

ILP / 4

Cristina Adriana Alexandru, PhD

Last lecture

- Spring Boot:
 - Spring annotations (Lombok, others)
 - Dependency injection (DI) and inversion of control (IOC)
 - @Configuration and @Bean with DI

This lecture

- Architecture, architectural design, issue of complexity, separation of concerns
 - Recap from SEPP
- Spring Boot web service architecture
- Testing in Spring Boot
 - Unit testing
 - Integration testing

Architecture: some definitions

*“An **architecture** is the fundamental organisation of a software system embodied in its **components**, their relationships to each other and to the environment, and the principles guiding its design and evolution” (IEEE)*

*A **component** is a “named software unit that offers one or more services to other software components or to end-users of the software”. In “can be anything from a program (large scale) to an object (small scale)”.*
(Sommerville)

*A **service** is a “coherent unit of functionality” (Sommerville)*

Architectural design

- Involves creating a description of the architecture showing components and their relationships.
- One of the main questions: ***How should the system be decomposed into a set of components?***
 - Approach: Identifying large-scale components, then analysing and splitting them up into smaller components

Complexity, separation of concerns

- Major concern: **complexity** due to the number of components and their relationships, the latter increasing exponentially.
 - The more complex a system is, the less maintainable, harder to understand, error prone, less secure.
- One solution:
 - **Separation of concerns**: components doing only one thing; grouping components with related functionality.

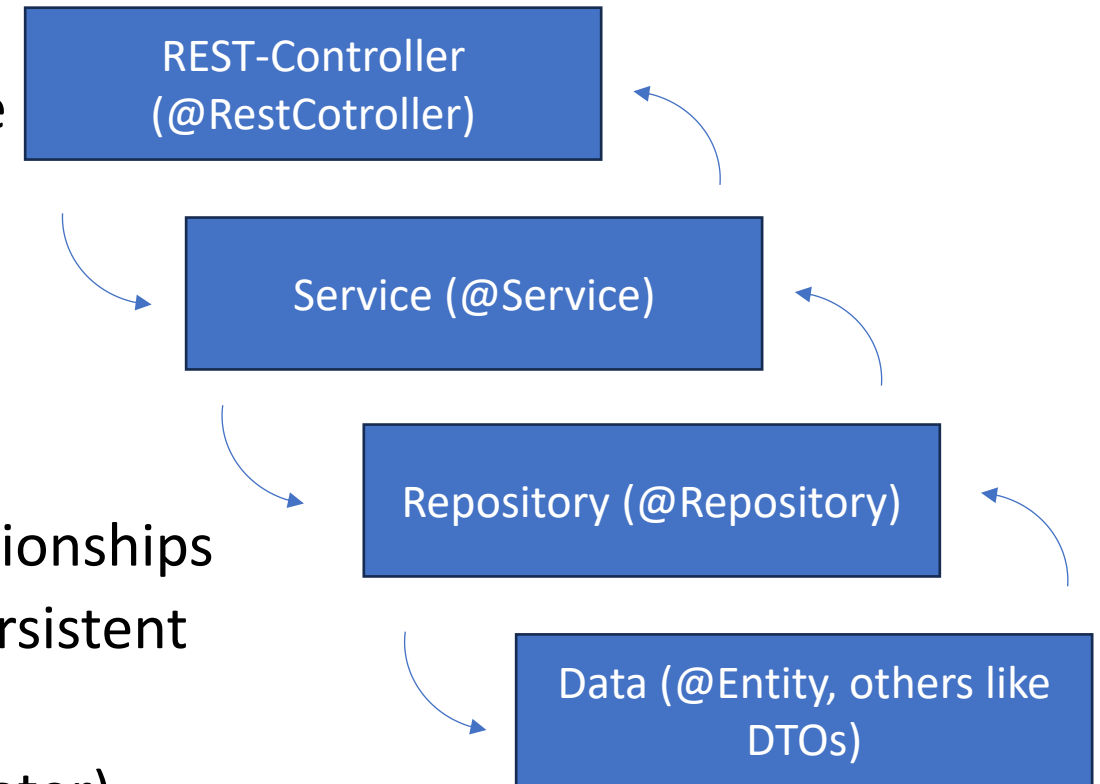
Spring Boot web service architecture

- Main components:

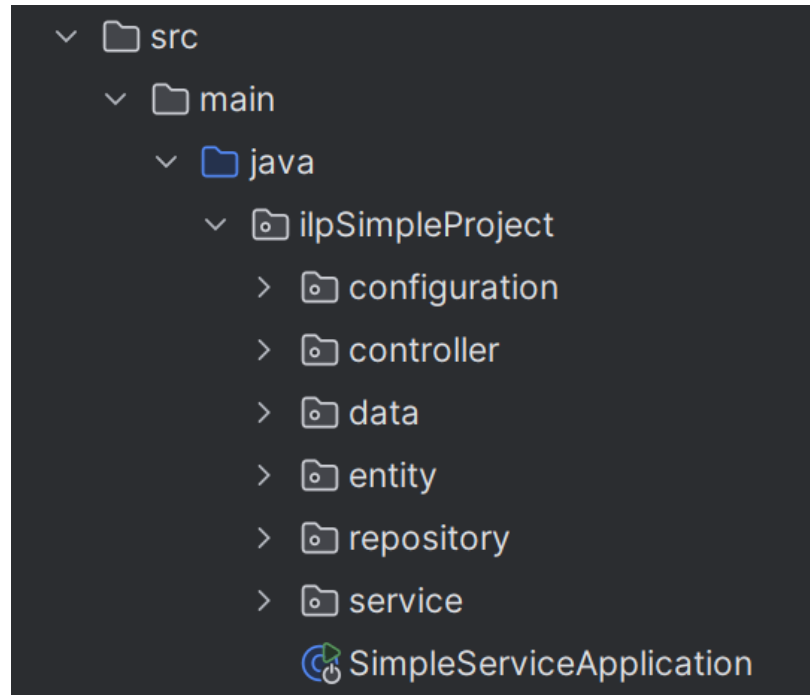
- REST-Controllers: handle REST interface
- Services: handle business logic
- Repositories: data access and storage, typically interacting with a database

- Data:

- @Entity define the properties and relationships of the application domain, mapped to persistent storage (e.g. database tables)
- DTOs: data transfer objects (see more later)



Spring Boot folder structure



- Folder components:
 - Configuration: configuration classes (great for DI, IOC)
 - Controller: REST-controller components
 - Data: DTO objects (see more later)
 - Entity: entities (see above)
 - Repository: repository components
 - Service: service components

Read more here: <https://malshani-wijekoon.medium.com/spring-boot-folder-structure-best-practices-18ef78a81819>

DTO (Data Transfer Object)

- DTO is a design pattern created by Martin Fowler (<https://martinfowler.com/eaCatalog/dataTransferObject.html>)
- Encapsulates relevant data to be transferred between software application subsystems or layers; in Spring Boot, often from controller-service, or service-repository.
- With DTOs, we can build different views from our domain models (entities), optimized to the clients' needs, without affecting domain design
- See example here: <https://www.baeldung.com/java-dto-pattern>

DTO (Data Transfer Object)

- DTOs are usually **POJOs** (plain old Java objects): flat data structures without a particular structure, naming conventions, business logic
- They can also be **Java beans**: all attributes private, getters/setters in getX/setX format, default constructor, implementing *Serializable*
- If Java beans, good to use the **MapStruct mapper** to convert entities to DTOs to vice versa: <https://www.baeldung.com/mapstruct>

DTO (Data Transfer Object)

- Advantages of DTOs:
 - Reduced network calls, because you can bundle relevant data and reuse DTO
 - Enhanced maintainability due to separation of concerns
 - DTOs can be versioned independently; backward compatibility
 - Reduced data transfer overheads because can transfer only necessary data

Testing in Spring Boot

- Unit testing: testing a single component
 - Testing services in isolation
 - Testing controllers in isolation
- Integration testing: testing that components interact properly
 - Testing how several components like controllers, services, and repositories work together
- Spring Boot already has all you need (in spring-boot-starter-test dependency):
 - JUnit
 - Mockito (for mocking)

Unit testing services in Spring Boot

- Does not normally require any special Spring Boot annotations
- Main components of a JUnit 5 test class for a component:
 - Test methods:
 - Annotated with `@Test` (or `@RepeatedTest(<Number>)` to repeat test)
 - Other possible annotations: `@DisplayName("<Name>")`, `@Disabled("reason")`, `@Tag("<TagName>")`
 - Including assert method(s): typically `assertThat` or `assertEquals` (see others: <https://junit.org/junit4/javadoc/4.8/org/junit/Assert.html>)
 - Optionally, methods to execute before the tests: `@BeforeEach`, `@BeforeAll`
 - Optionally, methods to execute after the tests: `@AfterEach`, `@AfterAll`

Unit testing in Spring Boot- Mocking

- Mocking allows replacing a dependency with a deterministic mock so that the component can be tested in isolation
- Mockito mocking instructions:
 - mock method or @Mock annotation to create a mock object
 - If using @Mock, you need to call MockitoAnnotations.openMocks(this) (former initMocks(this, now deprecated) in @BeforeEach method.
 - when(...).thenReturn(...) or when(...).thenAnswer(...) to set mock object behaviour
 - @InjectMocks to create mock and inject mocks annotated with @Mock into it
 - spy method or @Spy annotation to mock only specific behaviours

See more:

<https://www.digitalocean.com/community/tutorials/mockito-mock-examples>

Integration testing in Spring Boot

- Test and other methods, assert methods, mocking like in unit testing
- Require an application context
- `@SpringBootTest` annotation on test class bootstraps an entire application context automatically when running tests
 - `webEnvironment` attribute can take `MOCK` (to work with mock web environment, default), `RANDOM_PORT`, `DEFINED_PORT` (running server)
- `MockMvc`/ `TestRestTemplate` object to send HTTP requests and check results in mock environment/ running server
 - Require injecting with `@Autowired`
 - `@AutoConfigureMockMvc` to automatically configure `MockMvc`
- `@Autowired` to inject dependencies if application context can create them on its own

Unit testing controllers in Spring Boot

- `@WebMvcTest` annotation on test class to bootstrap only controllers (or specific one provided as argument) in a mock application context
- `@MockBean` to mock dependencies *and inject them in application context, replacing any existing bean of the same type.*
- `@Mock` mocks dependencies only!

Testing in Spring Boot

- Much more here: <https://medium.com/@bubu.tripathy/testing-spring-boot-applications-c5d8212f6e72>