Spring Fundamentals
Spring Boot Introduction

Table of Content

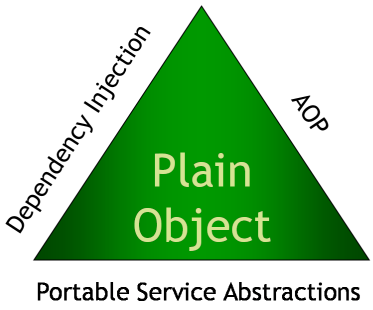
What is Spring Boot?

Spring

- In October 2002, Rod Johnson wrote a book titled "Expert One-onOne J2EE Design and Development".
- The book was accompanied by 30,000 lines of framework code also known as Interface21 (Spring 0.9).
- Since java Interfaces were the basic building blocks of dependency injection (DI), he named the root package of the classes com.interface21.
- Shortly after the release of the book, developers Juergen Hoeller and Yann Caroff persuaded Rod Johnson to create an open source project based on the infrastructure code. In March 2004, spring 1.0 was released.

Spring Main Concepts

- The four key concepts are:
  - Plain CLR objects (CLR=Common Language Runtime object is the same as POCO=Plain Old Class Object)
  - Dependency Injection (DI)
  - AOP (Aspect Oriented Programming)
  - Portable Service Abstractions



Inversion of Control (IoC)

- Spring provides **Inversion of Control** and **Dependency Injection**

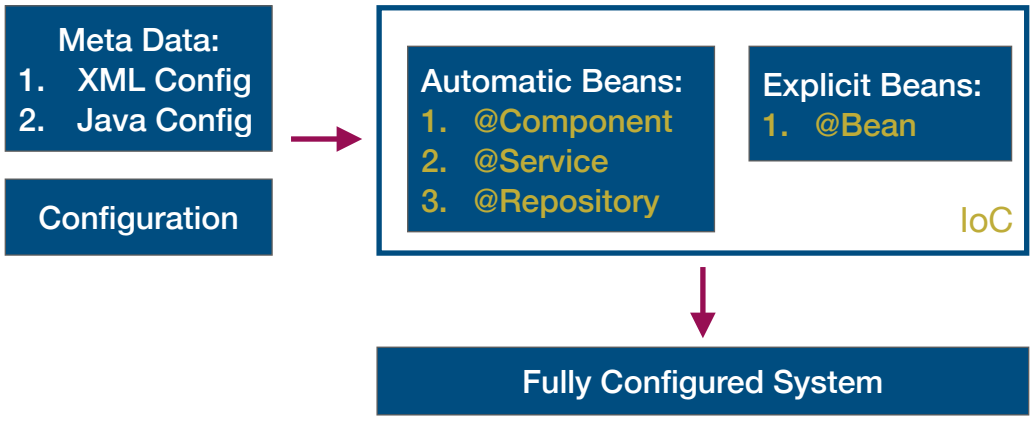
UserServiceImpl.java

```
// Traditional Way
public class UserServiceImpl implements
        UserService {
    private UserRepository userRepository =
        new UserRepositoryImpl( );
}
```

UserServiceImpl.java

```
// Dependency Injection
@Service
public class UserServiceImpl implements
        UserService {
    @Autowired
    private UserRepository userRepository;
}
```

Spring IoC



Beans

- Objects that is instantiated, assembled, and otherwise managed by a Spring IoC container

Dog.java

```
public class Dog implements Animal {
    private String name;
    public Dog( ) { }
    // Getters and Setters
}
```

Bean Declaration

MainApplication.java

```
@SpringBootApplication
public class MainApplication {
    ...
    @Bean
    public Animal getDog( ) {
        return new Dog( );
    }
}
```

Bean Declaration

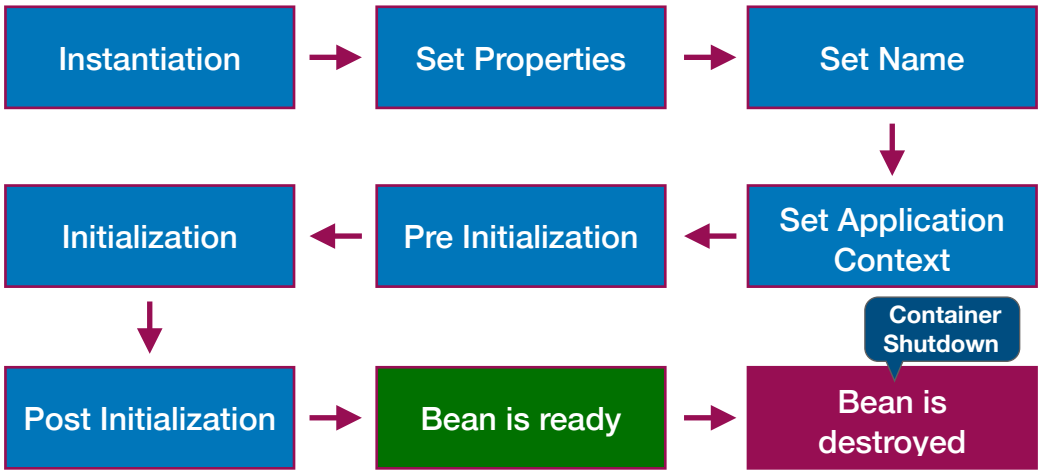
Get Bean from Application Context

MainApplication.java

```
@SpringBootApplication
public class MainApplication {
    public static void main(String[ ] args) {
        ApplicationContext context = SpringApplication.run(MainApplication.class, args);
        Animal dog = context.getBean(Dog.class);
        System.out.println("DOG: " + dog.getClass( ).getSimpleName( ));
    }
}
```

```
2017-03-05 12:59:19.389  INFO
2017-03-05 12:59:19.469  INFO
2017-03-05 12:59:19.473  INFO
DOG: Dog
```

Bean Lifecycle



Bean Lifecycle Demo

MainApplication.java

```
@SpringBootApplication
public class MainApplication {
    public static void main(String[ ] args) {
        ConfigurableApplicationContext context =
            SpringApplication.run(MainApplication.class, args);
       context.close( );
    }

    @Bean(destroyMethod = "destroy", initMethod = "init")
    public Animal getDog( ) {
        return new Dog( );
    }
}

class Dog implements Animal {
    public Dog( ) {
        System.out.println("Instantiation...");
    }

    public void init( ) {
        System.out.println("Initializing...");
    }

    public void destroy( ) {
        System.out.println("Destroying....");
    }
}

interface Animal {

    void init( );

    void destroy( );
}
```

```
2023-10-15T11:49:17.898+03:00  INFO
2023-10-15T11:49:17.901+03:00  INFO
Instantiation...
Initializing...
2023-10-15T11:49:18.181+03:00  INFO
Destroying...
```

PostConstruct Annotation

- Spring calls methods annotated with **@PostConstruct** only once, just after the initialization of bean

```
@Component
public class DbInit {
    private final UserRepository userRepository;
    public DbUnit(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @PostConstruct
    private void postConstruct( ) {
        User admin = new User("admin", "admin password");
        User normalUser = new User("user", "user password");
        userRepository.save(admin, normalUser);
    }
}
```

PreDestroy Annotation

- A method annotated with **@PreDestroy** runs only once, just before Spring removes our bean from the application context

```
@Component
public class UserRepository {
    private DbConnection dbConnection;
    @PreDestroy
    public void preDestroy( ) {
        dbConnection.close( );
    }
}
```

BeanNameAware Interface

- BeanNameAware makes the object aware of the bean name defined in the container

```
public class MyBeanName implements BeanNameAware {
    @Override
    public void setBeanName(String beanName) {
        System.out.println(beanName);
    }
}
```

```
@Configuration
public class Config {
    @Bean (name = "myCustomBeanName")
    public MyBeanName getMyBeanName( ) {
        return new MyBeanName( );
    }
}
```

BeanFactorAware Interface

- **BeanFactorAware** is used to inject the BeanFactory object
- With the setBeanFactory( ) method, we assign the BeanFactory reference from the IoC container to the beanFactory property

```
public class MyBeanFactory implements BeanFactoryAware {
    private BeanFactory beanFactory;
    @Override
    public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
    public void getMyBeanName( ) {
        MyBeanName myBeanName = beanFactory.getBean(MyBeanName.class);
        System.out.println(beanFactory.isSingleton("myCustomBeanName"));
}
```

InitializingBean Interface

- For bean implemented **InitializingBean**, it will run afterPropertiesSet( ) after all bean properties have been set

```
@Component
public class InitializingBeanExampleBean implements InitializngBean {
    private static final Logger LOG = Logger.getLogger(InitializingBeanExamleBean.class);

    @Autowired
    private Environment environment;

    @Override
    public void afterPropertiesSet( ) throws Exception {
        LOG.info(Arrays.asList(environment.getDefaultProperties( )));
    }
}
```

DisposableBean Interface

- For bean implemented DisposableBean, it will run destroy( ) after Spring container released the bean

```
@Component
public class Bean2 implements DisposableBean {
    @Override
    public void destroy( ) throws Exception {
        System.out.println("Callback triggered - DisposableBean.");
    };
}
```

Bean Scopes in Spring Framework

- The Bean scopes supported out of the box are listed below:
  - singleton (default)
  - prototype
  - request
  - session
  - application
  - web socket

Singleton Scope

- Container creates a **single instance** of that bean, and all requests for that bean name will return the **same object**, which is cached
- This is default scope

```
@Bean
@Scope("singleton") // <- can be omitted
public Student student( ) {
    return new Student( );
}
```

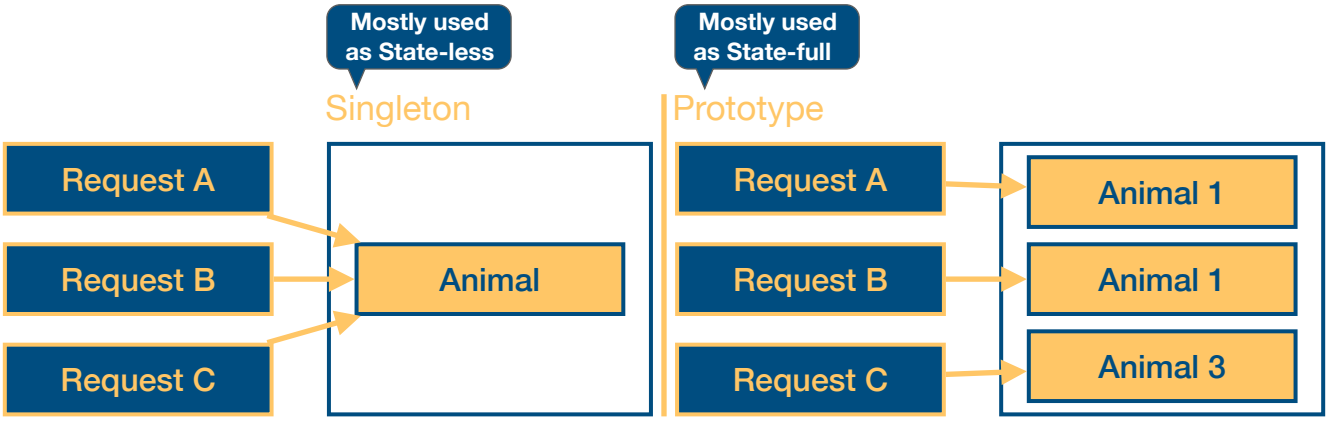
Prototype Scope

- Will return a different instance every time it is requested from the container

```
@Bean
@Scope("prototype")
public Student student( ) {
    return new Student( );
}
```

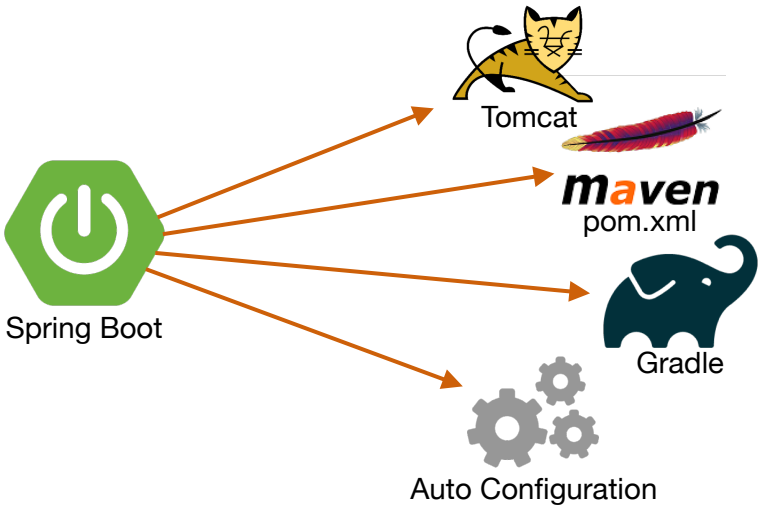
Bean Scope

- The default one is Singleton. It is easy to change to Prototype



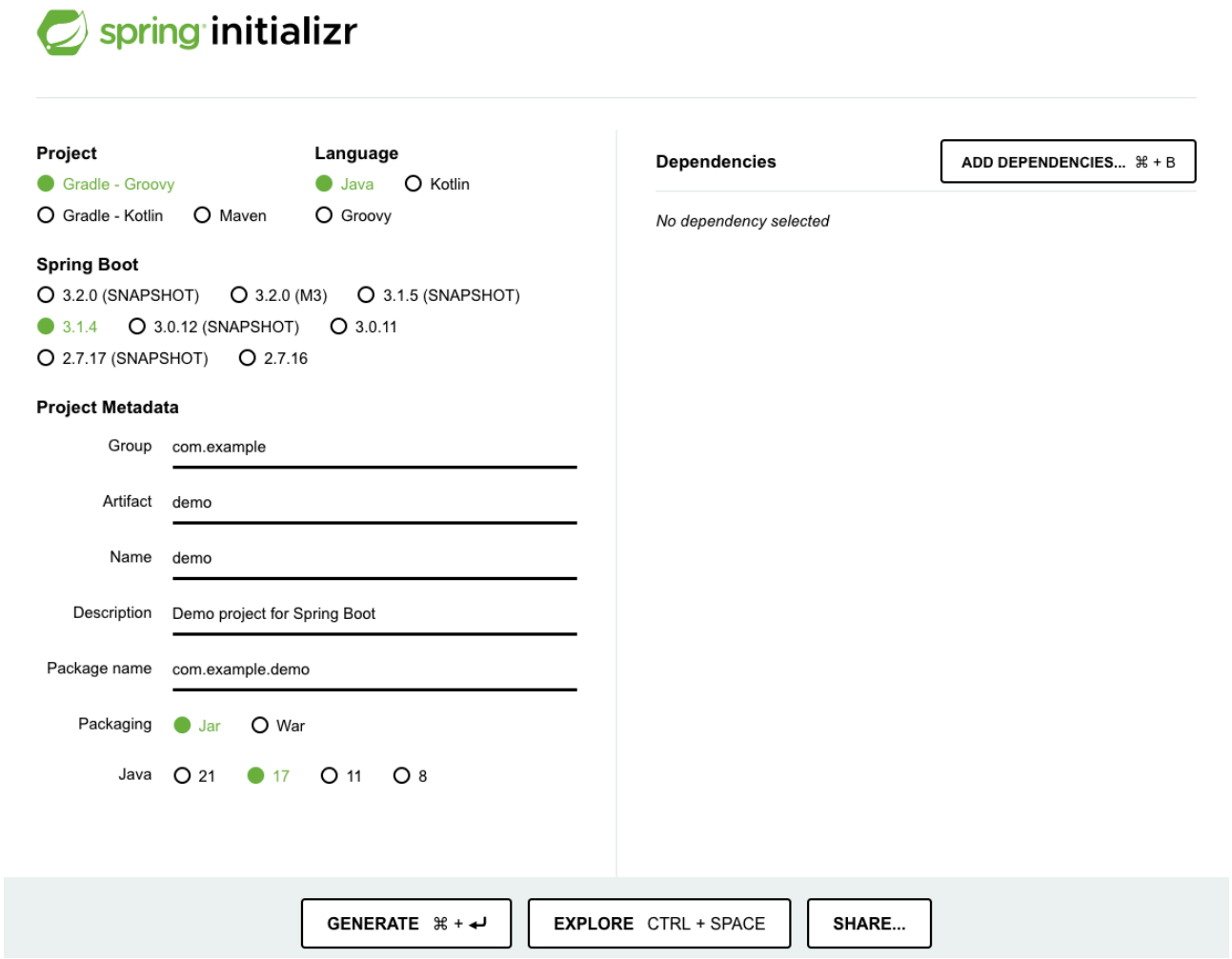
Spring Boot

- Opinionated view of building production-ready Spring applications



Creating Spring Boot Project

- Just go to Spring Initializr (https://start.spring.io/)



Spring Dev Tools

- Additional set of tools that can make the application development faster and more enjoyable
- In **Maven**:

pom.xml

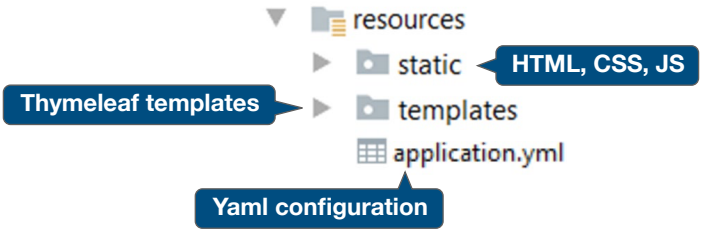
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
</dependency>
```

- In **Gradle**:

build.gradle

```
dependencies {
    compileOnly("org.springframework.boot:spring-boot-devtools")
}
```

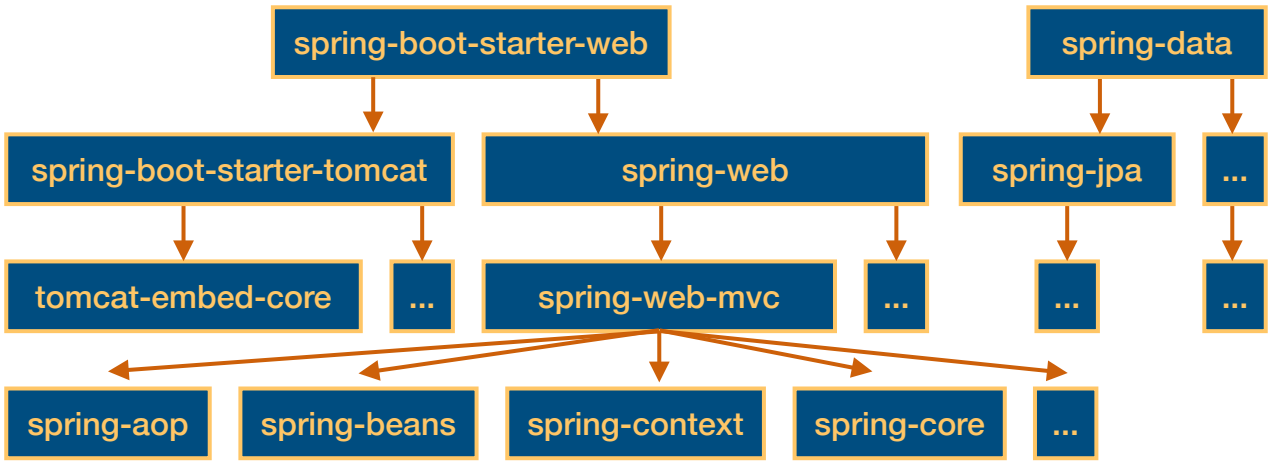
Spring Resources



Spring Boot Main Components

- Some main components:
  - Spring Boot Starters – combine a group of common or related dependencies int single dependency
  - Spring Boot Auto-Configuration – reduce the Spring Configuration
  - Spring Boot Actuator – provides EndPoints and Metrics
  - Spring Data – unify and ease the access to different kinds of database systems

Spring Boot Starters



Spring Boot Actuator

- Expose different types of information about the running application

build.gradle

```
dependencies {
   compileOnly("org.springframework.boot:spring-boot-starter-actuator")
}
```


```
{"status":"UP","diskSpace":
{"status":"UP","total":160571584512,"free":38033534976,"threshold":10485760},"db":
{"status":"UP","database":"MySQL","hello":1}}
```

Common Application Properties

- Various properties can be specified inside your **application.yaml** file
- Property contributions can come from **additional har files**
- You can define your **own properties**
- Link to documentation

Application Properties Example

application.properties

```
spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/my_db?createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=topsecret
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.open-in-view=false
logging.level.org=WARN
logging.level.blog=WARN
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor=TRACE
server.port=8000
```

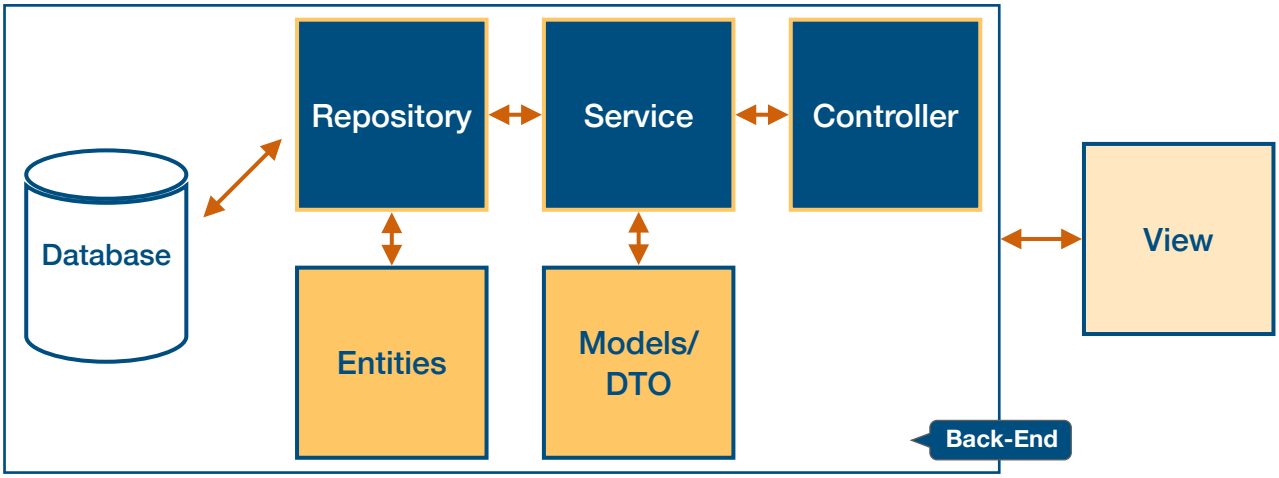
Application Yaml Example

application.yaml

```
spring:
  datasource:
    driverClassName: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/my_db?createDatabaseIfNotExist=true
    username: root
    password: topsecret
  jpa:
    properties:
      hibernate:
        dialect: org.hibernate.dialect.MySQL8Dialect
        format_sql: true
        ddl-auto: update
    open-in-view: false
logging:
  level:
    org: WARN
    blog: WARN
    hibernate:
      SQL: DEBUG
      type:
        descriptor: TRACE
server:
  port: 8000
```

Spring Data

Overall Architecture



Entities

- Entity is a lightweight **persistence domain object**

Cat.java

```
@Entity
@Table(name = "cats")
public class Cat {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    private String name;
    // Getters and Setters
}
```

Repositories

- **Persistence** layer that works with entities

CatRepository.java

```
@Repository
public interface CatRepository extends JpaRepository<Cat, Long> {
}
```

Services

- **Business Layer** – All the business logic is here

CatService.java

```
@Service
public interface CatServiceImpl implements CatService {
    private final CatRepository catRepository;

    @Autowired
    public CatServiceImpl(CatRepository catRepository) {
        this.catRepository = catRepository
    }

    @Override
    public void petCat(CatModel catModel) {
        // TODO Implement the method
    }
}
```

Summary

- Spring Boot – Opinionated view of building production-ready Spring applications
- Spring Data – Responsible for database related operations