# Web Technology / Internet Programming – Unit Notes

*Session Management • Cookies • Authentication and Authorization • Database Connectivity (JDBC / ORM)*

## 1. Session Management

### 1.1 Need for Session Management

HTTP is a stateless protocol: every request is treated as a new, independent request and the server keeps no memory of earlier requests from the same client. But real applications – login, shopping cart, online exam, banking – need to remember the user across many requests. The technique of maintaining the state of a user across multiple requests is called session management (or session tracking).

A session is a series of related requests coming from the same client within a period of time. The server identifies the client through a unique identifier called the Session ID.

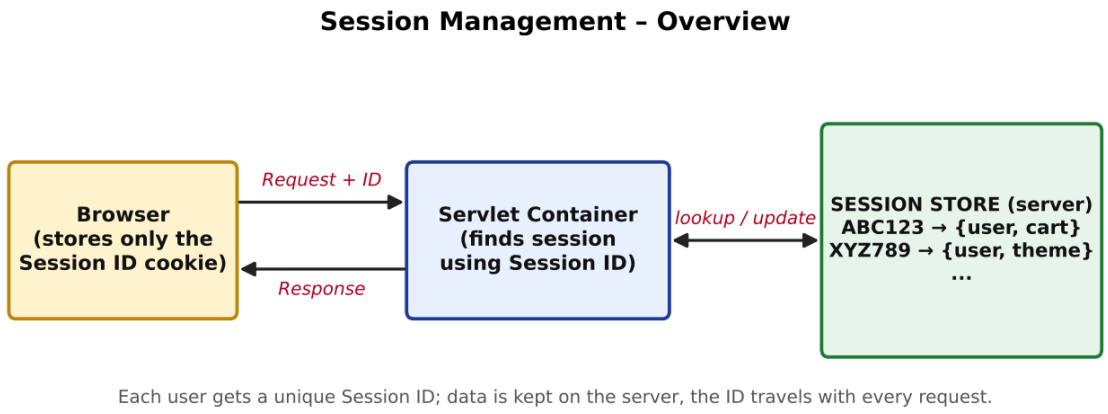



***Fig. 1 Session management overview***

### 1.2 Session Tracking Techniques

| Technique | How it works | Advantages | Limitations |
|---|---|---|---|
| Cookies | Server sends small data in a Set-Cookie header; browser returns it with every request | Simple, automatic, can persist after browser closes | May be disabled by user; size limited; stored on client |
| URL rewriting | Session ID or data appended to every URL (;jsessionid=…) using response.encodeURL() | Works even when cookies are disabled | Long URLs, ID visible in logs/history, must rewrite every link |

| Technique | How it works | Advantages | Limitations |
|---|---|---|---|
| Hidden form fields | Data stored in <input type="hidden"> fields of the form | Works in all browsers | Only for form submissions; visible in page source |
| HttpSession | Server stores data in a session object identified by JSESSIONID | Secure, any Java object can be stored, easy API | Consumes server memory; needs cookie or URL rewriting for the ID |

In practice the container uses HttpSession with a JSESSIONID cookie, and automatically falls back to URL rewriting when cookies are disabled.

## 1.3 URL Rewriting and Hidden Fields – Examples

```
// URL rewriting
String url = response.encodeURL("cart.jsp?item=101");
out.println("<a href='" + url + "'>View Cart</a>");

<!-- Hidden field -->
<form action="next" method="post">
   <input type="hidden" name="userId" value="U101">
   <input type="submit" value="Continue">
</form>
```

## 1.4 HttpSession Interface

HttpSession is the standard servlet mechanism for session management. The container creates a session object, assigns a unique ID, sends it to the browser (normally as the JSESSIONID cookie) and uses the ID on later requests to find the same object.

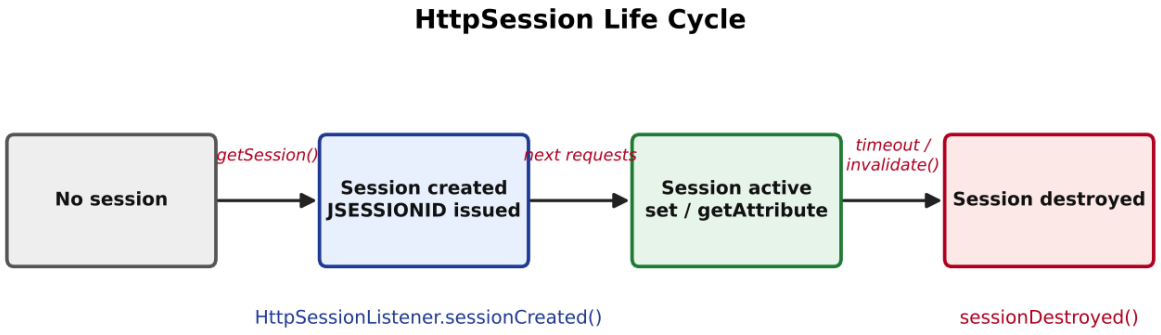



*Fig. 2 HttpSession life cycle*

| Method | Purpose |
|---|---|
| request.getSession() | Returns the current session; creates a new one if none exists |
| request.getSession(false) | Returns the current session or null (does not create) |
| setAttribute(String, Object) | Stores an object in the session |

| Method | Purpose |
|---|---|
| getAttribute(String) | Retrieves the stored object (returns null if absent) |
| removeAttribute(String) | Removes one attribute |
| getId() | Returns the unique session ID |
| isNew() | True if the client has not yet joined the session |
| getCreationTime() / getLastAccessedTime() | Time of creation and last request (milliseconds) |
| setMaxInactiveInterval(int sec) | Sets idle timeout; negative value means never expires |
| invalidate() | Destroys the session and removes all its data (used for logout) |
| request.changeSessionId() | Issues a new ID for the same session (prevents fixation) |

### *Example – counting visits using a session*

```
protected void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
  res.setContentType("text/html");
  PrintWriter out = res.getWriter();
  HttpSession session = req.getSession();
  Integer count = (Integer) session.getAttribute("visits");
  count = (count == null) ? 1 : count + 1;
  session.setAttribute("visits", count);
  out.println("Session ID : " + session.getId() + "<br>");
  out.println("Visit count: " + count);
}
```

### *Session timeout configuration (web.xml)*

```
<session-config>
  <session-timeout>20</session-timeout>   <!-- minutes -->
  <cookie-config>
    <http-only>true</http-only>
    <secure>true</secure>
  </cookie-config>
</session-config>
```

## 1.5 Sessions in JSP

In JSP the session object is an implicit object of type HttpSession, available by default. It can be turned off with <%@ page session="false" %>. Example: <% session.setAttribute("user", "Ravi"); %> and later ${sessionScope.user} or <%= session.getAttribute("user") %>.

## 1.6 Session Event Listeners

| Listener | Events handled |
|---|---|
| HttpSessionListener | sessionCreated(), sessionDestroyed() – e.g. count active users |
| HttpSessionAttributeListener | attributeAdded(), attributeRemoved(), attributeReplaced() |

| Listener | Events handled |
|---|---|
| HttpSessionActivationListener | sessionWillPassivate(), sessionDidActivate() – session moved between JVMs |
| HttpSessionBindingListener | valueBound(), valueUnbound() – object added/removed from a session |

```
@WebListener
public class ActiveUsers implements HttpSessionListener {
  static int active = 0;
  public void sessionCreated(HttpSessionEvent e)   { active++; }
  public void sessionDestroyed(HttpSessionEvent e) { active--; }
}
```

### 1.7 Session Security

| Threat | Countermeasure |
|---|---|
| Session hijacking (ID stolen) | Use HTTPS; HttpOnly and Secure cookies; short timeout |
| Session fixation (attacker sets known ID) | Call request.changeSessionId() after login |
| Session left open on shared PC | Provide logout that calls session.invalidate() |
| ID exposed in URL | Prefer cookies; avoid URL rewriting for sensitive apps |
| Sensitive data in session | Store only a user id/role, never raw passwords or card data |

### 1.8 Cookie vs Session

| Cookie | HttpSession |
|---|---|
| Data stored on the client (browser) | Data stored on the server |
| Only String values; small size (about 4 KB) | Any Java object; limited by server memory |
| Can persist for days or years | Ends on timeout, logout or server restart |
| Less secure – user can view or edit | More secure – only the ID is sent to the client |
| No server memory used | Uses server memory for every active user |

## 2. Cookies

### 2.1 Definition

A cookie is a small piece of text data (name–value pair with some attributes) that a web server sends to the browser. The browser stores it and sends it back to the same server with every later request that matches the cookie's domain and path. Cookies are the most common way of carrying session IDs and user preferences.

## 2.2 Working of a Cookie

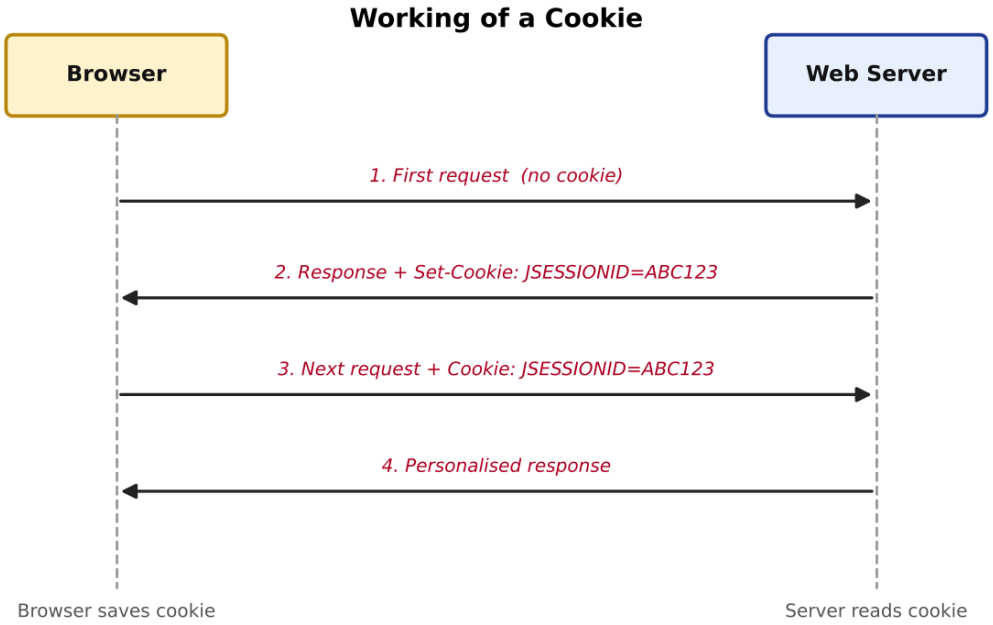



***Fig. 3 Working of a cookie***

1. The browser sends the first request; it has no cookie yet.
2. The servlet creates a Cookie and adds it to the response; the server sends it in the Set-Cookie header.
3. The browser stores the cookie (in memory or on disk).
4. On every later request to that site the browser automatically adds the Cookie header; the servlet reads it with request.getCookies().

## 2.3 Cookie Attributes

| Attribute | Meaning | Servlet method |
|---|---|---|
| Name / Value | Identifier and data of the cookie | new Cookie(name, value) |
| Max-Age / Expires | Lifetime in seconds; 0 deletes it; negative = session cookie | setMaxAge(int) |
| Domain | Domain to which the cookie is sent | setDomain(String) |
| Path | URL path prefix for which the cookie is sent | setPath(String) |
| Secure | Cookie sent only over HTTPS | setSecure(true) |
| HttpOnly | Not accessible from JavaScript (reduces XSS theft) | setHttpOnly(true) |
| SameSite | Limits cross-site sending (reduces CSRF): Strict, Lax, None | Set via container / header |

## 2.4 Types of Cookies

| Session (non-persistent) cookie | Persistent cookie |
|---|---|
| No expiry set (maxAge negative) | Expiry set using setMaxAge() |
| Stored in browser memory; deleted when the browser closes | Stored on disk until it expires or is deleted |
| Example: JSESSIONID | Example: "remember me", language, theme |

Based on origin, cookies are also first-party (set by the site being visited) or third-party (set by another domain, e.g. advertisers; increasingly blocked by browsers).

## 2.5 Cookie API in Servlets

### *Creating and sending a cookie*

```
Cookie c = new Cookie("username", "ravi");
c.setMaxAge(60 * 60 * 24 * 7);   // 7 days (persistent)
c.setPath("/");
c.setHttpOnly(true);
response.addCookie(c);
```

### *Reading cookies*

```
Cookie[] cookies = request.getCookies();   // null if none
if (cookies != null) {
  for (Cookie ck : cookies) {
    if (ck.getName().equals("username"))
       out.println("Welcome back, " + ck.getValue());
  }
}
```

### *Deleting a cookie*

```
Cookie c = new Cookie("username", "");
c.setMaxAge(0);       // expire immediately
c.setPath("/");       // path must match the original
response.addCookie(c);
```

## 2.6 Advantages and Limitations

- **Advantages:** simple, handled automatically by the browser, no server memory used, survives browser restarts.
- **Limitations:** about 4 KB per cookie and a limited number per domain; only text values; the user can disable or edit them; sent with every request (extra bandwidth); not suitable for confidential data.

# 3. Authentication and Authorization

## 3.1 Definitions

- **Authentication:** verifying the identity of a user – "Who are you?" – using credentials such as username/password, OTP, fingerprint or a digital certificate.
- **Authorization:** deciding what an authenticated user is allowed to access or do – "What can you do?" – based on roles or permissions.

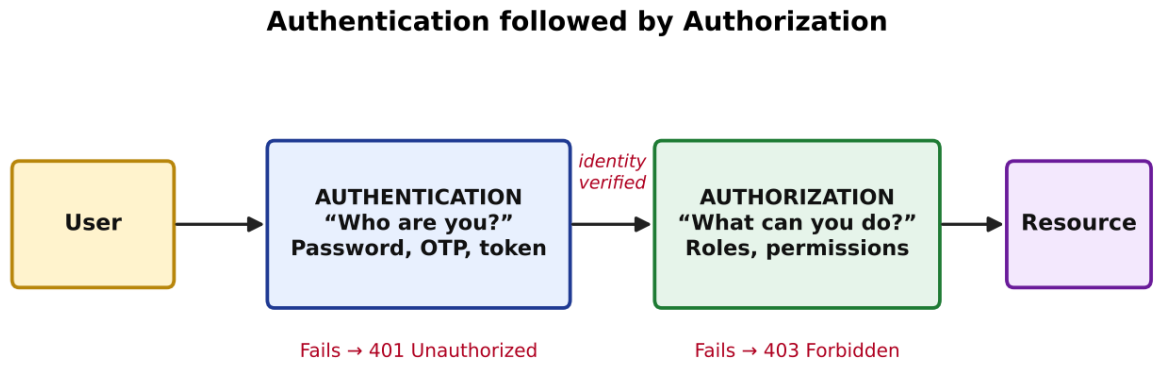



***Fig. 4 Authentication followed by authorization***

| Authentication | Authorization |
|---|---|
| Confirms the identity of the user | Decides the access rights of the user |
| Done first | Done after successful authentication |
| Uses passwords, OTP, biometrics, certificates | Uses roles, access-control lists, permissions |
| Failure gives HTTP 401 Unauthorized | Failure gives HTTP 403 Forbidden |
| Example: logging in to the college portal | Example: only faculty can enter marks |

## 3.2 Authentication Mechanisms

| Mechanism | Description | Remarks |
|---|---|---|
| HTTP BASIC | Browser pop-up; username:password sent Base64-encoded in the Authorization header | Weak; must be used with HTTPS |
| HTTP DIGEST | Password sent as a hash (digest) with a server challenge | Better than BASIC, rarely used now |
| FORM-based | Developer supplies a custom login page; credentials posted to j_security_check | Most common in web apps |
| CLIENT-CERT | Client proves identity with an X.509 certificate over HTTPS | Very strong, used in enterprises |
| Token-based (JWT) | Server issues a signed token after login; client sends it with each request | Stateless; common in REST APIs |

| Mechanism | Description | Remarks |
|---|---|---|
| OAuth 2.0 / SSO | Login delegated to a trusted provider (Google, college SSO) | Single sign-on |
| Multi-factor (MFA) | Password plus a second factor such as OTP | Greatly reduces account theft |

Authentication can be application-managed (the developer writes the login code, using a database and sessions) or container-managed (the servlet container, e.g. Tomcat, does it from declarations in web.xml).

## 3.3 Application-Managed Authentication (Custom Login)

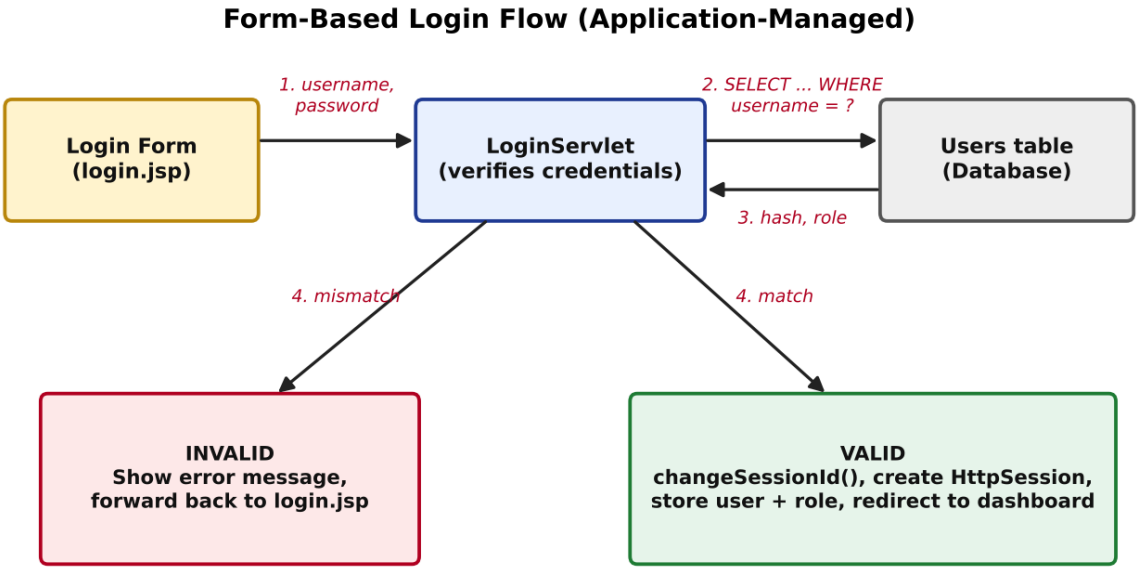



***Fig. 5 Form-based login flow***

1. User submits username and password from login.jsp using POST over HTTPS.
2. LoginServlet reads the parameters and queries the users table with a PreparedStatement.
3. The stored password hash is compared with the hash of the entered password (BCrypt or PBKDF2 – never store plain-text passwords).
4. If valid, a session is created (after changeSessionId()), the user and role are saved in it and the user is redirected to the dashboard; otherwise an error is shown.

### *login.jsp*

```
<form action="login" method="post">
  Username: <input type="text" name="username"><br>
  Password: <input type="password" name="password"><br>
  <input type="submit" value="Login">
</form>
```

***LoginServlet.java***

```
@WebServlet("/login")
public class LoginServlet extends HttpServlet {
  protected void doPost(HttpServletRequest req, HttpServletResponse res)
          throws ServletException, IOException {
    String user = req.getParameter("username");
    String pwd  = req.getParameter("password");
    try (Connection con = ds.getConnection();
         PreparedStatement ps = con.prepareStatement(
           "SELECT password_hash, role FROM users WHERE username = ?")) {
      ps.setString(1, user);
      ResultSet rs = ps.executeQuery();
      if (rs.next() && BCrypt.checkpw(pwd, rs.getString("password_hash"))) {
        req.changeSessionId();                        // stop session fixation
        HttpSession s = req.getSession();
        s.setAttribute("user", user);
        s.setAttribute("role", rs.getString("role"));
        res.sendRedirect("dashboard.jsp");
      } else {
        req.setAttribute("error", "Invalid username or password");
        req.getRequestDispatcher("login.jsp").forward(req, res);
      }
    } catch (SQLException e) { throw new ServletException(e); }
  }
}
```

***Logout***

```
HttpSession s = request.getSession(false);
if (s != null) s.invalidate();
response.sendRedirect("login.jsp");
```

## 3.4 Container-Managed Security (web.xml)

The servlet specification lets the container perform authentication and authorization through declarative configuration. The user–password–role data lives in a realm (database, LDAP, file, e.g. Tomcat's tomcat-users.xml).

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Admin Area</web-resource-name>
    <url-pattern>/admin/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>admin</role-name>
  </auth-constraint>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>  <!-- HTTPS -->
  </user-data-constraint>
</security-constraint>

<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
    <form-login-page>/login.jsp</form-login-page>
    <form-error-page>/error.jsp</form-error-page>
  </form-login-config>
</login-config>
```

```
<security-role><role-name>admin</role-name></security-role>
```

The FORM must post to the fixed action j_security_check with fields j_username and j_password. Programmatic checks inside a servlet use request.getRemoteUser(), request.getUserPrincipal() and request.isUserInRole("admin").

## 3.5 Authorization – Role-Based Access Control (RBAC)

In RBAC, permissions are not given to individual users but to roles; users are then assigned one or more roles. This makes administration simple and follows the principle of least privilege.

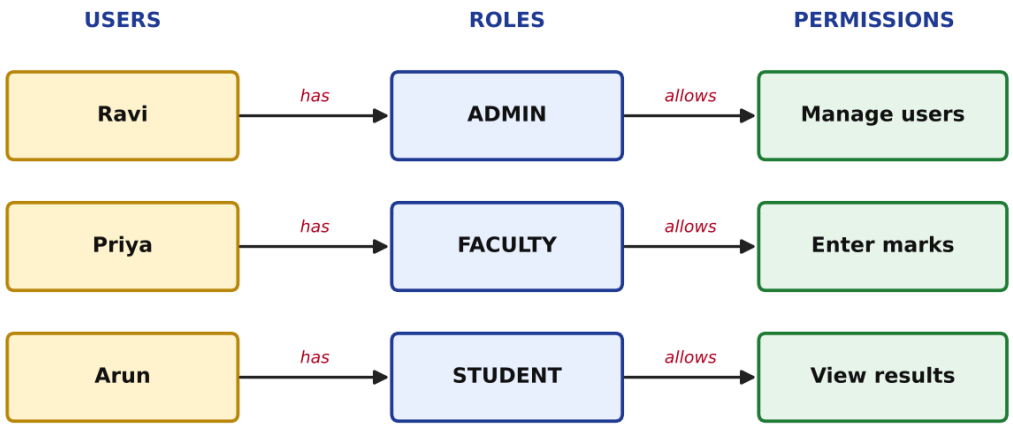



***Fig. 6 Role-based access control***

### *Authorization using a servlet Filter*

A filter intercepts every request before it reaches the servlet, so one class can protect a whole group of URLs.

```
@WebFilter("/admin/*")
public class AuthFilter implements Filter {
  public void doFilter(ServletRequest rq, ServletResponse rs, FilterChain
chain)
          throws IOException, ServletException {
    HttpServletRequest  req = (HttpServletRequest) rq;
    HttpServletResponse res = (HttpServletResponse) rs;
    HttpSession s = req.getSession(false);
    if (s == null || s.getAttribute("user") == null) {
      res.sendRedirect(req.getContextPath() + "/login.jsp");   // not
authenticated
    } else if (!"ADMIN".equals(s.getAttribute("role"))) {
      res.sendError(HttpServletResponse.SC_FORBIDDEN);          // not
authorised (403)
    } else {
      chain.doFilter(rq, rs);                                    // allowed
    }
  }
}
```

### 3.6 Common Threats and Countermeasures

| Threat | Description | Countermeasure |
|---|---|---|
| SQL injection | Malicious SQL typed into input fields | PreparedStatement, input validation |
| Plain-text passwords | Passwords readable if database leaks | Store salted hashes (BCrypt / PBKDF2) |
| Brute-force attack | Repeated password guessing | Account lock-out, delay, CAPTCHA, MFA |
| Cross-site scripting (XSS) | Script injected into a page | Escape output (JSTL c:out), HttpOnly cookie |
| CSRF | Forged request using the user's session | Anti-CSRF token, SameSite cookie |
| Eavesdropping | Credentials captured on the network | HTTPS / TLS for the entire site |

## 4. Database Connectivity using JDBC

### 4.1 Introduction

Java Database Connectivity (JDBC) is a standard Java API (packages java.sql and javax.sql) that allows a Java program – a servlet, JSP or desktop application – to connect to a relational database, send SQL statements and process the results. It hides the differences between database products: the program is written against the JDBC interfaces and a vendor-supplied JDBC driver translates the calls into the database's own protocol.

### 4.2 JDBC Architecture

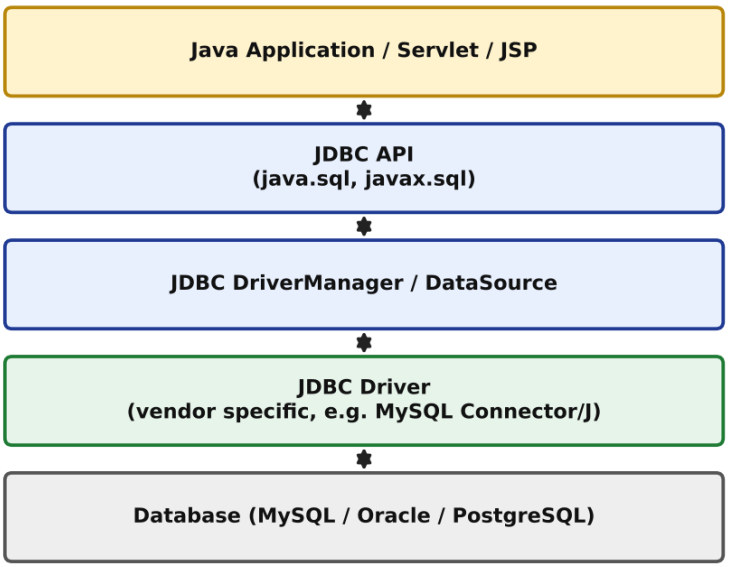



*Fig. 7 JDBC architecture*

- **JDBC API:** interfaces such as Connection, Statement and ResultSet used by the programmer.
- **DriverManager / DataSource:** manages drivers and creates connections.
- **JDBC Driver:** vendor-specific implementation of the interfaces (e.g. MySQL Connector/J, Oracle ojdbc).

## 4.3 Types of JDBC Drivers

| Type | Name | Working | Remarks |
|---|---|---|---|
| Type 1 | JDBC–ODBC bridge | Converts JDBC calls to ODBC calls | Slow, needs ODBC on client; removed from Java 8 onwards |
| Type 2 | Native-API driver | Converts JDBC calls to the database's native client library calls | Faster than Type 1; needs native library on client |
| Type 3 | Network-protocol driver | Sends calls to a middleware server that talks to the database | Flexible, no client install; extra server needed |
| Type 4 | Thin (pure Java) driver | Converts JDBC calls directly to the database protocol | Fastest and most portable; most widely used |

## 4.4 Important JDBC Interfaces

| Interface / Class | Purpose |
|---|---|
| DriverManager | Loads drivers and returns a Connection via getConnection(url, user, password) |
| Connection | Session with the database; creates statements; controls transactions (commit, rollback) |
| Statement | Executes simple static SQL |
| PreparedStatement | Precompiled SQL with ? placeholders; fast and safe from SQL injection |
| CallableStatement | Calls stored procedures and functions |
| ResultSet | Table of query results; moved row by row using next() |
| ResultSetMetaData | Information about columns (count, names, types) |
| SQLException | Exception for database errors |

## 4.5 Steps to Connect to a Database

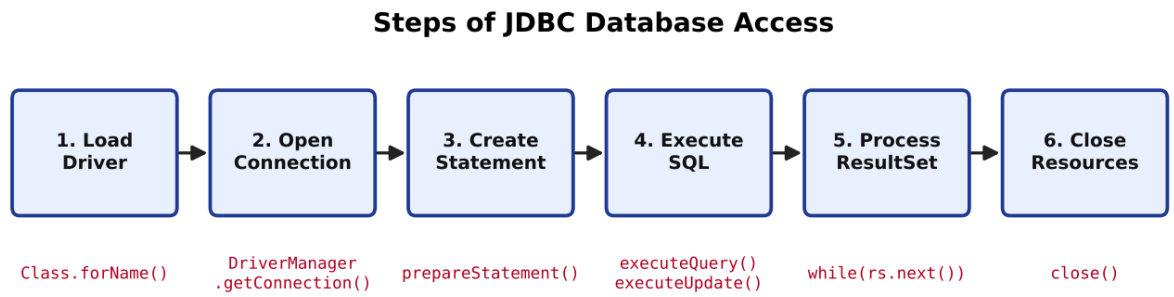



***Fig. 8 Steps of JDBC database access***

| Step | Description |
|---|---|
| 1. Load the driver | Class.forName("com.mysql.cj.jdbc.Driver") – optional in JDBC 4+ because drivers register automatically from the classpath |
| 2. Open a connection | Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/college", "root", "pwd") |
| 3. Create a statement | PreparedStatement ps = con.prepareStatement(sql) |
| 4. Execute the SQL | executeQuery() for SELECT, executeUpdate() for INSERT / UPDATE / DELETE |
| 5. Process the result | Loop with rs.next() and read values with getInt(), getString() |
| 6. Close resources | Close ResultSet, Statement and Connection (best with try-with-resources) |

The JDBC URL format is jdbc:<subprotocol>://<host>:<port>/<database>, for example jdbc:oracle:thin:@localhost:1521:xe for Oracle.

### *Complete example – SELECT (try-with-resources)*

```
import java.sql.*;

public class StudentList {
  public static void main(String[] args) {
    String url = "jdbc:mysql://localhost:3306/college";
    String sql = "SELECT regno, name, dept FROM student WHERE dept = ?";
    try (Connection con = DriverManager.getConnection(url, "root", "pwd");
         PreparedStatement ps = con.prepareStatement(sql)) {
      ps.setString(1, "CSE");
      try (ResultSet rs = ps.executeQuery()) {
        while (rs.next()) {
          System.out.println(rs.getInt("regno") + " " +
                             rs.getString("name") + " " +
rs.getString("dept"));
        }
      }
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

### *INSERT, UPDATE and DELETE*

```
// INSERT
PreparedStatement ins = con.prepareStatement(
     "INSERT INTO student(regno, name, dept) VALUES (?, ?, ?)");
ins.setInt(1, 101); ins.setString(2, "Anu"); ins.setString(3, "IT");
int rows = ins.executeUpdate();          // number of rows affected

// UPDATE
PreparedStatement upd = con.prepareStatement(
     "UPDATE student SET dept = ? WHERE regno = ?");
upd.setString(1, "CSE"); upd.setInt(2, 101); upd.executeUpdate();

// DELETE
PreparedStatement del = con.prepareStatement("DELETE FROM student WHERE regno
= ?");
del.setInt(1, 101); del.executeUpdate();
```

## 4.6 Statement vs PreparedStatement vs CallableStatement

| Statement | PreparedStatement | CallableStatement |
|---|---|---|
| Static SQL string | SQL with ? parameters | Calls stored procedure: {call proc(?, ?)} |
| Compiled every time | Precompiled once, reused | Procedure is stored in the database |
| Prone to SQL injection | Prevents SQL injection | Supports IN, OUT and INOUT parameters |
| Use for one-time simple queries | Use for repeated or user-input queries | Use for business logic kept in DB |

## 4.7 ResultSet Types

- **TYPE_FORWARD_ONLY (default):** cursor moves only forward using next().
- **TYPE_SCROLL_INSENSITIVE / SENSITIVE:** cursor can move in both directions (previous(), absolute(n)).
- **CONCUR_READ_ONLY / CONCUR_UPDATABLE:** whether the result set can be used to update the database.

## 4.8 Transaction Management

A transaction is a group of SQL statements that must succeed or fail as one unit (ACID properties: Atomicity, Consistency, Isolation, Durability). By default JDBC runs in auto-commit mode, where every statement is committed immediately. For a transaction, auto-commit is turned off.

```
try {
  con.setAutoCommit(false);
  debit.executeUpdate();     // UPDATE account SET bal = bal - 500 WHERE id =
1
```

```
  credit.executeUpdate();     // UPDATE account SET bal = bal + 500 WHERE id =
2
  con.commit();               // both succeed
} catch (SQLException e) {
  con.rollback();             // undo both on failure
}
```

## 4.9 Batch Processing

Many similar statements can be sent to the database in a single round trip using addBatch() and executeBatch(), which greatly improves speed for bulk inserts.

```
for (Student s : list) {
  ps.setInt(1, s.getRegno()); ps.setString(2, s.getName());
  ps.addBatch();
}
int[] result = ps.executeBatch();
```

## 4.10 Connection Pooling and DataSource

Creating a new database connection for every request is slow and expensive. A connection pool creates a set of connections in advance; a request borrows one, uses it, and calling close() simply returns it to the pool. Servlet containers provide a pool through the javax.sql.DataSource interface, looked up using JNDI. Common pool implementations are Tomcat JDBC Pool, Apache DBCP and HikariCP.

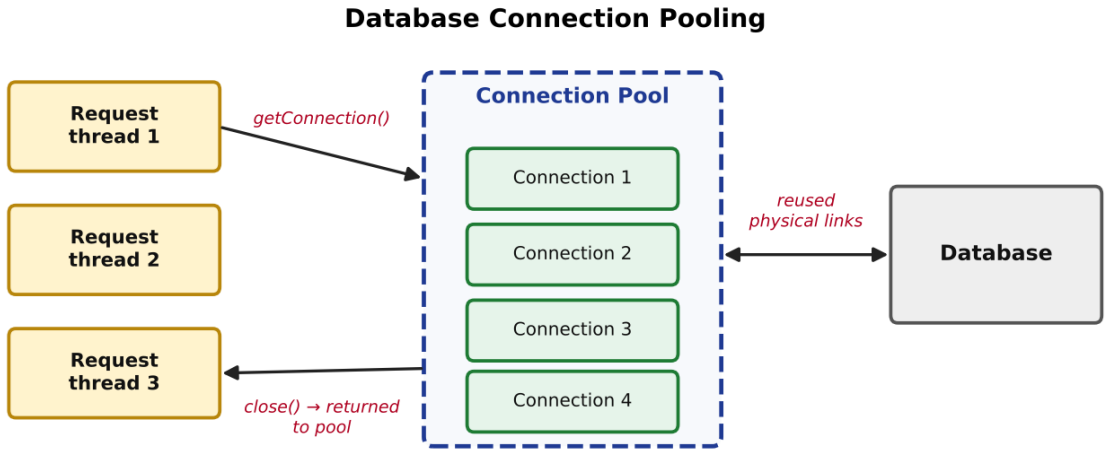



***Fig. 9 Database connection pooling***

### *Configuring a pool in Tomcat (META-INF/context.xml)*

```
<Context>
  <Resource name="jdbc/CollegeDB" auth="Container"
            type="javax.sql.DataSource"
            driverClassName="com.mysql.cj.jdbc.Driver"
            url="jdbc:mysql://localhost:3306/college"
            username="root" password="pwd"
            maxTotal="20" maxIdle="5" maxWaitMillis="10000"/>
</Context>
```

```
Context ctx = new InitialContext();
```

```
DataSource ds = (DataSource) ctx.lookup("java:comp/env/jdbc/CollegeDB");
Connection con = ds.getConnection();   // taken from the pool
```

Advantages: faster response, limited and controlled number of connections, better scalability, and the database credentials are kept out of the Java code.

### 4.11 DAO Pattern

The Data Access Object (DAO) pattern places all JDBC code for one table in a separate class (e.g. StudentDAO with insert(), findById(), update(), delete()). Servlets call the DAO and never write SQL themselves, so the database logic can be changed without touching the controller or view. In MVC, the DAO and JavaBean form the Model.

## 5. ORM Concepts

### 5.1 Object–Relational Impedance Mismatch

Java applications model data as objects (classes, inheritance, references), while relational databases store data as tables, rows and foreign keys. Using plain JDBC the programmer must write a lot of repetitive code to convert a ResultSet into objects and objects into SQL. The differences between these two models are called the object–relational impedance mismatch.

| Issue | Object model | Relational model |
|---|---|---|
| Structure | Classes with fields and methods | Tables with columns |
| Identity | Reference equality (==) and equals() | Primary key |
| Inheritance | Subclass extends superclass | No direct support |
| Associations | Object references, collections | Foreign keys, join tables |
| Navigation | Follow references (a.getB().getC()) | SQL joins |
| Data types | Java types | SQL types |

### 5.2 What is ORM?

Object–Relational Mapping (ORM) is a technique that automatically maps Java classes to database tables, fields to columns and objects to rows. The programmer works with objects and the ORM framework generates the SQL and handles the JDBC calls.

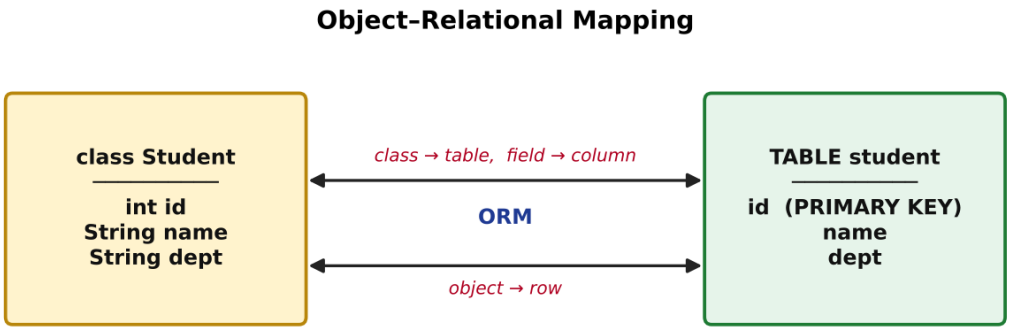



***Fig. 10 Object–relational mapping***

### *Advantages*

- Much less boilerplate code than JDBC; no manual ResultSet-to-object conversion.
- Database independence – only the dialect changes when moving from MySQL to Oracle.
- Automatic SQL generation, caching, lazy loading and transaction support.
- Better maintainability – business logic stays object-oriented.

### *Disadvantages*

- Learning curve; some performance overhead for very complex queries.
- Less direct control over the generated SQL.

## 5.3 ORM Tools and JPA

| Tool | Description |
|---|---|
| Hibernate | Most popular open-source ORM framework for Java; also a JPA implementation |
| JPA (Jakarta Persistence API) | Standard specification for ORM in Java; implemented by Hibernate, EclipseLink, OpenJPA |
| EclipseLink | Reference implementation of JPA |
| MyBatis | Semi-ORM: maps SQL statements to Java methods instead of generating SQL |
| Spring Data JPA | Simplifies repositories on top of JPA/Hibernate |

## 5.4 Hibernate Architecture

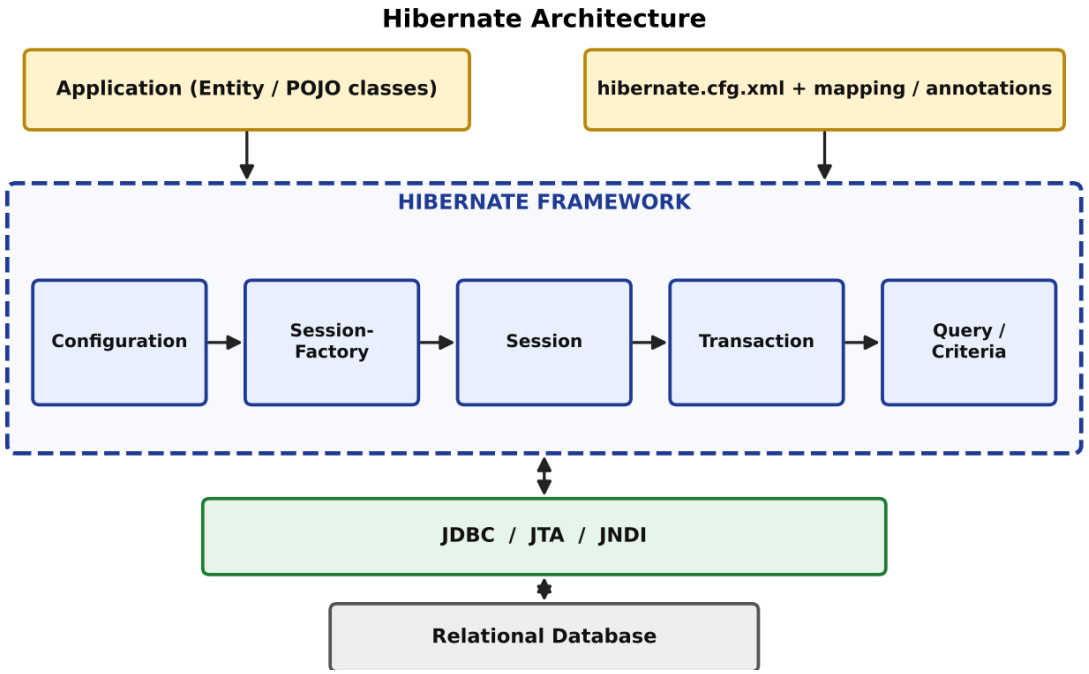



***Fig. 11 Hibernate architecture***

| Component | Role |
|---|---|
| Configuration | Reads hibernate.cfg.xml (DB URL, user, dialect) and mapping information |
| SessionFactory | Heavyweight, thread-safe object created once per database; produces Session objects; holds the second-level cache |
| Session | Lightweight, single-threaded object representing one conversation with the database; wraps a JDBC connection; first-level cache |
| Transaction | Defines atomic unit of work (beginTransaction, commit, rollback) |
| Query / Criteria | Fetches data using HQL, native SQL or the Criteria API |
| JDBC / JTA / JNDI | Underlying Java APIs used by Hibernate to talk to the database |

## 5.5 Mapping and CRUD with Hibernate

### *Entity class (annotation-based mapping)*

```
import jakarta.persistence.*;     // javax.persistence.* in older versions

@Entity
@Table(name = "student")
public class Student {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private int id;

  @Column(name = "name", nullable = false, length = 50)
  private String name;

  private String dept;
```

```
  // no-argument constructor, getters and setters
}
```

### *hibernate.cfg.xml*

```
<hibernate-configuration>
  <session-factory>
    <property
name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
    <property
name="hibernate.connection.url">jdbc:mysql://localhost:3306/college</property
>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">pwd</property>
    <property
name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.hbm2ddl.auto">update</property>
    <property name="hibernate.show_sql">true</property>
    <mapping class="Student"/>
  </session-factory>
</hibernate-configuration>
```

### *CRUD operations*

```
SessionFactory sf = new Configuration().configure().buildSessionFactory();
Session session = sf.openSession();
Transaction tx = session.beginTransaction();

Student s = new Student();  s.setName("Anu");  s.setDept("IT");
session.persist(s);                       // CREATE  (save() in older
versions)

Student x = session.get(Student.class, 1); // READ by primary key
x.setDept("CSE");                         // UPDATE – detected automatically
on commit
session.remove(x);                        // DELETE  (delete() in older
versions)

tx.commit();
session.close();
```

### *HQL (Hibernate Query Language)*

HQL is object-oriented: it uses class and property names, not table and column names.

```
List<Student> list = session
    .createQuery("from Student where dept = :d order by name", Student.class)
    .setParameter("d", "CSE")
    .list();
```

## 5.6 Association Mappings

| Annotation | Relationship | Example |
|---|---|---|
| @OneToOne | One entity linked to one entity | Student – IDCard |

| Annotation | Relationship | Example |
|---|---|---|
| @OneToMany / @ManyToOne | Parent with many children | Department – Students |
| @ManyToMany | Many on both sides (join table) | Student – Courses |

### 5.7 Entity (Object) States in Hibernate

| State | Meaning |
|---|---|
| Transient | New object created with new; not associated with any Session or database row |
| Persistent | Associated with an open Session; changes are saved automatically on commit |
| Detached | Was persistent but the Session is closed; can be re-attached using merge() / update() |
| Removed | Scheduled for deletion from the database |

### 5.8 JDBC vs ORM (Hibernate)

| JDBC | ORM / Hibernate |
|---|---|
| Developer writes SQL and mapping code | Framework generates SQL and does mapping |
| Lots of boilerplate (connection, statement, ResultSet) | Few lines: persist(), get(), remove() |
| Database dependent SQL | Database independent through dialects |
| No built-in caching | First-level and second-level caching |
| Full control, best for complex or bulk SQL | Productive for object-oriented domain models |
| Manual handling of relationships (joins) | Relationships handled by annotations |

## 6. Important Questions

### 6.1 Two-Mark Questions

**1. Why is session management required?**

HTTP is stateless; sessions let the server remember a user across multiple requests.

**2. List the session tracking techniques.**

Cookies, URL rewriting, hidden form fields and HttpSession.

**3. What is the difference between getSession() and getSession(false)?**

getSession() creates a session if none exists; getSession(false) returns null instead.

**4. How is a session invalidated?**

By calling session.invalidate() or when the inactive interval (timeout) expires.

**5. What is a cookie?**

A small name–value text sent by the server, stored by the browser and returned with later requests.

**6. Differentiate session cookie and persistent cookie.**

A session cookie has no expiry and is deleted when the browser closes; a persistent cookie has maxAge set and stays on disk.

**7. How do you delete a cookie?**

Create a cookie with the same name and path, call setMaxAge(0) and add it to the response.

**8. Differentiate authentication and authorization.**

Authentication verifies who the user is; authorization decides what the user may do.

**9. What is RBAC?**

Role-Based Access Control – permissions are assigned to roles and users get roles.

**10. What are the JDBC driver types?**

Type 1 JDBC–ODBC bridge, Type 2 native API, Type 3 network protocol, Type 4 thin pure-Java driver.

**11. Why is PreparedStatement preferred over Statement?**

It is precompiled, faster for repeated use and prevents SQL injection.

**12. What is connection pooling?**

Keeping a set of ready database connections that are reused, avoiding the cost of creating a new connection per request.

**13. What is ORM? Name a tool.**

Mapping of classes to tables and objects to rows by a framework; Hibernate.

**14. Define HQL.**

Hibernate Query Language – an object-oriented query language using entity names and properties.

## 6.2 Long-Answer Questions (13/16 marks)

1. Explain the various session tracking techniques in servlets with suitable examples.
2. Explain the HttpSession interface and its life cycle. Write a servlet to count the number of visits of a user.
3. Describe cookies in detail. Write servlet programs to create, read and delete a cookie.
4. Explain authentication and authorization. Develop a login application using servlets, JSP and a database with session handling.
5. Explain container-managed security and role-based access control with a web.xml example.

6. Explain JDBC architecture and the types of JDBC drivers. Write the steps and a program to perform CRUD operations.
7. Discuss transaction management and connection pooling in JDBC.
8. What is ORM? Explain the architecture of Hibernate and show how an entity is mapped and persisted.
9. Compare JDBC and ORM with examples.

*— End of Notes —*