KAPITOLA 3

Architektura aplikací na frameworku Rails

V této kapitole:

- modely,
- pohledy,
- řadiče.

Jedna ze zajímavých vlastností frameworku Rails spočívá v tom, že klade docela závažná omezení ohledně struktury vaší webové aplikace. Díky těmto omezením je ale tvorba aplikací překvapivě jednodušší – mnohem jednodušší. Podívejme se proč.

Modely, pohledy a řadiče

V roce 1979 přišel Trygve Reenskaug s novou architekturou pro vývoj interaktivních aplikací. V jeho návrhu byly aplikace rozdělené na tři typy komponent: modely, pohledy a řadiče.

Model je odpovědný za udržování stavu aplikace. Někdy je tento stav přechodný a trvá jen pár interakcí s uživatelem. Jindy je trvalý a ukládá se mimo aplikaci, často do databáze.

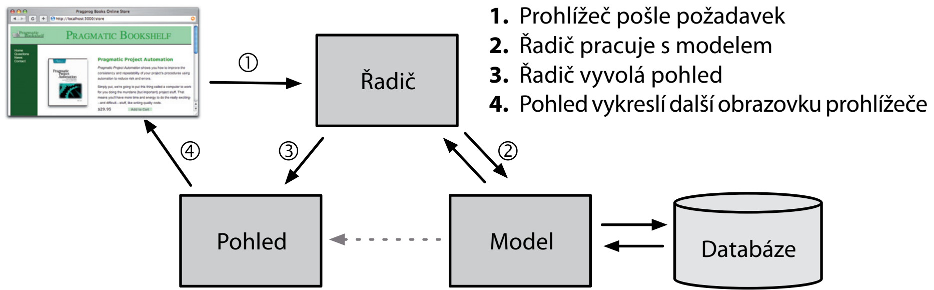
Model představuje více než jen data. Vynucuje si totiž všechna obchodní pravidla, jež se na daná data vztahují. Pokud by se například sleva neměla aplikovat na objednávky o celkové hodně menší než 500 Kč, model si toto omezení vynutí. Je to logické, neboť umístěním implementace těchto obchodních pravidel do modelu zajistíme, že nic v aplikaci nemůže způsobit neplatnost dat. Model funguje jednak jako strážný a jednak jako datové úložiště.

Pohled (view) je odpovědný za generování uživatelského rozhraní, které je obvykle založené na datech z modelu. Kupříkladu internetový obchod bude mít seznam produktů, které se zobrazují na obrazovce katalogu. Tento seznam bude přístupný skrze model, je to ale právě pohled, kdo k němu přistupuje a formátuje jej pro koncového uživatele. Pohled sice může prezentovat uživateli nejrůznější způsoby zadávání dat, sám ale příchozí data nikdy nezpracovává. Úkol pohledu je splněný, jakmile se data zobrazí. Může existovat řada pohledů, jež přistupují ke stejným datům modelu, často za jiným účelem. V internetovém obchodu může být pohled, který zobrazí informace o produktu na stránce katalogu, a jiná skupina pohledů používaná správci pro přidávání a úpravy produktů.

Řadiče (controller) řídí aplikaci. Přijímají události z vnějšího světa (obvykle vstup uživatele), pracují s modelem a zobrazují příslušný pohled uživateli.

Tento triumvirát (model, pohled a řadič) dohromady tvoří architekturu označovanou jako MVC (Model-View-Controller). Jejich vzájemné vztahy znázorňuje obrázek 3.1.

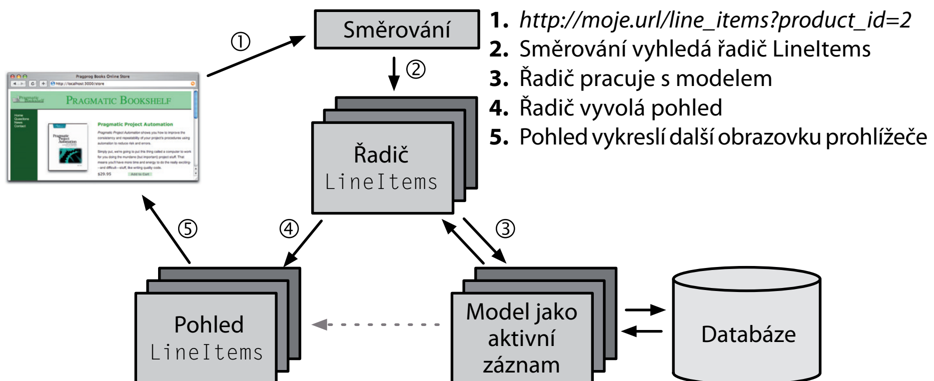
Architektura MVC byla původně navržená pro tradiční aplikace s grafickým uživatelským rozhraním. Vývojáři totiž zjistili, že rozdělení odpovědností vede k méně vazbám, díky čemuž se kód snadněji píše a udržuje. Každá koncepce nebo akce byla vyjádřená jen na jediném, dobře známém místě. Používání architektury MVC bylo jako budování mrakodrapu s již připravenými nosníky – s již připravenou strukturou bylo mnohem snazší zavěsit zbývající části. Během vývoje aplikace budeme zhusta využívat schopnosti frameworku Rails generovat základní *konstrukci* pro naši aplikaci.



Obrázek 3.1: Architektura MVC

Ruby on Rails je rovněž framework s architekturou MVC. Framework Rails vnucuje vaší aplikaci určitou strukturu – vyvíjíte modely, pohledy a řadiče jako samostatné kousky funkčnosti a framework je při provádění vašeho programu všechny splete dohromady. Jedním z potěšení při práci s frameworkem Rails je to, že tento proces splétání je založený na použití inteligentních výchozích nastavení, takže k tomu, aby vše fungovalo, většinou vůbec nemusíte psát žádná externí konfigurační metadata. Jedná se o příklad filozofie frameworku Rails upřednostňovat konvenci před konfigurací.

V aplikaci na frameworku Rails je příchozí požadavek nejdříve odeslán směrovači, který zjistí, kam v aplikaci se má odeslat a jak se má samotný požadavek analyzovat. Tato fáze nakonec identifikuje konkrétní metodu (označovanou v žargonu frameworku Rails jako *akce*) umístěnou někde v kódu řadiče. Akce se může podívat na data v požadavku, může pracovat s modelem a může způsobit vyvolání dalších akcí. Akce nakonec připraví informace pro pohled, který uživateli něco vykreslí.



Obrázek 3.2: Framework Rails a architektura MVC

Obsluha příchozího požadavku frameworkem Rails je znázorněná na obrázku 3.2. V tomto příkladu aplikace nejdříve zobrazila stránku s katalogem produktů a uživatel klepnul na tlačítko **Přidat do košíku** vedle jednoho z produktů. Toto tlačítko odeslalo požadavek na adresu *http://localhost:3000/line_items?product_id=2*, kde *line_items* je prostředek v naší aplikaci a *2* je interní identifikátor pro zvolený objekt.

Směrovací komponenta obdrží příchozí požadavek a okamžitě jej rozebere. Požadavek obsahuje cestu (*/line_items?product_id=2*) a metodu (toto tlačítko provedlo operaci POST, mezi další běžně používané metody patří GET, PUT a DELETE). V tomto jednoduchém případu vezme framework Rails první část cesty (*line_items*) jako název řadiče a *product_id* jako identifikátor produktu. Dle konvence jsou metody POST spojené s akcemi `create`. Výsledkem celé této analýzy je to, že směrovač ví, že má vyvolat metodu `create` ve třídě řadiče s názvem `LineItemsController` (konvencím pro pojmenování se budeme věnovat na straně 302).

Metoda `create` obsluhuje požadavky uživatele. V tomto případě vyhledá nákupní košík aktuálního uživatele (což je objekt spravovaný modelem). Dále si od modelu vyžádá informace pro produkt 2. Poté řekne nákupnímu košíku, aby k sobě přidal tento produkt. (Vidíte, jak se používá model pro sledování všech obchodních dat? Řadič mu řekne, co má dělat, a model ví, jak to má udělat.)

Košík nyní obsahuje nový produkt, můžeme jej tedy ukázat uživateli. Řadič vyvolá kód pohledu, ale ještě předtím uspořádá vše tak, aby pohled měl přístup k objektu košíku z modelu. Ve frameworku Rails je toto vyvolání často implicitní. A opět: konvence pomáhají propojit určitý pohled s danou akcí.

A to je vše, co se týká webových aplikací s architekturou MVC. Díky dodržování jisté sady konvencí a náležitému rozdělení funkčnosti zjistíte, že se vám bude s kódem snadněji pracovat a vaše aplikace se budou lehčeji rozšiřovat a udržovat. Vypadá to tedy, že se to jistě vyplatí.

Je-li architektura MVC jen otázkou určitého rozdělení kódu, tak se možná divíte, k čemuž je framework, jako je Ruby on Rails vlastně, nutný. Odpověď je jednoduchá: framework Rails se za vás stará o veškeré hospodaření na nízké úrovni (všechny ty komplikované detaily, jejichž ošetření by vám trvalo tak dlouho) a nechává vás, abyste se soustředili na ústřední funkčnost vaší aplikace. Podívejme se jak.

Podpora modelu ve frameworku Rails

Obecně lze říci, že chceme, aby naše webové aplikace uchovávaly své informace v relační databázi. Systémy pro vkládání objednávek budou uchovávat objednávky, položky a údaje o zákaznících v tabulkách databáze. Dokonce i aplikace, které běžně používají nestrukturovaný text, jako jsou blogy a zpravodajské weby, často používají pro ukládání dat právě databáze.

I když z jazyka SQL (Structured Query Language – strukturovaný dotazovací jazyk pro dotazování a aktualizaci relačních databází), který se používá pro přístup k relačním databázím, to nemusí být ihned patrné, relační databáze jsou ve skutečnosti navržené kolem matematické teorie množin. Z koncepčního hlediska je to sice dobré, ale zkombinování relačních databází s objektově orientovanými programovacími jazyky je kvůli tomu obtížnější. Objekty jsou veskrze o datech a operacích, kdežto databáze jsou veskrze o množinách hodnot. Operace, které lze snadno vyjádřit v relačních termínech, se v objektově orientovaném systému někdy obtížně programují. A platí to i naopak.

Časem přišli lidé na způsoby harmonizace relačních a objektově orientovaných pohledů na podniková data. Nyní si ukážeme způsob, který si pro mapování relačních dat na objekty zvolil framework Rails.

Objektově-relační mapování

Knihovny ORM mapují databázové tabulky na třídy. Má-li databáze tabulku s názvem `orders`, náš program bude mít třídu s názvem `Order`. Řádky v tabulce odpovídají objektům příslušné třídy – určitá objednávka je reprezentována jako objekt typu `Order`. V takovém objektu se hodnoty jednotlivých sloupců získávají a nastavují pomocí atributů. Náš objekt typu `Order` má metody pro získání a nastavení částky, DPH a tak dále.

Kromě toho mají třídy frameworku Rails, které obalují datové tabulky, skupinu metod na úrovni třídy, jež provádějí operace na úrovni tabulky. Můžeme tak například chtít vyhledat objednávku s určitým identifikačním číslem. Hledání je implementováno jako metoda, jež vrací odpovídající objekty typu `Order`. V kódu jazyka Ruby to může vypadat takto:

metoda třídy ➤ 75

```
order = Order.find(1)
puts "Customer #{order.customer_id}, amount=$#{order.amount}"
```

puts ➤ 68

Tyto metody na úrovni třídy někdy vracejí kolekce objektů:

```
Order.where(:name => 'dave').each do |order|
  puts order.amount
end
```

iterace ➤ 72

A nakonec objekty odpovídající jednotlivým řádkům v tabulce mají metody, jež operují na příslušném řádku. Pravděpodobně nejvíce používanou je metoda `save` provádějící operaci, jež uloží řádek do tabulky.

```
Order.where(:name => 'dave').each do |order|
  order.pay_type = "Purchase order"
  order.save
end
```

Vrstva ORM tedy mapuje tabulky na třídy, řádky na objekty a sloupce na atributy těchto objektů. K provádění operací na úrovni tabulky se používají metody třídy a operace na jednotlivých řádcích se provádějí pomocí metod instance.

V typické knihovně ORM se používají konfigurační data pro specifikaci mapování mezi entitami v databázi a entitami v programu. Programátoři používající takovéto nástroje ORM musejí často vytvářet a udržovat hromadu konfiguračních souborů XML.

Vrstva Active record

Active record (aktivní záznam) je vrstva ORM, kterou poskytuje framework Rails. Přísně se řídí standardním modelem ORM: tabulky se mapují na třídy, řádky na objekty a sloupce na atributy objektů. Od většiny ostatním knihoven ORM se liší ve způsobu konfigurace. Opírá se o konvence a začíná s rozumnými výchozími hodnotami, a proto minimalizuje množství konfigurace prováděné vývojářem.

Můžeme si to ukázat třeba na programu, který používá vrstvu Active Record pro obalení tabulky `orders`:

```
require 'active_record'

class Order < ActiveRecord::Base
end

order = Order.find(1)
order.pay_type = "Purchase order"
order.save
```

Tento kód používá novou třídu `Order` pro načtení objednávky s identifikačním číslem 1 a úpravu sloupce `pay_type`. (Prozatím jsme vynechali kód, který vytváří připojení k databázi.) Vrstva Active Record ulehčuje práci s podkladovou databází, díky čemuž se můžeme soustředit na obchodní logiku.

Vrstva Active Record toho ale umí daleko více. Jak uvidíte při vývoji naší aplikace s nákupním košíkem, vrstva Active Record se hladce integruje se zbytkem frameworku Rails. Pokud webový formulář odešle aplikační data vztahující se k obchodnímu objektu, může jej vrstva Active Record extrahovat do našeho modelu. Active Record podporuje sofistikovanou validaci dat modelu, a pokud formulářová data validací neprojdou, dokážou pohledy frameworku Rails extrahovat a naformátovat chyby.

Vrstva Active Record je stabilní modelový základ pro architekturu MVC frameworku Rails.

Komponenta Action Pack: Pohled a řadič

Když se nad tím zamyslíte, tak části architektury MVC představující pohled a řadič jsou docela důvěrné. Řadič dodává data pohledu a přijímá událostí ze stránek generovaných pohledy. Vzhledem k této interakci je podpora pro pohledy a řadiče ve frameworku Rails sdružená do jediné komponenty s názvem *Action Pack* (akční balíček).

Nenechte se zmást představou, že se kód pohledu aplikace a kód řadiče zpřehází jen proto, že Action Pack je jediná komponenta. Právě naopak. Framework Rails nabízí separaci, která je nezbytná pro psaní webových aplikací s jasně vyznačeným kódem pro řídicí a prezentační logiku.

Podpora pro pohledy

Ve frameworku Rails je pohled odpovědný za vytváření buď celé, nebo části odpovědi, která se má zobrazit v prohlížeči, zpracovat aplikací nebo odeslat na e-mail. V nejjednodušším případě je pohled kouskem kódu jazyka HTML, jenž zobrazuje nějaký fixní text. Častěji se ale používá dynamický obsah vytvořený metodou akce v řadiči.

Ve frameworku Rails generují dynamický obsah šablony, které mohou být trojího typu. Nejčastěji používané šablonové schéma označované jako Embedded Ruby (ERb) vkládá úryvky kódu jazyka Ruby do dokumentu pohledu, v mnoha ohledech podobně jako v jiných webových frameworcích, jako je PHP nebo JSP. Tento přístup je sice velice flexibilní, někteří však namítají, že narušuje ducha architektury MVC. Vkládáním kódu do pohledu totiž riskujeme, že přidáme logiku, která by měla být v modelu nebo řadiči. Stejně jako u všeho platí, že zatímco rozumné a zdrženlivé používání je blahodárné, nadužívání se může stát problémem. Udržování čistého oddělení zájmů je součástí práce vývojáře. (Na šablony HTML se podíváme na straně 453, Generování HTML pomocí systému ERb.)

Pro konstrukci dokumentů XML pomocí kódu jazyka Ruby lze použít XML Builder – struktura vygenerovaného kódu jazyka XML pak bude automaticky sledovat strukturu kódu. Šablonami *xml.builder* se budeme zabývat od strany 452.

Framework Rails dále nabízí pohledy RJS, které umožňují vytvářet na straně serveru fragmenty kódu jazyka JavaScript, které se pak provedou v prohlížeči. To je skvělé pro vytváření dynamických rozhraní postavených na technologii Ajax (více na toto téma od strany 168).

A řadič!

Řadič frameworku Rails je logickým center vaší aplikace. Koordinuje interakci mezi uživatelem, pohledy a modelem. Nicméně framework Rails se o většinu této interakce stará za oponou. V kódu, který píšete, se tak můžete soustředit na funkčnost na úrovni aplikace. Díky tomu lze kód řadiče ve frameworku Rails pozoruhodně snadno vyvíjet a udržovat.

Řadič je rovněž domovem pro řadu důležitých pomocných služeb:

- Je odpovědný za směrování externích požadavků do interních akcí. Velmi dobře se stará o lidsky čitelné adresy URL.
- Řídí práci s mezipamětí, což může dát aplikaci řádově lepší výkon.
- Spravuje pomocné moduly, jež rozšiřují možnosti šablon pohledů, aniž by se rozbujel jejich kód.
- Spravuje relace, čímž uživatelům poskytuje dojem plynulé interakce s aplikací.

V druhé kapitole jsme řadič již viděli a upravovali a při vývoji ukázkové aplikace uvidíme a upravíme také celou řadou řadičů. Prvním z nich bude řadič produktů na straně 122 (Iterace C1: Vytvoření výpisu z katalogu).

Framework Rails toho nabízí ještě daleko více. Než ale budeme pokračovat dál, dáme si malou rekapitulaci (a pro některé stručný úvod do) jazyka Ruby.