**Case Study: Auth for “ShopSphere” – A Retail Web Application**

**1. Business Context**

ShopSphere sells millions of products online, with:

* **End-users**: Customers browsing and buying.
* **Sellers**: Merchants listing and managing products.
* **Admins**: Internal employees managing the platform.

The system must:

1. Allow **secure login** for different user types.
2. Support **social logins** (Google, Facebook) for quick customer onboarding.
3. Provide **role-based access control** so that sellers can’t modify other sellers’ products and customers can’t see admin dashboards.
4. Handle **scalability** — millions of concurrent users.
5. Meet **compliance** requirements (e.g., GDPR, PCI-DSS).

**2. Authentication Design**

Authentication = “Are you who you claim to be?”

**2.1. Flows**

* **Customer Login**
  + Option 1: Email + password .
  + Option 2: OAuth 2.0 / OpenID Connect via Google/Facebook.
* **Seller Login**
  + Email + password + **mandatory MFA** (multi-factor authentication).
* **Admin Login**
  + Corporate SSO (Single Sign-On) with MFA, IP whitelisting.

**2.2. Architecture**

[Web/Mobile App] → [API Gateway] → [Auth Service] → [User DB / Identity Provider]

* **API Gateway**: Routes requests, enforces HTTPS, rate limits.
* **Auth Service**:
  + Issues **JWT (JSON Web Tokens)** or **opaque session tokens**.
  + Validates credentials.
  + Integrates with identity providers (Google, internal SSO).
* **User DB**:
  + Stores password hashes, MFA secrets, profile info.
  + No plaintext passwords.
  + Separate DB for audit logs.

**3. Authorization Design**

Authorization = “What are you allowed to do?”

**3.1. Roles & Permissions**

| **Role** | **Example Permissions** |
| --- | --- |
| Customer | View products, place orders, view own orders |
| Seller | Add/edit products, view own orders, manage stock |
| Admin | Manage sellers, view all orders, handle refunds |

**3.2. Models**

* **RBAC (Role-Based Access Control)**: Easy to manage predefined roles.

**3.3. Flow with JWT**

1. User logs in → Auth service issues JWT with:

{

"sub": "user-123",

"role": "seller",

"permissions": ["edit\_own\_products", "view\_orders"],

"exp": 1723545600

}

1. User sends JWT in **Authorization: Bearer <token>** header.
2. API Gateway or microservice validates token signature and exp.
3. Service checks role/permissions before executing the request.

**4. Security Best Practices**

* **HTTPS everywhere** (TLS 1.3).
* **Password policy**: Min length, complexity, breach checks (HIBP API).
* **MFA** for sensitive roles.
* **Refresh tokens** for long-lived sessions, stored securely (HttpOnly cookies).
* **Rate limiting** login attempts (prevent brute force).
* **Audit logs** for all login and authorization events.
* **Key rotation** for JWT signing keys.
* **Zero trust principle** — verify every request.

**5. Example Tech Stack**

* **Identity Provider**: Cognito / Auth0 / Okta.
* **API Gateway**: AWS/azure/gcp API Gateway / NGINX / Kong.
* **Token Format**: JWT (RS256).
* **Database**: PostgreSQL for user store, Redis for session cache.
* **Frontend**: React + Next.js with secure cookies for token storage.

**6. Diagram**

┌────────────┐

│ Browser │

└─────┬──────┘

│ HTTPS + Login Request

▼

┌───────────────┐

│ API Gateway │ ← Validates TLS, throttles

└─────┬─────────┘

▼

┌───────────────┐

│ Auth Service │ ← Verifies creds, issues JWT

└─────┬─────────┘

▼

┌───────────────┐

│ User Database │

└───────────────┘