TECH SHARING

Intro to Spring Framework and Spring Boot

Oct 2021

by: Joseph Gan

DISCLAIMER

- I may be wrong
- Correct me I'm wrong

Ask your questions, there are no stupid question!

AGENDA

- Intro to Spring Framework
- Intro to Spring Boot
- Common Annotation
- Questions and Answer?
- Best Practices
- What's next?

TAKEAWAY

- Know what Spring Framework and Boot provides
- Better understanding on how Spring works

INTRO TO SPRING FRAMEWORK

Spring Framework does all the hard work behind the scene, leaving you to focus on the business logic

Provides abstraction so it is easier to switch vendor with minimum effort

Provides familiar and consistent programming model

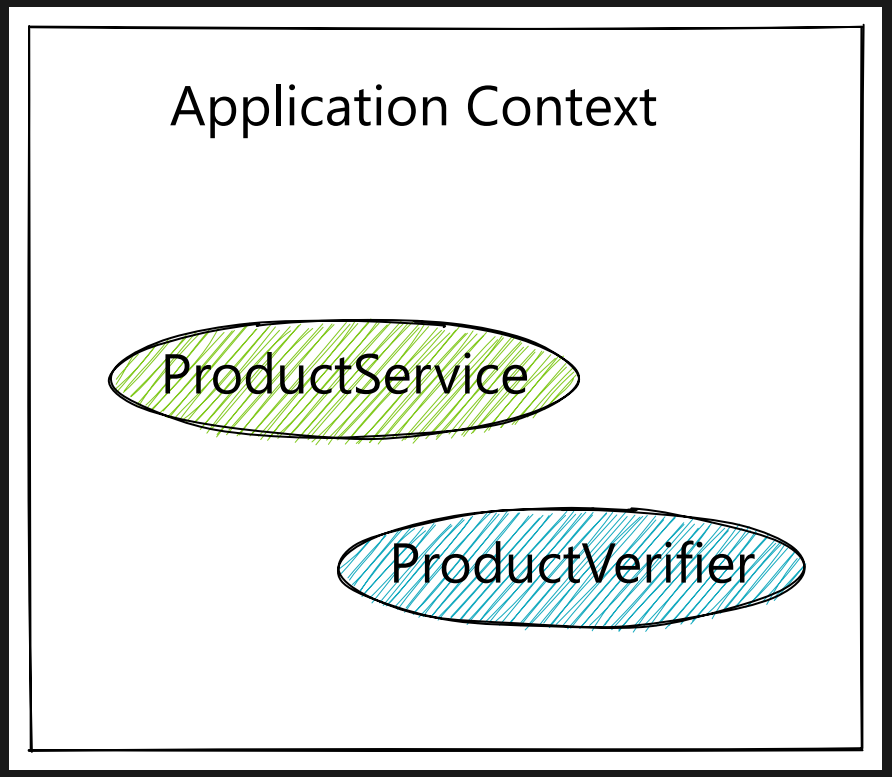
INVERSION OF CONTROL (IOC) CONTAINER

```
public class TechsharingApplication {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext appContext = new An
        var ps = appContext.getBean("profileService", ProfileS
        System.out.println(ps.getAllProfilePost(10));
    }
}
```

```
23:19:29.433 [main] DEBUG org.springframework.beans.factory.su
23:19:29.455 [main] DEBUG org.springframework.beans.factory.su
23:19:29.464 [main] DEBUG org.springframework.beans.factory.su
9
```

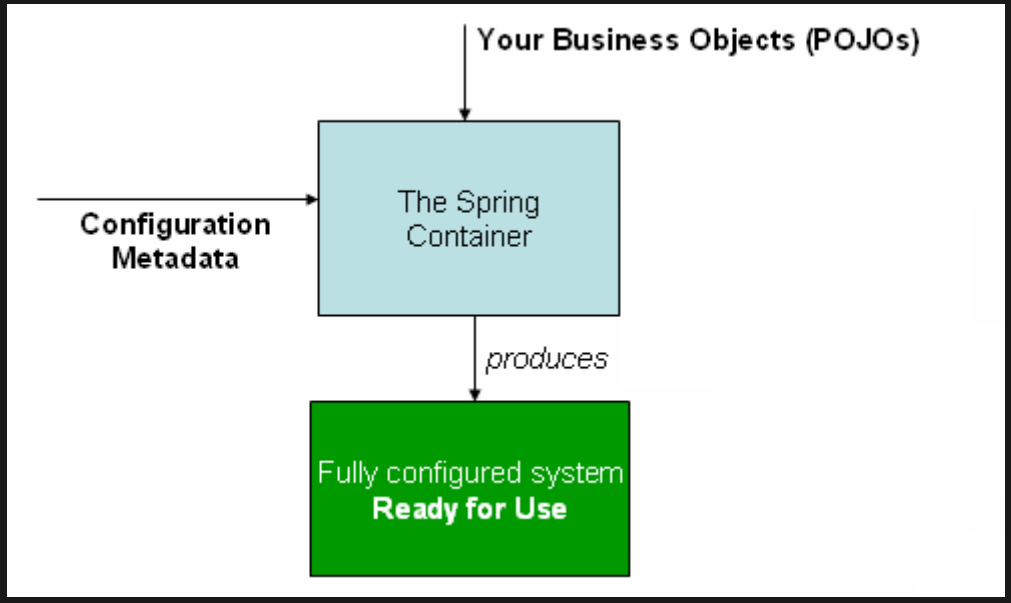
IoC means you don't control the dependencies, let someone else (framework) control/manage it

How does it do that?


Application Context
ProductService
ProductVerifier

It is a collection of Beans (classes), instantiated, assembled and managed by Spring IoC Container

```
public class PricingService {
    private final ProductVerifier productVerifier;

    public PricingService(ProductVerifier productVerifier) {
        this.productVerifier = productVerifier;
    }

    public BigDecimal calculatePrice(String productName) {
        if (productVerifier.isCurrentlyInStockOfCompetitor(pro
        return new BigDecimal("99.99");
    }

        return new BigDecimal("149.99");
    }
}
```



Different ways to configure metadata

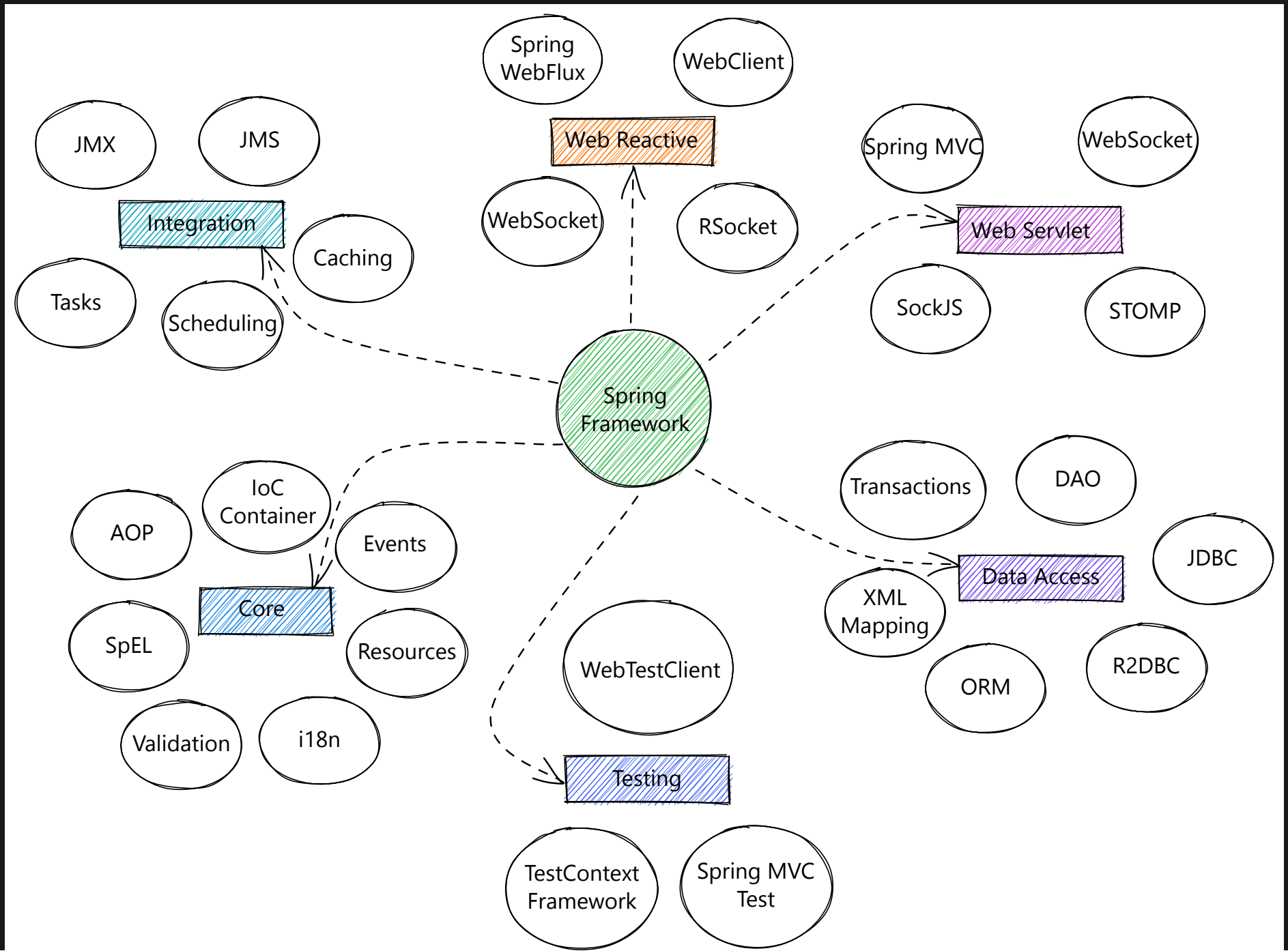
Java Annotation OR XML

@Primary, @Qualifier, @Order

DEPENDENCY INJECTION (DI)

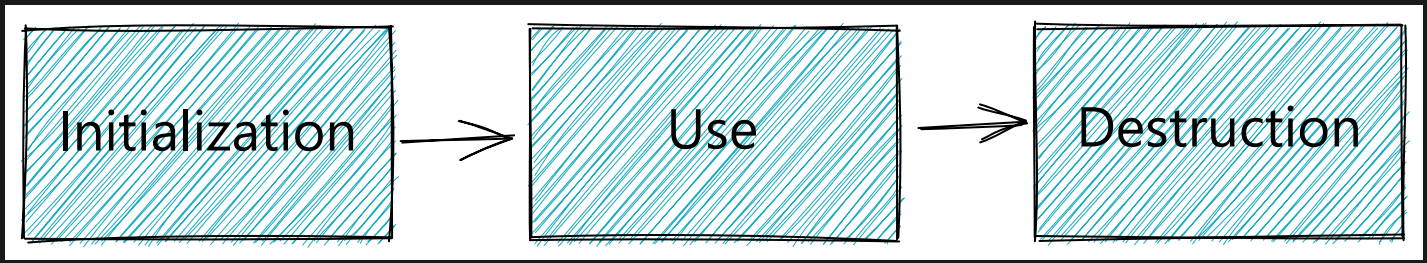
```java
// PricingService only indicates it needs a certain Bean
// But has no control over where it comes from
// Spring as the IoC Container will inject automatically
@Service
public class PricingService {
    private final ProductVerifier productVerifier;

    // via Constructor Injection (recommended)
    public PricingService(ProductVerifier productVerifier) {
        this.productVerifier = productVerifier;
    }

    // or setter injection (ok for optional dependency)
    public void setProductVerifier(ProductVerifier productVeri
        this.productVerifier = productVerifier;
```

DI means you request the object (bean) from IoC container

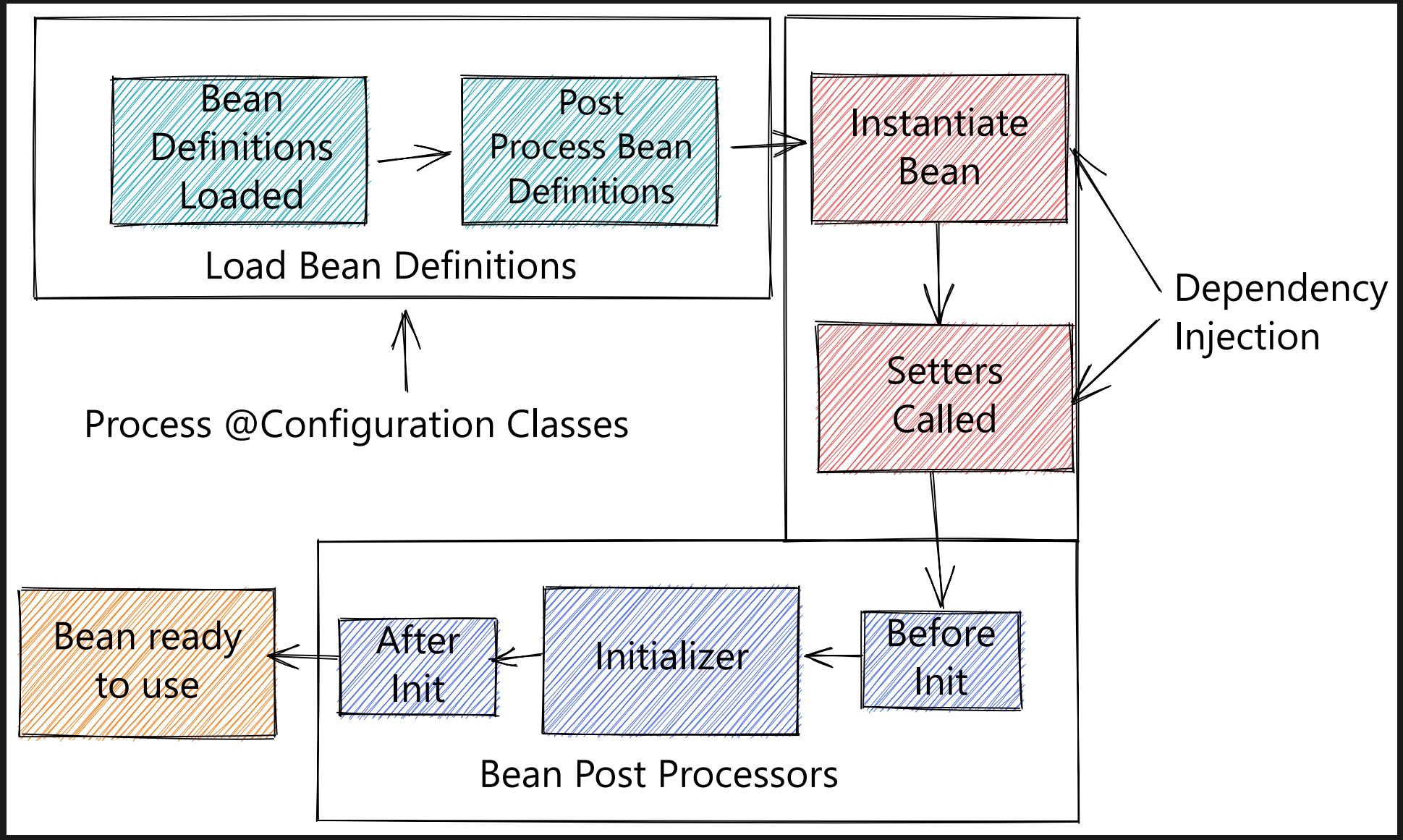

Spring WebFlux
WebClient
JMX
JMS
Web Reactive
Spring MVC
WebSocket
Integration
WebSocket
RSocket
Caching
Web Servlet
Tasks
Scheduling
SockJS
STOMP
Spring Framework
Transactions
DAO
IoC Container
AOP
Events
JDBC
Data Access
Core
XML Mapping
SpEL
Resources
R2DBC
ORM
WebTestClient
Validation
i18n
Testing
TestContext Framework
Spring MVC Test

DEMO

LIFECYCLE OF BEANS


Initialization
Use
Destruction

INITIALIZATION



```java
@Configuration
public class PriceConfiguration {
    @Bean(initMethods = "init", destroyMethod = "destroy")
    public PricingService ps() {
        return new PricingService(pv());
    }
}
```

```java
@Service
public class PricingService implements InitializingBean {
    private final ProductVerifier productVerifier;

    public PricingService(ProductVerifier productVerifier) {
        this.productVerifier = productVerifier;
    }

    @PostConstruct
    public void postConstruct() {
        // triggered first
        // do some initialization work
    }

    @Override
```

@PostConstruct and @PreDestroy are more commonly used compared to init and destroy method

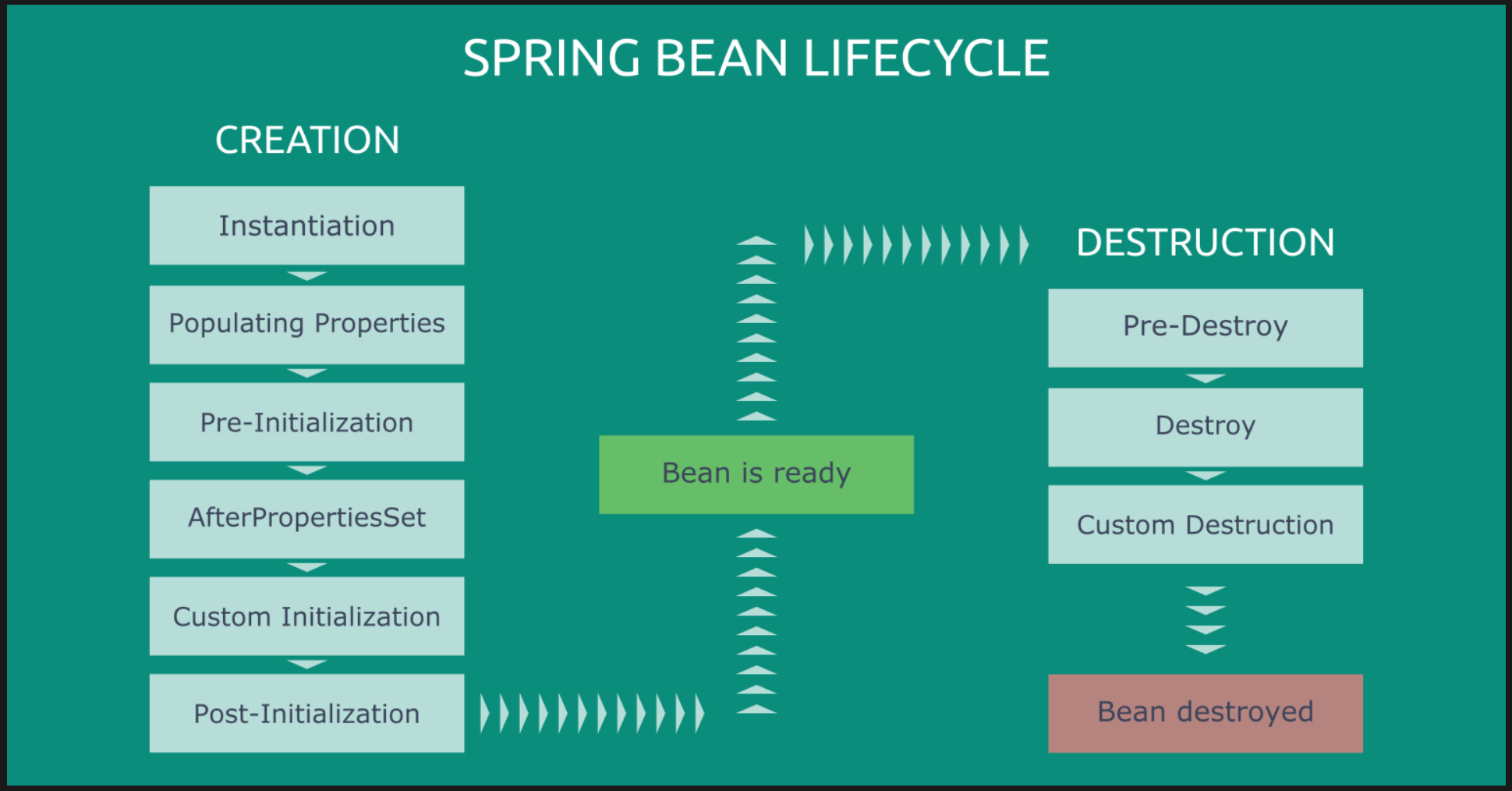
```java
@Component
public class CustomBeanProcessor implements BeanPostProcessor
    @Override
    public Object postProcessBeforeInitialization(Object bean,
        // do some work
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean,
        // do some work
        return bean;
    }
}
```

USE

```
1 @Service
2 public class PricingService {
3   private final PriceVerifier priceVerifier;
4
5   public PricingService(PriceVerifier priceVerifier) {
6     this.priceVerifier = priceVerifier;
7   }
8 }
```

DESTRUCTION

```
1  @Configuration
2  public class PriceConfiguration {
3    @Bean(destroyMethod = "destroy")
4    public PricingService ps() {
5      return new PricingService(pv());
6    }
7  }
8
9  @Service
10 public class PricingService {
11   public void destroy() {
12     // free up resource
13   }
14 }
```


SPRING BEAN LIFECYCLE
CREATION
Instantiation
Populating Properties
Pre-Initialization
AfterPropertiesSet
Custom Initialization
Post-Initialization
Bean is ready
DESTRUCTION
Pre-Destroy
Destroy
Custom Destruction
Bean destroyed

Source

DEMO

APPLICATION EVENTS

Spring has a few built-in Events and automatically published when it happens

- ContextStartedEvent
- ContextStoppedEvent
- ContextRefreshedEvent

Spring Boot also publish some events such as

- ApplicationReadyEvent
- AvailabilityChangeEvent

A couple of ways to listen to the event(s)

LISTENING TO EVENTS

```
// Using interface implementation
public class WebSocketConnection implements ApplicationListene
    public void onApplicationEvent(ApplicationReadyEvent event
        // connect to websocket server
    }
}

// Using @EventListener
public class WebSocketConnection {
    @EventListener(ApplicationReadyEvent.class)
    public void handleEvent() {
        // connect to websocket server
    }

    @EventListener
```

PUBLISHING EVENTS

Provides a ApplicationEventPublisher to publish events

```
@Service
public class EmailService {
    private final ApplicationEventPublisher publisher;

    public EmailService(ApplicationEventPublisher publisher) {
        this.publisher = publisher;
    }

    public void processEmail() {
        // publish custom event
        this.publisher.publishEvent(new EmailEvent());
    }
}
```

DEMO

VALIDATION

Specifications

Specifications

Bean Validation 1.0 (JSR 303)

Specifications

Bean Validation 1.0 (JSR 303)

Bean Validation 1.1 (JSR 349)

Specifications

Bean Validation 1.0 (JSR 303)

Bean Validation 1.1 (JSR 349)

Bean Validation 2.0 (JSR 380)

Specifications

Bean Validation 1.0 (JSR 303)

Bean Validation 1.1 (JSR 349)

Bean Validation 2.0 (JSR 380)

Reference Implementation

Specifications

Bean Validation 1.0 (JSR 303)

Bean Validation 1.1 (JSR 349)

Bean Validation 2.0 (JSR 380)

Reference Implementation

Hibernate Validator

Specifications

Bean Validation 1.0 (JSR 303)

Bean Validation 1.1 (JSR 349)

Bean Validation 2.0 (JSR 380)

Reference Implementation

Hibernate Validator

Apache BVal (exclude 2.0)

Common Annotation

- @NonNull
- @Size
- @Min
- @Max
- @Email
- @NotEmpty
- @Positive
- @Past
- @Future
- @Pattern

```java
// Specify Bean
public class User {
    // multiple assertion per field
    @NotNull
    // custom message
    @Size(min = 5, message = "username to be at least 5 charac
    public String username;
    @Positive(message = "age cannot be lesser than 0")
    public int age;
    @Email
    public String email;
    @Past
    public Date dateOfBirth;
    @FutureOrPresent
    public Date lastUpdated;
```

Spring provides Validator interface for all your validation needs

```
// sample taken from spring docs
public class PersonValidator implements Validator {

    /**
     * This Validator validates only Person instances
     */
    public boolean supports(Class clazz) {
        return Person.class.equals(clazz);
    }

    public void validate(Object obj, Errors e) {
        ValidationUtils.rejectIfEmpty(e, "name", "name.empty")
        Person p = (Person) obj;
        if (p.getAge() < 0) {
            e.rejectValue("age", "negativevalue");
```

Can be used in any layer, anywhere

Spring automatically performs the validation when annotated with @Valid or @Validated

```
@RestController
public class ProfileController {
    @PostMapping("/profile")
    public Profile createProfile(@Valid Profile profile) {
        return repository.create(profile);
    }
}

@ControllerAdvice
public class ProfileControllerAdvice {
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public Map<String, String> handle(MethodArgumentNotValidEx
        return ex.getBindingResult()
                 .getFieldErrors()
                 .stream()
```

We can consolidate all validation error and return to client with the use of @ControllerAdvice

SPRING EXPRESSION LANGUAGE (SPEL)

```java
// https://www.baeldung.com/spring-expression-language

// support Arithmetic Operators
@Value("#{19 + 1}") // 20
private double add;

// support Relational and Logical Operators
@Value("#{1 == 1}") // true
private boolean equal;

// support Logical Operators
@Value("#{400 > 300 || 150 < 100}") // true
private boolean or;

// support Conditional Operators
```

Supports querying and manipulating object at runtime

ASPECT ORIENTED PROGRAMMING (AOP)

Aspects enable the modularization of concerns (such as transaction management) that cut across multiple types and objects. (Such concerns are often termed "crosscutting" concerns in AOP literature.)

Source

Advice (Think like a hook of sort)

- Before advice
- Around advice
- After returning advice
- After throwing advice
- After (finally) advice

Format of execution expression:

- execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?name-pattern(param-pattern) throws-pattern?)

Example

```
execution(public * *(..)) // execution of any public method
execution(* set*(..)) // execution of any method with a name t
execution(* com.xyz.service.*.*(..)) // execution of any metho
```

Example

```
// must declare @Aspect
@Aspect
public class AroundExample {
    // Perform some action before and after the method
    @Around("com.bwgjoseph.app.*Service()")
    public Object timer(ProceedingJoinPoint pjp) throws Throwa
        // perform some action before the method run
        StopWatch clock = new StopWatch("Profiling Start");
        // method run
        Object retVal = pjp.proceed();
        // perform some action after the method run
        clock.stop();
        log.info("Profiling Stop, took {}", stopWatch.getTotal

        // return
```

USE-CASES

- Logging
- Auditing
- Access Control
- Caching

But most of them are provided ootb by Spring via Annotation

INTRO TO SPRING BOOT

- Build standalone and production ready Spring application
- Mostly just auto-configuration
  - Detects if certain class is in the classpath, and autoconfigure the Beans required
- Bootstrap everything for you via @SpringBootApplication
  - @EnableAutoConfiguration
  - @ComponentScan
  - @Configuration

- Reduce boilerplate codes
  - Configure Datasource
  - Configure Message Broker
  - Configure Webserver
  - Configure Spring Security, ...

Let's see Spring Boot source code

WHAT DOES SPRING BOOT PROVIDES?

Core Features: SpringApplication | External Configuration | Profiles | Logging

Web Applications: MVC | Webflux | Embedded Containers

Working with data: SQL | NO-SQL

Messaging: JMS | AMQP | Kafka

Testing: Boot Applications | Utils

Extending: Auto-configuration | @Conditions

Core Features

EXTERNALIZED CONFIGURATION

Read the value from the following order (with values from lower overriding earlier ones)

Total of 14 locations

- ...
- application.properties (inside jar (default, profile-specific), outside jar (default, profile-specific))
- ...
- OS Environment Variable
- ...
- SPRING_APPLICATION_JSON
- Command line arguments
- ...
- @TestPropertySource
- ...

TYPE-SAFE CONFIGURATION (@CONFIGURATIONPROPERTIES)

```java
@ConfigurationProperties("my.app.service")
@Validated
public class MyProperties {
    private boolean enabled;
    private final Security security = new Security();

    public static class Security {
        @NotEmpty
        private String userName;
        @Size(min = 5)
        private String password;
        private List<String> roles = new ArrayList<>(Collectio
    }
}
```

```
// env var = MY_APP_SERVICE_ENABLED
my.app.service.enabled=true
my.app.service.security.user-name=joseph
my.app.service.security.password=pw
my.app.service.security.roles=user,admin
```

PROFILES

```
// allow for multiple active profile
spring.profiles.active=dev,production
```

```
@Configuration
@Profiles("dev")
public class DevConfiguration { }

@Configuration
@Profiles("production")
public class ProductionConfiguration { }
```

Can override

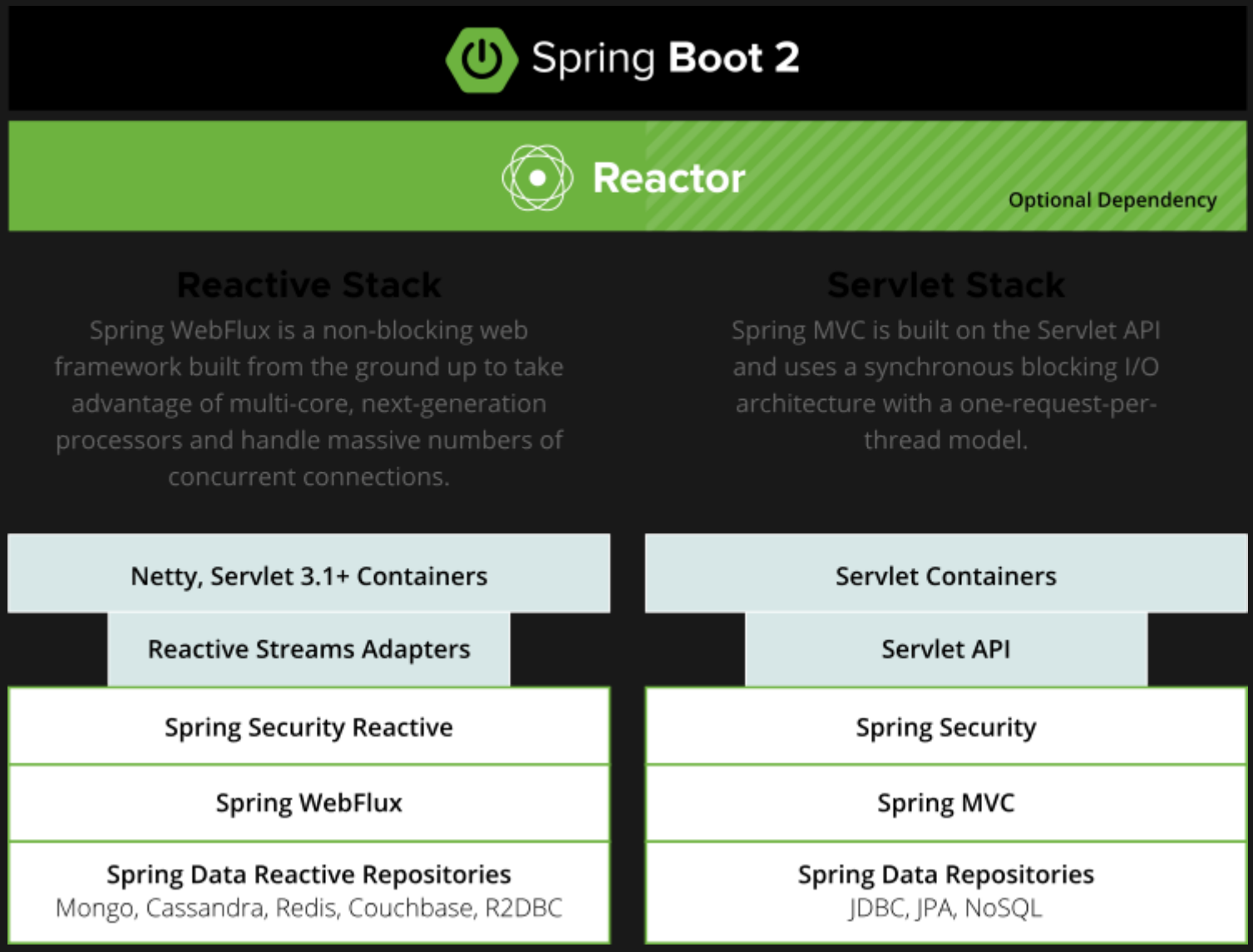
- via command line with --spring.profiles.active=dev
- via env var SPRING_PROFILES_ACTIVE=dev

LOGGING

```
logging.level.root=warn
logging.level.org.springframework.web=debug

// group different package logs together for easier control
logging.group.tomcat=org.apache.catalina,org.apache.coyote,org
logging.level.tomcat=trace

// predefined by Spring
logging.group.web=debug
logging.group.sql=warn
```

Web Applications

SPRING WEB FRAMEWORK



CALLING REST SERVICE

```
@Service
public class MyService {
    // In maintenance mode, use WebClient instead
    private final RestTemplate restTemplate;

    // Always use RestTemplateBuilder
    public MyService(RestTemplateBuilder restTemplateBuilder)
        // builder allows further customization such as Auth
        this.restTemplate = restTemplateBuilder.basicAuthentic
    }

    public Details someRestCall(String name) {
        return this.restTemplate.getForObject("/{name}/details
    }
}
```

CALLING REST SERVICE

```
@Service
public class MyService {
    // Similar to RestTemplate but support reactive
    private final WebClient webClient;

    public MyService(WebClient.Builder webClientBuilder) {
        this.webClient = webClientBuilder.baseUrl("https://exa
    }

    public Mono<Details> someRestCall(String name) {
        return this.webClient.get().uri("/{name}/details", nam
    }

}
```

Working with data

SPRING DATA

- Supports a wide range of SQL and NoSQL databases
  - Wide range from either the core, or community supported
- Derived Query
- Providing familiar programming model for both type of database

```
@Respository
public class UserRespository extends CrudRepository<User, Long
public class UserRespository extends PagingAndSortingRepositor
```

SPRING BOOT ACTUATOR

- Monitor Application via Metrics and Endpoints
  - health, metrics, env, loggers, etc
- Ability to create custom `@WebEndpoint`
- Can enable/disable via properties configuration

```
management.endpoints.web.exposure.include=*
management.endpoints.web.exposure.exclude=env,beans
```

COMMON ANNOTATION

CORE

- @Bean
- @Primary
- @Qualifier
- @Value
- @Configuration
- @ConfigurationProperties
- @Autowired
- @Profile
- @ComponentScan

GENERIC STEROTYPE

- @Component

SPECIALIZE STEROTYPE

- @Service
- @Controller / @RestController
- @Repository

LIFECYCLE

- @PostConstruct
- @PreDestroy

WEB

- @RequestMapping
- @RequestBody
- @GetMapping
- @PostMapping
- @PatchMapping
- @PutMapping
- @DeleteMapping
- @ExceptionHandler
- @PathVariable
- @RequestParam
- @CrossOrigin

CONFIGURATION

- @ConditionOnClass / MissingClass
- @ConditionOnBean / MissingBean
- @ConditionOnProperty
- @Conditional

TEST

- @SpringBootTest
  - Setup entire ApplicationContext, usually for IntegrationTest
- @TestConfiguration
- @TestPropertySource
- @ContextConfiguration

SLICED TEST

Setup specific context to run test

- @WebMvcTest
- @WebFluxTest
- @JdbcTest
- @DataJpaTest
- @DataMongoTest
- @JsonTest
- @RestClientTest

QUESTIONS AND ANSWER?

What is the main difference in the workings behind @Bean and @Component? It seems that @Bean is commonly used in @Configuration class

Let's take a quick look at this beans-scanning-autodetection first

Remember that @SpringBootApplication is a consolidated annotation for @EnableAutoConfiguration, @ComponentScan and @Configuration

@ComponentScan scans for all classes registered below it by default (com.bwgjoseph.app.*)

Then it scans for all sterotype annotation such as @Configuration, @Component, @Controller, @Service, @Repository to register the Beans (class) to application context

@Component is a generic sterotype annotation, while the rest are specialize sterotype annotation

Spring does a little bit more when you annotate a class with @Respository by turning the checked exception into runtime exception

When define with @Bean annotation, you are instructing how you want the class to be created [meta-data]

Spring picks up @Bean from @Configuration annotated class to register the @Bean to application context

- You can use XML to configure the beans too!

Can talk a little about Spring Cloud Gateway and Spring Webflux?

Spring cloud cannot be used with spring web because spring cloud gateway runs on top of Netty and requires webflux. Not really sure how Netty and Webflux come in for the case of Spring Cloud

- Referring back to the Spring Web diagram
- Basically, it's just how they design and wrote SCG
- Traditionally, SCG is deployed as standalone, and doesn't matter if it's built using netty and all
- See scg-github

What goes behind @Autowired

- Maybe can also talk about bean lifecycle
- @Autowired and @Bean
  - Saw something on stackoverflow and I am not very sure what distinguish the “outside” and “context” world:

@Autowire lets you inject beans from context to "outside world" where outside world is your application. Since with @Configuration classes you are within "context world ", there is no need to explicitly autowire (lookup bean from context).

- Refer back to the lifecycle diagram
- I believe what the author meant
  - **within** is the Initialization phase of the lifecycle, whereas,
  - **outside world** refers to the Use phase
- Why do you not need to explicitly call @Autowired in B bean then?
  - Spring resolve that automatically for you
  - See docs

Typically what is the right convention to run logic at startup, and what are the considerations (e.g. dependencies, nature of initialisation logic etc): @PostConstruct for specific components, @AfterPropertiesSet ,EventListeners, initMethod, CommandLineRunner/ ApplicationRunner?

- I think there's no right convention, it depends on when you want the logic to run
- There is a order to these and also happens at different stage of the lifecycle

See startup

Mostly only encountered beans with default scope of singleton. What are some examples of using other scopes e.g. Prototype? How does the spring container handle such scopes (since singleton is just caching such the beans?)

- The different type of bean scope are
  - Singleton, Prototype, Request, Session, Websocket, Application
  - See guide for detailed explanation
- Most people (I think) don't use prototype because you can simply call `new User()`
- Using `prototype` scope means allowing Spring to manage the lifecycle for you but unsure of the actual benefit
- See docs

We cannot inject values from application.properties into static fields. Should we use setter method to initialise the value, or should this not be a static field (assuming it is a component which is singleton).

- In short, do not use static field
- Need use case of when you think you will need it

What are some use-cases of @Lazy Loading & when to use it?

- TLDR; don't unless you know what you are doing

- Use when you want the application to startup faster
- Remember that when application starts, Spring will scans for all beans and initialize it
- With `@Lazy`, it will skip during startup, and init only when it's first called
- But, I think the savings in slightly faster startup does not gives alot of benefit as compared to the disadvantage it brings
  - you won't know if the bean creation/init has problem until it is first called during runtime
  - the time saving in application startup defers to the initial request
- See lazy-initialization

There's spring-native anyway, if you want really fast startup

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.4.3)

2021-03-10 11:15:52.817  INFO 18897 --- [           main] com.example.webmvc.WebmvcApplication     : Starting WebmvcApplication v0.0.1-SNAPSHOT using Java 11.0.10 on m1 with PID 18897
 (/home/seb/workspace/spring-native/samples/webmvc-tomcat/target/webmvc-tomcat started by seb in /home/seb/workspace/spring-native/samples/webmvc-tomcat)
2021-03-10 11:15:52.818  INFO 18897 --- [           main] com.example.webmvc.WebmvcApplication     : No active profile set, falling back to default profiles: default
2021-03-10 11:15:52.836  INFO 18897 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8082 (http)
Mar 10, 2021 11:15:52 AM org.apache.coyote.AbstractProtocol init
INFO: Initializing ProtocolHandler ["http-nio-8082"]
Mar 10, 2021 11:15:52 AM org.apache.catalina.core.StandardService startInternal
INFO: Starting service [Tomcat]
Mar 10, 2021 11:15:52 AM org.apache.catalina.core.StandardEngine startInternal
INFO: Starting Servlet engine: [Apache Tomcat/9.0.43]
Mar 10, 2021 11:15:52 AM org.apache.catalina.core.ApplicationContext log
INFO: Initializing Spring embedded WebApplicationContext
2021-03-10 11:15:52.837  INFO 18897 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 19 ms
2021-03-10 11:15:52.844  INFO 18897 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing ExecutorService 'applicationTaskExecutor'
Mar 10, 2021 11:15:52 AM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["http-nio-8082"]
2021-03-10 11:15:52.849  INFO 18897 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8082 (http) with context path ''
2021-03-10 11:15:52.850  INFO 18897 --- [           main] com.example.webmvc.WebmvcApplication     : Started WebmvcApplication in 0.037 seconds (JVM running for 0.038)
```

SPRING BEST PRACTICES

Prefer Constructor Injection over field and setter

```
@Service
public class UserService {
    // set to final
    // @Autowired not required
    private final UserRepository userRepository;

    // @Autowired not required
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
}
```

Use setter based injection for optional dependency

Never use field based injection!

Prefer @ConfigurationProperties if injecting multiple @Value

```
@ConfigurationProperties("my.app.service")
@Validated
public class MyProperties {

    private boolean enabled;
    private final Security security = new Security();

    public static class Security {
        @NotEmpty
        private String userName;
        @Size(min = 5)
        private String password;
        private List<String> roles = new ArrayList<>(Collectio
    }
}
```

```
my.app.service.enabled=true
my.app.service.security.user-name=joseph
my.app.service.security.password=pw
my.app.service.security.roles=user,admin
```

Prefer RuntimeException and centralized error handling via @ExceptionHandler

```
@RestController
public class ProfileController {
    @PostMapping("/profile")
    public Profile createProfile(@Valid Profile profile) {
        // always return success message
        // leave error to @ExceptionHandler
        return repository.create(profile);
    }
}

@ControllerAdvice
public class ProfileControllerAdvice {
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public Map<String, String> handle(MethodArgumentNotValidEx
        return ex.getBindingResult()
```

Trigger validation at @Controller via @Valid and @Validated

```
public class User {
    // multiple assertion per field
    @NotNull
    // custom message
    @Size(min = 5, message = "username to be at least 5 charac
    public String username;
    @Positive(message = "age cannot be lesser than 0")
    public int age;
    @Email
    public String email;
    @Past
    public Date dateOfBirth;
    @FutureOrPresent
    public Date lastUpdated;
}
```

Apply DTO Model Pattern

```
// used in @Controller
public class UserDTO {
    // contains extra stuff for client-server communcation
}

// used in service and below
public class UserEntity {
    // contains stuff for server-db communication
}
```

Drop ResponseEntity response type unless full control is required

```
// instead of
@RestController
public class UserController {
    @PostMapping("/user")
    public ResponseEntity<User> createUser(@RequestBody @Valid
        return new ResponseEntity.ok(user);
    }
}

// use this
@RestController
public class UserController {
    @PostMapping("/user")
    public User createUser(@RequestBody @Valid User user) {
        return user;
```

Externalised Configuration

- Environment Variables
- K8s ConfigMap
- Spring Cloud Config Server
  - Supports git, filesystem, vault, jbdc, redis, s3, etc

Always rely on RestTemplateBuilder (or any Builder)

~~SPRING~~ BEST PRACTICES

Prefer package-by-feature over package-by-layer

Prefer builder pattern

```
// take advantage of ide intellisense
User user = User.builder()
                .username('joseph')
                .password('helloworld')
                .roles(List.of('engineer'))
                .build();

// easily miss out fields to set
User user = new User();
user.setUsername('joseph');
user.setPassword('helloworld');
user.setRoles(List.of('engineer'));
```

Prefer coding to interface (not when it’s single class or when it make sense)

```
public interface CompressionAlgorithm {
    public T compress(T data) {}
}

public class Simple implements CompressionAlgorithm {
    @Override
    public T compress(T data) {}
}

public class Advance implements CompressionAlgorithm {
    @Override
    public T compress(T data) {}
}

public class CompressionAlgorithmFactory {
```

Strategy Pattern

```
public interface CompressionAlgorithm {
    public T compress(T data) {}
}

public class Simple implements CompressionAlgorithm {
    @Override
    public T compress(T data) {}
}

public class Advance implements CompressionAlgorithm {
    @Override
    public T compress(T data) {}
}

public class CompressService {
```

Use of Inversion of Control

```
public interface ProfileDeletionEvent {
    void onDeleteProfileEvent(String profileId);
}

class ProfileService {
    private final List<ProfileDeletionEvent> profileDeletionEv

    public void delete(String profileId) {
        profileDeletionEvent.forEach(event -> event.onDeletePr
    }
}

class SolrProfileDeletion implements ProfileDeletionEvent {
    @Override
    public void onDeleteProfileEvent(String profileId) {
```

Use Event Publisher works too!

Fail Fast

```
public class UserService {
    // don't do this
    public User validateUser(User user) {
        if (user.name != null) {
            if (user.password != null) {
                processUser(user);
            } else {
                throws new Exception("no password");
            }
        } else {
            throws new Exception("no name");
        }
    }

    // much easier to see what's going on
```

Exit Early

```
public class UserService {
    public boolean processUser(@NonNull User user) {
        if (user.username == null || user.username.length == 0

        return true;
    }

    public List<User> transformUser(List<User> user) {
        if (user.isEmpty()) return new ArrayList<>();

        // continue
        return users;
    }
}
```

Declare interface instead of implementation

```
public class UserService {
    // don't do this
    public HashMap<String, String> getUserMap() {
        // some implementation
        return someMap;
    }

    // do this
    public Map<String, String> getUserMap() {
        // some implementation
        return someMap;
    }
}
```

Prefer using `this`

```
public class UserService {
    private final ProfileService profileService;

    public UserService(ProfileService profileService) {
        // reduce ambiguity
        this.profileService = profileService;
    }

    public boolean checkProfileExist(User user) {
        // clearer indication of class variable
        return this.profileService.profileExist(user.getUserna
    }
}
```

Prefer non inverse logic

```
// don't do this
const isValid = response.results.length === 0;
if (!isValid) { ... }

// do this
const hasResult = response.result.length > 0;
if (hasResult) { ... }
```

WHAT'S NEXT?

Upcoming tech sharing

- Testing Spring Application
- Git
- Suggestion?

THANK YOU, AND QUESTIONS?

Feedback welcome at google-form

also, hacktoberfest is here!