Implementing Dependency Injection with Spring

Spring Feature Description

Dependency Injection (DI) is a design pattern where an object receives its dependencies from an external source rather than creating them itself. In Spring, DI is facilitated by the Inversion of Control (IoC) container, which manages the lifecycle and configuration of application components (beans). This promotes loose coupling and easier testing.

Why Do We Need It in Flight Application?

In the current Flight Application, object dependencies are manually managed, leading to tightly coupled code and difficulties in testing and maintenance. By using Spring's DI, we can decouple components, making the application more modular, testable, and easier to manage.

Refactoring Application

We will refactor the application to leverage Spring's IoC container for managing dependencies. This involves:

- Defining beans for our components.
- Configuring Spring to manage these beans.
- Injecting dependencies using annotations.

The Updated Code

- **Annotate Classes with @Service and @Component**

Add the @Service annotation to DataService:

```
import org.springframework.stereotype.Service;

@Service
public class DataService {
    // Existing code...
}
```

Add the @Component annotation to CSVDataLoader and inject DataService using constructor injection with @Autowired:

```
import org.springframework.stereotype.Component;
import org.springframework.beans.factory.annotation.Autowired;
```

```
@Component
public class CSVDataLoader {
    private final DataService dataService;

    @Autowired
    public CSVDataLoader(DataService dataService) {
        this.dataService = dataService;
    }

    // Existing code...
}
```

Note that @Autowired is optional for constructor injection in Spring 4.3+.

- **Update Main Application to Use Spring Context**

Modify FlightsApplication to initialize the Spring ApplicationContext and retrieve beans:

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

@ComponentScan(basePackages = "com.luxoft.flights")
public class FlightsApplication {

    public static void main(String[] args) throws Exception {
        ApplicationContext context = SpringApplication.run(FlightsApplication.class,
args);

        CSVDataLoader dataLoader = context.getBean(CSVDataLoader.class);
        dataLoader.loadData();
        System.out.println("Data loaded successfully.");

        // Extract dataService bean and use it

        // Existing code...
    }
}
```

Benefits of Using Dependency Injection

- **Loose Coupling**: Components are less dependent on each other.
- **Easier Testing**: Dependencies can be easily mocked or stubbed.
- **Better Maintainability**: Changes in one component have minimal impact on others.
- **Configuration Flexibility**: Easily switch between implementations.

Further Advices

Consider using constructor injection for mandatory dependencies and setter injection for optional ones. Avoid field injection to keep code testable and maintainable.