# Moduł 7: Warstwa Web

W poprzednich modułach skupialiśmy się głównie na warstwach biznesowych, realizując pewne operacje związane np. z dostępem do bazy danych. Zupełnie pominięta została kwestia interakcji z użytkownikiem – wszystkie komunikaty wyświetlane były po prostu w konsoli systemowej. Komunikacja z użytkownikiem nie została pominięta lub zignorowana; wprowadzenie jej na wczesnym etapie budowania aplikacji, mogłoby znacznie zaciemnić obraz i wprowadzić niepotrzebne zamieszanie. Teraz gdy już potrafimy składać aplikację z komponentów oraz rozwiązywać zależności między nimi, możemy z powodzeniem dorobić do naszej aplikacji interfejs. W tym module zaprezentuje kilka sposób na udostępnienie aplikacji z wykorzystaniem protokołu HTTP. Specjalnie nie użyłem sformułowania strona WWW, ponieważ nie jest to jedyna opcja, którą omówimy w niniejszym module.

Na początek skupimy się na elemencie frameworka, czyli Spring MVC; implementacji wzorca model – widok – kontroler (ang. *model – view – controller*). Omówione zostanie tworzenie aplikacji WWW z wykorzystaniem istniejących komponentów biznesowych (jako tzw. *backend*) oraz ponowne użycie już opanowanych technik (jak np. walidacja) w kontekście aplikacji WWW.

Początkowo widoki będą budowane w czystym JSP (i jest to dla prostych aplikacji zupełnie wystarczające). Zasygnalizuje jednak możliwość użycia innych narzędzi do tworzenia widoków, takich jak Tiles lub Velocity.

Spring jako narzędzie niezwykle elastycznie nie wymusza na programiście użycia tylko i wyłączenie jednej konkretnej implementacji wzorca MVC, a możliwe jest połączenia komponentów z różnymi innymi frameworkami WWW (takimi jak Struts, Tapestry, WebWorks).

Na sam koniec pozbędziemy się zupełnie widoków i użyjemy istniejących komponentów do skonstruowania API umożliwiające komunikację z aplikacją poprzez wiadomości XML.

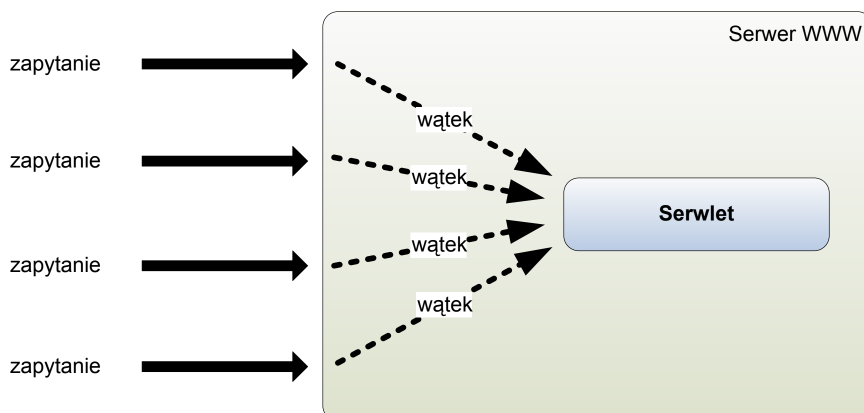
## Nowy projekt webowy

Na początek przygotujemy środowisko, na którym będziemy osadzać aplikację. Użyjemy serwer Apache Tomcat w wersji 6 (czyli kontenera serweltów, chociaż serwletów jako takich jawnie póki co nie będziemy wykorzystywać). Stworzymy także nowy projekt: aplikację webową. Jeżeli używamy SpringTool Suite, to możemy skorzystać z istniejących szablonów, gdzie wybieramy Spring MVC Project. Środowisko utworzy nam nowy projekt, wraz ze strukturą oraz z predefiniowanymi zależnościami. Aplikacja WWW opata o framework Spring nie różni się swoją strukturą od innych aplikacji wykorzystujących serwlety: w katalogu WEB-INF znajduje się plik web.xml, który jest podstawową konfiguracją aplikacji (tzw. deskryptorem wdrożenia). Dodatkowo, wzorzec wygenerował domyślną konfigurację oraz kilka podstawowych plików. Całość stanowi działający szkielet aplikacji, który można wdrożyć na serwer. Nim jednak to zrobimy, przyjrzyjmy sie poszczególnym elementom aplikacji.

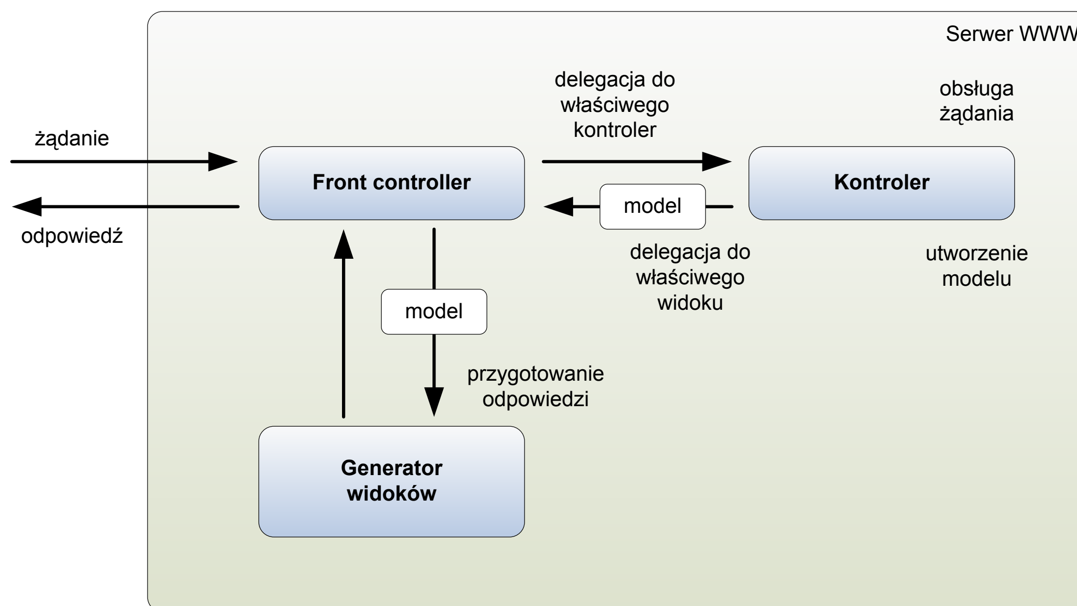
Jeżeli decydujemy się wykorzystać Spring MVC, to po pierwsze zmienia się sposób obsługi przychodzących do serwera żądań. W klasycznym ujęciu, serwlet jest to singleton, który bezstanowo obsługuje wiele przychodzących żądań. Poprzez mapowanie w pliku konfiguracyjnym web.xml,

copyright BNS MEDIA

kontener serwletów wie, którą klasę skojarzyć z adresem URL i obsługa kierowana jest bezpośrednio do serweletu. Całość przetwarzania może zostać zamknięta w serwlecie (np. cała strona HTML może zostać wygenerowana po stronie Javy) lub też serwlet może przekazać generowanie samej strony do pliku JSP.




Wykorzystanie Spring MVC zmienia ten paradygmat. Jak widać na poniższej ilustracji, całość rozwiązania oparta jest o wzorzec front controller; wszystkie żądania obsługiwane są przez jeden serwlet, który z kolei zarządzana przekazywaniem zadań po odpowiednich kontrolerów i widoków.




Jeżeli przyglądniemy się wygenerowanej konfiguracji w deskryptorze wdrożenia web.xml, to w sekcji odpowiedzialnej za mapowanie serwletów zobaczymy tylko pojedynczy wpis, mapujący wszystkie przychodzące żądania na klasę `DispatcherServlet`, klasę Front Controllera.

```
<!-- Processes application requests -->
<servlet>
	<servlet-name>appServlet</servlet-name>
	<servlet-class>
		org.springframework.web.servlet.DispatcherServlet
	</servlet-class>
	<init-param>
		<param-name>contextConfigLocation</param-name>
		<param-value>
			/WEB-INF/spring/appServlet/servlet-context.xml
		</param-value>
	</init-param>
	<load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
	<servlet-name>appServlet</servlet-name>
	<url-pattern>/</url-pattern>
</servlet-mapping>
```

Dodatkowy parametr konfiguracyjny serwletu to lokalizacja pliku konfiguracyjnego. Zwyczajowo, konfiguracja przechowywana poszukiwana jest w pliku [nazwa serwletu]-servlet.xml, w katalogu WEB-INF. Można jednak, jak na powyższym przykładzie, zmienić jego lokalizację. W pliku tym znajduje się konfiguracja widoków, kontrolerów, obsługa wyjątków, konfiguracja lokalizacji – konfiguracja całej warstwy związanej z webową częścią aplikacji. Jest to konfiguracja klasy `WebApplicationContext`, który posiada dostęp to wszystkich uprzednio zdefiniowanych komponentów biznesowych oraz tworzy własne (tylko odnoszące się do kontekstu WWW).

Poza powyższym fragmentem, w pliku web.xml znajdują się jeszcze dwa dodatkowe parametry konfiguracyjne: parametr konfiguracyjny wskazujący na plik root-context.xml oraz definicja klasy nasłuchującej (ang. listerner) `ContextLoaderListener`.

```
<!-- The definition of the Root Spring Container -->
<!-- shared by all Servlets and Filters -->
<context-param>
	<param-name>contextConfigLocation</param-name>
	<param-value>/WEB-INF/spring/root-context.xml</param-value>
</context-param>

<!-- Creates the Spring Container shared by all Servlets and Filters -->
<listener>
	<listener-class>
		org.springframework.web.context.ContextLoaderListener
	</listener-class>
</listener>
```

Jeżeli decydujemy się tylko i wyłącznie na korzystanie ze Spring MVC powyższe elementy nie są z naszego punktu widzenia tak bardzo istotne. Teoretycznie całą konfigurację można umieścić w pliku servlet-context.xml. Nie jest to jednak rozwiązanie najtrafniejsze, zaleca się bowiem dzielenie plików konfiguracyjnych na mniejsze elementy (wprowadzenie pewnej granulacji, o której więcej będę mówił w dalszych modułach). `ContextLoaderListener` umożliwia połączenie komponentów biznesowych napisanych z pomocą Springa, z dowolnym frameworkiem MVC, takim jak Struts czy WebWork. Klasa `ContextLoaderListener` tworzy `WebApplicationContext` (czyli dokładnie to samo co `DispatcherServlet`), z tym że samą obsługą żądań nie zajmuje się Front Controller i realizowana jest ona w sposób wybrany przez programistę.

Na początek skupimy się na wykorzystaniu Spring MVC i do alternatywnych sposobów konfiguracji kontekstu wrócimy w dalszej części modułu.

copyright BNS MEDIA

## Hello Web Spring

Poza plikiem web.xml wzorzec utworzył także kilka innych plików: klasę `HomeController`, dwa wspomniane powyżej pliki konfiguracyjne (root-context.xml i servlet-context.xml) oraz przykładowy widok home.jsp. Bez wgłębiania się w szczegóły, spróbujmy najpierw uruchomić aplikację.

W pierwszym kroku musimy skonfigurować serwer: w widoku Servers wybieramy nowy serwer – Apache Tomcat 6. W ostatnim kroku konfiguracji dodajemy do serwera naszą aplikację webową i uruchamiamy serwer. Środowisko samo dba o odpowiednią kompozycję pliku wdrożenia, kopiuje je do katalogu webapps serwera i uruchamia samego Tomcata. Jeżeli poprzez przeglądarkę wyświetlimy stronę http://localhost:8080/todo-list-webapp (todo-list-webapp to nazwa przykładowej aplikacji), zobaczymy napis „Hello World!" oraz aktualną godzinę. Skoro aplikacja uruchomiła się poprawnie, możemy przeanalizować co się wydarzyło.

Jak już opisywałem, jedynym serwletem zdefiniowanym w aplikacji jest front controller – czyli `DispatcherServlet`, jego konfiguracja znajduje się w pliku servlet-context.xml. Poza znaną z poprzednich plików z dyrektywą component-scan nakazującą skanowanie pakietów w poszukiwaniu adnotacji konfiguracyjnych, w pliku znajdziemy kilka dodatkowych konfiguracji: umożliwienie konfiguracji aplikacji poprzez adnotację (w analogiczny sposób jak miało to miejsce przy okazji omawianych we wcześniejszych modułach komponentów) oraz konfiguracja komponentu ViewResolver, odpowiadającego za poszukiwanie odpowiednich plików jsp w obrębie aplikacji.

```
<!-- Enables the Spring MVC @Controller programming model -->
<annotation-driven />

<!-- Resolves views selected for rendering by @Controllers to -->
<!-- .jsp resources in the /WEB-INF/views directory -->
<beans:bean
      class="org.springframework.web.servlet.view.InternalResourceViewResolver">
      <beans:property name="prefix" value="/WEB-INF/views/" />
      <beans:property name="suffix" value=".jsp" />
</beans:bean>
```

W momencie uruchamiania kontenera, skanowane są pakiety w poszukiwaniu komponentów oznaczonych adnotacją `@Controller`. W istniejącym szkielecie aplikacji istnieje tylko jedna klasa tak oznaczona: `HomeController`, do której zostało skierowane przychodzące żądanie. "Logika biznesowa" zawarta jest w metodzie home controllera, natomiast zwrócona przez metodę wartość "home" jest informacją dla komponentu `ViewResolver`, że należy wygenerować widok z pliku /WEB-INF/views/home.jsp – zgodnie z konfiguracją powyżej. Po wykonaniu tej operacji zawartość strony zostaje zwrócona do użytkownika. To właśnie kontrolery i widoki są sercem aplikacji i konstruując te komponenty programista spędza najwięcej czasu. Omówione pliki konfiguracyjne (w przypadku konfiguracji opartej o adnotację) przez większość czasu pozostają niezmienione – stanowią ramy aplikacji które wypełnia się logiką zawartą właśnie w kontrolerach.

## Kontroler i widok

Dotychczas nie napracowaliśmy się zbytnio, ponieważ całą aplikacja została wygenerowana przez środowisko. Nim jednak przystąpimy do tworzenia własnych kontrolerów, spójrzmy na już utworzony `HomeController`. Na początek najistotniejsze są dwie adnotacje: wspomniana `@Controller` oraz `@RequestMapping`.

copyright BNS MEDIA

```
@Controller
public class HomeController {

	//..

	@RequestMapping(value = "/", method = RequestMethod.GET)
	public String home(Locale locale, Model model) {
		//..
		model.addAttribute("serverTime", formattedDate );
		return "home";
	}

}
```

Pierwsza z wymienionych adnotacji jest informacją dla skanera pakietów, że ma do czynienia z kontrolerem. Natomiast poprzez `@RequestMapping`, rejestrowana jest obsługa konkretnego typu żądania (w tym przypadku metody GET dla bazowego URL "/"). Jak już wcześniej zasygnalizowałem, zwrócona wartość typu String jest nazwą widoku, który zostanie wygenerowany. Sam widok jest niezwykle prosty i dla osób znającej podstawy języka HTML nie wymaga dodatkowego omówienia.

```
<html>
<head>
	<title>Home</title>
</head>
<body>
<h1>
	Hello world!
</h1>

<P>  The time on the server is ${serverTime}. </P>
</body>
</html>
```

Jedną rzeczą potencjalnie zwracającą uwagę jest fragment ${serverTime}, który wyświetla aktualny czas, zdefiniowany uprzednio w metodzie kontrolera. Komunikacja pomiędzy kontrolerem a widokiem odbywa się poprzez obiekt Model, który przekazany został w parametrze metody kontrolera (Model.addAttribute()).

Sam kontroler przyjmuje dwa parametry: `Local` i wspomniany już `Model`. Nie są to jedynie możliwości, a lista potencjalnych argumentów jest znacząca i bardzo dokładnie opisana w dokumentacji. Z naszego punktu widzenia najistotniejsza jest adnotacja `@RequestParam`. Możemy ją oznaczyć dowolny obiekt będący parametrem obsługiwanego żądania. Siłą rzeczy (niejako z definicji) parametry mogą być jedynie prostymi typami lub ich obiektowymi odpowiednikami. Jeżeli zatem pragnęlibyśmy, aby istniejąca powitała nas imieniem, a nie suchym Hello World, możemy spróbować przekazać imię jako parametr. Jeżeli pragniemy tylko wyświetlić parametr na ekranie, to możemy po prostu dopisać w odpowiednim miejscu pliku JSP ${param.name}. Prawdziwa wartość, objawia się jednak gdy potrzebujemy skorzystać z parametru w kontrolerze. Metoda kontrolera przyjmie wtedy nieco zmienioną postać.

```
	@RequestMapping(value = "/", method = RequestMethod.GET)
	public String home(@RequestParam("name") String name,
				Locale locale, Model model) {
		//..
	}
```

Nie jesteśmy także ograniczenia do prostego ciągu znaków, który zwraca metoda kontrolera. Framework przewiduje, że może to być jeden z wielu typów: `Model`, `ModelAndView` (który umożliwia

przekazanie atrybutów do widoku jak również jawne określenie, który widok ma zostać wywołany), `Map` (mapa klucz – wartość parametrów przekazanych widokowi) itd.

## Kontroler a komponent

Kontroler MVC jest takim samym komponentem Spring, jak omawiane w poprzednich modułach komponenty biznesowe. Znaczy się, podlega tym samym prawom wstrzykiwania i rozwiązywania zależności. Oznacza to że możemy użyć w aplikacji webowej, komponentów utworzonych w poprzednich modułach. Jako że dotychczas zajmowaliśmy się dodawanie zadań do listy "todo" i wyświetlaniem tych zadań – użyjmy ponownie tych samych komponentów aby wyświetlić zadania w przeglądarce.

Jeżeli poprzednio utworzoną aplikację potraktujemy jako bibliotekę i dołączymy ją do projektu w postaci pliku jar, to ponowne wykorzystanie istniejącej konfiguracji staje się dziecinnie proste. W pliku servlet-context.xml dopisujemy linijki importujące poszczególne elementy modułu.

```
<beans:import resource="classpath:META-INF/spring/app-properties.xml"/>
<beans:import resource="classpath:META-INF/spring/app-context-db.xml"/>
<beans:import resource="classpath:META-INF/spring/app-context.xml"/>
```

Po przeładowaniu się kontekstu, wśród zainicjowanych komponentów zobaczymy znany z poprzednich części taskService. Możemy pokusić się o jego ponowne wykorzystanie. W tym celu utworzymy nowy kontroler (`TaskController`) oraz odpowiadający mu widok (tasks.jsp).

```
@Controller
public class TaskController {

	@Autowired
	TasksService service;

	@RequestMapping(value="/tasks", method=RequestMethod.GET)
	public String tasks(Model model) {
		List<Task> tasks = service.getTasks();
		model.addAttribute("tasks", tasks);
		return "tasks";
	}

}
```

copyright BNS MEDIA

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
<head>
	<title>Tasks</title>
</head>
<body>
	<h1>The current list of tasks. </h1>

<table>
	<c:forEach items="${tasks}" var="t">
	<tr>
		<td>${t.taskId}</td>
		<td>${t.title}</td>
		<td>${t.date}</td>
	</tr>
	</c:forEach>
</table>

</body>
</html>
```

Jeżeli wywołamy stronę http://localhost:8080/todo-list-webapp to wyświetlane zostaną wszystkie dotychczas utworzone zadania.

W takich właśnie sytuacjach objawia się siła frameworka, wpisana w jego naturę elastyczność i modularność. Luźne powiązania pomiędzy komponentami umożliwiają wielokrotne, ponowne wykorzystanie komponentów, w sytuacjach które początkowo w ogóle nie były przewidziane. Nigdzie w poprzednich modułach nie planowaliśmy wykorzystać komponentów biznesowych do prezentacji zadań poprzez stronę WWW. Mimo tego okazało się to być niezwykle proste i szybkie do osiągnięcia.

## Formularze

Ponownie wykorzystany, utworzony w poprzednich modułach `TaskService`, posiada dwie metody biznesowe: do odczytu zadań (wykorzystana została powyżej) oraz do zapisu zadania. Spróbujmy teraz wykorzystać tę drugą metodę i utworzyć formularz na stronie internetowej, poprzez który dodamy kolejne zadanie do naszej listy. W tym celu wykorzystamy bibliotekę znaczników JSP dostarczaną przez Springa, która umożliwia wygodne połączenie formularza HTML z obiektem modelu (ang. binding).

```
<form:form commandName="task">
        <table>
                <tr>
                        <td>Task title</td>
                        <td><form:input path="title" /></td>
                </tr>
                <tr>
                        <td>Description</td>
                        <td><form:textarea path="description"
                                rows="3" cols="20"/></td>
                </tr>
                <tr>
                        <td>Starting date</td>
                        <td><form:input path="date" /></td>
                </tr>
                <tr>
                        <td>Duration</td>
                        <td>
                                <form:select path="duration">
                                        <form:option value="3600000">1 hour</form:option>
                                        <form:option value="7200000">2 hours</form:option>
                                        <form:option value="10800000">3 hour</form:option>
                                </form:select>
                        </td>
                </tr>
          <tr>
              <td colspan="2">
                  <input type="submit" value="Submit" />
              </td>
          </tr>
        </table>
</form:form>
```

Dodając powyższy fragment do pliku tasks.jsp, tworzymy formularz, który będzie połączony z obiektem w modelu o nazwie "task" (w naszym przypadku jest to pusty obiekt typu Task). Formularz zostanie wysłany metodą POST na ten sam adres (/tasks) i wszystkie wypełnione pola formularza zostaną przypisane do odpowiednich atrybutów klasy, za pomocą nazwy wyszczególnionej w atrybucie path.

Aby poprawnie wygenerować i obsłużyć formularz należy także zmodyfikować kontroler.

```
        @RequestMapping(value="/tasks", method=RequestMethod.GET)
        public String tasks(Model model) {
                List<Task> tasks = service.getTasks();
                model.addAttribute("tasks", tasks);
                model.addAttribute("task", new Task());
                return "tasks";
        }
```

```
        @RequestMapping(value="/tasks", method=RequestMethod.POST)
        public String saveTask() {
                logger.info("Saving task...");
                return "redirect:/tasks";
        }
```

Metoda tasks() kontrolera została uzupełniona o utworzenie nowego obiektu i dodanie go do modelu – w celu poprawnej generacji formularza. Dodana została także druga metoda, obsługująca metodę POST – odpowiedzialna za zapisanie formularza. Zauważmy, że ciąg znaków zwracany przez metodę nie jest nazwą widoku, ale dyrektywą nakazującą przekierowanie na stronę /tasks – natychmiast po zapisie nowego zadania. Jak już pisałem, Spring jest bardzo elastyczny, jeżeli chodzi o typy danych

copyright BNS MEDIA

zwracanych przez metody kontrolera. Zamiast zwracać obiekt typu `View` nakazujący przekierowanie, można uzyskać ten sam efekt stosując przedrostek „redirect" - jak w powyższym przykładzie.

Na postawie informacji w konsoli systemowej widzimy, że metoda została wywołana (pojawił się tekst „Saving task...") - należy jednak zaimplementować samą procedurę zapisu danych.

```
@RequestMapping(value="/tasks", method=RequestMethod.POST)
public String saveTask(Task task) {
        logger.info("Saving task: [" + task + "]");
        service.addTask(task);

        return "redirect:/tasks";
}
```

Rzeczywistość okazuje się jednak bardziej skomplikowana: zwrócony został wyjątek, że spring nie potrafi sobie poradzić z konwersją daty, która została podana w formularzu. W zasadzie mamy tutaj dwa problemy: użytkownik dostaje mało czytelną informację poprzez wyrzucony wyjątek (a lepiej by było móc samodzielnie obsłużyć ten problem) oraz framework nie potrafi samodzielnie zamieć ciągu znaków na wartość typu `Date`. Wszystkie dane z formularza domyślnie są typu `String`, jednak framework samodzielnie próbuje dokonać konwersji na typy odpowiednie w modelowym obiekcie. O ile dla typów prostych (np. `Integer`) jest to łatwe, konwersja daty wymaga dodatkowych kroków. Nim jednak przejdziemy do konwersji, poprawmy kwestie związaną z obsługą błędów. Informacje o wszelkich problemach związanych z łączeniem formularza z obiektem (a także problemy związane z walidacją – o czym za chwilę) przechowywane są w obiekcie `BindingResult`, który musimy dołączyć jako dodatkowy parametr metody kontrolera.

```
@RequestMapping(value="/tasks", method=RequestMethod.POST)
public String saveTask(Task task, BindingResult results) {
        logger.info("Saving task: [" + task + "]");
        if (results.hasErrors()) {
                return "tasks";
        }

        service.addTask(task);
        return "redirect:/tasks";
}
```

Tak skonfigurowany kontroler, w przypadku jakichkolwiek błędów nie pozwoli na zapis obiektu i przekieruje użytkownika na powrót do formularza. Użytkownik staje przed szansą sprawdzenia formularza oraz poprawienia błędów (na podstawie informacji wyświetlonych na ekranie – o czym za chwilę, przy okazji walidacji).

Ostatnim krokiem jest konwersja typów, z ciągu znaków na wartość typu data. W najnowszych wersjach frameworka sprawa jest niezwykle prosta: należy rozszerzyć model o informację nt. formatu danych, posługując się odpowiednią adnotacją. W tym przypadku będzie to `@DateTimeFormat`, ale z powodzeniem można używać adnotacji `@NumberFormat`, aby konwertować typy liczbowe.

```
@Temporal(TemporalType.DATE)
@Column(name = "startdate")
@NotNull
@Future
@DateTimeFormat(pattern="MM-dd-yyyy")
private Date date;
```

Zauważmy także, że wciąż używamy tego samego modelu; jeden obiekt niesie komplet meta informacji, począwszy od typu danych i elementów bazy danych, poprzez walidację, na formatowaniu skończywszy.

Mimo że użycie adnotacji rozwiązuje problem niezwykle szybko, bardzo często można się spotkać z jawnymi konwerterami, z czasów sprzed Javy 1.5 – taka konwersja odbywała się wtedy w sposób diametralnie inny. Dla przykładu, poprzez samodzielnie rozszerzenie klasy `PropertyEditorSupport`, możliwa była dowolna konwersja pomiędzy tekstem a obiektem. Taki edytor należało zaimplementować samodzielnie, a następnie zarejestrować (w pliku konfiguracyjnym). W pewnym typowych przypadkach, można jednak skorzystać z narzędziowych klas frameworka i konwersję przeprowadzić w następujący sposób.

```
SimpleDateFormat dateFormat =
            new SimpleDateFormat("dd-MM-yyyy");
CustomDateEditor editor = new CustomDateEditor(dateFormat, false);
binder.registerCustomEditor(Date.class, editor);
```

Taki fragment należało umieścić w metodzie kontrolera oznaczonej adnotacją `@InitBinder` lub nadpisując initBinder() jeżeli tworzymy kontroler poprzez rozszerzenia klasy `SimpleFormController`. Finalnie, uzyskiwany jest identyczny efekt jak w przypadku użycia adnotacji.

## Walidacja formularzy

W powyższym przykładzie zaznaczyłem, że obiekt `BindingResult` zawiera informacje o błędach, które wystąpiły podczas przetwarzania formularza, mapowania danych na obiekt itd. Pojawił się on już w poprzednich modułach, jednak jego wykorzystanie było niewielkie. W przypadku, jeżeli wykorzystujemy Spring MVC, pojawiają się pewne nowe możliwości – dodając nowy znacznik w widoku, umożliwiamy automatyczną propagację informacji o błędach do plików JSP. Każdy znacznik <form:errors /> wyszuka wiadomości odpowiednie dla atrybuty określonego w path i wyświetli je na ekranie.

```
<tr>
      <td>Task title</td>
      <td>
            <form:input path="title" />
            <span><form:errors path="title"/></span>
      </td>
</tr>
```

Właściwości tej możemy użyć do wyświetlenia błędów związanych z mapowaniem obiektów – z poprzedniego przykładu, jak też to samej walidacji. Wywołanie walidatora okazuje się być niezwykle proste – używamy adnotacji `@Valid`. Samo oznaczenie parametru kontrolera wystarczy aby framework w całości zająć się walidacją obiektu, przypisanie informacji o błędach a na końcu – wyświetleniem ich po stronie formularza.

```
	@RequestMapping(value="/tasks", method=RequestMethod.POST)
	public String saveTask(@Valid Task task, BindingResult results) {
		logger.info("Saving task: [" + task + "]");
		if (results.hasErrors()) {
			return "tasks";
		}

		service.addTask(task);
		return "redirect:/tasks";
	}
```

Co więcej, jeżeli w projekcie mamy skonfigurowany komponent `MessageSource` (a zostało to zrobione w poprzednim module), Spring samodzielnie wybierze odpowiednie wiadomości (np. opierając się o Locale zdefiniowane w przeglądarce).

```
<bean id="messageSource"
class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
    <property name="basename" value="classpath:messages" />
    <property name="defaultEncoding" value="UTF-8"/>
</bean>
```

## Alternatywa dla JSP

We wszystkich powyższych przykładach jako metody generowania widoków używaliśmy JSP, sposobu bardzo często krytykowanego, jednakże równie często doskonale sprawdzającego się właśnie ze względu na swoją prostotę. JSP to domyślny sposób tworzenia szablonów HTML w Spring MVC, jednakże nic nie stoi na przeszkodzie, aby użyć bardziej zaawansowanej technologii. Poprzez odpowiednią konfiguracją komponentu `ViewResolver`, można podać dowolne inne komponenty odpowiedzialne za szablony stron WWW.

Komponenty implementujące interfejs `ViewResolver` zajmują się odnajdywaniem odpowiedniego widoku na podstawie nazwy przekazanej przez kontroler – co jest generowane i w jaki sposób – jest już wewnętrzną odpowiedzialności komponentu. I tak, możliwa jest zmiana JSP na Tiles, Velocity lub też w ogóle rezygnacja z HTML i generowanie plików PDF lub Excel. Kontroler przekazuje tylko zestaw danych (model), a sama forma prezentacji w całości zależy od wybranego widoku.

## Spring jako komponenty biznesowe dla innych frameworków

Na samym początku modułu, analizując wygenerowany przez IDE szkielet, w pliku web.xml widzieliśmy zarówno `DispatcherServlet`, jak i implementacje `ContextListener` – oba komponenty zajmowały się tworzeniem kontekstu aplikacji webowej. Na potrzeby wszystkich dotychczasowych przykładów usunięta została klasa nasłuchująca i całość aplikacji oparta została o Spring MVC. Jak jednak pisałem, w sytuacjach, gdy Spring ma być tylko zestawem komponentów biznesowych, a warstwę webową pragniemy realizować w innej technologii (np. Struts lub JSF), konieczne staje się użycie `ContextListenera`.

Jeżeli przywrócimy w pliku web.xml wcześniej usunięty zapis, to wraz z wdrożeniem aplikacji na serwer, utworzony zostanie kontekst, na podstawie plików konfiguracyjnych zdefiniowanych w parametrze contextConfigLocation – w tym przypadku root-context.xml.

```
<!-- The definition of the Root Spring Container -->
<!-- shared by all Servlets and Filters -->
<context-param>
	<param-name>contextConfigLocation</param-name>
	<param-value>/WEB-INF/spring/root-context.xml</param-value>
</context-param>

<!-- Creates the Spring Container shared by all Servlets and Filters -->
<listener>
	<listener-class>
		org.springframework.web.context.ContextLoaderListener
	</listener-class>
</listener>
```

Jeżeli, analogicznie do poprzednich przykładów, zaimportujemy istniejące konfiguracje z poprzednich modułów (komponenty biznesowe aplikacji do zarządzania zadaniami), zostaną one załadowane do kontekstu. W tym momencie możemy zacząć korzystać z już istniejących komponentów, pobierając je z kontekstu przywołanego w sposób statyczny z wykorzystaniem klasy `WebApplicationContextUtils`. Aby to zademonstrować, utwórzmy przykładowy serwlet wyświetlający istniejące zadania.

```
<servlet>
	<servlet-name>taskServlet</servlet-name>
	<servlet-class>
		pl.devcastzone.todo.webapp.TaskServlet
	</servlet-class>
</servlet>

<servlet-mapping>
	<servlet-name>taskServlet</servlet-name>
	<url-pattern>/tasks</url-pattern>
</servlet-mapping>
```

```
public class TaskServlet extends HttpServlet {

	private static final long serialVersionUID = 1410303626350509200L;

	ApplicationContext ctx;

	@Override
	public void init(ServletConfig config) throws ServletException {
		ctx = WebApplicationContextUtils.
			getWebApplicationContext(config.getServletContext());
	}

	@Override
	protected void doGet(HttpServletRequest req, HttpServletResponse resp)
			throws ServletException, IOException {

		TasksService service = ctx.getBean(TasksService.class);

		List<Task> tasks = service.getTasks();

		resp.setContentType("text/html");
		PrintWriter out = resp.getWriter();
		out.println("<html>");
		out.println("<head>");
		out.println("</head>");
		out.println("<body>");
		out.println("<ul>");
		for(Task t: tasks) {
			out.print("<li>");
			out.print(t.getTitle());
			out.print(" ["+t.getDate()+" for ");
			out.print(t.getDuration()/1000/60/60);
			out.println("]");
		}
		out.println("</ul>");
		out.println("</body>");
		out.println("</html>");
		out.close();

	}

}
```

Bez większych problemów wykorzystaliśmy komponenty Springa (w tym dostęp do bazy danych), w „kontrolerze" zupełnie niezwiązanym z frameworkiem – tutaj najzwyklejszy serwlet. Zaprezentowany przykład z wykorzystaniem „czystego serwletu" jest oczywiście z punktu widzenia rozwoju aplikacji zupełnym nonsensem. Nikt nie będzie korzystał z zaawansowanego frameworku, takiego jak Spring, aby warstwę webową oprzeć na archaicznych serwletach. Jednak zamiast serwletów, a analogiczny sposób można użyć dowolnego innego silnika MVC – pozostawiając logikę w postaci komponentów Spring. Niezależnie od tego czy będzie to Struts, Tapestry czy jakikolwiek inny framework – połączenie z warstwą komponentów opartą o Spring następować będzie w analogiczny do opisanego sposób.

## Webservice

Pojęciem zbliżonym do aplikacji WWW, ale jednak trochę z boku, jest możliwość integracji z aplikacją poprzez web service. Spring, jak każdy ceniący się „framework enterprise" umożliwia upublicznienie pewnych metod właśnie w taki sposób. Spośród kilku różnych sposobów omówimy ten najpopularniejszy, czyli wykorzystujący elementy JSR-181: Web Service Metadata for Java Platform (zwanym także JAX-WS). Są to adnotacje `@Webservice` i `@Webmethod`, którymi możemy oznaczyć

metody biznesowe w celu ich upublicznienia i udostępnienia poprzez serwis webowy oparty na protokole SOAP.

Podobnie jak na platformie JEE (wykorzystując EJB 3.x), także w Springu zaleca się utworzenie dedykowanego komponentu, który deleguje logikę biznesową do komponentu, samemu zajmując się tylko i wyłącznie obsługą serwisu.

```
@Service
@WebService(serviceName="taskService")
public class TaskServiceEndpoint {

	@Autowired
	TasksService service;

	@WebMethod
	public Task[] getTasks() {
		List<Task> tasks = service.getTasks();
		return tasks.toArray(new Task[tasks.size()]);
	}

}
```

Endpoint oznaczony jest adnotacją @Service, ponieważ usługa webowa musi być pełnoprawnym, zarządzanym przez Spring komponentem. Drugim krokiem, jest dodanie do kontekstu komponentu `SimpleJaxWsServiceExporter`, który przeszukuje pakiety w poszukiwaniu adnotacji zgodnych z JSR-181 i udostępnia je poprzez podany w konfiguracji adres (np. http://localhost:9999/). Dodajemy to do pliku konfiguracyjnego właściwego albo dla `DispatcherServlet` lub (poprawniej) dla `ContextListenera`. W tym drugim przypadku, serwis będzie niezależny od aplikacji WWW którą wdrażamy na serwer. Uruchomiony zostanie osobny kontener `WebApplicationContext`, tylko i wyłącznie na potrzeby metod webowych.

```
<beans:bean
	class="org.springframework.remoting.jaxws.SimpleJaxWsServiceExporter">
	<beans:property name="baseAddress" value="http://localhost:9999/" />
</beans:bean>
```

Tak przygotowany serwis dostępny będzie pod adresem http://localhost:9999/taskService?wsdl.

copyright BNS MEDIA