The background is a solid blue gradient. Overlaid on this are numerous thin, white, curved lines that flow from the left side towards the right, creating a sense of motion and depth. These lines are more densely packed in some areas, forming a wave-like pattern that peaks towards the right side of the frame.

# WPROWADZENIE DO SPRING



# BARTOSZ ANDREATTO



Programista w Banku Pekao S.A z bogatym doświadczeniem w zakresie technologii back-end'owych wykorzystujących wirtualną maszynę Javy. Pasjonat rozwiązań opartych na ekosystemie Spring oraz rozwiązań Oracle. Certyfikowany programista Java. Uwielbia nauczać oraz dzielić się wiedzą.

**LinkedIn:** <https://www.linkedin.com/in/bartosz-andreatto-02b03413a/>

**e-mail:** [bandreatto@gmail.com](mailto:bandreatto@gmail.com)





# ZAKRES SZKOLENIA

## DZIEŃ 1

- Charakterystyka frameworku
- Najważniejsze wzorce i praktyki wykorzystywane w Spring
- Tworzenie i konfiguracja projektu
- Praca z dokumentacją
- Idea inwersji kontroli (IoC) oraz wstrzykiwania zależności
- Działanie i odpowiedzialność kontenera
- Konfiguracja z użyciem adnotacji oraz JavaConfig



# ZAKRES SZKOLENIA

## DZIEŃ 2

- Komponenty zarządzane – tworzenie, definiowanie zasięgu oraz zależności, cykl życia
- Podstawy programowania aspektowego
- Tworzenie, konfigurowanie i użycie aspektów
- Event bus i programowanie przez zdarzenia



# FRAMEWORK A BIBLIOTEKA

Podczas gdy **biblioteka jest zbiorem obiektów, klas i metod**, które nasza aplikacja może wykorzystać i to **programiści decydują kiedy, gdzie i w jaki sposób wywołać i wykorzystać poszczególne jej części**, to framework jest specjalnie przygotowanym szkieletem, który definiuje podstawową strukturę i zachowanie aplikacji, ale pozwala programiście w konkretnych miejscach wpiąć swój kod. Jest to tzw. **mechanizm inversion of control (IoC)**.



# FRAMEWORK A BIBLIOTEKA

Frameworki, **w tym również Spring**, zwalniają programistę od tworzenia całej aplikacji od zera. Podstawową bazę aplikacji mamy „za darmo”, dzięki czemu podstawową funkcjonalność aplikacji jesteśmy w stanie stworzyć dużo szybciej. Ponadto **nie jesteśmy zmuszeni testować pewnych części naszej aplikacji** (zostały one już przetestowane przez twórców frameworka i społeczność).



# FRAMEWORK A BIBLIOTEKA

Wykorzystywanie frameworków, mimo wielu plusów, ma również swoje minusy. Przede wszystkim **dodają pewien narzut na wydajność aplikacji** i czasem, ze względu na **zamkniętą strukturę frameworka**, wymuszają pewien konkretny sposób rozwiązania problemu.

Twórcy frameworka Spring stworzyli doskonałą dokumentację, w której znajdziemy rozwiązania większości problemów, na które trafiamy podczas jego nauki.

<https://docs.spring.io/spring-framework/docs/current/reference/html/>



<a href="#">Overview</a>	History, Design Philosophy, Feedback, Getting Started.
<a href="#">Core</a>	IoC Container, Events, Resources, i18n, Validation, Data Binding, Type Conversion, SpEL, AOP, AOT.
<a href="#">Testing</a>	Mock Objects, TestContext Framework, Spring MVC Test, WebTestClient.
<a href="#">Data Access</a>	Transactions, DAO Support, JDBC, R2DBC, O/R Mapping, XML Marshalling.
<a href="#">Web Servlet</a>	Spring MVC, WebSocket, SockJS, STOMP Messaging.
<a href="#">Web Reactive</a>	Spring WebFlux, WebClient, WebSocket, RSocket.
<a href="#">Integration</a>	REST Clients, JMS, JCA, JMX, Email, Tasks, Scheduling, Caching, Observability.
<a href="#">Languages</a>	Kotlin, Groovy, Dynamic Languages.
<a href="#">Appendix</a>	Spring properties.
<a href="#">Wiki</a>	What's New, Upgrade Notes, Supported Versions, additional cross-version information.

## Note

This documentation is also available in [PDF](#) format.

# SPRING FRAMEWORK

Pierwsza wersja frameworka Spring miała swoją premierę w roku 2003. Pomimo swojego wieku, framework ten przechodził metamorfozy i jest ciągle dostosowywany do aktualnych wymagań świata IT, **poprzez integrację z najnowszymi popularnymi technologiami**. Aby móc wykorzystać wszystkie możliwości frameworka trzeba dobrze zrozumieć podstawowe jego części tzn.:

- **kontener IoC (Inversion of Control)**
- mechanizm odpowiedzialny za **dependency injection**
- część odpowiedzialną za **konfigurację**





# PODSTAWOWE ELEMENTY

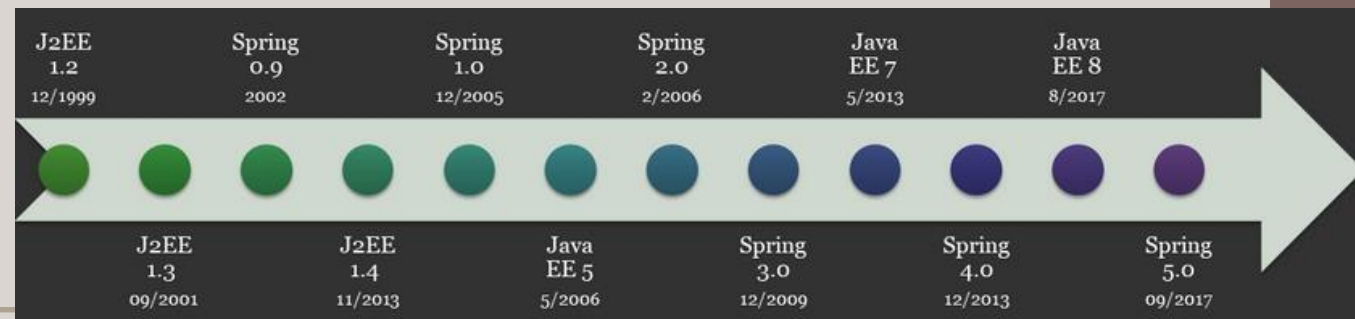
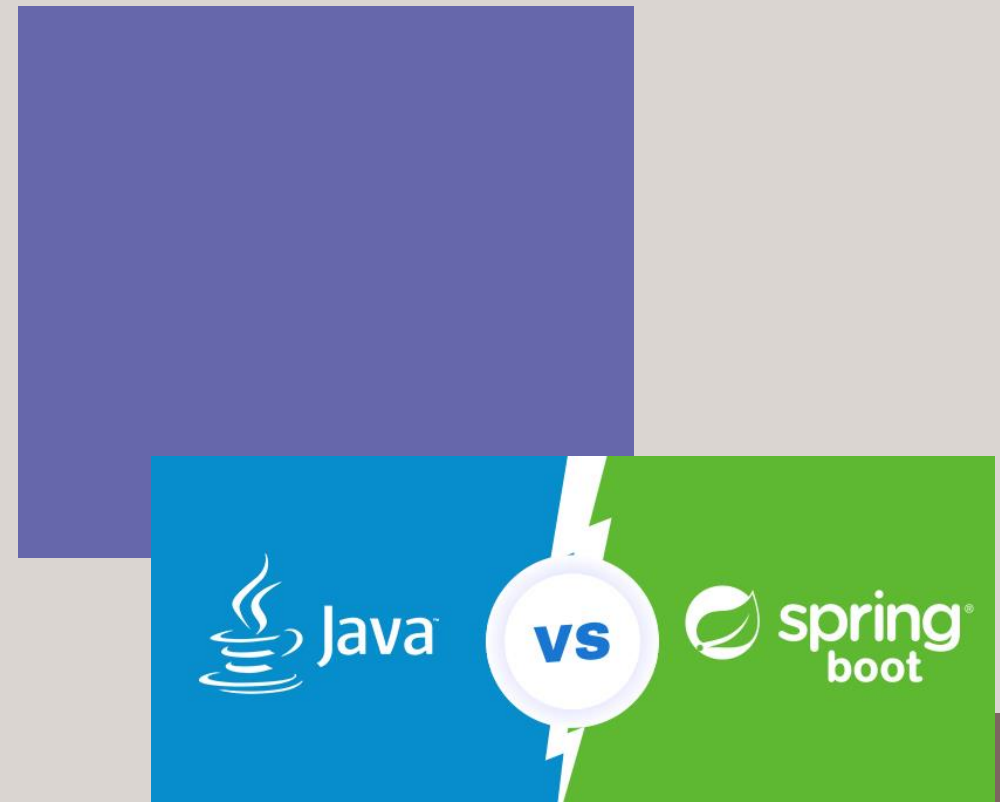
Framework Spring **można wykorzystywać do tworzenia aplikacji webowych**, dzięki bogatej i dojrzałej warstwie webowej. Ponadto Spring świetnie sprawdza się podczas tworzenia różnych aplikacji dzięki:

- wsparciu dla warstwy **persistence**
- łatwej implementacji **transakcyjności** dla poszczególnych operacji
- możliwości wykorzystania **modelu MVC** (tzw. **model - view - controller**)
- wsparciu dla **programowania aspektowego** (tzw. **AOP**)



# SPRING A JAVAEE/JAKARTAEE

Spring jest często porównywany z **Java Enterprise Edition**. O ile założenia obu technologii są podobne, tak podejście do rozwiązań jest nieco inne. JavaEE wymusza **uruchamianie aplikacji na specjalnym serwerze aplikacji**, który obsługuje **Java Enterprise**. Z kolei Spring nie ma takich ograniczeń. Aplikacje oparte o Spring **możemy uruchamiać, wykorzystując np. serwer Tomcat czy Netty**.



# SPRING - WADY

Aplikacja oparta o framework Spring **składa się z wielu elementów**. W celu wykorzystania możliwości frameworka musimy do zależności projektu dodać pewne jary, na przykład:

- spring-web
- spring-webmvc
- spring-core
- spring-context
- spring-aop
- jakarta.annotation-api
- tomcat-embedded-core





# SPRING - WADY

Zależności te, nie tylko pochodzą z grupy **org.springframework**. Stąd skonfigurowanie aplikacji, która ma wykorzystywać framework Spring **jest problematyczne** ponieważ musimy posiadać wiedzę:

- **jakie zależności z grupy org.springframework** wykorzystać
- **jakie pozostałe zależności** wykorzystać w naszym projekcie, aby poprawnie się on uruchomił
- **jakie wersje** tych zależności wykorzystać

Ponadto połączenie tych wszystkich zależności (tzn. ich konfiguracja) jest **nietrywialne**.



# PROJEKT SPRING-BOOT

Wokół frameworka Spring powstało dziesiątki mniejszych projektów, jednakże najważniejszym i najczęściej wykorzystywanym jest **Spring Boot**. Jego głównym celem jest wyeliminowanie opisanych wyżej wad, tzn. dzięki wykorzystaniu Spring Boota:

- **nie musimy pamiętać o konkretnych bibliotekach**
- nie musimy **znać wersji kompatybilnych bibliotek**
- otrzymujemy **domyślną konfigurację**
- nie musimy wykorzystywać plików war
- nie musimy wykorzystywać **plików xml do konfiguracji projektu**



spring boot

# STARTERY

Tworząc aplikacje i rozważając jaki stos technologiczny powinniśmy wybrać, najczęściej **daną technologię łączymy z daną warstwą aplikacji**. Twórcy Springa i jego społeczność przygotowuje **specjalne projekty, które zawierają zestawy potrzebnych bibliotek, ich automatyczną konfigurację**. Projekty te możemy bezproblemowo wykorzystać w aplikacji jako zwykłe zależności. Projekty te nazywamy starterami. **Każdy starter może składać się z wielu zależności, a nawet innych starterów**.





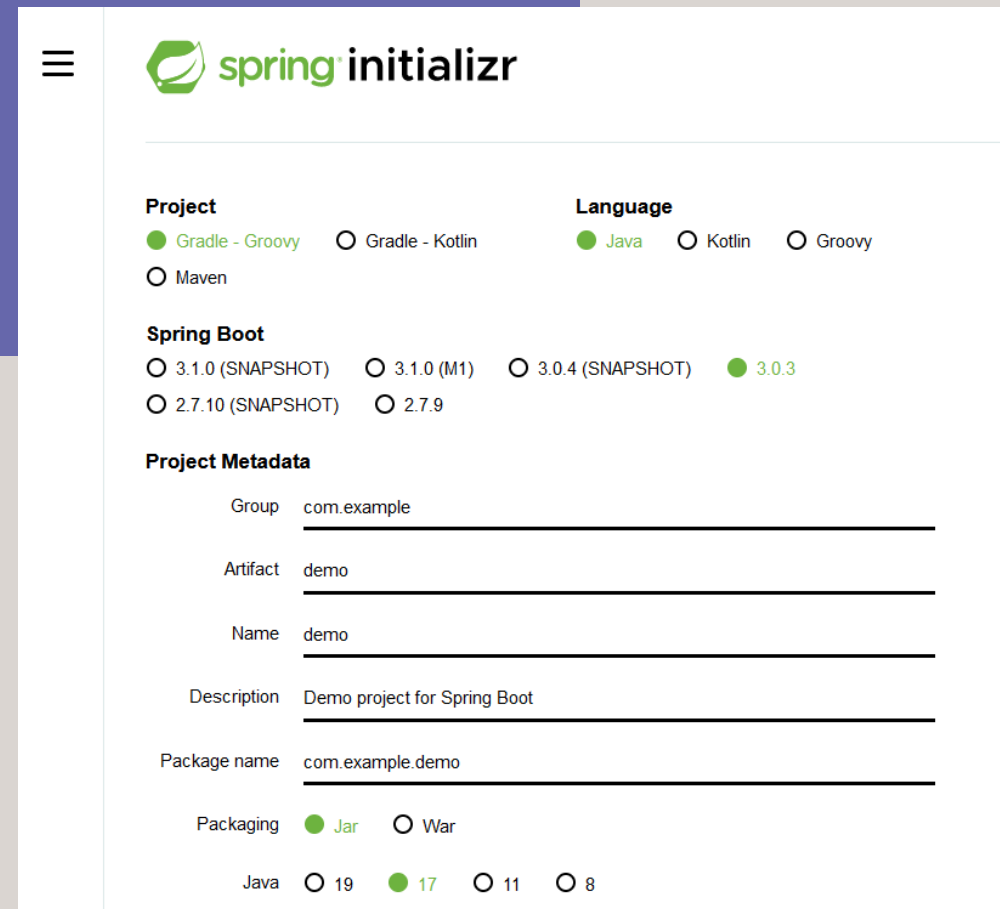
# TWORZENIE PROJEKTÓW

Chcąc stworzyć bazowy projekt oparty o Spring Boot, możemy wykorzystać:

- **stronę [start.spring.io](https://start.spring.io)**
- **IntelliJ IDEA Ultimate (za pomocą File -> New -> Project -> Spring Initializr)**

W obu przypadkach podjąć następujące decyzje:

- czy projekt powinien być oparty o Mavena czy Gradle
- jaką wersję Spring Boota chcemy wykorzystać
- podać podstawowe dane nt. projektu
- wybrać wersję Javy
- wybrać startery, jakie chcemy wykorzystać w projekcie
- wybrać zależności, niebędące starterami



The screenshot shows the Spring Initializr web interface. It features a hamburger menu icon on the left and the 'spring initializr' logo at the top. The form is organized into sections: 'Project' with radio buttons for 'Gradle - Groovy' (selected), 'Gradle - Kotlin', and 'Maven'; 'Language' with radio buttons for 'Java' (selected), 'Kotlin', and 'Groovy'; 'Spring Boot' with radio buttons for versions '3.1.0 (SNAPSHOT)', '3.1.0 (M1)', '3.0.4 (SNAPSHOT)', '3.0.3' (selected), '2.7.10 (SNAPSHOT)', and '2.7.9'; and 'Project Metadata' with text input fields for 'Group' (com.example), 'Artifact' (demo), 'Name' (demo), 'Description' (Demo project for Spring Boot), and 'Package name' (com.example.demo). At the bottom, there are radio buttons for 'Packaging' ('Jar' selected, 'War' unselected) and a 'Java' version selector with radio buttons for '19', '17' (selected), '11', and '8'.

# TWORZENIE PROJEKTÓW

Przykładowe startery:

- Spring-boot-starter-web
- Spring-boot-starter-data-jpa
- Spring-boot-starter-thymeleaf
- Spring-boot-starter-security

Wygenerowanie projektu z uwzględnieniem zależności:

- **Spring WEB**
- **Spring DATA JPA**
- **Thymeleaf**
- **H2 Database**
- **Lombok**



# URUCHAMIANIE PROJEKTU

W zależności od wykorzystanych starterów **aplikacja może uruchamiać się w inny sposób**. W projektach wykorzystujemy jednak bardzo często startery **spring-boot-starter-web** czy **spring-boot-starter-webflux**, które uruchamiają aplikację bezpośrednio na np. Tomcat'cie. Jeżeli wykorzystujemy **IntelliJ Ultimate**, po zaimportowaniu wygenerowanego projektu do IDE domyślna konfiguracja powinna być dostępna. Po uruchomieniu aplikacji w logach powinniśmy zobaczyć komunikat potwierdzający jej poprawne uruchomienie:

```
: Started SbApplication in 2.448 seconds (JVM running for 2.996)
```





# URUCHAMIANIE PROJEKTU

Jeżeli podczas uruchamiania widzimy następujący błąd:

```
Web server failed to start. Port 8080 was already in use.
```

w pliku **application.properties** dodajemy następujący wpis: **server.port=8081**.

UWAGA: Jeżeli nie posiadamy dostępu do IntelliJ Ultimate, aplikację można również uruchomić wykorzystując **Mavena** i linię komend:  
**mvn spring-boot:run**.



# KOMPONENTY I IOC

Jedną z najważniejszych i ciekawszych funkcjonalności Springa jest **kontener IoC**, który jest odpowiedzialny za mechanizm **dependency injection (DI)**. Dzięki temu jesteśmy zwolnieni z konieczności ręcznego tworzenia obiektów (np. za pomocą operatora `new`).

# KOMPONENTY I IOC

W celu zrozumienia jak działa DI, należy zrozumieć:

- **Czym są beany?**
- **W jaki sposób rejestrować beany w kontenerze IoC**  
(Inversion of Control)?
- **W jaki sposób wstrzykiwać zarejestrowane beany?**





# BEANY I REJESTRACJA W KONTENERZE IOC

Beanem (lub tzw. komponentem) nazywamy **obiekt, który jest zarządzany przez kontener IoC**. Z kolei kontener IoC jest mechanizmem, który jest odpowiedzialny za zarządzanie beanami. Oznacza to, że **programista nie jest bezpośrednio odpowiedzialny za tworzenie lub usuwanie takiego obiektu**.

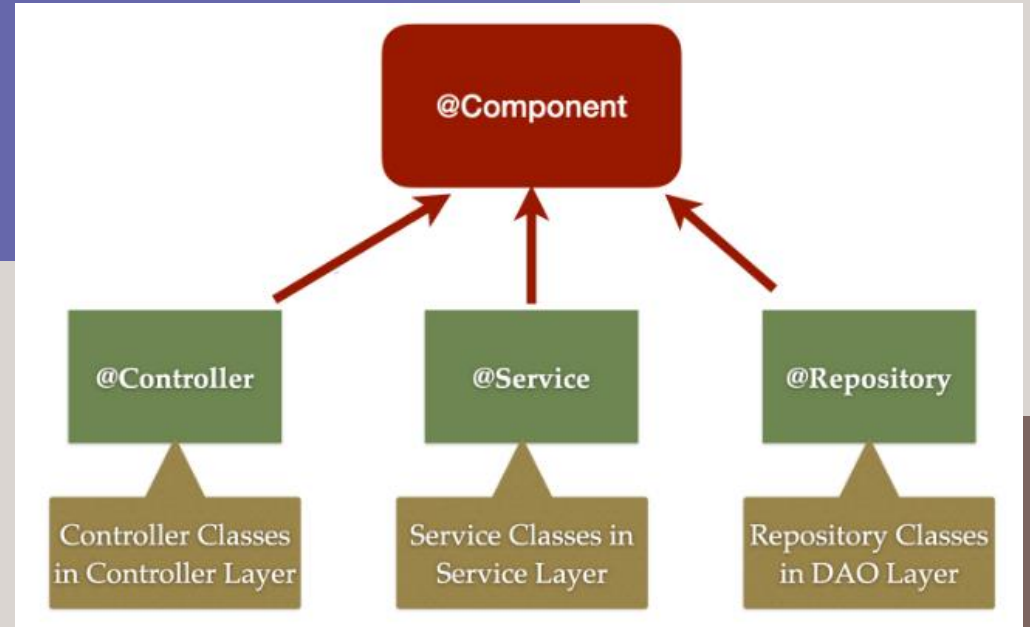


# BEANY I REJESTRACJA W KONTENERZE IOC

Istnieje kilka sposobów zarejestrowania obiektu w kontenerze IoC. Możemy do tego wykorzystać:

- odpowiednią konfigurację w pliku xml
- nad klasą wykorzystać jedną z adnotacji (tzw. stereotypów): **@Component**, **@Service**, **@Repository**, **@Controller**
- wykorzystać klasę oznaczoną adnotacją **@Configuration** z metodami oznaczonymi adnotacją **@Bean**

UWAGA: **Zbiór wszystkich beanów** w aplikacji często potocznie nazywamy **kontekstem**. Kontekst jest **rozwiązywany na starcie aplikacji**.



# STEREOTYPY

Wymienione adnotacje wykorzystywane nad klasami pełnią tę samą funkcję - **oddają kontrolę nad klasą do kontenera IoC**. Istnieją między nimi pewne różnice:

- **@Component** - oznacza generyczny komponent, jeżeli nie jesteś pewien, którą adnotację wykorzystać - **ta będzie poprawnym wyborem**
- **@Service** - funkcjonalnie nie różni się niczym od adnotacji @Component. Jest jedynie informacją, że **klasa reprezentuje warstwę serwisową**, tzn. taką, która **zawiera logikę biznesową**.



# STEREOTYPY

- **@Repository** - reprezentuje warstwę **dostępu do bazy danych** (tzn. klasę, która wykorzystuje, np. instancję EntityManager do wykonania zapytań do relacyjnej bazy danych). Od adnotacji @Component **różni się jedynie tym, że w przypadku wystąpienia błędu na warstwie bazodanowej, możemy otrzymać bardziej szczegółowe informacje w wyjątkach.**
- **@Controller** - działa tak jak @Component ale bean, nad którym znajduje się ta adnotacja, trafia również do tzw. **WebApplicationContext**, tzn. do tej części kontekstu, która reprezentuje warstwę webową aplikacji.





# STEREOTYPY

```
1 import org.springframework.stereotype.Component;
2
3 @Component
4 public class SimpleCalculator {
5
6     public Double add(final Double valA, final Double valB) {
7         return valA + valB;
8     }
9 }
```

```
1 import org.springframework.stereotype.Service;
2
3 import java.util.List;
4 import java.util.stream.Collectors;
5
6 @Service // można użyć @Component zamiast @Service
7 public class TextProcessorService {
8
9     public String processSentences(final List<String> sentences) {
10         return sentences.stream()
11             .filter(sentence -> !sentence.isBlank())
12             .map(sentence -> sentence.substring(0, 1).toUpperCase()
13                 + sentence.substring(1, sentence.length() - 1) + ".")
14             .collect(Collectors.joining(" "));
15     }
16 }
17
```

# COMMANDLINERUNNER

Spring jest frameworkiem, a więc posiada miejsca, **w których można wywołać swój kod**. Najczęściej możemy to zrobić poprzez stworzenie komponentu konkretnego typu (tzn. klasy implementującej pewien interfejs lub rozszerzającą pewną klasę). Jednym z takich interfejsów jest **CommandLineRunner**. Posiada on metodę **run**, która jest uruchamiana na starcie aplikacji.

```
1 import lombok.extern.slf4j.Slf4j;
2 import org.springframework.boot.CommandLineRunner;
3 import org.springframework.stereotype.Component;
4
5 @Component
6 @Slf4j
7 public class HelloWorldLogger implements CommandLineRunner {
8     @Override
9     public void run(final String... args) throws Exception {
10         log.info("Hello from one of our first spring components");
11     }
12 }
```

# @CONFIGURATION @BEAN

**Stereotypy pozwalają na zdefiniowanie klasy jako bean.** Czasami, dla pewnych funkcjonalności potrzebujemy zdefiniować kilka powiązanych ze sobą beanów (wykorzystując inne istniejące już obiekty). W takim przypadku można umieścić ich definicje w pojedynczej klasie. **Klasa taka powinna być oznaczona adnotacją @Configuration.** W definicji takiej klasy powinny znaleźć się metody, które zwracają instancje obiektów, które trafią do „puli zależności” (czyli do kontekstu). Każda z takich metod **musi być dodatkowo oznaczona adnotacją @Bean.**



# @CONFIGURATION @BEAN

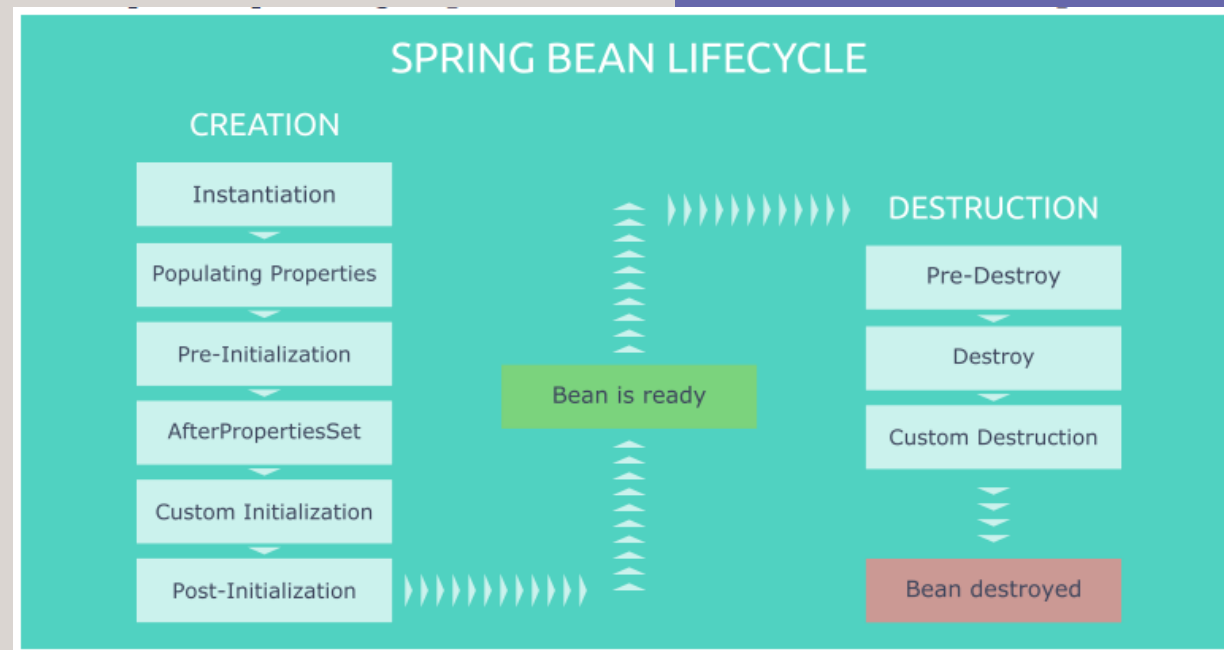
Obiekty tworzone w metodach oznaczonych adnotacją **@Bean** tworzymy „ręcznie” (tzn. za pomocą **operatora new** czy na przykład wykorzystując **wzorce konstrukcyjne**).

**UWAGA:** Klasa oznaczona adnotacją **@Configuration** jest również komponentem.

```
1 import com.fasterxml.jackson.annotation.JsonInclude;
2 import com.fasterxml.jackson.databind.JavaType;
3 import com.fasterxml.jackson.databind.ObjectMapper;
4 import com.fasterxml.jackson.databind.type.TypeFactory;
5 import com.fasterxml.jackson.databind.util.Converter;
6 import org.springframework.context.annotation.Bean;
7 import org.springframework.context.annotation.Configuration;
8
9 @Configuration
10 public class SimpleWebConfiguration {
11
12     @Bean
13     public ObjectMapper objectMapper() {
14         final ObjectMapper objectMapper = new ObjectMapper();
15         objectMapper.setSerializationInclusion(JsonInclude.Include.NON_NULL);
16         return objectMapper;
17     }
18
19     @Bean
20     public Converter<Object, Object> simpleObjectConverter() {
21         return new Converter<Object, Object>() {
22             @Override
23             public Object convert(final Object o) {
24                 // implementacja metody
25                 return null;
26             }
27
28             @Override
29             public JavaType getInputType(final TypeFactory typeFactory) {
30                 // implementacja metody
31                 return null;
32             }
33
34             @Override
35             public JavaType getOutputType(final TypeFactory typeFactory) {
36                 // implementacja metody
37                 return null;
38             }
39         };
40     }
41 }
```

# BEAN – CYKL ŻYCIA

W ramach **cyklu życia komponentu** Spring możemy wyróżnić wiele faz, **kreacyjnych** (konstrukcyjnych) oraz **destrukcyjnych**.





# BEAN – CYKL ŻYCIA

Istnieje wiele możliwości oraz sposobów konfiguracji cyklu życia komponentów Spring. Wśród nich można wymienić:

- implementację interfejsu **InitializingBean** (impl. metody `afterPropertiesSet()`),
- implementację interfejsu **DisposableBean** (impl. metody `destroy()`),
- wykorzystywanie adnotacji JSR-250: **@PostConstruct**, **@PreDestroy**,
- wykorzystanie właściwości **initMethod** oraz **destroyMethod** w ramach adnotacji `@Bean`
- implementacja interfejsu **BeanPostProcessor**,
- implementacja interfejsów grupy **Aware**



# INITIALIZINGBEAN DISPOSABLEBEAN

```
@Component
class MySpringBean implements InitializingBean {

    @Override
    public void afterPropertiesSet() {
        //...
    }

}
```

```
@Component
class MySpringBean implements DisposableBean {

    @Override
    public void destroy() {
        //...
    }

}
```

# POSTCONSTRUCT PREDESTROY

```
@Component
class MySpringBean {

    @PostConstruct
    public void postConstruct() {
        //...
    }

    @PreDestroy
    public void preDestroy() {
        //...
    }
}
```

# BEAN - WŁAŚCIWOŚCI

```
@Configuration
class MySpringConfiguration {

    @Bean(initMethod = "onInitialize", destroyMethod = "onDestroy")
    public MySpringBean mySpringBean() {
        return new MySpringBean();
    }
}
```

# BEANPOSTPROCESSOR

```
class MyBeanPostProcessor implements BeanPostProcessor {

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName)
        throws BeansException {
        //...
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName)
        throws BeansException {
        //...
        return bean;
    }
}
```



# INTERFEJSY AWARE

```
@Component
class MySpringBean implements BeanNameAware, ApplicationContextAware {

    @Override
    public void setBeanName(String name) {
        //...
    }

    @Override
    public void setApplicationContext(ApplicationContext applicationContext)
        throws BeansException {
        //...
    }
}
```

# KOLEJNOŚĆ WYKONYWANIA

```
class MySpringBean implements BeanNameAware, ApplicationContextAware,
    InitializingBean, DisposableBean {

    private String message;

    public void sendMessage(String message) {
        this.message = message;
    }

    public String getMessage() {
        return this.message;
    }

    @Override
    public void setBeanName(String name) {
        System.out.println("--- setBeanName executed ---");
    }

    @Override
    public void setApplicationContext(ApplicationContext applicationContext)
        throws BeansException {
        System.out.println("--- setApplicationContext executed ---");
    }

    @PostConstruct
    public void postConstruct() {
        System.out.println("--- @PostConstruct executed ---");
    }

    @Override
    public void afterPropertiesSet() {
        System.out.println("--- afterPropertiesSet executed ---");
    }

    public void initMethod() {
        System.out.println("--- init-method executed ---");
    }

    @PreDestroy
    public void preDestroy() {
        System.out.println("--- @PreDestroy executed ---");
    }

    @Override
    public void destroy() throws Exception {
        System.out.println("--- destroy executed ---");
    }

    public void destroyMethod() {
        System.out.println("--- destroy-method executed ---");
    }

}
```

# WSTRZYKIWANIE ZALEŻNOŚCI

Założmy, że mamy zdefiniowane klasy **ClassA**, **ClassB**, **ClassC** i **ClassD**.

```
1 public class ClassA { /* definicja klasy */ }
2 public class ClassB { /* definicja klasy */ }
3 public class ClassC { /* definicja klasy */ }
4 public class ClassD { /* definicja klasy */ }
```

Ze wszystkich tych klas korzysta klasa **ClassE**:

```
1 public class ClassE {
2     private final ClassA classA;
3     private final ClassB classB;
4     private final ClassC classC;
5     private final ClassD classD;
6
7     public ClassE(final ClassA classA,
8                  final ClassB classB,
9                  final ClassC classC,
10                 final ClassD classD) {
11         this.classA = classA;
12         this.classB = classB;
13         this.classC = classC;
14         this.classD = classD;
15     }
16 }
```



# WSTRZYKIWANIE ZALEŻNOŚCI – BRAK DI

Chcąc **stworzyć instancję klasy classE** bez wykorzystywania mechanizmu wstrzykiwania musimy stworzyć wszystkie zależne klasy ręcznie, a następnie przekazać je jako argumenty do konstruktora:

```
1 public static void main(String[] args) {  
2     final ClassA classA = new ClassA();  
3     final ClassB classB = new ClassB();  
4     final ClassC classC = new ClassC();  
5     final ClassD classD = new ClassD();  
6  
7     final ClassE classE = new ClassE(classA, classB, classC, classD);  
8 }
```

Podjęcie przedstawione powyżej jest bardzo niewygodne i błędogenne. Lepszym sposobem jest zrzucenie odpowiedzialności za tworzenie obiektów z zależnościami na pewien mechanizm **dependency injection**.



# SPOSOBY WSTRZYKIWANIA ZALEŻNOŚCI

Chcąc zachować zasady **SOLID** i dobre praktyki programowania staramy się **dzielić funkcjonalność na mniejsze, osobne klasy**. W pewnym momencie podczas developmentu programista chce jednak wykorzystać stworzone i zarejestrowane już beany. **W tym celu możemy wykorzystać adnotację @Autowired.**

Spring oferuje trzy sposoby wstrzykiwania zależności:

- **poprzez konstruktor**
- **za pomocą setera**
- **bezpośrednio do pola klasy**



# WSTRZYKIWANIE PRZY POMOCY KONSTRUKTORA

Wstrzykiwanie przy pomocy konstruktora jest preferowanym sposobem wykorzystywania mechanizmu DI, jaki oferuje Spring. Musimy jednak pamiętać, że:

- **wstrzykiwać możemy jedynie beany (tzn. klasy w kontekście aplikacji)**
- **wstrzykiwać beany możemy jedynie do innych beanów**

UWAGA: Zależności klasy często definiujemy jako pol **modyfikatorem final**.



```
1 import org.springframework.stereotype.Component;
2
3 @Component
4 public class Dependency {
5 }
```

```
1 import org.springframework.stereotype.Component;
2
3 @Component
4 public class ClassWithDependency {
5
6     private final Dependency dependency;
7
8     // inject poprzez konstruktor, zależność (bean) jest argumentem konstruktora
9     public ClassWithDependency(final Dependency dependency) {
10         this.dependency = dependency;
11     }
12 }
```



# WSTRZYKIWANIE PRZY POMOCY SETTERA

Wstrzykiwanie **za pomocą settera jest rzadziej wykorzystywanym mechanizmem DI w Springu**. Aby wstrzyknąć zależność za pomocą settera musi on być dodatkowo oznaczony adnotacją **@Autowired**:

```
1 import org.springframework.stereotype.Component;
2
3 @Component
4 public class Dependency {
5 }
6
```

```
1 import org.springframework.beans.factory.annotation.Autowired;
2 import org.springframework.stereotype.Component;
3
4 @Component
5 public class ClassWithDependency {
6
7     private Dependency dependency;
8
9     @Autowired
10    public void setDependency(final Dependency dependency) {
11        this.dependency = dependency;
12    }
13 }
```

# WSTRZYKIWANIE DO POLA

Trzecim sposobem wstrzykiwania zależności jest

**bezpośrednie wstrzykiwanie ich po pola klasy.**

Wymaga on wykorzystania adnotacji **@Autowired** nad polem klasy. **Sposób ten jest często wykorzystywany w testach, które wykorzystują mechanizmy Springa.**

```
1 import org.springframework.stereotype.Component;
2
3 @Component
4 public class DependencyA {
5 }
```

```
1 import org.springframework.stereotype.Component;
2
3 @Component
4 public class DependencyB {
5 }
6
```

```
1 import org.springframework.beans.factory.annotation.Autowired;
2 import org.springframework.stereotype.Component;
3
4 @Component
5 public class ClassWithDependencies {
6
7     @Autowired
8     private DependencyA dependencyA;
9
10    @Autowired
11    private DependencyB dependencyB;
12 }
13
```

# GŁÓWNA KLASA APLIKACJI

Tworząc aplikację bazową opartą w Spring Boota, możemy zauważyć, że zawsze zawiera ona klasę z metodą **main**, która oznaczona jest adnotacją **@SpringBootApplication**.

Najczęściej nie zmieniamy zawartości tej klasy.

# METODA MAIN

W projektach opartych o Spring Boota, **metoda main zazwyczaj posiada jedną linijkę:**

```
1 SpringApplication.run(SbApplication.class, args)
```

W zależności od dependencji, jakie mamy w pliku **pom.xml**, powyższa linijka kodu może robić co innego. Na przykład, gdy w zależnościach znajdzie się **starter spring-boot-starter-web**, aplikacja zostanie domyślnie **uruchomiona na serwerze Tomcat** (i porcie **8080**).



# ADNOTACJA @SPRINGBOOTAPPLICATION

Adnotacja @SpringBootApplication zapewnia aplikacji kilka funkcjonalności. Wynikają one z tego, że sama **adnotacja @SpringBootApplication jest adnotacją składającą się z kilku innych**. Są to m.in.:

- **@SpringBootConfiguration** - działa tak samo jak adnotacja **@Configuration**
- **@EnableAutoConfiguration** - włącza mechanizm automatycznej konfiguracji. Dzięki jej istnieniu aplikacja tworzy pewien kontekst na podstawie **konfiguracji i zależności w pliku pom.xml**
- **@ComponentScan**



# ADNOTACJA @COMPONENTSCAN

Adnotacja **@ComponentScan** jest adnotacją informującą, w jakich pakietach powinny być szukane beany, które trafią do kontekstu. Domyślnie adnotacja ta w poszukiwaniu komponentów, skanuje **aktualną paczkę (i paczki w jej wnętrzu)**. To znaczy, dla klasy oznaczonej adnotacją `@SpringBootApplication` w pakiecie **pl.sagesacademy.sb**, do kontekstu trafią wszystkie klasy, których nazwa pakietu **rozpoczyna się od pl.sagesacademy.sb**.

```
1 package pl.sagesacademy.sb;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class SbApplication {
8
9     public static void main(String[] args) {
10         SpringApplication.run(SbApplication.class, args);
11     }
12 }
```



# ADNOTACJA @COMPONENTSCAN

Obiekt poniższej klasy **nie trafi do kontekstu**:

```
1 package pl.sagescademy; // nazwa paczki nie rozpoczyna się od pl.sagescademy.sb;  
2  
3 import org.springframework.stereotype.Component;  
4  
5 @Component  
6 public class OutOfContextClass {  
7 }  
8
```

Problemy takie jak powyższy możemy rozwiązać dodając adnotację **@ComponentScan** nad dowolną klasą, która jest oznaczona adnotacją **Configuration** (a więc również **@SpringBootApplication**). W polu **basePackages** możemy wskazać paczki, które powinny zostać użyte do szukania w nich beanów.

```
1 @SpringBootApplication  
2 @ComponentScan(basePackages = "pl.sagesacademy")  
3 public class SbApplication {  
4  
5     public static void main(String[] args) {  
6         SpringApplication.run(SbApplication.class, args);  
7     }  
8 }
```

# ZAAWANSOWANE WSTRZYKIWANIE ZALEŻNOŚCI




# ZALEŻNOŚCI CYKLICZNE

Wykorzystując mechanizm **Dependency Injection** możemy przez przypadek wprowadzić tzw. **zależność cykliczną**, tzn. taką, która powoduje, że pewnych komponentów nie da się stworzyć, ponieważ do jego stworzenia jest potrzebny komponent, którego również nie da się stworzyć. Problem taki można wprowadzić już za pomocą dwóch klas (np. klasa A wymaga klasy B, a klasa B wymaga klasy A).

```
1 import org.springframework.stereotype.Component;
2
3 @Component
4 public class ClassAThatRequiresC {
5
6     private final ClassCThatRequiresB classCThatRequiresB;
7
8     public ClassAThatRequiresC(final ClassCThatRequiresB classCThatRequiresB) {
9         this.classCThatRequiresB = classCThatRequiresB;
10    }
11 }
12
13 import org.springframework.stereotype.Component;
14
15 @Component
16 public class ClassBThatRequiresA {
17
18     private final ClassAThatRequiresC classAThatRequiresC;
19
20     public ClassBThatRequiresA(final ClassAThatRequiresC classAThatRequiresC) {
21         this.classAThatRequiresC = classAThatRequiresC;
22     }
23 }
24
25 import org.springframework.stereotype.Component;
26
27 @Component
28 public class ClassCThatRequiresB {
29
30     private final ClassBThatRequiresA classBThatRequiresA;
31
32     public ClassCThatRequiresB(final ClassBThatRequiresA classBThatRequiresA) {
33         this.classBThatRequiresA = classBThatRequiresA;
34     }
35 }
```

# ZALEŻNOŚCI CYKLICZNE

W przypadku, gdy taki problem nastąpi, aplikacja nie wystartuje z odpowiednią wiadomością. W tym przypadku:

The dependencies of some of the beans in the application context form a cycle: 

```
graph TD
    A["classAThatRequiresC defined in file [...ClassAThatRequiresC.class]"]
    B["classBThatRequiresA defined in file [...ClassBThatRequiresA.class]"]
    C["classCThatRequiresB defined in file [...ClassCThatRequiresB.class]"]
    A --> C
    C --> B
    B --> A
```

Najczęściej, rozwiązaniem takiego problem jest **stworzenie dodatkowego komponentu lub komponentów**, do których wydzielamy logikę potrzebną do „zerwania” cyklu.



# INTERFEJSY

W rzeczywistych aplikacjach wstrzykiwane klasy mogą **implementować pewien interfejs**. Spring pozwala również wstrzyknąć implementację interfejsu, jeżeli implementacja ta znajduje się w kontekście.

```
1 public interface SimpleLogger {  
2  
3     void printMessage(String message);  
4 }  
5
```

```
9 import org.springframework.stereotype.Component;  
10  
11 // komponent zarejestrowany w kontekście implementujący interfejs,  
12 // który będzie wstrzykiwany  
13 @Component  
14 public class SimpleConsoleLogger implements SimpleLogger {  
15  
16     @Override  
17     public void printMessage(final String message) {  
18         System.out.println("Hello from component that implements interface");  
19     }  
20 }
```

```
1 import org.springframework.boot.CommandLineRunner;  
2 import org.springframework.stereotype.Component;  
3  
4 @Component  
5 public class HelloWorldLogger implements CommandLineRunner {  
6  
7     private final SimpleLogger simpleLogger;  
8  
9     // wstrzykiwanie poprzez konstruktor z wykorzystaniem interfejsu  
10     public HelloWorldLogger(final SimpleLogger simpleLogger) {  
11         this.simpleLogger = simpleLogger;  
12     }  
13  
14     @Override  
15     public void run(final String... args) throws Exception {  
16         simpleLogger.printMessage("Hello from command line runner");  
17     }  
18 }
```

# NAZWY KOMPONENTÓW

Mechanizm **dependency injection** w Springu bazuje na **nazwach komponentów w celu ich wstrzykiwania**.

Domyślnie, jeżeli wykorzystujemy stereotypy, nazwa klasy staje się nazwą komponentu, ale jego pierwsza litera zawsze mała. Istnieje również możliwość nadania innej nazwy komponentu, za pomocą **pola value** w adnotacji.

- Klasa `ExampleSpringComponent` oznaczona adnotacją `@Service`, będzie miała nazwę **exampleSpringComponent**
- Klasa `AnotherSpringComponent` oznaczona adnotacją `@Component("customName")`, będzie miała nazwę podaną w adnotacji, tzn. **customName**



# NAZWY KOMPONENTÓW

W przypadku tworzenia beanów za pomocą adnotacji **@Bean**, nazwa beanu jest **nazwą metody**, która została wykorzystana do jego stworzenia. Stąd też nazwy te najczęściej **nie spełniają wszystkich dobrych reguł nazewnictwa metod**. Tak jak w przypadku stereotypów, tak również w przypadku wykorzystania adnotacji @Bean, istnieje możliwość wskazania nazwy beanu (**za pomocą pola name**).

```
1 import com.fasterxml.jackson.annotation.JsonInclude;
2 import com.fasterxml.jackson.databind.ObjectMapper;
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.Configuration;
5
6 @Configuration
7 public class SimpleWebConfiguration {
8
9     // wykorzystywana nazwa to "sagesObjectMapper".
10    // Nadpisuje domyślną nazwę - "objectMapper"
11    @Bean(name = "sagesObjectMapper")
12    public ObjectMapper objectMapper() {
13        final ObjectMapper objectMapper = new ObjectMapper();
14        objectMapper.setSerializationInclusion(JsonInclude.Include.NON_NULL);
15        return objectMapper;
16    }
17 }
```

# WSTRZYKIWANIE Z WYKORZYSTANIEM NAZWY

Ponieważ Spring pozwala na wstrzykiwanie zależności z wykorzystaniem interfejsu, jaki ta zależność implementuje, może się zdarzyć, że wstrzyknięcie konkretnej implementacji może nie być możliwe, **ponieważ w kontekście znajdziemy kilka beanów, które taki interfejs implementują**. W takim przypadku otrzymamy błąd i aplikacja się nie uruchomi. W przypadku próby uruchomienia takiej aplikacji otrzymamy następujący błąd:

```
1 Description:
2
3 Parameter 0 of constructor in pl.sagesacademy.sb.HelloWorldLogger required a single bean, but 2 were found:
4   - simpleConsoleLogger: defined in file [...\classes\pl\sagesacademy\sbsimpleConsoleLogger.class]
5   - simpleLombokLogger: defined in file [...\classes\pl\sagesacademy\sbsimpleLombokLogger.class]
```

```
1 public interface SimpleLogger {
2     void printMessage(String message);
3 }
4
5
6 import org.springframework.stereotype.Component;
7
8 @Component
9 public class SimpleConsoleLogger implements SimpleLogger {
10
11     @Override
12     public void printMessage(final String message) {
13         System.out.println("Hello from component that implements interface");
14     }
15 }
16
17
18 @Slf4j
19 @Component
20 public class SimpleLombokLogger implements SimpleLogger {
21
22     @Override
23     public void printMessage(final String message) {
24         log.info("Hello from Lombok Logger: " + message);
25     }
26 }
```

```
1 import org.springframework.boot.CommandLineRunner;
2 import org.springframework.stereotype.Component;
3
4 @Component
5 public class HelloWorldLogger implements CommandLineRunner {
6
7     private final SimpleLogger simpleLogger;
8
9     // wstrzykiwanie poprzez konstruktor z wykorzystaniem interfejsu
10    public HelloWorldLogger(final SimpleLogger simpleLogger) {
11        this.simpleLogger = simpleLogger;
12    }
13
14    @Override
15    public void run(final String... args) throws Exception {
16        simpleLogger.printMessage("Hello from command line runner");
17    }
18 }
```



# KORZYSTANIE Z @QUALIFIER

Adnotacja **@Qualifier** służy do wskazania nazwy wstrzykiwanego komponentu. Wykorzystujemy ją przy wstrzykiwanym obiekcie, tzn.:

- **przy wstrzykiwanym obiekcie** w przypadku wstrzykiwania obiektu **przy pomocy konstruktora** lub setera
- **nad wstrzykiwanym polem** w przypadku wstrzykiwania **bezpośrednio do pola**

**UWAGA:** IntelliJ Ultimate podkreśli nazwę użytą wewnątrz adnotacji @Qualifier w przypadku wykorzystania nazwy nieistniejącego beanu.

```
1 import org.springframework.beans.factory.annotation.Qualifier;
2 import org.springframework.boot.CommandLineRunner;
3 import org.springframework.stereotype.Component;
4
5 @Component
6 public class HelloWorldLogger implements CommandLineRunner {
7
8     private final SimpleLogger simpleLogger;
9
10    // wstrzykiwanie poprzez konstruktor z wykorzystaniem interfejsu i adnotacji @Qualifier
11    public HelloWorldLogger(@Qualifier("simpleLombokLogger") final SimpleLogger simpleLogger) {
12        this.simpleLogger = simpleLogger;
13    }
14
15    @Override
16    public void run(final String... args) throws Exception {
17        simpleLogger.printMessage("Hello from command line runner");
18    }
19 }
```

# BEANY PRIMARY

W przypadku posiadania wielu beanów tego samego typu w kontekście możemy jeden z nich oznaczyć dodatkowo adnotacją **@Primary**. Adnotacja ta powoduje, że w przypadku istnienia wielu beanów danego typu (tzn. implementującego konkretny interfejs) i **niewykorzystania adnotacji @Qualifier**, zostanie wybrany ten oznaczony jako główny.

```
1 public interface SimpleLogger {
2     void printMessage(String message);
3 }
4
5
6 import lombok.extern.slf4j.Slf4j;
7 import org.springframework.context.annotation.Bean;
8 import org.springframework.context.annotation.Configuration;
9 import org.springframework.context.annotation.Primary;
10
11 @Configuration
12 @Slf4j
13 public class SimpleLoggersConfiguration {
14
15     @Bean
16     @Primary
17     public SimpleLogger verySimpleLogger() {
18         return System.out::println;
19     }
20
21     @Bean
22     public SimpleLogger lombokLogger() {
23         return log::info;
24     }
25 }
```

```
1 import org.springframework.boot.CommandLineRunner;
2 import org.springframework.stereotype.Component;
3
4 @Component
5 public class HelloWorldLogger implements CommandLineRunner {
6
7     private final SimpleLogger simpleLogger;
8
9     // wstrzyknięty zostanie verySimpleLogger
10    public HelloWorldLogger(final SimpleLogger simpleLogger) {
11        this.simpleLogger = simpleLogger;
12    }
13
14    @Override
15    public void run(final String... args) throws Exception {
16        simpleLogger.printMessage("Hello from command line runner");
17    }
18 }
```

# WSTRZYKIWANIE DO METODY

Zdarzają się przypadki, kiedy w celu stworzenia jednego beana za pomocą adnotacji **@Bean**, potrzebujemy drugiego. W takim przypadku bean taki możemy wstrzyknąć, **przyjmując go jako argument metody (nie musimy wykorzystywać żadnej adnotacji)**.

**UWAGA:** Spring daje możliwość wstrzykiwania obiektów (wielu) to wielu konkretnych metod. Metody te najczęściej opisane są w dokumentacji konkretnego projektu.

```
1 public interface SimpleLogger {
2     void printMessage(String message);
3 }
4
5
6 import org.springframework.stereotype.Component;
7 import java.text.DateFormat;
8
9 @Component
10 public class DateFormatProvider {
11
12     public DateFormat get() {
13         return DateFormat.getDateInstance();
14     }
15 }
```

```
1 import com.fasterxml.jackson.databind.ObjectMapper;
2 import org.springframework.context.annotation.Bean;
3 import org.springframework.context.annotation.Configuration;
4
5 @Configuration
6 public class SimpleLoggerConfiguration {
7
8     // metoda ta wstrzykuje zależny bean konieczny
9     // do stworzenia obiektu ObjectMapper
10     @Bean
11     public ObjectMapper objectMapper(final DateFormatProvider dateFormatProvider) {
12         final ObjectMapper objectMapper = new ObjectMapper();
13         objectMapper.setDateFormat(dateFormatProvider.get());
14         return objectMapper;
15     }
16 }
```

# KONFIGURACJA APLIKACJI

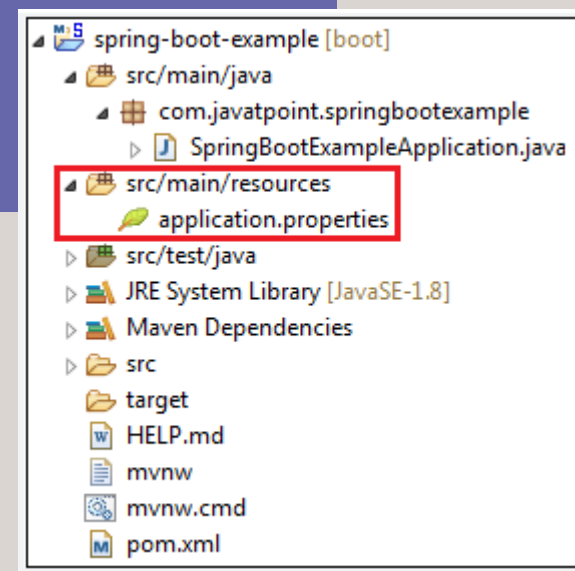


Spring oferuje mechanizm **autokonfiguracji**, ale istnieje również możliwość dostosowywania postaci istniejących beanów do potrzeb aplikacji oraz **tworzyć własne właściwości konfiguracyjne**.

# APPLICATION.PROPERTIES

Tworząc bazowy projekt, w katalogu **resources** znajdziemy plik o nazwie **application.properties**. Plik ten służy do konfigurowania aplikacji za pomocą pewnych właściwości, którym możemy przypisać wartość.

Klucze dostępne w pliku **application.properties** zależą od wykorzystanych zależności. Pozwalają one dostosować **postać beana** lub **przekazać do aplikacji pewną właściwość**. Najczęściej dostępne klucze znajdziemy w dokumentacji danego projektu (np. startera).



# APPLICATION.PROPERTIES

Właściwości w pliku **application.properties** pisane są według następujących reguł:

- poszczególne wyrazy **oddzielone są od siebie kropkami i pisane małymi literami**
- jeżeli pewna podgrupa właściwości składa się z wielu słów, to wykorzystujemy **konwencję kebab-case**
- klucz od wartości **oddzielony jest znakiem =**
- jeżeli klucz zaczyna się od słowa spring, to najczęściej jest to **właściwość konfiguracyjna oficjalnego startera**



# APPLICATION.PROPERTIES

Przykład **klucza** wykorzystujący przedstawione konwencje: **spring.datasource.hikari.jdbc-url**

**UWAGA:** W przypadku wykorzystywania IDE IntelliJ Ultimate, dostępne klucze powinny zostać podpowiedziane w przypadku wypełniania pliku application.properties.

**UWAGA:** Plik application.properties może mieć również tzw. **format YAML**. W takim przypadku plik ten nazywa się **application.yml**.

```
--- !clarkevans.com/^invoice
invoice: 34843
date : 2001-01-23
bill-to: &id001
  given : Chris
  family : Dumars
  address:
    lines: |
      458 Walkman Dr.
      Suite #292
    city : Royal Oak
    state : MI
    postal : 48046
ship-to: *id001
product:
  - sku : BL394D
    quantity : 4
    description : Basketball
    price : 450.00
  - sku : BL4438H
    quantity : 1
    description : Super Hoop
    price : 2392.00
tax : 251.42
total: 4443.52
comments: >
  Late afternoon is best.
  Backup contact is Nancy
  Billsmer @ 338-4338.
```

**SCALAR**

**COLLECTIONS**

**MULTI-LINE COLLECTIONS**

**LISTS/DICTIONARIES**

**MULTI-LINE FORMATTING**

# ZMIENNE ŚRODOWISKOWE

Przekazując gotową aplikację klientowi (lub np. obraz **dockerowy dev-opsom**) musi istnieć możliwość **nadpisania** wartości użytych w pliku **application.properties**. Możemy to zrobić za pomocą tzw. mechanizmu **relaxed binding**. Dzięki niemu każdy klucz **można zastąpić zmienną środowiskową**, w której każda litera jest wielka, a **kropka zastąpiona jest znakiem \_**

application.properties	zmienna środowiskowa
server.port	SERVER_PORT
spring.h2.console.path	SPRING_H2_CONSOLE_PATH





# WSTRZYKIWANIE WŁAŚCIWOŚCI

Framework Spring daje również możliwość **wstrzykiwania własnych, specyficznych dla aplikacji właściwości**. Służy do tego adnotacja **@Value**. Klucz, który umieszczamy wewnątrz adnotacji, powinien spełniać takie same warunki jak te opisane poprzednio. Właściwości takie możemy **wstrzykiwać bezpośrednio do pola klasy lub przy pomocy konstruktora klasy, która znajduje się w kontekście**.

```
1 import org.springframework.stereotype.Component;
2
3 @Component
4 public class AnotherBean {
5 }
```

```
8 import org.springframework.beans.factory.annotation.Value;
9 import org.springframework.stereotype.Component;
10
11 @Component
12 public class SomeBean {
13
14     @Value("${pl.sagesacademy.example}")
15     private String injectedValue;
16
17     private final AnotherBean anotherBean;
18
19     public SomeBean(final AnotherBean anotherBean) {
20         this.anotherBean = anotherBean;
21     }
22     // pozostałe metody
23
24 }
25
26 // w pliku application.properties:
27 // pl.sagesacademy.example=hi
```

```
1 import org.springframework.beans.factory.annotation.Value;
2 import org.springframework.stereotype.Component;
3
4 @Component
5 public class SomeBean {
6
7     private final String injectedValue;
8     private final AnotherBean anotherBean;
9
10    public SomeBean(final AnotherBean anotherBean,
11                    @Value("${pl.sagesacademy.example:default_value}") final String injectedValue) {
12        this.anotherBean = anotherBean;
13        this.injectedValue = injectedValue;
14    }
15
16    public String getInjectedValue() {
17        return injectedValue;
18    }
19 }
```

# GRUPOWANIE WŁAŚCIWOŚCI

W przypadku, gdy do aplikacji chcemy wstrzyknąć wiele właściwości **możemy to zrobić wykorzystując wiele razy adnotację @Value**. Jednakże lepszym pomysłem jest wykorzystanie adnotacji **@ConfigurationProperties**. Adnotację tę wykorzystujemy **nad klasą, która reprezentuje grupę właściwości** (najczęściej ściśle ze sobą powiązanych). Wymaga ona podania od nas **wspólnego prefiksu dla wszystkich pól**. Ponadto, aby klasa ta była widoczna w kontekście aplikacji, powinna definiować **getter** i **setter** oraz być użyta z jedną z adnotacji: **@Component** lub **@EnableConfigurationProperties(PropertyClass.class)**.

```
1 import lombok.AllArgsConstructor;
2 import lombok.Data;
3 import lombok.NoArgsConstructor;
4 import org.springframework.boot.context.properties.ConfigurationProperties;
5 import org.springframework.boot.context.properties.EnableConfigurationProperties;
6 import org.springframework.stereotype.Component;
7
8
9 @Component // lub //@EnableConfigurationProperties(PropertiesGroup.class)
10 @ConfigurationProperties(prefix = "pl.sagesacademy.group")
11 @Data
12 @NoArgsConstructor
13 @AllArgsConstructor
14 public class PropertiesGroup {
15     private String propA;
16     private Integer propB;
17 }
18
19 /* w pliku application.properties
20 pl.sagesacademy.group.prop-a=prop_A_value
21 pl.sagesacademy.group.prop-b=17
22 */
```

# GRUPOWANIE WŁAŚCIWOŚCI

Klasy oznaczone adnotacją **@ConfigurationProperties**, mogą również **zawierać kolekcje**. W takim przypadku:

- **dla list** wykorzystujemy na końcu nazwy właściwości zapis tablicowy - indeks wewnątrz znaków [ i ]
- **dla map** po pełnej nazwie właściwości stawiamy znak ., a następnie podajemy nazwę klucza, któremu przypisujemy wartość

```
1 import lombok.AllArgsConstructor;
2 import lombok.Data;
3 import lombok.NoArgsConstructor;
4 import org.springframework.boot.context.properties.ConfigurationProperties;
5 import org.springframework.stereotype.Component;
6
7 import java.util.List;
8 import java.util.Map;
9
10 @Component
11 @ConfigurationProperties(prefix = "sages.collections")
12 @Data
13 @NoArgsConstructor
14 @AllArgsConstructor
15 public class ConfigurationWithCollections {
16
17     private List<String> usernames;
18     private Map<String, Integer> values;
19 }
```

```
1 sages.collections.usernames[0]=user1
2 sages.collections.usernames[1]=user2
3 sages.collections.values.key1=1
4 sages.collections.values.key2=2
```

# PROFILE

Kolejnym mechanizmem, bardzo często wykorzystywanym w praktyce, ze względu na elastyczność, jest **możliwość definiowania profili**. Profile w aplikacji dają możliwość szybkiego **przełączania się pomiędzy gotowymi konfiguracjami**.



# PROFILE

Profil jest zwykłym ciągiem znaków i daje możliwość:

- **stworzenia specyficznego dla profilu pliku `application.properties`**
- **stworzenie specyficznych dla profilu beanów**




# APPLICATION.PROPERTIES A PROFILE

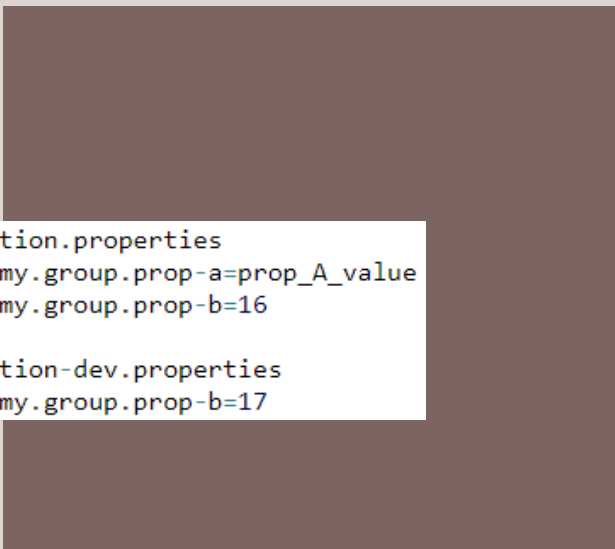
Każdy profil umożliwia tworzenie specyficznego dla tego profilu **pliku konfiguracyjnego**. Plik ten powinien znajdować się w zasobach projektu i mieć nazwę:

- **application-NAZWAPROFILU.properties**

Co ważne, plik ten **nadpisuje wartości znajdujące się w pliku domyślnym application.properties**, ale gdy pewien klucz ma wartość w **application.properties** ale nie ma w **application-NAZWAPROFILU.properties**, to wtedy wartość zostanie **pobrana z pliku application.properties**.



```
1 #plik application.properties
2 pl.sagesacademy.group.prop-a=prop_A_value
3 pl.sagesacademy.group.prop-b=16
4
5 #plik application-dev.properties
6 pl.sagesacademy.group.prop-b=17
```



# BEANY A PROFILE

Istnieje również **możliwość zdefiniowania beanu**, który będzie stworzony jedynie, gdy aplikacja zostanie uruchomiona w konkretnym profilu. Służy do tego **adnotacja @Profile**, którą łączymy z adnotacjami **@Component** czy **@Bean** i wewnątrz której umieszczamy:

- **nazwę profilu**, w którym bean powinien zostać stworzony
- **nazwę profilu poprzedzoną znakiem !**. Dzięki temu bean zostanie stworzony dla każdego innego profilu niż podany.

```
1 import java.text.DateFormat;
2
3 public interface DateFormatProvider {
4     DateFormat get();
5 }
6
```

```
8 import org.springframework.context.annotation.Profile;
9 import org.springframework.stereotype.Component;
10
11 import java.text.DateFormat;
12 import java.text.SimpleDateFormat;
13
14 @Profile("dev")
15 @Component
16 public class DevDateFormatProvider implements DateFormatProvider {
17     @Override
18     public DateFormat get() {
19         return new SimpleDateFormat();
20     }
21 }
```

```
24 import com.fasterxml.jackson.databind.util.StdDateFormat;
25 import org.springframework.context.annotation.Profile;
26 import org.springframework.stereotype.Component;
27
28 import java.text.DateFormat;
29
30 @Profile("!dev") // dla wszystkich profili oprócz "dev"
31 @Component
32 public class ProdDateFormatProvider implements DateFormatProvider {
33     @Override
34     public DateFormat get() {
35         return new StdDateFormat();
36     }
37 }
```

# URUCHAMIANIE APLIKACJI Z PROFILAMI

W celu **uruchomienia danej aplikacji z konkretnym profilem (lub profilami)** powinniśmy tę informację przekazać w jakiś sposób do aplikacji. Jeżeli chcemy uruchomić aplikację **wykorzystując wiele profili**, powinniśmy ich **nazwy oddzielić od siebie przecinkami**.

Możemy to zrobić na wiele sposobów, na przykład:

- możemy ustawić profil w zmiennej środowiskowej `SPRING_PROFILES_ACTIVE`, tzn. w linii komend wykonać:

```
export SPRING_PROFILES_ACTIVE=nazwa_profilu  
java -jar sb-0.0.1-SNAPSHOT.jar # lub inny jar reprezentujący zbudowaną aplikację opartą o spring boot
```



# URUCHAMIANIE APLIKACJI Z PROFILAMI

- możemy przekazać jego wartość do wirtualnej maszyny w argumencie o nazwie **spring.profiles.active**. Na przykład, wykorzystując IDE IntelliJ, możemy w konfiguracji uruchamiania w polu **VM options** podać: -  
**Dspring.profiles.active=nazwa\_profilu**
- gdy posiadamy IntelliJ Ultimate, możemy otworzyć okno konfiguracji uruchamiania (tzw. **Run Configurations**) i w polu **Active Profiles** podać nazwy profili

```
1 2020-09-25 21:17:27.076 INFO 12628 --- [main] ... : The following profiles are active: dev
```

# SCOPE

Na tym etapie, wiemy już, że to **nie programista jest odpowiedzialny za tworzenie komponentów, a kontener IoC.**

Domyślnie Spring tworzy **po jednej kopii zdefiniowanych przez nas komponentów.**

Oznacza to, że tworzone instancje klas oznaczone adnotacjami @Component, @Repository, @Service czy @Controller są **singletonami.**

# SCOPE

Czasami istnieje jednak potrzeba stworzenia **kilku kopii danego komponentu**. Oznacza to, że chcemy zmienić jego tzw. **scope**. Spring umożliwia zmianę scope'u za pomocą adnotacji **@Scope**, a możliwe wartości, które podajemy wewnątrz adnotacji to:

- **singleton** - domyślna i najczęściej wykorzystywana wartość
- **prototype** - powoduje stworzenie **nowej instancji klasy w przypadku, gdy kontener zostanie poproszony o dostarczenie komponentu**



# SCOPE

Czasami istnieje jednak potrzeba stworzenia **kilku kopii danego komponentu**. Oznacza to, że chcemy zmienić jego tzw. **scope**. Spring umożliwia zmianę scope'u za pomocą adnotacji **@Scope**, a możliwe wartości, które podajemy wewnątrz adnotacji to:

- **singleton** - domyślna i najczęściej wykorzystywana wartość
- **prototype** - powoduje stworzenie **nowej instancji klasy w przypadku, gdy kontener zostanie poproszony o dostarczenie komponentu**



# SCOPE

Kolejne wartości, są możliwe do wykorzystania, gdy stworzymy **aplikację webową**, tzn. wykorzystujemy, np. starter **spring-boot-starter-web**. Wartości stałych znajdują się wewnątrz klasy **WebApplicationContext**:

- **request** - tworzy nową kopię beana dla każdego żądania HTTP
- **session** - tworzy nową kopię beana dla każdej nowej sesji HTTP
- **application**
- **websocket**



# SCOPE

Oprócz możliwości podania stałej bezpośrednio do adnotacji **@Scope**, możemy również wykorzystać adnotacje, dla których wartość wynika z jej nazwy, tzn.:

- **@RequestScope**
- **@SessionScope**
- **@ApplicationScope**

```
1 public class MyBean {  
2  
3     public MyBean() {  
4         System.out.println("MyBean instance created");  
5     }  
6  
7 }  
8
```

```
10 import org.springframework.context.annotation.Bean;  
11 import org.springframework.context.annotation.Configuration;  
12 import org.springframework.context.annotation.Scope;  
13  
14 @Configuration  
15 public class MyConfiguration {  
16  
17     @Bean  
18     @Scope(value="singleton")  
19     public MyBean myBean() {  
20         return new MyBean();  
21     }  
22  
23 }
```

```
28 import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
29  
30 public class MySpringApp {  
31  
32     public static void main(String[] args) {  
33         AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();  
34         ctx.register(MyConfiguration.class);  
35         ctx.refresh();  
36  
37         MyBean mb1 = ctx.getBean(MyBean.class);  
38         System.out.println(mb1.hashCode());  
39  
40         MyBean mb2 = ctx.getBean(MyBean.class);  
41         System.out.println(mb2.hashCode());  
42  
43         ctx.close();  
44     }  
45  
46 }
```

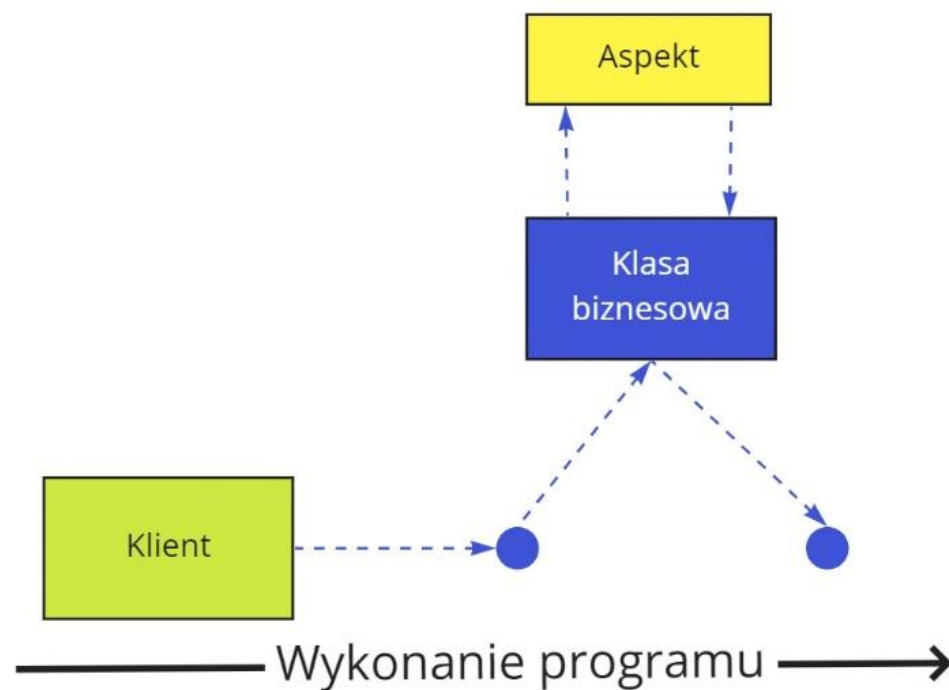
# AOP

Programowanie **zorientowane aspektowo** (w springu jest to moduł **Spring AOP**) to sposób programowania, który pozwala oddzielić pewne fragmenty kodu od siebie, poprawiając dzięki temu **modularność** tworzonego programu. **Programowanie aspektowe** pozwala wprowadzić dodatkową warstwę do kodu, która jest wyraźnie odseparowana od właściwego kodu.

# AOP

Programowanie aspektowe (**AOP**) uzupełnia **programowanie obiektowe (OOP)**, dostarczając innego sposobu myślenia o strukturze programu. Kluczową jednostką modularności programowania obiektowego jest klasa, podczas gdy w AOP **jednostką modularności jest aspekt**. Aspekty umożliwiają modularyzację zagadnień (takich jak zarządzanie transakcjami), które dotyczą wielu typów i obiektów.

**AOP jest paradygmatem programowania**, który ma na celu zwiększenie modularności. Czyni to poprzez dodawanie dodatkowych zachowań do istniejącego kodu bez modyfikowania samego kodu.





# AOP

Spring IoC, aby zapewnić wydajne rozwiązania.

**AOP w Springu** jest używany m.in. do:

- **dostarczania deklaratywnych usług** – najważniejszą taką usługą jest **zarządzanie transakcjami**.
- pozwala użytkownikom implementować niestandardowe aspekty, **uzupełniając ich użycie OOP**
  - o **AOP**.

# AOP – KONCEPT W SPRINGU

- **JoinPoint** - jest punktem podczas wykonywania programu, takim jak **wykonanie metody** lub **obsługa wyjątku**. W Spring AOP, **JoinPoint** zawsze **reprezentuje wykonanie metody**.
- **Pointcut** - jest predykatem, który pomaga dopasować poradę (**Advice**), która ma być zastosowana przez aspekt w konkretnym **JoinPoincie**.
- **Advice** - jest akcją podejmowaną przez aspekt w konkretnym JoinPoincie



# AOP – RODZAJE ADVICE

- **Before** - porada, która działa przed JoinPointem, ale która nie ma możliwości zatrzymania wykonania programu
- **After returning** - porada, którą należy uruchomić po normalnym zakończeniu JoinPointu
- **After throwing** - porada, którą należy uruchomić, jeśli metoda zakończy działanie rzucając wyjątek
- **After (finally) advice** - porada, którą należy uruchomić niezależnie od sposobu wykonania
- **Around advice** – porada, która otacza JoinPoint, taki jak wywołanie metody

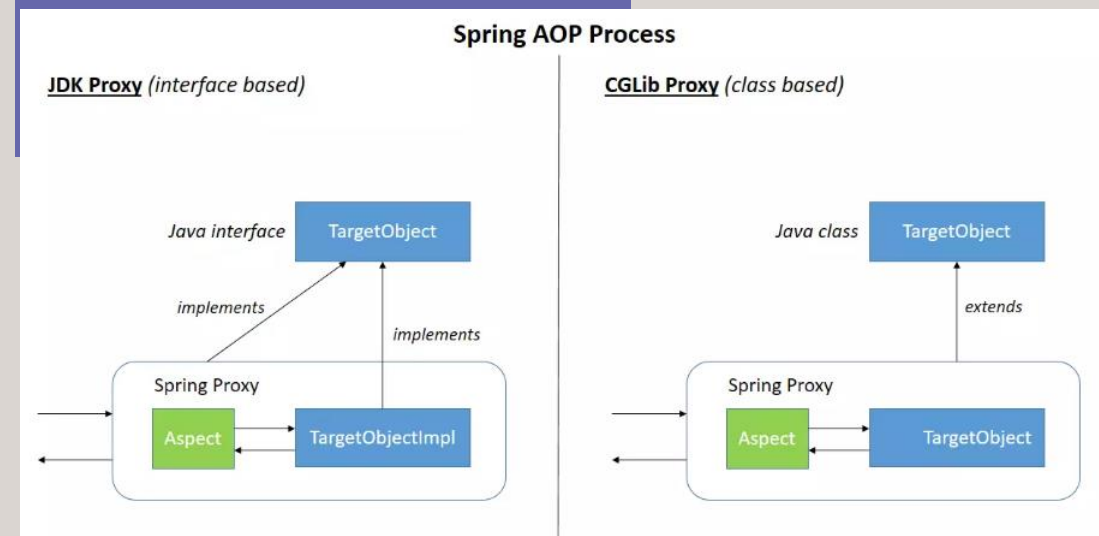


# MECHANIZM PROXY

Spring AOP do utworzenia proxy dla danego obiektu docelowego używa albo:

- **dynamicznych proxy JDK**
- albo **CGLIB**

Jeśli obiekt docelowy, który ma być „proxowany”, **implementuje przynajmniej jeden interfejs**, używane jest **dynamiczne proxy JDK**. Natomiast jeżeli obiekt docelowy nie implementuje żadnych interfejsów, tworzone jest **proxy przez CGLIB**.



# AOP - PRZYKŁAD

```
1 @Component
2 public class BusinessClass{
3     public void validateTransaction() {
4         System.out.println("...validating");
5     }
6 }
```

```
1 @Aspect
2 @Component
3 public class LoggingAspect {
4
5     @Pointcut("execution(* BusinessClass.validateTransaction(..))")
6     public void logAfter() {
7         //pointcut
8     }
9
10    @Before("execution(* BusinessClass.validateTransaction())") //point-cut expression
11    public void logBefore(JoinPoint joinPoint) {
12        System.out.println("BusinessClass.logBefore() : " + joinPoint.getSignature().getName());
13    }
14
15    @AfterReturning(pointcut = "logAfter()")
16    public void logujWyjscieZUslugi(JoinPoint joinPoint) {
17        System.out.println("BusinessClass.afterRetuning() : " + joinPoint.getSignature().getName());
18    }
19 }
```

```
1 BusinessClass.logBefore() : validateTransaction
2 ...validating
3 BusinessClass.afterRetuning() : validateTransaction
4
```

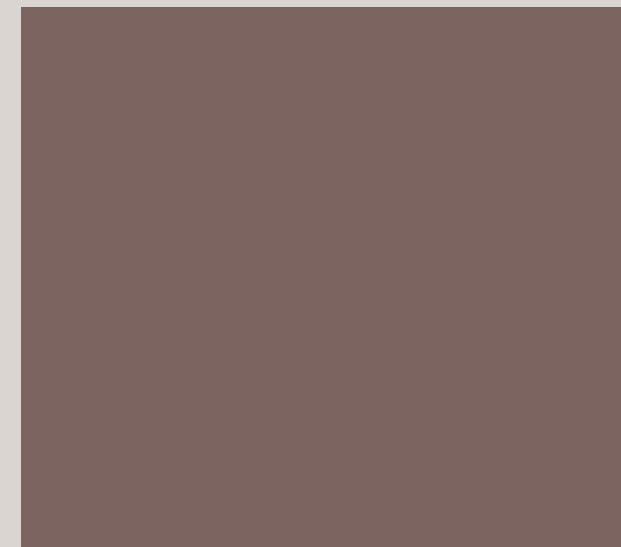
# PROGRAMOWANIE PRZEZ ZDARZENIA



# MECHANIZM PROXY

Zarówno zdarzenia jak również bezpośrednie wywołania metod są dedykowane odpowiednim sytuacjom.

W przypadku programowania przez zdarzenia, istotny jest dla nas fakt wystąpienia danego „eventu”. **Nie jesteśmy zainteresowani informacją, który dokładnie moduł przejmie odpowiedzialność związaną z obsługą danego zdarzenia.** Najczęściej, przypadki te dotyczą zdarzeń asynchronicznych, których wykonanie nie powinno mieć wpływu na główny przebieg programu (np. zdarzenie związane z wysyłką wiadomości e-mail).



# APPLICATION EVENT

Zdarzenia są **nośnikiem informacji pomiędzy komponentami, które nie są związane ze sobą w sposób bezpośredni**. Ze względu na fakt, iż obiekty pełniące rolę producenta (ang. publisher) nie wchodzi w bezpośrednią interakcję z obiektami konsumenta, możliwe jest modyfikowanie kontraktu (API) obu komponentów.

Począwszy od wersji 4.2 Springa, zdarzenia **nie muszą rozszerzać klasy ApplicationEvent**.

```
1 class UserCreatedEvent extends ApplicationEvent {  
2     private String name;  
3  
4     UserCreatedEvent(Object source, String name) {  
5         super(source);  
6         this.name = name;  
7     }  
8     ...  
9 }
```

```
12 class UserRemovedEvent {  
13     private String name;  
14  
15     UserRemovedEvent(String name) {  
16         this.name = name;  
17     }  
18     ...  
19 }
```



# PUBLIKOWANIE ZDARZEŃ

```
1 @Component
2 class Publisher {
3
4     private final ApplicationEventPublisher publisher;
5
6     Publisher(ApplicationEventPublisher publisher) {
7         this.publisher = publisher;
8     }
9
10    void publishEvent(final String name) {
11        // Publishing event created by extending ApplicationEvent
12        publisher.publishEvent(new UserCreatedEvent(this, name));
13        // Publishing an object as an event
14        publisher.publishEvent(new UserRemovedEvent(name));
15    }
16 }
```

# NASŁUCHIWANIE NA ZDARZENIA

Istnieją dwa sposoby definiowania „nasłuchiacza zdarzeń” w Springu. Możemy posłużyć się adnotacją **@EventListener** lub zaimplementować interfejs **ApplicationListener**. W obu przypadkach, komponent odpowiedzialny za nasłuchiwanie zdarzeń musi być komponentem zarządzanym przez framework Spring.



# @EVENTLISTENER

**Typy zdarzeń** mogą być określone w ramach parametrów adnotacji @EventListener. Przykładowo:

```
1 @EventListener({ContextStartedEvent.class, ContextRefreshedEvent.class}).
```

Dodatkowo, Spring umożliwia obsługę zdarzeń tylko w określonych okolicznościach. Odpowiednie warunki możemy zdefiniować przy pomocy **SpEL** wykorzystując właściwość **condition**.

```
1 @Component
2 class UserRemovedListener {
3
4     @EventListener
5     ReturnedEvent handleUserRemovedEvent(UserRemovedEvent event) {
6         // handle UserRemovedEvent ...
7         return new ReturnedEvent();
8     }
9
10    @EventListener
11    void handleReturnedEvent(ReturnedEvent event) {
12        // handle ReturnedEvent ...
13    }
14    ...
15 }
```

```
1 @Component
2 class UserRemovedListener {
3
4     @EventListener(condition = "#event.name eq 'reflectoring'")
5     void handleConditionalListener(UserRemovedEvent event) {
6         // handle UserRemovedEvent
7     }
8 }
```

# APPLICATIONLISTENER

Innym sposobem obsługi zdarzenia jest implementacja interfejsu **ApplicationListener**.

```
1 @Component
2 class UserCreatedListener implements ApplicationListener<UserCreatedEvent> {
3
4     @Override
5     public void onApplicationEvent(UserCreatedEvent event) {
6         // handle UserCreatedEvent
7     }
8 }
```

# ASYNCHRONICZNE ZDARZENIA

Domyślnie wszystkie zdarzenia są przetwarzane przez Springa w **sposób synchroniczny**, tzn. główny wątek **oczekuje na ukończenie** obsługi danego zdarzenia przez wszystkich zarejestrowanych „nasłuchiwczy”. Chcąc przetwarzać dane zdarzenia w sposób asynchroniczny, powinniśmy posłużyć się adnotacją **@Async**. Dodatkowo, w klasie konfiguracyjnej powinniśmy użyć adnotacji **@EnableAsync**.

```
1 @Component
2 class AsyncListener {
3
4     @Async
5     @EventListener
6     void handleAsyncEvent(String event) {
7         // handle event
8     }
9 }
```

# OBSŁUGA TRANSAKCJI – PRZYKŁAD ZDARZEŃ

Możliwe fazy:

- **AFTER\_COMMIT**: obsługa zdarzenia po zatwierdzeniu transakcji
- **AFTER\_COMPLETION**: obsługa zdarzenia po zatwierdzeniu lub wycofaniu transakcji
- **AFTER\_ROLLBACK**: obsługa zdarzenia po wycofaniu transakcji
- **BEFORE\_COMMIT**: obsługa zdarzenia przed zatwierdzeniem transakcji

```
1 @Component
2 class UserRemovedListener {
3
4     @TransactionalEventListener(phase=TransactionPhase.AFTER_COMPLETION)
5     void handleAfterUserRemoved(UserRemovedEvent event) {
6         // handle UserRemovedEvent
7     }
8 }
```

# SPRING – PREDEFINIOWANE ZDARZENIA

Spring dostarcza kilka predefiniowanych zdarzeń związanych z cyklem życia aplikacji. Najczęściej, w tym przypadku, odpowiednie obiekty nasłuchujące na te zdarzenia rejestrujemy w sposób manualny.

Przykładowe zdarzenia:

- **ApplicationStartingEvent**
- **ApplicationEnvironmentPreparedEvent**
- **ApplicationContextInitializedEvent**
- **ContextRefreshedEvent**
- **ApplicationStartedEvent**
- **ApplicationFailedEvent**

```
1 @SpringBootApplication
2 public class EventsDemoApplication {
3
4     public static void main(String[] args) {
5         SpringApplication springApplication =
6             new SpringApplication(EventsDemoApplication.class);
7         springApplication.addListeners(new SpringBuiltInEventsListener());
8         springApplication.run(args);
9     }
10 }
```

```
1 class SpringBuiltInEventsListener
2     implements ApplicationListener<SpringApplicationEvent>{
3
4     @Override
5     public void onApplicationEvent(SpringApplicationEvent event) {
6         // handle event
7     }
8 }
```

DZIĘKUJĘ ZA UWAGĘ

