**✅ 1. Java Collections**

Use collections to manage in-memory data structures before hitting DBs or in batch operations.

**🔹 Use Cases:**

* Cache Map<EmpId, TaskList> in Task Service for fast retrieval
* Use List, Map, Set when filtering feedbacks, leave records, etc.
* Example: Sort employees based on attendance using Collections.sort() with a custom comparator

➡️ Implement in: **Task Service**, **Attendance Service**, **Leave Service**

**✅ 2. Threads (Concurrency / Scheduling)**

**🔹 Use Cases:**

* Use @Async for sending email/notifications (Notification Service)
* Use @Scheduled in Attendance Service to:
  + Auto-mark "Absent" if no check-in before 10 AM
  + Send reminders to submit feedback

**🔹 Tools:**

* ExecutorService, @EnableAsync, @Scheduled

➡️ Implement in: **Attendance**, **Notification**, **Feedback**

**✅ 3. Authentication & Role-Based Access**

**🔹 Use JWT + Spring Security in User Service**

* Manager vs Employee roles
* Secure API endpoints with @PreAuthorize or @Secured
* Validate token across all services (via shared auth utility or gateway filter)

**🔐 JWT Token Flow:**

1. Login → Generate token in Auth Service
2. Store JWT in frontend (React: localStorage)
3. Attach JWT in all API calls
4. Gateway (or internal filters) validates JWT

➡️ Implement in: **User Service**, **API Gateway**, and use filters/interceptors in other services

**✅ 4. Exception Handling**

**🔹 Use @ControllerAdvice globally in each microservice:**

* Handle custom exceptions like UserNotFoundException, TaskAlreadyExistsException
* Send back proper status codes + messages

➡️ Implement in: **All microservices**

**✅ 5. API Layer + REST Practices**

**🔹 Use Controllers + Services + Repositories pattern**

* Separate business logic from routing
* Use DTOs for transferring data between services and frontend

➡️ Implement in: **All services**

**✅ 6. Logging**

**🔹 Use Slf4j or Log4j2**

* Log incoming requests, errors, and key events
* Optional: Add centralized logging later via ELK

➡️ Implement in: **All services**

**✅ 7. Feign Clients / Service Communication**

Once your services are stable:

* Use **Feign** clients for internal REST calls (like Task Service calling User Service to validate assignee)
* Add fallback methods with **Resilience4j** (circuit breaker)

➡️ Example:

@FeignClient(name = "user-service")

public interface UserClient {

@GetMapping("/api/users/{id}")

UserDTO getUserById(@PathVariable Long id);

}

**✅ 8. Dockerization**

* Each microservice will have its own Dockerfile
* Use **Docker Compose** to run all services locally
* Start with local DB per service, then try central DB cluster or separate DB per service

**✅ React Integration**

* JWT handling (store and attach to requests)
* Conditional rendering based on user roles
* Async calls to all microservice APIs
* Add loading indicators, form validations, and Toast notifications for UX

**📚 Learning Checklist per Topic**

| **Spring Topic** | **Microservice to Practice** |
| --- | --- |
| Collections | Task, Leave, Attendance |
| Threads / Schedulers | Feedback, Notification, Attendance |
| JWT Auth | User Service + Gateway |
| Role Access | All APIs via Security |
| Feign Clients | Cross-service calls (e.g. Task ↔ User) |
| API Docs (Swagger) | All |
| Exception Handling | All |
| Logging | All |