JAVA UNIT 2 NOTES

Advanced Java - Unit II: Server-Side Programming with JavaBeans and Servlets

This unit delves into key technologies for building server-side Java applications, focusing on reusable components (JavaBeans), enterprise components (EJBs - although note the deprecation of Entity Beans), and the fundamental web programming technology (Servlets). It builds upon the core Java and basic networking concepts covered in Unit 1.

A. JavaBeans

- **What it is?**
  - A JavaBean is simply a Java class that follows specific naming and design conventions, making it a reusable software component. (Notes Images 2, 7).
  - It encapsulates data and logic into a single object, often representing data structures or simple objects used in applications. (Notes Images 2, 7).
- **Why is it important?**
  - **Reusability:** JavaBeans are designed to be easily used and manipulated by development tools or other components. (Notes Image 2, 6).
  - **Encapsulation:** Data is hidden (usually via private fields) and accessed through standard public methods (getters and setters). (Notes Image 2, 7).
  - **Interoperability:** Following the conventions allows beans to be understood and used in various environments and tools (like IDEs or GUI builders, although this is less common now, the concept is fundamental). Properties and methods can be exposed to other applications. (Notes Image 6).
  - **Maintainability:** Standardized access methods make code easier to understand and maintain. (Notes Image 2).
- **Key Concepts:**
  - **JavaBean Conventions:** For a Java class to be considered a JavaBean, it should follow these rules (Notes Images 2, 7):
    1. **Public No-Argument Constructor:** Must have a public constructor that takes no arguments. This allows tools and frameworks to easily create instances of the bean. (`public MyBean() { }`)
    2. **Serializable:** Should implement the `java.io.Serializable` interface. This allows the bean's state to be saved (persisted) and restored. (`class MyBean implements java.io.Serializable { ... }`)
    3. **Properties with Getter and Setter Methods:** Data fields (properties) in the bean should be private and exposed through public getter and setter methods following specific naming

conventions. (Notes Images 2, 5, 7, 8).

- **JavaBean Property:** A named feature accessed via getter and setter methods. (Notes Image 5). Can be of any Java data type.
- **Getter Method (Accessor):** A public method used to read the value of a property. (Notes Image 5).
  - Convention: `public <ReturnType> getPropertyName()` or `public boolean isPropertyName()` for boolean properties (recommended `is` for boolean - Notes Image 8). Example: `getName()` for a `name` property, `isEmployed()` for an `employed` boolean property.
  - Takes no arguments. (Notes Image 8).
  - Return type matches the property's type. (Notes Image 8).
- **Setter Method (Mutator):** A public method used to write or change the value of a property. (Notes Image 5).
  - Convention: `public void setPropertyName(<ParameterType> parameter)` (Notes Image 5). Example: `setName(String name)`.
  - Return type is `void`. (Notes Image 8).
  - Takes one argument whose type matches the property's type. (Notes Image 8).
- **Advantages:** Reusable components, expose properties/methods, easy to reuse software components (Notes Image 6).
- **Disadvantages:** JavaBeans are mutable by default (state can be changed), creating many getters/setters can lead to boilerplate code (Notes Image 6).

- **Simple Code Example (Employee.java):**

```
package mypack; // Package declaration (as in Notes Image 3)

import java.io.Serializable; // Must be Serializable

public class Employee implements Serializable { // Implements
Serializable

    // Private properties (fields)
    private int id;
    private String name;

    // 1. Public No-Argument Constructor
    public Employee() {
        // Default constructor - can be empty or set default values
    }

    // 3. Getter and Setter methods for 'id' property
```

```
    public int getId() { // Getter for id (Accessor)
        return id;
    }

    public void setId(int id) { // Setter for id (Mutator)
        this.id = id;
    }

    // 3. Getter and Setter methods for 'name' property
    public String getName() { // Getter for name (Accessor)
        return name;
    }

    public void setName(String name) { // Setter for name (Mutator)
        this.name = name;
    }

    // Optional: You can add other business methods, but they aren't part
of the property conventions
    // public void displayInfo() { ... }
}
```

- **Explanation of Code:**
  - The `Employee` class is in package `mypack` and implements `Serializable`.
  - It has private fields `id` and `name`.
  - It provides a `public Employee()` constructor with no arguments.
  - For each property (`id` and `name`), it has a public `get` method (e.g., `getId()`, `getName()`) to read the value and a public `set` method (e.g., `setId(int)`, `setName(String)`) to modify the value. These follow the standard JavaBean naming conventions.
- **Accessing the JavaBean:** (Code Example - Test.java from Notes Image 4)

```
package mypack; // Must be in the same package or imported

// import mypack.Employee; // If in a different package

public class Test {
    public static void main(String args[]) {
        // Create an object of the JavaBean class using the no-arg
constructor
        Employee e = new Employee();

        // Set the value of the 'name' property using the setter method
        e.setName("Arjun"); // Corresponds to e.setName("XYZ") in Notes
```

```
Image 4

        // Get the value of the 'name' property using the getter method
and print it
        System.out.println(e.getName()); // Corresponds to
System.out.println(e.getName())
    }
}
```

- **Explanation of Access Code:**
  - We create an instance of the `Employee` JavaBean using `new Employee()`.
  - We use the setter method `e.setName("Arjun")` to set the value of the `name` property.
  - We use the getter method `e.getName()` to retrieve the value of the `name` property and print it.
  - This demonstrates how external code interacts with JavaBean properties using the defined getter/setter methods.
- **References:** Notes Images 2-10 cover the definition, conventions, purpose, examples, advantages, disadvantages, and getter/setter rules for JavaBeans.

B. Enterprise Java Beans (EJBs)

- **What is it?**
  - Enterprise Java Beans (EJBs) are server-side software components that run in an EJB container within a Java EE (Enterprise Edition) application server (like JBoss, Glassfish, WebLogic, WebSphere - Notes Image 11).
  - They are part of the broader JEE platform designed to develop robust, scalable, and distributed enterprise applications. (Notes Image 11, 13).
- **Why is it important?**
  - EJBs handle complex aspects of enterprise application development automatically (handled by the EJB container), allowing developers to focus on business logic. These aspects include:
    - **Lifecycle Management:** The container creates, manages, and destroys bean instances. (Notes Image 11).
    - **Transaction Management:** Ensuring data consistency, even if multiple operations occur or errors happen. (Notes Image 11).
    - **Security:** Access control and authentication. (Notes Image 11).
    - **Concurrency:** Handling multiple clients accessing the bean simultaneously.
    - **Remote Access:** Making business logic available to clients running on different machines (Notes Image 12).
    - **Scalability & Clustering:** EJBs can be deployed in clusters to handle high load and provide failover. (Notes Image 12).

- EJBs encapsulate business logic and are separated from presentation (like Servlets/JSPs) and persistence layers. (Notes Image 12).

- **Key Concepts:**
  - Run in an **EJB Container** (Notes Image 11).
  - Needs an Application Server (Notes Image 11).
  - **Types of EJBs:** There are three main types of Enterprise Java Beans (Notes Image 13):
    1. **Session Beans:** Represent business logic or a sequence of related tasks performed for a client. They capture the "session" between a client and the business logic. (Notes Image 14). They are *not* persistent (their state doesn't typically survive server restarts or long periods of inactivity). (Notes Image 14, 17, 18).
       - **Stateless Session Bean:** Does *not* maintain client-specific conversational state between method calls. Any instance from a pool of instances can serve any client request. Suitable for general business logic that doesn't rely on past interactions within a session (e.g., a calculation service). Efficient due to pooling. (Notes Images 15, 16, 18). Often used for web services. (Notes Image 16).
       - **Stateful Session Bean:** Maintains client-specific conversational state across multiple method calls within a specific client-bean session. There is a one-to-one relationship between a client and a bean instance during the session. Suitable for tasks involving multiple steps where state needs to be remembered (e.g., a shopping cart). Less scalable than stateless beans. (Notes Image 17). Not suitable for standard web services because their state cannot be shared. (Notes Image 17).
       - **Singleton Session Bean:** A single instance of the bean exists for the entire application. This instance is shared by all clients. Suitable for shared resources, application-wide configuration, or tasks that need to run on application startup/shutdown. (Notes Image 18).
    2. **Entity Beans:** Represent persistent data stored in a database. Each Entity Bean typically corresponds to a row in a database table. They encapsulate data and the logic to interact with that data (CRUD operations). (Notes Image 19). They have a primary key to uniquely identify data. (Notes Image 20, 27).
       - **Persistence Management:** How the bean's state is synchronized with the database.
         - **Bean-Managed Persistence (BMP):** The bean's code contains the explicit JDBC (or other database access) calls to load and save its state. More flexible but more work for the developer and ties the bean to a specific database technology. (Notes Image 21).
         - **Container-Managed Persistence (CMP):** The EJB container automatically handles the loading and saving of the bean's state to the database based on mapping information (defined in annotations or deployment descriptors). Developer focuses only on business logic. Less flexible than BMP but easier. (Notes Image 21).

- **NOTE:** Entity Beans, especially CMP, were complex and have been largely replaced by the Java Persistence API (JPA), often implemented by frameworks like Hibernate. JPA uses plain Java Objects (POJOs) instead of requiring beans to extend specific EJB classes. (Notes Image 22). *So, while in the syllabus and notes, understand that in modern Java EE/Jakarta EE, JPA is the standard for persistence, not Entity Beans.*

3. **Message-Driven Beans (MDBs):** Act as asynchronous message consumers (listeners). They listen for messages arriving on a Java Messaging Service (JMS) queue or topic. When a message arrives, the EJB container invokes the MDB's `onMessage()` method to process the message. (Notes Images 23, 25, 26).
    - Asynchronous: The client sending the message doesn't wait for the MDB to process it.
    - Decoupled: Sender and receiver don't need to be available at the same time.
    - Handle multiple incoming messages concurrently from the JMS queue/topic pool. (Notes Image 23).
    - Key Method: `onMessage(Message msg)` (Notes Image 26).

- **Difference Between Session and Entity Beans:** (Notes Image 27 provides a good table summarizing this).
    - **Primary Key:** Entity Beans have a primary key (for identifying persistent data); Session Beans do not.
    - **State:** Entity Beans are stateful (representing a specific data record); Session Beans can be stateless or stateful (representing a client conversation or task).
    - **Span:** Entity Bean state persists beyond the client session; Session Bean state is limited to the client conversation or application lifecycle.
    - **Persistence:** Entity Beans manage persistent data (BMP/CMP/JPA); Session Beans do not store data persistently themselves.
    - **Accessibility:** Entity Beans (data) can be shared by multiple clients; Session Beans (conversation) are typically tied to a single client (Stateful) or any client (Stateless/Singleton).
- **Code Example:** EJB development requires an application server and specific deployment procedures. Simple, runnable examples are not practical without that setup. However, we can show the *structure* based on annotations used in modern EJBs (as hinted in your notes Images 25, 26, 29).

```java
// Example structure for a Stateless Session Bean
package com.example.beans; // User-defined package

import javax.ejb.Stateless; // Annotation for Stateless Session Bean
import javax.ejb.Remote; // Annotation if bean is accessed remotely

// Define the remote interface (what the client sees)
@Remote // This interface can be accessed remotely
public interface MyBusinessService {
```

```
    String processData(String input);
}

// Implement the bean class
@Stateless(mappedName="MyStatelessBean") // Annotation marks this as a
Stateless bean
public class MyBusinessServiceImpl implements MyBusinessService {

    @Override
    public String processData(String input) {
        // Business logic here
        System.out.println("Processing input: " + input);
        return "Processed: " + input.toUpperCase();
    }

    // Container-managed lifecycle methods (optional to implement)
    // @PostConstruct // Called after bean creation
    // public void init() { ... }
    // @PreDestroy // Called before bean is destroyed
    // public void cleanup() { ... }
}

// Client code (runs in a different process/machine) would use JNDI
lookup
// (like the RMI example in Unit 1 practicals or EJB client notes Images
27, 31)
// to get a reference to the Remote interface and call its methods.
```

- **Explanation of Code:**
  - This shows the typical structure for a modern EJB (Java EE 5+), relying heavily on annotations (`@Stateless`, `@Remote`).
  - The `MyBusinessService` interface defines the methods that clients can call.
  - The `MyBusinessServiceImpl` class implements this interface and is marked as a `@Stateless` EJB.
  - The actual business logic goes inside the methods (here, `processData`).
  - The container handles creating instances, managing the pool, and making the bean available via the `@Remote` interface.
  - Clients would use JNDI lookup (like in the RMI example from Unit 1) to find and use instances of this bean.
- **References:** Notes Images 11-27 cover the various types of EJBs, their purpose, and basic characteristics. Images 25, 26, 29, 31 provide code snippets showing EJB structure and annotations (`@Stateless`, `@Remote`).

C. Servlets

- **What is it?**
  - Servlets are Java programs that run on a web server to extend the server's capabilities, primarily to handle dynamic web content and interact with web clients using the HTTP protocol. (Notes Images 28, 29, 56).
  - They are the foundational technology for server-side Java web programming.
- **Why is it important?**
  - **Dynamic Content:** Generate HTML or other content dynamically based on client requests, database information, etc. (Notes Image 28).
  - **Handle Requests/Responses:** Process incoming HTTP requests (GET, POST, etc.) and generate appropriate HTTP responses. (Notes Image 28, 63, 65).
  - **Improved Performance (over CGI):** Servlets run within a single process (the web container) and use threads to handle each request, which is much more efficient than traditional CGI which starts a new process for every request. (Notes Image 29).
  - **Platform Independent, Robust, Scalable:** Leverage Java's benefits. (Notes Image 29, 35).
  - **Integration:** Can easily integrate with other Java APIs (like JDBC for database access, as hinted in Notes Image 29) and other server-side components (like JSPs and EJBs).
- **Key Concepts:**
  - Run in a **Web Container** (or Servlet Container). Examples: Apache Tomcat, Jetty, Glassfish, WildFly. (Notes Images 31, 32, 33, 56, 57).
  - Web Container Role: Manages the servlet lifecycle, maps incoming URLs to the correct servlet, creates threads for requests, handles the HTTP request/response objects. (Notes Images 32, 33, 56, 57).
  - Servlet Architecture: Web Browser -> Web Server -> Web Container -> Servlet. (Notes Images 31, 32, 33).
  - Request Flow: Client sends request (via Browser/Server) -> Web Container receives, finds the right servlet (using mapping from `web.xml` or annotations), creates/initializes servlet instance (if needed) -> Container calls `service()` method on a thread -> Servlet processes request using `HttpServletRequest` and `HttpServletResponse` objects -> Servlet writes response -> Container/Server sends response back. (Notes Images 32, 34, 56, 57).
  - Servlets are protocol independent *in principle* (`GenericServlet`), but most commonly used for HTTP (`HttpServlet`). (Notes Image 29, 30).
- **Types of Servlets (by implementation path):** Servlets are ultimately classes implementing the `javax.servlet.Servlet` interface. You can do this in a few ways (Notes Image 30, 48, 50):
  1. **Implementing `Servlet` Interface Directly:** Implement all 5 methods of the `Servlet` interface (`init`, `service`, `destroy`, `getServletConfig`, `getServletInfo`). This is the most basic way but requires implementing the `service` method to handle *all* requests and potentially dispatch them based on HTTP method yourself. (Notes Images 36, 38).

2. **Extending `javax.servlet.GenericServlet`:** This is an abstract class that implements the `Servlet`, `ServletConfig`, and `Serializable` interfaces. It provides default implementations for `init`, `destroy`, `getServletConfig`, `getServletInfo`, and a basic `service` method. You only *must* override the `service(ServletRequest req, ServletResponse res)` method. It's protocol-independent. (Notes Images 30, 39, 40, 41).
3. **Extending `javax.servlet.http.HttpServlet`:** This is an abstract class that extends `GenericServlet`. It's designed specifically for HTTP. Its `service()` method is already implemented to read the HTTP request method (GET, POST, etc.) and dispatch the request to corresponding methods like `doGet()`, `doPost()`, `doPut()`, `doDelete()`. This is the **most common** way to write servlets for the web. You override the specific `doXxx()` methods for the HTTP methods you want to support. (Notes Images 30, 42, 43, 48, 50).

- **Servlet Life Cycle:** (Notes Images 44, 45, 46, 56 are detailed on this).
  1. **Loading:** The web container loads the servlet class (usually on the first request for it, or on container startup if configured). (Notes Image 44, 45, 56).
  2. **Instantiation:** The container creates an instance of the servlet class using its no-arg constructor. This happens only *once*. (Notes Image 44, 45, 56).
  3. **Initialization (`init()`):** The container calls the `init(ServletConfig config)` method *once* after instantiation. Use for one-time setup. (Notes Image 44, 45, 56).
  4. **Request Handling (`service()` / `doXxx()`):** After initialization, the servlet is "ready". For *each* client request, the container calls the `service(ServletRequest req, ServletResponse res)` method (or the appropriate `doXxx()` method in `HttpServlet`). (Notes Image 44, 46, 56). This is where the main request processing logic resides.
  5. **Destruction (`destroy()`):** When the web application is stopped or the container shuts down, the container calls the `destroy()` method *once* before the servlet instance is garbage collected. Use for cleanup (releasing resources like database connections). (Notes Image 44, 46, 56).
- **Interface `Servlet` Methods:** (Notes Images 36, 37 cover these).
  - `public void init(ServletConfig config)`: Initializes the servlet.
  - `public void service(ServletRequest req, ServletResponse res)`: Called for each request.
  - `public void destroy()`: Called when the servlet is destroyed.
  - `public ServletConfig getServletConfig()`: Returns configuration object.
  - `public String getServletInfo()`: Returns servlet information.
- **How to Create and Deploy a Servlet (General Steps):** (Notes Images 47-62 detail this process).
  1. **Create Directory Structure:** Follow the standard web application structure (`WEB-INF`, `WEB-INF/classes`, `WEB-INF/lib`). Servlet `.class` files go into `WEB-INF/classes`. (Notes Image 49).
  2. **Create Servlet Class:** Write a Java class that extends `HttpServlet` (most common) and override `doGet` or `doPost` or both. (Notes Images 50, 51).

3. **Compile Servlet:** Compile the `.java` file using `javac`. You need the servlet API JAR (`servlet-api.jar` or equivalent) in your classpath, provided by the server vendor (Tomcat, Glassfish, JBoss provide different JARs - Notes Image 52). Place the compiled `.class` file into the `WEB-INF/classes` directory.
4. **Create Deployment Descriptor (`web.xml`):** An XML file located in `WEB-INF`. It configures the web application, including mapping URL patterns to specific servlet classes. (Notes Images 53, 54). *Note: In modern Java EE/Jakarta EE, annotations (`@WebServlet`) can often replace or supplement `web.xml` for mapping.*
    - `<web-app>`: Root element.
    - `<servlet>`: Defines a servlet, giving it a name and specifying its class.
    - `<servlet-mapping>`: Maps a URL pattern (e.g., `/myurl`) to a defined servlet name.
    - `<welcome-file-list>`: Defines default files (like `index.html`, `index.jsp`) to serve when a directory is requested. (Notes Images 61, 62).
5. **Package (Optional but Recommended):** Create a WAR (Web Archive) file. This is a `.zip` file with a `.war` extension containing all your web application files (`.html`, `.jsp`, `WEB-INF` folder, etc.). Use the `jar` command (`jar -cvf myapp.war *`) (Notes Image 60).
6. **Deploy WAR/Directory:** Place the WAR file or the web application directory structure into the web server's deployment directory (e.g., `webapps` in Tomcat). (Notes Image 61).
7. **Start Server:** Ensure the web server (like Tomcat) is running. (Notes Image 55).
8. **Access Servlet:** Access the servlet through a web browser using the server's address, port, context root (application name), and the URL pattern defined in `web.xml` (e.g., `http://localhost:8080/myapp/myurl`). (Notes Image 55).

- **Reference to Provided Materials:** Notes Images 28-62 cover the Servlet overview, architecture, types, lifecycle, methods, and the detailed steps for creation and deployment using `web.xml`.

D. Handling HTTP GET Requests

- **What is it?**
    - GET is one of the primary HTTP methods used by clients (browsers) to request data from a server. When a form is submitted using GET, or a user clicks a link, parameters are appended to the URL as a query string (`?param1=value1&param2=value2`). (Notes Image 63, later OCR pages).
    - Handling GET requests in a servlet involves receiving this request and sending back a response, often retrieving and displaying data based on the parameters.
- **Why is it important?**
    - GET is fundamental for retrieving information from the web (loading pages, searching, filtering). Servlets need to be able to process these requests, read parameters from the URL, and generate appropriate content.
- **Key Concepts:**

- Extend `javax.servlet.http.HttpServlet`.
- Override the `protected void doGet(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException` method. (Notes Images 43, 63, 64, later OCR pages).
- `HttpServletRequest req`: Represents the incoming request. Use its methods to get request details.
  - `req.getParameter(String name)`: Retrieves the value of a specific parameter from the request (whether it came from the URL query string or the POST body). Returns `null` if the parameter is not found. Returns a `String`. (Notes Image 63, later OCR pages).
  - `req.getParameterValues(String name)`: Returns an array of `String` if a parameter has multiple values (e.g., from checkboxes). (Later OCR pages).
  - `req.getParameterNames()`: Returns an `Enumeration` of all parameter names. (Later OCR pages).
- `HttpServletResponse res`: Represents the response to send back.
  - `res.setContentType(String type)`: Sets the content type of the response (e.g., `text/html`). Must be called before getting the `Writer`. (Notes Image 51, 64, later OCR pages).
  - `res.getWriter()`: Returns a `PrintWriter` object that can be used to write text data (like HTML) to the response body. (Notes Image 51, 64, later OCR pages).
  - `res.getOutputStream()`: Returns a `ServletOutputStream` for writing binary data.
- Processing: Read parameters using `req.getParameter`, perform necessary logic, generate the response content (usually HTML), write the content to `res.getWriter()`.
- Close the writer/stream (`out.close()`) when done. (Notes Image 64, later OCR pages).

- **Simple Code Example (Basic GET handling from later OCR pages):**

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet; // Extend HttpServlet
import javax.servlet.http.HttpServletRequest; // Use HttpServletRequest
import javax.servlet.http.HttpServletResponse; // Use HttpServletResponse

// Map this servlet using web.xml or @WebServlet("/helloform")
// Example web.xml mapping in Notes Image 54, or later OCR pages
// @WebServlet("/helloform") // Annotation based mapping (modern
approach)
public class HelloForm extends HttpServlet {

    // Override doGet method to handle GET requests
    @Override
```

```
    protected void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {

        // Set response content type BEFORE getting the writer
        response.setContentType("text/html");

        // Get the PrintWriter to write the response body
        PrintWriter out = response.getWriter();

        // Get parameters from the request URL query string
        String firstName = request.getParameter("first_name");
        String lastName = request.getParameter("last_name");

        // Write the HTML response
        out.println("<html>");
        out.println("<head><title>GET Request Handled</title></head>");
        out.println("<body>");
        out.println("<h2>Hello from GET Request!</h2>");
        out.println("<p>First Name: " + (firstName != null ? firstName :
"N/A") + "</p>");
        out.println("<p>Last Name: " + (lastName != null ? lastName :
"N/A") + "</p>");
        out.println("</body>");
        out.println("</html>");

        // Close the writer
        out.close();
    }

    // doPost method could be implemented here if needed for POST
requests
    // @Override
    // protected void doPost(HttpServletRequest request,
HttpServletResponse response) ...
}
```

- **Explanation of Code:**
  - The `HelloForm` class extends `HttpServlet`.
  - The `doGet` method is overridden. This method is automatically called by the web container when an HTTP GET request comes in mapped to this servlet's URL pattern.
  - `response.setContentType("text/html")` tells the browser to interpret the response as HTML.

    - `response.getWriter()` gets an object to write the HTML.
    - `request.getParameter("first_name")` and `request.getParameter("last_name")` retrieve the values of parameters named "first_name" and "last_name" from the request (these would typically be in the URL query string like `?first_name=Alice&last_name=Wonderland`).
    - HTML is generated and written to the `PrintWriter`.
    - `out.close()` finishes the response.
- **Reference to Provided Materials:** Notes Images 43, 63, 64 discuss handling GET requests and overriding `doGet`. The later OCR pages starting around page 58/59 provide detailed notes and complete code examples for handling GET requests, including parameter retrieval.

E. Handling HTTP POST Requests

- **What is it?**
    - POST is another primary HTTP method, typically used to send data to the server to create or update a resource, or submit form data. Parameters are sent in the *body* of the HTTP request, not in the URL. (Notes Images 65, later OCR pages).
    - Handling POST requests involves receiving this request, reading parameters from the body, performing server-side actions based on the data, and sending back a response.
- **Why is it important?**
    - POST is used for submitting forms with potentially large amounts of data or sensitive information (like passwords), as it doesn't expose the data in the URL. Servlets need to process this data, often interacting with databases or other backend systems, and provide feedback to the user.
- **Key Concepts:**
    - Extend `javax.servlet.http.HttpServlet`.
    - Override the `protected void doPost(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException` method. (Notes Images 43, 65, 66, later OCR pages).
    - `HttpServletRequest req`: Use `req.getParameter(String name)` just like with GET requests. The web container parses the POST body and makes the parameters available via this method. (Notes Image 65, 66, later OCR pages).
    - `HttpServletResponse res`: Same usage as in `doGet` for setting content type and writing the response. (Notes Image 65, 66, later OCR pages).
    - Processing: Read parameters using `req.getParameter`, perform business logic (e.g., saving data to a database), generate the response content, write the content to `res.getWriter()`.
    - Close the writer/stream (`out.close()`) when done.
- **Simple Code Example (Basic POST handling from later OCR pages):**

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
```

```java
import javax.servlet.http.HttpServlet; // Extend HttpServlet
import javax.servlet.http.HttpServletRequest; // Use HttpServletRequest
import javax.servlet.http.HttpServletResponse; // Use HttpServletResponse

// Map this servlet using web.xml or @WebServlet("/posthandler")
// @WebServlet("/posthandler")
public class PostHandlerServlet extends HttpServlet {

    // Override doPost method to handle POST requests
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {

        // Set response content type
        response.setContentType("text/html");

        // Get the PrintWriter
        PrintWriter out = response.getWriter();

        // Get parameters from the request body (typically from a form
submission)
        String username = request.getParameter("username");
        String email = request.getParameter("email");

        // Write the HTML response
        out.println("<html>");
        out.println("<head><title>POST Request Handled</title></head>");
        out.println("<body>");
        out.println("<h2>Thank you for submitting!</h2>");
        out.println("<p>Received Username: " + (username != null ?
username : "N/A") + "</p>");
        out.println("<p>Received Email: " + (email != null ? email :
"N/A") + "</p>");
        // In a real app, you'd save this data to a database here
        out.println("</body>");
        out.println("</html>");

        // Close the writer
        out.close();
    }

    // doGet method could be implemented here if you want to handle GET
```

```
    requests to this URL as well
        // @Override
        // protected void doGet(HttpServletRequest request,
    HttpServletResponse response) ...
    }
```

- **Explanation of Code:**
  - The `PostHandlerServlet` class extends `HttpServlet`.
  - The `doPost` method is overridden. This method is called by the container for HTTP POST requests mapped to this servlet.
  - `request.getParameter("username")` and `request.getParameter("email")` retrieve values from the POST request body (e.g., submitted from an HTML `<form method="post">`).
  - An HTML response is generated and sent back, confirming the received data.
- **Reference to Provided Materials:** Notes Images 43, 65, 66 discuss handling POST requests and overriding `doPost`. The later OCR pages provide detailed notes and complete code examples for handling POST requests, including parameter retrieval from forms.

F. Session Tracking, Cookies

- **What is it?**
  - HTTP is a **stateless** protocol, meaning the server treats each request as completely independent and has no built-in memory of previous requests from the same client. (Notes Image 67).
  - **Session Tracking** is the technique used in web applications to maintain state or recognize a specific user across multiple requests within a single visit or "session". (Notes Image 67).
  - A **Session** represents a conversation or a series of interactions between a specific client (like a user using a browser) and the web server over a period of time. (Notes Image 67).
  - **Cookies** are one of the techniques used for session tracking. (Notes Images 68, 69, later OCR pages).
- **Why is it important?**
  - Session tracking is essential for almost all interactive web applications to provide a personalized and continuous user experience. Without it, features like user logins, shopping carts, remembering preferences, or tracking user activity across pages would be impossible.
- **Key Concepts:**
  - **HTTP is Stateless:** The core problem session tracking solves.
  - **Session Tracking Techniques:** (Notes Image 68 lists four).
    1. **Cookies:** Small text files or pieces of data sent by the server to the browser, stored by the browser, and sent back to the server with subsequent requests for the same domain. (Notes Images 69, 70, later OCR pages).

2. **Hidden Form Fields:** Adding hidden input fields in HTML forms to pass state data between requests when forms are submitted. Limited to form submissions.
3. **URL Rewriting:** Appending data (like a session ID) to the URL path or query string for every link and form action. Works when cookies are disabled but makes URLs look messy.
4. **`HttpSession`:** Java Servlet API's built-in, most robust mechanism. The web container manages session data on the server and uses a session ID (typically sent via a cookie) to link requests from the same client to their server-side session data. (Notes Image 68 mentions this as a technique but doesn't detail its use beyond that).

- **Cookies (`javax.servlet.http.Cookie`):** (Notes Images 69-73, later OCR pages detail the `Cookie` class and methods).
  - `Cookie` Object: Represents a single cookie with a name and a value. (Notes Images 69, 71).
  - Constructor: `new Cookie(String name, String value)` (Notes Image 71).
  - Useful Methods:
    - `getName()`: Get the cookie's name. (Notes Images 71, 73).
    - `getValue()`: Get the cookie's value. (Notes Images 71, 73).
    - `setValue(String value)`: Set the cookie's value. (Notes Image 71).
    - `setMaxAge(int seconds)`: Set how long the cookie should live. `> 0` for persistent (stored on disk), `0` to delete, `-1` for non-persistent (default, lives until browser closes). (Notes Images 71, 73).
    - `setPath(String path)`: Set the path on the server for which the cookie is valid.
    - `setDomain(String domain)`: Set the domain for which the cookie is valid.
  - Adding Cookies to Response: Use `response.addCookie(Cookie cookie)`. This sends the cookie to the browser in the `Set-Cookie` HTTP header. (Notes Images 72, 75).
  - Getting Cookies from Request: Use `request.getCookies()`. This returns an array of `Cookie` objects sent by the browser in the `Cookie` HTTP header. (Notes Images 72, 73, 76). Returns `null` if no cookies are sent.
  - Deleting Cookies: Create a new cookie with the same name, set its maximum age to 0, and add it to the response. (Notes Image 73).

- **How Cookies Work for Session Tracking (Simplified):** (Notes Image 69 shows a diagram).
  1. Client sends a request.
  2. Server creates a cookie (e.g., containing a unique session ID or user identifier) and adds it to the response using `response.addCookie()`.
  3. Browser receives the response and stores the cookie.
  4. Client sends a subsequent request to the same server/domain.
  5. Browser automatically includes the stored cookie(s) in the request's `Cookie` header.
  6. Server receives the request, gets the cookie(s) using `request.getCookies()`, reads the session ID/identifier from the cookie, and uses it to retrieve the user's state (e.g., from server-

side storage like `HttpSession` or a database).

7. Server processes the request based on the user's state.

- **Simple Code Example (Setting and Getting a Cookie - based on Notes Images 74-77):**

  (Requires two servlets and web.xml mapping)

  index.html (Initial page - not provided, but implied)

```
<!DOCTYPE html>
<html><body>
    <h2>Enter Your Name</h2>
    <!-- Form submits name to FirstServlet using POST -->
    <form action="servlet1" method="post">
        Name: <input type="text" name="userName"/><br/>
        <input type="submit" value="go"/>
    </form>
</body></html>
```

  FirstServlet.java (Sets the cookie):

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.Cookie; // Import Cookie class
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

// Mapped to /servlet1 in web.xml (as in Notes Image 76)
public class FirstServlet extends HttpServlet {
    @Override
    public void doPost(HttpServletRequest request, HttpServletResponse
response) // Use doPost for form submission
            throws ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        // Get the username from the POST request parameters
        String userName = request.getParameter("userName");

        // Write a welcome message
        out.println("Welcome " + (userName != null ? userName :
"Guest"));

        // --- Create and Add a Cookie ---
```

```
        // 1. Create a Cookie object (name="uname", value=userName)
        Cookie userCookie = new Cookie("uname", userName);

        // Optional: Set max age for persistent cookie (e.g., 24 hours)
        // userCookie.setMaxAge(60 * 60 * 24);

        // Optional: Set path
        // userCookie.setPath("/"); // Make cookie available to entire
application

        // 2. Add the cookie to the response
        response.addCookie(userCookie);
        // The container will add a "Set-Cookie" header to the HTTP
response

        // Create a link/form to go to the next servlet (SecondServlet)
        out.println("<br/><br/>");
        out.println("<form action='servlet2' method='post'>"); // Use
POST for simplicity consistent with notes example
        out.println("<input type='submit' value='go to second
servlet'/>");
        out.println("</form>");

        out.close();
    }
}
```

SecondServlet.java (Gets and reads the cookie):

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.Cookie; // Import Cookie class
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

// Mapped to /servlet2 in web.xml (as in Notes Image 76)
public class SecondServlet extends HttpServlet {
    @Override
    public void doPost(HttpServletRequest request, HttpServletResponse
response) // Use doPost consistent with FirstServlet
            throws ServletException, IOException {
```

```
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        out.println("<h2>Hello from Second Servlet!</h2>");

        // --- Get Cookies from the Request ---
        // 1. Get the array of Cookies sent by the browser
        Cookie[] cookies = request.getCookies(); // Returns null if no
cookies are sent

        String username = "Guest"; // Default if cookie not found

        // 2. Check if cookies exist and find the specific cookie
        if (cookies != null) {
            for (Cookie cookie : cookies) {
                // Check if the cookie name is "uname"
                if (cookie.getName().equals("uname")) {
                    // Get the value of the "uname" cookie
                    username = cookie.getValue();
                    break; // Found the cookie, no need to loop further
                }
            }
        }

        // Use the value retrieved from the cookie
        out.println("<p>Hello " + username + ", welcome back!</p>");
        out.println("<p>This site is under construction.</p>"); // As in
Notes Image 74

        out.close();
    }
}
```

web.xml (Deployment Descriptor):

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="4.0" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd">

    <!-- Mapping for FirstServlet -->
    <servlet>
        <servlet-name>FirstServlet</servlet-name> <!-- Logical name for
```

```
the servlet -->
        <servlet-class>FirstServlet</servlet-class> <!-- Full class name
-->
    </servlet>
    <servlet-mapping>
        <servlet-name>FirstServlet</servlet-name>
        <url-pattern>/servlet1</url-pattern> <!-- URL pattern clients use
-->
    </servlet-mapping>

    <!-- Mapping for SecondServlet -->
    <servlet>
        <servlet-name>SecondServlet</servlet-name>
        <servlet-class>SecondServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>SecondServlet</servlet-name>
        <url-pattern>/servlet2</url-pattern>
    </servlet-mapping>

    <!-- Welcome file list (optional, sets default page) -->
    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
    </welcome-file-list>

</web-app>
```

- **Explanation of Cookie Code Example:**
  - The `index.html` (or similar page) has a form that submits the username to `/servlet1`.
  - `FirstServlet` receives the POST request, gets the `userName` parameter. It then creates a `Cookie` named "uname" with the user's name as its value and adds this cookie to the response using `response.addCookie()`. The browser receives this cookie. It also generates HTML with a link/form to `/servlet2`.
  - When the user navigates (or submits the form) to `/servlet2`, the browser automatically sends the "uname" cookie (along with any others for that domain/path) in the request headers.
  - `SecondServlet` receives the request and uses `request.getCookies()` to get the array of cookies. It iterates through them to find the cookie named "uname" and retrieves its value (`cookie.getValue()`).
  - It then uses this value to print a personalized "Hello" message, demonstrating that the server remembered the user's name across two different servlet requests using the cookie.
  - The `web.xml` file maps the URLs `/servlet1` and `/servlet2` to `FirstServlet` and `SecondServlet` classes respectively, allowing the web container to direct requests correctly.

- **Reference to Provided Materials:** Notes Images 67-77 provide detailed explanations and code examples for Cookie-based session tracking in Servlets, covering creating, adding, getting, and deleting cookies. The example structure (two servlets, `web.xml`) matches the notes' illustration.

This concludes the detailed notes for Advanced Java Unit 2, covering JavaBeans, EJBs (with the note on JPA), Servlets (Overview, Architecture, Interface, Lifecycle, Types), Handling GET/POST requests, and Session Tracking using Cookies, including simple code examples for key concepts.