Java Technologies

Lecture 1

Introduction to Enterprise Programming

Fall, 2025

Course Description

- https://edu.info.uaic.ro/tehnologii-java
- https://profs.info.uaic.ro/cristian.frasinaru
- The Goal
- The Motivation
- Lectures and Assignments
- Programming Platform
- Resources
- Evaluation
  - Lab: problems, quizzes, projects
  - Exam: written test

Course Syllabus

1. Introduction to Enterprise Programming
2. Spring Boot
3. Beans, Dependency Injection & Cross-Cutting Concerns
4. Java Persistence API & Spring Data JPA
5. API Development: RESTful Services
6. Securing Applications and Services
7. Intermezzo
8. Asynchronous Communication & Messaging
9. Microservice Architectures
10. Building Cloud-Native Applications
11. Data Management in Microservices
12. Testing & Deployment

Today's Agenda

- Enterprise Programming
- Software Development Life Cycle
- Monolithic vs Microservices Architectures
- Reliable, Scalable, Maintainable Systems
- Application Frameworks & Application Servers
- Java/Jakarta Enterprise Edition
- Eclipse MicroProfile
- Spring Framework & Spring Boot
- The Lifecycle of a Java EE Web App
- Servlets API
- HTTP Request Threads, Concurrency
- Attributes, Scopes, Web Sessions

Enterprise Applications

- EA = Software system designed to operate in a corporate environment such as business or government:
  $\rightarrow$ **targets an organization rather than individual users**.
- **Complex, large-scale, mission critical, portable, scalable, reliable, secure, transactional, distributed system**.
- EA software consists of a group of applications
  - **sharing** various resources (users, clients, data) and
  - **integrating** core business processes: sales, accounting, finance, human-resources, inventory and manufacturing, document management.
- Martin Fowler: *"EAs are about the display, manipulation, and storage of large amounts of often complex data and the support or automation of business processes with that data."*

Software Development Life Cycle

Creating a complex enterprise application is like building a city. Instead of a single house — every stage of the lifecycle comes with its own structural, political, and logistical challenges.

1. Requirements Gathering & Analysis
2. System Architecture & Design
3. Implementation / Development
4. Testing & Quality Assurance
5. Deployment & Integration
6. Maintenance & Support
7. Scaling & Evolution
8. Decommissioning / Replacement

1. Requirements Gathering & Analysis

Challenges

- **Ambiguity**: The stories can be vague or contradictory.
- **Changing scope**: Business needs evolve mid-way (scope creep).
- **Communication gaps**: Tech team vs. business team.
- **Overlooking edge cases**: Critical scenarios get missed.
- **Prioritization conflicts**: Essential features disagreement.

1. Requirements Gathering & Analysis

Challenges

- **Ambiguity**: The stories can be vague or contradictory.
- **Changing scope**: Business needs evolve mid-way (scope creep).
- **Communication gaps**: Tech team vs. business team.
- **Overlooking edge cases**: Critical scenarios get missed.
- **Prioritization conflicts**: Essential features disagreement.

Solutions

- Use UML or C4 diagrams.
- Define API contracts with OpenAPI Specification (OAS),
- Apply Domain-Driven Design (DDD).
- Use Agile iterations and prototypes.

2. System Architecture & Design

Challenges

- **Technology stack decision paralysis**: Which frameworks? 😟
- **Scalability foresight**: Designing for today, forgetting tomorrow.
- **Integration complexity**: Connecting with legacy systems.
- **Security planning**: Not treated as a foundational concern.
- **Over-engineering**: Adding unnecessary abstractions.

2. System Architecture & Design

Challenges

- **Technology stack decision paralysis**: Which frameworks? 😵
- **Scalability foresight**: Designing for today, forgetting tomorrow.
- **Integration complexity**: Connecting with legacy systems.
- **Security planning**: Not treated as a foundational concern.
- **Over-engineering**: Adding unnecessary abstractions.

Solutions

- Decide between a *monolithic* or a *microservice* architecture. Use modular microservices to avoid technology lock-in.
- Use technologies that support out-of-the-box scalability.
- Apply layered architecture (Controller $\rightarrow$ Service $\rightarrow$ Repository)
- Use Maven/Gradle for dependency management.
- Integrate apps using event-driven, message-based approaches.
- Design for security from the start.

3. Implementation / Development

Challenges

- **Team coordination**: Merge conflicts.
- **Code quality**: Technical debt accumulates.
- **Dependency management**: Library update problems.
- **Performance pitfalls**: Inefficient queries, memory leaks.
- **Knowledge silos**: Only one developer understands it.

3. Implementation / Development

Challenges

- **Team coordination**: Merge conflicts.
- **Code quality**: Technical debt accumulates.
- **Dependency management**: Library update problems.
- **Performance pitfalls**: Inefficient queries, memory leaks.
- **Knowledge silos**: Only one developer understands it.

Solutions

- Enforce coding standards with Checkstyle, PMD, SonarQube.
- Adhere to SOLID principles and established design patterns.
- Use ORMs (Object-Relational Mapping) to avoid boilerplate SQL, but fall back to native queries when performance is critical.
- Use shared code reviews and pair programming.

4. Testing & Quality Assurance

Challenges

- **Incomplete test coverage**: Focusing on happy paths.
- **Flaky tests**: Tests pass locally but fail in CI/CD.
- **Environment mismatches**: "It works on my machine".
- **Data setup complexity**: Creating realistic test datasets.
- **Security testing gaps**: No penetration or vulnerability testing.

4. Testing & Quality Assurance

Challenges

- **Incomplete test coverage**: Focusing on happy paths.
- **Flaky tests**: Tests pass locally but fail in CI/CD.
- **Environment mismatches**: "It works on my machine".
- **Data setup complexity**: Creating realistic test datasets.
- **Security testing gaps**: No penetration or vulnerability testing.

Solutions

- Unit testing (JUnit, Mockito).
- Integration tests (Testcontainers).
- Full-stack context testing.
- Load testing (JMeter, Gatling).
- Security vulnerability scanning (OWASP).

5. Deployment & Integration

Challenges

- **Environment drift**: Dev, test, and prod configs don't match.
- **Downtime during rollout**: Users experience disruption.
- **Version compatibility**: New release breaks older API clients.
- **Deployment fails**: Poor rollback strategies.
- **Configuration secrets**: Storing passwords or API keys securely.

5. Deployment & Integration

Challenges

- **Environment drift**: Dev, test, and prod configs don't match.
- **Downtime during rollout**: Users experience disruption.
- **Version compatibility**: New release breaks older API clients.
- **Deployment fails**: Poor rollback strategies.
- **Configuration secrets**: Storing passwords or API keys securely.

Solutions

- Build fat JARs or WARs for portability.
- Containerization and Orchestration (Docker, Kubernetes).
- Automate CI/CD (GitHub Actions, Jenkins, GitLab CI).
- Externalize configuration with environment variables.
- Secure secrets (Vault, AWS Secrets Manager).

6. Maintenance & Support

Challenges

- **Bug backlog**: Small defects pile up over time.
- **Knowledge turnover**: Key engineers leave.
- **Performance degradation**: As data grows, queries slow down.
- **Legacy technical debt**: Old versions become unsupported.
- **User change requests**: New business needs emerge.

6. Maintenance & Support

Challenges

- **Bug backlog**: Small defects pile up over time.
- **Knowledge turnover**: Key engineers leave.
- **Performance degradation**: As data grows, queries slow down.
- **Legacy technical debt**: Old versions become unsupported.
- **User change requests**: New business needs emerge.

Solutions

- Dashboards monitoring (Micrometer, Prometheus, Grafana).
- Log effectively (Logback, ELK: Elasticsearch, Logstash, Kibana)
- Keep dependencies updated with Maven Versions Plugin.
- Write self-documenting code and maintain an ADR log (Architecture Decision Record).

7. Scaling & Evolution

Challenges

- **Architectural limits**: The design can't handle increased load.
- **Cost optimization**: Scaling resources without overspending.
- **Feature bloat**: The app becomes slow and overly complex.
- **Migration pain**: Moving to new technologies.
- **Security posture drift**: Threat landscape changes.

7. Scaling & Evolution

Challenges

- **Architectural limits**: The design can't handle increased load.
- **Cost optimization**: Scaling resources without overspending.
- **Feature bloat**: The app becomes slow and overly complex.
- **Migration pain**: Moving to new technologies.
- **Security posture drift**: Threat landscape changes.

Solutions

- Scale horizontally or on-demand (autoscaling).
- Use reactive programming for high concurrency
- Apply CQRS (Command Query Responsibility Segregation) + Event Sourcing patterns for large-scale data changes.
- Migrate incrementally using the Strangler Fig pattern.

8. Decommissioning / Replacement

Challenges

- **Data migration**: Moving interconnected data without loss.
- **User adoption**: Moving everyone to the new system.
- **Parallel systems complexity**: Old and new side-by-side.
- **Legal compliance**: Retain necessary, remove obsolete.
- **Cultural resistance**: "We've always done it this way".

8. Decommissioning / Replacement

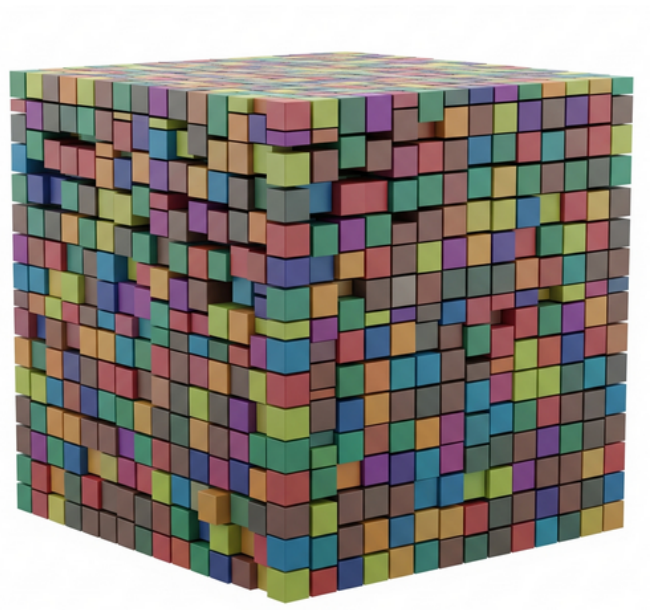
Challenges

- **Data migration**: Moving interconnected data without loss.
- **User adoption**: Moving everyone to the new system.
- **Parallel systems complexity**: Old and new side-by-side.
- **Legal compliance**: Retain necessary, remove obsolete.
- **Cultural resistance**: "We've always done it this way".

Solutions

- Use technologies for large-scale data migration.
- Keep backward compatibility with API versioning.
- Use feature toggles to switch from old to new versions gradually.
- Maintain archival systems or other auditing tools.

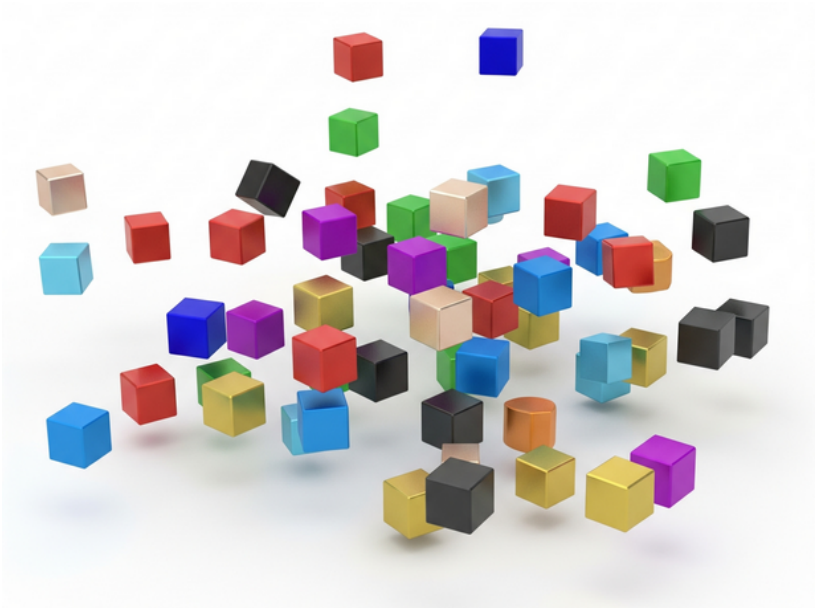
Monolithic Architecture



The application is built as a single and indivisible unit, where all modules (UI, business logic, data access) are packaged and deployed together.

- **Pros**: Easy to develop, test, deploy. Single codebase.
- **Cons**: Less flexible, Difficult to scale, Hard to maintain.

Microservices Architecture



The application is split into small, independent modules each responsible for one business capability, running in its own process and communicating over the network.

- **Pros**: Flexibility, Fault tolerance, Autoscaling.
- **Cons**: Complexity, Performance and Communication "issues"

Reliable, Scalable, Maintainable Systems

- **Reliable**: Working correctly, even when things go wrong.
  - Hardware Faults
  - Software Errors
  - Human Errors
- **Scalable**: Coping with increased load.
  - Hardware Scaling
  - Load Balancing
  - Caching
  - Efficient Design, Algorithms and Data Structures
- **Maintainable**: Making the system work over time.
  - Simplicity
  - Operability
  - Evolvability

High Availability

- **High Availability (HA)** is the ability of a system to operate at maximum performance, continuously, without failing for a long period of time.
- **Mission Critical Application** - its failure or disruption would cause an entire operation or business to grind to a halt.
- Methods for obtaining HA:
  - **Redundancy**: Hard / Soft / Network / Geographic.
  - **Failover**: The transfer of workload from a primary system to a secondary system, hot / cold.
  - **Clustering**: Group of servers acting as one.
  - **Data Replication**: Copy data across servers.
- **Measuring HA**: Percentage of uptime over a given period.
  - **99.9%** (Three Nines) $\approx$ 8.76 hours of downtime per year.
  - **99.99%** (Four Nines) $\approx$ 53 minutes of downtime per year.
  - **99.999%** (Five Nines) $\approx$ 5 minutes and 15 seconds 

Scalability

- **Scalability** refers to the ability of a system, network, or process to handle increased workload or demand by adding resources (CPU, memory, storage), without compromising performance, reliability, or efficiency.
- **Types of Scalability**
  - Vertical Scalability (Scaling Up)
  - Horizontal Scalability (Scaling Out)
  - On-Demand Scalability (Autoscaling)
- **Metrics**
  - **Throughput** (number of requests handled per time unit)
  - **Latency** (time taken to complete a request)
  - **Elasticity** (how quickly a system can scale)
  - **Cost-Efficiency** (resources used versus performance gained)

Application Frameworks and Application Servers

- An **application framework** is a set of libraries, components, and tools that provides a foundation for building an application. Its primary role is **to simplify development** by providing common functionalities, such as: logging, testing, data access, security, MVC, AOP, IOC.
  $\rightarrow$ Spring Framework, Apache Struts, Google Guice, Vaadin...

Application Frameworks and Application Servers

- An **application framework** is a set of libraries, components, and tools that provides a foundation for building an application. Its primary role is **to simplify development** by providing common functionalities, such as: logging, testing, data access, security, MVC, AOP, IOC.
  $\rightarrow$ Spring Framework, Apache Struts, Google Guice, Vaadin...

- An **application server** is a software environment that provides a platform for **running and managing an application**. It handles session management, resources, security, and others.
  $\rightarrow$ Apache Tomcat, Jetty, TomEE, GlassFish, Payara, WildFly, IBM WebSphere, Oracle WebLogic, Geronimo...

- Sometimes, the distinction may not be clear. 😟

Jakarta Enterprise Edition

- **Jakarta EE**, formerly known as Java EE, is a set of open-source specifications for building enterprise-grade Java applications.
- **Based on Specifications**: Jakarta EE API is a set of specifications, primarily consisting of interfaces and abstract classes, that define how an enterprise application should be developed, what components it should include, and how they should interact with each other.
- **Not an Implementation**: the implementation of Jakarta EE specifications are included in **application servers** such as Tomcat (lightweight) or GlassFish (heavy, reference impl.).
- It offers **deployment portability** and avoids **vendor lock-in**.
- `https://jakarta.ee`

We'll just call it Java EE.

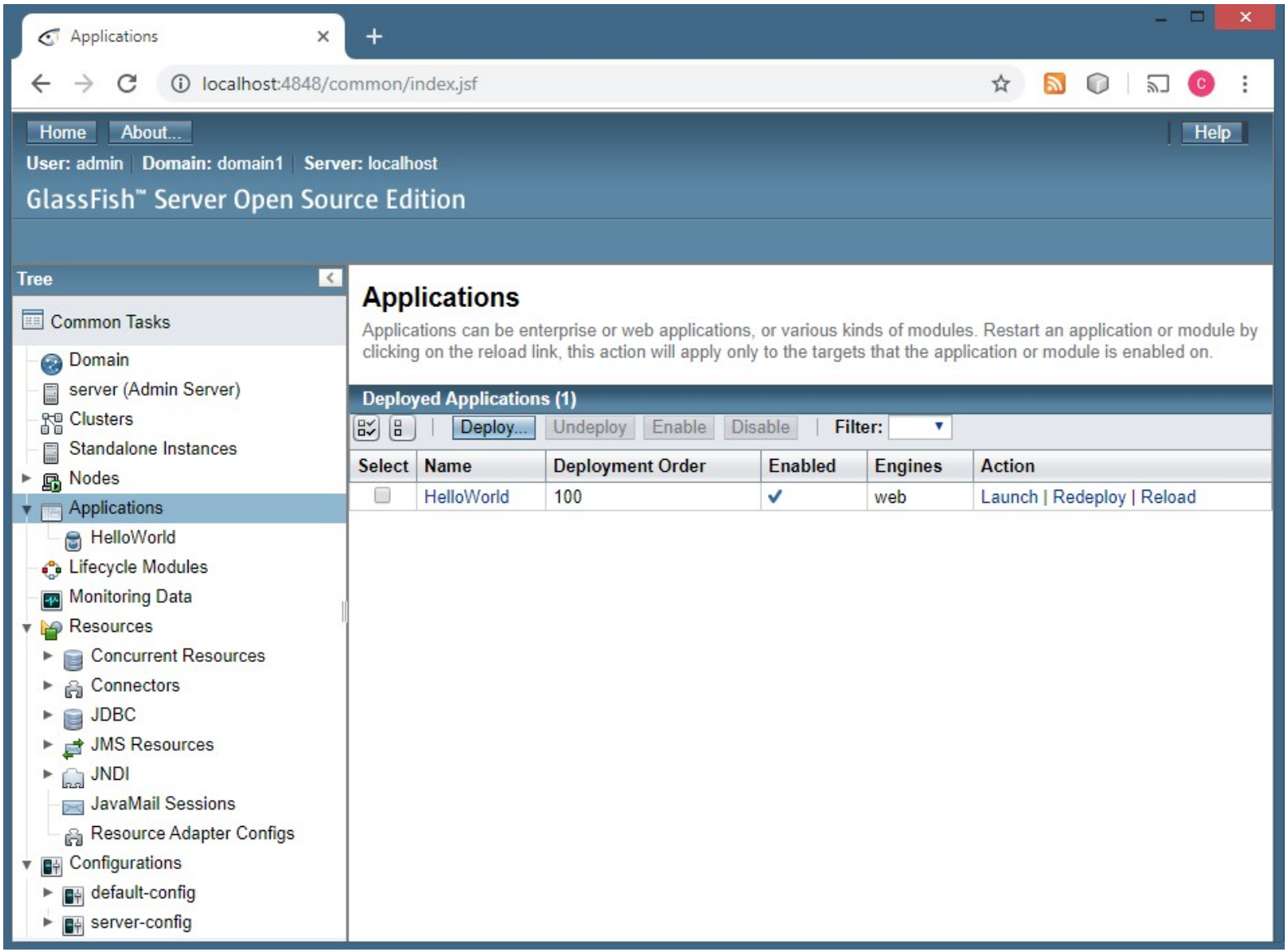
Core Java EE Specifications

- Servlet API
- JavaServer Pages (JSP)
- JavaServer Faces (JSF)
- Enterprise JavaBeans (EJB)
- Context and Dependency Injection (CDI)
- Java Persistence API (JPA)
- Java Transaction API (JTA)
- Java Message Service (JMS)
- JavaMail API
- Java API for RESTful Web Services (JAX-RS)
- Java API for XML Web Services (JAX-WS)
- Java EE Security API
- Java API for JSON Processing and Binding (JSON-P, JSON-N)
- Java API for WebSocket
- Java Connector Architecture (JCA), ...

EE Application Servers

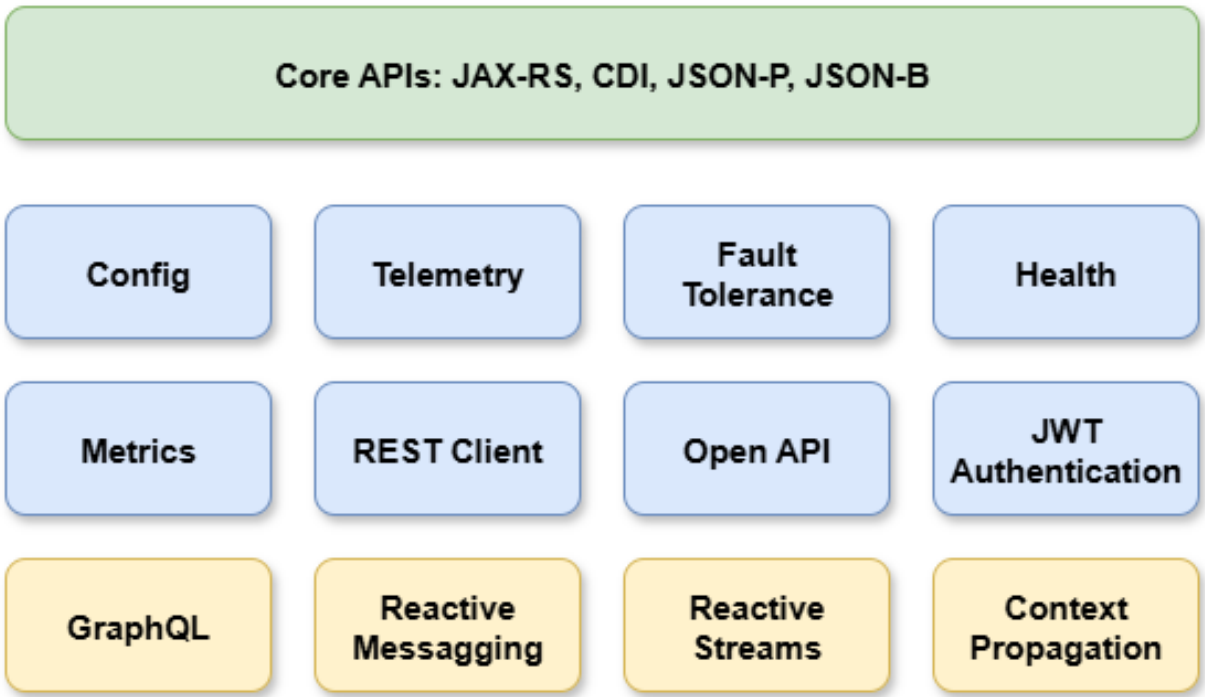
- A **Java EE server** provides a runtime environment to deploy, manage, and execute enterprise Java applications.
- It **runs as a JVM process** and provides network services on various ports, such as 8080 (HTTP), 8181 (HTTPS).
- It **manages multiple applications**, which can be configured independently and may share common resources.
- It offers various **administration utilities**, such as command line deployment tools or an admin console.
- Can be **Full Java EE (heavy)**, implementing all enterprise specifications (such as GlassFish, Payara, or WildFly) or may offer only the **Web Profile (light)** (such as Tomcat, or Jetty).
- It includes a **standalone HTTP web server**.

Example: GlassFish Administration Console



Eclipse MicroProfile

- **Eclipse MicroProfile** is an open-source initiative designed to simplify and optimize Java EE specifications for building microservices and cloud-native applications.
- Offers a **standardized, portable API** for microservices.



Spring Framework and Spring Boot

- **Spring Framework** is a comprehensive, modular framework for building enterprise-level applications. Its core features are:
  - DI, IOC, AOP, Transaction management, data access.
  - MVC framework for building web applications.
  - Security, messaging, batch processing, and many more modules.
  - Requires manual config and deployment to Java EE server.

Spring Framework and Spring Boot

- **Spring Framework** is a comprehensive, modular framework for building enterprise-level applications. Its core features are:
  - DI, IOC, AOP, Transaction management, data access.
  - MVC framework for building web applications.
  - Security, messaging, batch processing, and many more modules.
  - Requires manual config and deployment to Java EE server.

- **Spring Boot** is a **convention-over-configuration** extension of the Spring Framework designed to simplify setup and development of Spring applications. Its core features are
  - Auto-configuration (based on dependencies in your project).
  - Embedded servers (Tomcat, Jetty, or Undertow).
  - Starter dependencies (easy-to-use "starters").
  - Defaults for most settings, reducing boilerplate.
  - Production-ready features like health checks, metrics.

The Relation Between Java EE and Spring

- Spring Framework
  - was create to simplify the complexity of early Java EE (J2EE);
  - provided a more lightweight approach;
  - focused on rapid development, rather than strict standards.

- Modern Spring versions have adopted many of the same concepts and specifications as Java EE, such as:
  - **JPA** (Java Persistence API)
  - **JTA** (Java Transaction API)
  - **JMS** (Java Message Service)
  - **Servlet API** (Spring MVC)

- Spring Boot uses embedded servers like Tomcat, which are implementations of Java EE specifications.

The Lifecycle of a Simple Java EE Web App

- **Develop** the components and the deployment descriptors.

```
\MyApplication
    Web Pages
    Resources
    \WEB-INF
        web.xml
        Configuration files
        \classes
            .class
        \lib
            .jar
```

- **Compile** the application classes.
- **Package** the application into a deployable unit: war, ear.
- **Deploy** the application using the application server tools.
- **Access** the URL that references the welcome file of the web app.

Where to Start?

- We are in the context of developing a **distributed application**, a software system in which agents (components):
  - communicate over the network and
  - coordinate their actions in order to achieve a common goal.
- What is the most common way of communication?
  $\rightarrow$ **Message Passing**
- What kind of messages are the components passing?
  $\rightarrow$ **Requests / Responses**
- There should be **a standard way** to implement a component that receives a request and returns a response over a network protocol (any protocol, not just HTTP).

The Servlet Component

- A **servlet** is a component that offers a service over the network, using a request-response programming model.

```
package javax.servlet;

public interface Servlet {

    public void init(ServletConfig config)
        throws ServletException;

    public void service(ServletRequest req, ServletResponse res)
        throws ServletException, IOException;

    public void destroy();
    public ServletConfig getServletConfig();
    public String getServletInfo();
}
```

```
public abstract class GenericServlet implements Servlet { … }
public abstract class HttpServlet extends GenericServlet { … }
```

Example: The "HelloWorld" Servlet

```java
@WebServlet("/hello") // ← value, urlPatterns
public class HelloServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
            throws ServletException, IOException {

        // Get the "name" parameter from the query string
        String name = request.getParameter("name");
        if (name == null || name.isBlank()) {
            name = "World";
        }
        // Set response type to plain text
        response.setContentType("text/plain"); // MIME type
        // Write the response to the client
        try (PrintWriter out = response.getWriter()) {
            out.println("Hello " + name);
        }
    }
} // An alternative to @WebServlet is to declare it in web.xml
```

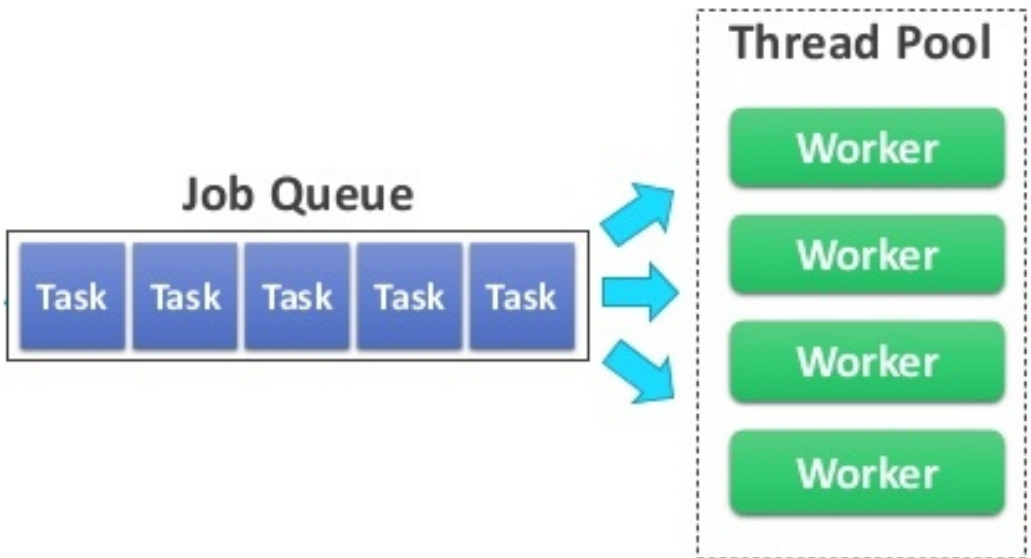
```
http://localhost:8080/MyApplication/hello?name=Joe
```

The Architectural Role of Servlets

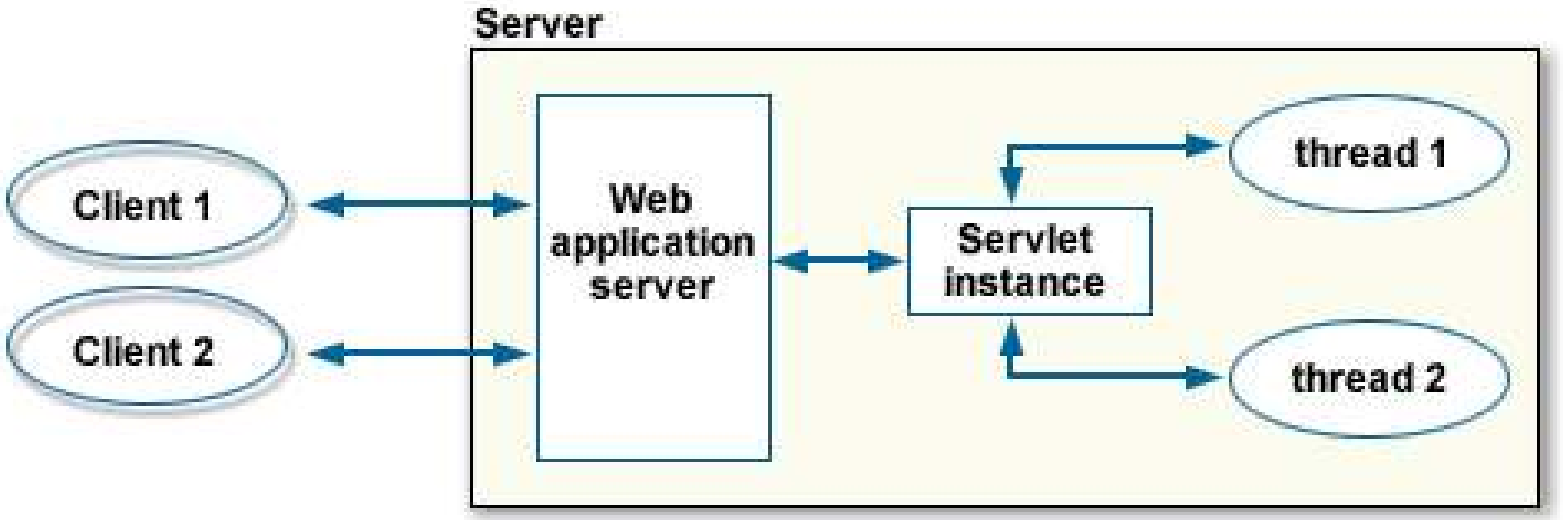
- **Request Handling**
  - Receive HTTP requests (GET, POST, etc.) from clients.
  - Extract request parameters, headers, cookies, and body content.
  - Perform validation, preprocessing, or forwarding.
- **Business Logic Integration**
  - Act as a controller in MVC-like architectures
  - Invoke business services, database operations, or backend APIs.
  - Coordinate the flow between the view and the model.
- **Response Generation**
  - Construct responses in any format, by using helpers.
  - Set HTTP response headers (content type, caching, etc.).
  - Send the response back to the client.
- Many higher-level frameworks, like Spring Boot, Java Server Faces, or Struts are **built on top of the servlet specification**.
  $\rightarrow$ A core component of Spring is `DispatcherServlet`

HTTP Request Threads

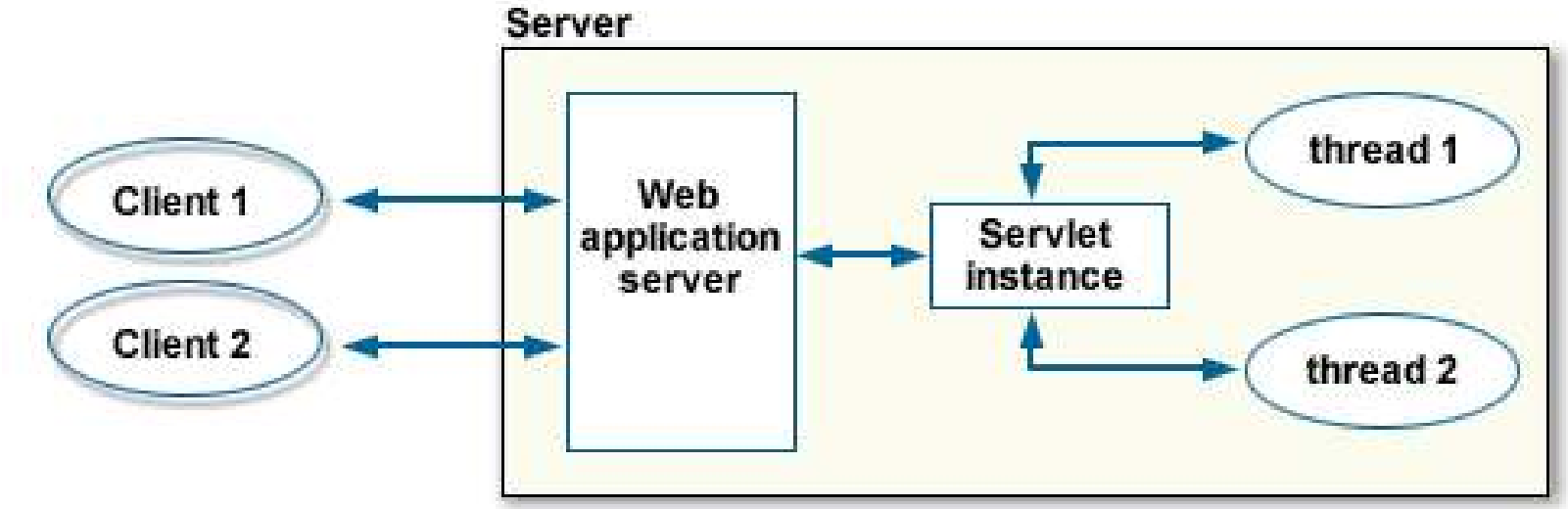
- **HTTP Listener**: part of a server that listens for incoming requests, e.g. http-listener-1 (8080), httplistener-2 (8181)
- **HTTP Request Threads**: the worker threads from the application server's thread pool that process incoming requests.
  The pool is configurable using admin console: Max Queue Size (4096), Min/Max Thread Pool (50/200), Idle Thread Timeout (900 sec)
- **Blocking** (default) vs. **Non-blocking** (@Async, reactive)



Servlet Concurrency



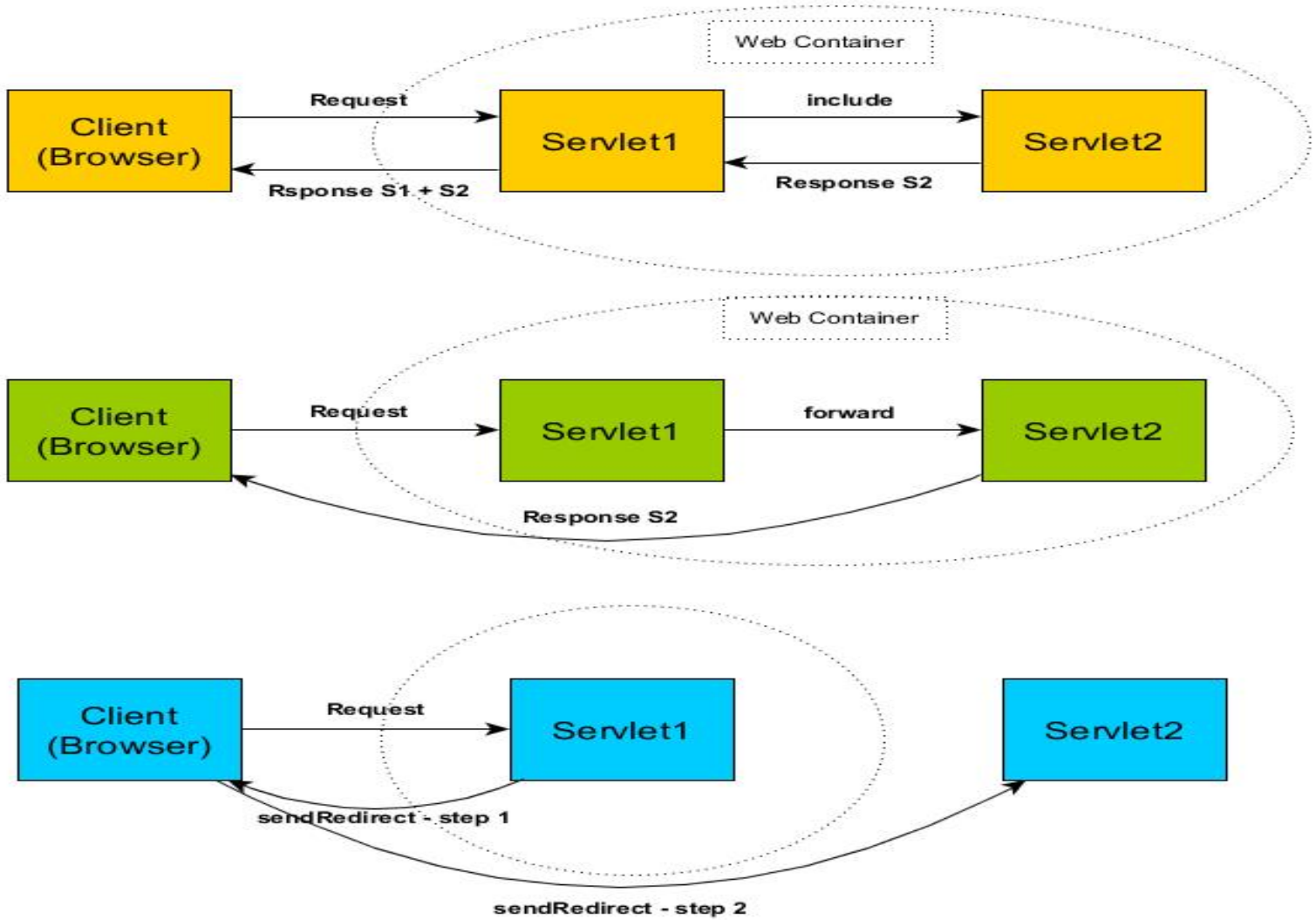
Servlet Concurrency




```
public class ThreadSafeTestServlet extends HttpServlet {
  private String notThreadSafe; // Shared member variable
  public void doGet(HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException {
    notThreadSafe = req.getParameter("param");      ⚠
    System.out.println(notThreadSafe);
    String threadSafe = req.getParameter("param"); 👍
  }
}
```

Servlet Communication

RequestDispatcher (Include / Forward) vs. Redirect



Attributes and Scopes

- Servlets are also able to communicate with other components using the **shared memory** paradigm.
- An **attribute** is an object that can be set, get or removed from a specified **scope**: `request, session, application`.

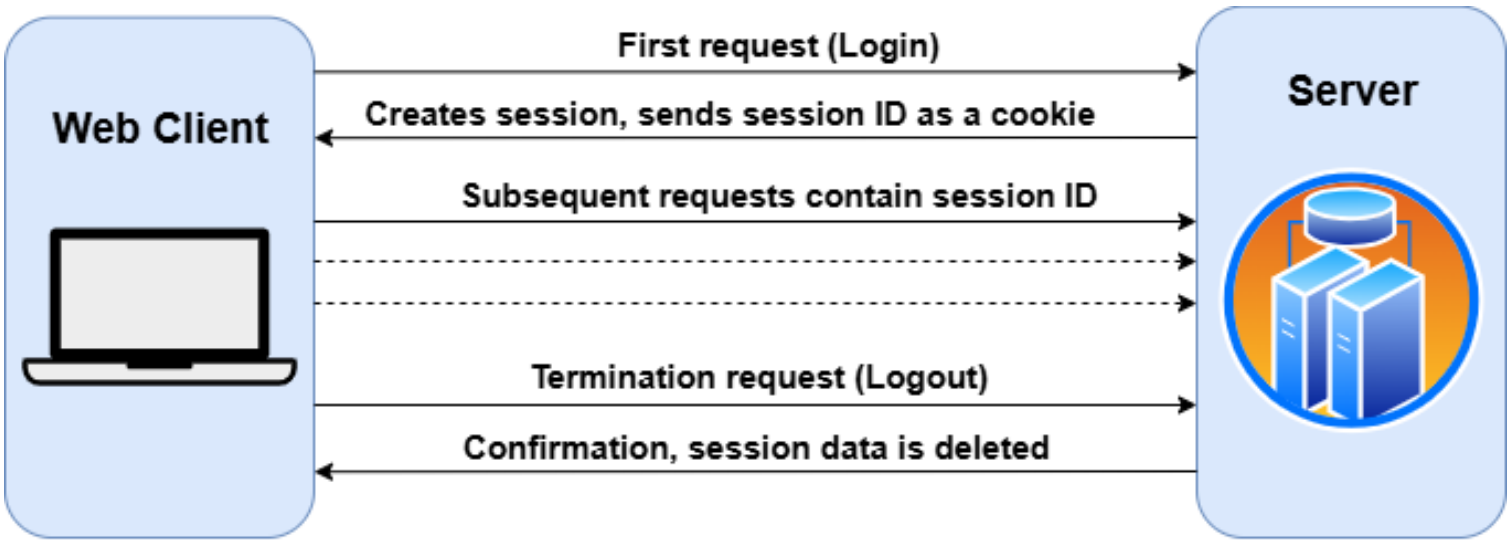
```
request.setAttribute("message", "Hello World");
session.setAttribute("shoppingCart",new ShoppingCart());
servletContext.setAttribute("version", "0.0.1")
```

- A `ServletContext` object is the application scope. It allows servlets to communicate and share data with each other. There is one such global context per application, per JVM.

```
ServletContext context = this.getServletContext();
context.log(context.getServerInfo());
context.getAttribute("version");
context.getInitParameter("resourcesPath");
```

Web Sessions

- A **web session** is a "dialogue" between a client and a web application: *login* $\rightarrow$ *buy stuff* $\rightarrow$ *logout*.
- HTTP is a **stateless** protocol; therefore, a mechanism is required to **maintain state** across multiple HTTP requests from the same client.



- **Session Data**: In-memory (default) vs. Distributed (Redis)
- **URL Rewriting**: when cookies are disabled.

Example: Creating and Using a Web Session

```
@WebServlet("/sum")
public class SumNumbersServlet extends HttpServlet {
  protected void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();
    // Get the current session or create a new one.
    HttpSession session = req.getSession(true);

    Double sum = (Double) session.getAttribute("sum");
    if (sum == null) sum = 0.0;  // or use session.isNew()
    double number = Double.parseDouble(req.getParameter("number")); ⚠
    if (number == 0) {
        session.invalidate(); // Terminate the session
        out.println("Session terminated.");
    } else {
        sum += number;
        session.setAttribute("sum", sum);
        out.println("Sum =  " + sum);
    }
  }
} // A Cookie API is available through request/response objects.
```