High-level project structure (final state)

```
vpn-for-office/
├─ vpn-common/          # shared utilities: encryption, DTOs, constants
├─ vpn-server/          # Spring Boot backend (REST, auth, sessions)
├─ vpn-client/          # JavaFX desktop client
├─ vpn-admin/           # Web admin (Thymeleaf or React) — choose one
├─ infra/               # Dockerfiles, docker-compose, deployment
scripts
├─ docs/                # README, architecture diagrams, ERD, API spec,
demo script
└─ .github/             # CI (GitHub Actions)
```

Branching & Git rules (use from Day 1)

- `main` — stable working demo
- `dev` — integration branch
- Feature branches: `feature/<short-desc>` (e.g. `feature/jwt-auth`)
- Use small commits with clear messages.
- PR process: open PR from `feature/*` → target `dev` → merge after review & tests → periodic merge `dev` → `main`.

Suggested initial Git commands:

```
git init
git remote add origin <your-github-repo-url>
git checkout -b dev
```

Developer tools you’ll use (install if missing)

- JDK 17+ (or 11 if you prefer) — set `JAVA_HOME`
- Maven (or Gradle) — I'll assume Maven
- IDE: IntelliJ IDEA Community/Ultimate (recommended)
- Database: XAMPP (MySQL/MariaDB) or Docker MariaDB
- Postman / Insomnia or `curl`
- Scene Builder (for JavaFX) — optional
- Git & GitHub CLI
- Docker & docker-compose (for containerizing later)
- Node/npm (only if you pick React for admin)

Phase-by-phase, day-by-day roadmap (extremely detailed)

I’ll give 8 phases. Each phase has **daily tasks** with concrete commands, files to create, minimal code skeletons, tests, and commit messages. If you want a calendar mapping later, tell me and I’ll format dates.

Phase 1 — Project skeleton, Maven multi-module, Java sockets (learning + minimal working client/server)

Goal: Create multi-module project, confirm basic socket comms between a Java client and a plain Java server. Learn sockets & I/O.

Day 1 — Create repo & Maven multi-module

Do

Create repo and project folders:

```
mkdir vpn-for-office
cd vpn-for-office
mvn -B archetype:generate -DgroupId=com.vpnoffice
-DartifactId=vpn-parent \
  -DinteractiveMode=false
-DarchetypeArtifactId=maven-archetype-quickstart
```

- (you can instead create parent `pom.xml` manually)

Create modules:

```
mkdir vpn-common vpn-server vpn-client vpn-admin docs
```

-

Create parent `pom.xml` (top-level) with modules section:

```
<modules>
  <module>vpn-common</module>
  <module>vpn-server</module>
  <module>vpn-client</module>
  <module>vpn-admin</module>
</modules>
```

-

Files to create

- `vpn-common/pom.xml`
- `vpn-server/pom.xml`
- `vpn-client/pom.xml`

- `README.md` basic

Commit

```
git add .
git commit -m "init: project skeleton and maven multi-module setup"
```

Day 2 — vpn-common: add DTO & util skeletons

Do

- Create package `com.vpnoffice.common`

Add `Message.java` DTO for socket messages:

```
package com.vpnoffice.common;
public class Message implements Serializable {
    private String type; // e.g., HELLO, AUTH, DATA
    private String payload;
    // getters/setters + constructors
}
```

-

Add `Config.java` for constants:

```
public class Config {
    public static final int SERVER_PORT = 5000;
    public static final String SERVER_HOST = "127.0.0.1";
}
```

-

Commit

```
git commit -am "feat(common): add Message DTO and Config constants"
```

Day 3 — Plain Java TCP server

Learn

- Java `ServerSocket`, `Socket`, `InputStream`, `ObjectInputStream`

Do

Create `vpn-server/src/main/java/com/vpnoffice/server/SimpleServer.java`:

```
 public class SimpleServer {
    public static void main(String[] args) throws IOException,
ClassNotFoundException {
        ServerSocket server = new ServerSocket(Config.SERVER_PORT);
        System.out.println("Listening on " + Config.SERVER_PORT);
        while(true) {
            Socket client = server.accept();
            ObjectInputStream in = new
ObjectInputStream(client.getInputStream());
            Message msg = (Message) in.readObject();
            System.out.println("Received: " + msg.getType() + " - " +
msg.getPayload());
            client.close();
        }
    }
}
```

-

Run

```
mvn -pl vpn-server exec:java
-Dexec.mainClass="com.vpnoffice.server.SimpleServer"
```

Commit

```
 git commit -am "chore(server): add SimpleServer that prints incoming
Message DTOs"
```

Day 4 — Plain Java TCP client

Do

Create `vpn-client/src/main/java/.../SimpleClient.java`:

```
Socket socket = new Socket(Config.SERVER_HOST, Config.SERVER_PORT);
ObjectOutputStream out = new
ObjectOutputStream(socket.getOutputStream());
out.writeObject(new Message("HELLO", "Hi from client"));
out.flush();
socket.close();
```

-

Test

- Start server, run client — server prints message
  Commit
  `git commit -am "feat(client): add SimpleClient to send HELLO message to server"`

Day 5 — Improve server: threaded client handler + logging

Do

- Implement `ClientHandler` to accept multiple simultaneous clients using `ExecutorService`.
- Add basic console logging with timestamp.
  Tests
- Run 3 clients concurrently (open multiple terminals).
  Commit
  `git commit -am "feat(server): add threaded ClientHandler and executor service"`

Day 6 — Add protocol message types + simple handshake

Do

- Extend `Message` to include `sessionId`.
- Implement server handshake: on HELLO -> respond with HELLO_ACK.

- Create a small `Response` DTO or reuse `Message`.
  Commit
  `git commit -am "feat(common+server+client): add handshake protocol messages"`

Day 7 — Write small README for Phase 1 and Postman/cURL equivalents

Do

- Document test steps in `docs/phase1.md`
  Commit
  `git commit -am "docs: add Phase 1 README and test steps"`

Deliverable: Working basic multi-module project with socket handshake.

Phase 2 — Spring Boot introduction & REST basics (server becomes Spring app)

Goal: Convert `vpn-server` to Spring Boot; create `/api/test` and basic User DTO with hardcoded auth.

Day 1 — Convert vpn-server to Spring Boot

Do

- Add Spring Boot parent in `vpn-server/pom.xml` and dependencies:
  - `spring-boot-starter-web`, `spring-boot-starter-actuator`, Lombok (optional)

Create main class:

```
@SpringBootApplication
public class VpnServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(VpnServerApplication.class, args);
    }
}
```

-

Commit

```
 git commit -am "feat(server): convert to Spring Boot app skeleton"
```

Day 2 — Add test endpoint & run

Do

Create `TestController`:

```
 @RestController
@RequestMapping("/api")
public class TestController {
    @GetMapping("/test")
    public ResponseEntity<String> test() {
        return ResponseEntity.ok("Server Running");
    }
}
```

-

Run

```
mvn -pl vpn-server spring-boot:run
# Test
curl http://localhost:8080/api/test
```

Commit

```
 git commit -am "feat(server): add /api/test endpoint"
```

Day 3 — Create User DTO and service (hardcoded)

Do

- Add `User` DTO and `UserService` that stores users in memory (Map)
- Add `/api/auth/login` POST that accepts username/password and returns 200/401
  Sample Controller

```
@PostMapping("/auth/login")
public ResponseEntity<?> login(@RequestBody AuthRequest req) {
    if(userService.validate(req.username, req.password)) {
        return ResponseEntity.ok(Map.of("message","ok"));
    }
    return
ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("invalid");
}
```

Commit

```
git commit -am "feat(server): add in-memory user authentication
endpoint"
```

Day 4 — Add DTO validations and simple error handling

Do

- Use `@Valid` and `@ControllerAdvice` for error responses
  Commit
  ```
  git commit -am "chore(server): add validation and global
  exception handling"
  ```

Day 5 — Document API with OpenAPI/Swagger (optional but recommended)

Do

- Add `springdoc-openapi-ui` dependency and test
  `http://localhost:8080/swagger-ui.html`
  Commit
  ```
  git commit -am "feat(server): add OpenAPI swagger for API docs"
  ```

Day 6–7 — Unit tests for controllers (JUnit + MockMVC)

Do

- Add tests for `/api/test` and `/api/auth/login`.
  Commit

```
git commit -am "test(server): add controller unit tests"
```

Deliverable: `vpn-server` is a Spring Boot app with a working test endpoint and in-memory auth.

Phase 3 — Database (MySQL/MariaDB) & Hibernate (JPA) integration

Goal: Persist users to DB via Spring Data JPA; learn mappings and basic CRUD.

Day 1 — Setup DB locally

Do

Start XAMPP or run Docker:

```
docker run -e MYSQL_ROOT_PASSWORD=root -e MYSQL_DATABASE=vpn -p
3306:3306 -d mysql:8
```

-
- Create DB user or use root for dev.
  Commit
  ```
  git commit -am "docs: add db setup instructions"
  ```

Day 2 — Add JPA & MySQL deps, configure `application.properties`

Add deps

- `spring-boot-starter-data-jpa`, MySQL connector
  application.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/vpn
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Commit
```
git commit -am "feat(server): add JPA dependencies and datasource config"
```

Day 3 — Create `User` Entity, `UserRepository`, service layer

Entity sample

```
@Entity
@Table(name="users")
public class User {
  @Id @GeneratedValue
  private Long id;
  private String username;
  private String passwordHash;
  private String role;
  // getters, setters
}
```

Repository

```
public interface UserRepository extends JpaRepository<User, Long> {
  Optional<User> findByUsername(String username);
}
```

Commit
```
git commit -am "feat(server): add User entity and UserRepository"
```

Day 4 — Implement registration & DB-backed login

Do

- Create register API: `/api/auth/register` that stores hashed password (BCrypt).

Use `spring-boot-starter-security` later; for now simple service with BCrypt:

```
String hash = new BCryptPasswordEncoder().encode(rawPassword);
```

-

Commit
`git commit -am "feat(server): add registration endpoint and BCrypt password storing"`

Day 5 — Migration scripts (Flyway or Liquibase) — optional but recommended

Do

- Add Flyway and create `V1__init.sql` for `users` table.
  Commit
  `git commit -am "chore(server): add flyway migrations for schema"`

Day 6–7 — Integration tests with Testcontainers (advanced / optional)

Do

- Add Testcontainers for MySQL to run integration tests.
  Commit
  `git commit -am "test(server): add integration tests with Testcontainers"`

Deliverable: DB-backed user management with registration + login.

Phase 4 — Authentication (JWT) & Encryption for data channels

Goal: Secure REST APIs with JWT; encrypt client-server socket/tunnel data using AES + RSA.

Day 1 — Learn JWT basics & add dependencies

Do

- Add `jwt` or `spring-boot-starter-oauth2-resource-server` if using Spring Security.

- Create `JwtUtil` for token creation/validation.
  Commit

```
git commit -am "feat(server): add jwt util skeleton"
```

Day 2 — Integrate Spring Security with JWT

Do

- Add `SecurityConfig` to accept `/api/auth/**` publicly and protect other endpoints.
- Implement `JwtFilter` to parse `Authorization: Bearer <token>`
  Key files
- `SecurityConfig.java`
- `JwtAuthenticationFilter.java`
  Commit
  `git commit -am "feat(server): add Spring Security + JWT filter"`

Day 3 — Client: obtain JWT then use for protected REST calls

Client HTTP

- Use `HttpClient` or `OkHttp` from Java client.
- Implement login flow: send credentials → receive JWT → store locally in memory
  Commit
  `git commit -am "feat(client): add REST login and JWT storage"`

Day 4 — Encryption plan (AES + RSA hybrid)

Learn

- RSA for key exchange (asymmetric) — small data: exchange AES session key
- AES (e.g., AES-256 GCM) for message encryption
- Use `javax.crypto` with proper IVs and authenticate (GCM)

Do

- Add `CryptoUtil` in `vpn-common`:
    - Generate RSA keypair (server holds private key, client gets server public key)
    - Client generates AES key for session, encrypt AES key with server public key, send it
    - Server decrypts AES key, use it for symmetric encryption afterward
      Pseudo

```
// Client
KeyGenerator kg = KeyGenerator.getInstance("AES");
SecretKey aes = kg.generateKey();
byte[] encryptedAes = rsaEncrypt(aes.getEncoded(), serverPublicKey);
// send encryptedAes via authenticated REST or initial socket
handshake
```

Commit

```
git commit -am "feat(common): add CryptoUtil RSA/AES skeleton"
```

Day 5 — Socket channel encryption implementation

Do

- When a client opens a socket connection, first perform JWT validation (server-side) for authorization via REST OR send a token with a handshake message.
- After auth, perform AES key handshake (client -> encrypted AES -> server).
- Wrap socket streams with `CipherInputStream`/`CipherOutputStream` using AES GCM.
  Tests
- Send a small encrypted message from client; server decrypts and prints.
  Commit
  ```
  git commit -am "feat(server+client): implement AES handshake and
  encrypted socket streams"
  ```

Day 6 — Password hashing & account security hardening

Do

- Use `BCryptPasswordEncoder`, add password policies (min length), and rate-limiting for auth attempts (simple in-memory throttle).
  Commit
  `git commit -am "chore(server): password policy and auth throttling"`

Day 7 — Document encryption flow & threat model

Do

- `docs/encryption.md` — describe RSA/AES flow, key storage decisions, what's secure for demo vs production caveats.
  Commit
  `git commit -am "docs: encryption flow and security considerations"`

Deliverable: JWT-authenticated APIs and encrypted client-server socket channel.

Phase 5 — JavaFX Desktop Client (UI, connect/disconnect, session display)

Goal: Build a user-friendly JavaFX client to login, connect/disconnect, show status and logs.

Day 1 — JavaFX project setup

Do

- Add JavaFX dependencies to `vpn-client/pom.xml`. If using JDK 11+, add `org.openjfx` libs.

- Basic `MainApp.java` to show a window.
  Commit
  `git commit -am "feat(client): add JavaFX skeleton application"`

Day 2 — Login screen UI (FXML suggested)

Files

- `login.fxml` — username, password fields, login button, status label
- Controller: `LoginController.java` to call REST `/api/auth/login`
  Do
- On success: store JWT and navigate to Dashboard scene
  Commit
  ```
  git commit -am "feat(client): add login screen and REST login integration"
  ```

Day 3 — Dashboard UI: Connect / Disconnect

UI

- Buttons: Connect, Disconnect
- Labels: Connection status, Session ID
- Logs: TextArea for live logs
  Do
- On Connect:
  - open socket to server
  - send handshake + AES key encrypted with server public key
  - on success set status `Connected`
- On Disconnect: close stream/sockets and set `Disconnected`
  Commit
  ```
  git commit -am "feat(client): add dashboard with connect/disconnect and logs"
  ```

Day 4 — Background threading & UI responsiveness

Do

- Use `Task`/`Service` to run network operations off the JavaFX thread.
- Show progress indicators and handle exceptions gracefully.
  Commit
  `git commit -am "chore(client): move network ops off UI thread, add loading indicators"`

Day 5 — Logging & local history

Do

- Implement local encrypted log file using AES session key or user-specific key for storing connection history.
- Provide export button to save logs as `.txt`.
  Commit
  `git commit -am "feat(client): add local log history and export feature"`

Day 6 — Auto-reconnect/keepalive

Do

- Send periodic keepalive pings via the encrypted socket. If ping fails, attempt reconnect (with retry limits).
  Commit
  `git commit -am "feat(client): add keepalive pings and basic auto-reconnect"`

Day 7 — Packaging JavaFX app

Do

- Create a runnable jar or native installer (jpackage) for demo distribution.
  Example

```
mvn clean package
# or use jpackage to build installer (advanced)
```

Commit
```
git commit -am "chore(client): package javafx application for demo"
```

Deliverable: User can login, connect to server securely, see status and logs, and package the client app.

Phase 6 — Admin Dashboard (Thymeleaf or React) & session management

Goal: Create admin UI to manage users, view sessions and logs, force disconnect users.

Decision: If you want faster dev and Java-only stack, use **Thymeleaf**. If you prefer modern UI and richer front-end, use **React**. I'll show essential tasks that apply to either.

Day 1 — Admin requirements & API endpoints

APIs to implement

- `GET /api/admin/clients` — list active clients
- `GET /api/admin/users` — list users (paged)
- `POST /api/admin/users` — create user
- `POST /api/admin/sessions/{sessionId}/disconnect` — force disconnect
- `GET /api/admin/logs` — view server logs (last N lines)
  Commit
  ```
  git commit -am "docs: admin API contract and requirements"
  ```

Day 2 — Backend: session tracking

Do

- Implement `SessionService` that tracks connected clients in memory (ConcurrentHashMap<sessionId, SessionInfo>)
- Expose session info via `AdminController`
  Commit

```
git commit -am "feat(server): add session tracking and admin endpoints"
```

Day 3 — Admin UI skeleton

Thymeleaf approach

- New Spring MVC controllers + templates `admin/index.html`
  React approach
- `vpn-admin` React app scaffold (`create-react-app` or Vite)
- Build pages for Users, Sessions, Logs

Commit

```
git commit -am "feat(admin): add admin UI skeleton"
```

Day 4 — Implement Users page (create/remove)

Do

- Admin creates users: hit `/api/admin/users`
- Implement forms + validations on UI
  Commit
  ```
  git commit -am "feat(admin): implement user create/remove UI"
  ```

Day 5 — Sessions & Force disconnect

Do

- Show list of sessions with IP, username, connectedAt
- Implement `disconnect` action that calls server endpoint — server closes socket / invalidates session.
  Commit
  ```
  git commit -am "feat(admin): sessions list and force-disconnect action"
  ```

Day 6 — Logs viewer with tail-like behavior

Do

- Implement an endpoint that streams last N lines
- Admin UI shows tail and refresh button
  Commit
  `git commit -am "feat(admin): add logs tail viewer"`

Day 7 — Secure admin routes & RBAC

Do

- Protect admin endpoints with role check (`ROLE_ADMIN`), only JWT tokens with admin role can access.
  Commit
  `git commit -am "chore(server): secure admin endpoints with role-based access"`

Deliverable: Fully functional admin panel for users & sessions.

Phase 7 — VPN "tunnel" simulation, routing, packet handling & testing

Goal: Simulate sending files/messages through the tunnel, show routing info and packet encryption/decryption.

Day 1 — Define tunnel packet structure & routing logic

Packet DTO

```
class TunnelPacket {
    String src; String dest; String type; byte[] data; long timestamp;
}
```

Routing

- For demo: route is just server acting as gateway that forwards to internal mock service or loopback.

Commit

```
git commit -am "feat(common): add TunnelPacket DTO and routing skeleton"
```

Day 2 — Implement encrypted pass-through on server

Do

- Client sends encrypted `TunnelPacket` to server; server decrypts to read metadata but forwards the encrypted data to intended endpoint (or simulates).
  Commit
  ```
  git commit -am "feat(server): add tunnel packet processing and forward simulation"
  ```

Day 3 — Simulate internal network (mock services)

Do

- Implement mock internal services (e.g., mock HTTP service inside server) to receive forwarded packets and respond.
  Commit
  ```
  git commit -am "feat(server): add mock internal services for tunnel testing"
  ```

Day 4 — Client: file transfer through tunnel

Do

- Add UI to select file, chunk file into packets, encrypt with AES and send. Show progress.
  Commit
  ```
  git commit -am "feat(client): add file transfer through tunnel with progress"
  ```

Day 5 — Visualize routing & packet history

Do

- On admin UI or client dashboard show packet history: timestamp, size, src, dest, status (encrypted/decrypted).
  Commit
  `git commit -am "feat(ui): add packet history and routing visualization"`

Day 6 — Multi-client concurrency tests & load basic checks

Do

- Simulate 5-10 clients connecting and sending small messages. Observe server thread usage.

- Add basic metrics endpoints (or Spring Actuator) to monitor thread pools.
  Commit
  `git commit -am "test(server): add concurrency smoke tests and actuator metrics"`

Day 7 — Add unit & integration tests for tunnel logic

Do

- Write tests to ensure encryption/decryption round-trip for `TunnelPacket` and file chunking integrity.
  Commit
  `git commit -am "test(common+server+client): add encryption and tunnel unit tests"`

Deliverable: Working VPN tunnel simulation (encrypted file/message transfer) with routing visualization.

Phase 8 — Polish, CI, packaging, docs, deployment & resume-ready deliverables

Goal: Make the project presentable: CI, Docker, documentation, demo video/screenshots, final packaging.

Day 1 — Dockerize services

Do

- Add `Dockerfile` for `vpn-server` (build jar and run)
- Add `docker-compose.yml`:
  - `db` (mysql), `server`, `admin` (if separate), and optional `mock-internal`
    Sample `docker-compose.yml` snippet

```
services:
  db:
    image: mysql:8
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: vpn
  server:
    build: ./vpn-server
    ports: ["8080:8080"]
    depends_on: ["db"]
```

Commit
```
git commit -am "chore(infra): add Dockerfiles and docker-compose"
```

Day 2 — GitHub Actions CI (build & test)

Do

- Add `.github/workflows/ci.yml` to:
  - Checkout
  - Set up JDK
  - Build & run `mvn test`
  - Build Docker images (optional)
    Commit
    `git commit -am "ci: add GitHub Actions build & test`

```
workflow"
```

Day 3 — Prepare README, architecture diagrams, ERD, API docs

Do

- `docs/architecture.md` with sequence diagrams (text + ascii or mermaid)
- `docs/ERD.png` or draw in text
- Add `docs/DEMO.md` with step-by-step demo script to reproduce
  Commit
  ```
  git commit -am "docs: add architecture, ERD and demo script"
  ```

Day 4 — Create demo video/screenshots and GIFs

Do

- Record short video: login, connect, file transfer, admin disconnect
- Save assets to `docs/media/`
  Commit
  ```
  git commit -am "docs: add demo video and screenshots"
  ```

Day 5 — Final security checklist & “what’s not production-ready”

Do

- Add `docs/security-checklist.md` listing production gaps (key management, certificate usage, authentication hardening, using TLS / mTLS, secrets store)
  Commit
  ```
  git commit -am "docs: add security checklist and production
  caveats"
  ```

Day 6 — Publish to GitHub & link in resume/LinkedIn

Do

- Push `main` branch with tag v1.0
- Create release notes with demo screenshots and short summary of tech stack and responsibilities
  Commit
  ```
  git tag -a v1.0 -m "VPN-for-Office v1.0 demo"
  ```

Day 7 — Prepare interview talking points & resume bullet

Suggested resume bullet

```
Built "VPN for Office" — a secure, Java-based mini-VPN demonstrating
socket and Spring-based server, JWT auth, AES/RSA hybrid encryption,
JavaFX client, admin dashboard, Docker deployment & CI. Implemented
encrypted tunnel simulation, session management, and user
administration.
```

Commit
```
git commit -am "docs: add resume bullets and interview talking points"
```

Deliverable: Demo-ready repo with docs, packaged client, docker deployment, CI, demo video, and resume-ready assets.

Testing & Validation (continual)

- Unit tests (JUnit 5) for core utils: CryptoUtil, TokenUtil, TunnelPacket serialization.
- Controller tests: MockMVC.
- Integration tests: Testcontainers MySQL (for DB flows).
- End-to-end manual smoke tests:
  1. Register admin, login (REST) → get JWT
  2. Admin creates user

3. Client logs in -> obtains JWT, connects via socket
4. Send encrypted tunnel packets -> admin sees session and logs
5. Force disconnect via admin -> client disconnects

- Add Postman collection with all APIs exported to `docs/postman_collection.json`

Quality & coding standards

- Use Lombok sparingly (explicit code is ok for learning).
- Use `SLF4J` with `logback` for server logs.
- Handle secrets via environment variables; never commit private keys (use `.env` or Docker secrets).
- Keep `vpn-common` backward-compatible; use semantic versioning between modules.

Useful code snippets & config (copy-paste friendly)

Minimal `application.properties` (server)

```
server.port=8080
spring.datasource.url=jdbc:mysql://db:3306/vpn
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=false
jwt.secret=very-secret-key-for-dev-only
```

Simple BCrypt usage

```
PasswordEncoder encoder = new BCryptPasswordEncoder();
String hash = encoder.encode("myPassword");
boolean matches = encoder.matches("candidate", hash);
```

Minimal RSA keypair generation (for dev only)

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
kpg.initialize(2048);
KeyPair kp = kpg.generateKeyPair();
PublicKey pub = kp.getPublic();
PrivateKey priv = kp.getPrivate();
```

AES encryption (GCM) skeleton

```
Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
GCMParameterSpec spec = new GCMParameterSpec(128, iv);
cipher.init(Cipher.ENCRYPT_MODE, secretKey, spec);
byte[] cipherText = cipher.doFinal(plainText);
```

Daily habit & learning plan (how to learn while building)

- **Morning (30–60m)**: Learn concept (watch short tutorial / read docs) — e.g., JWT in Spring (30m)
- **Afternoon (2–3h)**: Implement the feature for the day (coding)
- **Evening (30–60m)**: Write tests & docs, commit changes, update README