Moduł 8: Konfiguracja kontenera

W dotychczasowych przykładach głównie korzystaliśmy z konfiguracji automatycznej, z wykorzystaniem adnotacji. Było to wystarczające w naszych przykładowych aplikacjach, jednakże nie zawsze taka konfiguracja jest wystarczająco. Już przy okazji konfigurowania źródła danych uwydatniły się jej braki i bazę danych trzeba było konfigurować explicite. W niniejszym module usystematyzujemy wiedzę dotyczącą konfiguracji, zarówno tej opartej na plikach XML, jak i tej bazującej na JavaConfig.

Konfiguracja XML przez cały okres szkolenia była przeze mnie skrzętnie pomijana (sprowadzana do niezbędnego minimum). To minimum to były komponenty wykorzystywane w aplikacji, lecz będące częścią frameworka; komponenty, które należało tylko skonfigurować i użyć. Konfiguracja XML nie była ona konieczna do wyjaśnienia zasad działania frameworka, a miejscami wprowadzałaby tylko dodatkowy szum, mogący utrudnić zrozumienie omawianych zagadnień. Teraz gdy już omówiona została większość materiału, można z powodzeniem przeanalizować, w jaki sposób konfigurować aplikację wykorzystując pliki XML oraz JavaConfig.

Którą konfigurację wybrać?

Automatyczna konfiguracja (zwana także *autowiring*) ma swoje wady i zalety. Jeżeli konfiguracja przez adnotację używana jest konsekwentnie, przez cały projekt, z wykorzystaniem tych samych konwencji, nie ma niebezpieczeństwa wprowadzenia bałaganu w projekcie. Największym bowiem problemem dla konfiguracji automatycznych (w ogólności, nieważne już czy w są to aplikację oparte o Spring, czy EJB) może być zarządzanie konfiguracją, jeżeli różni programiści, w jednej aplikacji, będą stosować różne konwencje; część aplikacji będzie konfigurowana poprzez adnotacje, a cześć poprzez pliki konfiguracyjne XML. Abstrahując od potencjalnych problemów z odnalezieniem właściwej konfiguracji, pojawiają się dodatkowe niebezpieczeństwa związane z pierwszeństwem konfiguracji: konfiguracja poprzez adnotację wykonywany jest zawsze przed konfiguracją zapisaną explicite – istnieje zatem możliwość (lub niebezpieczeństwo – zależnie od punktu widzenia) wzajemnego nadpisywania konfiguracji.

Dodatkowo, pewną niedogodnością związaną z konfiguracją XML jest brak sprawdzenia poprawności typów – pliki XML to zwykłe pliki tekstowe nie ma więc możliwości statycznego sprawdzania np. typów przez kompilator Javy. Co prawda, niedogodność ta została częściowo usunięta przez wsparcie ze strony środowiska deweloperskiego (Spring IDE jako wtyczka do Eclipse'a lub SpringTool Suite), które nie dopuszcza literówek lub pilnuje, aby pełne nazwy klas komponentów miały odzwierciedlenie w rzeczywistości. Alternatywnie, począwszy od wersji 3.x frameworka, możliwe jest użycie JavaConfig, czyli także konfiguracji poprzez adnotację (ze sprawdzaniem typów) – jest to jednak podejście nie inwazyjne, niedotykające konfigurowanych komponentów.

Dużo zatem zależy od preferencji samych programistów, niektórzy wolą, aby meta dane konfiguracyjne połączone były z kodem, inni preferują swobodę i elastyczność umożliwiającą zmianę konfiguracji bez konieczności ponownej kompilacji aplikacji (taka możliwość daje konfiguracja XML). Z drugiej strony, nikt przy zdrowych zmysłach nie pozwoli na zmiany w konfiguracji aplikacji produkcyjnej, bez ponownej fazy testów – więc często elastyczność i tak jest iluzoryczna. Spring, po

raz kolejny udostępnia programiście kilka rozwiązań dla jednego problemu i umożliwia wybór optymalnego rozwiązania optymalnie dostosowanego do potrzeb.

Jak podzielić pliki konfiguracyjne?

Niezależnie od wybranego podejścia (XML, JavaConfig), nigdy nie warto przechowywać konfiguracji w całości, w jednym pliku. Dobrą praktyką jest podział konfiguracji na obszary. Jeżeli aplikacja to jeden, monolityczny projekt, niepodzielony na moduły, warto chociaż wprowadzić podział plików konfiguracyjnych. Mniejsza struktura jest zdecydowanie łatwiejsza w utrzymaniu i zarządzaniu, niż rozwlekły na kilkanaście tysięcy linii plik XML.

Najbardziej intuicyjnym sposobem dzielenia struktury jest podział ze względu na warstwy w aplikacji: baza danych, komponenty biznesowe, frontend, API do wywołań zdalnych itd. Ułatwia to utrzymanie kodu, poprzez zmniejszenie liczby osób pracujących i modyfikujących jeden fragment rozwiązania, a także znacząco upraszcza testowanie (o czym w następnym module) poprzez możliwość selektywnego testowania fragmentów aplikacji, zaślepiając (poprzez namiastki – ang. *mock*) pozostałe części rozwiązania. Poprzez prosty wybór plików konfiguracyjnych, w zależności od środowiska, można zastąpić produkcyjną bazę danych, lokalną (programisty) lub bazę typu *embedded*, przechowywaną w pamięci (tylko na czas testów).

Podstawowa konfiguracja XML

Po tym abstrakcyjnym wprowadzeniu, spójrzmy, jak wyglądałaby konfiguracja przykładowych programów, gdyby korzystać tylko i wyłącznie z konfiguracji XML (lub JavaConfig). Poniższe przykłady obejmują tylko i wyłączenie tworzenie komponentów i inne kwestie nieporuszane dotychczas w szkoleniu. Konfiguracje nie obejmują konfigurowana źródła danych (lub jakichkolwiek innych komponentów dostarczanych przez framework) – zostało to omówione w innych modułach, gdzie było konieczne do zrozumienia zagadnienia.

Konfiguracja komponentów

Zacznijmy od najprostszych przykładów – czyli konfiguracji komponentów. Dotychczas, komponenty konfigurowane były automatycznie: odpowiedni wpis w pliku XML nakazywał skanowanie pakietów w poszukiwaniu komponentów (klas oznaczonych adnotacją `@Component`), do który wstrzykiwane były zależności (poprzez atrybuty oznaczone `@Autowired` lub `@Resource`). Jeżeli usunięty zostanie wpis konfiguracyjny, to aplikacja przestaje działać i należy powiązania pomiędzy komponentami skonfigurować ręcznie. Na samym początku stajemy przed pierwszy problemem: wszystkie zależności rozwiązywane były przez kontener i nie było konieczności używania modyfikatorów pól – przez co nie było ich wcale. Abstrahując od dyskusji o zasadności tego podejścia – jest to często spotykane działanie, szczególnie w zespołach pracujących zarówno z platformą JEE (EJB 3.x) jak i Spring. Faktem jednak jest, należy dopisać metody umożliwiające poprawną inicjację obiektu (albo poprzez metody get i set lub z wykorzystaniem konstruktora).

```
public class TasksService {

	@Resource
	MessageSource messages;

	@Autowired
	SqlMapClient sqlMapClient;

	@Autowired
	Validator validator;

	//..

}
```

Dla zmodyfikowanej klasy TaskService konfiguracja XML może wyglądać następująco.

```
	<bean id="taskService" class="pl.devcastzone.spring.todo.TasksService">
		<property name="validator" ref="validator" />
		<property name="sqlMapClient" ref="sqlMapClient" />
		<property name="messages" ref="messageSource" />
	</bean>
```

```
	<bean id="taskService" class="pl.devcastzone.spring.todo.TasksService"
		p:validator-ref="validator"
		p:sqlMapClient-ref="sqlMapClient"
		p:messages-ref="messageSource" />
```

```
	<bean id="taskService" class="pl.devcastzone.spring.todo.TasksService">
		<constructor-arg ref="validator" />
		<constructor-arg ref="messageSource"/>
		<constructor-arg ref="sqlMapClient" />
	</bean>
```

Pierwszy i drugi zapis są równoznaczne, notacja „p” jest po prostu skróconym zapisem odwołania się do pola (pojawiło się to w wersji 2.0 frameworka). Nie zmienia to niczego poza rozmiarem pliku konfiguracyjnego i polepszeniem jego czytelności. Ostatni zapis używa konstruktora do utworzenia obiektu. Warto zauważyć, że użycie konstruktora uniemożliwia skorzystanie z pierwszych dwóch sposobów tworzenia komponentów (chyba że wcześniej został utworzony dodatkowy, bezargumentowy konstruktor – z niego bowiem korzysta framework do tworzenia obiektów w sytuacjach, gdy zależności rozwiązywane są poprzez modyfikatory).

Pliki zewnętrzne

Jeden z przykładowych serwisów był dodatkowo konfigurowany za pomocą danych zapisanych w pliku .properties.

```
public class PrettyStringTaskPrinter implements TaskPrinter {

	@Value("${date.pattern}")
	String pattern;

	@Autowired
	TasksService tasksService;

	//..

}
```

Konfiguracja i wstrzyknięcie wartości odbywa się w sposób analogiczny jak w poprzednich przykładach, z tą różnicą, że zamiast atrybuty „ref" oznaczającego referencje do już istniejącego komponentu, użyty zostanie atrybut „value". Sama notacja pozostaje taka sama.

```
	<bean id="taskPrinter"
		class="pl.devcastzone.spring.todo.PrettyStringTaskPrinter">
		<property name="tasksService" ref="taskService" />
		<property name="pattern" value="${date.pattern}" />
	</bean>
```

Precyzowanie zależności za pomocą kwalifikatorów

Kwalifikatory to kolejna nowość w Spring (wprowadzona w wersji 3.0). Umożliwiała ona dodatkowe rozróżnienie komponentów implementujących ten sam interfejs (lub tych samych komponentów ale inaczej utworzonych – np. z innymi danymi). W przykładowych programach były to dwa komponenty, rozróżnione kwalifikatorem `@PrintQualifier`

```
@Component
@PrintQualifier(printerType = Printers.TO_STRING)
public class ToStringTaskPrinter implements TaskPrinter {
	//..
}
```

```
@Component
@PrintQualifier(printerType = Printers.PRETTIFY)
public class PrettyStringTaskPrinter implements TaskPrinter {
	//..
}
```

Przekładając to na konfigurację XML, możemy utworzyć parę komponentów o różnych nazwach i jawnie, w konfiguracji XML przywołujemy oczekiwaną przez nas instancję – ignorując zupełnie kwalifikatory.

```
<bean id="prettyTaskPrinter"
      class="pl.devcastzone.spring.todo.PrettyStringTaskPrinter">
      <property name="tasksService" ref="taskService" />
      <property name="pattern" value="${date.pattern}" />
</bean>

<bean id="toStringTaskPrinter"
      class="pl.devcastzone.spring.todo.PrettyStringTaskPrinter">
      <property name="tasksService" ref="taskService" />
</bean>

<!-- wykorzystanie odpowiedniego komponentu -->
<bean id="app" class="pl.devcastzone.spring.App">
      <property name="printer" ref="prettyTaskPrinter" />
</bean>
```

To jest właśnie największa zaleta jawnej konfiguracji w XMLu – programista ma pełną kontrolę na tworzonymi komponentami, nic nie dzieje się automatycznie (automagicznie) – nie ma zatem problemu z wyborem implementacji lub wersji komponentu. Wszelkie zależności rozwiązuje programista, samodzielnie.

Oczywiście – jest możliwość wymieszania konfiguracji XML z konfiguracją poprzez adnotację w taki sposób, że po stronie XML jawnie zadeklarowane zostaną kwalifikatory, natomiast wybór implementacji będzie się odbywał na poziomie docelowego komponentu, wykorzystując adnotacje.

```
<bean id="prettyTaskPrinter"
      class="pl.devcastzone.spring.todo.PrettyStringTaskPrinter">
      <qualifier
            type="pl.devcastzone.spring.todo.annotations.PrintQualifier">
            <attribute key="printerType" value="TO_STRING"/>
      </qualifier>
      <property name="tasksService" ref="taskService" />
      <property name="pattern" value="${date.pattern}" />
</bean>

<bean id="toStringTaskPrinter"
      class="pl.devcastzone.spring.todo.PrettyStringTaskPrinter">
      <qualifier
      type="pl.devcastzone.spring.todo.annotations.PrintQualifier">
            <attribute key="printerType" value="PRETTIFY"/>
      </qualifier>
      <property name="tasksService" ref="taskService" />
</bean>
```

Tak utworzone komponenty mogą zostać wybrane w sposób automatyczny.

```
<!-- instrukcja dla kontenera aby przeprowadzić -->
<!-- konfigurację w sposób automatyczny -->
<context:annotation-config />

<!-- brak jawnej deklaracji, który komponent ma zostać wykorzystany -->
<bean id="app" class="pl.devcastzone.spring.App" />
```

```
public class App {

	@Autowired
	@PrintQualifier(printerType=Printers.PRETTIFY)
	TaskPrinter printer;

	//..

}
```

Konfiguracja Spring MVC za pomocą XML

Z poprzednich modułów wiemy, że kontrolery Spring MVC także są komponentami zarządzanymi przez framework, a co za tym idzie – także mogą być konfigurowane za pomocą XML zawartego w pliku konfiguracyjnym `DispatcherServlet`. Dotychczas robione było to automatycznie, za pomocą adnotacji, ograniczając konfigurację XML to 4 deklaracji, z czego najistotniejsza to `mvc:annotation-driven`.

```
	<mvc:annotation-driven />
	<context:component-scan base-package="pl.devcastzone.todo" />

	<mvc:resources mapping="/resources/**" location="/resources/" />

	<bean
	class="org.springframework.web.servlet.view.InternalResourceViewResolver">
		<property name="prefix" value="/WEB-INF/views/" />
		<property name="suffix" value=".jsp" />
	</bean>
```

W ten sposób, komponenty zawartych w pakiecie `pl.devcastzone.todo` były skanowane w poszukiwaniu adnotacji konfiguracyjnych. Pozostałe dwie dyrektywy konfigurują lokalizację plików JSP (widoków) oraz położenie plików resources (np. CSS, JavaScript, obrazy itd.).

Usunięcie fragmentu `mvc:annotation-driven` spowoduje natychmiastową zmianę konfiguracji, widoczną już w momencie uruchomienia serwera – w logach serwera zabraknie charakterystycznej informacji dotyczącej mapowania. Za pomocą tego jednego wpisu konfiguracyjnego, deklarowany był komponent typu `HandlerMapper`, służący do wyboru kontrolera obsługującego żądanie oraz `HandlerAdapter` mający na celu dopasowanie metody kontrolera do wartości żądania (metoda http, parametry). W przypadku adnotacji był to `DefaultAnnotationHandlerMapping` oraz `AnnotationMethodHandlerAdapter`. Jeżeli jednak nie decydujemy się na konfigurację automatyczną, możemy analogiczną pracę wykonać ręcznie. Wiąże się to z pewnymi niedogodnościami, ponieważ zmusza to nasze kontrolery do implementowania specyficznych dla Springa interfejsów – np. `AbstractController`.

```
public class GetTasksController extends AbstractController {

	TasksService service;

	public void setService(TasksService service) {
		this.service = service;
	}

	protected ModelAndView handleRequestInternal(HttpServletRequest request,
			HttpServletResponse response) throws Exception {
		ModelAndView modelAndView = new ModelAndView("tasks");
		List<Task> tasks = service.getTasks();
		modelAndView.getModel().put("tasks", tasks);
		modelAndView.getModel().put("task", new Task());
		return modelAndView;
	}

}
```

Jak widać na powyższym przykładzie, kontroler nie posiada jakichkolwiek adnotacji – całość konfiguracji odbywa się w pliku XML.

```
<bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter" />

<bean
	class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping" />

<bean name="/tasks"
	class="pl.devcastzone.todo.GetTasksController">
	<property name="service" ref="taskService"/>
</bean>
```

W ten sposób, można łączyć istniejące aplikacje (nie zawierające adnotacji) z nowymi komponentami napisanymi w Spring 3.x, opartych w całości o automatyczną konfigurację.

Konfiguracja JavaConfig

Wykorzystując klasy konfiguracyjne JavaConfig, można uzyskać zbliżone rezultaty jak w przypadku powyższej konfiguracji XML. Kompletna konfiguracja wymaga jednak zarówno konfiguracji JavaConfig jak i fragmentu konfiguracji XML

```
@Configuration
@ImportResource("classpath:META-INF/spring/app-properties.xml")
public class AppConfiguration {

	@Value("${date.pattern}")
	String pattern;

	@Bean
	public TasksService tasksService() throws Exception {
		return new TasksService(messages(), sqlMapClient(), validator());
	}

	@Bean
	public TaskPrinter taskPrinter() throws Exception {
		PrettyStringTaskPrinter p = new PrettyStringTaskPrinter();
		p.setTasksService(tasksService());
		p.setPattern(pattern);
		return p;
	}

	@Bean
	public App app() throws Exception {
		App a = new App();
		a.setPrinter(taskPrinter());
		return a;
	}

	@Bean
	public DataSource dataSource() {
		BasicDataSource ds = new BasicDataSource();
		ds.setDriverClassName("org.hsqldb.jdbcDriver");
		ds.setUrl("jdbc:hsqldb:file:target/localdb/testdb");
		ds.setUsername("sa");
		ds.setPassword("");
		return ds;
	}

	@Bean
	public SqlMapClient sqlMapClient() throws Exception {
		SqlMapClientFactoryBean factory = new SqlMapClientFactoryBean();
		factory.setDataSource(dataSource());
		factory.setConfigLocation(new ClassPathResource(
			"META-INF/sqlmap/sqlmap-config.xml"));
		factory.afterPropertiesSet();
		SqlMapClient sqlMapClient = factory.getObject();
		return sqlMapClient;
	}

	@Bean
	public Validator validator() {
		return new LocalValidatorFactoryBean();
	}

	@Bean
	public MessageSource messages() {
		ReloadableResourceBundleMessageSource ms =
			new ReloadableResourceBundleMessageSource();
		ms.setBasename("message");
		ms.setDefaultEncoding("UTF-8");
		return ms;
	}
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.0.xsd">

       <context:property-placeholder
              location="classpath:date-format.properties" />

</beans>
```

Należy przy tym zaznaczyć, iż konfiguracja poprzez adnotację `@Configuration` nie ma na celu całkowitego zastąpienia konfiguracji XML. Jak widać na powyższym przykładzie, nie ma możliwości sięgnięcia do zewnętrznego pliku z danymi (pliku properties) – i musi się to odbywać poprzez zewnętrzny plik konfiguracyjny XML.

Przy okazji jest to kolejny przykład, że pliki konfiguracyjne należy dzielić na mniejsze, możliwe do współdzielenia części. Jeżeli zatem wydzielimy część odpowiedzialną za meta konfigurację (plik properties) oraz część bazodanową do dwóch oddzielnych plików – elementy konfiguracji co do której zakładamy brak zmian – konfiguracja JavaConfig znacznie się uprości.

```
@Configuration
@ImportResource({"classpath:META-INF/spring/app-properties.xml",
        "classpath:META-INF/spring/app-context-db.xml"})
public class AppConfiguration {

        @Autowired
        MessageSource messages;

        @Autowired
        SqlMapClient sqlMapClient;

        @Autowired
        Validator validator;

        @Value("${date.pattern}")
        String pattern;

        @Bean
        public TasksService tasksService() {
                return new TasksService(messages, sqlMapClient, validator);
        }

        @Bean
        public TaskPrinter taskPrinter() {
                PrettyStringTaskPrinter p = new PrettyStringTaskPrinter();
                p.setTasksService(tasksService());
                p.setPattern(pattern);
                return p;
        }

        @Bean
        public App app() {
                App a = new App();
                a.setPrinter(taskPrinter());
                return a;
        }
}
```

Wciąż jednak pozostaje aktualne pytanie – kiedy używać, którego podejścia i czy warto łączyć je ze sobą. Moim zdaniem warto, szczególnie w sytuacjach, gdy tworzone są testy wymagające inicjacji kontenera; z pomocą JavaConfig można w najprostszy i najszybszy możliwy sposób utworzyć konfigurację frameworka i następnie wykorzystać ją w testach. Jest to o tyle nęcące, że całość testu przechowywana jest jednym pliku, jest zamkniętą całością – znacznie ułatwiając późniejsze utrzymanie takiego kodu. Temat ten zostanie jeszcze znacząco rozwinięty w kolejnym module.

Konfiguracja bez konfiguracji

Ostatnią metodą konfigurowania kontekstu, metodą zdecydowanie najmniej formalną, jest pominięcie plików konfiguracyjnych i konfiguracja kontekstu poprzez podanie klas bezpośrednio do `ApplicationContext`. Sprawdza się to w małych aplikacjach (lub podobnie jak JavaConfig) w testach, gdzie zależności sprowadzają się do minimum i wykorzystanie pliku XML do konfiguracji staje się zbędną nadmiarowością. Za pomocą klasy `AnnotationConfigApplicationContext` można zbudować kontekst podając tylko klasy komponentów odpowiednio opatrzone adnotacjami.

```
AbstractApplicationContext parent =
      new ClassPathXmlApplicationContext(new String[] {
            "META-INF/spring/app-properties.xml",
            "META-INF/spring/app-context-db.xml"});

AnnotationConfigApplicationContext context =
      new AnnotationConfigApplicationContext();
context.setParent(parent);
context.register(PrettyStringTaskPrinter.class);
context.register(TasksService.class);
context.refresh();
```

Sama klasa `AnnotationConfigApplicationContext` posiada kilka przeciążonych konstruktorów; ja wybrałem ten niepociągający za sobą automatycznego odświeżenia kontekstu. Możliwe jest podanie listy klas bezpośrednio w konstruktorze. Wtedy kontekst zostanie samodzielnie odświeżony (bez konieczności wywoływania metody refresh()). Niestety nie będzie wtedy możliwe podanie kontekstu nadrzędnego (w tym konkretnym przypadku – utworzonego na bazi plików konfiguracyjnych).

Podsumowując

Opisałem kilka różnych sposobów budowania aplikacji aby (po raz kolejny) pokazać bardzo dużą elastyczność frameworka. W zależności od doświadczenia, potrzeb, inwencji programisty, funkcjonalnych bądź niefunkcjonalnych wymagań, możliwe jest swobodne budowanie aplikacji; zarówno w całości jak i wydzielając z systemu pomniejsze fragmenty. Dobrze przygotowana konfiguracja powoduje że aplikacja przestaje być monolitem; zyskuje niesamowitą elastyczność umożliwiającą tworzenie wydajnych testów integracyjnych lub próby wdrożenia fragmentów aplikacji na środowiska, aby zweryfikować połączenia z bazą danych, której z wielorakich względów (np. licencyjnych) nie jesteśmy w stanie zainstalować lokalnie.