Spring 2 Schema

Własne przestrzenie nazw w Spring 2.x

Osoby intensywnie wykorzystujące Springa w swoich aplikacjach często narzekają na ogrom konfiguracji XML koniecznej do utworzenia aplikacji o większym stopniu zaawansowania. Autorzy Spring Framework pisząc o dobrym oprogramowaniu promują zasadę DRY (Don't Repeat Yourself) – wyraźnie widać to w mechanizmie rozszerzania konfiguracji XML kontenera Springa. Ten artykuł wprowadzi Cię szybko w podstawowe techniki tworzenia własnych przestrzeni nazw XML Schema dla plików konfiguracyjnych Spring IOC.

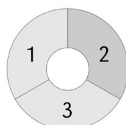
Dowiesz się:

- Jak za pomocą własnej przestrzeni nazw XML stworzyć najprostsze rozszerzenia składni plików konfiguracyjnych XML dla Spring Framework w wersji 2.x.

Powinieneś wiedzieć:

- Podstawowa znajomość języka Java oraz konfiguracja kontenera Inversion Of Control dla Spring Framework 2.x za pomocą plików XML;
- Wskazana jest również minimalna wiedza z zakresu języka XML oraz API programowej obsługi kontenera Spring IOC.

Poziom trudności



Spring jest popularnym frameworkiem IOC (*Inversion Of Control*) dla języka Java. Oferuje on kilka sposobów konfiguracji aplikacji oraz wstrzykiwania zależności, jednak najbardziej popularną techniką uzyskiwania tych celów jest konfiguracja w pliku XML. Jak powszechnie wiadomo XML oferuje czytelną, hierarchiczną strukturą prezentowanych przez siebie informacji, które to m.in. zapewniły mu rolę niepisanego standardu konfiguracji aplikacji webowych. XML jednakże cechuje się również tendencją do *gadatliwości* tzn. do konieczności opisywania zawartych w nim danych za pomocą dużej ilości tekstu. Programiści używający Springa w wersji niższej niż 2.0 skazani byli na używanie dość ograniczonego zestawu narzędzi redukujących rozmiar plików konfiguracyjnych XML oraz zjawiska tzw. *XML hell* (np. stosowanie dziedziczenia definicji beanów lub automatycznego wiązania). Na szczęście twórcy Springa od wersji 2.0 swojego frameworka dodali możliwość konfiguracji aplikacji za pomocą plików XML zdefiniowanych nie tylko zgodnie z DTD, ale i mechanizmem XML Schema. Oczywiście stare konfiguracje napisane pod kątem DTD w dalszym ciągu działają w wersji 2.0 Springa (ze względu na słynne umiłowanie kompatybilności wstecz przez jego autorów), jakkolwiek zalecaną formą konfiguracji XML jest teraz XML Schema.

Motywacja

Nie trzeba nawet specjalnie zachęcać do stosowania nowej definicji plików konfiguracyjnych XML, gdyż oferuje nam ona rozliczne korzyści o których użytkownicy DTD mogą tylko pomarzyć. Zaliczyć do nich należy głównie usprawnione uzupełnianie składni w edytorach XML dostępnych w popularnych IDE oraz zestaw dodatkowych przestrzeni nazw włączonych do dystrybucji Springa począwszy do wersji 2.0. W szczególności predefiniowane przestrzenie nazw okazują się być przydatne w codziennej pracy – pozwalają one m.in. na zastąpienie rozbudowanych definicji fabryk typu FieldRetrievingFactoryBean, ListFactoryBean lub TransactionProxyFactoryBean ich kompaktowymi odpowiednikami wyrażonymi za pomocą określonej przestrzeni nazw.

Twórcy Springa nie poprzestali na szczęście na dostarczeniu nam zestawu predefiniowanych przestrzeni nazw, ale udostępnili również mechanizm do samodzielnego tworzenia tychże. Nie muszę chyba wspominać o tym jak bardzo konfiguracja naszej aplikacji zyska na czytelności po zastosowaniu elementów oraz atrybutów własnej konstrukcji, dostosowanych do

Listing 1. *Szkielet konfiguracji kontenera Springa oparty na XML DTD*

```
<!-- Stara konfiguracja -->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
"http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
</beans>
```

Listing 2. *Szkielet konfiguracji kontenera Springa oparty na XML Schema*

```
<!-- Nowa konfiguracja -->
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/
                        beans/spring-beans-2.0.xsd">
</beans>
```

naszych konkretnych potrzeb. Zysk ten będzie widoczny w szczególności jeżeli planujemy wielokrotne wykorzystywanie naszego kodu w aplikacjach używających np. pisanej przez nas biblioteki. W artykule tym chciałbym właśnie przedstawić podstawowe techniki rozszerzania możliwości konfiguracyjnych XML w Springu.

Migracja na nowy styl konfiguracji

Przejście na nowy styl konfiguracji jest bardzo proste. Wystarczy zamienić w istniejących plikach konfiguracyjnych parę pierwszych linijek dokumentu.

Migracja zaprezentowana na Listingu 2 jest niemalże bezbolesna. Niemalże, gdyż istnieje parę (dosłownie parę) szczegółów konfiguracji które zostały uznane przez twórców Springa za przestarzałe (deprecated) i które należy w związku z tym dostosować do nowej konfiguracji opartej na XML Schema. Do szczegółów tego typu należy np. atrybut `singleton` elementu `<bean>`, który w nowej konfiguracji powinien zostać zamieniony na atrybut `scope` (ze względu na wprowadzenie mechanizmu rozszerzalnych zasięgów w nowszych wersjach Springa).

Witaj przestrzenio – przypadek użycia

Na początku spróbujemy napisać jak najszybciej najprostszą przestrzeń nazw z jednym elementem. Na potrzeby przykładów minimalnych zawartych w tym artykule załóżmy, że pracujemy dla firmy Foo i naszym zadaniem jest stworzenie elementu XML który pozwoli nam na bardziej efektywne używanie komponentu `Bar` w aplikacjach korzystających ze sprzedawanej przez nas biblioteki. Wyobraźmy sobie, że docelowo nasz klient chciałby konfigurować swoją aplikację za pomocą komponentu `Bar` w następujący sposób – Listing 3.

Oczekujemy, że powyższa konfiguracja utworzy w kontenerze instancję klasy `Bar` o `id` równym `bar` oraz wartości prywatnej właściwości `value` równej `barValue`. Sam komponent `Bar` wygląda następująco – Listing 4.

Schemat kroków

Klasyczny schemat kroków wykonywany podczas tworzenia własnej przestrzeni nazw to kolejno:

- napisanie dokumentu XSD;
- stworzenie parserów dla nowych elementów lub atrybutów XML;
- zaprogramowanie własnej klasy typu `NamespaceHandler`, która pozwoli nam na zmapowanie parserów i dekoratorów do określonych elementów i atrybutów XML;
- dodanie do wynikowej aplikacji lub biblioteki meta-danych potrzebnych Springowi.

W kolejnych sekcjach opiszę szerzej każdy z tych kroków.

Listing 3. *Przykład użycia komponentu z własnej przestrzeni nazw*

```
<?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:myns="http://www.foo.com/customSchema"
      xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/
                       beans/spring-beans-2.0.xsd
http://www.foo.com/customSchema http://www.foo.com/customSchema.xsd">

    <myns:bar id="bar" value="barValue" />

</beans>
```

Listing 4. *Kod przykładowego komponentu Bar*

```
package foo;
public class Bar {
    // wartość komponentu
    private String value;
    public Bar(String value) {
        this.value = value;
    }
    @Override
    public String toString() {
        return value;
    }
}
```

Listing 5. *Dokument XML Schema definiujący przykładową przestrzeń*

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.foo.com/customSchema"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:beans="http://www.springframework.org/schema/beans"
    targetNamespace="http://www.foo.com/customSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">

    <xsd:import namespace="http://www.springframework.org/schema/beans" />

    <xsd:element name="bar">
        <xsd:complexType>
            <xsd:complexContent>
                <xsd:extension base="beans:identifiedType">
                    <xsd:attribute name="value" type="xsd:string" use="required" />
                </xsd:extension>
            </xsd:complexContent>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

W Sieci

- Strona projektu Spring Framework – *http://springframework.org*
- Spring 2.5 API – *http://static.springframework.org/spring/docs/2.5.x/api*
- Konfiguracja oparta na XML Schema w Spring 2.5 – *http://static.springframework.org/spring/docs/2.5.x/reference/xsd-config.html*
- Własna przestrzeń nazw wg Spring 2.5 Reference – *http://static.springframework.org/spring/docs/2.5.x/reference/extensible-xml.html*
- XML Schema – *http://www.w3.org/XML/Schema*
- XMl DOM – *http://www.w3.org/DOM*

Dokument XSD

Dokument XML Schema pozwoli nam określić składnię naszej przestrzeni nazw, czyli m.in. jakie elementy oraz atrybuty są w niej dopuszczalne, a jakie nie. Schema uwzględnia również informacje o tym jakie typy danych zawarte są w poszczególnych elementach i atrybutach naszej przestrzeni oraz czy są one obowiązkowe. Interesujący nas schemat mógłby mieć następującą postać – Listing 5.

Tematyka tworzenia definicji XML Schema jest poza zakresem tego artykułu. Ograniczę się zatem do podsumowania, że zdefiniowaliśmy właśnie element o nazwie `bar` zawierający jeden (wymagany) atrybut o nazwie `value`. Element ten dziedziczy również po identyfikowalnym typie Springa, co w praktyce oznaczna tyle, że również `implicite` dziedziczy atrybut `id`. Następnym krokiem zbliżającym nas do działającej przestrzeni nazw będzie zmapowanie elementu XML zdefiniowanego powyżej do Javy.

Listing 6. *Przykładowy parser komponentu Bar*

```
package foo;
import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.xml.AbstractSingleBeanDefinitionParser;
import org.w3c.dom.Element;
public class BarParser extends AbstractSingleBeanDefinitionParser {
    @Override
    // Oczekujemy instancji klasy Bar
    protected Class<Bar> getBeanClass(Element element) {
        return Bar.class;
    }
    @Override
    protected void doParse(Element element, BeanDefinitionBuilder builder) {
        // Odczytaj wartość atrybutu
        String value = element.getAttribute("value");
        //  i dodaj ją jako argument konstruktora wynikowej instancji
        builder.addConstructorArgValue(value);
    }
}
```

Listing 7. *Handler mapujący przykładowy element 'bar' do wybranego parsera*

```
package foo;
import org.springframework.beans.factory.xml.NamespaceHandlerSupport;
public class BarNamespaceHandler extends NamespaceHandlerSupport {
    @Override
    public void init() {
        // zarejestruj instancję parsera dla
        // elementów <bar/> z tej przestrzeni nazw
        registerBeanDefinitionParser("bar", new BarParser());
    }
}
```

Listing 8. *Zawartość pliku META-INF/spring.handlers*

```
http\://www.foo.com/customSchema=foo.BarNamespaceHandler
```

Listing 9. *Zawartość pliku META-INF/spring.schema*

```
http\://www.foo.com/customSchema.xsd=foo/foo.xsd
```

Listing 10. *Kod minimalnego komponentu 'Bar' rozszerzonego o listę swoich potomków*

```
package foo;
import java.util.List;
public class Bar {
    private String value;
    private List<Bar> children;
    public Bar(String value) {
        this.value = value;
    }
    @Override
    public String toString() {
        return value;
    }
    public void setChildren(List<Bar> children) {
        this.children = children;
    }
    public List<Bar> getChildren() {
        return children;
    }
}
```

Parser elementu

Aby przekonwertować element XML do instancji zarejestrowanej w kontenerze musimy napisać parser implementujący interfejs `org.springframework.beans.factory.xml.BeanDefinitionParser`. Klasy implementujące wspomnianego interfejsu służą do analizy pojedynczego elementu XML znajdującego się bezpośrednio między elementami `<beans></beans>`, wraz z ew. podelementami znajdującymi się wewnątrz naszej definicji. Nasz konkretny element `<bar>` jest raczej prosty tzn. nie zawiera zagnieżdżonych elementów i zwróci tylko jedną instancję (klasy `Bar`) do rejestracji w kontrolerze. Ograniczmy się zatem do dziedziczenia z klasy `AbstractSingleBeanDefinitionParser` – Listing 6.

Klasa `BeanDefinitionBuilder` jest jedną z najważniejszych klas programowej obsługi kontenera Springa.

Temat ten zasługuje na co najmniej parę osobnych artykułów, tak więc ograniczę się tylko do podania informacji, że klasa ta służy do programowego tworzenia instancji klasy przeznaczonej do rejestracji w kontenerze. W tym konkretnym przykładzie chcemy, aby nowy obiekt klasy `Bar` został stworzony za pomocą jednoargumentowego konstruktora.

Zarejestrowanie elementu w NamespaceHandler

Instancje interfejsu `org.springframework.beans.factory.xml.NamespaceHandler` służą do łączenia przestrzeni nazw z parserami elementów oraz atrybutów XML. Sam `NamespaceHandler` nie wykonuje logiki związanej z analizą elementów i atrybutów XML, ponieważ całą logikę związaną z czytaniem dokumentu XML zawarliśmy w poprzednim kroku (w definicji parsera). `NamespaceHandler` oferuje parę metod pozwalających na interakcję z nim w określonych momentach jego cyklu życia, jakkolwiek na nasze skromne potrzeby zainteresujemy się jedynie bezargumentową metodą `init()` wywoływaną w momencie inicjalizacji handlera.

W powyższym przykładzie (Listing 7) zmapowaliśmy parser napisany w poprzednim kroku do elementu XML `<bar>`. W miażdżącej większości przypadków jeden handler bę-

dzie mapował wiele parserów i/lub dekoratorów –nie implementujemy wielu handlerów dla jednej przestrzeni nazw. Jak wspomniałem wcześniej nasze potrzeby względem handlera są bardzo skromne – dlatego właśnie dziedziczymy z klasy `org.springframework.beans.factory.xml.NamespaceHandlerSupport`. Dzięki wspomnianej klasie pomocniczej możemy nadpisać metodę `init()` interfejsu oraz nie martwić się o logikę związaną z rejestracją parsera.

Meta-dane dla Springa

Ostatnim krokiem jest dodanie do korzenia ścieżki klas naszej aplikacji lub biblioteki folderu META-INF wraz z plikami *spring.handlers* oraz *spring.schemas* – Listing 8 i Listing 9.

Jak widać pierwszy z nich mapuje URI przestrzeni nazw na napisany przez nas wcześniej handler – tą metodą Spring wie których parserów powinien użyć dla poszczególnych elementów i atrybutów w określonej przestrzeni nazw. Drugi zaś wskazuje fizyczną lokalizację pliku XSD z definicją naszej przestrzeni – w tym konkretnym przypadku plik z definicją powinien znajdować się w pakiecie *foo* i nazywać się *foo.xsd*. Pragę zwrócić uwagę na konieczność poprzedzania znaków dwukropka w obydwu plikach znakiem *backslash* (ze względu na fakt, że nie tylko znak równości, ale i dwukropek są poprawnymi symbolami końca klucza w plikach właściwości Javy).

Zagnieżdżanie elementów – rozszerzony przypadek użycia

Powyższy przykład jest nieco prosty, gdyż zakłada że sparsowany element XMLa zarejestruje tylko jedną instancję określonej klasy w kontrolerze. Jeżeli chcielibyśmy rekurencyjnie parsować i rejestrować zagnieżdżone elementy XML powinniśmy stworzyć parser dziedziczący po czymś potężniejszym niż `AbstractSingleBeanDefinitionParser` – tą klasą jest np. `AbstractBeanDefinitionParser`. Wyobraźmy sobie, że nasz przypadek użycia rozszerzymy w następujący sposób – otóż nasz potężny komponent `Bar` został rozszerzony o listę swoich *dzieci* (dla uproszczenia przykładu również instancji klasy `Bar`) – Listing 10.

Przykładowy scenariusz konfiguracji drzewa komponentów `Bar` w pliku XML mógłby wyglądać następująco – Listing 11.

Powyższa zmiana konfiguracji pociąga za sobą oczywiście konieczność aktualizacji dokumentu XML Schema – Listing 12.

Poza schematem XML w stosunku do poprzedniego przykładu zmianie uległby w zasadzie tylko sam parser – Listing 13.

Powyższy parser mógłby dziedziczyć po klasie `AbstractSingleBeanDefinitionParser` i dalej zwracać wynik zgodny z zadanym scenariuszem, jakkolwiek dziedziczenie po `AbstractBeanDefinitionParser` pozwala na jego nieco czytelniejszą implementację.

Listing 11. *Przykład użycia zagnieżdżonego komponentu*

```
<myns:bar id="bar" value="parent">
      <myns:bar value="child1">
            <myns:bar value="grandChild" />
      </myns:bar>
      <myns:bar value="child2" />
</myns:bar>
```

Listing 12. *Uaktualniony dokument XML Schema uwzględniający zagnieżdżanie elementów*

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.foo.com/customSchema"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.foo.com/customSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">

    <xsd:element name="bar">
        <xsd:complexType>
            <xsd:choice minOccurs="0" maxOccurs="unbounded">
                <xsd:element ref="bar" />
            </xsd:choice>
            <xsd:attribute name="id" type="xsd:ID" />
            <xsd:attribute name="value" use="required" type="xsd:string" />
        </xsd:complexType>
    </xsd:element>

</xsd:schema>
```

Listing 13. *Parser elementu 'bar' uwzględniający rekurencję*

```
package foo;
import java.util.List;
import org.springframework.beans.factory.support.AbstractBeanDefinition;
import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.support.ManagedList;
import org.springframework.beans.factory.xml.AbstractBeanDefinitionParser;
import org.springframework.beans.factory.xml.ParserContext;
import org.springframework.util.xml.DomUtils;
import org.w3c.dom.Element;
public class BarParser extends AbstractBeanDefinitionParser {

    protected AbstractBeanDefinition parseInternal(Element element,
            ParserContext parserContext) {
        // Stwórz budowniczego definicji elementu
        BeanDefinitionBuilder barBuilder = BeanDefinitionBuilder
                .rootBeanDefinition(Bar.class);
        barBuilder.addConstructorArgValue(element.getAttribute("value"));

        // Wyszukanie dzieci elementu określonego typu
        List<Element> childElements = DomUtils.getChildElementsByTagName(
                element, "bar");
        if (childElements != null) {
            // Dodanie potomków do listy zarządzanej przez kontener
            ManagedList children = new ManagedList(childElements.size());
            for (Element e : childElements) {
                // Rekurencja
                children.add(parseInternal(e, parserContext));
            }
            barBuilder.addPropertyValue("children", children);
        }
        return barBuilder.getBeanDefinition();
    }

}
```

Własne atrybuty – przypadek użycia

Ostatnim istotnym przypadkiem użycia jest dodanie własnego atrybutu do dowolnego elementu w pliku konfiguracyjnym – Listing 14. Załóżmy, że chcemy aby podczas parsowania elementu oznaczonego atrybutem `myns:bar` (o dowolnej wartości) na konsoli pojawiała się wartość licznika wskazującego ile elementów oznaczonych identyczną wartością znaleziono dotychczas w kontenerze.

Listing 14. *Przykład użycia atrybutu XML z własnej przestrzeni nazw*

```
<bean class="java.lang.String" myns:bar="barValue" >
    <constructor-arg value="someValue"/>
</bean>
```

Listing 15. *Dokument XML Schema dla minimalnego atrybutu*

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.foo.com/customSchema"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.foo.com/customSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">

    <xsd:attribute name="bar" type="xsd:string" />

</xsd:schema>
```

Listing 16. *Parser minimalnego atrybutu z własnej przestrzeni*

```
package foo;
import java.util.HashMap;
import org.springframework.beans.factory.config.BeanDefinitionHolder;
import org.springframework.beans.factory.xml.BeanDefinitionDecorator;
import org.springframework.beans.factory.xml.ParserContext;
import org.w3c.dom.Attr;
import org.w3c.dom.Node;

public class BarParser implements BeanDefinitionDecorator {
    HashMap<String, Integer> counters = new HashMap<String, Integer>();
    @Override
    public BeanDefinitionHolder decorate(Node node,
            BeanDefinitionHolder holder, ParserContext ctx) {
        // odczytaj wartość atrybutu
        String value = ((Attr) node).getValue();
        // sprawdź wartość licznika
        Integer i = counters.get(value);
        if (i == null) {
            i = 0;
        }
        // zwiększ licznik
        counters.put(value, ++i);
        // wyświetl wynik
        System.out.println(value + ": " + i);
        return holder;
    }
}
```

Listing 17. *Handler mapujący parser do atrybutu 'bar'*

```
package foo;

import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class BarNamespaceHandler extends NamespaceHandlerSupport {

    @Override
    public void init() {
        registerBeanDefinitionDecoratorForAttribute("bar", new BarParser());
    }

}
```

Zmiany w implementacji

Jak łatwo się domyśleć na pierwszy ogień pójdzie aktualizacja schematu XML – Listing 15.

Podobnie jak w przypadku własnych elementów, dla nowego atrybutu również musimy napisać własny parser. Możemy to wygodnie zrobić poprzez implementację interfejsu `BeanDefinitionDecorator` – Listing 16.

Tym razem będziemy również musieli zmodyfikować `BarNamespaceHandler`.

Wynika to z faktu, iż tym razem nie rejestrujemy parsera definicji *beana*, a tylko tzw. dekorator, czyli mniejszy parser służący do modyfikacji lub wzbogacania wybranego fragmentu definicji. Oczywiście wyświetlanie informacji o tym którą z kolei definicję oznaczoną konkretną wartością parsuje kontener, jest w realnym świecie średnio przydatne, jakkolwiek dostęp do parametrów `BeanDefinitionHolder` oraz `ParserContext` daje nam możliwość manipulacji kolejno definicją dekorowanego elementu (możemy również zwrócić całkowicie nową instancję zamiast modyfikować istniejącą) oraz całym rejestrem kontenera. Niestety ze względu na ograniczony rozmiar niniejszego artykułu jestem zmuszony przemilczeć szczegóły dotyczące możliwości programowej manipulacji definicjami beanów oraz zawartością kontenera.

Podsumowanie

W niniejszym artykule poznaliśmy podstawowe sposoby rozszerzania możliwości konfiguracyjnych plików XML w Springu za pomocą własnych przestrzeni nazw. Znając sposoby na analizę elementów XML najwyższego poziomu (wraz z ich ew. zagnieżdżeniami) oraz atrybutów XML możemy rozpocząć tworzenie własnych przestrzeni nazw, które przy minimalnej znajomości XML DOM oraz API programowej manipulacji zawartością kontenera IOC Springa pozwolą nam na napisanie w pełni funkcjonalnych mapowań dla naszych komponentów. Tematyka tego artykułu powinna w szczególności zainteresować osoby dostarczające dla innych firm programistycznych gotowe rozwiązania oparte na Springu ze względu na zminimalizowanie wiedzy wymaganej przez klienta do poprawnego stosowania naszych komponentów.

HENRYK KONSEK

Autor pracuje na stanowisku projektanta JEE w warszawskiej firmie Artegence. Udziela się również jako administrator serwisu javablackbelt.com. W wolnym czasie interesuje się swoją Anią oraz m.in. psychiatrią oraz dobrymi technologiami.
Kontakt z autorem: www.hekonsek.pl lub hekonsek@gmail.com.