AGILNÍ VÝVOJ WEBOVÝCH APLIKACÍ V GRAILS

Tomáš Pitner
Masarykova univerzita, Fakulta informatiky, tomp@fi.muni.cz

ABSTRAKT:
Příspěvek ukazuje principy a možnosti rychlého agilního vývoje menších a středně velkých webových aplikací s použitím moderních nástrojů, jako jsou Grails, založených na kombinaci skriptovacích jazyků (Groovy) a silné aplikační platformy (Java EE, Spring).

KLÍČOVÁ SLOVA:
Agilní vývoj, webové aplikace, dynamické jazyky, Grails, Ruby on Rails, Java EE

AGILNÍ VÝVOJ WEBOVÝCH APLIKACÍ

Vývoj moderních aplikací zejména menšího a středního rozsahu se stále častěji odehrává podle scénářů agilního vývoje, který dokáže pružněji reagovat na zákaznické požadavky a na výsledek si nehraje, nýbrž ho dosahuje. Cílem tohoto článku ovšem není hodnocení agilních metodik, ale seznámení s nástroji umožňujícími agilní vývoj.

CO JSOU GRAILS?

Grails [Grails], [Brown, 2006] jsou moderním vývojovým prostředím redukujícím složitost vývoje Java EE (webových) aplikací na skutečné minimum. Umožní vygenerovat kostru aplikace, spustit ji v předpřipraveném prostředí a dále rozvíjet.

Grails jsou silně inspirovány *úspěšnými Ruby on Rails* [RoR], ovšem základem je jazyk *Groovy*, což je jakási skriptovací obdoba Javy (viz dále), a populární Java EE rámec *Spring* [Pitner, 2004].

Cílem je zkrátit vývoj a nenutit vývojáře k opakování čehokoli (princip *DRY – Don't Repeat Yourself*), přičemž se ale na rozdíl od Ruby on Rails používá robustní a výkonné běhové zázemí třídy Java EE.

Již po několika málo úkonech vytvoříme v Grails – přímo z příkazové řádky nebo v některém z populárních IDE prostředí (Eclipse, NetBeans, IDEA) – poměrně impresivně působící prototyp aplikace, viz ukázka 1.

Ukázka 1. Vizáž aplikace v Grails

Prostředí jako Grails neignorují ani velké firmy jako je *Oracle*, viz [Grall, 2006] nebo *BEA* [Harshad, 2006].

DYNAMICKÉ JAZYKY

Rámec Grails je postavený na programovacím skriptovacím jazyku Groovy [Subramanian, 2008], což je dynamický jazyk moderní konstrukce určený k běhu pod JVM a syntakticky vycházející z jazyka Java. Programátor v Javě se zde bude cítit jako doma, na první pohled ho zaujme stručný styl zápisu a celá řada programovacích obratů, které se v Javě musejí řešit jen těžkopádně, viz ukázka 2. Příjemná je rovněž možnost použít stávající javový kód a knihovny. Jedinou podmínkou k provozování Groovy a následně Grails je instalované prostředí Java aspoň 1.4 (lze doporučit Java 5 nebo 6).

```
f = {k,v -> println "field=${k}, value=${v}" }
tomp = [name:"Tom", last:"Pitner", salary:14000.3]
tomp.each(f)
```

Ukázka 2. Program v Groovy

PRINCIPY A MODELY

Z pohledu architektury výsledné aplikace nepřinášejí agilní dynamická prostředí nic nového – budují totiž na osvědčených webových rámcích jako je v případě Grails rámec Spring. Rámec předepisuje, jak má být aplikace nakonfigurována, určuje tok zpracování klientských požadavků a poskytuje či zpřístupňuje celou řadu dnes standardně požadovaných služeb – od internacionalizace aplikace až po datovou perzistenci.

Aplikace v Grails respektují osvědčenou architekturu *MVC* (Model-View-Controller). Výchozí struktura aplikace je tvořena doménovými třídami, které představují *model* aplikace. Návazně na doménovou třídu k ní může být vygenerován zárodek *kontroléru* (controller), který nad objekty této třídy realizuje základní operace CRUD – create/read/update/delete, takže zbývá doprogramovat jen ty „zajímavější“ operace nebo upravit chování těch vygenerovaných. Pokročilejším vývojářům lze doporučit změnit si šablony pro generování kódu podle vlastních potřeb.

Jak bývá obvyklé, po přijetí klientského požadavku jej rámec Spring nasměruje na vykonání příslušné metody kontroléru, jejíž provádění je typicky zakončeno předáním dat (modelu) příslušnému *pohledu* (view), který zajistí zobrazení výstupů uživateli s webovým prohlížečem. Název a umístění kódu příslušného pohledu (obvykle realizovaného jako JSP nebo GSP stránka – obdoba JSP pro jazyk Groovy) je opět dáno konvencí, která se rovněž uplatní, když si *view* pro danou doménovou třídu necháme vygenerovat.

```
class User extends Resource {
   String login
   Content avatar
   Role role

   static searchable = true
   static belongsTo = [Role]
   static constraints = {
      login(unique:true, blank: false)
      avatar(nullable:true)
   }

   String toString() {
"#${id} ${name}, ${login} (role ${role})"
   }
}
```

Ukázka 3. Kód doménové třídy

VÝVOJ ŘÍZENÝ DOMÉNOU

Implementaci aplikace v Grails, stejně jako v obdobných prostředích typu Rails, zahajujeme (prototypovým) nadefinováním *modelu aplikace*, tzn. *doménových tříd*. Příkazem `grails generate-app <název_aplikace>` a následně `grails generate-domain-class <název_třídy>` zajistíme vygenerování kostry doménové třídy, kterou následně běžným editorem upravíme. Výsledek po úpravách vidíme na ukázce 3.

Dalším charakteristickým rysem Grails je intenzivní využívání *doménově specifických jazyků* (domain-specific languages, DSL), [Fowler, 2004] na popisy mnoha různých záležitostí, např. validačních pravidel pro ověřování přípustnosti hodnot atributů doménových objektů. Pravidla popíšeme a Grails se nám při manipulacích s objekty samy postarají o jejich dodržování.

OBJEKTOVĚ-RELAČNÍ MAPOVÁNÍ

```
class Author {
    static hasMany = [books:Book]
    String name
}

class Book {
    Author author
    String title
}
```

Ukázka 4. Mapování vztahů 1:N

Již od vzniku objektového paradigmatu se traduje spor, co je ideálním řešením persistence v informačních systémech budoucnosti. Programuje-li se dnes převážně objektově, převládnou nové objektové databáze postavené na konceptuálně stejných přístupech? Z mnoha důvodů (tradice, spolehlivost, lepší optimalizace, stávající databáze plné dat…) jsou ale relační databáze stále dobrým artiklem a situace se tak rychle nezmění – už proto, že objektoví vývojáři mají nyní nástroje, kterým objekty relativně snadno do relační databáze uloží. Tyto postupy a nástroje se označují jako *Object-Relational Mapping* a jsou v javovém světě reprezentovány jednak dnes už klasickými nástroji, jako jsou např. *Hibernate* [Hibernate] nebo *Toplink*, nebo je přístup k databázi „obalen“ standardizovaným rozhraním – *Java Persistence API*, za nímž je ORM nástroj skryt. Slabinou ORM přístupů může být jednak horší efektivita (přístup k databázi nemáme „stoprocentně v ruce, ve své moci“), jednak občas komplikované popisování mapování.

```
class UrlMappings {
    static mappings = {
      "/rest/$controller/$id?" {
            action = [GET: "rest_read",
                      POST: "rest_create",
                      PUT: "rest_update",
                      DELETE: "rest_delete"]
                                }
        "/$controller/$action?/$id?"{
            constraints { }
        }
      }
}
```

Ukázka 5. Konfigurace mapování URL na kontroléry a akce

V Grails se dá s pomocí mapování GORM (Grails ORM) druhé nevýhodě do značně míry vyhnout. GORM funguje zcela transparentně.

Aplikace v Grails ji prostě používá tak, že se nadefinují doménové třídy a deklarativně se popíší vztahy mezi nimi. Grails na pozadí vytvoří mapování pro Hibernate i samy vytvoří příslušnou databázi s tabulkami – pokud už existuje nebo jsou v ní dokonce data,

vytvářet se nebude. Na ukázce 4 je vidět, jak zachytíme vztah typu 1:N mezi objekty doménových tříd.

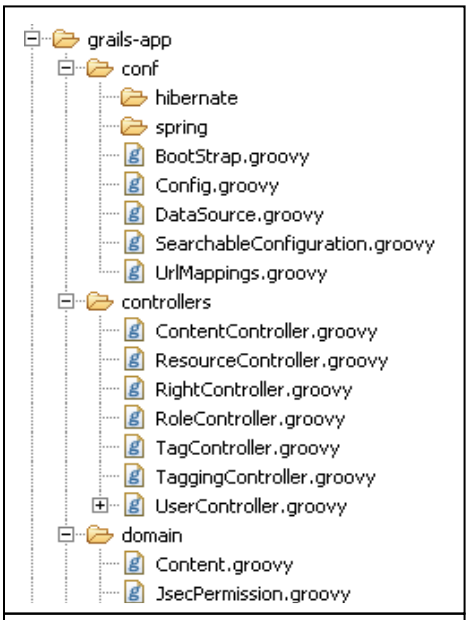
Přístup k datové perzistenci v GORM také pravděpodobně nejviditelněji využívá *dynamických vlastností* jazyka Groovy. Každý doménový objekt má dynamicky dostupné metody pro ukládání, mazání a vyhledání z databáze, kam se objekty díky mapování GORM (Grails Object-Relational Mapping), pod nímž běží osvědčený ORM nástroj Hibernate. Dynamická dostupnost metod znamená, že např. metoda `save` na uložení objektu do databáze neexistuje hotová v kódu každé doménové třídy, ale její volání je na „metaúrovni" rozpoznáno a metoda je dynamicky vytvořena.

Zcela unikátní je možnost použít metod *find*, zpřístupňující takřka kompletně datový dotazovací jazyk HQL (Hibernate Query Language). Např. `Employee.findBySalaryGreaterThan(20000)` jsou vráceni všichni zaměstnanci s platem vyšším než 20000, aniž bychom napsali jediný řádek této metody.

KONVENCE MÍSTO KONFIGURACE

Prakticky veškeré psaní kódu v Grails musí respektovat konvence, jež se týkají především *pojmenování* a *umístění* tříd. Už samotná struktura aplikace je předepsaná (či spíše předgenerovaná), jak je vidět na ukázce 6.

Použití konfiguračních XML souborů či popisovačů je dnes velmi časté, představuje ale spíše nutné zlo při tvorbě a nasazení velkých aplikací Java EE a aplikačních rámců. XML forma je sice strojově snadno zpracovatelná, ale špatně se píše ručně a je stejně třeba si syntaxi pamatovat. Grails nahrazují XML soubory buďto konvencemi zhruba ve smyslu: „je řečeno, kam se má co umístit a pak se to nemusí konfigurovat" nebo jsou konfigurace psány v doménově specifických jazycích, jejichž syntaxe je stručná, intuitivní a není XML – viz např. ukázka 5.



Ukázka 6. Struktura aplikace

SLUŽBY A ŘÍZENÍ ZÁVISLOSTÍ

Služby (services) představují v arzenálu Grails další sil-nou zbraň. Umožňují psát *komponenty* (znovu)použitelné aplikační logiky, které lze následně použít buďto v dalších službách, v doménových třídách, jejich kontrolérech, pohledech, knihovnách značek nebo testech.

Hlavní přínos spočívá v automatickém vkládání závislostí na službách – stačí, aby např.

```
class CountryService {
    def String sayHello(String name) {
        return "hello ${name}"
    }
}

class GreetingController {
    def countryService
    def helloAction = {
render(countryService.sayHello(params.name))
    }
}
```

Ukázka 7. Služba a její použití v kontroléru

kontrolér měl proměnnou s názvem služby a při jeho instanciaci se automaticky „sežene“ odkaz na instanci služby – buď už existuje, nebo se taková instance služby vytvoří (viz ukázka 7). Za řešením těchto závislostí (Dependency Injection) stojí opět rámec Spring.

Služba není fyzicky nic jiného, než třída psaná v Groovy s názvem končícím na *Service* a umístěná do adresáře *services*. Služba může mít i proměnné, ale je dobrým zvykem psát služby jako bezestavové komponenty kvůli lepší znovupoužitelnosti – za běhu totiž existují typicky v jedné instanci.

INTERNACIONALIZACE A LOKALIZACE

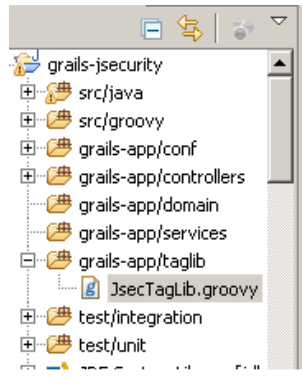
Java EE je naštěstí od počátku vybavena solidním zázemím pro tvorbu aplikací určených k lokalizaci, tzn. přizpůsobení národním zvyklostem uživatele. V Grails aplikaci stačí namísto pevných hlášek v programu používat odkazy na ně a skutečné národní jazykové varianty umístit do souborů v předepsaném adresáři. Zvolené národní prostředí uživatele (tzv. *Locales*) se pozná podle hlaviček požadavků (*Accept-Language*), které přicházejí z klientova prohlížeče nebo je lze ručně změnit zadáním parametru dotazu.

```
home=Home
create=Invoeren
edit=Wijzigen
update=Opslaan
delete=Verwijderen
delete.confirm=Ben je zeker?
```

Ukázka 8. Hlášení v národním jazyce NL

KNIHOVNY ZNAČEK

Knihovny značek (tzv. *Tag Libraries – taglibs*) k použití ve stránkách pohledů (tedy JSP a v Grails i *Groovy Server Pages* – GSP), jsou pro vývojáře běžné Java EE aplikace noční můrou. Vývojová prostředí sice dnes již umějí usnadnit práci s umístěním a konfigurací popisovačů knihoven i s tvorbou jejich kódu, ale nic nepřekoná jednoduchost Grails v tomto




```
import org.jsecurity.context.support.ThreadLocalSecurityContext

class JsecTagLib {
    static namespace = 'jsec'

    def isLoggedIn = {attrs, body ->
        // Get the user's security context.
        def securityContext = new ThreadLocalSecurityContext()

        if (securityContext.authenticated) {
            out << body()
        }
    }
```

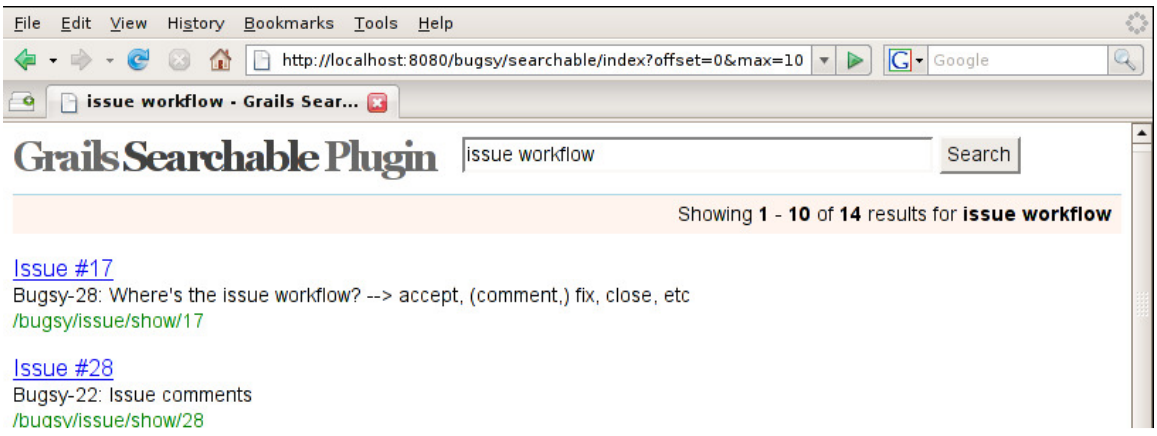
Ukázka 9. Knihovna značek JsecTagLib

ohledu. Knihovnou značek je každá třída umístěná v adresáři knihoven a každou značkou je metoda (tzv. *closure*), která dostane jako parametr tělo a atributy značky a vrátí její obsah, viz ukázka 9. Použití je opět triviální, stačí prostě značku v kódu GSP uvést; nalezení její definice se provede automaticky.

ZÁSUVNÉ MODULY

Zásuvné moduly (plugins) jsou silným mechanizmem Grails, umožňujícím výrazně měnit fungování aplikací, které modul použijí. Kromě toho, že moduly se mohou samy chovat jako aplikace – tzn. mohou mít své doménové třídy, kontroléry, pohledy a služby –, díky dynamické povaze Groovy mohou zásuvné moduly ovlivňovat chování aplikace, v níž jsou použity – např. vstupovat pomocí techniky *interception* do práce kontrolérů doménových tříd.

To dovoluje např. napsat plugin, který automaticky dodá do doménových tříd schopnost být indexován a plnotextově prohledáván – stačí, aby třída deklarovala, že chce být indexovaná a



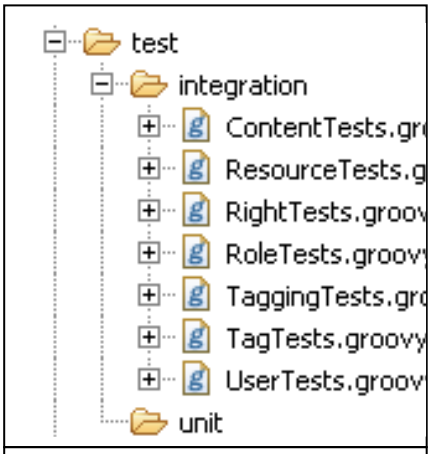
Ukázka 10. Práce zásuvného modulu 'searchable'

prohledávaná – o ostatní se postará plugin *searchable*, ukázka 10.

TESTOVÁNÍ

Agilní vývoj nelze realizovat bez důkladného testování vzniklých artefaktů [Pitner, 2006]. Grails nabízejí v tomto smyslu komfort očekávaný od prostředí postaveného na Java EE:

- Lze testovat *jednotky* pomocí standardních JUnit testů. Pro něj je v Grails kompletní podpora – stačí rozšiřovat třídy *GroovyTestCase* a umisťovat je do příslušného adresáře, viz ukázka 11. Kostry testů jsou dokonce automaticky vygenerovány při vytváření tříd – stačí tedy „jen“ napsat kód testovacích metod. Lze samozřejmě použít i v Javě napsané existující nebo nové testy jednotek.
- Pokud jde o integrační testy (tzn. chování programu mezi vrstvami aplikace), lze kontroléry testovat i *bez spuštění ve webovém prostředí* pomocí *zástupných objektů* (Mock) nahrazujících příslušné objekty Servlet API (HttpServlet, -Request, -Response,…), což oproti ostrému spouštění ve webovém kontejneru ušetří obrovské množství času.



Ukázka 11. Testy

CO JE ZA TÍM?

Šikovnou vlastností, která i začátečníkovi otevře dveře Grails, je kompletní předpřipravené vývojové a běhové prostředí, jenž buďto použijeme, anebo nahradíme něčím výkonnějším. K tomu, abychom aplikaci vyvinuli a spustili skutečně postačí jen nainstalovaná Java a distribuce Grails, které přicházejí i s vestavěnou relační databází HSQLDB postačující pro úvodní pokusy. Ve vhodnou chvíli může být v intuitivní konfiguraci nahrazena připojením k databázi samostatné.

Rovněž rámec Spring potřebný ke konfiguraci a běhu aplikace je přibalen, stejně jako úsporný webový kontejner *Jetty*. Grails však kvůli možnosti nasazení aplikace jinde umějí vyprodukovat běžný balíček *WAR* k nasazení na libovolný webový kontejner.

NEGATIVA

Abychom přílišný optimizmus trochu přibrzdili: je třeba přiznat, že Grails do dokonalosti mnoho chybí. Současná verze (1.0.1) sice už vyrostla z rudimentárních chyb svých předchůdkyň, trpí už méně syndromy překotného vývoje, jako byla chybějící dokumentace, či podivné faktory, které verze od verze způsobovaly nepříjemné regrese – nefunkčnost již jednou fungujících záležitostí. O to dříve si ovšem povšimneme již hůře odstranitelných chyb koncepčních. Např. chování aplikace se často deklarativně řídí uváděním nejrůznějších statických či instančních atributů u tříd, podle nichž např. zásuvný modul pozná, co se se třídou má dít – např. jak validovat data v ní (doménové třídy) nebo jak autorizovat přístup k metodám (třídy kontrolérů). Tyto atributy jsou obvykle deklarovány jen jménem bez typu a jména mohou snadno kolidovat např. s jinými atributy třídy nebo jiného zásuvného modulu. Rovněž starší verze měly problémy s umísťováním tříd do balíků (jmenných prostorů), což opět zvyšovalo riziko kolizí. I zde se ukazuje, že jmenné prostory jsou při návrhu jazyka či prostředí naprosto klíčovým požadavkem, jehož důsledné vymáhání hned od počátku je nutnost, na níž se, bohužel, i zde zapomnělo.

ZÁVĚR

Vývojovým nástrojům postaveným na robustních a osvědčených běhových platformách a přitom nabízejícím výhody dynamických programovacích jazyků přichází na chuť stále více vývojářů menších a středně velkých webových aplikací. Pionýrským a vývoj dosud určujícím prostředím bylo známé Ruby on Rails, jehož slabinou bohužel zůstává absence robustního běhového zázemí srovnatelného s Java EE aplikačními servery, což ukazují i výkonnostní testy [Roche et al, 2007].

Nástroj Grails sledující stejné principy a přitom postavený na jazyce Java s prověřeným běhovým prostředím daným rámcem Spring a řadou progresivních Java EE technologií a nástrojů (Hibernate, Sitemesh), je alternativou, která nabízí podobné výhody a navíc je dokáže bez problémů kombinovat s existujícím kódem pro Java EE.

Přes momentální nedospělost a řadu chyb jsou Grails příkladem přístupu, který bude v dalších letech s vysokou pravděpodobností určovat směr vývoje. Stojí proto za to se s ním přinejmenším seznámit.

PODĚKOVÁNÍ

Tento příspěvek vznikl za podpory projektu Národního programu Informační společnost „E-learning v kontextu sémantického webu“, č. 1ET208050401.

LITERATURA

[Brown, 2007] Brown, J. *Introduction To Grails*, http://www.ociweb.com/jnb/jnbMar2007.html

[Fowler, 2004] Fowler, M. *Domain Specific Language,* http://martinfowler.com/bliki/DomainSpecificLanguage.html

[Grails] *Vývojové prostředí Grails*, http://grails.org

[Grall, 2006] Grall, T. *Grails on Oracle for Java Developers*, http://www.oracle.com/technology/pub/articles/grall-grails.html

[Harshad, 2006] Harshad, O. *An Introduction to Groovy and Grails*, http://dev2dev.bea.com/pub/a/2006/10/introduction-groovy-grails.html

[Hibernate] *HIBERNATE - Relational Persistence for Java and .NET*, http://www.hibernate.org

[Pitner, 2006] Pitner, T. *Testování softwaru na platformě Java EE*, sborník konference Tvorba software, ČSSI a VŠB-TU, Ostrava, 2006

[Pitner, 2004] Pitner, T. *Webové aplikační rámce v Javě*, sborník konference Tvorba software, ČSSI a VŠB-TU, Ostrava, 2004

[Roche & Davis, 2008] Roche, G., Davis, S. *The Definitive Guide to Grails: Agile Java Web Development using this Groovy-based Framework*, Second Edition, APRESS, 2008

[Roche et al, 2007] Roche, G. et al. *Grails vs Rails Performance Benchmarking*, http://docs.codehaus.org/display/GRAILS/Grails+vs+Rails+Benchmark

[RoR] *Ruby on Rails: The Web Development That Doesn't Hurt*, http://www.rubyonrails.org

[Subramaniam, 2008] Subramaniam, V. *Programming Groovy: Dynamic Productivity for the Java Developer*, The Pragmatic Programmers, 2008