrive datas necessary fieds only.
when ever you want we should use entity
projection.

Topics

OverView

- 1.Lambda expressions.
- 2.Funtional Interface
- 3.Stream Api
- 4.Default methods
- 5.Optional Class
- 6.Date and Time.
- 7.Nashorn javaScript Engine
- 8.Parallel Array Sorting metho
- 9.Collectors
- 10.Annotation on java types
- 11.Repeating annotation.
- 12.Completabe future and the
java.util.concurrent package
enhancements.

Lambda Expression

1.Lambda expressions.

Stream API

Stream API

- 1 • **filter()**: Used in web applications to filter search results based on user criteria. collection of object la irrunth base on some condition used to filter and stroe then use this.
- 2 • **map()**: Employed in transforming data between different layers of an application (e.g., from database entities to DTOs). object ha normal to any change like lower to upper then use this.
- 3 • reduce() -> collection of element into a single result use and also. identity use first and a, result and second int product = numbers.stream().reduce(1, (a, b) -> a * b);
- 4 • Collectors.groupingBy() -> department vachi group pandrathu ,

```
Map<String, List<Employee>>
employeesByDepartment =
employees.stream() .collect(Collectors.groupingBy(Employee::getDepartment));
```
- 5 • Sorted() -> natural and customize sort pannikalam.

example

```
// Custom sorting: by department, then by salary
(descending)
```

```
List<Employee>sortedEmployees=employees.stream()
.sorted(Comparator.comparing(Employee::getDepartment)
        .thenComparing(Comparator.comparing(Employee::getSalary).reversed()))
.collect(Collectors.toList());
```

6•anyMatch() -> ethavathu match agutha check panna use.

```
// Check if any number is greater than 4
boolean anyMatch = numbers.stream()
    .anyMatch(n -> n > 4);
```

```
// Find the first number greater than 3
int firstMatch = numbers.stream()
    .filter(n -> n > 3)
    .findFirst()
    .orElse(-1);
```

7.parallelStream() -> multi-core process used to compare and more efficiently mangae .

example of all

=====

```
import java.util.Arrays;
import java.util.List;
import java.util.Optional;
```

```
import java.util.stream.Collectors;

public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4,
5, 2, 3, 4);

        // 1. filter(Predicate)
        List<Integer> filteredList = numbers.stream()
            .filter(n -> n % 2 == 0) //
Keep only even numbers
            .collect(Collectors.toList());
        System.out.println("Filtered list: " + filteredList);

        // 2. map(Function)
        List<Integer> mappedList = numbers.stream()
            .map(n -> n * n) // Square
each number
            .collect(Collectors.toList());
        System.out.println("Mapped list: " + mappedList);

        // 3. flatMap(Function)
        List<String> words = Arrays.asList("Hello",
"World");
        List<Character> characters = words.stream()
            .flatMap(word ->
word.chars().mapToObj(c -> (char) c))
            .collect(Collectors.toList());
        System.out.println("Characters: " + characters);

        // 4. distinct()
        List<Integer> distinctNumbers =
```

```
numbers.stream()
        .distinct() // Remove
duplicates
        .collect(Collectors.toList());
System.out.println("Distinct numbers: " +
distinctNumbers);

// 5. sorted()
List<Integer> sortedList = numbers.stream()
        .sorted() // Natural order
        .collect(Collectors.toList());
System.out.println("Sorted list: " + sortedList);

// 6. limit(long)
List<Integer> limitedList = numbers.stream()
        .limit(3) // Limit to first 3
elements
        .collect(Collectors.toList());
System.out.println("Limited list: " + limitedList);

// 7. skip(long)
List<Integer> skippedList = numbers.stream()
        .skip(3) // Skip first 3
elements
        .collect(Collectors.toList());
System.out.println("Skipped list: " + skippedList);

// 8. forEach(Consumer)
System.out.print("For each element: ");
numbers.stream().forEach(System.out::print);

// 9. collect(Collector)
```

```
String result = numbers.stream()
    .map(Object::toString)
    .collect(Collectors.joining(", ", "[",
    "]"));
System.out.println("\nCollected result: " + result);

// 10. reduce(identity, BinaryOperator)
Optional<Integer> sum = numbers.stream()
    .reduce(Integer::sum); // Sum of
all elements
System.out.println("Sum: " + sum.orElse(0));

// 11. anyMatch(Predicate)
boolean anyMatch = numbers.stream()
    .anyMatch(n -> n > 3); // Check if
any element is greater than 3
System.out.println("Any match: " + anyMatch);

// 12. allMatch(Predicate)
boolean allMatch = numbers.stream()
    .allMatch(n -> n > 0); // Check if all
elements are positive
System.out.println("All match: " + allMatch);

// 13. noneMatch(Predicate)
boolean noneMatch = numbers.stream()
    .noneMatch(n -> n < 0); // Check
if no element is negative
System.out.println("None match: " + noneMatch);

// 14. findFirst()
Optional<Integer> firstElement =
```

```
numbers.stream().findFirst(); // First element of the stream
```

```
    System.out.println("First element: " +  
firstElement.orElse(null));
```

```
// 15. findAny()
```

```
Optional<Integer> anyElement =  
numbers.stream().findAny(); // Any element of the stream
```

```
    System.out.println("Any element: " +  
anyElement.orElse(null));
```

```
// 16. count()
```

```
long count = numbers.stream().count(); // Count  
of elements in the stream
```

```
    System.out.println("Count: " + count);
```

```
// 17. toArray()
```

```
Integer[] array =  
numbers.stream().toArray(Integer[]::new); // Convert  
stream to array
```

```
    System.out.println("Array: " +  
Arrays.toString(array));
```

```
// 18. min(Comparator)
```

```
Optional<Integer> minElement =  
numbers.stream().min(Integer::compareTo); // Minimum  
element
```

```
    System.out.println("Min element: " +  
minElement.orElse(null));
```

```
// 19. max(Comparator)
```

```
    Optional<Integer> maxElement =  
numbers.stream().max(Integer::compareTo); //  
Maximum element  
    System.out.println("Max element: " +  
maxElement.orElse(null));  
  
    // 20. forEachOrdered(Consumer)  
    System.out.print("For each ordered element: ");  
  
numbers.stream().forEachOrdered(System.out::print);  
}  
}
```

output

=====

Filtered list: [2, 4, 2, 4]

Mapped list: [1, 4, 9, 16, 25, 4, 9, 16]

Characters: [H, e, l, l, o, W, o, r, l, d]

Distinct numbers: [1, 2, 3, 4, 5]

Sorted list: [1, 2, 2, 3, 3, 4, 4, 5]

Limited list: [1, 2, 3]

Skipped list: [4, 5, 2, 3, 4]

For each element: 12345234

Collected result: [1, 2, 3, 4, 5, 2, 3, 4]

Sum: 24

Any match: true

All match: true

None match: true

First element: 1

Any element: 1

Count: 8

Array: [1, 2, 3, 4, 5, 2, 3, 4]

Min element: 1

Max element: 5

For each ordered element: 12345234

flatMap

=====

```
List<Integer> flattenedList = listOfLists.stream()
    .flatMap(List::stream) // Flatten
the list of lists
    .collect(Collectors.toList());
```

```
System.out.println(flattenedList);
```

OPTIONAL

=====

```
import java.util.Optional;
```

```
public class Main {
```

```
    public static void main(String[] args) {
```

```
        // Creating an Optional
```

```
        Optional<String> optionalString =
```

```
Optional.of("Hello"); if the value is emty it will throw
NullpointerException
```

```
        Optional<String> optionalString =
```

```
Optional.ofNullable("Hello"); //if the value is emty it will
not throw NullpointerException
```

// isPresent(): Checks if the Optional contains a value

```
boolean isPresent = optionalString.isPresent(); // true
```

// ifPresent(Consumer): Executes the specified action if a value is present

```
optionalString.ifPresent(System.out::println); // Prints: Hello
```

// get(): Gets the value from the Optional (use with caution to avoid NoSuchElementException)

```
String value = optionalString.get(); // "Hello"
```

// orElse(T): Returns the value if present, otherwise returns the specified default value

```
String orElseValue = optionalString.orElse("Default"); // "Hello"
```

// orElseGet(Supplier): Returns the value if present, otherwise invokes the supplier to generate a default value

```
String orElseGetValue = optionalString.orElseGet(() -> "Default"); // "Hello"
```

// orElseThrow(Supplier): Returns the value if present, otherwise throws an exception generated by the supplier

```
try {
```

```
    String orElseThrowValue = optionalString.orElseThrow(() -> new
```

```
IllegalStateException("Value not present"));
} catch (IllegalStateException e) {
    System.out.println(e.getMessage()); // Prints:
Value not present
}

// map(Function): Applies a mapping function if a
value is present, otherwise returns an empty Optional
Optional<Integer> mappedOptional =
optionalString.map(String::length); // Optional[5]

// flatMap(Function): Applies a mapping function
that returns an Optional if a value is present, otherwise
returns an empty Optional
Optional<Integer> flatMappedOptional =
optionalString.flatMap(s -> Optional.of(s.length())); // 
Optional[5]

// filter(Predicate): Filters the Optional if a value is
present according to the predicate
Optional<String> filteredOptional =
optionalString.filter(s -> s.startsWith("H")); // 
Optional[Hello]
}
```

JUnit and Mockito

Junit Annotations

1. @Test

- **Purpose:** Marks a method as a test method.
- **How it works:** JUnit runs this method as a test case

2. @BeforeEach

- **Purpose:** Runs before each test method in the class.
- **How it works:** Used to set up test data or initialize resources

3. @AfterEach

- **Purpose:** Runs after each test method in the class.
- **How it works:** Used to clean up resources or reset states.

4. @BeforeAll

- **Purpose:** Runs once before all test methods in the class.
- **How it works:** Used to set up static resources shared across tests

5.

5. @AfterAll

- **Purpose:** Runs once after all test methods in the class.
 - **How it works:** Used to clean up static resources.
-
- `assertEquals(expected, actual)`: Checks if two values are equal.
 - `assertTrue(condition)`: Checks if a condition is true.
 - `assertFalse(condition)`: Checks if a condition is false.
 - `assertNull(object)`: Checks if an object is null.
 - `assertNotNull(object)`: Checks if an object is not null.
 - `assertThrows(exceptionClass, executable)`: Checks if a specific exception is thrown.

@Before

```
public void setUp() { ... }
```

@After

```
public void tearDown() { ... }
```

Mockito Annotations

1. @Mock

- **Purpose:** Creates a mock object.
- **How it works:** Mockito replaces the real object with a mock version.

2. @InjectMocks

- **Purpose:** Injects mock objects into the tested object.
- **How it works:** Mockito automatically injects mocks annotated with `@Mock` into the object annotated with `@InjectMocks`.

spring interview

Core Concepts:

1. Spring Core:

- Inversion of Control (IoC)
- Dependency Injection (DI)
- Beans and Bean Lifecycle
- ApplicationContext

2. Spring AOP (Aspect-Oriented Programming):

- Aspects
- Advice
- Pointcuts
- Joinpoints
- Weaving

3. Spring ORM (Object-Relational Mapping):

- Integration with Hibernate
- JPA (Java Persistence API) support
- Transaction Management

4. Spring Data:

- Spring Data JPA
- Spring Data MongoDB
- Spring Data Redis
- Spring Data Cassandra
- Spring Data Elasticsearch

5. Spring JDBC:

- JdbcTemplate
- NamedParameterJdbcTemplate
- SimpleJdbcInsert and SimpleJdbcCall

Web Framework:

6. Spring MVC (Model-View-Controller):

- Controllers
- Views (JSP, Thymeleaf)
- Model and ModelMap
- Request Mappings
- Form Handling
- RESTful Web Services

7. Spring WebFlux:

- Reactive Programming
- Reactive Streams
- Mono and Flux
- Functional Endpoints

Data Access and Integration:

8. Transaction Management:

- Declarative Transactions
- Programmatic Transactions

9. Messaging:

- Spring JMS (Java Message Service)
- Spring AMQP (Advanced Message Queuing Protocol)
- **Spring Kafka**

10. Spring Batch:

- **Batch Processing**

- **Job, Step, Tasklet**

- Readers, Processors, Writers
- Job Repository

11. Spring Integration:

- Enterprise Integration Patterns (EIP)
- Channels
- Message Endpoints

Security:

12. Spring Security:

- Authentication and Authorization
- Method Security
- Web Security
- OAuth2 and JWT

Cloud and Microservices:

13. Spring Cloud:

- Service Discovery (Eureka)
- Circuit Breaker (Hystrix, Resilience4j)
- Configuration Management (Spring Cloud Config)
- API Gateway (Spring Cloud Gateway)
- Cloud Messaging (Spring Cloud Stream)
- Kubernetes Integration

14. Spring Boot:

- Auto-configuration
- Embedded Servers
- Spring Boot Starters
- Spring Boot Actuator
- Spring Boot CLI

Additional Modules:

15. Spring REST Docs:

- REST API Documentation

16. Spring HATEOAS:

- Hypermedia as the Engine of Application State

17. Spring LDAP:

- LDAP Support

18. Spring Web Services:

- SOAP Web Services

Testing:

19. Spring Testing:

- Unit Testing with JUnit
- Integration Testing
- Mocking with Mockito
- Spring TestContext Framework

Utility Modules:

20. Spring Expression Language (SpEL):

- Expression Parsing and Evaluation

21. Spring Cache:

- Caching Abstraction
- Cache Providers (EhCache, Redis, etc.)

22. Spring Internationalization (i18n):

- Locale Support
- MessageSource

23. Spring Scripting:

- Scripting Languages Integration (Groovy, JRuby, etc.)

24. Spring Mobile:

- Mobile Web Application Development

25. Spring Social:

- Integration with Social Networks (Facebook, Twitter, etc.)

1. Core Container :

- IoC Container : Inversion of Control container manages Java objects.
- Dependency Injection : Allows objects to be injected at runtime.

2. Aspect-Oriented Programming (AOP) :

- Aspect : Modularization of concerns like transaction management.
- Advice : Action taken by an aspect at a particular join point.
- Join Point : Point during program execution, like a method execution or exception handling.

3. Data Access/Integration :

- JDBC : Java Database Connectivity for database

operations.

- ORM : Object-Relational Mapping integration with Hibernate, JPA, etc.
- Transaction Management : Declarative and programmatic transaction control.

4. Web :

- Spring MVC : Model-View-Controller framework for web applications.
- REST : Spring MVC RESTful web services with `@RestController`.
- WebSocket : Full-duplex communication channels over a single TCP connection.
- Security : Authentication and authorization using filters and annotations.

5. Testing :

- Unit Testing : JUnit and TestNG for unit tests.
- Integration Testing : Spring TestContext Framework for integration tests.
- Mocking : Mockito and Spring's MockMvc for mocking dependencies.

6. Miscellaneous :

- Spring Boot : Simplifies Spring application development.
- Spring Data : Simplifies database access.
- Spring Batch : Batch processing framework.
- Spring Cloud : Tools for building and deploying cloud-native applications.
- Spring Security : Authentication, authorization, and other security features.

webapp & webservice

Web Application:

1.web application is a software application and it runs on a web server and also it is accessed through a web browser over a network.

2.components:

 frontend:user interface, usually developed using html,css,javascript

 backend:the server-side logic is responsible for processing request, interacting with database, and generating dynamic content.

3.Architecture:

 web application follow client-side architecture. client is typically browser and server is nothing but powerfull computer that stores web application files, database and also logic.

 request send -> server process the request -> return the response, from the server sends the request data like(images, html, json).

 large scale web apps contains presentation layer, business layer, database layer.

4.Technologies

 framework like spring boot for java , django for python help to develop web application quickly by providing pre-build tool and structures.

 Database intergration for web application like mysql, postgresql

Web Service:

1.A web service is a type of software. and it is used to communicate machine-to-machine over a network. and also it provide **functionality** to other applications via standard web protocols.

2.**SOAP**(simple object Access protocol) : it produce data xml format and also HTTP & SMTP for message transmission.

3.**REST**(Representational state Transfer): Uses standard HTTP methods (GET,POST,PUT,DELETE) and data exchange for JSON/XML and also it is widely used.

4.two types : producer, consumer

Producer : this is like a shop that puts its services on display. To publish our services web service and make its available for others use.

Consumer:this is like a customer shopping at the store. It uses the web service by sending requests and get response back.

5.Advantages: we can access service in different platforms(windows,linux) and programming languages.they understand each others message. The services can manage many request without slowing down or crashing . they can handle lot of work efficiently.Service Can change and grow independently.If one service changes they won't affect any other service.

6.Development:

Implementation: Using framework like spring for SOAP or RESTful services using JAX-RS(Java Api for RESTfull web Services).

Documentation: swagger provides for clear documentation for API end points and usage.

Security: implementing authentication and authorization security for our web services endpoints.
(OAuth)

Examples: SOAP is used in enterprise system for complex transaction(banking).

REST is widely used in mobile application and modern web development.

web server & application server

web server is nothing but it can process front technology only.

Application server is used to both front end and backend logic process and also enterprise application runs on that server.

web Application vs Enterprise application

Web application is used for any one like ecommerce.
Enterprice application is used for only within organization people and also it is mainly used for bussiness management.

Spring Core Container

Core Container:

Core: It is a basic building blocks for the spring container. it's fundamental to how spring framework works.

Container: It is holds and manage beans and their dependencies. it is like a container to hold and organize the items.

1.IOC container: The IOC container is a central part of the spring framework.

It manages lifecycle and configuration of application Objects(bean) like creation, configuration, and management of java object(bean).

we don't have to manually manage the **lifecycle and dependencies** of objects and also it is used **dependency injection** design pattern.

ApplicationContext

context = new

AnnotationConfigApplicationContext(AppConfig.class);

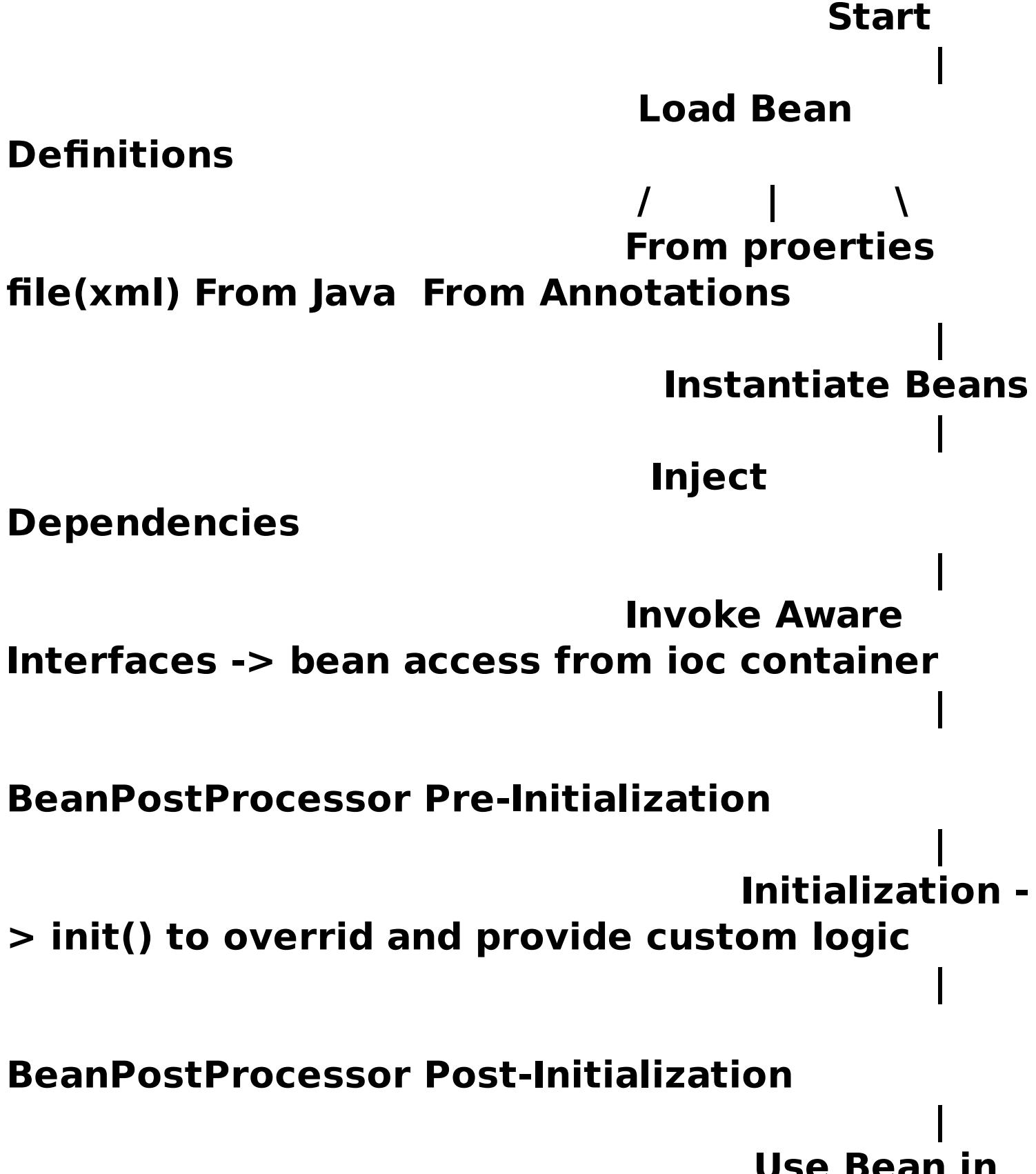
AnnotationConfigApplicationContext it is create IOC container and manage the object.

AppConfig.class it is scanning the class and create the Object @component & @Bean

2.Dependency Injection(DI): Dependency injection is a design pattern it is used by IOC container to inject object dependency at runtime by using

@Autowired annotation. DI makes easier to manage dependencies, and also promote loose coupling and enhance testability.

Life cycle of bean



Application

Destruction
End

Anotation spring core

- **@Component**
 - **@Repository**
 - **@Service**
 - **@Controller**
 - **@Autowired**
 - **@Qualifier**
 - **@Configuration**
 - **@Bean**
 - **@Value**
 - **@Scope**
 - **@PostConstruct**
 - **@PreDestroy**
 - **@Lazy**
 - **@Primary**
 - **@DependsOn**
 - **@Profile**
-
- **@Component:**
 - Used to denote a class as a Spring bean. Spring container will automatically detect and instantiate such classes.
 - **@Repository:**

Specialized form of @Component used for DAO classes. It indicates that the annotated class is a repository, typically used for database operations.

- **@Service:**
Specialized form of @Component used for service classes. It indicates that the annotated class is a service component in the business layer.
- **@Controller:**
Specialized form of @Component used for controller classes in MVC applications. It indicates that the annotated class serves as a Spring MVC controller.
- **@Autowired:**
Used for automatic dependency injection. Spring will automatically resolve and inject dependencies into the annotated fields, constructors, or methods.
- **@Qualifier:**
Used in conjunction with @Autowired to specify which bean should be injected when multiple beans of the same type are available.
- **@Configuration:**
Indicates that a class contains Spring configuration. This is typically used on classes annotated with @ComponentScan and/or @Bean.
- **@Bean:**

Used inside @Configuration classes to declare a bean. Spring container will manage the lifecycle of the bean created by a method annotated with @Bean.

- **@Value:**

Used to inject values from properties files, environment variables, or system properties into Spring beans.

- **@Scope:**

Specifies the scope of Spring beans (e.g., singleton, prototype, request, session).

- **Singleton:** One instance for the entire application.
- **Prototype:** New instance each time it's requested.
- **Request:** New instance for each HTTP request.
- **Session:** New instance for each user session (from login to logout).
- **Global Session:** Single instance across all Portlets in the same user session.
- **Application:** Single instance across the entire web application (ServletContext).

- **@PostConstruct:**

Used on methods that need to be executed after dependency injection(loc container inject the object) is done to perform any initialization.

opening a database connection, initializing configuration settings, or setting up other

```
@Service
public class MyService {

    @PostConstruct
    public void initialize() {
        // Initialization logic
        // For example, establish a database
connection or initialize other resources
        System.out.println("MyService initialized");
    }

    // Other methods and properties of MyService
}
```

- **@PreDestroy**

Purpose: Indicates a method to be invoked just before the bean is removed from the Spring context, allowing for cleanup or releasing resources.

```
@Component
public class MyService {

    @PreDestroy
    public void cleanup() {
        // Cleanup logic
        System.out.println("Bean destroyed");
    }
}
```

- **@Lazy:**

Delays the initialization of a bean **until it is first requested.**

```
import org.springframework.context.annotation.Lazy;
import org.springframework.stereotype.Component;

@Component
@Lazy
public class HeavyResource { -> this class take some time so that after application load it will object instantiate in spring container

    public HeavyResource() {
        System.out.println("HeavyResource bean initialized");
        // Simulating heavy initialization process
        try {
            Thread.sleep(2000); // Simulate heavy initialization time
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public void useResource() {
        System.out.println("Using HeavyResource...");
    }
}
```

- **@Primary:**

Used in conjunction with @Autowired to specify a primary bean when multiple beans of the same type are present.

- **@DependsOn:**

Specifies the bean dependencies. It ensures that the beans specified are initialized before the dependent bean is initialized.

```
@Component("dataSourceInitializer")
public class DataSourceInitializer {
    public DataSourceInitializer() {
        System.out.println("DataSourceInitializer
bean initialized");
    }
}
```

```
@Component("databaseService")
@DependsOn("dataSourceInitializer") -> this
line first check and object instantiate then crete this
object instatiate
public class DatabaseService {
```

```
    public DatabaseService() {
        System.out.println("DatabaseService bean
initialized");
    }
}
```

```
// Other methods and properties of
DatabaseService
```

}

- **@Profile:**

Specifies the environment-specific configurations for a bean. Beans annotated with @Profile will be loaded only when the specified profiles are active.

spring AOP

What is Aop:

Aspect-Oriented Programming (AOP) is a programming that helps you **organize your code better by separating common functionalities**. This common functionality is known as "cross-cutting concerns."

Why is important?

Aspect-Oriented Programming (AOP) helps developers **handle common tasks** like logging, security, and managing data transactions separately from the main code. This makes the **code easier to organize**, update, and keep free from unnecessary repetition. Overall, AOP improves how software is structured and makes it easier to manage as it grows.

How to implement?

1. **Dependencies**: Adds the Spring AOP starter to the project.

2. **LoggingAspect**: Defines an aspect that logs a message before any method in the specified package executes.

3. **ExampleService**: A service class with a method to demonstrate the logging aspect.

4. **DemoApplication**: The main class that runs the application and invokes the service method.

```
import  
org.springframework.boot.SpringApplication;
```

```
import  
org.springframework.boot.autoconfigure.SpringBootApplication;  
import  
org.springframework.context.annotation.EnableAspectJAutoProxy;  
  
@SpringBootApplication  
@EnableAspectJAutoProxy  
public class YourSpringBootApplication {  
  
    public static void main(String[] args) {  
  
        SpringApplication.run(YourSpringBootApplication.class,  
        args);  
    }  
}
```

Alternative way to config:

```
@Configuration  
@ComponentScan(basePackages =  
"com.example.aspect") -> which package to  
public class AppConfig { // Other configuration  
beans if needed}
```

Advantages of Aop

Modularity : **Improves code modularity by separating** cross-cutting concerns from the **business logic**. Each concern is managed in its aspect, making the **code cleaner and easier to understand**.

Maintainability: Changes to a cross-cutting concern need to be made in **only one place** (the aspect), reducing the risk of errors and improving maintainability.

Reusability: Aspects can be reused across different parts of an application or even in different applications.

Separation of Concerns: AOP promotes a clear separation of concerns, where different aspects of the program (e.g., logging, security) are handled independently.

Disadvantages

Complexity: AOP adds additional layers of understanding and management to your code due to its modular and cross-cutting nature.

Debugging Difficulty: Identifying and fixing issues becomes harder as AOP aspects can modify program flow in unexpected ways.

Performance Overhead: Implementing AOP can cause programs to run slower due to the extra processing needed for aspect weaving and execution.

Tool Support: Some development tools may not fully support AOP, making it challenging to debug, analyze, or refactor aspects.

Code Readability: Aspects can make code harder to understand and maintain, especially when aspects are scattered across different parts of the codebase.

Problem Resolution

1. Repetitive Code: Avoids writing the same code in multiple places (e.g., logging, security checks).

2) Scattered Code: Keeps cross-cutting concerns separate from core business logic.

3) Maintainability: Makes it easier to update common functionality in one place.

4) Clean Code: Keeps your main code focused on its primary tasks, improving readability.

5) Centralized Control: Allows you to manage and apply changes to cross-cutting concerns centrally.

How work in AOP?

Imagine you're a teacher grading papers in a classroom. Without AOP, you'd have to stop after each paper to record grades and comments manually. This breaks your flow and slows down the grading process.

With AOP, it's like having a grading assistant who automatically records grades and comments based on predefined rules. You set criteria for grading, such as checking for spelling errors and providing feedback on structure. The assistant handles these tasks seamlessly as you review each paper, allowing you to focus on assessing the content without interruption. This way, grading is more efficient, and you maintain consistency in feedback across all papers.

1. Cross-Cutting Concerns:

2. Aspects

3. Join Points:

4. Advice

5. Pointcuts:

Cross-cutting concerns

1. Cross-Cutting Concerns

These are functionalities that affect multiple parts of an application. Examples include logging, security, and transaction management.

Example: Imagine you want to log every method call in your application. Without AOP, you'd need to add logging code to every method, leading to scattered and duplicated logging code.

2. Aspect

An aspect is a module that encapsulates a cross-cutting concern. It contains the code (advice) that should be applied at certain points in the program.

Example: A logging aspect would encapsulate the logging logic, keeping it separate from the core business logic.

@Aspect

```
public class LoggingAspect {  
    @Before("execution(*  
com.example.service.*.*(..))")  
    public void logBefore() {  
        System.out.println("Logging before method  
execution");  
    }  
}
```

3. Join Point

The primary purpose of using `JoinPoint` in Aspect-

Oriented Programming (AOP) is to access information about the method or execution point where the advice is being applied. Here are the main purposes of using **JoinPoint**: Log Information, perform security check, monitoring and audioting.

When we use Aspect in choose like method, field, constructor etc..

Example: A join point could be any method execution within a particular package.

```
@Aspect  
public class LoggingAspect {  
  
    @Before("execution(*  
com.example.service.*.*(..))")  
    public void logBefore(JoinPoint joinPoint) {  
        System.out.println("Executing method: " +  
joinPoint.getSignature().getName());  
        System.out.println("Arguments: " +  
Arrays.toString(joinPoint.getArgs()));  
        System.out.println("Target class: " +  
joinPoint.getTarget().getClass().getName());  
    }  
}
```

4. Advice

Code that is executed at a join point. There are several types of advice: before, after, and around.

Example: An advice can log a message **before** a method execution or **after** a method.

```
@Before("execution(*  
com.example.service.*.*(..))")  
public void logBefore() {  
    System.out.println("Logging before method  
execution");  
}
```

5. Pointcut

A pointcut is an **expression** that matches join points. It defines where (in which methods or classes) the **advice should be applied**. it will apply join points where we use in that place it will apply advice.

Example: A pointcut could match all method executions in a certain package.

```
@Before("execution(*  
com.example.service.*.*(..))")  
public void logBefore() {  
    System.out.println("Logging before method  
execution");  
}
```

6. Weaving

The process of applying aspects to the target objects to create advised objects. Weaving can occur at compile-time, load-time, or runtime.

Example: The logging aspect is woven into the specified methods, **so the logging code runs automatically before these methods**. execute the above 5

```
public class ExampleService {  
    174/310
```

```
public void performTask() {  
    System.out.println("Performing task");  
}  
}
```

anotations

@Aspect

Purpose: Declares a class as an aspect.

@Before

- **Purpose:** Runs advice before a method execution.

@AfterReturning

- **Purpose:** Runs advice after a method successfully returns a value.

Imagine you manage a store, and you want to log the details of each sale after it is completed. You have a `SalesService` class that handles sales, and you want to log the sale details automatically after the `completeSale` method is executed.

```
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;
import
org.springframework.stereotype.Component;
```

```
@Aspect
```

```
@Component
```

```
public class LoggingAspect {
```

```
    @AfterReturning(
```

```
        pointcut = "execution(*
```

```
com.example.service.SalesService.completeSale(..))",
```

```
        returning = "result"
```

```
)
```

```
    public void logAfterSale(String result) {
```

```
// Log the result after the sale is completed  
System.out.println("After sale advice: " +  
result);  
}  
}
```

@AfterThrowing

- **Purpose:** Runs advice after a method throws an exception.

Imagine you manage a bank, and you want to log details whenever a money transfer fails due to an error. You have a `BankService` class that handles money transfers, and you want to log the error details automatically whenever the `transferMoney` method throws an exception.

```
package com.example.aspect;
```

```
import org.aspectj.lang.annotation.AfterThrowing;  
import org.aspectj.lang.annotation.Aspect;  
import org.springframework.stereotype.Component;
```

```
@Aspect
```

```
@Component
```

```
public class LoggingAspect {
```

```
    @AfterThrowing(
```

```
        pointcut = "execution(*
```

```
com.example.service.BankService.transferMoney(..))",  
        throwing = "error"
```

```
)
```

```
    public void logAfterThrowing(Exception error)
```

```
{  
    // Log the error details  
    System.out.println("Error occurred: " +  
error.getMessage());  
}  
}
```

@After

- **Purpose:** Runs advice after a method (regardless of outcome).

```
@After("execution(* com.example.service.*.*(..))")  
public void afterAdvice(JoinPoint joinPoint) {  
    // Advice logic  
}
```

@Pointcut

- **Purpose:** Declares a reusable pointcut (expression that matches certain method executions).

```
@Pointcut("execution(* com.example.service.*.*(..))")  
public void serviceMethods() {}
```

```
@Before("serviceMethods())"  
public void beforeAdvice(JoinPoint joinPoint) {  
    // Advice logic  
}
```

@Order

- **Purpose:** Specifies the order in which aspects should be applied.

In programming, `@Order` works similarly. It specifies the sequence (or order) in which aspects (special tasks that handle certain parts of your code) should be applied. This helps ensure that aspects are executed in the correct sequence, which can be crucial for maintaining the integrity and functionality of your software.

```
@Aspect  
@Component  
@Order(1)  
public class LoggingAspect {  
    @Before("execution(* com.example.service.*.*(..))")  
    public void logBeforeMethodExecution() {  
        System.out.println("Logging: Before executing  
method...");  
    }  
}
```

```
@Aspect  
@Component  
@Order(2)  
public class ExceptionHandlingAspect {  
    @AfterThrowing(pointcut = "execution(*  
com.example.service.*.*(..))", throwing = "ex")  
    public void handleException(Exception ex) {  
        System.out.println("Exception occurred: " +  
ex.getMessage());  
        // Additional handling logic can be added here  
    }
```

```
}
```

```
@Aspect  
@Component  
@Order(3)  
public class AuthorizationAspect {  
    @Before("execution(* com.example.service.*.*(..))")  
    public void authorizeMethodExecution() {  
        System.out.println("Authorization: Checking user  
permissions...");  
        // Authorization logic here  
    }  
}
```

Data integration/Access

What is Spring ORM?

Spring ORM (Object-Relational Mapping) is a framework that helps integrate ORM tools like Hibernate with Spring applications. It allows you to use Java objects to interact with a relational database.

Why is it Important?

Spring ORM is important because it makes database interactions simpler and more efficient. It provides a way to manage database connections, handle transactions, and perform data operations in a consistent and straightforward manner.

What Problem Does Spring ORM Resolve?

Spring ORM resolves the complexity of managing database operations manually. Without ORM, you have to write a lot of boilerplate code to handle database connections, queries, and transactions. Spring ORM automates these tasks, reducing the amount of code you need to write and maintain.

How to Implement Spring ORM?

Step-by-Step Implementation:

1. Set Up Dependencies:

- Add the necessary Spring ORM and Hibernate dependencies to your project (usually via Maven or Gradle).

2. Configure Data Source:

- Define a DataSource bean to specify database connection details.

3. Configure SessionFactory:

- Set up a SessionFactory bean to manage sessions for interacting with the database.

4. Set Up Transaction Management:

- Configure a transaction manager bean and enable annotation-driven transaction management.

5. Create Entity Classes:

- Define Java classes that represent your database tables and map them using annotations.

6. Write DAO Classes:

- Create Data Access Object (DAO) classes that use SessionFactory to perform database operations.

How Does It Work?

Working Mechanism (Imagine):

1. Connecting to the Database:

- Imagine your application has a power cable (DataSource) that connects to a power source (the database).

2. Creating Sessions:

- Imagine a factory (SessionFactory) producing workers (sessions) who handle database tasks like saving and retrieving data.

3. Managing Transactions:

- Imagine a safety net ensuring all tasks complete successfully before they're permanently applied. If something goes wrong, it rolls everything back.

4. Performing Operations:

- Imagine tools (HibernateTemplate) handling common tasks like opening a session and managing transactions automatically.

Advantages and Disadvantages

Advantages:

1. Simplified Database Access:

- Imagine a toolbox that handles complex tasks for you, making it easier to interact with the database.

2. Reduced Boilerplate Code:

- Imagine not having to write repetitive code, allowing you to focus on business logic.

3. Consistent Transaction Management:

- Imagine a reliable system managing your database transactions, ensuring data consistency.

4. Integration with Spring:

- Imagine seamlessly integrating ORM with the powerful features of the Spring framework.

Disadvantages:

1. Learning Curve:

- Imagine having to learn new concepts and configurations, which might be challenging for beginners.

2. Performance Overhead:

- Imagine additional layers of abstraction potentially impacting performance in highly demanding applications.

3. Complex Configuration:

- Imagine the initial setup requiring detailed configurations, which can be time-consuming.

By using Spring ORM, you can manage database interactions more efficiently, allowing you to focus on building your application rather than dealing with low-level database operations.

What is Spring Data JPA?

Spring Data JPA is a part of the larger Spring Data family, which provides convenient data access layers for various kinds of databases. Specifically, Spring Data JPA simplifies the implementation of JPA (Java Persistence API) repositories, reducing the amount of

boilerplate code needed for data access and manipulation.

Why is it Important?

1. Reduced Boilerplate Code: Spring Data JPA handles a lot of the repetitive code involved in data access.
2. Consistency and Integration: It integrates seamlessly with the rest of the Spring ecosystem, ensuring consistent configuration and behavior across the application.
3. Rapid Development: Enables developers to focus more on business logic rather than on data access implementation details.
4. Flexibility: Offers both automatic query generation from method names and the ability to write custom queries.

How Does It Work?

Spring Data JPA works by extending the JPA specification and using repository interfaces to handle data access. Key components include:

1. Repository Interfaces: You define interfaces that extend JpaRepository or CrudRepository, and Spring Data JPA provides the implementation at runtime.
2. Automatic Query Generation: Based on method names in the repository interface, Spring Data JPA can automatically generate queries.
3. Custom Queries: You can define custom queries using the @Query annotation for more complex data

access requirements.

How to Implement?

1. Add Dependencies: Include the Spring Data JPA and database-specific dependencies in your pom.xml (for Maven) or build.gradle (for Gradle).

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</
artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</
artifactId>
</dependency>
```

2. Configure Data Source: Set up your data source properties in application.properties or application.yml.

```
spring.datasource.url=jdbc:mysql://localhost:3306/
mydb
spring.datasource.username=root
spring.datasource.password=secret
spring.jpa.hibernate.ddl-auto=update
```

3. Define Entity Class: Create your JPA entity class.

```
@Entity  
public class User {  
    @Id  
    @GeneratedValue(strategy =  
GenerationType.IDENTITY)  
    private Long id;  
    private String name;  
    private String email;  
    // getters and setters  
}
```

4. Define Repository Interface: Create a repository interface for your entity.

```
public interface UserRepository extends  
JpaRepository<User, Long> {  
    List<User> findByName(String name);  
}
```

5. Use Repository: Autowire and use the repository in your service layer.

```
@Service  
public class UserService {  
    @Autowired  
    private UserRepository userRepository;  
  
    public List<User> findUsersByName(String name)  
{
```

```
        return userRepository.findByName(name);
    }
}
```

Advantages of Spring Data JPA

1. Simplifies Data Access: Significantly reduces the amount of code required for data access layers.
2. Automatic Query Generation: Methods in repository interfaces can automatically generate queries based on method names.
3. Consistency: Ensures consistent data access patterns and configuration across the application.
4. Integration: Seamless integration with other Spring components like Spring Boot and Spring Security.
5. Custom Queries: Supports custom queries using JPQL, SQL, and the @Query annotation.

Disadvantages of Spring Data JPA

1. Complexity for Simple Applications: For very simple applications, the added complexity of JPA may not be necessary.
2. Abstraction Overhead: The abstractions provided by Spring Data JPA can sometimes hide the underlying complexity and behavior, leading to potential performance issues.
3. Learning Curve: Requires understanding of both Spring Data and JPA, which can be steep for beginners.
4. Limited Control: Automatic query generation can limit fine-grained control over queries, especially in complex scenarios.

Problems it Resolves

1. Boilerplate Code: Reduces the amount of repetitive code needed for CRUD operations.
2. Query Complexity: Simplifies query creation with derived query methods.
3. Transaction Management: Integrates with Spring's transaction management, ensuring consistent and reliable data access.
4. Integration: Facilitates easy integration with other Spring components and external systems.
5. Flexibility: Provides a flexible and consistent way to access and manipulate data, which can be tailored to specific needs through custom queries and repository methods.

annotation

```
@Entity // Marks the class as a database entity
@Table(name = "user") // Specifies the table name in
the database
@NamedQuery(
    name = "User.findByUsername",
    query = "SELECT u FROM User u WHERE u.username
= :username"
) // Defines a named query to find a user by username
public class User {

    @Id // Marks the field as a primary key
    @GeneratedValue(strategy =
GenerationType.IDENTITY) // Specifies that the primary
key is auto-generated by the database (using auto-
increment in MySQL, PostgreSQL, etc.)
    private Long id;

    @Column(name = "username", nullable = false,
unique = true) // Specifies column details for the
'username' field
    private String username;

    @Column(name = "password", nullable = false) //
Specifies column details for the 'password' field
    private String password;

    @Temporal(TemporalType.DATE) // Specifies the
temporal type for a date field (in this case, 'birthDate')
    private Date birthDate;
```

@Lob // Marks the field as a large object (typically used for storing large text or binary data)

```
private String bio;
```

@OneToOne // Defines a one-to-one relationship with the 'Profile' entity

```
@JoinColumn(name = "profile_id") // Specifies the foreign key column name in the database  
private Profile profile;
```

@OneToMany(mappedBy = "user") // Defines a one-to-many relationship with the 'Post' entity, where 'user' is the field in the 'Post' entity that maps back to this 'User' entity

```
private List<Post> posts;
```

@ManyToOne // Defines a many-to-one relationship with the 'Department' entity

```
@JoinColumn(name = "department_id") // Specifies the foreign key column name in the database  
private Department department;
```

@ManyToMany // Defines a many-to-many relationship with the 'Role' entity

```
@JoinTable(
```

```
    name = "user_role", // Specifies the join table name in the database
```

```
    joinColumns = @JoinColumn(name = "user_id"), // Specifies the foreign key column name for this 'User' entity
```

```
    inverseJoinColumns = @JoinColumn(name =
```

```
"role_id") // Specifies the foreign key column name for  
the 'Role' entity  
)  
private Set<Role> roles;
```

@Transient // Ignores the 'temporaryField' for persistence (it won't be stored in the database)

```
private String temporaryField;
```

Purpose: The **@Transient** annotation is used to indicate that a field should not be persisted in the database. When an entity is saved, updated, or retrieved, the fields marked with **@Transient** will be ignored by JPA.

@Embedded // Embeds the 'Address' embeddable class into this entity

```
private Address address;
```

Explanation: Indicates that the annotated field (address in this case) is an embedded object. An embedded object is stored as part of the owning entity's table.
Usage: Useful when you have a complex object (Address class) that logically belongs to another entity (User), but you want it to be stored together in the same table.

@Embedded

@AttributeOverride(name = "street", column = @Column(name = "billing_street")) // Overrides the column mapping for the 'street' attribute of the 'billingAddress' embedded object

```
private Address billingAddress;  
  
// Getters and setters  
}
```

Derived Query Methods: Spring Data JPA can create queries based on method names.

@Query Annotation: Write custom JPQL or SQL queries directly in the repository interface.

Named Queries: Define named queries in the entity class and reference them in the repository.

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

```
public interface UserRepository extends  
JpaRepository<User, Long> {  
    // This will automatically generate the query  
    "SELECT u FROM User u WHERE u.username = ?1"  
    User findByUsername(String username);  
}
```

```
import  
org.springframework.data.jpa.repository.JpaRepository;  
import  
org.springframework.data.jpa.repository.Query;  
import  
org.springframework.data.repository.query.Param;  
public interface UserRepository extends  
JpaRepository<User, Long> {
```

```
    @Query("SELECT u FROM User u WHERE  
u.username = :username")  
        User findByUsername(@Param("username")  
String username);  
    }  
    import javax.persistence.*;  
  
@Entity  
@Table(name = "user")  
@NamedQuery(  
    name = "User.findByUsername",  
    query = "SELECT u FROM User u WHERE u.username  
= :username"  
)  
public class User {  
  
    @Id  
    @GeneratedValue(strategy =  
GenerationType.IDENTITY)  
    private Long id;  
  
    @Column(name = "username", nullable = false,  
unique = true)  
    private String username;  
  
    @Column(name = "password", nullable = false)  
    private String password;  
  
    // Getters and setters  
}
```

```
import  
org.springframework.data.jpa.repository.JpaRepository;  
import org.springframework.data.jpa.repository.Query;  
import  
org.springframework.data.repository.query.Param;  
  
public interface UserRepository extends  
JpaRepository<User, Long> {  
  
    @Query(name = "User.findByUsername")  
    User findByUsername(@Param("username") String  
username);  
}  
public User findUserByUsername(String username) {  
    EntityManager em = getEntityManager();  
    return em.createNamedQuery("User.findByUsern-  
ame", User.class)  
        .setParameter("username", username)  
        .getSingleResult();  
}
```

ExplanationMethod Declaration: public User findUserByUsername(String username) This is a public method that returns a User object.

It takes a String parameter called username.

Getting the EntityManager: EntityManager em = getEntityManager(); EntityManager is used to interact with the persistence context.

getEntityManager() is a method that returns an instance of EntityManager. (This method is not shown

in your code, but it's assumed to be defined elsewhere in your class.)Creating a Named Query:return em.

createNamedQuery("User.findByUsername", User.class)createNamedQuery is a method of EntityManager that creates a query using a named query defined in the User entity.

"User.findByUsername" is the name of the query defined with the @NamedQuery annotation in the User entity class.User.class specifies that the result of the query should be mapped to the User entity.

Setting the Parameter:.setParameter("username", username)setParameter is used to set the value of the named parameter :username in the query.The first argument is the name of the parameter in the query ("username").

The second argument is the value to bind to the parameter (username, which is the method argument).Executing the Query:.getSingleResult(); getSingleResult executes the query and returns a single result.If the query finds a matching user, it returns the User entity.If no matching user is found, it throws a NoResultException.If more than one result is found, it throws a NonUniqueResultException.

```
public class UserService {
```

```
    @PersistenceContext
```

```
    private EntityManager em;
```

```
    public List<User> findAllUsers() {
```

```
        TypedQuery<User> query =  
        em.createNamedQuery("User.findAll", User.class);
```

```
    return query.getResultList();
}
}
```

Explanation EntityManager

Injection:@PersistenceContext

private EntityManager em;

@PersistenceContext injects the EntityManager into the UserService class.

Creating the Named Query:TypedQuery<User> query

= em.createNamedQuery("User.findAll", User.class);

createNamedQuery creates a query using the named query User.findAll.User.class specifies the result type of the query.

Executing the Query:return query.getResultList();

getResultSet executes the query and returns the result as a list of User entities.

Spring MVC

Spring MVC Overview

what is MVC?

Spring MVC is a framework for building **web applications** in Java, using the Model-View-Controller (MVC) pattern.

Concepts Explained

1. Web Framework

- A platform to develop and run web applications.

2. Model-View-Controller (MVC)

- Model: Holds application data and business logic.
- View: Displays data to the user (e.g., HTML, JSP, Thymeleaf).

- Controller: Handles user input, updates Model and View.

3. Controllers

- Classes annotated with @Controller.
- Manage incoming web requests and return views.

4. Views (JSP, Thymeleaf)

- Templates to render data into HTML.

5. Model and ModelMap

- Model: Holds data for the view.
- ModelMap: A map structure to pass data from

controller to view.

6. Request Mappings

- `@RequestMapping`: Maps HTTP requests to controller methods.

7. Form Handling

- Manages form submissions and validates user input.

8. RESTful Web Services

- Services that follow REST principles, using `@RestController` and `@RequestMapping`.

Why These Concepts are Important

- Separation of Concerns: Divides the application into distinct sections (Model, View, Controller), making it easier to **manage and maintain**.
- Reusability: Components can be reused across the application.
- Scalability: Simplifies scaling the application by separating different concerns.

Implementation Steps

1. Set up Spring MVC Project: Create a new Spring project using tools like Spring Boot.
2. Define Models: Create classes to represent the data.
3. Create Controllers: Write classes with `@Controller` to handle web requests.
4. Design Views: Use JSP, Thymeleaf, or other templating engines to create the user interface.

5. Map Requests: Use `@RequestMapping` to link web requests to controller methods.
6. Handle Forms: Manage user input through forms and validate data.

Advantages of Spring MVC

- Modularity: Breaks application into modules.
- Testability: Easier to test components individually.
- Flexibility: Supports different types of user interfaces, such as mobile apps, desktop applications, and web pages.

Disadvantages of Spring MVC

- Complexity: Can be complex for beginners.
- Configuration: Requires configuration, though Spring Boot simplifies it.

Common Problems Resolved by Spring MVC

- Maintainability: Easier to maintain code with clear separation.
- Scalability: Supports scaling with modular design.
- Flexibility: Adapts to various view technologies and development needs.

1. Spring MVC is a framework used to build web applications in Java using the Model-View-Controller (MVC) pattern.

2. Spring MVC is important because dividing the

application into different parts (Model, View, Controller) makes it easier to manage and maintain.and also

3.Components can be reused within the application.

4.It simplifies scaling the application by using different parts (Model, View, Controller).

5.We can implement the MVC pattern by creating a Spring Boot project and adding the necessary dependencies, including the MVC starter. Then, we write entity, controller, and view classes. When a request comes into the controller, it handles the request and displays the data on the view page.

6. The main advantage of Spring MVC is that it breaks the application into modules,

7. Easier to test components individually.

8.it supports various types of user interfaces, including desktop, mobile applications, and web applications

how works

How Spring MVC Works

Spring MVC operates on a series of steps that align with the MVC pattern, involving Controllers, Views, and Models.

Step-by-Step Workflow

1. Client Request

- A user sends a request from their browser to the web server.

2. DispatcherServlet

- The request first hits the DispatcherServlet, which is the central controller in Spring MVC. It's configured in the web application's configuration file (web.xml or via Java configuration).

3. Handler Mapping

- DispatcherServlet consults HandlerMapping to find the appropriate controller method to handle the request.

4. Controller

- The mapped controller method processes the request. It may involve business logic, interacting with services, and updating the Model with data.

5. Model and View

- The controller returns a ModelAndView object. The Model contains data for the view, and the View is the template name (e.g., JSP, Thymeleaf).

6. **View Resolver**

- DispatcherServlet uses a ViewResolver to resolve the view name to the actual view template.

7. **Render View**

- The resolved view template (like a JSP page) is rendered with the data from the Model.

8. **Response to Client**

- The rendered view (HTML) is sent back to the user's browser as a response.

Example Flow

1. Client Request: User requests <http://example.com/home>.
2. DispatcherServlet: Receives the request and delegates to HandlerMapping.
3. Handler Mapping: Finds the controller method mapped to /home.
4. Controller: Method processes the request, updates the Model with data (e.g., user information).
5. Model and View: Controller returns ModelAndView("home", model).
6. View Resolver: Resolves home to home.jsp.
7. Render View: home.jsp is rendered with data from the Model.
8. Response to Client: Rendered HTML is sent back to the client.

the browser.

Diagram

Client -> DispatcherServlet -> HandlerMapping ->
Controller -> ModelAndView -> ViewResolver -> View ->
Response to Client

anotation

- **@Controller:** Indicates that a class is a Spring MVC controller.
- **@RequestMapping:** Maps HTTP requests to handler methods of MVC and REST controllers. Can be used at class level and method level.
- **@GetMapping:** Specializes the @RequestMapping annotation to handle HTTP GET requests.
- **@PostMapping:** Specializes the @RequestMapping annotation to handle HTTP POST requests.
- **@PutMapping:** Specializes the @RequestMapping annotation to handle HTTP PUT requests.
- **@DeleteMapping:** Specializes the @RequestMapping annotation to handle HTTP DELETE requests.
- **@PatchMapping:** Specializes the @RequestMapping annotation to handle HTTP PATCH requests.
- **@RequestParam:** Binds a web request parameter to a method parameter in the controller.
- **@PathVariable:** Binds a URI template variable to a method parameter.

- **@ModelAttribute**: Binds a method parameter or method return value to a named model attribute and then exposes it to a web view.
- **@RequestBody**: Binds the body of the web request to a method parameter.
- **@ResponseBody**: Binds the return value of a method to the web response body.
- **@ResponseStatus**: Marks a method or exception class with the status code and reason that should be returned.
- **@ExceptionHandler**: Defines a method that handles exceptions thrown by handler methods.
- **@RestController**: A convenience annotation that combines @Controller and @ResponseBody, indicating that the class handles RESTful web services.
- **@InitBinder**: Identifies methods to initialize the WebDataBinder, which is used for data binding from web request parameters to JavaBean objects.
- **@SessionAttributes**: Specifies the names of model attributes that should be stored in the session.
- **@CookieValue**: Binds the value of an HTTP cookie to a method parameter.
- **@RequestHeader**: Binds the value of an HTTP

header to a method parameter.

- **@CrossOrigin**: Enables cross-origin resource sharing (CORS) on the annotated handler methods or class.

anotations

What is @Controller?

The @Controller annotation in Spring Framework marks a class as a controller where methods are used to handle web requests.

Why is it important?

@Controller is important because it helps in creating web applications by handling HTTP requests and returning appropriate responses. It simplifies the development of web applications by managing request and response handling.

How does it work?

- When a request is sent to a web application, the Spring DispatcherServlet intercepts it.
 - The DispatcherServlet then looks for a controller class annotated with @Controller.
 - It finds the appropriate method in the controller to handle the request based on the URL mapping.
 - The method processes the request, interacts with the service layer if needed, and returns a view or response.
-
- ~

What is @RequestMapping?

@RequestMapping is an annotation used in the Spring Framework to **map web requests** to specific handler

classes or handler methods. It is used to specify **the URL patterns** that a controller class or a specific method will handle.

Why is it important?

@RequestMapping is important because it allows you to define the routes that different parts of your application will handle. It provides a clear and concise way to map URLs to the appropriate controller methods, making the application more organized and easier to maintain.

Mapping URLs: @RequestMapping maps HTTP requests to handler methods **based on the URL patterns**.

What is @GetMapping?

@GetMapping is an annotation in the Spring Framework that maps HTTP GET **requests** to specific handler methods. It is a **shortcut** for @RequestMapping(method = RequestMethod.GET)

Why is it important?

@GetMapping is important because it **simplifies** the process of **mapping GET requests to methods in your controller**. It makes your code more readable and easier to understand by explicitly indicating that a method is intended to **handle GET requests**.

How does it work?

- Annotation: You annotate a method with `@GetMapping` to specify that it should **handle GET requests for a particular URL**.
- URL Mapping: The URL pattern is defined within the `@GetMapping` annotation. When a **GET request matches this pattern**, the annotated method is invoked.

Disadvantages

- Specific to GET: **Only handles GET requests**, so you'll need other annotations for different HTTP methods (like `@PostMapping` for POST requests).
- Less Flexibility: Provides **less flexibility compared to `@RequestMapping`** when handling complex request mappings.

~~~~~

~

- **What is `@PostMapping`?**

- `@PostMapping` is an annotation in Spring that tells your application to handle HTTP POST requests for a specific URL.

- **Why is it important?**

It is important because it allows your application to accept data from forms, apps, or other clients and process it, such as saving user information or handling uploads.

## **How does it work?**

- When a POST request is sent to your server, `@PostMapping` matches the URL of the request to a specific method in your controller. This method then processes the request data.

•

## **How to implement?**

- To use `@PostMapping`, you add it above a method in your Spring controller

## **What problems does this resolve?**

- It helps in handling data sent via POST requests in a structured way. This is useful for processing form submissions, saving data to a database, and handling uploads or API requests.

~~~~~  
~

• What is `@PutMapping`?

- `@PutMapping` is an annotation in Spring that tells your application to handle HTTP PUT requests for a specific URL.

• Why is it important?

It is important because it allows your application to update existing resources. This is useful for modifying data that already exists, like updating user details or changing product information.

- **How does it work?**

When a PUT request is sent to your server, @PutMapping matches the URL of the request to a specific method in your controller. This method then processes the request data and updates the corresponding resource.

How to implement?

- To use @PutMapping, you add it above a method in your Spring controller

What problems does this resolve?

- It helps in handling updates to existing resources in a structured way. This is useful for modifying existing data, such as updating user information or changing details of a product.
-
-

- **What is @DeleteMapping?**

- @DeleteMapping is an annotation in Spring that tells your application to handle HTTP DELETE requests for a specific URL.

- **Why is it important?**

It is important because it allows your application to delete existing resources. This is useful for removing

data, such as deleting a user account or removing a product from inventory.

- **How does it work?**

When a DELETE request is sent to your server, @DeleteMapping matches the URL of the request to a specific method in your controller. This method then processes the request and deletes the corresponding resource.

- **How to implement?**

To use @DeleteMapping, you add it above a method in your Spring controller

What problems does this resolve?

- It helps in handling the deletion of resources in a structured way. This is useful for removing unwanted or obsolete data, such as deleting user accounts or removing products from a catalog.

What is @PatchMapping?

@PatchMapping is an annotation in the Spring Framework used to map HTTP PATCH requests to specific handler methods in a Spring controller. It

allows for partial updates to resources.

Why is it important?

@PatchMapping is important because it allows for more efficient and targeted updates to resources. Instead of sending a complete representation of a resource, you can send only the fields that need to be updated, reducing the amount of data transmitted and processed.

How does it work?

When an HTTP PATCH request is made to a specific URL, the @PatchMapping annotation maps that request to a handler method in a Spring controller. The method processes the request and applies the specified updates to the resource

How to implement?

To implement @PatchMapping, you need to:

1. Annotate a method in your Spring controller with @PatchMapping.
2. Define the endpoint and the method parameters (such as @PathVariable for the resource ID and @RequestBody for the updates).
3. Implement the logic to apply the partial updates to the resource.

What are the problems resolved by using this?

- **Reducing Bandwidth Usage:** By sending only the changes instead of the entire resource, bandwidth usage is minimized.
 - **Improving Performance:** Less data processing leads to better performance, especially for large resources.
 - **Enhancing User Experience:** Faster updates improve the user experience, particularly in applications with frequent or real-time updates.
-
-

What is **@RequestParam**?

`@RequestParam` is an annotation in the Spring Framework used to extract query parameters from HTTP requests and bind them to method parameters in a Spring controller.

Why is it important?

`@RequestParam` is important because it allows easy access to query parameters in HTTP requests, making it simpler to handle and process data sent by clients through the URL.

How does it work?

When an HTTP request with query parameters is made, `@RequestParam` maps those parameters to method arguments in the controller

How to implement?

To implement `@RequestParam`, you need to:

1. Annotate a method parameter in your Spring controller with `@RequestParam`.
2. Specify the name of the query parameter you want to bind.
3. Optionally, set default values or make the parameter optional.

What are the problems resolved by using this?

- Accessing Query Parameters: Simplifies the extraction and usage of query parameters in HTTP requests.
- Parameter Validation: Ensures required parameters are provided and allows setting default values for optional parameters.

~~~~~

~

## What is `@PathVariable`?

- `@PathVariable` is an annotation in Spring MVC used to extract values from the URI path and bind them to method parameters in a controller.

## Why is it important?

It allows for dynamic handling of web requests where parts of the URL can be variable. This makes the web application more flexible and capable of handling different types of requests with a single method.

## **How does it work?**

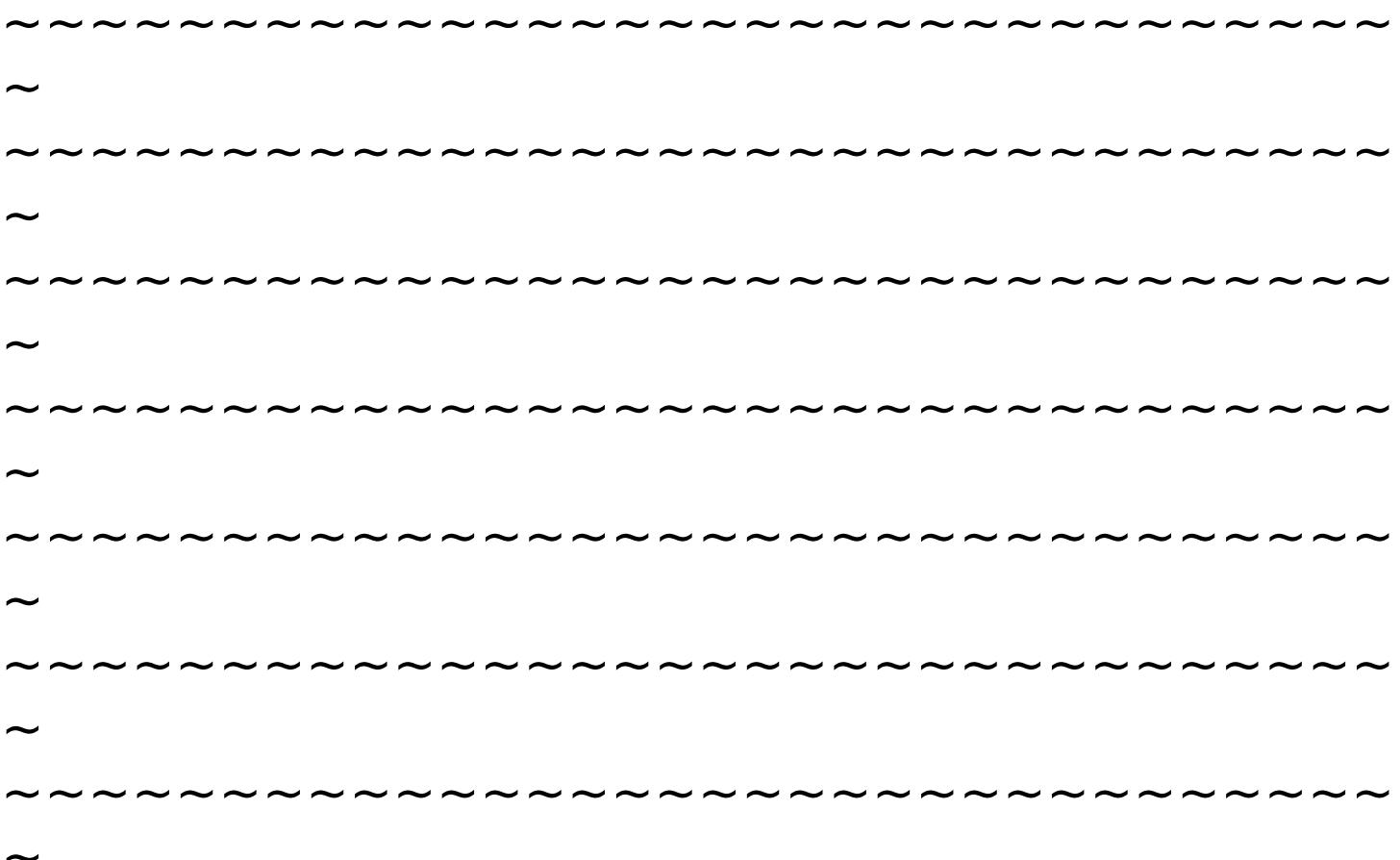
When a URL request is received, the value in the URL path that matches the variable part of the path pattern is captured and passed to the method parameter annotated with `@PathVariable`.

## **How to implement?**

Define a controller method and use `@PathVariable` to bind parts of the URL to method parameters.

## **What problems does it resolve?**

- **Dynamic URL Handling:** Enables dynamic and RESTful URL structures.
- **Simplified Code:** Reduces the need for multiple handler methods for different URL patterns.





# **difference**

- **@ModelAttribute:**
    - Used for binding form data (HTML form submissions) to a Java object.
    - Automatically binds form fields to object properties based on their names.
    - Suitable for handling traditional HTML form submissions.
  - **@RequestBody:**

Used for extracting and deserializing JSON or XML data from the request body.  
Converts JSON or XML data directly into a Java object.  
Suitable for handling data sent via AJAX requests or in RESTful API scenarios where data is typically in JSON or XML format.
- 
- ~

# **Spring Boot**

## What is Spring Boot?

Spring Boot is a framework built on top of the Spring Framework. It simplifies the development of stand-alone, production-grade Spring-based applications by providing default configurations and a variety of tools to streamline the process.

## Why is it Important?

Spring Boot is important because it simplifies the setup and development of new Spring applications. It reduces the amount of boilerplate code, configuration, and setup needed to start a new project, making it quicker and easier to get applications up and running.

## How Does it Work?

Spring Boot works by providing:

- **Auto-Configuration:** Automatically configures your Spring application based on the dependencies you have added. For example, if you include the spring-boot-starter-web dependency, it will automatically set up a web server.
- **Embedded Servers:** Includes embedded servers like Tomcat or Jetty, so you can run your applications directly without needing a separate server setup.
- **Spring Boot Starters:** Pre-configured sets of dependencies for different use cases (e.g., web, data access) that simplify the addition of required libraries.

- **Spring Boot Actuator:** Adds monitoring and management capabilities to your application.
- **Spring Boot CLI:** Command-line tool to quickly bootstrap new Spring applications.

## Advantages

- **Quick Setup:** Reduces setup and configuration time.
- **Embedded Servers:** No need for external servers.
- **Simplified Dependency Management:** Spring Boot Starters make it easy to add dependencies.
- **Production-Ready Features:** Actuator provides metrics, health checks, and more.
- **Scalability:** Built on top of the robust Spring Framework.

## Disadvantages

**Complexity for Simple Projects:** Might be overkill for very simple applications.

**Hidden Magic:** Auto-configuration might sometimes make it harder to understand what's going on under the hood.

## Problems and Resolutions

**Over-Auto-Configuration:** If auto-configuration does not fit your needs, you can exclude certain configurations manually.javaCopy code

```
@SpringBootApplication(exclude =
```

**Performance Issues:** Can be mitigated by careful profiling and tuning.

**Learning Curve:** The initial learning curve can be steep, but comprehensive documentation and community support can help.

# **Notes in CoderUllagam**

servlet mapping config

spring configuration

server config

dependency management the above all spring boot application used to overcome the boiler plate code

spring boot feautures?

- > automatic configuration

- > starter depedency

- > actuator -> beans configured in runtime check,  
spring boot auto configuration

1.Thymelef for frontend dependency

2.each steps run pani pathukanum

3.template folder kulla create view page name

4.H2 database dependency add panita need not to configure application properties file. automatically store that database

5.spring.h2.console.enabled=true just this add is enough

after endpoint and jdbc name show in log file get and

**<http://localhost:8080/h2-console>** then his show

that after jdbc link past after no change anything default ha vitudu

profile

=====

1.spring.profiles.active=qa which means application-

qa.properties - hypen apram enna kudukuriyo athutha profile ha consider pannikum. this is first way to profile set

2. second way set profile right-click-> runConfiguration-> corresponding app la profiles show athula set pannikalam

3.3rd way command line build and jar or war after right-click-> properties path copy then commandline java -jar -Dspring.profiles.active=qa home-SNAPSHOT-2.34.jar

4.Restlet extension or postman tool, json formate extensiion

spring security

=====

default username =user and password is generated in console

1.spring.security.username=hari

2.spring.security.password=mani

basically request reach before controller check spring security in 15 filter chain after comes in controller.

3.logout panna security <http://localhost:8080/logout> -> spring provide default also

# **Log4J**

## Comprehensive Overview of Log4j Concepts and Topics

### 1. Introduction to Log4j

- Purpose of Logging : Why logging is important in software development.
- History of Log4j : Evolution and versions (Log4j 1.x, Log4j 2.x).

### 2. Core Concepts

- Loggers : Components that capture logging information.
- Appenders : Components that define where the log output goes (e.g., console, files, databases).
- Layouts : Components that define the format of the log messages.

### 3. Configuration

- Configuration Files : Using XML, JSON, YAML, and properties files for configuring Log4j.
- Programmatic Configuration : Setting up Log4j configuration through code.
- Configuration Parameters : Detailed configuration options for loggers, appenders, and layouts.

### 4. Loggers

- Logger Hierarchy : Understanding parent-child

relationships and inheritance in loggers.

- Logger Levels : Trace, DEBUG, Info, Warn, Error, Fatal, and Off.
- Creating Loggers : How to create and use loggers in an application.
- Logger Methods : Various methods provided by Log4j for logging (e.g., `logger.DEBUG()`, `logger.info()`).

## 5. Appenders

- Types of Appenders : Commonly used appenders like ConsoleAppender, FileAppender, RollingFileAppender, JDBCAppender, and more.
- Appender Configuration : How to configure different appenders.
- Custom Appenders : Creating and configuring custom appenders.

## 6. Layouts

- PatternLayout : Using pattern strings to format log messages.
- XMLLayout : Formatting logs in XML format.
- JSONLayout : Formatting logs in JSON format.
- HTMLLayout : Formatting logs in HTML format.
- Custom Layouts : Creating and configuring custom layouts.

## 7. Filters

- Introduction to Filters : Purpose and use of filters in Log4j.
- Types of Filters : LevelRangeFilter, ThresholdFilter, MarkerFilter, and more.

- Filter Configuration : How to configure filters for loggers and appenders.

## 8. Logging Context

- ThreadContext (MDC) : Mapping diagnostic context, managing contextual information.
- Nested Diagnostic Context (NDC) : Using NDC for per-thread context data.

## 9. Performance

- Asynchronous Logging : Improving performance by using asynchronous logging.
- Log4j 2 Asynchronous Logging : Configuring and using Log4j 2's asynchronous logging capabilities.
- Benchmarking : Measuring and optimizing Log4j performance.

## 10. Advanced Topics

- Log4j 1.x vs. Log4j 2.x : Differences, improvements, and migration strategies.
- Custom Components : Developing custom loggers, appenders, and layouts.
- Plugins : Using and creating plugins in Log4j 2.
- JMX : Monitoring and managing Log4j via Java Management Extensions.

## 11. Integration

- Spring Framework : Integrating Log4j with Spring.
- Apache Commons Logging : Using Log4j as the backend for Commons Logging.
- SLF4J : Integrating Log4j with Simple Logging Facade for Java (SLF4J).

- Other Frameworks : Integration with other Java frameworks and libraries.

## 12. Security

- Log4j Vulnerabilities : Understanding and mitigating security risks.
- Best Practices : Ensuring secure logging practices.

## 13. Best Practices

- Effective Logging : Guidelines for logging effectively in an application.
- Log Management : Strategies for managing and analyzing logs.
- Error Handling : Logging errors and exceptions appropriately.

## 14. Troubleshooting

- Common Issues : Identifying and resolving common Log4j problems.
- DEBUGging Configuration : Techniques for DEBUGging Log4j configuration issues.

## 15. Documentation and Resources

- Official Documentation : Utilizing the official Log4j documentation.
- Community Resources : Forums, tutorials, and community support.

Each of these topics encompasses various subtopics and detailed aspects, providing a comprehensive understanding of Log4j and its usage in Java applications.

<https://chatgpt.com/c/>

05e166c8-2e00-45a0-911e-1f7b253e594a

### 3 components

1.Logger(**which classess**) -> it is used to service and controller class only check logs need not set entity dao class.

2.Appender (**where to store**)-> it is used to where we store that log like file, database, console, SMTP etc...

3.Layout (**Layout**)-> it is used to store that log in which format like xml, Html, pattern format

ex: 2022-11-09 9:50 - EmployeeController - saveEmp() -INFO - Employee

# **Introduction**

## **What is Log4j?**

Log4j is a **tool used** in Java programming **to record log messages**. These **messages** help track what an application is doing, making it **easier to find and fix problems**.

why it is important?

- **DEBUGging:** Helps find errors in the code.
- **Monitoring:** Keeps an eye on application performance.
- **Maintenance:** Makes it easier to maintain the application by understanding its behavior.
- **Security:** Logs can help detect unauthorized activities.

How to implement?

- **Add Log4j Library:** Include Log4j in your project, typically using a dependency manager like Maven or Gradle.
- **Configure Log4j:** Create a configuration file (like log4j.properties or log4j.xml) to specify how and where to log messages.

**Write Log Messages:** Use Log4j in your code to log messages. For example

```
import org.apache.log4j.Logger;
```

```
public class MyApp {  
    private static final Logger logger =  
        Logger.getLogger(MyApp.class);
```

```
public static void main(String[] args) {  
    logger.info("Application started");  
    // Your code here  
    logger.error("An error occurred");  
}  
}
```

## How does it works?

Log4j **captures log messages** from your application and sends them to different destinations like the **console, files, or databases**. It uses a configuration file to know what messages to capture and where to send them.

## Advantages of Log4J?

- **Easy to Use:** Simple API for logging messages.
- **Configurable:** Flexible configuration for different logging needs.
- **Performance:** Efficiently handles logging without slowing down the application.
- **Extensible:** Can be extended to support custom logging behaviors.

## Disadvantages of Log4J?

**Complex Configuration:** Setting up **advanced configurations can be tricky**.

**Security Risks:** Misconfigurations can expose sensitive **information**.

**Resource Usage:** Excessive logging can use up **disk space and CPU**.

Problem to resolve?

- **Finding Errors:**
  - **Problem:** Hard to find where and why the application is failing.
  - **Solution:** Log4j records error messages and details, making it easier to spot and fix issues.
- **Tracking Application Activity:**
  - **Problem:** Can't see what the application is doing at any moment.
  - **Solution:** Log4j logs important **actions and events**, helping you monitor the application's behavior.
- **Keeping Records:**
  - **Problem:** Need to keep track of what happened for future reference or legal reasons.
  - **Solution:** Log4j keeps a **history of activities and changes**, providing a **record for audits**.
- **Improving Performance:**
  - **Problem:** Difficult to find what is slowing down the application.
  - **Solution:** Log4j logs performance data, helping **identify and fix slow parts** of the application.
- **Communication Tracking:**
  - **Problem:** Hard to follow how different parts of the application talk to each other.

**Solution:** Log4j logs interactions between components, making it easier to trace and DEBUG communication issues.

- **Security Monitoring:**

**Problem:** Difficult to spot security breaches or suspicious activities.

**Solution:** Log4j logs security-related events, helping detect and investigate security problems.

# **Core concept**

1. FATAL
2. ERROR
3. WARN
4. INFO
5. DEBUG
6. TRACE (if supported)

## **//Levels**

- **DEBUG level:** Captures DEBUG, INFO, WARN, ERROR, and FATAL messages.
- **INFO level:** Captures INFO, WARN, ERROR, and FATAL messages.
- **WARN level:** Captures WARN, ERROR, and FATAL messages.
- **ERROR level:** Captures ERROR and FATAL messages.
- **FATAL level:** Captures only FATAL messages.

## Core Concepts of Log4j

1. **Loggers:** Parts of the application that collect log messages.
2. **Appenders:** Parts that decide where the log messages go (e.g., console, files).
3. **Layouts:** Parts that determine how the log messages look.

## How does it work?

- **Loggers** capture messages from the application.
- **Appenders** decide where to send these messages

(e.g., to a file or the console).

- **Layouts** format these messages to make them readable.

# **Configuration**

## How to Implement?

### 1. Add Log4j to Your Project:

- Use a build tool like Maven or Gradle to include Log4j.

### • Configure Log4j:

Create a configuration file (like `log4j.properties` or `log4j.xml`).

Define your loggers, appenders, and layouts in this file.

```
log4j.rootLogger= DEBUG, console, file, database
```

#### **//console**

```
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d [%t] %-5p %c - %m%n
```

#### **//file**

```
log4j.appender.file=org.apache.log4j.FileAppender
log4j.appender.file.File=logfile.log
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d [%t]
%-5p %c - %m%n
```

#### **//jdbc**

```
log4j.appendер.database=org.apache.log4j.jdbc.JDBCA-
ppender
log4j.appendер.database.URL=jdbc:mysql://localhost:
3306/logs
log4j.appendер.database.driver=com.mysql.jdbc.Driver
log4j.appendер.database.user=root
log4j.appendер.database.password=password
log4j.appendер.database.sql=INSERT INTO logs (date,
thread, level, category, message) VALUES('%d', '%t',
'%p', '%c', '%m')
log4j.appendер.database.layout=org.apache.log4j.Patt-
ernLayout
log4j.appendер.database.layout.ConversionPattern=%d
[%t] %-5p %c - %m%n
```

## **Use Log4j in Your Code:**

- Import Log4j classes and create a logger instance.
- Use the logger to record messages.

## **Example log4j.properties:**

```
import org.apache.log4j.Logger;

public class MyApp {
    private static final Logger logger =
Logger.getLogger(MyApp.class);

    public static void main(String[] args) {
        logger.info("Application started");
        // Your code here
        logger.error("An error occurred");
```

```
    }  
}
```

## Note:

====

- Sends logs to the console (org.apache.log4j.ConsoleAppender).
- Uses PatternLayout to format logs with date, thread ID, log level, logger category, and log message (%d [%t] %-5p %c - %m%n).

# Logger

## Logger Hierarchy

- **Definition:** Loggers in Log4j are organized in a hierarchical structure, where each logger inherits **settings and behaviors** from its **parent logger**.
- **Why It's Important:** Helps manage **logging settings more efficiently** across different parts of an application.

## Logger Levels

- **Levels:** Trace, DEBUG, INFO, WARN, ERROR, FATAL, and OFF.
- **Function:** Each level corresponds to the severity of the message being logged, allowing developers to **filter and prioritize logs based on importance**.

## Logger Methods

- **Logging Methods:** logger.debug(), logger.info(), logger.warn(), logger.error(), logger.fatal().
- **Purpose:** Each method logs a message at a specific level, helping developers capture different types of events and errors.

## How to Implement

1. **Add Log4j to Your Project:** Include Log4j dependencies using **Maven** or Gradle.
2. **Configure Loggers:** Define logger hierarchy and levels in a configuration file (**log4j.properties** or **log4j.xml**).
3. **Create Loggers:** In your Java code, create loggers

using `Logger.getLogger("LoggerName")`.

**4. Use Logger Methods:** Use logger methods (`logger.debug()`, `logger.info()`, etc.) to log messages at appropriate levels.

## How Does It Work

- **Hierarchy:** Loggers inherit settings from their parent loggers, allowing for centralized configuration.
- **Levels:** Messages are filtered based on their `severity (level)`, ensuring only relevant logs are captured.
- **Methods:** Logging methods send messages to `configured appenders` (e.g., console, file) based on their level.

# ***Spring Ashok It***

Spring framework introductin

=====

Programming Language

Core Java (Language Syntaxes, fundamentals)

-> Using this core java we will build GUI/CUI based application

> Graphical User Interface

> Commandline User Interface (CLI)

-> Adv Java

JDBC : Database Connectivity -> API

Servlets: Web applications developement -> web technology

JSP: Web pages development -> web technology

→ Frameworks

Struts -> controller layer only we build

Hibernate -> persitence only DB

Spring Framework -> application framework end to end app.

-> Framework is a semi software which provides some common logics which are required for applications development

- load jdbc driver
- get connection
- create stmt
- execute query
- close connection

application to database

-> Frameworks are divided into 2 types

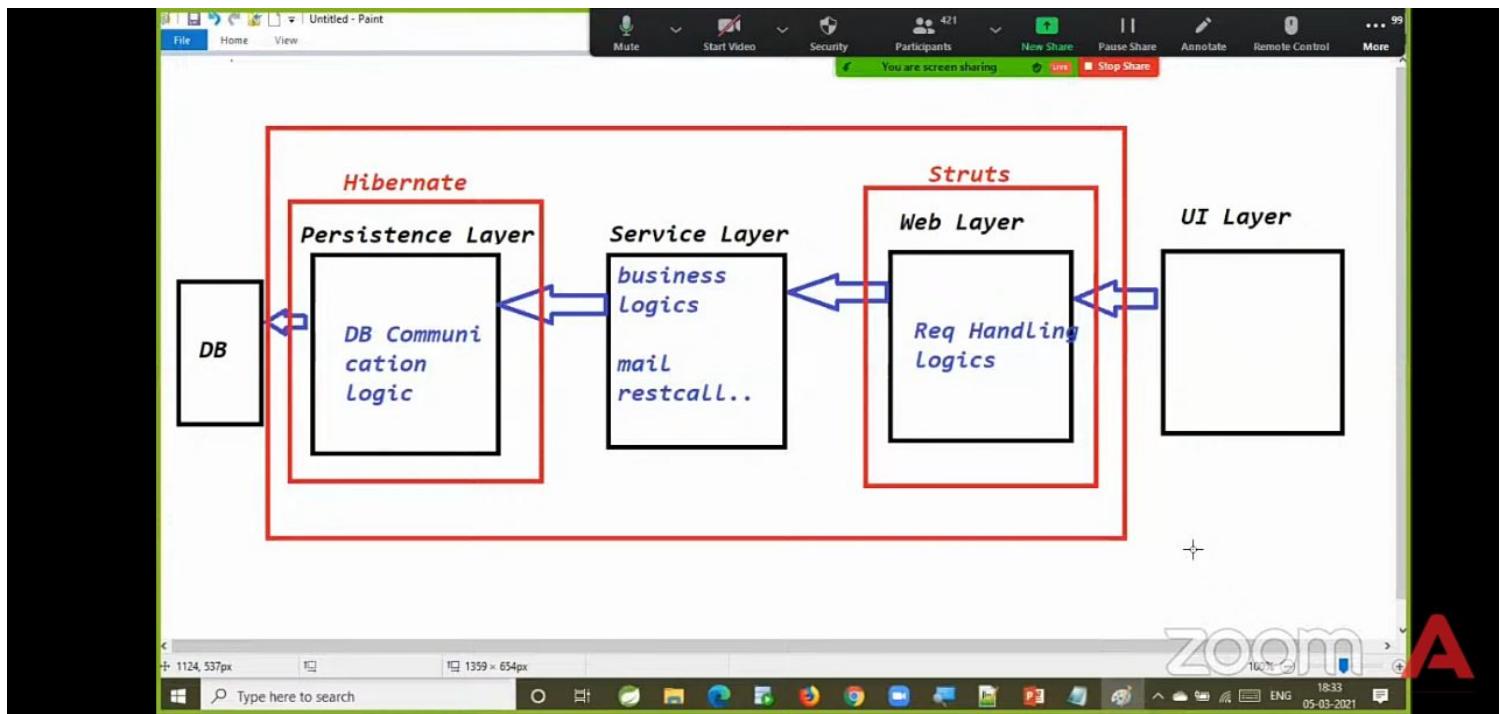
- 1) ORM Framework
- 2) Web Framework

-> ORM (Object Relational Mapping) frameworks are used to develop Persistence Layer in applications.

Ex: Hibernate

-> Web frameworks are used to develop Web applications

Ex: Struts



- > Spring is an application development framework.
- > Spring is Non Invasive framework. Spring it will not force the programmer to extend/implement any framework related classes and interfaces.
- > By Using Spring framework we can develop End to End application.
- > Spring is versatile framework. Spring can be integrated with any other framework which is available in the market.
- > Spring is loosely coupled framework. It is developed in modular fashion.
- > Spring is not a single framework. It is collection of frameworks.
- > Spring framework is developed in Modular Fashion.

- > Spring Core
- > Spring Context
- > Spring AOP
- > Spring DAO

-> Spring Web MVC

-> Spring ORM

-> Spring Core is Base Module for Spring Framework. This module providing fundamental concepts of Spring they are IOC and DI.

IOC: Inversion of Control

DI: Dependency Injection

-> Spring Context module will take care of Configurations required in our applications.

-> Spring AOP module is used to separate business logic and secondary logic in our application.

Aspect Oriented Programming

-> Spring DAO/Spring JDBC module is used to develop Persistence Layer. Spring JDBC module is developed on top of JDBC api.

-> Spring Web MVC module is used to web applications.

Spring ORM module is used to develop Persistence

Layer.

## ORM> Object Relational Mapping

It is used to represent data in the form of objects

# **Spring Core Modules**

- > Spring core is base module in Spring Framework
- > All the modules in the Spring are developed on top of Spring Core Module
- > Spring core module is providing fundamental concepts of Spring framework they are IOC & DI.

IOC: Inversion Of Control

DI: Dependency Injection

-> When we develop our application using layered architecture our classes will become dependent.

-> One java class should talk to another java class to process the request.

How one java class can talk to another java class?

One java class can talk to another java class in 2 ways

1) Inheritance

2) Composition

-> If we use Inheritance one java class extend the properties from another java class then we can call

super class methods in sub class directly.

Note: Here Inheritance option already utilized for HttpServlet so we can't extend

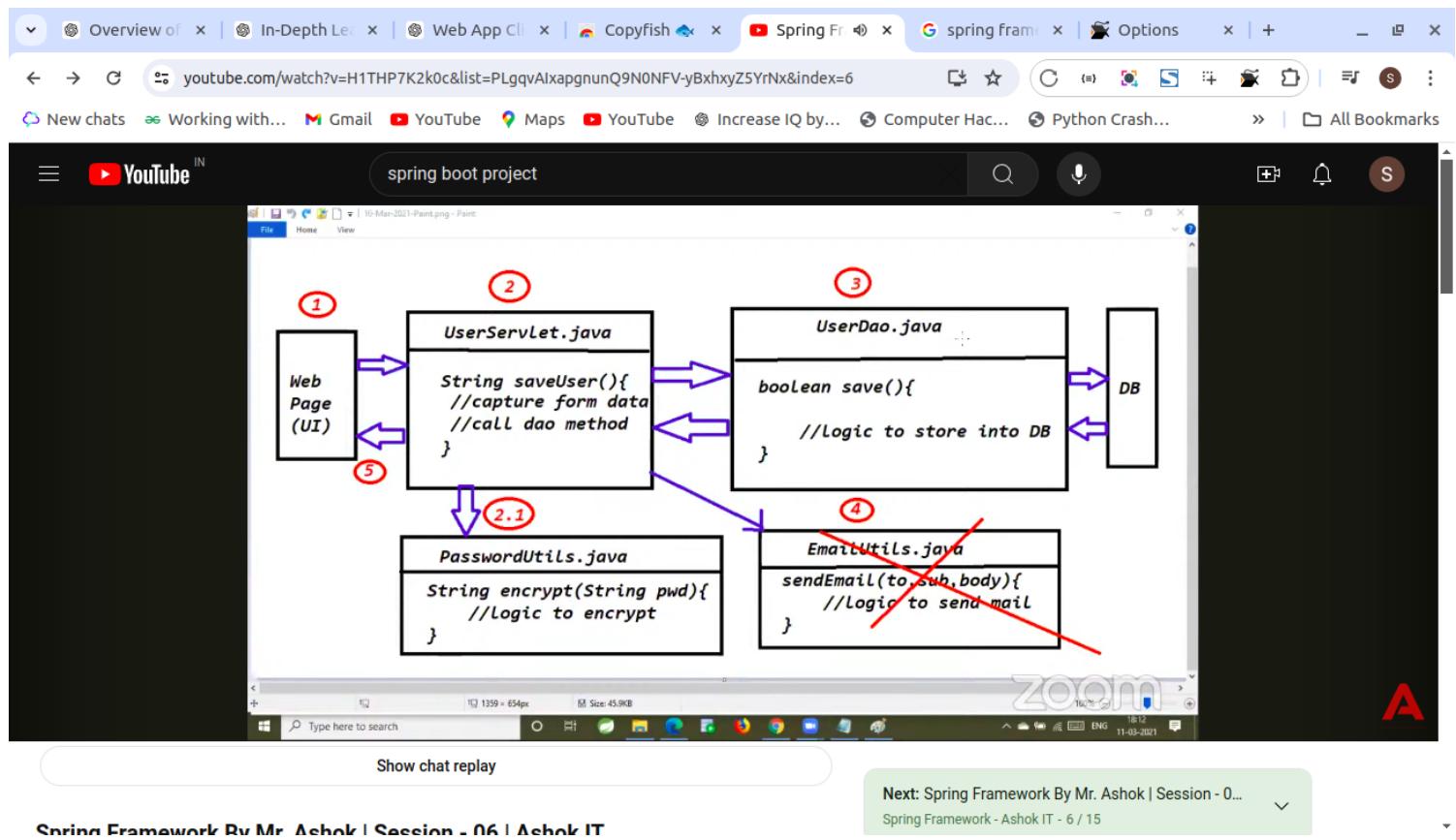
the properties from any other class (Gate is closed).

->By using composition one java class can call the methods of another java class.

```
public class UserRegServlet extends HttpServlet{  
    public void doPost(Req, Res) {  
        PasswordUtils pwd= new PasswordUtils();  
        UserDao dao new UserDao();      EmailUtils email =  
        new EmailUtils(); //call methods here  
    }  
}
```

-> In the above approach we are directly creating objects for PasswordUtils, UserDao and EmailUtils in our servlet class.

-> If we create objects for other classes directly then our classes will be tightly coupled then Maintenance of the project will become difficult.



- > To avoid this tightly coupling problem Spring Provided Core Module.
- > Spring Core Module is all about managing dependencies among the classes in the application and making our classes as loosely coupled.
- > Spring Core Module provided IOC container and DI to manage dependencies among the classes in the application.
- > If we give any two classes for IOC to manage it can't manage with loosely coupling.
- > IF we want IOC to manage our classes with Loosely coupling then we need to develop our classes by

following strategy design pattern.

## Strategy Design Pattern

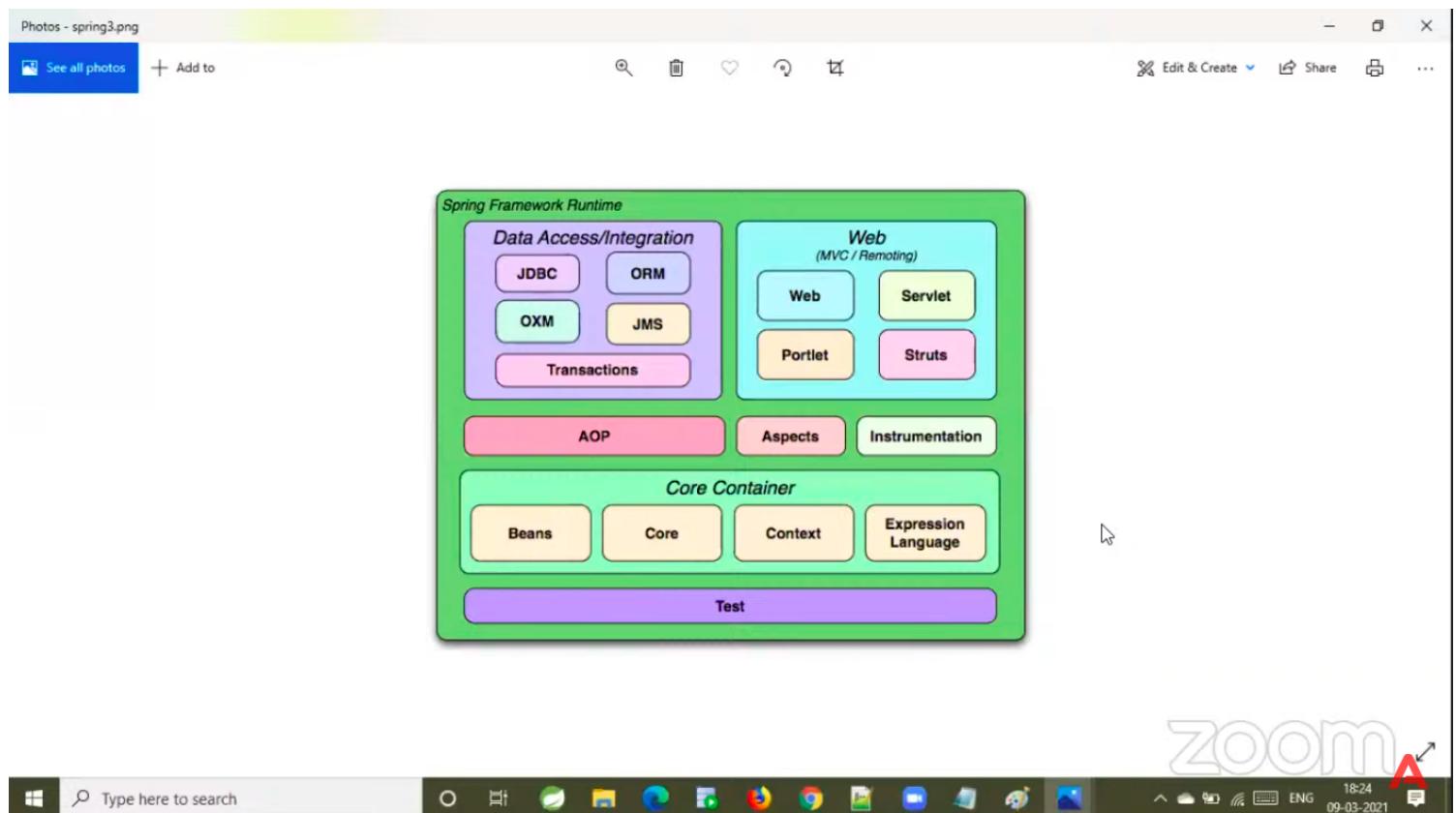
# ***Strategie pattern***

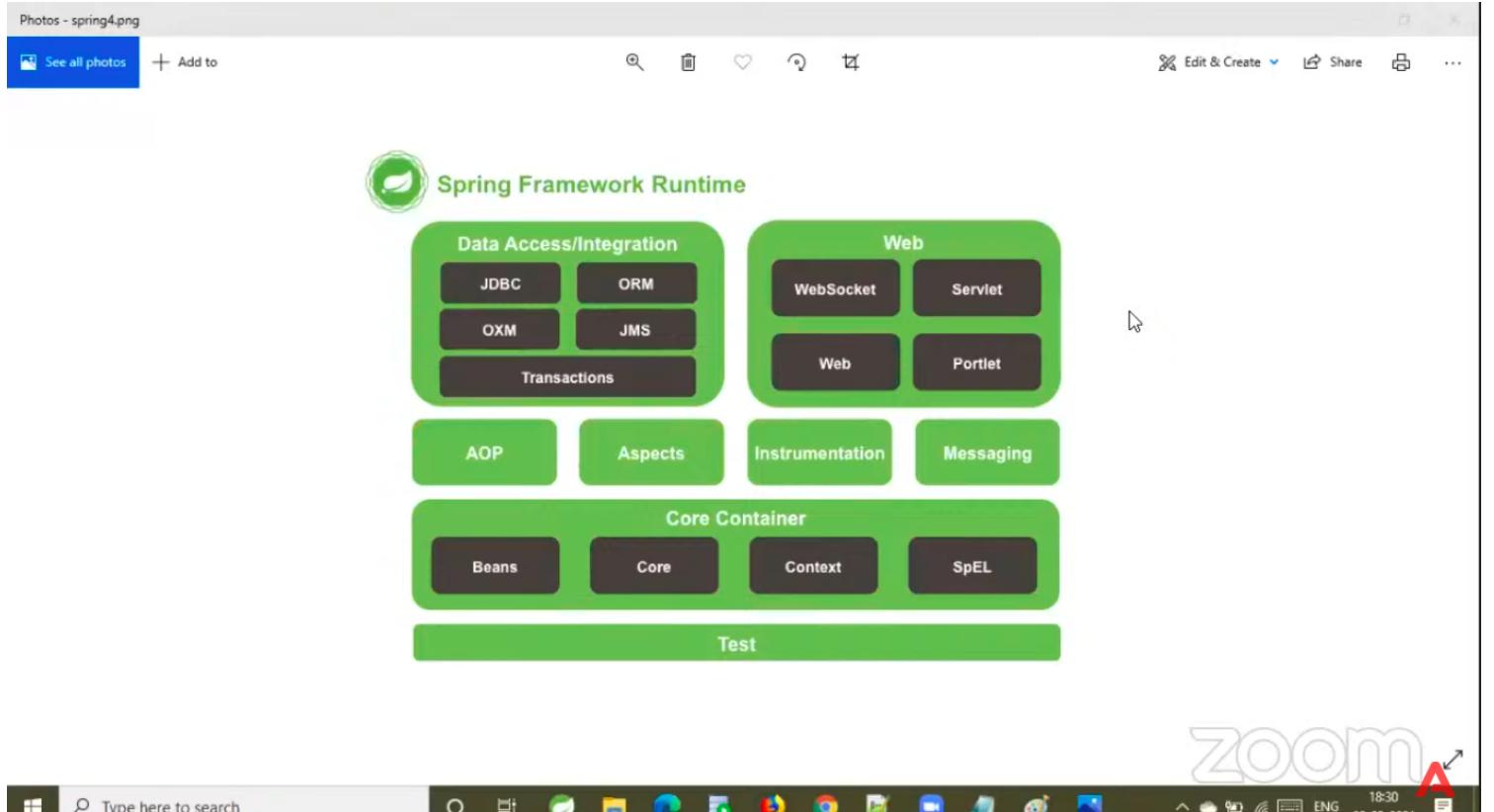
# ***Spring framework Architecture***

- > Spring DAO / Spring JDBC
- > Spring Web MVC
- > Spring ORM
- > Spring Core is the base module for spring framework.
- > Spring Context module deals with Configurations in application
- > Spring AOP module is used to separate business logic and cross-cutting logics
- > Spring DAO module is used to develop Persistence Layer.
- > Spring Web MVC module is used to web applications & distributed applications
  
- > Spring Core is the base module for spring framework.
- > Spring Context module deals with Configurations in application.

- > Spring AOP module is used to separate business logic and cross-cutting logics.
- > Spring DAO module is used to develop Persistence Layer.
- > Spring Web MVC module is used to web applications & distributed applications.
- > Spring ORM module is used to develop persistence layer using ORM principles

S





zoom A

The screenshot shows a Windows taskbar with various pinned icons and a system tray indicating the date and time (09-03-2021). The browser window displays a diagram of the Spring Framework architecture, which is structured as follows:

- Spring Core:** Supporting utilities, Bean container.
- Spring DAO:** Transaction infrastructure, JDBC support, DAO support.
- Spring ORM:** Hibernate support, Beans support, JDO support.
- Spring Context:** Application context, UI support, Internationalization, JNDI, EJB support & Remoting, Mail.
- Spring Web MVC:** Web application framework, Web views, JSP / Velocity, PDF / Excel.

A zoomed-in view of the chat replay sidebar is visible on the right, showing a list of messages from participants like b jagan, naresh kumar, siva prasad, and Krishna Mishra.

- > In Spring 1.x version we have 7 modules
- > In Spring 2.x version we have 6 modules
- > In Spring 3.x version we have 18+ modules
- > In Spring 4.x version Messaging concept got added
- > In Spring 5.x version Reactive Programming added

# **JavaProjectsDocument**

[action/description][DataType] - variable name followed

- **retrievedData**
- **retrievedValue**
- **fetchedResult**
- **obtainedInformation**
- **receivedData**
- **acquiredResource**
- **accessedRecord**
- **pulledData**
- **gatheredInfo**
- **loadedContent**
  
- **setData**
- **assignedValue**
- **insertedElement**
- **placedItem**
- **storedValue**
- **updatedInformation**
- **assignedData**
- **addedEntry**
- **putValue**
- **assignedContent**
  
- **verificationResult**
- **validationOutcome**

- **inspectionStatus**
- **examinationResult**
- **assessmentOutcome**
- **evaluationStatus**
- **scrutinyResult**
- **analysisOutcome**
- **confirmationStatus**
- **authenticationResult**

# **ControllerClass**

## Controller class

---

1. interface venam ,  
interface complicate use our  
controller class

2. class name must  
functionality name and end suffix  
controller add.

3. class only have that class  
related methods only.

4. single responsibility  
pattern compare to groupin which  
groupin is best practice and maintain  
code.

6. method name prifix with  
"handle" and action name .

7. controller class method  
return type RespnseEntity is best  
aproch ,

8. each method have single

action only.

9. class name prefix with name customer, client or corporate after functionality Name for example

"CustomerHomeController", "UserPaymentController"

10. method name prefix with action and add name of the class prefix like " addCustomer" , "processUserPayment"

11. Exception Handling class also write in controller class.

example:

=====

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class UserController {

    // Class level comment explaining the purpose of the UserController
    // This controller handles HTTP requests related to users.

    /**
     * Handles HTTP DELETE requests to delete a user by ID.
     * @param id The ID of the user to be deleted.
     * @return ResponseEntity indicating the success of the operation.
     */
    @DeleteMapping("/{id}")
    public ResponseEntity<User> handleDeleteUser(@PathVariable Long id) {
        // Invoke the service method to delete the user by ID
        userService.deleteUser(id);
        // Return ResponseEntity with HTTP status indicating successful deletion
        return new ResponseEntity<>(user, HttpStatus.NO_CONTENT);
    }
}
```



# **ServiceClass**

Service

=====

SOLID principle

=====

=

S - single responsibility that is class have only one functionality.

O - Open / Closed principle that is existing modules add new functionality that is add new class and existing module without affect

any functionality.

L - Liskov principle that is superclass method implement class compulsory override and also override method use otherwise method not using it define another interface.

I - Interface segregation that is every interface have single method or multiple that functionality related group of method.

D - dependency injection which means loose coupling

# ***RepositoryClass***

repository

=====

just interface only create  
in entity class related.

# **ProjectStructure**

## Project Structure

---

```
src/
└ main/
    └─ java/
        └─ com/
            └─ yourcompany/
                └─ yourproject/
                    ├ Application.java
                    ├ config/
                    ├ controller/
                    ├ service/
                    ├ repository/
                    ├ model/
                    ├ dto/
                    └ exception/
    └ resources/
        ├ application.properties (or application.yml)
        ├ static/
        └ templates/
```



```
src/
└── main/
    ├── java/
    │   └── com/
    │       └── example/
    │           ├── controller/
    │           │   ├── UserController.java
    │           │   └── ProductController.java
    │           ├── service/
    │           │   ├── UserService.java
    │           │   └── ProductService.java
    │           └── model/
    │               ├── User.java
    │               └── Product.java
    └── repository/
        ├── UserRepository.java
        └── ProductRepository.java
    └── resources/
        ├── static/
        │   └── css/
        │       └── styles.css
        ├── templates/
        │   ├── user/
        │   │   ├── user-form.html
        │   │   └── user-details.html
        │   └── product/
        │       ├── product-form.html
        │       └── product-details.html
        └── application.properties
└── test/
    └── java/
        └── com/
            └── example/
                ├── controller/
                │   ├── UserControllerTest.java
                │   └── ProductControllerTest.java
                └── service/
                    ├── UserServiceTest.java
                    └── ProductServiceTest.java
```



# **EndPoint Name**

## **EndPoint Name**

---

1.End point name compulsory noun that is name of things and also provide plurals forms like "users"

2.user is singular which means we don't write b'z in every case json provide collection of users so that complexity it will create.

## **Products:**

---

1.GET /products - Retrieve a list of products

2.GET /products/{id} - Retrieve a specific product

3.POST /products - Create a new product

4.PUT /products/{id} - Update an

existing product

5.DELETE /products/{id} - Delete a specific product

6.GET /products/search?query={keyword} - Search for products based on a keyword



Sure, here are some common RESTful API endpoints for an eCommerce platform:

## 1. Products:

- `GET /products` - Retrieve a list of products
- `GET /products/{id}` - Retrieve a specific product
- `POST /products` - Create a new product
- `PUT /products/{id}` - Update an existing product
- `DELETE /products/{id}` - Delete a specific product
- `GET /products/search?query={keyword}` - Search for products based on a keyword

## 2. Categories:

- `GET /categories` - Retrieve a list of categories
- `GET /categories/{id}` - Retrieve a specific category
- `POST /categories` - Create a new category
- `PUT /categories/{id}` - Update an existing category
- `DELETE /categories/{id}` - Delete a specific category

## 3. Orders:

- `GET /orders` - Retrieve a list of orders
- `GET /orders/{id}` - Retrieve a specific order
- `POST /orders` - Create a new order
- `PUT /orders/{id}` - Update an existing order
- `DELETE /orders/{id}` - Delete a specific order
- `GET /orders/{id}/items` - Retrieve items of a specific order
- `POST /orders/{id}/items` - Add items to a specific order
- `DELETE /orders/{id}/items/{itemId}` - Remove a specific item from an order

## 4. Users:

- `GET /users` - Retrieve a list of users
- `GET /users/{id}` - Retrieve a specific user
- `POST /users` - Create a new user
- `PUT /users/{id}` - Update an existing user
- `DELETE /users/{id}` - Delete a specific user

## 5. Authentication & Authorization:

- `POST /auth/login` - User login
- `POST /auth/register` - User registration
- `POST /auth/logout` - User logout
- `GET /auth/profile` - Retrieve user profile
- `PUT /auth/profile` - Update user profile
- `PUT /auth/password` - Update user password

## 6. Reviews & Ratings:

- `GET /products/{id}/reviews` - Retrieve reviews for a specific product
- `POST /products/{id}/reviews` - Add a review to a specific product
- `GET /products/{id}/ratings` - Retrieve ratings for a specific product
- `POST /products/{id}/ratings` - Add a rating to a specific product

# ***Segregation Principle***

Segregation principle ??????

---

---

1. Directly Implement Interface:

---

that class is only have that functionality gor for direct implement. example notification is a method

implement email , email related functionality write. this is not have shared logic.

2. Use Common Class for Shared Logic:

---

that is common shared logic write and aggregation used to get common logic also override common logic.

# Example

---

```
public interface ProductService {  
    List<Product> getAllProducts();  
    Product getProductById(String id);  
    Product createProduct(Product  
product);  
    Product updateProduct(String id,  
Product product);  
    void deleteProduct(String id);  
    List<Review>  
getProductReviews(String id);  
    Review addReviewToProduct(String id,  
Review review);  
    List<Rating>  
getProductRatings(String id);  
    Rating addRatingToProduct(String id,  
Rating rating);  
    List<Product> searchProducts(String  
query);  
}
```

```
public interface RatingService {  
    List<Rating>  
getProductRatings(String id);
```

```
    Rating addRatingToProduct(String id,  
Rating rating);  
}  
  
public interface ProductSearchService {  
    List<Product> searchProducts(String  
query);  
}  
  
public interface ReviewService {  
    List<Review>  
getProductReviews(String id);  
    Review addReviewToProduct(String id,  
Review review);  
}
```

# **Abstract Class**

Abstract class when use

=====

==

abstract class use  
pannanumna common method  
functionality is not change extend any  
class like static variable provide  
common thing

same in abstract method  
after some dynamic method also add .  
by extend any class compulsory  
implement and already implement  
abstract  
method just using.

# **Java Doc**

java doc

=====

class:

=====

write class name same as  
functionality name may need not  
write doc.

methods

=====

write method name same as  
action name however write complex  
logic place may write doc.

```

/*
 * Service class for managing user
operations.
 */


This service provides methods
to perform CRUD operations on user
data,
such as creating, retrieving,
updating, and deleting users.


*/


See User
See UserRepository
See UserService
*/
public class UserServiceImpl
implements UserService {
    private final UserRepository
    userRepository;
    private final PasswordEncoder
    passwordEncoder;
    /**
     * Constructor to initialize
the service with necessary
dependencies
     */
    @param userRepository the
    user repository
    @param passwordEncoder the
    password encoder
    */
    public
    UserServiceImpl(UserRepository
    userRepository, PasswordEncoder
    passwordEncoder) {
        this.userRepository =
    userRepository;
        this.passwordEncoder =
    passwordEncoder;
    }

    /**
     * Creates a new user in the
system.
     */
    @param user the user to be
    created
    @return the created user
    @override
    public User createuser(User
    user) {
        validateUser(user);
        user.setPassword(passwordEncoder.en
        code(user.getPassword()));
        return
        userRepository.save(user);
    }

    /**
     * Retrieves a user by their
ID.
     */
    @param userId the ID of the
    user to retrieve
    @return the retrieved user
    @throws
    UserNotFoundException if the user
is not found
    @override
    public User getUserById(Long
    userId) {
        return
        userRepository.findById(userId)
        .orElseThrow(() ->
new UserNotFoundException("User not
found with id: " + userId));
    }

    /**
     * Updates an existing user.
     */
    @param userId the ID of the
    user to update
    @param userdetails the new
    user details
    @return the updated user
    @throws
    UserNotFoundException if the user
is not found
    @override
    public User updateUser(Long
    userId, User userdetails) {
        User existingUser =
        userRepository.findById(userId);

        existingUser.setname(userdetails.get
        name());
        if
        (userdetails.getPassword() != null
        && userdetails.getPassword().isEmpty())
        {

            existingUser.setPassword(passwordEn
            coder.encode(userdetails.getPassword()));
        }
        return
        userRepository.save(existingUser);
    }

    /**
     * Deletes a user by their ID.
     */
    @param userId the ID of the
    user to delete
    @throws
    UserNotFoundException if the user
is not found
    @override
    public void deleteUser(Long
    userId) {
        User existingUser =
        userRepository.findById(userId);

        userRepository.delete(existingUser);
    }

    /**
     * Validates the user data
before processing.
     */
    @param user the user to
    validate
    @throws ValidationException
    if the user data is invalid
    */
    private void validateUser(User
    user) {
        if (user.getName() == null
        || user.getName().isEmpty()) {
            throw new
            ValidationException("User name
            cannot be empty");
        }
        if (user.getEmail() == null
        || user.getEmail().isEmpty()) {
            throw new
            ValidationException("User email
            cannot be empty");
        }
        if (user.getPassword() == null
        || user.getPassword().isEmpty()) {
            throw new
            ValidationException("User password
            cannot be empty");
        }
    }
}


```

# Common Comments

---

```

Java 0 day old

import java.util.*;
import java.text.SimpleDateFormat;
public class Example {
    // Initialize logger
    private static final Logger logger = LoggerFactory.getLogger(Example.class);
    double[] items;
    double totalPrice = 0;
    double taxRate = 5;
    double discount = 0;
    List<String> items = new ArrayList();
    // Add some items
    items.add("apple");
    items.add("banana");
    items.add("orange");
    // Calculate the total
    amount = totalPrice + calculateTotal(items);
    // Apply discount to the total
    totalPrice *= (1 - discount);
    // Check if the object is null before proceeding
    if (items == null) {
        // Check if the list is empty before proceeding
        if (items.isEmpty()) {
            // If it's empty
            System.out.println("No items found!");
        }
    }
    // Handle exception
    try {
        // Validate the input parameters before proceeding
        validateInput(items, taxRate, discount);
    } catch (Exception e) {
        // Calculate the tax
        double taxAmount = calculateTax(amount, taxRate);
        // Update the total
        price = totalPrice + taxAmount;
        // Check if the index is within bounds
        if (items.size() > 0) {
            // Log an informational message
            logger.info("Items processed successfully.");
            // Search (description e)
            displayAnError(message);
            System.err.println("An error occurred: " + e.getMessage());
            // Check if the string is null or empty
            if (e.getMessage() != null && !e.getMessage().isEmpty()) {
                // Parse the input string to a date
                String dateStr = e.getMessage();
                Date parsedDate = parseDate(dateStr);
                // Format the date to the required pattern
                String status = formatDate(parsedDate);
                updateStatus(status);
                // Fetch data from the server
                fetchDataFromServer();
                // Check if the user has the necessary permissions
                boolean hasPermission = checkPermissions("admin");
                if (!hasPermission) {
                    // Notify the operation if it fails
                    for (int i = 0; i < 3; i++) {
                        if (performOperation()) {
                            break;
                        }
                    }
                    // Initialize UI components
                    initializeUI();
                    // Send a notification to the user
                    sendNotification("processing completed.");
                    file // Save the output to a file
                    saveToFile("output.txt");
                    // Initialize resources for database connection
                    initializeResources();
                    // Close resources to prevent memory leaks
                    closeResources();
                    // Return the result to the caller
                    return totalPrice;
                }
            }
            private static void validateInput(List<String> items,
                                              double taxRate,
                                              double discount) {
                if (items == null || items.isEmpty())
                    if (items.isEmpty())
                        logger.error("List is empty!");
                // Check the discount value is within valid range before applying it
                if (discount < 0 || discount > 100) {
                    throw new IllegalArgumentException("Discount must be between 0 and 100%");
                }
                // Ensure the tax rate is not negative
                if (taxRate < 0) {
                    if (taxRate < 0)
                        logger.error("Tax rate cannot be negative!");
                }
            }
            private static double calculateTotal(List<String> items) {
                // Initialize the total
                totalPrice = priceOfTheFirstProductAdded();
                for (String item : items) {
                    if (item.equals("apple"))
                        total += 10;
                    else if (item.equals("banana"))
                        total += 15;
                    else if (item.equals("orange"))
                        total += 20;
                }
                return total;
            }
            private static double calculateTotal(List<String> items,
                                                 double totalPrice,
                                                 double taxRate) {
                // Calculate the tax amount
                totalPrice *= (1 + taxRate / 100);
            }
            private static void performOperation() {
                // Perform the specified operation
                return true;
            }
            private static void initializeUI() {
                // Initialize UI components for the screen
            }
            private static void sendNotification(String message) {
                logger.info("Sent a notification to the user");
            }
            private static void saveToFile(String filename) {
                // Save the output to a file
            }
            private static void initializeResources() {
                // Initialize resources for database connection
            }
            private static void closeResources() {
                // Close resources to prevent memory leaks
            }
            private static void parseData(String data) {
                // Parse the input string to extract data
            }
            private static String formatDate(Date date) {
                // Format the date to the required pattern
                SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd");
                formatter.format(date);
            }
            private static void updateStatus(String status) {
                // Update the status of the order
            }
        }
    }
}

```

# **HibernetQuery**

## Query

---

default methods:

---

1. save(S entity):
2. findOne(ID id):
3. findAll():
4. existsById(ID id):
- 5.

findAllById(Iterable<ID> ids):

6. count():
7. deleteById(ID id):
8. delete(T entity):

## Using Method Names

If your query can be expressed using Spring Data's method naming conventions, you can simply define a method in your repository interface. For example:

```
java Copy code
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    List<User> findByUsername(String username);
}
```

1.

findByFirstNameAndLastName(String  
firstName, String lastName):

2.

findByAgeGreater Than(int age):

3.

findBySalaryLessThan(double salary):

4.

findByCity(String city):

5.

findByState(String state):

6.

findByCountry(String country):

7.

findByFirstNameOrLastName(String  
firstName, String lastName):

8.

- findByAgeLessThanEqual(int age):  
9.
- findByActiveTrue():  
10.
- findByActiveFalse():  
11.
- findByFirstNameIgnoreCase(String firstName):  
12.
- findByLastNameIgnoreCase(String lastName):  
13.
- findByFirstNameLike(String pattern):  
14.
- findByLastNameLike(String pattern):  
15.
- findByFirstNameStartingWith(String prefix):  
16.
- findByLastNameEndingWith(String suffix):  
17.
- findByDateOfBirthBetween(Date startDate,  
Date endDate):  
18.
- findByFirstNameOrderByLastNameAsc(String  
firstName):  
19.
- findByLastNameOrderByFirstNameDesc(String  
lastName):  
20.

findByAgeGreaterThanOrEqualTo(int age,  
String city):

21.

findByStateAndCountry(String state, String  
country):

22.

findByFirstNameNotNull():

23.

findByLastNameNull():

24.

findByDateOfBirthNotNull():

25.

findByAgeIn(Collection<Integer> ages):

26.

findBySalaryNot(double salary):

27.

findByFirstNameAndActiveTrue(String  
firstName):

28.

findByLastNameAndActiveFalse(String  
lastName):

29.

findByCityOrState(String city, String state):

30.

findByCountryNot(String country):

31.

findByFirstNameContainingIgnoreCase(String

- keyword):
  - 32.
- findByLastNameContainingIgnoreCase(String keyword):
  - 33.
- findByAgeGreaterThanOrEqualToAndAgeLessThan(int minAge, int maxAge):
  - 34.
- findByDateOfBirthBefore(Date date):
  - 35.
- findByDateOfBirthAfter(Date date):
  - 36.
- findByDateOfBirthNotNullAndActiveTrue():
  - 37.
- findByFirstNameAndLastNameAndAge(String firstName, String lastName, int age):
  - 38.
- findByLastNameNotIn(Collection<String> lastNames):
  - 39.
- findByAgeGreaterThanOrEqualToAndCityOrState(int age, String city, String state):
  - 40.
- findBySalaryGreaterThanOrEqualToAndSalaryLessThan(double minSalary, double maxSalary):
  - 41.
- findByActiveTrueAndCityNotNull():

42.

findByActiveFalseAndStateNull():

43.

findByDateOfBirthBetweenAndCountry(Date  
startDate, Date endDate, String country):

44.

findByFirstNameIgnoreCaseOrLastNameIgnor-  
eCase(String firstName, String lastName):

45.

findByCityStartingWithIgnoreCase(String  
prefix):

46.

findByStateEndingWithIgnoreCase(String  
suffix):

47.

findByCountryIn(Collection<String>  
countries):

48.

findByFirstNameNotNullAndLastNameNotNull(  
):

49.

findByAgeLessThanOrEqualToCityOrState(int  
age, String city, String state):

50.

findBySalaryGreaterThanOrEqualToSalaryLessT-  
hanEqual(double minSalary, double  
maxSalary):

# common criteria keywords

---

| No. | Keyword      | No. | Keyword             | No. | Keyword         | No. |
|-----|--------------|-----|---------------------|-----|-----------------|-----|
| 1   | And          | 26  | TrueOrderBy         | 51  | AfterAnd        | 76  |
| 2   | Or           | 27  | FalseOrderBy        | 52  | BetweenAnd      | 77  |
| 3   | Containing   | 28  | InOrderBy           | 53  | LessThanAnd     | 78  |
| 4   | StartingWith | 29  | NotInOrderBy        | 54  | GreaterThanAnd  | 79  |
| 5   | EndingWith   | 30  | BeforeOrderBy       | 55  | StartingWithAnd | 80  |
| 6   | GreaterThan  | 31  | AfterOrderBy        | 56  | EndingWithAnd   | 81  |
| 7   | LessThan     | 32  | LessThanOrderBy     | 57  | ContainingAnd   | 82  |
| 8   | Between      | 33  | GreaterThanOrEqual  | 58  | BeforeOr        | 83  |
| 9   | IsNotNull    | 34  | BetweenOrderBy      | 59  | AfterOr         | 84  |
| 10  | IsNull       | 35  | StartingWithOrderBy | 60  | BetweenOr       | 85  |
| 11  | Not          | 36  | EndingWithOrderBy   | 61  | LessThanOr      | 86  |

## Keyword

BeforeOrderBy

AfterOrOrderBy

BetweenOrOrderBy

LessThanOrOrderBy

GreaterThanOrOrderBy

StartingWithOrOrderBy

EndingWithOrOrderBy

IsNotNullOrderBy

IsNullOrderBy

NotOrderBy

TrueOrderBy

|    |            |    |                   |    |                        |    |
|----|------------|----|-------------------|----|------------------------|----|
| 12 | OrderBy    | 37 | ContainingOrderBy | 62 | GreaterThanOr          | 87 |
| 13 | IgnoreCase | 38 | AndNotNull        | 63 | StartingWithOr         | 88 |
| 14 | True       | 39 | OrNotNull         | 64 | EndingWithOr           | 89 |
| 15 | False      | 40 | IgnoreCaseAnd     | 65 | ContainingOr           | 90 |
| 16 | In         | 41 | IgnoreCaseOr      | 66 | BeforeAndOrderBy       | 91 |
| 17 | NotIn      | 42 | TrueAnd           | 67 | AfterAndOrderBy        | 92 |
| 18 | Before     | 43 | TrueOr            | 68 | BetweenAndOrderBy      | 93 |
| 19 | After      | 44 | FalseAnd          | 69 | LessThanAndOrderBy     | 94 |
| 20 | Like       | 45 | FalseOr           | 70 | GreaterThanAndOrderBy  | 95 |
| 21 | NotNull    | 46 | InAnd             | 71 | StartingWithAndOrderBy | 96 |
| 22 | Null       | 47 | InOr              | 72 | EndingWithAndOrderBy   | 97 |
| 23 | AndOrderBy | 48 | NotInAnd          | 73 | BeforeOrOrderBy        | 98 |

FalseOrderBy

InOrderBy

NotInOrderBy

BeforeOrderBy

AfterOrderBy

LessThanOrderBy

GreaterThanOrEqualOrderBy

StartingWithOrderBy

EndingWithOrderBy

ContainingOrderBy

AndNotNullOrderBy

OrNotNullOrderBy

|    |                   |
|----|-------------------|
| 24 | OrOrderBy         |
| 25 | IgnoreCaseOrderBy |

|    |           |    |                  |     |                      |
|----|-----------|----|------------------|-----|----------------------|
| 49 | NotInOr   | 74 | AfterOrOrderBy   | 99  | IgnoreCaseAndOrderBy |
| 50 | BeforeAnd | 75 | BetweenOrOrderBy | 100 | IgnoreCaseOrOrderBy  |

## Using @Query Annotation

For more complex queries or when method naming conventions are not sufficient, you can write JPQL queries using the `@Query` annotation. Here's an example:

```
java Copy code
@Repository
public interface UserRepository extends JpaRepository<User, Long> {

    @Query("SELECT u FROM User u WHERE u.username = :username")
    List<User> findByUsername(@Param("username") String username);
}
```

## Explanation of Each Component

---

- **@Query**: Defines a custom query.
  - **SELECT**: JPQL keyword to specify the selection of data.
    - **u**: Alias for the `User` entity.
    - **FROM**: JPQL keyword indicating the data source.
  - **User**: The entity class being queried.
  - **WHERE**: JPQL keyword to specify conditions.
    - **u.email**: Field in the `User` entity.

- **= ?1**: Parameter placeholder for the first method argument.

## Native Queries

You can also execute native SQL queries using the `@Query` annotation with the `nativeQuery` attribute set to `true`. Here's an example:

```
java Copy code  
  
@Repository  
public interface UserRepository extends JpaRepository<User, Long> {  
  
    @Query(value = "SELECT * FROM users WHERE username = :username", nativeQuery =  
        true)  
    List<User> findByUsername(@Param("username") String username);  
}
```

## Explanation of Each Component

---

- **@Query**: Annotation to define a custom query.
  - **value** =: Indicates the beginning of the query string.
    - "**SELECT \* FROM users WHERE email = ?1**": The SQL query string.
    - **SELECT \***: SQL keyword to specify the selection of all columns.
    - **FROM**: SQL keyword

indicating the data source.

- **users**: The database table name.
  - **WHERE**: SQL keyword to specify conditions.
  - **email**: Column in the `users` table.
  - `= ?1`: Parameter placeholder for the first method argument.
  - **nativeQuery = true**: Specifies that this is a native SQL query.

Return Type

=====

1. **Entity**:
  - **Example**:

User `findById(Long id)`;

Returns a single entity object. Use this when you expect the query to return exactly one result.

**Entity>:**

- **Optional<**

**Example**: 0-

```
optional<User> findById(Long id);
```

### **Purpose:**

Wraps the result in an `Optional` to handle the case when the result might be null (i.e., no entity found). This is useful for avoiding `NullPointerException` and for explicitly handling the absence of a result.

- **List<Entity>:**

```
>:
```

### **Example:** L-

```
list<User> findByLastName(String  
lastName);
```

### **Purpose:**

Returns a list of entities. Use this when you expect multiple results.

- **Page<Entity>:**

```
ty>:
```

### **Example:** P-

```
age<User> findByLastName(String  
lastName, Pageable pageable);
```

### **Purpose:**

Returns a paginated list of entities. Useful for implementing pagination in your application.

- **Slice<Entity>:**

**Example:** `Slice<User> findByLastName(String lastName, Pageable pageable);`

**Purpose:**

Similar to `Page`, but only fetches a slice of data (partial page) without needing the total count. Useful for infinite scrolling scenarios.

- **Stream<Entity>:**

**Example:** `Stream<User> findAllByAgeGreater Than(int age);`

**Purpose:**

Returns a Java 8 stream of entities, allowing for lazy processing of large datasets.

- **Collection<Entity>:**

**Example:** `Collection<User> findByCity(String city);`

**Purpose:**

Returns a collection of entities. Similar to `List` but more general.

`>:`

- `Set<Entity`

```
Set<User> findDistinctByLastName(String  
lastName);
```

**Purpose:**

Returns a set of entities. Ensures that the results are unique.

- `Entity`

**Projection (Interface):**

**Example:** `List<`

```
list<UserNameOnly> findBy();
```

**Purpose:**

Returns a projection interface, allowing for partial retrieval of entity data.

- `DTO (Data`

**Transfer Object):**

**Example:** `List<`

```
list<UserDTO> findBy();
```

**Purpose:**

Returns a DTO object, often used to shape the data returned from the query to meet the specific needs of the client.

## Why Use These Return Types?

### **Entity:**

Directly interact with the entity in the database. Suitable for straightforward CRUD operations.

### **Optional<Entity>:**

Handle potential absence of data gracefully.

### **List<Entity>:**

Retrieve multiple entities when you expect more than one result.

### **Page<Entity>:**

Efficiently manage large datasets by breaking them into pages. Ideal for pagination.

### **Slice<Entity>:**

Fetch portions of data without needing the total count. Good for scenarios like infinite scrolling.

### **Stream<Entity>:**

Process large datasets in a memory-

efficient way.

**Collection<Entity>**

: More generic than `List`, but serves a similar purpose.

**Set<Entity>**

: Ensure uniqueness of results.

**Entity**

**Projection**: Optimize performance by retrieving only the necessary fields.

**DTO**: Tailor

the query results to fit the specific requirements of the client, enhancing performance and reducing data transfer overhead.

# **ProjectSetup**

Project SetUp

=====





# **Exceptions**

## Exception

---

---

### 1. ResourceNotFoundException:

---

**Scenario:** When a requested resource (like a database entity) is not found.

**Example Use Case:** A user tries to retrieve a record from the database by ID, but the record does not exist.

### 2. InvalidInputException

---

**Scenario:** When the input provided by the user is invalid or does not meet certain criteria.

**Example Use Case:** A user submits a form with invalid data (e.g., a required field is missing).

### 3. UnauthorizedAccessException

---

**Scenario:** When a user tries to access a resource without proper authorization.

**Example Use Case:** A user tries to access an admin-

only endpoint without admin privileges.

## 4. DataIntegrityViolationException

---

**Scenario:** When there is a data integrity violation, such as a constraint violation in the database. **Example Use Case:** A user tries to insert a duplicate entry that violates a **unique constraint**.

## 5. ServiceUnavailableException

---

**Scenario:** When a service is temporarily unavailable. **Example Use Case:** A dependent external service is down, and your application cannot proceed with the operation.

## 6. OperationNotAllowedException

---

**Scenario:** When an operation is not allowed due to business logic constraints.

**Example Use Case:** A user tries to delete an account that has active subscriptions.

## 7. ConflictException

---

**Scenario:** When there is a conflict, such as a duplicate

entry or version conflict.

**Example Use Case:** A user tries to create a record with a **unique field** that already exists in the database.

## 8. BadRequestException

---

**Purpose:** Used when the request made by the client is invalid. This can be due to malformed syntax, invalid request parameters, or missing required fields. It is typically mapped to an HTTP 400 Bad Request status code.

**Scenario:** A user submits a form without providing a required field, such as a username.

## 9. ForbiddenException

---

**Purpose:** Used when the user does not have permission to perform the requested operation. It is typically mapped to an HTTP 403 Forbidden status code.

**Scenario:** A regular user tries to access an admin-only page.

## 10. NotFoundException

---

**Purpose:** Used when a requested resource cannot be found. This is often used when a specified resource ID does not exist. It is typically mapped to an HTTP 404 Not Found status code.

**Scenario:** A user tries to access a blog post that does not exist.

## 11. InternalServerErrorException

---

**Purpose:** Used when an unexpected condition was encountered and no more specific message is suitable. It is typically mapped to an HTTP 500 Internal Server Error status code.

**Scenario:** A server error occurs during data processing due to an unexpected condition.

## 12. ServiceUnavailableException

**Purpose:** *ServiceUnavailableException* is used when a service is temporarily unavailable. It is typically mapped to an HTTP 503 Service Unavailable status code.

**Scenario:** An external API your application depends on is down.

## 13. IllegalArgumentException

---

- **Purpose:** Thrown to indicate that a method has been passed an illegal or inappropriate argument.
- **Scenario:** A method receives a null value where a non-null value is required.

## 14. IllegalStateException

- **Purpose:** Signals that a method has been invoked at an illegal state.

an illegal or inappropriate time.

- **Scenario:** A method is called on an object that is not in an appropriate state for that method to be called.

## 15. NullPointerException

---

- **Purpose:** Thrown when an application attempts to use `null` in a case where an object is required.
- **Scenario:** Attempting to call a method on a `null` object reference.

```
public void process(User user) {  
    if (user == null) {  
        throw new NullPointerException("User cannot be  
null.");  
    }  
    // Process user  
}
```

## 16. IndexOutOfBoundsException

---

- **Purpose:** Thrown to indicate that an index of some sort (e.g., an array, a list, etc.) is out of range.
- **Scenario:** Trying to access an array element with an invalid index.

## 17. RateLimitExceededException

---

- **Purpose:** Used when a user exceeds the allowed number of requests in a given timeframe.
- **Scenario:** A user makes too many API requests in a short period.

```
@Service
public class ApiService {

    private static final int MAX_REQUESTS = 100;
    private Map<String, Integer> userRequestCounts =
        new HashMap<>();

    public void handleRequest(String userId) {
        int requests =
            userRequestCounts.getOrDefault(userId, 0);
        if (requests >= MAX_REQUESTS) {
            throw new RateLimitExceeded("Rate
limit exceeded.");
        }
        userRequestCounts.put(userId, requests + 1);
        // Handle request
    }
}
```

## 18. **TimeoutException**

---

- **Purpose:** Used to indicate that an operation has timed out.
- **Scenario:** A long-running process exceeds the allowed time limit.

```
@Service
public class LongRunningService {

    public void performLongRunningTask() {
```

```
boolean isTimeout = checkForTimeout();
if (isTimeout) {
    throw new TimeoutException("Operation timed
out.");
}
// Perform long-running task
}

private boolean checkForTimeout() {
    // Simulated timeout check
    return true;
}
}
```

## 19. AuthenticationException

---

- **Purpose:** Used to indicate authentication failure.
- **Scenario:** A user provides invalid credentials.

```
@Service
public class AuthService {

    public void authenticate(String username, String
password) {
        boolean isAuthenticated =
checkCredentials(username, password);
        if (!isAuthenticated) {
            throw new AuthenticationException("Invalid
username or password.");
        }
        // Proceed with authentication
    }
}
```

```
private boolean checkCredentials(String username,  
String password) {  
    // Simulated credential check  
    return false;  
}  
}
```

## 19. AuthorizationException

---

- **Purpose:** Used to indicate authorization failure.
- **Scenario:** A user tries to access a resource they are not authorized to access.

```
@Service  
public class AuthService {  
  
    public void authorize(String userId, String resource)  
{  
        boolean isAuthorized = checkAuthorization(userId,  
resource);  
        if (!isAuthorized) {  
            throw new AuthorizationException("You are not  
authorized to access this resource.");  
        }  
        // Proceed with authorization  
    }  
  
    private boolean checkAuthorization(String userId,  
String resource) {  
        // Simulated authorization check  
        return false;
```

}

}

# **HttpStatusCode**

## HttpStatus Code

---

1. **1xx Informational:** The request was received, continuing process.
2. **2xx Success:** The request was successfully received, understood, and accepted.
3. **3xx Redirection:** Further action needs to be taken in order to complete the request.
4. **4xx Client Error:** The request contains bad syntax or cannot be fulfilled.
5. **5xx Server Error:** The server failed to fulfill an apparently valid request.

## 1xx Informational

---

1. **1xx Informational:** The request was received, continuing process.
2. **2xx Success:** The request was successfully received, understood, and accepted.
3. **3xx Redirection:** Further action needs to be taken in order to complete the request.
4. **4xx Client Error:** The request contains bad syntax or cannot be fulfilled.
5. **5xx Server Error:** The server failed to fulfill an apparently valid request.

## 2xx Success

---

## 200 OK

- **Purpose:** The request has succeeded.
- **Scenario:** A successful GET or POST request.

## 201 Created

- **Purpose:** The request has been fulfilled and resulted in a new resource being created.
- **Scenario:** After successfully creating a new resource, such as a new user or a blog post.

## 202 Accepted

- **Purpose:** The request has been accepted for processing, but the processing has not been completed.
- **Scenario:** When an asynchronous process is initiated.

## 204 No Content

- **Purpose:** The server successfully processed the request, but is not returning any content.
- **Scenario:** A successful DELETE request that doesn't return any content.

# 3xx Redirection

---

## 301 Moved Permanently

- **Purpose:** The requested resource has been permanently moved to a new URL.
- **Scenario:** When a resource has a new permanent URI and all future requests should use this new URI.

## 302 Found

- **Purpose:** The requested resource resides temporarily under a different URI.
- **Scenario:** A temporary redirect, such as redirecting from an HTTP to an HTTPS URL.

## 304 Not Modified

- **Purpose:** Indicates that the resource has not been modified since the version specified by the request headers.
- **Scenario:** When using caching to validate if the client has the latest version of a resource.

# 4xx Client Error

---

## 400 Bad Request

- **Purpose:** The server cannot or will not process the request due to a client error (e.g., malformed request syntax).
- **Scenario:** When the client sends a request with invalid syntax or invalid request parameters.

## 401 Unauthorized

- **Purpose:** The request requires user authentication.
- **Scenario:** When a client tries to access a resource without providing valid authentication credentials.

## 403 Forbidden

- **Purpose:** The server understood the request but refuses to authorize it.
- **Scenario:** When a user tries to access a resource they do not have permission to access.

## 404 Not Found

- **Purpose:** The server cannot find the requested resource.
- **Scenario:** When a user tries to access a URL that does not exist.

## 409 Conflict

- **Purpose:** The request could not be completed due to a conflict with the current state of the resource.
- **Scenario:** When there is a version conflict in an update operation, or a duplicate resource creation attempt.

## 422 Unprocessable Entity

- **Purpose:** The server understands the content type of the request entity, but was unable to process the contained instructions.
- **Scenario:** When validation on input data fails (common in RESTful APIs).

# 5xx Server Error

---

## **500 Internal Server Error**

- **Purpose:** The server encountered an unexpected condition that prevented it from fulfilling the request.
- **Scenario:** When an unexpected error occurs on the server side.

## **502 Bad Gateway**

- **Purpose:** The server was acting as a gateway or proxy and received an invalid response from the upstream server.
- **Scenario:** When a server is acting as a proxy and gets an invalid response from another server.

## **503 Service Unavailable**

- **Purpose:** The server is not ready to handle the request, usually due to maintenance or overload.
- **Scenario:** When the server is temporarily down for maintenance or is overloaded.

## **504 Gateway Timeout**

- **Purpose:** The server was acting as a gateway or proxy and did not receive a timely response from the upstream server.
- **Scenario:** When a server is acting as a proxy and a timeout occurs waiting for another server's response.

**example**

=====

## 1. 200 OK

```
java Copy code  
  
    @GetMapping("/users/{id}")  
    public ResponseEntity<User> getUserById(@PathVariable Long id) {  
        User user = userService.getUserById(id);  
        return new ResponseEntity<>(user, HttpStatus.OK);  
    }
```

## 2. 201 Created

```
java Copy code  
  
    @PostMapping("/users")  
    public ResponseEntity<User> createUser(@RequestBody User user) {  
        User createdUser = userService.createUser(user);  
        return new ResponseEntity<>(createdUser, HttpStatus.CREATED);  
    }
```

