Case Study 1: Java-Based Configuration

Project Title: Online Food Ordering System

Configuration Type: Java-based Spring Configuration

POJO Classes: `Restaurant` and `Customer`

Scenario:

An online food ordering platform allows customers to order food from various restaurants. The system must manage customer information and restaurant offerings. The logic for selecting restaurants and placing orders is handled in a service class. Java-based configuration is used to wire beans explicitly.

Components:

- **Customer.java**: Holds customer details like name, contact info, and preferred cuisine.
- **Restaurant.java**: Holds restaurant details like name, location, and available cuisines.
- **FoodOrderService.java**: Service that processes the food order by matching customer preferences with restaurant availability.
- **AppConfig.java**: A `@Configuration` class that defines and wires all beans manually using `@Bean` methods.
- **MainApp.java**: Initializes the Spring context using `AnnotationConfigApplicationContext` and executes the order flow.

Why Java-Based Config?

- Useful when full control over bean creation is required.
- Suitable for projects where configuration is centralized and separated from the POJO classes (which may not be editable).

Case Study 2: Annotation-Based Configuration

Project Title: Smart Home Automation System

Configuration Type: Annotation-based Spring Configuration

POJO Classes: `Device` and `User`

Scenario:

A smart home system manages various IoT devices like lights, fans, and ACs. Users can control these devices through an application. Each user can register and manage multiple devices. Spring annotations like `@Component`, `@Autowired`, and `@Service` are used to auto-wire dependencies and manage components.

Components:

- **User.java**: Annotated with `@Component`, contains user details like name and home ID.
- **Device.java**: Annotated with `@Component`, represents smart devices with attributes like device type and status.
- **AutomationService.java**: Annotated with `@Service`, uses `@Autowired` to inject both User and Device beans to manage device control logic.
- **AppConfig.java**: A minimal `@Configuration` class with `@ComponentScan` to auto-detect components in the package.
- **MainApp.java**: Loads the context and triggers methods to control devices.

Why Annotation-Based Config?

- Reduces boilerplate and simplifies bean wiring.
- Ideal for component-based development where classes are self-contained and annotated.
- Encourages cleaner separation of concerns with automatic scanning and DI.