Spring jest frameworkiem, który powstał z myślą o uproszczeniu developmentu z wykrozystaniem języka JAVA.

Aby tego dokonać Spring stosuje cztery kluczowe strategie:

* Lekki i mało inwazyjny sposób tworzenia aplikacji z wykorzystaniem obiektów POJO (Plain Old Java Object),
* Wolne wiązanie elementów aplikacji poprzez wykorzystanie DI and interface orientation,
* Programowanie deklaratywne poprzez wykorzystanie aspektów i przyjętych konwencji,
* Usunięcie zbędnego kodu poprzez aspekty i template’y.

**IoC – Inversion of Control** – Odwrócenie sterowania – paradygmat mający na celu zmniejszenie zależności między obiektami.

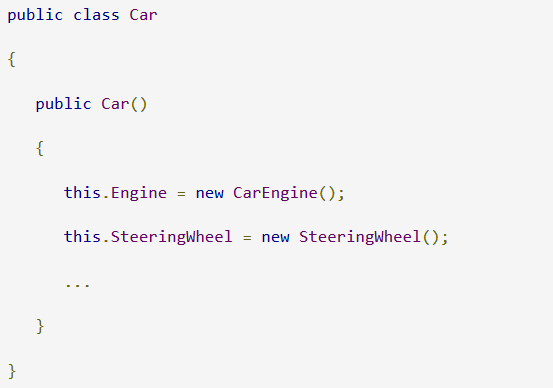
Zasada Hollywood – „Nie dzwoń do nas. My oddzwonimy do Ciebie”

Oznacza to, że każdy z modułów aplikacji nie interesuje się tym skąd przychodzą żądania i dokąd kierowane są odpowiedzi. Wykonuje jedynie postawione przed nim zadania.

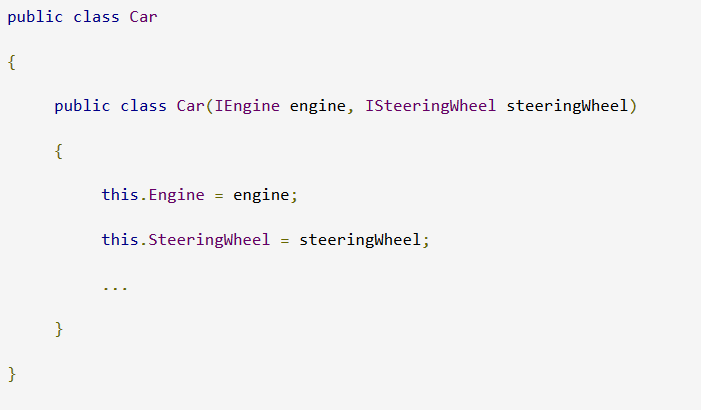
**Dependency Injection** – Obiekty zamiast tworzyć własne zależności lub prosić obiekt fabryki o stworzenie ich, otrzymują je poprzez argument konstruktora lub setery. W ten sposób funkcjonalność obiektów jest niezależna od sposobu tworzenia zależności.

Przykład realizacji IoC poprzez Dependency Injection.

Bez IoC:



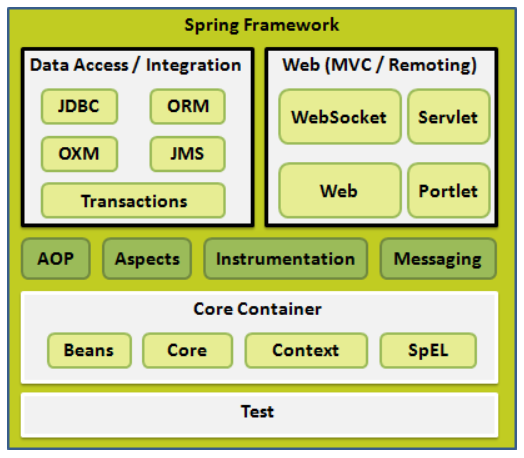
Z IoC:



**Zalety Springa:**

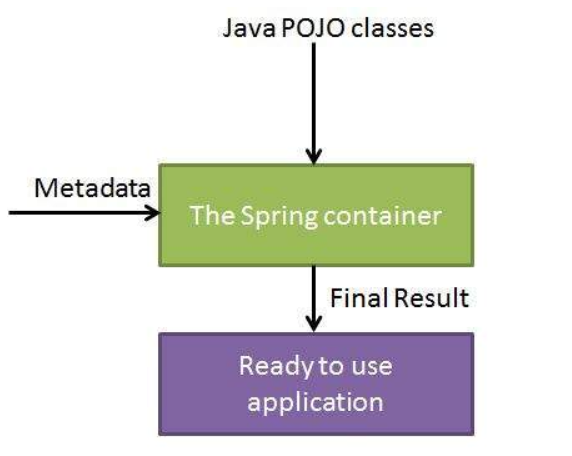
* Spring pozwala na tworzenie aplikacji klasy enterprise używając jedynie obiektów POJO, dzięki czemu nie ma konieczności używania kontenera EJB. Możliwość wykorzystania jedynie prostego kontenera serwletów takiego jak Tomcat.
* Modularność
* Łatwość testowania – ułatwiona dzięki używaniu POJO i DI do wstrzykiwania danych testowych.
* Spring zapewnia dobrze zaprojektowany framework MVC.
* Kontener IoC (Inversion of Control) jest lekki, zwłaszcza w porównaniu do kontenerów EJB co jest najbardziej użyteczne w przypadku komputerów z ograniczoną pamięcią RAM lub zasobami CPU.
* Spring zapewnia łatwy i skalowalny sposób zarządzania transakcjami, od lokalnych transakcji z jedną bazą danych do globalnych transakcji.

**Architektura Springa (modularność):**



* Core Kontener:
  + **Core** – zapewnia fundmanetalną część framewroku takie jak IoC i DI.
  + **Bean module** – zapewnia Fabrykę Bran’ów, będącą implementacją wzorca fabryki.
  + **Context** – zapewnia dostęp do zdefiniowanych i skonfigurowanych obiektów.

Kontener IoC Springa na podstawie obiektów typu POJO i matadanych generuje w pełni skonfigurowaną aplikację.



Dwa główne kontenery:

* Spring BeanFactory Container – zapewnie podstawowe wsparcie dla DI i funkcjonalności zapewniane przez interfejsy z *org.springframework.beans.factory.BeanFactory.* Jest wciąż dostępny głównie ze względu na kompatybilność wsteczną.
* Spring ApplicationContextContainer – posiada więcej funkcjonalności dla aplikacji typu enterprise jak np. delegowanie zdarzeń do konkretnych listner’ów. Zapewnia pełną funkcjonalność kontenera BeanFactory i jest rekomendowany do używania. BeanFactory może być używane dla lekkich aplikacji.

**Konfiguracja Bean’ów:**

* W pliku XML,
* Adnotacje,
* Kod Jave’owy



**Dostępne Bean scopes:**

* singleton
* prototype – wiele instancji
* request http request scope
* session – http session scope
* global-session – global http session scope

W aplikacji Spring’owej definicje bean’ów są ładowane i wiązane razem poprzez *application context*. Spring posiada kilka implementacji contextu aplikacji, różniących się między sobą sposobem ładowania konfiguracji.

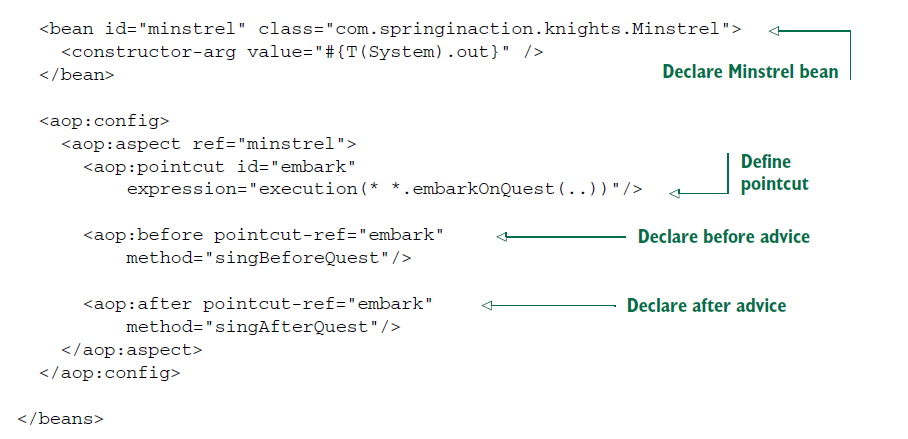
Przykładowo jeśli konfiguracja springowa została zadeklarowana w pliku/plikach xml odpowiednim wyborem dla kontextu aplikacyjnego może być klasa *ClassPathApplicationContext*.



Tutaj w metodzie main() jest tworzony kontext aplikacji na podstawie pliku konfiguracyjnego xml, który następnie jest używany jako fabryka do tworzenia bean’ów.

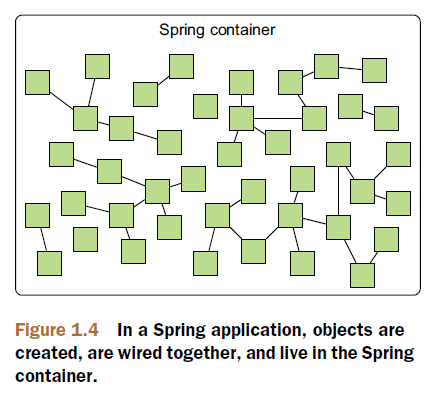
**Aspect Oriented Programming**

AOP pozwala na podzielenie elementów aplikacji, na podstawie ich funkcjonalności w reużywalne komponenty – aspekty. Poprzez odpowiednią konfigurację możliwa jest współpraca różnych aspektów np. aspektu odpowiedzialnego za logikę biznesową z aspektem logującym bez bezpośredniej zależności między ich elementami.



**Spring container**

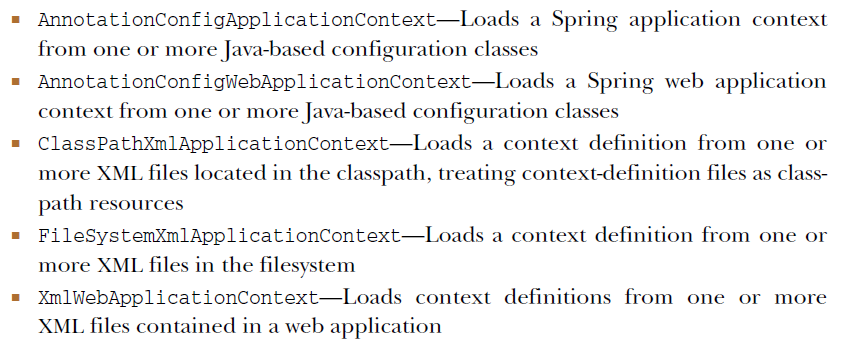
W aplikacji springowej wszystkie obiekty aplikacyjne żyją wewnątrz *Spring kontenera*. Kontener tworzy obiekty, wiąże je, konfiguruje i zarządza ich cyklem życia.



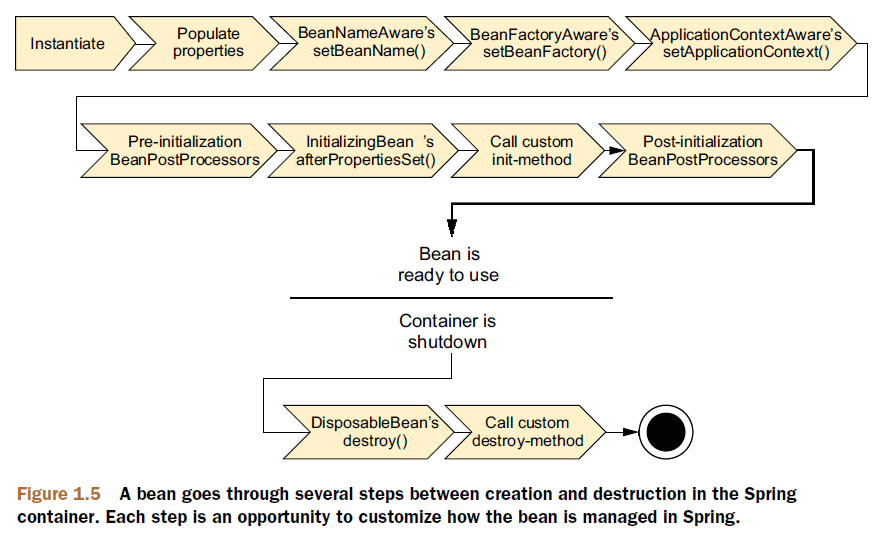
W rzeczywistości nie ma jednego kontenera springowego, a istnieje wiele ich implementacji, które mogą zostać podzielone na dwa typy. *Bean factories* ***(org.springframework.beans.factory.BeanFactory*** *interface)* są najprostrzymi kontenerami, zapewniającymi podstawowe wsparcie dla DI. *Application contexts (****org.springframework.context.ApplicationContext*** *interface)* zbudowane na idei Bean Factory, poprzez zapewnienie dodatkowych usług, takich jak umiejętność odczytywania tekstowych wiadomości z plików i zdolności do publikowania zdarzeń aplikacyjnych do zainteresowanych listener’ów.

Można używać obu rodzajów kontenerów poniekąd zamiennie, jednak Bean factory jest często uważane za zbyt niskopoziomowe dla większości aplikacji. Dlatego application contexts są zazwyczaj preferowane ponad bean factory.

Rodzaje kontextów:

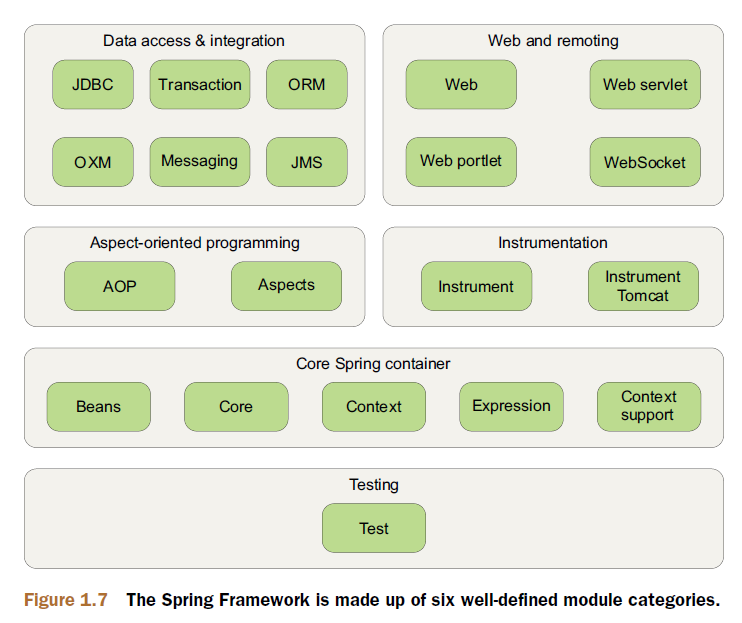


***Cykl życia bean’ów***



**Moduły Spring’a**

Od Spring’a 4.0 Spring Framework składa się z 20 oddzielnych modułów, które można podzielić pod względem funkcjonalności na 6 kategorii.



Core Spring container

Jest centralną częścią Framework’u Spring’owego i odpowiada za zarządzanie tworzenie, konfigurację oraz zarządznie cyklem życia bean’ów. W tym module znajduje się Spring’owy Bean Factory, zapewniający Dependency Injection.

Spring AOP module

Zapewnia wsparcie dla programowania aspektowego.

Data access and integration

API do komunikacji z bazami danych, JMS (Java Message Service), OXM (Object XML Mapping).

Web and remoting

Usługi do komunikacji takie jak RMI (Remote Mehthod Invocation), JAX-WS, Spring’s own http invoker, REST APIs.

Instrumentation

Ten moduł zawiera wsparcie dla dodawania agentów do maszyny wirtualnej Javy.

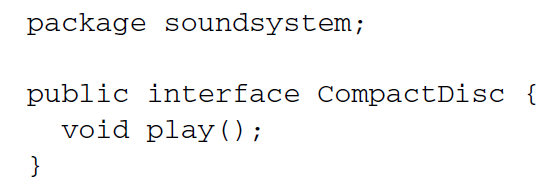
Testing

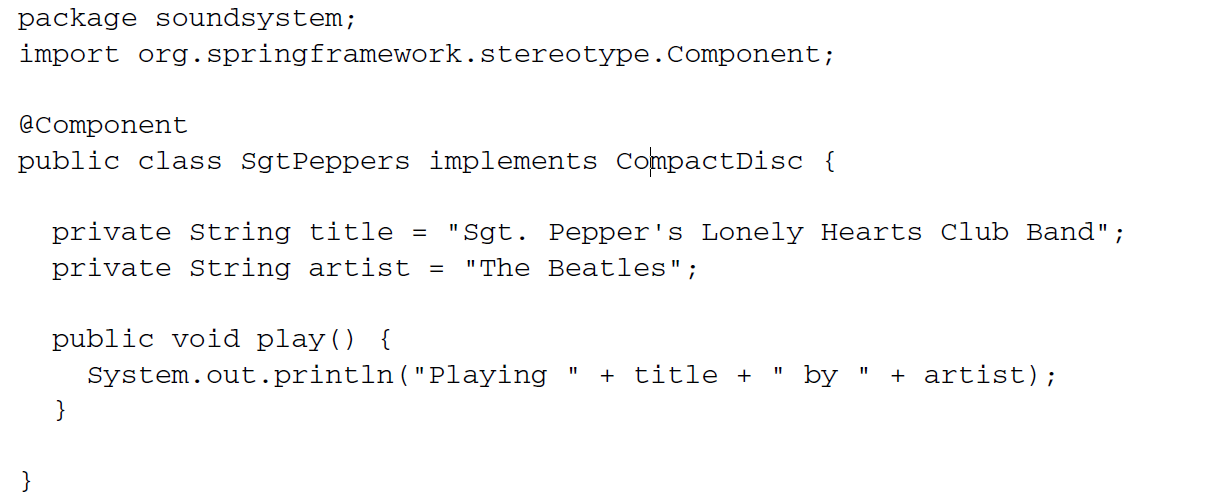
W tym module można znaleźć kolekcje implementacji MOK obiektów do unit testów dla zastosowań wykorzystujących JNDI, servlety, portlety itp.

**Spring portfolio – ciekawe API**

* Spring Web Flow
* Spring Web Services
* Spring Security
* Spring Integration – oferuje implementację kilku często wykorzystywanych patternów integracyjnych
* Spring Batch – do pisania programów wsadowych
* Spring Data – API do komunikacji z bazami danych zarówno relacyjnymi jak i NoSQL
* Spring Social – rozszerzenie do integracji naszych aplikacji z portalami społecznościowymi, ale nie tylko. Ten moduł pomaga połączyć naszą aplikacje z dowolnym API Restowyym.
* Spring Mobile – wsparcie dla tworzenia mobilnych aplikacji webowych.
* Spring For Android – możliwość tworzenia natywnych aplikacji na Androida
* Spring Boot – framework do automatyzacji konfiguracji aplikacji Spring’owych.

**Automatyczne wiązanie bean’ów**



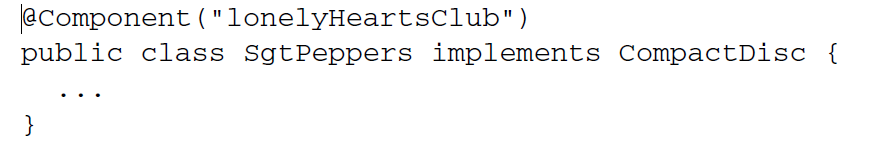
  
**@Component**

Klasa SgtPeppers jest oznaczona annotacją **@Component**, ta prosta annotacja identyfikuje tę klasę jako komponent i służy jako wyznacznik dla Springa, aby stworzył odpowiedniego Bean’a.

Skanowanie komponentów nie jest default’owo włączone i należy je włączyć w odpowiedniej konfiguracji.

Wszystkie bean’y Spring’owe mają przypisane ID. Jeśli nie przypisze się go w sposób wyraźny zostanie mu nadane automatyczne Id na podstawie nazwy klasy np. SgtPeppers dostanie id *sgtPeppers*.

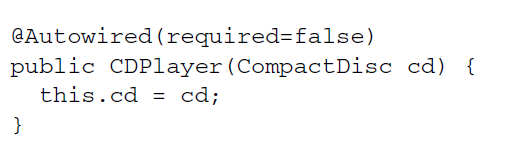
Nadanie własnego id jest możliwe poprzez podanie parametru do annotacji @Component



Zamiennie z annotacją @Component można używać annotacji @Named, która różni się od niej szczegółami i jest gorzej nazwana dlatego jednak lepiej używać @Component.

**@Autowired**

Annotacja mówiąca Springowi, że powinien przeszukać kontener w poszukiwaniu odpowiadającego bean'a i związać go z oznaczonym elementem w sposób automatyczny. Jeśli nie ma pasujących bean'ów, Spring wyrzuci wyjątek podczas tworzenia kontextu aplikacji, aby tego uniknąć można ustawić atrybut required na annotacji @Autowired na false.



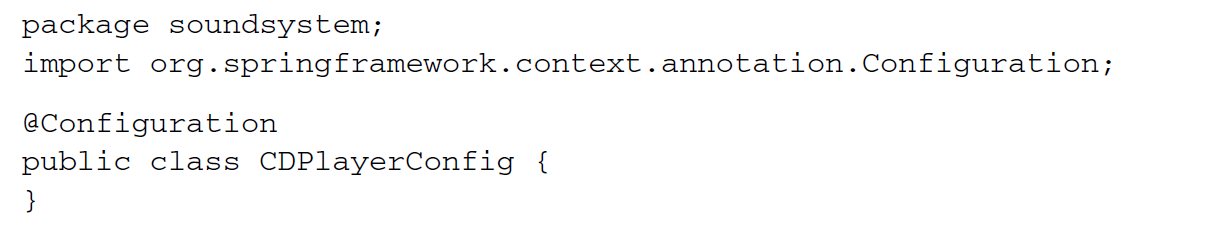
Zamiennie ze Spring'ową annotacją @Autowired można używać annotacji ze specyfikacji Java Dependecy Injection @Inject.

**Ręczna konfiguracja wiązania bean'ów**

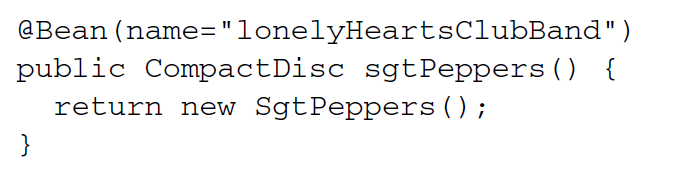
W większości przypadków wiązanie automatyczne jest preferowane, jednak czasami może istnieć konieczność ręcznego skonfigurowania wiązania między bean'ami. Przykładowo, jeśli chcemy w naszej aplikacji wykorzystać komponenty z zewnętrznej biblioteki, do której kodu źródłowego nie mamy dostępu nie będziemy w stanie odpowiednich klas i obiektów oznaczyć annotacjami @Component i @Autowired. Wówczas będzie konieczna konfiguracja ręczna. W takim przypadku mamy dwie możliwości:

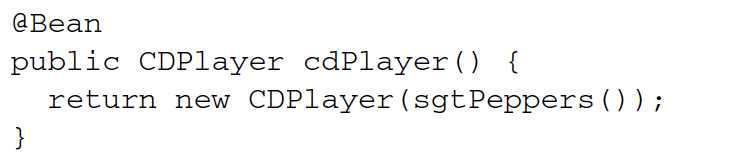
Konfiguracja Jave'owa

Ten typ konfiguracji jest zazwyczaj preferowany ponieważ jest bardziej type-safe oraz refactor-friendly. Należy jednak pamiętać, iż mimo iż jest to kod javovy są to pliki konfiguracyjne i nie powinny być powiązane z logiką biznesową aplikacji.

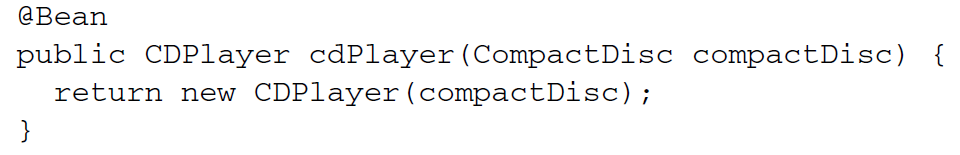


**@Configuration** definiuje daną klasę jako klasę konfiguracyjną Spring'a.



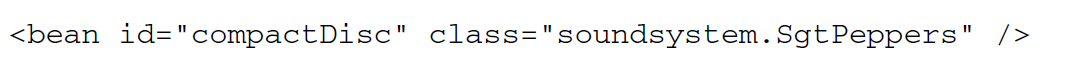


lub



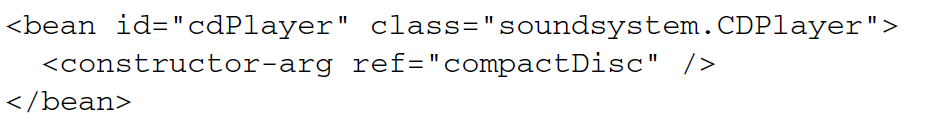
Konfiguracja poprzez XML

Deklaracja beanu:

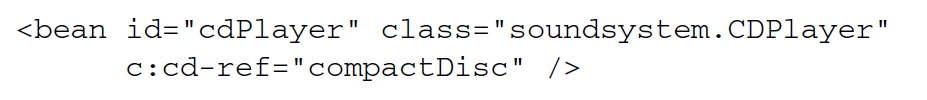


**DI:**

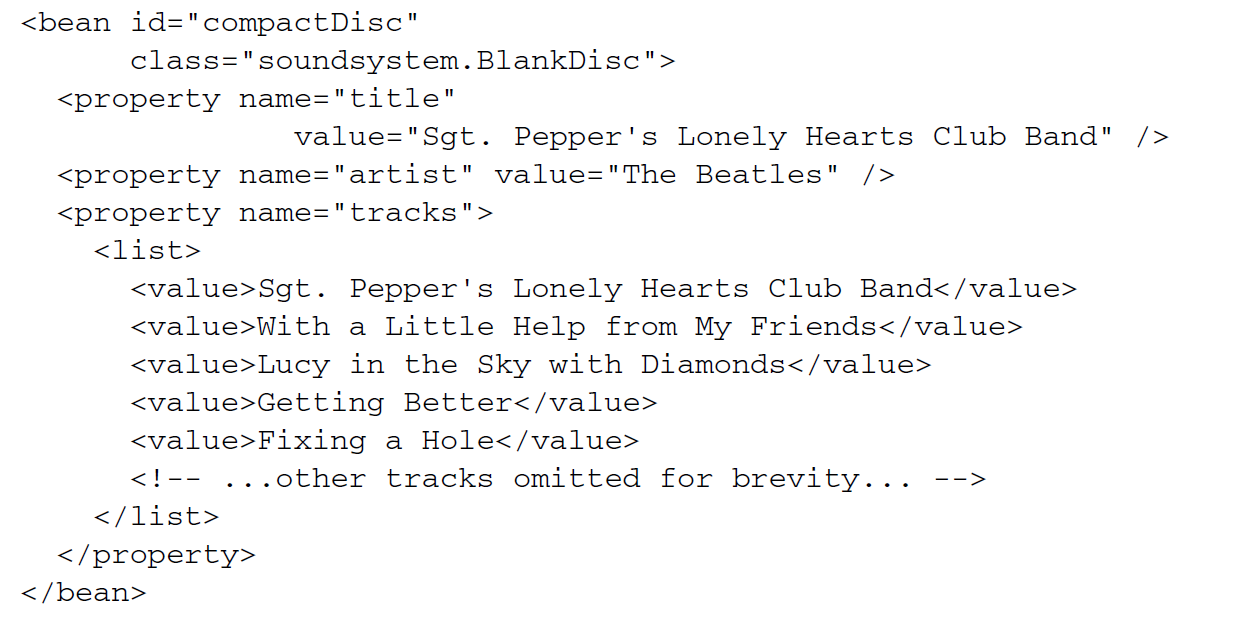
* <constructor-arg> element



* c-namespace (wprowadzone w Spring 3.0)



* setter properties



**Łączenie różnych sposobów konfiguracji**

Żaden ze sposobów konfiguracji nie wyklucza innych. beany powstałe na skutek automatycznego wiązania, konfiguracji Javovej lub XML'owej znajdują się w tym samym kontenerze i są sobie równoważne.

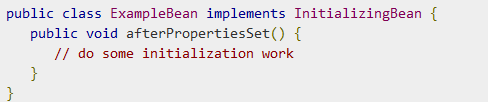
**Inicjalizacja Bean’ów**

1. **W kodzie**

Interfejs org.springframework.beans.factory.InitializingBean specyfikuje jedną metodę

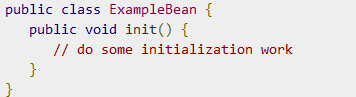


Inicjalizacji można dokonać poprzez implementację tego interfejsu.

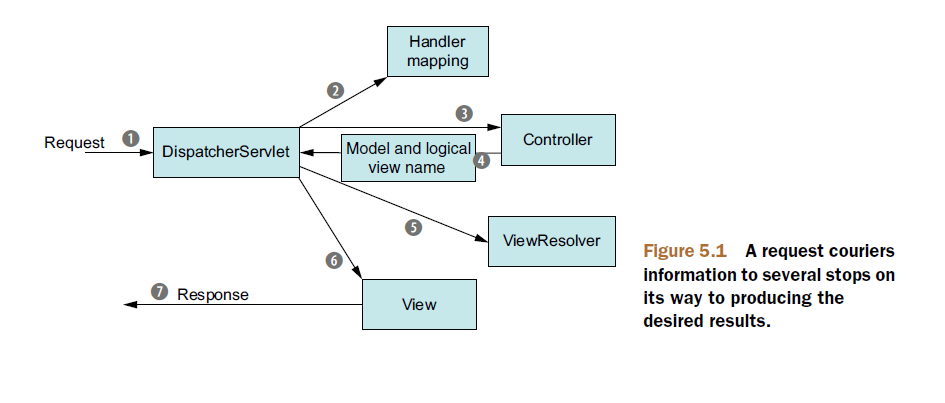


1. **XML – poprzez atrybut init-method**



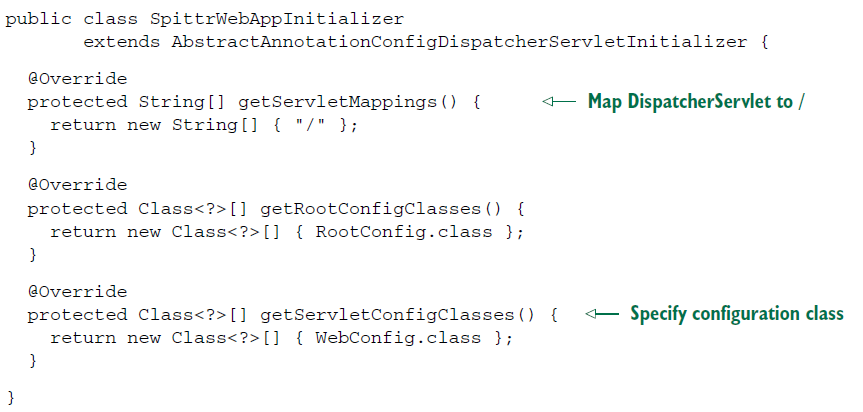


**Spring on the WEB**

****

**Konfiguracja**

**DispatcherServlet**

****

Każda klasa, która rozszerza AbstractAnnotationConfigDispatcherServletInitializer będzie automatycznie używana do konfiguracji DispatcherServlet'u i kontextu aplikacji Springowej w kontenerze servletu.

Metoda getSerrvletMapping() identyfikuje jedną lub więcej ścieżek przypisanych do DispatcherServletu. W tym przypadku jest do niego przypisana ścieżka '/', zatem będzie on aplikacyjnym defaultowym servletem i będzie obsługiwał wszystkie nadchodzące requesty.

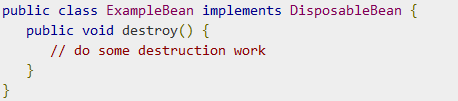
Kiedy DispatcherServlet jest tworzony, tworzy również kontekst aplikacji Spring'owej i rozpoczyna ładowanie bean'ów zadeklarowanych w plikach lub klasach konfiguracyjnych. Poprzez metodę getServletConfigClasses() poprosiliśmy Spring'a o załadowanie do kontekstu bean'ów zdefiniowanych w klasie WebConfig.

Jednak w aplikacji Spring'owej występuje jeszcze jeden rodzaj kontekstu, a mianowicie ContextLoaderListener. Podczas, gdy DispatcherServlet odpowiada za ładowanie bean'ów zawierających komponenty web'owe aplikacji takie jak kontrolery, view resolver'y, etc. ContextLoaderListener odpowiada za załadowanie innych bean'ów aplikacyjnych, odpowiadających zazwyczaj za logikę biznesową aplikacji.

AbstractAnnotationConfigDispatcherServletInitializer tworzy oba konteksty, zarówno DispatcherServlet, jak i ContextLoaderListener. Klasy oznaczone annotacją @Configuration, zwrócone z getServletConfigClasses() będą bean;ami dla kontekstu DispatcherServlet'u. Podczas gdy klasy konfiguracyjne zwrócone przez metodę getRootConfigClasses() będą użyte do konfiguracji kontekstu aplikacji stworzonego przez ContextLoaderListener.

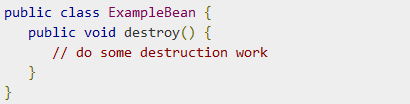
**Desktrukcja Bean’ów**

1. W kodzie – implementacja interfejsu *org.springframework.beans.factory.DisposableBean*

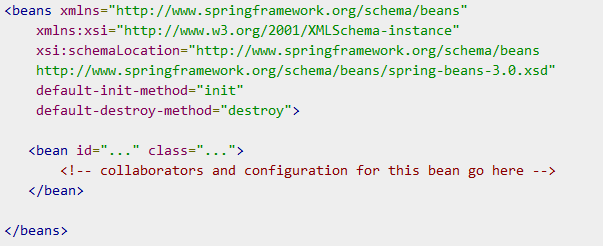


1. XML – poprzez atrubut destroy-method





Defaultowe:



Destroy, a scope:

W przypadku scope=”prototype” nie działa metoda niszcząca. Spring nie zarządza całym cyklem życia beanu prototype. Kontener tworzy instancję, konfiguruje ją, dekoruje i wstrzykuje i dalej nie ma żadnej wiedzy na temat tej instancji.

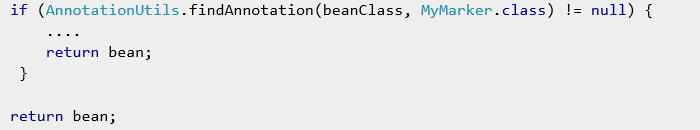
**BeanPostProcessors**

ApplicationContext automatycznie wykrywa wszystkie beany, które zostały zadeklarowane z implementacją interfejsu **BeanPostProcessor.**

Podpięcie pod konkretnego beana przez

instanceof albo bean.getClass().equals(OurBean.class)

lub adnotacja jako marker



**Dziedziczenie Bean’ów**

Deklaracja poprzez atrybut *parent* w konfiguracji dziecka.

**Dependency Injection Przykład**

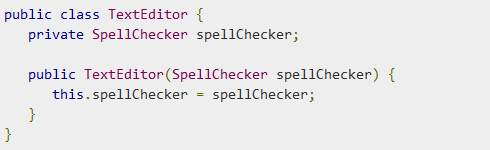
Prosta aplikacja typu edytor tekstowy z opcją sprawdzania pisowni. Potrzebny jest nam więc główny

moduł edytora, a w nim moduł odpowiedzialny za sprawdzanie pisowni.

Bez IoC:



Z IoC:



W ten sposób TextEditora nie interesuje sposób tworzenia, czy konkretna implementacja SpellCheckera. SpellChecker jest tworzony niezależnie i dostarczany do TestEditora w momencie jego tworzenia.

Spring ułatwia to zadanie ponieważ kontroluje całą tę procedurę.

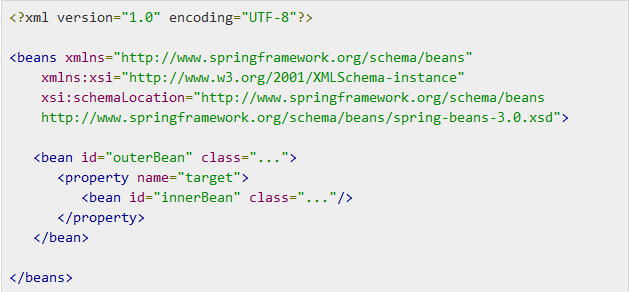
W powyższym przykładzie kontrola została usunięta z TextEditora i przechowywana gdzieś indziej (np. w pliku XML), a zależność jest wstrzykiwana w konstruktrorze. Stąd przepływ sterowania został „odwrócony” poprzez wykorzystanie Wstrzykiwania Zależności (DI) bo zależności zostały oddelegowane do jakiegoś zewnętrznego systemu.

**Drugim sposobem wstrzykiwania zależności jest DI poprzez setery.**

W tym przypadku kontener wywołuje odpowiednie setery bean’ów, po wywołaniu bezargumentowego konstruktora lub statycznej metody fabryki.

**Beany wewnętrzne – Inner Beans**

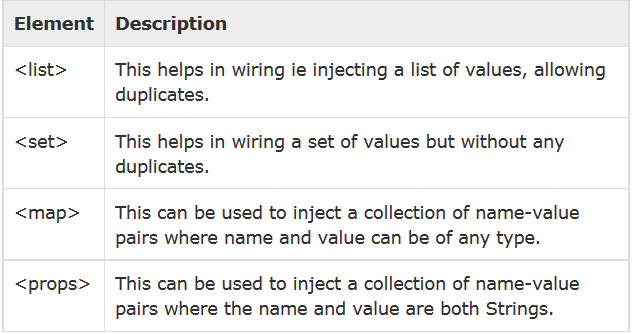
Tak jak klasy wewnętrzne tak samo istnieje możliwość tworzenia wewnętrznych bean’ów. Są to beany zadeklarowane wewnątrz elementów <property> lub <constructor-arg>:



Takie deklarowanie bean’ów jest zalecane zawsze wtedy, gdy dany bean jest własnością tylko jednego Bean’a. innerBean jest własnością tylko i wyłącznie outerBean’a.

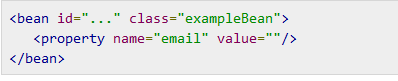
**Wstrzykiwanie kolekcji**

Spring oferuje 4 typy elementów konfiguracyjnych:



Dwa przypadki wstrzykiwanie wartości i wstrzykiwanie referencji.

Wstrzykiwanie pustych Stringów:



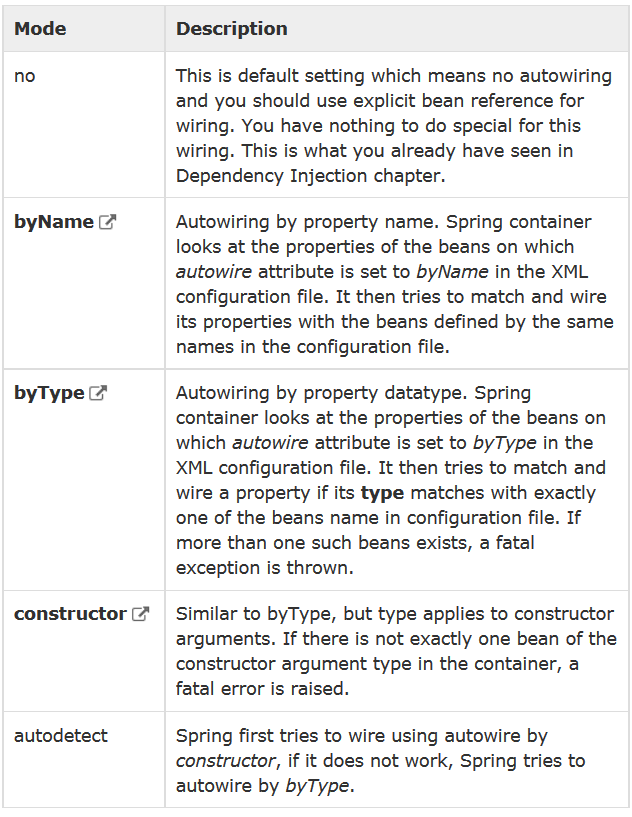
Wstrzykiwanie nulli:



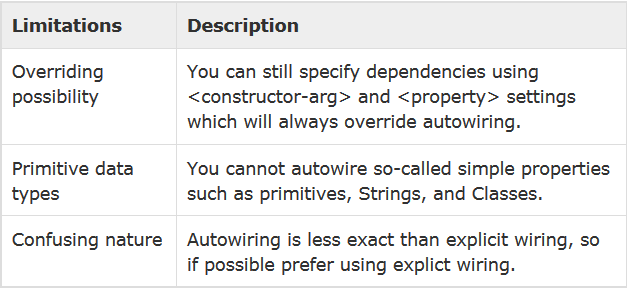
**Autowiring**

W przypadku dużych aplikacji wstrzykiwanie zależności poprzez elementy <constructor-arg> i <property> w XML’ach byłoby bardzo pracochłonne. Dlatego kontener Springowy zapewnia mechanizm automatycznego łączenia relacji między współpracującymi bean’ami, bez użycia tych tagów. Znacznie zmniejsza to ilość XML’owego kodu.

Dostępne tryby autowiringu:



byType i construktor są wykorzystywane do wirowania tablic, czy kolekcji.

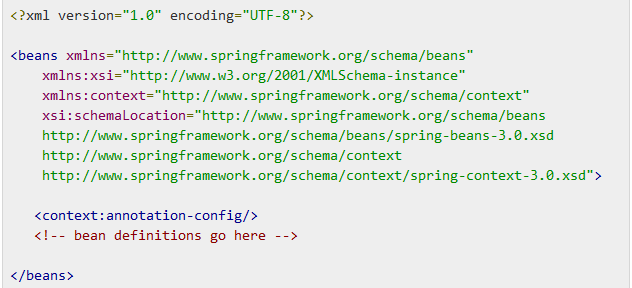


**Annotation based configuration**

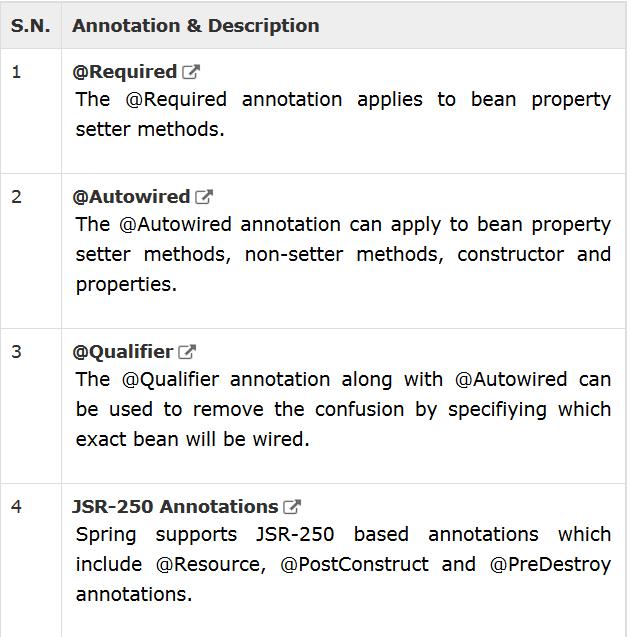
Spring 2.5 wprowadził możliwość konfiguracji DI z wykorzystaniem adnotacji.

Wstrzykiwanie na podstawie adnotacji jest wykonywane przez XML injection i przez to XML’owe konfiguracje nadpisują konfiguracje przez adnotacje.

Kontener Springowy defaultowo ma wyłączoną możliwość wiązania przez adnotacje.

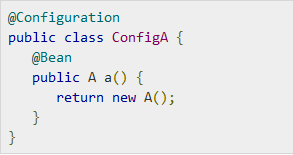


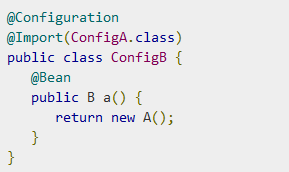
Gdy mamy skonfigurowany paramter <context:annotation-config/> Spring powinien automatycznie wykrywać nasze adnotacje i wiązać wartości z własnościamia, metody i konstruktury.

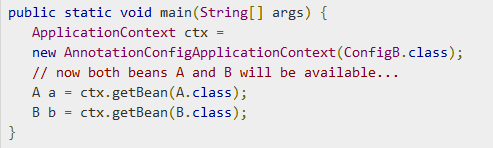


Adnotacja @Import

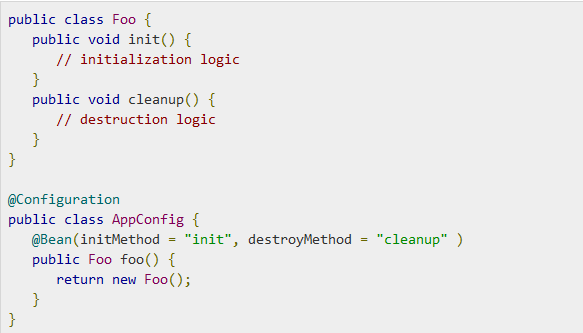
Pozwala na załadowanie definicji @Bean z innego pliku konfiguracyjnego np.:

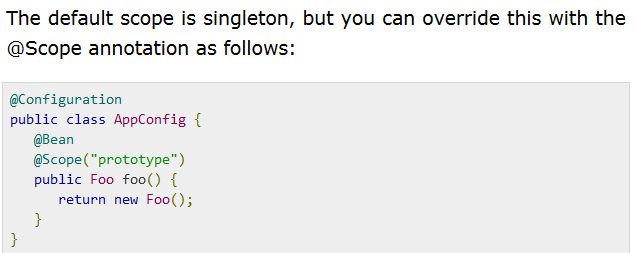






Init i destroy



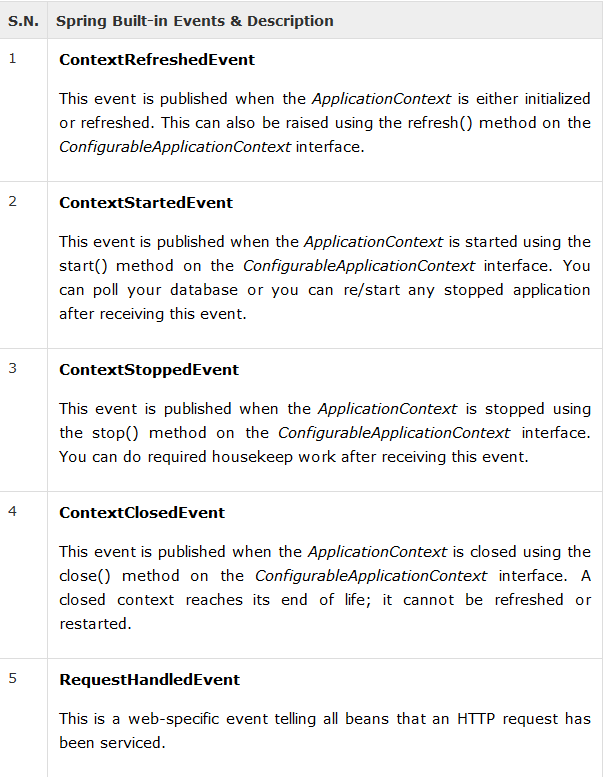


**Spring Event Handling**

Jak widzieliśmy w poprzednich przykładach podstawą Spring’a jest ApplicationContext, który odpowiada za zarządzanie całym cyklem życia bean’ów.

ApplicationContext wywołuje konkretne zdarzenia podczas ładowania bean’ów. Np. ContextStartedEvent, ContextStoppedEvent

Obłsługa tych zdarzeń jest zapewniania poprzez klasę ApplicationEvent i interfejs ApplicationListener.



Springowa obsługa zdarzeń jest jednowątkowa. Należy o tym pamiętać w trakcie projektowania aplikacji.