**3.1 Microservice Scenario (3 marks)**  
Your microservice implements a product store with two roles—**ADMIN** and **USER**—governed by role-based access control (RBAC). The service exposes endpoints for listing products and serving images to everyone, while creation, editing, and deletion of products are restricted to ADMINs only. Users can view products and make purchases, but only ADMINs may modify product data or manage inventory.

Keeping this data secure is critical because product details, pricing information, and purchase records are sensitive business assets. An attacker who bypasses access controls could tamper with prices, delete products, or exfiltrate sales data, leading to financial loss, reputational damage, and potential regulatory liability. Moreover, customer purchase histories and any associated personal data must be protected to maintain privacy and comply with data-protection regulations such as GDPR or POPIA. Failing to secure the microservice could open the door to unauthorized data modification (OWASP A01: Broken Access Control) and account takeover (OWASP A07: Identification and Authentication Failures), both of which rank among the top web-application risks

**3.2 Security Analysis (7 marks)**  
Your frontend and backend employ multiple, layered security strategies:

* **Transport Security**: All HTTP traffic is assumed to run over HTTPS, ensuring TLS encryption of credentials, session cookies, and product data in transit, preventing eavesdropping or man-in-the-middle attacks (OWASP A02: Cryptographic Failures) [owasp.org](https://owasp.org/Top10/?utm_source=chatgpt.com).
* **Authentication**: You use Spring Security’s form-login with a custom login page, backed by BCrypt password hashing (BCryptPasswordEncoder), mitigating password-hash cracking and rainbow-table attacks [cheatsheetseries.owasp.org](https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html?utm_source=chatgpt.com).
* **Authorization**: Method-level @PreAuthorize("hasRole('ADMIN')") annotations enforce RBAC, ensuring only ADMINs can invoke create/edit/delete endpoints; all other endpoints default to authenticated or anonymous access as configured in SecurityFilterChain [owasp.org](https://owasp.org/Top10/A01_2021-Broken_Access_Control/?utm_source=chatgpt.com).
* **Session Management**: You disable CSRF protection only on safe endpoints, manage session fixation by issuing a new session on login, enforce a single concurrent session per user, and invalidate sessions on logout (HttpSessionEventPublisher), preventing session hijacking and fixation attacks [SiteLock](https://www.sitelock.com/blog/owasp-top-10-broken-authentication-session-management/?utm_source=chatgpt.com).
* **Input Validation & Error Handling**: DTOs annotated with @Valid and explicit FieldError checks prevent malformed or malicious payloads (e.g., oversized file uploads, thwarted by MaxUploadSizeExceededException handlers), reducing injection and resource-exhaustion risks [cheatsheetseries.owasp.org](https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html?utm_source=chatgpt.com).
* **Logging & Monitoring**: SLF4J logging at key junctures (login, registration, session invalidation) provides audit trails for login attempts and administrative actions, supporting anomaly detection and forensic analysis (OWASP A09: Security Logging and Monitoring Failures) [owasp.org](https://owasp.org/Top10/?utm_source=chatgpt.com).

Overall, these measures collectively enforce the principle of least privilege, safeguard credential confidentiality, and provide resilience against common web-application attacks.

**3.3 Authentication and Authorization Methods (10 marks)**

| **Method** | **Description** | **Pros** | **Cons** |
| --- | --- | --- | --- |
| **Session-based (Cookie)** | Server creates session on login, stores ID in HttpOnly cookie | Widely supported; CSRF tokens; easy revocation | Requires server-side state; vulnerable if cookies stolen |
| **JWT (JSON Web Tokens)** | Stateless tokens signed by server, stored client-side (localStorage/cookie) | Scales horizontally; no server state required | Revocation complex; token theft risk |
| **OAuth 2.0 / OpenID Connect** | Delegated authorization; issues access + refresh tokens | Third-party SSO; fine-grained scopes | Complexity; external dependencies |
| **Basic Auth / API Keys** | Credentials or keys sent with each request | Simple to implement | Credentials in every request; replay risk |

For your store microservice, **session-based authentication** with Spring Security is most suitable:

* It integrates natively with Thymeleaf views and form login.
* Supports CSRF protection by default.
* Allows server-side session invalidation on logout or timeout.
* Simplifies role checks via hasRole(...).

While JWTs scale well for stateless APIs, they complicate revocation and require careful token-blacklisting. OAuth 2.0 is ideal for third-party integrations but is overkill for a two-role internal store service. Thus, session cookies with RBAC annotations strike the right balance of simplicity, security, and control.

**3.4 Real-World Security Failures and Analysis (10 marks)**

1. **Equifax (2017)**
   * **Failure**: Attackers exploited an unpatched Apache Struts vulnerability (CVE-2017-5638), gained access to sensitive PII of 143 million consumers, and exfiltrated Social Security numbers, birthdates, and addresses [Federal Trade Commission](https://www.ftc.gov/news-events/news/press-releases/2019/07/equifax-pay-575-million-part-settlement-ftc-cfpb-states-related-2017-data-breach?utm_source=chatgpt.com)[The Guardian](https://www.theguardian.com/us-news/2017/sep/07/equifax-credit-breach-hack-social-security?utm_source=chatgpt.com).
   * **Root Causes**: Poor patch management; inadequate vulnerability scanning; lack of enforcement of least-privilege for web-app components.
   * **Prevention**:
     + Implement automated patch orchestration and integrity checks (OWASP A06: Vulnerable and Outdated Components).
     + Enforce network segmentation and minimal privileges for web servers.
     + Deploy runtime application self-protection (RASP) to detect exploitation attempts.
2. **Uber (2016)**
   * **Failure**: Hackers used stolen employee credentials (via credential stuffing) to access GitHub repos, extracted AWS keys, and downloaded data on 57 million users. Uber then concealed the breach for over a year, paying hackers to delete data [WIRED](https://www.wired.com/story/uber-exec-joe-sullivan-data-breach-indictment?utm_source=chatgpt.com)[justice.gov](https://www.justice.gov/usao-ndca/pr/uber-enters-non-prosecution-agreement?utm_source=chatgpt.com).
   * **Root Causes**:
     + Lack of multi-factor authentication (MFA) on developer accounts.
     + Insufficient monitoring of privileged-credential usage.
     + Corporate decision to hide breach rather than promptly disclose.
   * **Prevention**:
     + Enforce MFA for all administrative and developer access.
     + Monitor and alert on anomalous cloud-credential usage.
     + Establish clear incident-response and disclosure policies aligned with legal obligations.

These cases illustrate that broken authentication (OWASP A07) and broken access control (OWASP A01) can have catastrophic consequences. Robust patch management, MFA, session security, and transparent incident response are essential controls to prevent similar failures.