SecureCourse: The Comprehensive Security Handbook

Version: 2.0.0 (Enterprise Edition)
Date: November 2025
Classification: Internal Engineering & Security Masterclass
Author: SecureCourse Security Team

Table of Contents

1. Executive Summary
2. System Origin & Vision
3. Complete Threat Model
4. Feature Overview
5. End-to-End Workflow
6. Security Architecture
7. Deep Dive: SQL Injection
8. Deep Dive: Cross-Site Scripting (XSS)
9. Deep Dive: File Upload Vulnerabilities
10. Deep Dive: Authentication & Session Security
11. Backend Technical Architecture
12. Frontend Technical Architecture
13. Privacy & Data Handling
14. Performance & Scalability
15. Roadmap & Future Enhancements
16. Security Best Practices (OWASP)
17. Glossary

---

1. Executive Summary

1.1 Purpose

SecureCourse is an enterprise-grade educational security platform designed to bridge the critical gap between theoretical security knowledge and practical application. It serves as a controlled, interactive environment where developers, security engineers, and QA professionals can observe, exploit, and mitigate common web vulnerabilities in real-time.

1.2 Problem Statement

Modern software engineering often treats security as an afterthought or a compliance checklist. Developers rarely have the opportunity to witness the mechanical execution of an attack against their own code. This lack of "offensive visibility" leads to:

- Misunderstanding of root causes (e.g., *why* string concatenation is dangerous).
- Ineffective remediation (e.g., using blacklists instead of parameterized queries).
- Recurring vulnerabilities in production systems.

1.3 Target Audience

- **Software Engineers:** To understand secure coding practices and the "why" behind security requirements.
- **Security Champions:** To demonstrate vulnerabilities during internal training sessions.

- **QA Engineers:** To learn how to test for basic security flaws.
- **Engineering Managers:** To understand the technical debt associated with insecure patterns.

1.4 Security-First Design Philosophy

SecureCourse follows a **"Security by Visibility"** philosophy. Rather than hiding complexity, it exposes the underlying mechanics of both vulnerabilities and their defenses. The platform is built on the principle that *understanding the attack is the first step to building the defense*.

---

2. System Origin & Vision

2.1 Origin Story

SecureCourse was born from the observation that many developers struggle to internalize security concepts because they rarely see them "break" in a controlled way. Production systems are (ideally) secure, and vulnerable apps (like WebGoat or DVWA) can be difficult to set up or too abstract. SecureCourse aims to be the "Goldilocks" solution: a modern, realistic application stack (Spring Boot + React) that behaves like a real enterprise app but breaks like a vulnerable one when told to.

2.2 Core Goals

1. **Demystification:** Remove the "black box" nature of security exploits.
2. **Interactive Learning:** Enable "cause and effect" learning through the Toggle System.
3. **Safe Playground:** Provide a sandboxed environment where destructive actions (like dropping tables) can be simulated without consequence.
4. **Modern Stack Relevance:** Demonstrate vulnerabilities in the context of modern frameworks (React, Spring Boot), showing that frameworks alone do not guarantee security.

2.3 Long-term Mission

To become the standard reference implementation for demonstrating OWASP Top 10 vulnerabilities in a Java/React stack, eventually expanding to include advanced topics like SSRF, deserialization attacks, and JWT manipulation.

---

3. Complete Threat Model

3.1 Attacker Profiles

Profile	Description	Motivation	Capabilities
Script Kiddie	Uses automated tools to scan for low-hanging fruit.	Notoriety, vandalism.	Low. Relies on pre-made scripts (SQLMap, Burp Suite).
Malicious Insider	Employee or contractor with valid credentials.	Revenge, financial gain, espionage.	High. Knows internal logic, has valid access.
Opportunistic Attacker	Scans internet for unpatched vulnerabilities.	Botnet recruitment, crypto mining.	Medium. Automated but persistent.

Advanced Persistent Threat (APT)	Targeted, funded attacker.	Data theft, sabotage.	Very High. Custom exploits, zero-days.

3.2 Attack Surface Breakdown

The attack surface is the sum of all points where an unauthorized user can try to enter data to or extract data from an environment.

1. **Public Endpoints:**
   - `POST /api/auth/login` : Entry point for SQL Injection and Credential Stuffing.
   - `GET /` : React frontend assets (potential for sensitive info leakage in bundles).
2. **Authenticated Endpoints:**
   - `POST /api/comments` : Entry point for Stored XSS.
   - `POST /api/files/upload` : Entry point for Malicious File Uploads (RCE, XSS).
   - `GET /api/courses/*` : Data retrieval (IDOR potential).
3. **Data Layer:**
   - MySQL Database: Target for SQL Injection.
   - Local Filesystem: Target for Path Traversal and Shell Uploads.

3.3 Risk Classification (STRIDE)

We utilize the Microsoft STRIDE model to categorize threats:

Threat	Category	Risk	Mitigation Strategy
Spoofing	Identity	**High**	Strong Session Management, BCrypt Hashing, HttpOnly Cookies.
Tampering	Integrity	**High**	Input Validation, Parameterized Queries, Immutable Logs (planned).
Repudiation	Logging	**Medium**	Comprehensive Audit Logging of all security-critical events.
Info Disclosure	Confidentiality	**Critical**	Generic Error Messages, Access Controls, Least Privilege DB Users.
DoS	Availability	**Medium**	Rate Limiting (planned), Resource Quotas on File Uploads.
Elevation	Authorization	**High**	Role-Based Access Control (RBAC), Strict Controller Logic.

3.4 Security Goals (CIA Triad)

- **Confidentiality:** User credentials and private course data must remain protected. *Failure Scenario: SQL Injection dumping the user table.*
- **Integrity:** Course content and user comments must not be alterable by unauthorized parties. *Failure Scenario: Stored XSS modifying page content.*
- **Availability:** The platform must remain resilient against basic flooding attacks. *Failure Scenario: Large file uploads filling disk space.*

4. Feature Overview

4.1 Authentication Module

- **Secure Login:** Uses Spring Security with a custom `AuthenticationProvider`.
- **Session Management:** Stateful session management backed by `HttpSession`.
- **Password Security:** BCrypt hashing with a strength of 10.

4.2 Course Module

- **Content Delivery:** Serves rich-text course materials.
- **Dynamic Rendering:** React components fetch course data via REST API.
- **Access Control:** Only authenticated users can view course details.

4.3 File Upload Pipeline

- **Dual-Mode Handling:**
  - *Secure Mode:* Validates MIME types, extensions, and renames files using UUIDs.
  - *Insecure Mode:* Accepts any file type and preserves original filenames.
- **Storage:** Files are stored in a local `uploads/` directory outside the web root.

4.4 Comments & XSS Playground

- **Interactive Comments:** Users can post comments on courses.
- **XSS Demonstration:**
  - *Secure Mode:* HTML-encodes all output.
  - *Insecure Mode:* Renders raw HTML, allowing script execution.

4.5 Security Toggle System

The heart of SecureCourse is the `SecurityToggleService`. It maintains the state of three flags:

1. `sqlInjectionProtection`: Controls DB query construction.
2. `xssProtection`: Controls output encoding.
3. `fileUploadSecurity`: Controls file validation logic.

4.6 Administrative Tools

- **Dashboard:** Provides an overview of system status (simulated).
- **User Management:** (Planned) Interface to reset passwords or ban users.

4.7 Logging and Monitoring

- **Audit Logs:** Tracks login attempts, failed uploads, and toggle switches.
- **Error Logs:** Captures stack traces (server-side only).

---

5. End-to-End Workflow Explanation

5.1 User Interaction Flow

1. **Login:** User enters credentials. Backend verifies hash. Session cookie (`JSESSIONID`) set.
2. **Dashboard:** User sees list of available courses.
3. **Course View:** User selects a course. Frontend fetches details + comments.
4. **Interaction:**
   - *Comment:* User posts a question. Backend sanitizes (or not) and saves.

- *Upload:* User uploads a screenshot. Backend validates (or not) and saves.

5. **Logout:** Session invalidated. Cookie cleared.

5.2 File Upload Workflow

1. **Client:** Selects file -> `POST /api/files/upload`.
2. **Controller:** Receives `MultipartFile`.
3. **Service:** Checks `SecurityToggleService.isFileUploadSecurityEnabled()`.
   - *If ON:* Check Magic Bytes -> Check Extension -> Generate UUID -> Save.
   - *If OFF:* Save as `originalFilename`.
4. **Response:** Returns file URL (e.g., `/api/files/download/{filename}`).

5.3 Comment Sanitization Workflow

1. **Client:** Submits comment text -> `POST /api/comments`.
2. **Service:** Checks `SecurityToggleService.isXssProtectionEnabled()`.
   - *If ON:* `Encode.forHtml(content)` -> Save to DB.
   - *If OFF:* Save raw content to DB.
3. **Retrieval:** When fetching comments, the stored content is sent to frontend.
4. **Rendering:** Frontend uses `dangerouslySetInnerHTML` (which is safe ONLY if backend sanitized).

---

6. Security Architecture

6.1 Layer-by-Layer Breakdown

Presentation Layer (React)

- **Responsibility:** Rendering UI, managing client state.
- **Defense:** Auto-escaping in JSX (default).
- **Vulnerability:** Usage of `dangerouslySetInnerHTML` for the XSS demo.

API Layer (Spring Boot Controllers)

- **Responsibility:** Request routing, DTO mapping, basic validation.
- **Defense:** `@Valid` annotations, HTTP verb enforcement.

Service Layer (Business Logic)

- **Responsibility:** Core logic, Toggle checks.
- **Defense:** Transaction management, input sanitization logic.

Data Layer (JPA/Hibernate)

- **Responsibility:** DB interaction.
- **Defense:** Parameterized queries (when enabled).

6.2 Security Boundaries & Trust Zones

- **Zone 1: The Internet (Untrusted)**
  - Any request originating here is treated as potentially malicious.
  - Defense: WAF (simulated), Input Validation.
- **Zone 2: The DMZ (Semi-Trusted)**
  - The Application Server.
  - Defense: Least privilege OS user, limited network egress.
- **Zone 3: The Data Store (Trusted)**
  - The Database and Filesystem.

- Defense: Strong credentials, no direct internet access.

6.3 Defense Mechanisms

- **SQL Injection:** Prepared Statements (Java `PreparedStatement` / Hibernate HQL).
- **XSS:** Output Encoding (OWASP Java Encoder).
- **File Upload:** Whitelisting (MIME + Ext), Renaming, Path Normalization.
- **Auth:** BCrypt, Secure Cookies.

6.4 Attack Toggle Architecture

The toggle architecture is designed to be **thread-safe** and **instant**.

- **Storage:** Toggles are stored in a singleton Service bean (in-memory).
- **Access:** Accessed via `SecurityToggleService` methods.
- **Impact:** Changes take effect on the very next request. No restart required.

7. Deep Dive: SQL Injection

7.1 Vulnerability Mechanics

SQL Injection (SQLi) occurs when untrusted user input is concatenated directly into a database query string, allowing an attacker to manipulate the query's structure. This effectively allows the attacker to "break out" of the data context and execute commands in the code context.

The Anatomy of a Query: Consider a standard login query:

```
SELECT * FROM users WHERE username = '$username' AND password = '$password';
```

If `$username` is `admin`, the query is safe:

```
SELECT * FROM users WHERE username = 'admin' AND password = '...';
```

However, if `$username` is `admin' OR '1'='1`, the query becomes:

```
SELECT * FROM users WHERE username = 'admin' OR '1'='1' AND password = '...';
```

Because `'1'='1'` is always true, the database returns the first record (usually the admin) regardless of the password.

7.2 Types of SQL Injection

1. **In-Band SQLi (Classic):** The attacker uses the same communication channel to launch the attack and gather results.
   - *Error-Based:* Forces the DB to generate an error message revealing information (e.g., table names).
   - *Union-Based:* Uses the `UNION` operator to combine the results of the original query with the results of an injected query.
2. **Inferential SQLi (Blind):** No data is transferred, but the attacker can reconstruct data by asking true/false questions.
   - *Boolean-Based:* The application behaves differently if a query returns true vs false.

- *Time-Based:* The attacker injects a sleep command (e.g., `SLEEP(5)`). If the page takes 5 seconds to load, the condition was true.

3. **Out-of-Band SQLi:** The attacker forces the DB to make a DNS or HTTP request to a server they control (rare in modern web apps).

7.3 SecureCourse Implementation

In SecureCourse, we demonstrate **In-Band SQLi** via the Login form.

Vulnerable Code (Toggle OFF):

```
// AuthService.java
String sql = "SELECT * FROM users WHERE username = '" + username + "'";
Query query = entityManager.createNativeQuery(sql, User.class);
return query.getResultList();
```

Why it's bad: The `+ username +` concatenation happens *before* the query is sent to the database parser. The DB sees the injected SQL as valid commands.

Secure Code (Toggle ON):

```
// UserRepository.java
@Query("SELECT u FROM User u WHERE u.username = :username")
User findByUsername(@Param("username") String username);
```

Why it's safe: JPA uses `PreparedStatement` under the hood. The database treats `:username` as a strictly typed parameter. Even if the input contains SQL syntax, it is treated as a literal string value.

7.4 Advanced Exploitation Techniques

- **Bypassing Filters:** Using encoding (URL, Hex) or case variation (`SeLeCt`) to evade weak WAFs.
- **Second-Order SQLi:** Injecting malicious data that is stored safely, but then used unsafe in a *different* query later (e.g., creating a username `admin'--` that later deletes tables when an admin views the user list).
- **Database Fingerprinting:** Using version-specific functions (`@@version` , `version()`) to identify the DB backend.

7.5 Defense in Depth

1. **Primary Defense:** Parameterized Queries (Prepared Statements).
2. **Secondary Defense:** Input Validation (Allow-listing).
3. **Tertiary Defense:** Least Privilege (DB user should only have `SELECT` , `INSERT` , `UPDATE` permissions, never `DROP`).
4. **ORM Usage:** Modern ORMs (Hibernate, Entity Framework) default to secure parameterization.

8. Deep Dive: Cross-Site Scripting (XSS)

8.1 Vulnerability Mechanics

Cross-Site Scripting (XSS) allows attackers to inject malicious scripts into web pages viewed by other users. These scripts execute within the context of the victim's session, allowing the attacker to bypass the Same-Origin Policy (SOP).

The Impact:

- **Session Hijacking:** Stealing `document.cookie`.
- **Keylogging:** Capturing keystrokes.
- **Phishing:** Injecting fake login forms.
- **Defacement:** Modifying the page content.

8.2 Types of XSS

1. **Stored XSS (Persistent):** The malicious script is permanently stored on the target server (e.g., in a database, forum post, or comment field). The victim retrieves the malicious script when they view the stored data.
   - *Risk:* Critical. Affects every user who views the page.
2. **Reflected XSS (Non-Persistent):** The malicious script is reflected off the web server, such as in an error message or search result. The attack is delivered via a link.
   - *Risk:* High. Requires social engineering (tricking user to click a link).
3. **DOM-Based XSS:** The vulnerability exists in client-side code rather than server-side code. The attack payload is executed as a result of modifying the DOM "environment" in the victim's browser used by the original client-side script.

8.3 SecureCourse Implementation

We demonstrate **Stored XSS** via the Comments section.

Vulnerable Code (Toggle OFF):

```
// CommentComponent.jsx
<div dangerouslySetInnerHTML={{ __html: comment.content }} />
```

Why it's bad: React's `dangerouslySetInnerHTML` is named explicitly to warn developers. It bypasses React's default escaping mechanism and renders whatever string is passed as raw HTML. If `comment.content` contains `<script>alert(1)</script>`, it executes.

Secure Code (Toggle ON):

```
// CommentService.java (Backend)
String safeContent = Encode.forHtml(content);
comment.setContent(safeContent);
```

Why it's safe: The backend uses the OWASP Java Encoder to convert special characters into their HTML entity equivalents:

- `<` becomes `<`
- `>` becomes `>`
- `"` becomes `&quot;`
- `'` becomes `&#39;`

When the frontend renders `<script>`, the browser displays the text `<script>` but does not execute it.

8.4 Advanced Exploitation Techniques

- **Polyglots:** Strings that are valid in multiple contexts (HTML, JS, CSS) to break out of complex filters.
  - Example: `javascript://%250Aalert(1)//" onload=alert(1)//`
- **Event Handlers:** Using `onerror`, `onload`, `onmouseover` instead of `<script>` tags.
  - Example: `<img src=x onerror=alert(1)>`
- **SVG Injection:** SVGs are XML and can contain `<script>` tags.
  - Example: `<svg/onload=alert(1)>`

8.5 Defense in Depth

1. **Primary Defense:** Context-Aware Output Encoding. Encode data *right before* it is rendered, for the specific context (HTML Body, HTML Attribute, JavaScript Variable, CSS).
2. **Secondary Defense:** Content Security Policy (CSP). A HTTP header that restricts the sources from which the browser is allowed to load resources.
   - `Content-Security-Policy: default-src 'self'; script-src 'self' https://trusted.cdn.com;`
3. **Tertiary Defense:** HttpOnly Cookies. Prevents JavaScript (`document.cookie`) from reading session tokens, mitigating session hijacking even if XSS succeeds.
4. **Framework Defaults:** Modern frameworks (React, Angular, Vue) auto-escape content by default. Avoid bypass hatches like `dangerouslySetInnerHTML` or `v-html`.

9. Deep Dive: File Upload Vulnerabilities

9.1 Vulnerability Mechanics

File upload vulnerabilities occur when a server allows users to upload files without sufficiently validating their name, type, contents, or size.

The Risks:

- **Remote Code Execution (RCE):** Uploading a web shell (e.g., `shell.jsp`, `cmd.php`) that executes commands on the server.
- **Defacement:** Overwriting critical system files (e.g., `index.html`).
- **Client-Side Attacks:** Uploading malicious HTML/JS files (Stored XSS).
- **DoS:** Uploading massive files to fill disk space (Zip Bombs).

9.2 SecureCourse Implementation

We demonstrate **Unrestricted File Upload** in the Course module.

Vulnerable Code (Toggle OFF):

```
// FileService.java
Path targetLocation = this.fileStorageLocation.resolve(file.getOriginalFilename());
Files.copy(file.getInputStream(), targetLocation,
StandardCopyOption.REPLACE_EXISTING);
```

Why it's bad:

1. Trusts `getOriginalFilename()`: An attacker can send `../../../../etc/passwd` (Path Traversal).
2. No Type Check: An attacker can upload `exploit.html` or `malware.exe`.

3. Overwrites Existing: An attacker can overwrite legitimate files.

Secure Code (Toggle ON):

```
// FileService.java
// 1. Check MIME Type (Magic Bytes)
String contentType = file.getContentType();
if (!ALLOWED_MIME_TYPES.contains(contentType)) throw new SecurityException();

// 2. Check Extension
String extension = FilenameUtils.getExtension(file.getOriginalFilename());
if (!ALLOWED_EXTENSIONS.contains(extension)) throw new SecurityException();

// 3. Generate Random Name
String newFilename = UUID.randomUUID().toString() + "." + extension;
Path targetLocation = this.fileStorageLocation.resolve(newFilename);
```

9.3 Advanced Exploitation Techniques

- **MIME Type Spoofing:** Changing the `Content-Type` header to `image/jpeg` while uploading a PHP script.
- **Polyglot Files:** Creating a valid GIF image that also contains valid PHP code in its metadata comments.
- **Double Extensions:** `shell.php.jpg`. Apache might execute this if configured to handle `.php` anywhere in the name.
- **Null Byte Injection:** `shell.php%00.jpg` (older systems might truncate at the null byte).

9.4 Defense in Depth

1. **Extension Whitelisting:** Only allow specific, safe extensions (`.jpg`, `.png`, `.pdf`).
2. **Content Validation:** Verify "Magic Numbers" (file signatures) to ensure the file content matches the extension.
3. **Renaming:** Always rename uploaded files to a random UUID. Never use user-provided names.
4. **Storage Location:** Store uploads outside the web root so they cannot be executed directly via URL.
5. **Permissions:** Set file permissions to `644` (Read/Write Owner, Read Group/World, NO EXECUTE).
6. **Virus Scanning:** Scan all uploads with ClamAV or similar.

10. Deep Dive: Authentication & Session Security

10.1 Authentication Architecture

SecureCourse uses a standard username/password authentication flow backed by Spring Security.

Password Storage:

- **Algorithm:** BCrypt (Blowfish Crypt).
- **Cost Factor:** 10 (2^10 iterations).
- **Salting:** Automatic. BCrypt generates a random salt for every hash, preventing Rainbow Table attacks.

Why BCrypt? BCrypt is an adaptive hashing function. As hardware gets faster, we can increase the work factor to keep brute-forcing computationally expensive. It is superior to fast hashes like MD5 or SHA-256 which are designed for speed, not security.

10.2 Session Management

We use **Stateful Session Management** via `HttpSession`.

Lifecycle:

1. **Creation:** Upon successful login, the server creates a session object in memory and assigns a unique ID.
2. **Transmission:** The ID is sent to the client in a cookie named `JSESSIONID`.
3. **Validation:** On subsequent requests, the browser sends the cookie. The server looks up the session in memory.
4. **Destruction:** Session is destroyed on logout or after 30 minutes of inactivity.

Security Attributes:

- `HttpOnly`: **Yes.** Prevents client-side scripts from accessing the cookie (mitigates XSS).
- `Secure`: **Yes (in Prod).** Ensures cookie is only sent over HTTPS.
- `SameSite`: **Strict.** Prevents CSRF by only sending the cookie for first-party requests.

10.3 Common Vulnerabilities

1. **Session Fixation:** Attacker sets a victim's session ID before they log in.
   - *Mitigation:* Spring Security automatically rotates the session ID upon login (`migrateSession`).
2. **Credential Stuffing:** Attackers use username/password pairs leaked from other sites.
   - *Mitigation:* Rate limiting and Multi-Factor Authentication (MFA).
3. **Weak Session IDs:** IDs that are predictable (e.g., sequential numbers).
   - *Mitigation:* Use cryptographically secure random number generators (Java's `SecureRandom`), which Spring Boot does by default.

10.4 CSRF (Cross-Site Request Forgery)

CSRF forces an end user to execute unwanted actions on a web application in which they're currently authenticated.

Attack Scenario:

1. User is logged into SecureCourse.
2. User visits `evil.com`.
3. `evil.com` has a hidden form that POSTs to `securecourse.com/api/change-password`.
4. Browser automatically sends the `JSESSIONID` cookie.
5. Server processes the request as if the user intended it.

Defense:

- **Synchronizer Token Pattern:** Server sends a random CSRF token in a meta tag. Forms must include this token. The browser does *not* send this automatically like cookies.
- **SameSite Cookie Attribute:** Setting `SameSite=Strict` or `Lax` prevents the browser from sending cookies on cross-site POST requests.

11. Backend Technical Deep-Dive

11.1 Spring Boot Architecture

The backend is built as a monolithic Spring Boot application, chosen for its robustness and ease of deployment.

Layered Architecture:

1. **Controller Layer (`com.securecourse.backend.controller`):**
   - Handles HTTP requests.
   - Maps JSON payloads to DTOs (Data Transfer Objects).
   - Performs basic validation (`@Valid`).
   - Delegates business logic to Services.
2. **Service Layer (`com.securecourse.backend.service`):**
   - Contains the core business logic.
   - Manages transactions (`@Transactional`).
   - Interacts with the `SecurityToggleService` to determine which code path (secure/insecure) to execute.
3. **Repository Layer (`com.securecourse.backend.repository`):**
   - Interfaces extending `JpaRepository` .
   - Abstracts the data access logic.
   - Hibernate generates the SQL queries automatically.
4. **Entity Layer (`com.securecourse.backend.entity`):**
   - POJOs annotated with `@Entity` .
   - Maps directly to database tables.

11.2 Key Components

- **`SecurityToggleService` :** A singleton bean that holds the state of security flags. It uses `AtomicBoolean` for thread safety.
- **`GlobalExceptionHandler` :** A `@ControllerAdvice` class that catches exceptions globally. In a real secure app, this ensures that stack traces are never returned to the client.
- **`CorsConfig` :** Configures Cross-Origin Resource Sharing to allow the React frontend (on port 5173) to talk to the backend (on port 8080).

11.3 Database Interaction

We use **Hibernate ORM** for 90% of interactions.

- *Standard:* `userRepository.save(user)`
- *Vulnerable Demo:* `entityManager.createNativeQuery(...)` is used explicitly to bypass Hibernate's built-in protections for the SQL Injection demo.

---

12. Frontend Technical Deep-Dive

12.1 React Architecture

The frontend is a Single Page Application (SPA) built with React 18 and Vite.

Directory Structure:

- `components/` : Reusable UI elements (Buttons, Inputs, Cards).
- `pages/` : Full-page views (Login, Dashboard, CourseViewer).
- `context/` : Global state management using React Context API.
- `services/` : Axios instances for API communication.

12.2 State Management

We avoid Redux to keep the architecture simple. Instead, we use:

- **`AuthContext`:** Manages the user's login state (`user` object, `isAuthenticated` boolean) and provides `login()` and `logout()` methods.
- **`ToggleContext`:** Fetches the current state of security toggles from the backend on load and exposes them to components. This allows the UI to show warnings like "⚠️ SQL Injection Protection is OFF".

12.3 Safe vs. Unsafe Rendering

React is secure by default. It escapes all values embedded in JSX.

- **Safe:** `<div>{userInput}</div>` -> Renders text.
- **Unsafe:** `<div dangerouslySetInnerHTML={{ __html: userInput }} />` -> Renders HTML.
  - *Usage:* We use this *only* in the `CommentList` component to demonstrate XSS when the backend fails to sanitize input.

12.4 API Integration

We use **Axios** for HTTP requests.

- **Interceptors:** We use response interceptors to handle 401 (Unauthorized) errors globally by redirecting the user to the login page.
- **Credentials:** `axios.defaults.withCredentials = true` ensures that the `JSESSIONID` cookie is sent with every request.

13. Privacy and Data Handling

13.1 Data Inventory

- **User Data:** Username, Password Hash (BCrypt). No email or PII is currently collected.
- **User Generated Content:** Comments, Uploaded Files.
- **Metadata:** IP Addresses, User Agents (in logs).

13.2 Retention & Lifecycle

- **Session:** Ephemeral, destroyed on logout or timeout (30 min default).
- **Files:** Persisted until manual deletion.
- **Logs:** Rotated daily, retained for 30 days.

13.3 Minimization Strategy

We adhere to the principle of **Data Minimization**. We do not collect:

- Real names.
- Email addresses.
- Phone numbers.
- Location data.

---

14. Performance Considerations

14.1 Database Optimization

- **Indexing:** The `users` table has a unique index on `username` to ensure O(1) lookup time during login.
- **Connection Pooling:** HikariCP is used to manage DB connections efficiently.

14.2 Caching Strategy

- **Browser Caching:** Static assets (JS/CSS) are hashed by Vite (e.g., `index.a1b2c.js`) and served with long-term cache headers.
- **Server Caching:** (Planned) Second-level Hibernate cache for Course content that rarely changes.

14.3 Scalability

- **Statelessness:** While we currently use `HttpSession`, the architecture allows for migration to **Spring Session with Redis** to support horizontal scaling across multiple instances.
- **File Storage:** The `FileStorageService` interface abstracts the storage logic. It can be swapped for an S3 implementation for cloud scalability.

---

15. Roadmap and Future Enhancements

Phase 1: Advanced Attacks (Q1 2026)

- **CSRF Module:** Add a "Change Password" form vulnerable to CSRF.
- **SSRF Module:** Add a "Fetch URL" feature vulnerable to Server-Side Request Forgery.

Phase 2: Enterprise Features (Q2 2026)

- **RBAC:** Implement Roles (ADMIN, USER, AUDITOR).
- **Audit Dashboard:** Visual timeline of security events.

Phase 3: DevOps & Cloud (Q3 2026)

- **Docker Compose:** One-click deployment.
- **CI/CD Pipeline:** Automated SAST/DAST scanning with SonarQube and OWASP ZAP.

---

16. Best Practices Implemented

OWASP Top 10 Mapping

ID	Vulnerability	SecureCourse Coverage
A01	Broken Access Control	Authenticated Routes, IDOR checks (planned).
A02	Cryptographic Failures	BCrypt Hashing, HTTPS enforcement.
A03	Injection	SQLi Toggle, XSS Toggle.
A04	Insecure Design	Threat Modeling, Defense in Depth.
A05	Security Misconfiguration	Secure Headers, Error Handling.
A07	Identification Failures	Strong Session Management.
A09	Logging Failures	Centralized Logging.

Secure Coding Principles

1. **Validate Input:** Never trust client data.
2. **Encode Output:** Always neutralize data before rendering.
3. **Fail Securely:** Default to "Deny" if an error occurs.
4. **Keep it Simple:** Complexity hides vulnerabilities.

---

17. Glossary

- **BCrypt:** A password hashing function designed to be slow to resist brute-force attacks.
- **CSRF:** Cross-Site Request Forgery.
- **DAST:** Dynamic Application Security Testing (Black-box testing).
- **Idempotency:** The property that an operation can be applied multiple times without changing the result beyond the initial application.
- **JWT:** JSON Web Token. A stateless authentication token.
- **MIME Type:** Multipurpose Internet Mail Extensions. A standard way to classify file types.
- **ORM:** Object-Relational Mapping.
- **Prepared Statement:** A feature used to execute the same (or similar) SQL statements repeatedly with high efficiency and security against SQL injection.
- **RCE:** Remote Code Execution.
- **SAST:** Static Application Security Testing (White-box testing).
- **SQLi:** SQL Injection.
- **XSS:** Cross-Site Scripting.