

# VIVA LA EVOLUCIÓN!

Peter Backlund

Mer dynamiskt, enklare, mindre repetition men mer konvention och inte minst roligare och mer produktivt: Peter Backlund har tittat närmare på Grails, Java-plattformens alldeles egna Rails.

En av de tydligaste trenderna inom webbutveckling de senaste tre-fyra åren har varit att det ska vara mycket enklare att bygga applikationer som matchar eller ligger nära den traditionella mallen, en treskiktssapplikation med databas i botten. Man har tröttnat på tungrodda komponent- och containerramverk, och på att behöva konfigurera saker som i de flesta fall är underförstådda, och dessutom på flera ställen. De två mest populära förkortningarna det pratas om är DRY - *Don't Repeat Yourself* och CoC - *Convention over Configuration*. Dessutom har dynamiska språk fått ett kraftigt uppsving, tillsammans med intresset för att konstruera domänspecifika språk.

Det språk och ramverk som i det här sammanhanget har ojämförligt högst profil är förstås Ruby och Rails, som är ett oerhört lättfotat och uttrycksfullt ramverk för att i första hand bygga ganska raka webbgränssnitt ovanpå en databas. Rails, och även Ruby med tillhörande bibliotek, utvecklas i rask takt mot bredare kompetens även om det ännu inte täcker in allt som finns inom Java EE. I den här artikeln ska vi titta lite närmare på ett högintressant projekt som heter Grails, ett Java- och Groovybaserat projekt som har en hel del likheter med Rails, men också en del viktiga skillnader som gör det väldigt intressant för Javautvecklare.

Grails, som i skrivande stund är väldigt nära att släppa version 1.0, är Apache-licenserat och har utvecklats sedan början på 2006 under ledning av framför allt Graeme Rocher. Groovyprofilen Guillaume Laforge var inblandad i projektets tillkomst, och nyligen startades företaget G2One av Graeme, Guillaume och Alex Tkachman från JetBrains för att tillhandahålla kommersiell support för Groovy och Grails. Upplägget liknar förhållandet mellan Spring och Interface21.

Vad som gör Grails speciellt är att det inte är ett från grunden nytt ramverk, utan snarare ett slags dynamiskt "klistre" mellan ett antal beprövade Java-ramverk:

- Spring, för hantering av beroenden (*dependency injection*), transaktioner och MVC-webbframverket, och mycket annat,
- Hibernate, för domänmodell-till-databas-mappning (*Object-Relational Mapping*).
- Sitemesh, för dekoration av vyer.

Därutöver finns ett system för plugins, som snabbt växer i antal. Indexmotorn Lucene, schemaläggaren Quartz och säkerhetsramverket Acegi är tre exempel där det finns plugins. Själva Grailsplattformen består till ca 80 % av Javakod och 20 % Groovy.

Det här medför att det är väldigt lätt att integrera med befintliga Javakomponenter och -bibliotek, eller över huvud taget vad som helst som går att exponera som en böna i en Spring-kontext. Groovy har ju som bekant heller inga problem att anropa Java. Dessutom får man omedelbart tillgång till all den kraft och flexibilitet som finns i exempelvis Hibernate, med cachning och mappning mot gamla och komplicerade databasschema, drivrutiner och dialekter för obskyra databaser och så vidare. Slutligen blir steget inte särskilt långt för den som redan är bekant med något eller flera av de här ramverken - och det är rätt många - när man kan utnyttja mycket av den kunskap redan har.

Så hur är det då att utveckla i Grails? Den som tittat på Ruby on Rails kommer att känna igen en del saker, exempelvis har man tre profiler: utveckling, test och produktion, med var sin datakälla bland annat. Dessutom finns samma typ av scaffolding-funktionalitet, som givet en entitet i en domänmodell (eller en tabell i databasen, i Rails fall) automatiskt kan skapa MVC-controller och vyer för de vanliga CRUD-operationerna (skapa, läsa, uppdatera, radera) samt listning och paginering. I huvudsak kan man också se ändringar omedelbart efter en omladdning i webbläsaren utan att ladda om applikationen, förutom när man gjort förändringar i domänmodellen. En viktig skillnad i Grails är dock att man driver domänmodellen från Groovy-klasserna och låter Hibernate generera schemat, istället för att skriva databasdefinitionen för att skapa domänklasserna.

## PERSISTENS OCH DOMÄNMODELL

En enkel Person-entitet ser ut så här i Grails, när den ligger i katalogen grails-app/domain

```
class Person {
  String name
}
```

Det är allt som behövs för att persistera klassen. JavaBean-properties är inbyggda i Groovy, alla entiteter får en identitetsegenskap med namnet "id" och typen Long. Hibernate instrueras att mappa klassen och genererar en tabell som heter "Person" med kolumnerna "id", "name" och "version", där "id"-kolumnen är autoinkrementerad, och "version"-

kolumnen används för optimistisk låsning. En sak som används flitigt i Grails är statiska medlemmar av GroovyBuilder-typ (som bygger på closures), och kan betraktas som en sorts domänspecifikt språk. Specialkonfigurering av Hibernate är ett exempel, och validering är ett annat. Här definerar vi några ramar (*constraints*) för vår Person-klass:

```
class Person {
  String name
  static constraints = {
    name(range:4..100,match:"^[A-Z][a-z]*$")
  }
}
```

Det här säger att namnet måste vara mellan fyra och 100 tecken långt, och att det ska börja med en stor bokstav och sedan vara små bokstäver. Standard är att ingen property tillåts vara null, vill man tillåta det sätter man "nullable:true" i exemplet ovan. Det finns ett stort antal inbyggda ramar, och man kan själv bygga godtyckliga regler med hjälp av closures. Validering sker automatiskt när man sparar entiteten, eller så kan man själva anropa metoden validate() som dynamiskt byggs in i alla entiteter. Grails använder i likhet med RoR ActiveRecord-mönstret, dvs domänklassen hanterar själv sin persistens. Det ser alltså ut på det här viset:

```
def p = Person.find(1)
p.name = "Newname"
p.save()
```

Det finns även ett mycket finurligt sätt att skriva find-metoder med CamelCase och HQL (Hibernate Query Language). Om man till exempel vill ha en slagning där man letar efter ett visst namn och vill sortera på ålder, skriver man helt enkelt:

```
def peters = Person.findByNameOrderByAge("Peter")
```

Man behöver däremot inte implementera den här slagningen alls! Grails kommer att använda CamelCase-konventionen för att tolka ovanstående som följande HQL-fras:

```
"from Person where name = ? order by age"
```

Inte nog med det, den här nya metoden kommer dessutom dynamiskt att läggas till i Person-klassdefinitionen så att (den relativt kostsamma) tolkningen inte behöver ske igen. Självklart kan man även jobba med handskrivna HQL direkt mot Session-interfacet, Criteria-API:t eller till och med rå SQL. Kort sagt, allt man kan göra med Hibernate - eller rättare sagt med Java - kan man göra i Grails, om behovet uppkommer.

Alla typer av relationer mellan domänklasser stöds, och om man exempelvis säger att en person har ett hus och flera husdjur, ser det ut så här:

```
class Person {
    String name
    House house

    static hasMany = [pets : Animal]
}
```

Relationen har underförstått mängdsemantik (Set), vill man ha ändra det lägger man till ett fält med önskad typ och samma namn, exempelvis SortedSet eller List.

```
...
List pets
static hasMany = [pets : Animal]
```

## TRANSAKTIONER

Det finns två förenklade sätt att spänna upp en transaktion kring flera operationer. Det ena är att skapa en klass vars namn slutar på "Service", med en flagga som aktiverar transaktioner:

```
class PartnershipService {
    static transactional = true
    void marryTwoPeople(Person one, Person theOther) { ... }
}
```

Alla metoder i klassen kommer då att kläs in i Springs transaktionslogik, och fungerar precis som det gör i vanliga fall. Grails stöder dependency injection (utifrån egenskapsnamn, *autowire-by-name*) även i domänklasser, så man kan till exempel göra så här för att ge domänmodellen ett rikt beteende:

```
class Person {
    PartnershipService partnershipService
    void marry(Person partner) {
        partnershipService.marryTwoPeople(this, partner)
    }
}
```

Det andra sättet är att helt enkelt skicka in en closure i den speciella domänklassmetoden withTransaction:

```
Person adam, eve, steve
```

```
[...]
Person.withTransaction { tx ->
    adam.divorce(eve)
    adam.marry(steve)
    if (anyObjections()) {
        tx.setRollbackOnly()
    }
}
```

Parametern tx är en instans av Spring-interfacet TransactionStatus, så att man kan se transaktionens status och eventuellt programmatiskt markera den för rollback, som i det här exemplet.

## WEBBLAGRET

Grails använder Spring MVC i bakgrunden, men det märker man inte alls när man skriver sina controllers. I likhet med Rails finns full scaffolding, och mappningen mellan controller och URL sköts med namnkonvention (även om det är enkelt att skriva egna mappningsregler). En minimal controller som hanterar CRUD-operationer för Person-klassen ser ut så här:

```
class PersonController {
    def scaffold = Person
}
```

och får automatiskt ett standardbeteende, inklusive HTML-vyer, för anrop mot URL-er som "/person/show/1" eller "/person"list" och så vidare. Värdet av scaffolding sjunker förstås ju längre utvecklingsarbetet fortskrider, men tekniken kan vara bra för att snabbt få ihop något grundläggande att utgå från. De "inbyggda" controllermetoderna och vyerna går att generera till filer för att jobba vidare med. För att exempelvis visa alla med personer med samma namn som en person med id 10 under URL-en "/person/namesakes/10", lägger vi till följande metod:

```
def namesakes = {
    def p = Person.get(params.id)
    def ns = Person.findByName(p.name).minus(p)
    [person:p, namesakes:ns]
}
```

Här ser man en av många dynamiska medlemmar i en Grails-controller: params. Det är som väntat en hashtabell (Map) av parametrarna och deras värden. Andra medlemmar är bland annat request, response och session, samt ett antal medlemmar för introspektion av aktuell action och controller. På den andra raden använder vi först en dynamisk finder som

vi bekantade oss med tidigare, samt ett av Groovys tillägg till Javas standardbibliotek - `minus()`, applicerat på en lista, för att undvika att vi listar den första personen som namne till sig själv. Slutligen returnerar vi en modell som innehåller personen och hans/hennes "namnar".

Alla Grails-controllers jobbar med ModelAndView-konceptet i Spring MVC, med ett antal konventioner för att förenkla returvärdena. I exemplet ovan kommer vynamnet "namesakes" att användas, eftersom inget annat sägs. Ett annat centralt begrepp i webblagret är `render`-metoden, som är en oerhört kraftfull allt-i-allo för att generera innehåll. Den kan användas för att rätt och slätt skriva text:

```
render "Hello"
```

eller för specificera vy och modell:

```
render (view: "friendsOf", model: [person: Person.get (params.id)])
```

eller för att rendera markup-strukturer som exempelvis XML:

```
render (contentType: "text/xml") {  
  people (count: Person.count ()) {  
    Person.list ().each { p->  
      person (name: p.name, p.storyOfMyLife ())  
    }  
  }  
}
```

som producerar följande dokument:

```
<people count="3">  
  <person name="Peter Backlund">I was born in 1977...</person>  
  [...]  
</people>
```

Mycket användbart i AJAX-sammanhang, eftersom även JSON stöds av `render`-metoden. Här såg vi för övrigt ytterligare ett exempel på `GroovyBuilder`-konceptet.

Det finns även stöd för `default-action`, kedjning och `redirect` mellan olika actions. Grails har en variant av taglibs, som man kan använda tillsammans med Groovy-scriptlets i vyerna, eller så kan man men lite knep och knåp konfigurera det underliggande Spring MVC-ramverket att använda Velocity- eller Freemarker-mallar, exempelvis. Vill man styra webbapplikationen med hjälp av flöden kan man utnyttja Spring Web Flow, återigen genom att använda builders för att formulera tillstånd och transformationer.

## VERKTYG OCH DRIFT

Jag rekommenderar varmt IntelliJ 7 med JetGroovy-pluginen för utveckling av Groovy och Grails. Det är redan nu en kapabel kombination, med korskompilering mellan Groovy och Java och omfattande refaktoreringsstöd. Många av kommandoradverktygets åtgärder kan även köras inbäddat i IDE:n.

Grails kan paketeras och drifas precis som vilken Javaapplikation som helst, genom det förberedda kommandot "grails war". Tillsammans med det faktum att Hibernate används som ORM betyder det att man omedelbart kan dra nytta av de investeringar man gjort i applikationsserver och databas, både i form av pengar och erfarenhet. Dessutom kan man utnyttja mogna och högpresterande skalbarhetslösningar för HTTP-sessionsreplikering och annat.

Prestandan hos en Grailsapplikation kommer av naturliga skäl aldrig att överstiga den hos Java, och på grund av ramverkets filtiga utnyttjande av dynamiska lösningar kommer det i praktiken att ligga en bit efter. Det är inte heller särskilt besvärligt att skriva om identifierade flaskhalsar i Java, om man skulle behöva. Många är förstås nyfikna på hur Grails står sig i förhållande till Ruby on Rails, och det ser lovande ut - tidiga, enkla tester visar ett betydande övertag för Grails. Det här är förstås någonting som varierar med tiden och med uppgiften, så det är väldigt svårt att säga att den ena alltid är snabbare än den andra, men sannolikt behöver man inte välja bort Grails på grund av sämre prestanda.

## SAMLADE INTRYCK

Min avsikt med den här artikeln är att demonstrera vilka möjligheter som existerar när man kombinerar Groovy med etablerade Javabibliotek, och en stor portion inspiration från framför allt Ruby on Rails, med en evolutionär snarare än revolutionär förändring. Det är såklart inte heller en heltäckande genomgång, utan jag har försökt göra nedslag här och där för att man ska få en känsla för hur det hänger ihop.

Alldeles oavsett vilket fotfäste Ruby on Rails kommer att få inom de närmaste åren, så har dess blotta existens mer eller mindre tvingat den 400 kg tunga Java-gorillan att rannsaka sig själv ur ett enkelhets- och produktivitetsspektiv. När rädslan, osäkerheten och tveksamheten klingar av så börjar man fråga sig *hur kan det vara så enkelt för dem, när det är så besvärligt för oss?* En sak som ska sitta i ryggraden hos varje programmerare är att duplicering är dåligt, så det känns intuitivt fel när man måste säga "person" på 6-7 ställen för att komma från webbläsaren till databasen. En annan grundbult inom programmering är att ju mindre kod man behöver för att utföra en uppgift, desto bättre. Mindre kod att läsa, förstå, förändra, underhålla och testa. Och förstås att skriva. På den här punkten har Java fått svår konkurrens av språk som Ruby och Groovy, bland andra.

Det viktigaste är att vi som utvecklare hela tiden strävar efter att öka vår produktivitet, och att vi har roligare och roligare på jobbet under tiden. ■ ■ ■



**Peter Backlund** är konsult inom Javautveckling på Citerus. Missa inte chansen att prata Grails med honom på JFokus den 29-30 januari 2008, där han kommer att föreläsa på temat.

#### Lär dig mer om Grails

- Grails: <http://grails.org> - här finns omfattande och välskriven referensdokumentation, FAQ, och länkar till bloggar, wiki och mailinglistor, bland annat. En del information kan vara en smula inaktuell, eftersom saker kan förändras väldigt snabbt innan 1.0 är släppt.
- Groovy: <http://groovy.codehaus.org>. Jag rekommenderar särskilt en titt på builder-konceptet (<http://groovy.codehaus.org/Builders>) och XML-stödet (<http://groovy.codehaus.org/Processing+XML>).
- JetGroovy för IntelliJ: <http://docs.codehaus.org/display/GRAILS/IDEA+Integration>
- En bok om Grails i PDF-format från InfoQ: <http://www.infoq.com/minibooks/grails> (går även att beställa i pappersform)
- Artikelförfattaren har initierat en portning av Suns Pet Store-applikation till Grails: <http://code.google.com/p/grails-petstore/>