Ans-16 The MVC (Model-View-Controller) pattern divides an application into three interconnected components:

1) Model: Represents the data and business logic
   Handles data operations like fetching, updating and validating (e.g., Student class, database interac.

2) View: Presents data to the user (UI)
   Usally JSP pages that display information and user forms

3) Controller: Act as intermediary between view and model
   Handles user request, invokes model operations and selects the view to display

Advantages of MVC

Maintainability: Changes in UI (view) don't affect business logic (Model) or control flow (Controller)
Developers can work independently on M, V, C

Scalability: Easier to add new features by extending one component without impacting othes
Supports multiple views for the same Model (web, mobile)

Model: Student class manages student data and

database operations

Controller: RegistrationServlet processes registration requests, validates input and calls model methods

View: registration.jsp shows the registration form and confirmation messages.

Flow: User submits forms → Controller handles request → Model saves student data → Controller forward view → view shows success page

Ans 17: Servlet Controller Managing Flow Between Model and View in Java EE:

① Receives client requests

② Interacts with Model to process or fetch data

③ Forwards the data to the view to generate the response

The servlet uses RequestDispatcher to forward requests along with data

Servlet (controller):

```
protected void (HttpServletRequest request, HttpServlet-
    Response response) throws ServletException, IO-
        Exception {
    User user = new User ("John");
    //Interact with model
```

```
request.setAttribute("user", user)
// Add data to request scope
RequestDispatcher dispatcher = reques.getRe-
        questDispatcher("userProfile.jsp");
dispatcher.forward(request, response);
// Forward to JSP(view)
}
```

JSP - userProfile.jsp:

```
<html>
<body>
  <h1> Welcome, ${user.name}! </h1>
<body>
<html>
```

Ans 20: Spring mvc handles an Http request in the following steps:

Request flow in Spring MVC:

1) Browser sends an HTTP request to the server
2) DispatcherServlet intercept the request
3) HandlerMapping maps the request to a method in a @Controller using @RequestMapping

4) The controller method executes:
It processes the request
Interacts with the model
Add data to a Model object
5) It returns a view name
6) The ViewResolver maps the view name to a JSP
7) The view is rendered with data from the model and returned to the browser

@Controller :- Marks a class as a controller that handles HTTP requests

@RequestMapping :- Maps HTTP requests to specific controller methods

Model :- An object used to pass data from the controller to the view, keeping business logic separate from presentation

Login form Submission:

```
@Controller
public class LoginController {
    @RequestMapping(value = "/login", method = RequestMethod
                                                .POST)
    public String login(@RequestParam("username") String
        username, @RequestParam("password") String
        password, Model model) {
        if("admin".equals(username) && "pass123".equals
            (password)) {
```

```
            model.addAttribute("message", "Login Successful");
            return "welcome";
        } else {
            model.addAttribute("error", "Invalid
                                credentials");
            return "login";
        }
    }
}
```

Ans 21: The DispatcherServlet is a core component of the Spring MVC framework. It acts as the entry point for all incoming HTTP requests in a Spring web application. It follows the Front Controller Design Pattern - meaning all request go through a single servlet that delegates the request to appropriate handlers.

Here is the step-by-step breakdown of how DispatcherServlet handles a request:

1) Client sends HTTP Request:

2) DispatcherServlet Receives Request from the browser

3) It consults HandlerMapping to find the appropriate @Controller and @RequestMapping method that matches the URL

4) It uses a HandlerAdapter to invoke the matched controller method

5) The controller returns a logical view name and adds model data

6) DispatcherServlet passes the view name to the View Resolver

7) The View Resolver resolves the view name to an actual view (JSP file

8) The DispatcherServlet forwards the request, model data to the resolved view for rendering

9) The generated response is sent back to the client

Handler Mapping: Maps request URLs to controller method DispatcherServlet uses it to route requests

View Resolver: Converts logical view name to actual view resources. Delegates view rendering to it

Ans 22: Performance: Prepared Statement precompiles the SQL query once and reuses it for multiple execution, reducing parsing time on the database server. This improves performance especially for repeated queries with different parameters

Security: PreparedStatement uses placeholders (?) and sets parameter values safely, preventing SQL injection attacks by treating input as data, not executable code. On the other side Statement concatenates string to build SQL, making it vulnerable to injection.

Example (Insert Record using Prepared Statement)

```
String sql = "Insert INTO users (username, email)
        VALUES (?, ?)";
try (Connection conn = DriverManager.getConnection
    (dbURL, user, pass);
    PreparedStatement ps = conn.prepareStatement
                                        (sql){
        ps.setString (1, "alice");
        ps.setString (2, "alice@example.com");
        int rowInserted = ps.executeUpdate();
```

```
        System.out.println(rowInserted + "record (s) inserted");
    }catch (SQLException e){
        e.printStackTrace();
    }
```

Ans 23: A ResultSet is an object returned by executing a SQL query using JDBC. It holds the data retrieved from the database in a tabular form and allows navigation through rows

next() → Moves the cursor to the next row of the result set. Returns false when no more rows

getString() → Retrieves a column's value as a String

getInt() → Retrieves a column's value as an Int

```
String query = "SELECT id, name, email FROM users";
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(query);
while (rs.next()){
        int id = rs.getInt("id");
        String name = rs.getString("name");
        String email = rs.getString("email");
        System.out.println("ID. "+id+", Name: "+name+", Email: "
                                +email);
}
```

Ans 25: Difference between EntityManagers persist(), merge(), remove()

persist(Object Entity): Adds a new entity instance to persistence context. Makes the entity managed and schedules for insert in the database when transaction commits. Use when creating and saving a new object for the first time

merge(object entity): Updates an existing entity by copying the state of the given detached entity into the manged entity. Returns managed entity. Use when you have a detached object and want to update the database with its changes

remove(object entity): Marks a managed entity for deletion from the database upon transaction commit. Use when you ~~have~~ want to delete an existing entity.

Ans 26: Student Entity Example:

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

Repository Interface:

```
public interface StudentRepository extends JpaRepository
                                    <Student, Long> {
}
```

CRUD operation using Repository:

Create:

```
Student student = new Student();
Student.setName("Alice");
student.setEmail("alice@example.com");
StudentRepository.save(student);
```

Read:

Get all students: `List<Student> students = StudentRepository.findAll();`

Get by ID: `Optional<Student> student = studentRepository.findById(id);`

Update: Retrieve entity, update fields and save

```
Student existing = studentRepository.findById(id)
                                    .orElseThrow();
existing.setEmail("newemail@example.com");
studentRepository.save(existing);
```

Delete:

```
studentRepository.deleteByID(id);
```

Ans 27: Spring Boot greatly simplifies the development of RESTful web series by:

i) Auto-Configuration: Automatically configures application components like Jackson (for JSON), Tomcat (server) etc

ii) Embedded Server: No need for external Tomcat/Jetty setup

iii) Starter Dependencies: Quickly adds needed dependencies with spring-boot-starter-web

iv) No XML config: Everything can be annotated using Java

v) Integrated JSON Support: Handles JSON serialization out of the box

@RestController : Combines @Controller and @ResponseBody marking the class as REST API handler that returns JSON/XML responses directly

@GetMapping and @PostMapping: Map HTTP GET and POST requests to controller methods

Example REST controller Handling JSON:

```
@RestController
@RequestMapping("/api/students")
public class StudentController {
    private List<Student> students = new ArrayList<>();
    @GetMapping
    public List<Student> getAllStudents() {
        return students; }
    @PostMapping
    public Student addStudent(@RequestBody Student
                              student){
        students.add(student);
        return student;
    }
}
```