Four Days on Rails2.3

Originally Compiled by John McCreesh

日本語版(Ver.1.0:2009/7/16) 板垣 正敏

目次

Rails 2.3 対応版にあたって

John MaCreesh 氏によって"Four Days on Rails"(2.0)が書かれたのは、Rails　0.12.1 の時代であり、その後私が日本語訳を出すにあたって、Rails 1.1.2、Rails 1.2.x に合わせて一部修正を行ってきた。しかしながら、Rails 2.0 が 2007 年の 12 月にリリースされ、Rails 1.x にあった scaffold generator が姿を消し、Rails 1.2.x で scaffold-resource generator が新しい scaffold generator となったこともあり、大幅な改訂の必要が生じてきた。

そこで 2008 年5月に、Rails 2.0 で初めて Rails に触れる初心者用のチュートリアルとして特に前半の部分を中心に大幅な書き換えを行った。その後、2009年7月に Rails 2.3.2 を対象に修正と見直しを行った。

したがって、この書き変えによる誤りなどの問題については、原著作者の John McCreesh 氏には責任はなく、私が責任を負うことになる。

板垣正敏

はじめに

これまで Rails について何度も大げさな主張がされてきた。たとえば、OnLAMP.com[1]の記事は「Rails を使うと一般的な Java フレームワークに比較して最低でも 10 倍早く Web アプリケーションが開発できる」と公言している。この記事ではさらに PC に Ruby と Rails をインストールし、実質上 1 行もコードを書くことなく、動く scaffold アプリケーションを生成する方法を示している。

この記事は印象が強烈だが、「本当の」Web アプリ開発者は、これが手品であることを知っている。「本当の」Web アプリケーションはこんなに単純ではない。Rails の舞台裏では何が行われているのだろう。さらに進んで「本当の」アプリケーションを開発するのはどれぐらい大変なのだろうか。

この辺から問題は少しややこしくなる。Rails には充実したオンラインドキュメントが完備している。実際、リファレンスマニュアルの形で 30,000 語を超えるオンラインドキュメントは、初心者には多すぎるくらいだ。欠けているのは、Rails で開発を始め、進めていくために必要なページを示してくれる鉄道路線図である。

このドキュメントは、このギャップを埋めるために用意したものだ。ここでは、PC に Ruby on Rails を設定し、動くところまで実行済みであることを想定している。(まだインストールしていないのであれば、Curt の記事の通りにすること)ここまで終わっていれば、1 日目の終わりまで進んだことになる。

「Rails の 2 日目」は、手品の舞台裏を知ることから始まる。"scaffold"コードを順に見ていることになる。新しい機能は太字で表示され、解説があり、その後に、Rails あるいは Ruby のドキュメントへの参照を示す。それを読むことで理解を深めることができる。

「Rails の 3 日目」は、"scaffold"コードを足場に、「本当の」アプリケーションに見えるようなものを構築する。一日をかけて Rails を使い、あなた自身の工具箱を作り上げていくことになる。何よりも、オンラインドキュメントの参照が楽にできるようになり、自分で答えを見つけることができるようになるだろう。

「Rails の 4 日目」には、もうひとつテーブルを追加し、参照一貫性を維持するという複雑な仕事をこなすことになる。最後には、使えるアプリケーションと、自分で開発を始めるのに十分な道具、そして、更なる情報をどこで得ればよいかという知識が身についているはずだ。

本当に 10 倍速いかって?4日間 Rails を使った後で、ご自分で判断してほしい。

ドキュメンテーション: このドキュメントにはハイライトされた下記への参照が含まれる:

ドキュメンテーション —Rails のドキュメンテーション http: //api.rubyonrails.com (このドキュメントは、gems でのインストールの際に PC の次の場所にもインストールされる。`C:\Program Files\ruby\lib\ruby\gems\n.n\doc\actionpack-n.n.n\rdoc\index.html`)

1 *Rolling with Ruby on Rails,* Curt Hibbs 20-Jan2005 http://www.onlamp.com/pub/a/onlamp/2005/01/20/rails.html

Ruby のドキュメンテーション—“Programming Ruby — The Pragmatic Programmers Guide”は次の場所でオンラインで利用可能であり、あるいはダウンロードできる。http://www.ruby-doc.org/docs/ruby-doc-bundle/ProgrammingRuby/index.html

謝辞: IRC チャネル[2]とメーリングリスト[3]上の親切な方々に感謝する。オンラインアーカイブは、私が Rails や Ruby のラーニングカーブを上っていく際に、それらの人々にいかに助けられたかの証である。

バージョン: 2.3 は Rails の 0.12.1 を使用している。最新版と ToDo のプログラムコードをダウンロードするには http://rails.homelinux.org を参照のこと。ドキュメントは OpenOffice.org の Writer で作成し、pdf ファイルを出力した。(訳注:日本語版の改訂にあたり、日本語版独自の Rails 1.2.5 での作動確認と加筆修正を行った)

翻訳について:翻訳および Rails2.x 対応版改訂については、板垣 正敏 masatoshi@rails.to が全ての責任を負っている。翻訳版についても、*Creative Commons Attribution-NonCommercial-ShareAlike License* が適用される。誤訳等の指摘や感想などについては、上記のメールアドレスに送信いただきたい。

2 irc://irc.freenode.org/rubyonrails
3 http://lists.rubyonrails.org/mailman/listinfo/rails

Rails 1日目

Ruby 及び Rails のインストールについて

ここでは、始めて Ruby および Ruby on Rails に触れる方向けに、Ruby および Rails のインストール方法を簡単に紹介する。

Windows の場合

Windows で使える Ruby は色々あるが、もっとも簡単でよく使われている One-Click Ruby Installer を紹介しよう。One-Click Ruby Installer は Ruby のオープンソースプロジェクトがたくさん登録されている RubyForge にある。インストーラーを実行するだけで、Ruby の標準ライブラリと、Ruby パッケージの標準的な配布ツールである RubyGems、さらにはテキストエディタの SciTE が含まれており、PATH 環境変数も自動的に設定されるため、すぐにプログラミングができる。

http://rubyforge.org/projects/rubyinstaller/

Mac　OS X の場合

Leopard の場合には、Ruby および Ruby on Rails が標準搭載されているため、特にインストールは必要ない。Mac OS X 10.3/10.4 の場合には、すでに開発が終了しているが Locomotive を使うことができる。

http://locomotive.raaum.org/

Linux（Ubuntu）の場合

Ubuntu Linux の場合、Symaptic パッケージマネージャを使うことで、Ruby をインストールできる。およそ下記のようなパッケージをインストールすればよいだろう。

ruby1.8 ruby1.8-dev irb1.8 rubygems libopenssl-ruby1.8

この方法では rubygems の利用に root 権限が必要となる。（例えば、Ubuntu Desktop 8.04 の場合、GEM_PATH は/var/lib/gems/1.8 となる。このディレクトリの所有者は root であり、アクセス権は 0755 である）

rubygems 1.3.0 からは、こうした場合、ユーザーのホームディレクトリに.gem ディレクトリを作成し、この下に gem をインストールする機能が追加されている。ただし、gem にコマンドスクリプトが含まれる場合、下記のように PATH 環境変数にこのパスを追加する必要がある。

```
export PATH=/home/masatoshi/.gem/ruby/1.8/bin:$PATH
```

Rubygems のアップグレード

Ruby on Rails 2.3 では、rubygems 1.3 以降が必要である。このため、これ以前のバージョンの場合、下記のコマンドでアップグレードする必要がある。

```
gem update --system
```

SQLite3 のインストール

Windows の場合

下記のページから、Windows 用の DLL(sqlitedll-3.x.x.zip)をダウンロードし、解凍して得られた DLL ファイルを実行パスが通っているディレクトリ、例えば、One-Click Ruby Installer でインストールされた ruby の実行ディレクトリ（C:\ruby\bin)にコピーする。

実際に作成されたデータベースの確認には実行ファイル（sqlite-3.x.x.zip)をダウンロードし、同様に解凍した EXE ファイルを実行パスの通っているディレクトリにコピーする。

なお、バージョン 3.3 では Rails との互換性に問題があるため、3.4 以降のバージョンを使用すること。

http://www.sqlite.org/download.html

Mac OS X の場合

Leopard の場合には SQLite3 のバージョン 3.4 が含まれているため、とくにインストールの必要はないだろう。必要であれば上記の SQLite の公式サイトからダウンロード出来る。

Linux(Ubuntu）の場合

Ubuntu Linux の場合には、Synaptic でバージョン 3.4 をインストール出来る。

Ruby on Rails のインストール

rubygems を使って Ruby on Rails をインストールする。

```
gem install rails
```

“To Do リスト”アプリケーション

このドキュメントでは、簡単な To Do リストアプリケーションの構築をなぞってゆく。このアプリケーションは、PDA にあるように、アイテムのリストが、カテゴリーによりグループ化され、オプションのノートがつけられる。（およその外観は30ページの図5「To Do リスト」の画面を参照のこと）

Rails のスクリプトを実行する

これは私のPCでの例である。私の Web 関連ファイルは、`c:\www\webroot` にある。私は次のようにコマンドを発行して、これを W:ドライブとしてマッピングした。

```
C:\> subst w: c:\www\webroot
C:\>W:
W:\> rails ToDo
W:\> cd ToDo
W:\ToDo>
```

`rails ToDo` を実行することで新しく ToDo ディレクトリが作成され、その下には一連のファイルとサブディレクトリが生成される。中でも重要なのは下記のディレクトリである。

（訳注：Rails 2.0.2 以降では、これまでの MySQL に代わり、SQLite3 が既定のデータベースとなった。Rails アプリケーションの作成時にデータベースを指定しなければ、SQLite3 用に `config/database.yml` が生成されるが、`rails ToDo -d mysql` のようにデータベースを指定して実行することができる。）

app
アプリケーションの核になる部分を含む。その中は、model，view，controller，および helper の各サブディレクトリに分かれている。
config
アプリケーションで使用するデータベースの詳細を定義した database.yml ファイルを含む。
db/migrate
データベース定義を管理する migration スクリプトが格納される。
log
アプリケーション特有のログを含む。注：development.log は、エラー追跡に便利な Rails でのアクションを逐次記録したトレースを含むが、定期的に削除する必要がある。
public
Apache 用のディレクトリで、images，javascripts，および stylesheets の各サブディレクトリを含む。

Rails が稼動していることを確認する

開発環境に於いては、Rails に標準で組み込まれた Web サーバーアプリケーションである、Webrick を使うことができる。アプリケーションのディレクトリで下記のようなコマンドを起動することで、開発用のサーバを起動し、動作を確認する事ができる。

```
W:\Todo>ruby script/server
```

上記のようにオプションなしで Webrick を起動した場合、ポート番号 3000 が使用される。（ポート番号 3000 がすでに使用されている場合、次の空き番号が使用される）したがって、http://localhost:3000/をブラウザで参照する事ができる。(Congratulations, you've put Ruby on Rails!というページが参照できるはずだ）

3000 番以外のポートを使用したい場合には、-p ポート番号というオプションスイッチで指定できる。

データベースを設定する

Rails 2.0.2 で新しく既定の開発用データベースとなった SQLite3 はサーバー型ではなくライブラリ型のデータベースであり、必要に応じてデータベースファイルが作成されるため、あらかじめデータベースを用意する必要がない。これが既定のデータベースに選ばれたもっとも大きな理由だろう。

このデータベースへの接続は config\database.yml ファイルに指定する。

config\database.yml （抜粋）

```
# SQLite version 3.x
#   gem install sqlite3-ruby (not necessary on OS X Leopard)
development:
  adapter: sqlite3
  database: db/development.sqlite3
  pool: 5
  timeout: 5000

# Warning: The database defined as "test" will be erased and
# re-generated from your development database when you run "rake".
# Do not set this db to the same as development or production.
test:
  adapter: sqlite3
  database: db/test.sqlite3
  pool: 5
  timeout: 5000

production:
  adapter: sqlite3
  database: db/production.sqlite3
  pool: 5
  timeout: 5000
```

Scaffold によるアプリケーションの生成

Rails 2.0 からは、ソースコードを一行も編集することなく、基本的な機能を持つアプリケーションを生成できる様になった。

```
W:\ToDo>ruby script/generate scaffold Category category:string
      exists  app/models/
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/categories
      exists  app/views/layouts/
      exists  test/functional/
      exists  test/unit/
      create  test/unit/helpers/
      exists  public/stylesheets/
      create  app/views/categories/index.html.erb
      create  app/views/categories/show.html.erb
      create  app/views/categories/new.html.erb
      create  app/views/categories/edit.html.erb
      create  app/views/layouts/categories.html.erb
      create  public/stylesheets/scaffold.css
      create  app/controllers/categories_controller.rb
      create  test/functional/categories_controller_test.rb
      create  app/helpers/categories_helper.rb
      create  test/unit/helpers/categories_helper_test.rb
       route  map.resources :categories
  dependency  model
      exists    app/models/
      exists    test/unit/
      exists    test/fixtures/
      create    app/models/category.rb
      create    test/unit/category_test.rb
      create    test/fixtures/categories.yml
      create    db/migrate
      create    db/migrate/20090711033412_create_categories.rb
W:\ToDo>
```

生成されたマイグレーションファイルの確認

上記の scaffold generator によって、データベーステーブルを生成するためのマイグレーションファイルが生成される。

db\migrate\20090711033412_create_categories.rb

```
class CreateCategories < ActiveRecord::Migration
  def self.up
    create_table :categories do |t|
      t.string :category

      t.timestamps
    end
  end

  def self.down
    drop_table :categories
  end
end
```

参考までに、Rails におけるデータベースの命名規則などに関する規約とヒントを示す。

テーブルおよび列の命名に関するヒントとポイント:

- 列名に含まれるアンダースコアは Rails が人に読みやすい名前を生成する際には取り除かれる。
- 列名に大文字小文字を混ぜて使用するときは注意すること。Rails の一部は大文字小文字を区別する。
- 全てのテーブルには、`'id'`という名前の主キーがあること。
- 他のテーブルへのリンクすなわち外部キーは同様に`'_id'`という命名規則に従う。
- Rails は `created_at / created_on` および `updated_at / updated_on` という名前の列の更新を自動的に行う。scaffold ジェネレータは自動的にこれらの列を生成するため、`t.timestamps` というメソッド呼び出しをマイグレーションに追加する。

Documentation: ActiveRecord::Timestamp

- 有用なヒント:マルチユーザーシステムを構築しているのであれば(いまは関係ないが)、もしテーブルに `lock_version (integer default 0)` という列があると、Rails は自動的に楽観的ロックを実装してくれる。しなければならないのは、更新フォームに `rock_version` を隠しフィールドとして含めることだけである。

Documentation: ActiveRecord::Locking

マイグレーションを実行しテーブルを生成する

rake によりマイグレーションタスクを実行し、テーブルを生成する。

```
W:\ToDo>rake db:migrate
(in W:/ToDo)
== CreateCategories: migrating ================================================
-- create_table(:categories)
   -> 0.1700s
== CreateCategories: migrated (0.1700s) =======================================


W:\ToDo>
```

ブラウザで、http://todo:3000/categories を参照すれば、Rails がいかに利口かびっくりするだろう。:-)

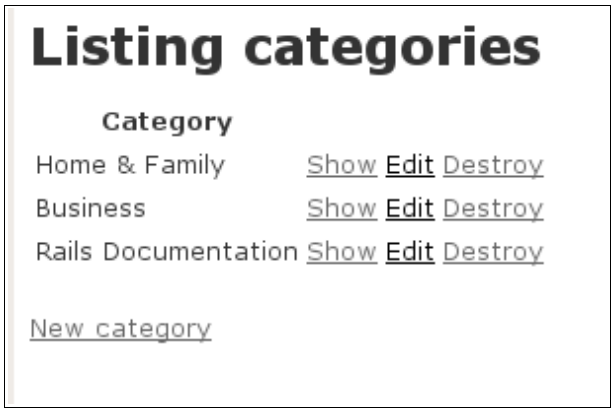


図1 Scaffold 'List' 画面

Validation

Rails がそれほど利口でないことを理解するため、Category に何も入力せずに一度追加してみよう。何の警告もなしに追加されてしまう。これは、モデルに検証機能を追加することで直すことができる。

モデルを拡張する

モデルには、データに関係するルールが保存される。その中にはデータの検証や参照整合性が含まれる。つまり、ルールを一度書くだけで、Rails がデータのアクセスされる場所では必ずそのルールを自動的に適用してくれるということだ。

データ検証ルールを作成する

Rails はたくさんのエラー処理を（ほとんど）タダでやってくれる。これをデモするために、空の category モデルに、いくつか検証ルールを加えてみよう:

```
app\models\category.rb
class Category < ActiveRecord::Base
  validates_length_of :category, :within => 1..20
  validates_uniqueness_of :category, :message => "already exists"
end
```

これにより次のチェックが自動的に行われる:

- `validates_length_of`: 列が空ではなく、また長すぎないこと
- `validates_uniqueness_of`: 重複値が捕らえられる。私は、Rails の既定エラーメッセージ`'xxx has already been taken'`が嫌いなので、自前のものを使用している。これは、Rails の一般的な機能であり、まず既定値を試してみて、気に入らなければ上書きをすればよい。

  訳注：Rails のこの重複検出機能は、残念ながら100%信頼できるものではない。テーブルやインデックスのロックを伴うものではないため、データ登録や更新が輻輳する条件下では、重複検出のための SELECT とデータ登録のための INSERT の間に割り込まれる可能性が生じるためである。このリスクを回避するには、データベース側でユニークなインデックスを設定するなどする必要がある。

Documentation: ActiveRecord::Validation::ClassMethods

この機能を試すために、重複するレコードを再び入力してみよう。今回は、エラーを処理してくれる。このスタイルは気に入らないだろう—けしてユーザーインターフェイスを考慮した繊細なものではない。だが、タダで手に入るのだからこんなものだろう。

図 2: データエラーを捕捉する

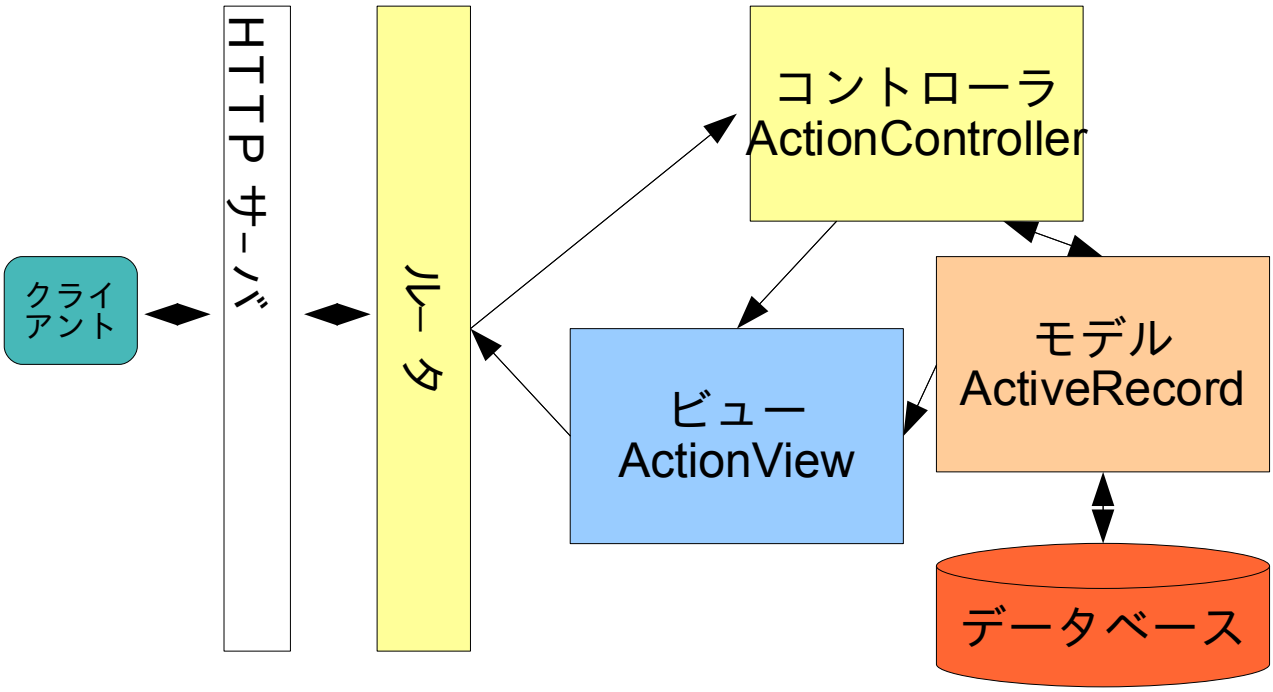
Rails 2 日目

ここから先に進むためには、舞台裏で何が行われているかを知る必要がある。2 日目を通して、Rails が生成する Scaffold コードを体系的に検証し、そのコードの意味を解き明かしてゆきたい。

このスクリプトは、完全なアプリケーションを構成するのに必要な、たくさんのファイルを生成する。ファイルには、コントローラ、ビュー、レイアウトそして、スタイルシートまでもが含まれる。

Rails の仕組み

Rails はいわゆる MVC に基づく Web アプリケーションフレームワークである。クライアントから HTTP サーバを通じて送られてきたリクエストはルータと言われるプログラムによって解析され、コントローラのメソッドが呼び出される。コントローラは、モデルである ActiveRecord を通じてデータベースの操作を行い、モデルのデータを反映したビューを生成してクライアントに返す。



Rails 2.0 と REST

Ruby on Rails 2.0 以降では、REST(Representational State Transfer)の考え方が取り入れられ、Rails 1.x とは Scaffold で生成されるコードが大きく変わっている。

REST は Roy T. Fielding 氏によって考え出された概念であり、詳しくは氏の論文や他の書籍を参照していただきたいが、「プログラマブル Web」つまり、コンピュータプログラムが Web 上のデータを扱うことを配慮し、HTTP の世界に一定の「制約(秩序)」を提案したものである。主要な概念として次のようなものが挙げられる。

・通信が状態に依存しないこと(ステートレスであること)

・Web 上の資源は URI によってアドレス指定できること、またキャッシュ可能であるかどうかを伝達可能であること

・HTTP の「動詞」である GET/PUT/POST/DELETE を適切に扱えること

上記の特性は、SOAP を使用する Web サービスとは一線を画すものであり、Ruby on Rails の開発チームは、SOAP/Web サービスに対して、REST を支持することを明確にするため、従来 Rails に標準搭載されていた ActiveWebservice を標準からはずしプラグインとした。

Rails 2.0 の REST によるリソース（データ）のハンドリング

データベース操作	REST に基づく HTTP の動詞	Rails 2.0 の操作	Rails のアクション（対応するコントローラのメソッド）
Create（データの作成）	PUT	GET /コントローラ/new で入力フォームを取得する。 POST /コントローラ にフォームデータを送信し、データを作成する。	new create
Read（一覧取得）	GET	GET /コントローラ	index
Read（1 件を読み出し）	GET	GET /コントローラ/id 番号	show
Update（更新）	PUT	GET /コントローラ/id 番号/edit で編集フォームを取得。 POST /コントローラ/id 番号 ただし、Web ブラウザがクライアントの場合、method=put というクエリパラメータで PUT をシミュレートする。	edit update
Delete（削除）	DELETE	PUT /コントローラ/id 番号 ただし、Web ブラウザがクライアントの場合、method=delete というクエリパラメータで DELETE をシミュレートする。	destroy

生成された Scaffold コード

コントローラ

コントローラの背後のコードを見てみよう。コントローラにはアプリケーションのプログラミングロジックが格納される。コントローラはビューを通じてユーザーとやり取りを行い、モデルを通してデータベースとやり取りする。コントローラのコードを読んで、アプリケーションがどのように成り立っているかを理解できるようになってほしい。

Generate scaffold スクリプトによって生成されるコントローラを下記に示す。

\app\controllers\categories_controller.rb

```
class CategoriesController < ApplicationController
  # GET /categories
  # GET /categories.xml
  def index
    @categories = Category.all

    respond_to do |format|
      format.html # index.html.erb
      format.xml  { render :xml => @categories }
    end
  end

  # GET /categories/1
  # GET /categories/1.xml
  def show
    @category = Category.find(params[:id])

    respond_to do |format|
      format.html # show.html.erb
      format.xml  { render :xml => @category }
    end
  end

  # GET /categories/new
  # GET /categories/new.xml
```

```
  def new
    @category = Category.new

    respond_to do |format|
      format.html # new.html.erb
      format.xml  { render :xml => @category }
    end
  end

  # GET /categories/1/edit
  def edit
    @category = Category.find(params[:id])
  end

  # POST /categories
  # POST /categories.xml
  def create
    @category = Category.new(params[:category])

    respond_to do |format|
      if @category.save
        flash[:notice] = 'Category was successfully created.'
        format.html { redirect_to(@category) }
        format.xml  { render :xml => @category, :status => :created, :location =>
@category }
      else
        format.html { render :action => "new" }
        format.xml  { render :xml => @category.errors, :status => :unprocessable_entity }
      end
    end
  end

  # PUT /categories/1
  # PUT /categories/1.xml
  def update
    @category = Category.find(params[:id])

    respond_to do |format|
      if @category.update_attributes(params[:category])
        flash[:notice] = 'Category was successfully updated.'
        format.html { redirect_to(@category) }
        format.xml  { head :ok }
      else
        format.html { render :action => "edit" }
        format.xml  { render :xml => @category.errors, :status => :unprocessable_entity }
      end
    end
  end

  # DELETE /categories/1
  # DELETE /categories/1.xml
  def destroy
    @category = Category.find(params[:id])
    @category.destroy

    respond_to do |format|
      format.html { redirect_to(categories_url) }
      format.xml  { head :ok }
    end
  end
end
```

respond_to

```
respond_to do |format|
  format.html { ... }
  format.xml { ... }
  ...
end
```

respond_to は Rails 1.2 から導入された新しいメソッドで、一つのコントローラで、ブラウザ向けの Web アプリケーションと、REST に基づく Web サービスを提供することを目的としている。

HTTP リクエストにおけるリソースの表現指定にしたがって、処理をブロックとして渡し、レスポンスを生成することができる。フォーマットの既定値は html であるが、ブロックを省略した場合、メソッド名と同名のテンプレートがレンダリングされる。

試しに、http://localhost:3000/categories.xml のようなリクエストを送ると、下記の様な ActiveRecord モデルの XML 表現を得ることができる。

```
<categories type="array">
<category>
<category>Category 1</category>
<created-at type="datetime">2008-02-26T00:59:35+09:00</created-at>
<id type="integer">1</id>
<updated-at type="datetime">2008-02-26T00:59:35+09:00</updated-at>
</category>
<category>
<category>Category 2</category>
<created-at type="datetime">2008-02-26T00:59:50+09:00</created-at>
<id type="integer">2</id>
<updated-at type="datetime">2008-03-06T00:41:39+09:00</updated-at>
</category>
<category>
<category>New Category</category>
<created-at type="datetime">2008-03-05T21:25:08+09:00</created-at>
<id type="integer">3</id>
<updated-at type="datetime">2008-03-09T13:25:57+09:00</updated-at>
</category>
</categories>
```

- `render :action => ...` によって、異なるテンプレートをレンダリングさせることができる。たとえば、新規のカテゴリーを登録する create アクションが失敗した場合、create.`html.erb`（これは存在しないのだが）の代わりに new.`html.erb` をレンダリングできる。
- `redirect_to` はもう一歩進んで、外部の'302 Found'という HTTP レスポンスを利用し、コントローラに戻ってこさせる。たとえば、destroy アクションは、テンプレートをレンダリングする必要が無い。その主な目的（カテゴリーを破棄すること）を実行した後、destroy メソッドは単純にユーザーを `categories_url` に誘導する。categories_url は、カテゴリ一覧の取得のための URL を生成するメソッドである。また、`redirect_to(@category)` は、`@category` で示されているリソースを表示するための URL を生成する。

Documentation: ActionController::Base

コントローラは、`find, new, save, update_attributes` そして destroy という Active Record のメソッドを使って、データベーステーブルとの間でデータをやり取りする。いかなる SQL 文も書く必要が無い点に注目してほしい。ただ、Rails がどのような SQL を使っているか知りたければ、`development.log` ファイルに全て記録されている。

Documentation: ActiveRecord::Base

ユーザーの視点から見ると１つの論理的アクションを実行する際に、コントローラを２回通らなければならないことに注意してほしい。たとえば、テーブルの中のレコードを更新するような場合である。ユーザーが'Edit'を選択すると、コントローラはモデルの中からユーザーが編集したいレコードを抽出し、それからeditビューをレンダリングする。ユーザーが編集を終えると、editビューは、モデルを更新し、showアクションを起動するupdateアクションを起動する。

ビュー

ビューはユーザーインターフェイスが定義される部分である。Railsは次の3つの部品から最終的なHTMLを生成することができる。

レイアウト(Layout)	*テンプレート(Template)*	*パーシャル(Partial)*
app\views\layouts\の下 *デフォルト:* ***application.html.erb*** *又は* ***<controller>.html.erb***	***app\views\<controller>*** ***の下*** *デフォルト:* ***<action>.html.erb***	***app\views\<controller>*** ***の下*** *デフォルト:* ***_<partial>.html.erb***

- レイアウトは、全てのアクションに共通なコードを提供する。典型的なものとしてブラウザに送られるHTMLの始めと終わりがある。
- テンプレートは、アクションに特有なコードを提供する。たとえば、'index'のコードや、'edit'のコードなどである。
- パーシャルは、複数のアクションで利用可能な、共通なコードの‘サブルーチン’を提供する。たとえば、フォームのための表をレイアウトするためのコードなどである。

レイアウト

Railsの命名規則: app\views\layouts\ディレクトリの中に、コントローラと同じ名前のレイアウトがあれば、特に明示的に異なる指定がされていない限り、そのコントローラのレイアウトとして使用される。

コントローラと同じ名前のレイアウトが存在せず、明示的に指定されたレイアウトが無い場合、application.html.erbあるいはapplication.xml.erbという名前のファイルがレイアウトとして使用される。

Scaffoldスクリプトが生成するレイアウトは、下記のようなものである。

app\vies\layouts\categories.html.erb

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="content-type" content="text/html;charset=UTF-8" />
  <title>Categories: <%= controller.action_name %></title>
  <%= stylesheet_link_tag 'scaffold' %>
</head>
<body>

<p style="color: green"><%= flash[:notice] %></p>

<%= yield  %>

</body>
</html>
```

これは、ほとんどがHTMLだが、中に少しだけ<% %>で囲まれたRubyのコードが含まれている。このレイアウトは、どのようなアクションが実行されるかにかかわらず、レンダリングプロセスから呼び出される。このレイアウトには、<html><head>.. .</head><body>.. .</body></html>という標準的なHTMLタグが含まれている。これは全てのページに表れるものだ。Railsのレンダリングプロセスにより、太字で示したRubyのコードが次のようにしてHTMLに変換される:

- action_nameはActionControllerのメソッドで、コントローラが実行中のアクション(たとえば'index')の名前を返す。これにより実行されたアクションに応じたタイトルがページ上に表示される。

Documentation: ActionController::Base

- `stylesheet_link_tag` は railsの Helper、すなわち楽をしてコードを生成する方法である。Railsにはたくさんの“Helper”がある。この Helper は単純に次の HTML を生成する:

  ```
  <link href="/stylesheets/scaffold.css?xxxxxxxxxx" media="screen"
  rel="Stylesheet" type="text/css" />
  ```

  (訳注:?xxxxxxxxxx はcssがキャッシュされることで更新がすぐに反映されない不具合を防止するためのダミーパラメータである)

Documentation: ActionView::Helpers::AssetTagHelper

`yield` は次に何が起こるかのカギである。これによって、たった一つのレイアウトの中に、実行されるアクション(たとえば`'edit', 'new', 'update'`)に応じて、レンダリング時に動的なコンテントを挿入することができるのだ。この動的なコンテントは同名のテンプレートからくるものだ。下記を参照すること。

Documentation: ActionController::Layout::ClassMethods

テンプレート

Railsの命名規則: テンプレートは次の場所にある `app\views\categories\<アクション>.html.erb`

Scaffold スクリプトによって作成された new.html.erb は下記の通り:

app\views\categories\new.html.erb

```
<h1>New category</h1>

<% form_for(@category) do |f| %>
  <%= f.error_messages %>

  <p>
    <%= f.label :category %><br />
    <%= f.text_field :category %>
  </p>
  <p>
    <%= f.submit 'Create' %>
  </p>
<% end %>

<%= link_to 'Back', categories_path %>
```

- `form_for` は HTML のフォームを生成するヘルパーで次のタグを生成する。ブロックを引数とし、ブロックからの出力は下記の...に挿入される。<%= %>ではなく<% %>である事に注意。

  ```
  <form  action="/categories/create" method="post">...</form>
  ```

- `submit` はそれ自体

  ```
  <input name="submit" type="submit" value="Save changes" />
  ```

  というタグを生成するが、“Create”パラメータにより、既定の“Save changes”が“Create”に書き換えられる。

- `link_to` は単純にリンク、HTML の最も基本的な部分を生成する。

  ```
  <a href="/categories/list">Back</a>
  ```

Documentation: ActionView::Helpers::UrlHelper

- **`error_messages_for`** は、フォームをサブミットする際に発生したエラーメッセージをマークアップしたテキストを返す。ひとつ以上のエラーが検出された場合、当該の HTML は次のようになる:

```
<div class="errorExplanation" id="errorExplanation">
<h2>n errors prohibited this xxx from being saved</h2>
<p>There were problems with the following fields:</p>
<ul>
<li>field_1 error_message_1</li>
<li>... ...</li>
<li>field_n error_message_n</li>
</ul>
</div>
```

これの実例を 1 日目のページ 9 の「図 2:データエラーを捕捉する」で目にしている。CSS のタグが、scaffold スクリプトで生成されたスタイルシートの文に対応していることに注意。

Documentation: ActionView::Helpers::ActiveRecordHelper

- `label` はつぎの `HTML` を生成する

  ```
  <label for="category_category">Category</label><br />
  ```

- `text_field` はつぎの HTML を生成する Rails ヘルパーである:

  ```
  <input id="category_category"  name="category[category]" size="30"
  type="text" value="" />。
  ```

  最初のパラメータがモデル名、2 番目が属性名であるが、form_for のブロックパラメータに対して呼び出す際にはモデル名は省略される。

Documentation: ActionView::Helpers::FormHelper

‘New’アクション用のレンダーされたビュー

さて、いよいよ‘New’アクションの結果として、ブラウザに返されるコードと、それがどこから来たかを見ることにしよう。レイアウトからの出力は太字、テンプレートからの出力は普通のフォントで示してある:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="content-type" content="text/html;charset=UTF-8" />
  <title>Categories: new</title>
  <link href="/stylesheets/scaffold.css?1247486792" media="screen"
rel="stylesheet" type="text/css" />
</head>
<body>

<p style="color: green"></p>

<h1>New category</h1>

<form action="/categories" class="new_category" id="new_category"
method="post"><div style="margin:0;padding:0"><input name="authenticity_token"
type="hidden" value="os27acNECI3pkwMWN4DY+5YcU6wBCZT2+uzy8vcL1eA=" /></div>


  <p>
    <label for="category_category">Category</label><br />
```

```
    <input id="category_category" name="category[category]" size="30"
type="text" />
  </p>
  <p>
    <input id="category_submit" name="commit" type="submit" value="Create" />
  </p>
</form>

<a href="/categories">Back</a>

</body>
</html>
```

'Index'アクションのためのビューの分析

'Edit'とのビューは、'New'ビューと同様である。'Index' にはいくつかの新しい仕掛けが含まれている。テンプレートは次の通り:

```
app\views\categories\index.html.erb
<h1>Listing categories</h1>

<table>
  <tr>
    <th>Category</th>
  </tr>

<% @categories.each do |category| %>
  <tr>
    <td><%=h category.category %></td>
    <td><%= link_to 'Show', category %></td>
    <td><%= link_to 'Edit', edit_category_path(category) %></td>
    <td><%= link_to 'Destroy', category, :confirm => 'Are you sure?', :method => :delete
%></td>
  </tr>
<% end %>
</table>

<br />

<%= link_to 'New category', new_category_path %>
```

- h は自動的に'HTML'コードをエスケープする。ユーザーが入力し、それが表示される場合の問題のひとつは、ユーザーが偶然(あるいは悪意を持って)表示されたときにシステムを破壊するコードを入力できることにある[4]。この問題に対する防御策として、ユーザーが入力したデータについて HTML をエスケープするのは良い習慣である。たとえば、`</table>`は、`< /table&gt` のようにレンダリングされるが、これは無害である。Rails ではこれは非常に簡単であり、ここに示したとおり、'h'をつけるだけでよい。
- confirm は、`link_to` ヘルパーの有用なオプションであり、リンク先を実行する前に、ユーザーに対して削除の確認を求める JavaScript のポップアップボックスを表示する。

図 3: JavaScript ポップアップ

Documentation: ActionView::Helpers::UrlHelper

4 *たとえば、ユーザーが Category に</table>と入力した場合を考えていただきたい。*

生成された Scaffold コードを仕上げる

Scaffold スクリプトが生成したコードは、そのままでも完璧に機能するし、データモデルに十分な検証機能を付け加えれば、強固になる。しかしながら、それが Rails アプリケーション開発でしなければならないことの全てであれば、プログラマは職を失うだろう。それは明らかにいいことではない。:-) だから、仕上げをしよう。

コントローラ

'Index'ビューでは、レコードがアルファベット順に並んでいることを期待するだろう。これには、コントローラにほんの少しだけ変更が必要だ:

```
app\controllers\categories_controller.rb (抜粋)
  def index
    @categories = Category.all(:order => :category)

    respond_to do |format|
      format.html # index.html.erb
      format.xml  { render :xml => @categories }
    end
  end
```

別解:
Rails 2.3 では named_scope という機能が使えるようになっている。これはモデルの find 機能にあらかじめ条件や並び順などを設定し、メソッドとして使用できるようにしたものである。また、モデルには default_scope として暗黙の条件や並び順を指定する事もできる。これを利用して Category モデルに並び順を指定したものが下記のコードである。

```
app\models\category.rb
class Category < ActiveRecord::Base
  default_scope :order => 'category'
  validates_length_of :category, :within => 1..20
  validates_uniqueness_of :category, :message => "already exists"
end
```

このアプリケーションでは、Show 画面は必要ない—全ての列が一覧表の画面にきちんと収まる。だから、`show` の定義は無くても良く、'Create', 'Edit' の後は、直接一覧表示に戻ることにしよう:

```
app\controllers\categories_controller.rb (抜粋)
  def create
    @category = Category.new(params[:category])

    respond_to do |format|
      if @category.save
        flash[:notice] = 'Category was successfully created.'
        format.html { redirect_to(categories_url) }
        format.xml  { render :xml => @category, :status => :created,
                                                :location => @category }
      else
        format.html { render :action => "new" }
        format.xml  { render :xml => @category.errors,
                             :status => :unprocessable_entity }
      end
    end
  end

  def update
    @category = Category.find(params[:id])

    respond_to do |format|
      if @category.update_attributes(params[:category])
        flash[:notice] = 'Category was successfully updated.'
        format.html { redirect_to(categories_url) }
        format.xml  { head :ok }
      else
```

```
        format.html { render :action => "edit" }
        format.xml  { render :xml => @category.errors,
                             :status => :unprocessable_entity }
      end
    end
  end
```

フラッシュメッセージは次の画面でピックアップされて表示される。この場合には index 画面である。Scaffold スクリプトでは、既定ではフラッシュメッセージは表示されないが、まもなく変更してみることにする。下記を参照のこと。

ビュー

フラッシュメッセージを表示する

Rails は、ユーザに対してフラッシュメッセージを返すテクニックを提供している。たとえば、'正常に更新されました'などというメッセージが、次の画面に表示され、次には消えるというものだ。これらのメッセージは、レイアウトにちょっとした変更を加えることで取り出すことができる。(レイアウトに変更を加えるということは、全ての画面に表示されるということである):

(訳注:Rails 1.2.x 以降では、レイアウトですでに次のような指定がされている。

```
<p style="color: green"><%= flash[:notice] %></p>
```

このため、フラッシュメッセージは緑色で表示される。ここではこれをスタイルシートで変更する手順として紹介する)

app\views\layouts\categories.html.erb

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="content-type" content="text/html;charset=UTF-8" />
  <title>Categories: <%= controller.action_name %></title>
  <%= stylesheet_link_tag 'scaffold' %>
</head>
<body>

<h1><%= @heading %></h1>
<% if flash[:notice] %>
<span class="notice">
<%= h flash[:notice] %>
</span>
<% end %>

<%= yield  %>

</body>
</html>
```

Documentation: ActionController::Flash

スタイルシートに簡単な追加を行うことで、フラッシュメッセージを目立たせることができる:

public\stylesheets\scaffold.css (抜粋)

```
.notice {
color: red;
}
```

テンプレートとレイアウトの間で変数を共有する

<h1>...</h1> というヘッダテキストが、フラッシュメッセージの上に表示されるようにするために、これをテンプレートからレイアウトに移したことに注意してほしい。テンプレートそれぞれには異なったヘッダがあるので、テンプレートの中で、@heading という変数に値をセットした。Rails では、これは何の問題もない—テンプレート変

数は、レンダリング時にレイアウトから参照可能である。

app\views\categories\index.html.erb
```
<% @heading = "Category" %>

<table>
  <tr>
    <th>Category</th>
  </tr>

<% @categories.each do |category| %>
  <tr>
    <td><%=h category.category %></td>
    <td><%= link_to 'Edit', edit_category_path(category) %></td>
    <td><%= link_to 'Destroy', category, :confirm => 'Are you sure?',
                                         :method => :delete %></td>
  </tr>
<% end %>
</table>

<br />

<%= link_to 'New category', new_category_path %>
```

Edit 画面と New 画面の仕上げ

NewとEditで使われるテンプレートのいくつかの変更:ユーザーが`Category`列に長すぎる文字列を入力することを防止するなどである。

二つのテンプレートへのわずかばかりの変更(とくに`@heading`の使用に注目):

app\views\categories\edit.html.erb
```
<% @heading = "Editing category" %>

<% form_for(@category) do |f| %>
  <%= f.error_messages %>

  <p>
    <%= f.label :category %><br />
    <%= f.text_field :category, :size => 20, :maxlength => 20 %>
  </p>

  <p>
    <%= f.submit "Update" %>
  </p>
<% end %>

<%= link_to 'Show', @category %> |
<%= link_to 'Back', categories_path %>
```

app\views\categories\new.html.erb
```
<% @heading = "New category" %>

<% form_for(@category) do |f| %>
  <%= f.error_messages %>

  <p>
    <%= f.label :category %><br />
    <%= f.text_field :category, :size => 20, :maxlength => 20 %>
  </p>
```

```
  <p>
    <%= f.submit "Create" %>
  </p>
<% end %>

<%= link_to 'Back', categories_path %>
```

これで、Rails の旅二日目は終わりである。Categories テーブルの保守をするための実用的なシステムが出来上がった。Rails が生成した Scaffold コードの扱い方を習得し始めたわけだ。

Rails　3 日目

さて、アプリケーションの中核部分の作業に入ろう。Item テーブルは、‘ToDo’のリストを格納する。それぞれの Item は 2 日目に作成したカテゴリのひとつに属する。Item にはオプションで、別のテーブルに Note を持つことがある。これは明日見るとしよう。それぞれのテーブルには’id’という主キーがあり、これを使って複数のテーブルの間のリンクを構成する。

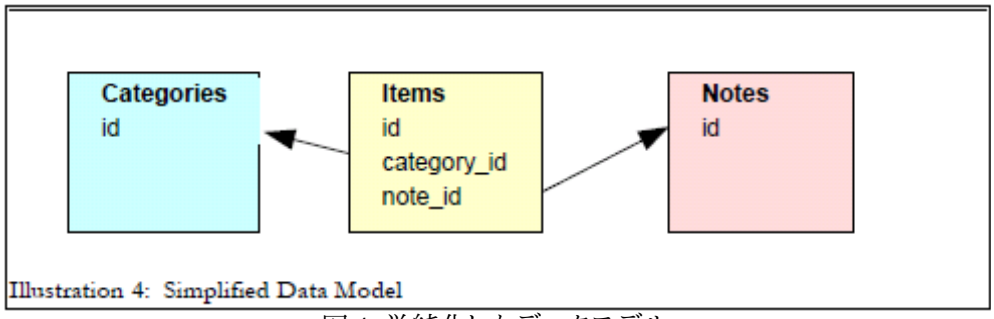


図 4: 単純化したデータモデル

アイテムテーブル

Scaffold の生成

Category と同様、スクリプトを使用して Item モデルの Scaffold を生成する。（コマンドは 1 行で入力する）

```
W:\ToDo>ruby script/generate scaffold item done:boolean priority:integer
description:string due_date:date category_id:integer note_id:integer  private:boolean
      exists  app/models/
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/items
      exists  app/views/layouts/
      exists  test/functional/
      exists  test/unit/
      exists  test/unit/helpers/
      exists  public/stylesheets/
      create  app/views/items/index.html.erb
      create  app/views/items/show.html.erb
      create  app/views/items/new.html.erb
      create  app/views/items/edit.html.erb
      create  app/views/layouts/items.html.erb
   identical  public/stylesheets/scaffold.css
      create  app/controllers/items_controller.rb
      create  test/functional/items_controller_test.rb
      create  app/helpers/items_helper.rb
      create  test/unit/helpers/items_helper_test.rb
       route  map.resources :items
  dependency  model
      exists    app/models/
      exists    test/unit/
      exists    test/fixtures/
      create    app/models/item.rb
      create    test/unit/item_test.rb
      create    test/fixtures/items.yml
      exists    db/migrate
      create    db/migrate/20090711092102_create_items.rb
W:\ToDo>
```

scaffold.css を編集した場合、上書きするかどうかを聞いてくる。n で上書きをさせないようにしよう。

マイグレーションファイルを確認する

Item テーブルの列は次の通り:

- done ― true は To Do アイテムが完了したことを示す[5]
- priority ―1 (重要度高) から 5 (重要度低)
- description ― 何をしなければならないかを記述した自由文
- due_date ― いつまでに行われなければならないかの日付
- category_id ― このアイテムが属するカテゴリーへのリンク（Category テーブルの‘id’列）
- note_id ― アイテムを説明するオプションのノートへのリンク（Notes テーブルの‘id’列）
- private ― true は ToDo アイテムがプライベートであることを示す。

```
db\migrate\20090711092102_create_items.rb
class CreateItems < ActiveRecord::Migration
  def self.up
    create_table :items do |t|
      t.boolean :done
      t.integer :priority
      t.string :description
      t.date :due_date
      t.integer :category_id
      t.integer :note_id
      t.boolean :private

      t.timestamps
    end
  end

  def self.down
    drop_table :items
  end
end
```

マイグレーションの実行

rake スクリプトでマイグレーションを実行する。

```
W:\ToDo>rake db:migrate
(in W:/ToDo)
== CreateItems: migrating =============================================
-- create_table(:items)
   -> 0.8320s
== CreateItems: migrated (0.8320s) ====================================


W:\ToDo>
```

モデルの編集

マイグレーションの後、モデルファイルを次のように編集する:

```
app\models\item.rb
class Item < ActiveRecord::Base
  belongs_to :category
```

5 MySQL には'boolean'型はないので、0/1 を使わねばならないが、マイグレーションは自動的に boolean と 0/1 のマッピングを行う。

```
  validates_inclusion_of :priority, :in => 1..5,
                         :message => 'must be between 1(high) and 5(low)'
  validates_presence_of :description
  validates_length_of :description, :maximum => 40
  validates_presence_of :category
end
```

テーブル間のリンクを検証する

- belongs_to と validates_presencse_of を使って、Item テーブルの category_id を Category テーブルの id とリンクを検証する。

  （訳注：本テキストの以前のバージョンでは validates_associated を使って検証を行うとされていたが、validates_associated は、関連するモデルについても検証を行うためのメソッドである）

Documentation: ActiveRecord:Associations::ClassMethods

ユーザー入力の検証

- validates_presence_of は、'NOT NULL' 列に対してユーザー入力がないという事態を防止する。
- validates_inclusion_of ユーザー入力を許されている値の範囲と照合する。
- validates_length_of ユーザーが保存されるときに切り捨てられてしまうようなデータを入力することを防止する。[6]
- validates_format_of はユーザー入力のフォーマットを正規表現と比較する。
- ユーザーが数値列に対して入力すると、Rails は常にそれを数値に変換し、もし変換に失敗した場合には 0 として扱う。ユーザーが実際に入力したものが数値かどうかをチェックしたい場合には、入力を before_type_cast を使って検証する必要がある。これによって「生の」入力にアクセスできる。[7]

ActiveRecord::Validations::ClassMethods

'Notes'テーブル

このテーブルは、特定の ToDo アイテムに関する詳細な情報を保持する、1 個の自由文入力列を持つ。この情報は、Item テーブルの列にあっても良い情報であるが、こうしたほうが Rails について、より多くのことを知ることができる。:-)

Scaffold を生成する

```
W:\ToDo>ruby script/generate scaffold note more_notes:text
      exists  app/models/
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/notes
      exists  app/views/layouts/
      exists  test/functional/
      exists  test/unit/
      exists  test/unit/helpers/
      exists  public/stylesheets/
      create  app/views/notes/index.html.erb
      create  app/views/notes/show.html.erb
      create  app/views/notes/new.html.erb
      create  app/views/notes/edit.html.erb
      create  app/views/layouts/notes.html.erb
   identical  public/stylesheets/scaffold.css
      create  app/controllers/notes_controller.rb
```

6 Description 列用の 2 つの検証をひとつにまとめることもできる: validates_length_of :description, :within => 1..40

7 もっと明白に使えそうな代替案: validates_inclusion_of :done_before_type_cast,:in=>"0".."1", :message=>"must be between 0 and 1"は、もし入力フィールドがブランクのままだとうまく動作しない。(訳注:0/1 を使用する場合の参考として)

```
      create  test/functional/notes_controller_test.rb
      create  app/helpers/notes_helper.rb
      create  test/unit/helpers/notes_helper_test.rb
       route  map.resources :notes
  dependency  model
      exists    app/models/
      exists    test/unit/
      exists    test/fixtures/
      create    app/models/note.rb
      create    test/unit/note_test.rb
      create    test/fixtures/notes.yml
      exists    db/migrate
      create    db/migrate/20090711115913_create_notes.rb
W:\ToDo>
```

マイグレーションファイルの確認

db\migrate\20090711115913_create_notes.rb

```
class CreateNotes < ActiveRecord::Migration
  def self.up
    create_table :notes do |t|
      t.text :more_notes

      t.timestamps
    end
  end

  def self.down
    drop_table :notes
  end
end
```

マイグレーションの実行

```
W:\ToDo>rake db:migrate
(in C:/work/ToDo)
== CreateNotes: migrating =====================================================
-- create_table(:notes)
   -> 0.6510s
== CreateNotes: migrated (0.6510s) ============================================


W:\ToDo>
```

モデルファイルの編集

モデルファイルを編集する。今のところ新しいものは何も無い:

app\models\note.rb

```
class Note < ActiveRecord::Base
  has_one :item
  validates_presence_of :more_notes
end
```

しかし、Items モデルに次のリンクを加えておくことは忘れることができない:

app\models\item.rb (抜粋)

```
class Item < ActiveRecord::Base
  belongs_to :note
```

参照整合性維持のために dependtdent オプションを利用する

これから開発するコードでは、ユーザーは任意の Item に対して 1 件の Note を加えられるようにする。しかし、ユーザーが Note を加えた後に Item を削除したらどうなるだろう?明らかに、われわれは Note レコードも削除する方法を見つける必要がある。さもなければ、「孤児となった」Note レコードが残ってしまう。

モデル／ビュー／コントローラという設計手法では、このコードはモデルに帰属する。なぜか?後ほど、われわれは、ToDo 画面において、ゴミ箱アイコンをクリックすることで Item を削除できるようになるが、完了した Item を”Purge”をクリックすることでも削除できるようになる。このコードをモデルに入れておけば、削除アクションがどこで行われても、実行されることになる。Rails の ActiveRecord のリレーション定義では、dependent というオプションを使用して、この「連鎖削除」の機能を簡単に実現する事ができる。

app\models\item.rb (抜粋)
```
class Item < ActiveRecord::Base
  belongs_to :note, :dependent => :destroy
  belongs_to :category
```

:dependent => :destory という指定を行うと、Item モデルのオブジェクトが削除される際、それと関係する Note オブジェクトの destroy メソッドが呼び出される。

同様に、Notes テーブルからレコードが削除される際には、Items テーブル側でそのレコードへの参照を削除する必要がある:

app\models\note.rb
```
class Note < ActiveRecord::Base
  has_one :item, :dependent => :nullify
  validates_presence_of :more_notes
end
```

:dependent => :nullify という指定により Note モデルのオブジェクトが削除される際に、それと関係する Item モデルのオブジェクトの外部キー note_id が nil に設定される。

Documentation: ActiveRecord::Associations::ClassMethods

訳注:原著では、dependent 機能がないバージョンの Rails を使用していたため、これらの連鎖削除機能は before_destroy コールバックを定義する事で実現されていた。

もう少しビューについて

アプリケーション用のレイアウトを作る

ここまでで、全てのテンプレートに数行の同じコードが入っていることが明らかになっている。したがって、この共通コードをアプリケーション全体のレイアウトに移すことには意味がある。
app\views\layouts*.html.erb ファイルを全て削除し、共通の application.html.erb で置き換える。

app\views\layouts\application.html.erb
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="content-type" content="text/html;charset=UTF-8" />
<title><%= @heading %></title>
<%= stylesheet_link_tag 'todo' %>
<script language="JavaScript">
<!-- Begin
function setFocus() {
  if (document.forms.length > 0) {
    var field = document.forms[0];
    for (i = 0; i < field.length; i++) {
      if ((field.elements[i].type == "text") || (field.elements[i].type == "textarea")||
```

```
(field.elements[i].type.toString().charAt(0) == "s")) {
        document.forms[0].elements[i].focus();
        break;
      }
    }
  }
}
// End -->
</script>
</head>
<body OnLoad="setFocus()">
<h1><%=@heading %></h1>
<% if flash[:notice] %>
<span class="notice">
<%=h flash[:notice] %>
</span>
<% end %>
<%= yield %>
</body>
</html>
```

Templateで設定された`@heading`は、今回は`<h1>`だけでなく`<title>`でも使われている。`public/stylesheets/scaffold.css`はキチンとするために`todo.css`という名前に変え、レイアウトを美しくするために色、テーブルの罫線に手を加えてある。また、簡単なJavaScriptを加え、ブラウザで、ユーザーがすぐに入力できるように、最初の入力フィールドにカーソルが来るようにした。

‘To Do List’画面

目指しているのは、PalmPilotや似たようなPDAの画面である。最終製品は、次ページ図5の‘To Do List’に示す。[8]

ポイント:

- 列見出しの前のチェックマークボタンをクリックすると、完了済み（チェックがついたもの）のアイテム全てが削除される。
- 表示画面は、‘Pri’, ‘Description’, ‘Due Date’, そして‘Category’の各列見出しをクリックすることで並べ替えができる。
- ‘Done’列のtrue/false値は、小さなチェックマークアイコンに変換される
- 期日が過ぎたアイテムは赤字の太字で表示される
- 付随するnoteがある場合にはnoteアイコンで表示される
- ‘Private’列のtrue/false値は南京錠のアイコンで表示される
- それぞれのアイテムは右端のアイコンをクリックすることで、編集や削除ができる
- ディスプレイはきれいな縞模様効果で表示される
- 新しいアイテムは画面下部の‘New To Do...’ボタンをクリックすることで追加できる
- 2日目に作成したCategoryへのボタンリンクもある

8　スタイルシートの数行と、アイコンがいくつかあるだけで、画面の感じがずいぶん違うものだ・・・

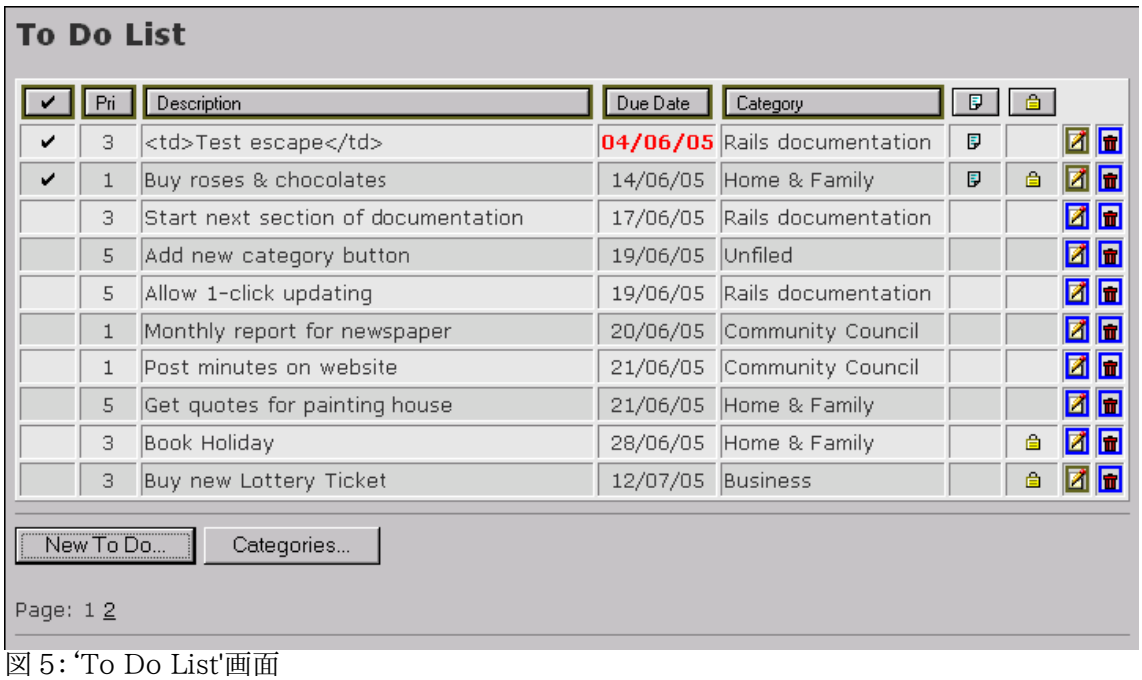


図 5：'To Do List'画面

この画面を構成するテンプレートは次の通り：

app\views\items\index.html.erb

```
<% @heading = "To Do List" %>
<% form_tag :action => 'new' do %>
<table>
  <tr>
    <th><%= link_to(image_tag('done.png'), {:action => 'purge_completed'},
    :confirm => 'Are you sure you want to permanently delete all completed To Dos?',
    :method => :post) %></th>
    <th><%= link_to(image_tag('priority.png', 'alt' => 'Sort by Priority'),
                                {:action => 'index', :order => 'priority'}) %></th>
    <th><%= link_to(image_tag('description.png', 'alt' => 'Sort by Description'),
                                {:action => 'index',:order => 'description'}) %></th>
    <th><%= link_to(image_tag('due_date.png', 'alt' => 'Sort by Due Date'),
                                {:action => 'index'}) %></th>
    <th><%= link_to(image_tag('category.png', 'alt' => 'Sort by Category'),
                                {:action => 'index', :order => 'category'}) %></th>
    <th><%= show_image 'note' %></th>
    <th><%= show_image 'private' %></th>
    <th> </th>
    <th> </th>
  </tr>
<%= render :partial => 'item', :collection => @items %>
</table>
<hr />
<%= submit_tag "New To Do..." %>
<%= submit_tag "Categories...", {:type => 'button',
                                :onClick => "parent.location='#{categories_url}'" } %>
<% end %>
```

アイコンをクリックすることで完了した'To Dos'を削除する

クリッカブルイメージは、`link_to` と `image_tag` の 2 つのヘルパーメソッドを組み合わせて使用する。`image_tag` ヘルパーでは、拡張子の省略されたファイル名を使用可能である。これはデフォルトでは、イメージが `pub/images` の下にあり、拡張子が `.png` であることを想定している。イメージをクリックすると、指定されたメソッドが実行される。

`:confirm` パラメータを指定することで、JavaScript のポップアップダイアログが表示されることは前にも説明した。

Documentation: ActionView::Helpers::UrlHelper
Documentation: ActionView::Helpers::AssetTagHelper

'OK' をクリックすると purge_completed メソッドが呼び出される。この新しい purge_completed メソッドはコントローラの中で定義する必要がある:

```
app\controllers\items_controller.rb (抜粋)
  def purge_completed
    Item.destroy_all(:done => true)
    redirect_to :action => 'index'
  end
```

Item.destroy_all は、Items テーブルで、done 列が't'の全ての行を削除し、もう一度 list アクションを実行する。

Documentation: ActiveRecord::Base

列見出しをクリックしてソート順を変える

Pri アイコンをクリックすると priority(優先順位)の順番に ToDo が表示されるようにしたい。

順番を変えた一覧に別の URL を与え、別のリソースとして扱うこともできるが、ここではソート順を表すクエリパラメータ order を使って、ソート順を変える方法を選択しよう。

```
app\controllers\items_controller.rb (抜粋)
  def index
    @items =  case params[:order]
              when 'priority'     then  Item.all(:order => 'priority,due_date')
              when 'description'  then  Item.all(:order => 'description,due_date')
              else                      Item.all(:order => 'due_date,priority')
              end

    respond_to do |format|
      format.html # index.html.erb
      format.xml  { render :xml => @items }
    end
  end
```

クエリ文字列に order が定義されていない場合、つまり、http://localhost:3000/items が指定された場合には、params[:order]は nil となる。この場合には既定のソート順
:order => 'due_date,priority'が使用される。

ヘルパーの追加

Note 列および Private 列の列見出しはイメージだが、クリックはできない。ただイメージを表示するだけの show_image(name)メソッドを定義することにする。

```
app\helpers\application_helper.rb
# Methods added to this helper will be available to all templates in the application.
module ApplicationHelper
  def show_image(src)
    img_options = { "src" => src.include?('/') ? src : "/images/#{src}" }
    img_options["src"] += '.png' unless img_options["src"].include?('.')
    img_options["border"] = "0"
    tag('img', img_options)
  end
end
```

このヘルパーは、コントローラでリンクされている:

Documentation: ActionView::Helpers

JavaScript で作ったナビゲーションボタン

`onClick` は標準的な Javascript のテクニックであり、新しい Web ページに飛ぶなどのボタンアクションをハンドリングするのに使用される。しかし、Rails は、URL をきれいにすることに長けているため、Rails に正しい URL はどうなっているかをたずねる必要がある。コントローラとアクションを引数で渡すことで、url_for はその URL を返してくれる。

Documentation: ActionController::Base

パーシャルを使ってテーブルをフォーマットする

Item のリストは、きれいなストライプ効果を使って表示したい。パーシャルが答えを提供してくれる。

```
<% for item in @items %>
<%= render :partial => 'item' %>
<% end %>
```

あるいは、もっと経済的な記述方法がある。

```
<%= render :partial => 'item', :collection => @items %>
```

Documentation: ActionView::Partials

Rails は、連番である `item_counter` を Partial に渡す。これは、表の中の行を、ライトグレーと、ダークグレーの 2 色でフォーマットするための鍵である。カウンタが奇数か偶数かをテストし、奇数ならライトグレー、偶数ならダークグレーだ。

完成したパーシャルは次の通りだ:

app\views\items_item.html.erb

```
<tr class="<%= item_counter.modulo(2).nonzero? ? "dk_gray" : "lt_gray" %>">
  <td style="text-align: center">
    <%= item.done ? show_image('done_ico.gif') : ' '%>
  </td>
  <td style="text-align: center"><%= item.priority %></td>
  <td><%= item.description %></td>
  <% if item.due_date.nil? %>
       <td> </td>
  <% else %>
    <% if item.due_date < Date.today %>
      <td  class="past_due" style="text-align: center">
      <% else %>
        <td style="text-align: center">
      <% end %>
      <%= item.due_date.strftime("%Y/%m/%d") %></td>
  <% end %>
  <td><%=h item.category ? item.category.category : 'Unfiled' %></td>
  <td><%= item.note_id.nil? ? ' ' : show_image('note_ico.gif') %></td>
  <td><%= item.private? ? show_image('private_ico.gif') : ' ' %></td>
  <td><%= link_to(image_tag('edit.png', :alt => 'Edit Item'),
                                      {:action => 'edit', :id => item.id }) %></td>
  <td><%= link_to(image_tag('delete.png', :alt => 'Delete Item'),
                    {:action => 'destroy', :id => item.id},
                     :confirm => 'Are you sure you want to delete this item?',
                     :method => 'delete') %></td>
</tr>
```

Ruby を少しだけ使って、カウンターが奇数か偶数かをテストし、`class="dk_gray"`または `class="lt_gray"`をレンダーしている:

```
item_counter.modulo(2).nonzero? ? 'dk_gray' : 'lt_gray'
```

最初のクエスチョンマークまでのコードは: item_counter を 2 で割ったあまりはゼロ以外かということをたずねている

Ruby Documentation: class Numeric

行の残りの部分は本当に暗号的な *if then else* 式である。簡潔さのために読みやすさを犠牲にしている:

クエスチョンマークの前の式が真なら、コロンの前の値を返し、そうでなければコロンの後の値を返す。

Ruby Documentation: Expressions

dk_gray と lt_gray という二つのタグは、37ページのスタイルシートで定義されることになる:

public\stylesheets\todo.css (抜粋)

```
.lt_gray { background-color: #e7e7e7; }
.dk_gray { background-color: #d6d7d6; }
```

注:同様の *if then else* 構造がチェックマークを表示するためにも使われている。index_stripes["done"] が true であるときはチェックマークを、そうでなければ HTML の空白文字を表示する。

```
item.done ? show_image('done_ico') : ' '
```

データ値に基づくフォーマッティング

特定のアイテム、たとえば期日が過ぎてしまったものをハイライトするのは簡単である。

```
item.due_date < Date.today ? '<td class="past_due">' : '<td>'
```

繰り返しになるが、このためにはスタイルシートに対応する .past_due が必要である。

参照値が無い場合に対応する

ユーザーが ToDo アイテムで参照されている Category を削除してしまった場合にも対応できるようにしたい。このような場合には、Category は 'Unfiled'として表示されるべきである。

```
item.category ? item.category.category : 'Unfiled'
```

ここまで演習を進めてくれば、30ページの図 5「ToDo リスト画面」のような画面になっているはずである。

'New To Do'画面

次は、New ToDo ボタンが押された場合のことを見てみよう。またしてもここには新しい技が潜んでいる。

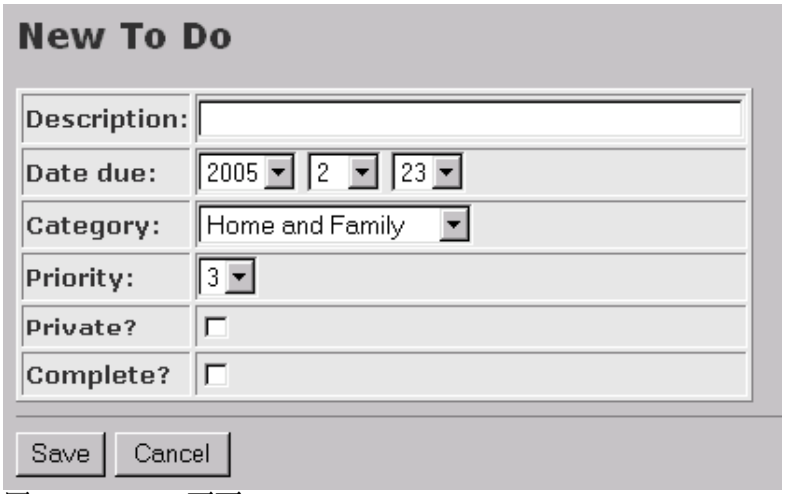


図 6 New ToDo 画面

テンプレートは最小限である:

```
app\views\items\new.html.erb
<% @heading = "New To Do" %>
<%= render :partial => 'form', :locals => {:button_name => "Save"} %>
```

renderメソッドで、:localsオプションを使う事で、パーシャルにローカル変数を渡す事ができる。
そして、本当の仕事はパーシャルの中で行われる。これは、Editアクションと共有できるものだ。

```
app\views\items\_form.html.erb
<% form_for @item do |f| %>
  <%= f.error_messages %>
  <table>
    <tr>
      <td><%= f.label :description %>:</td>
      <td><%= f.text_field :description, :size => 40, :maxlength => 40 %></td>
    </tr>
    <tr>
      <td><%= f.label :due_date %>:</td>
      <td><%= f.date_select :due_date, :user_month_number => true %></td>
    </tr>
    <tr>
      <td><%= f.label :category %>:</td>
      <td><%= f.collection_select :category_id, @categories, :id, :category %></td>
      <td><%= link_to(image_tag('note.png', :alt => 'Add category'),
                                                    new_category_url) %></td>
    </tr>
    <tr>
      <td><%= f.label :priority %>:</td>
      <td><%= f.select :priority, [1,2,3,4,5] %></td>
    </tr>
    <tr>
      <td><%= f.label :private %>?</td>
      <td><%= f.check_box :private %></td>
    </tr>
    <tr>
      <td><%= f.label :completed %>?</td>
      <td><%= f.check_box :done %></td>
    </tr>
  </table>
  <hr />
  <%= f.submit button_name %>
  <%= f.submit "Cancel", :type => 'button',
                         :onClick => "parent.location='#{items_url}'" %>
<% end %>
```

日付フィールドにドロップダウンリストを作成する

`date_select` は、日付の入力に初歩的なドロップダウンメニュを生成する。

```
date_select "item","due_date",:use_month_numbers => true
```

Documentation: ActionView::Helpers::DateHelper

Rubyで例外を捕捉する

以前のRailsでは、`date_select` は、2月31日などという日付を選択した場合、この値をデータベースに保存しようとして例外が発生していた。現在では自動的に3月3日(うるう年でなければ)に変換して処理を行うが、データベースの保存などに例外が発生した場合の回避方法のひとつとして、Rubyの例外処理メソッドであるrescueでこの保存の失敗を捕捉してしまう方法を紹介しておく。

app\controllers\items_controller.rb (抜粋)

```
  def create
    begin
      @item = Item.new(params[:item])
      if @item.save
        respond_to do |format|
          flash[:notice] = 'Item was successfully created.'
          format.html { redirect_to(items_url) }
          format.xml  { render :xml => @item, :status => :created,
                                              :location => @item }
        end
      else
        raise
      end
    rescue
      respond_to do |format|
        flash[:notice] = 'Item could not be saved.'
        @categories = Category.all
        format.html { render :action => "new" }
        format.xml  { render :xml => @item.errors, :status => :unprocessable_entity }
      end
    end
  end
```

Ruby Documentation: Exceptions, Catch, and Throw

参照テーブルからドロップダウンリストを作成する

これは、Rails がしょっちゅう発生するプログラミングの問題を、非常に簡単に解決するもうひとつの例である。この例では:

```
collection_select @categories, "id", "category", @item.category_id
```

collection_select は、select タグを生成する。そして選択肢を生成するため、@categories から全てのレコードを読み込み、<option value="*[value of id]*">*[value of category]*</option>の形でレンダリングする。@item.category_id に一致したものが選択されたものとしてタグされる。それだけではなく、データを html_escape してくれる。見事だ。

Documentation: ActionView::Helpers::FormOptionsHelper

データから作成されるドロップダウンリストは、そのデータをどこかから得る必要がある。ということは、コントローラへの追加を意味する:

app\controllers\items_controller.rb (抜粋)

```
  def new
    @categories = Category.all
    @item = Item.new

    respond_to do |format|
      format.html # new.html.erb
      format.xml  { render :xml => @item }
    end
  end

  def edit
    @categories = Category.all
    @item = Item.find(params[:id])
  end

  def create
    @item = Item.new(params[:item])

    respond_to do |format|
```

```
    if @item.save
      flash[:notice] = 'Item was successfully created.'
      format.html { redirect_to(items_url) }
      format.xml  { render :xml => @item, :status => :created, :location => @item }
    else
      @categories = Category.all
      format.html { render :action => "new" }
      format.xml  { render :xml => @item.errors, :status => :unprocessable_entity }
    end
  end
end

def update
  @item = Item.find(params[:id])

  respond_to do |format|
    if @item.update_attributes(params[:item])
      flash[:notice] = 'Item was successfully updated.'
      format.html { redirect_to(items_url) }
      format.xml  { head :ok }
    else
      @categories = Category.all
      format.html { render :action => "edit" }
      format.xml  { render :xml => @item.errors, :status => :unprocessable_entity }
    end
  end
end
```

コンスタントのリストからドロップダウンを作成する

これは、前のシナリオの単純なバージョンである。セレクションボックスにリストをハードコーディングするのは、必ずしも良い考えとはいえない。テーブル内のデータを変更するほうが、コード内のデータを変更するよりも簡単だからだ。しかし、ハードコーディングが完全に有効なアプローチである場合も存在する。そのような時は、Rails では次のように書く:

```
select :priority,[1,2,3,4,5]
```

前行で、どのようにして既定値をセットするのか注意しておくこと。

Documentation: ActionView::Helpers::FormOptionsHelper

チェックボックスを生成する

もうひとつ常に必要なこと;もうひとつの Rails のヘルパー:

```
check_box :private
```

(訳注:check_box ヘルパーの既定の値は 1,0 であるため、boolean に適合させるためには true,false のパラメータを指定する必要がある)

Documentation: ActionView::Helpers::FormHelper

仕上げ

スタイルシートを仕上げる

ここまでで、‘To Do List’画面や、‘New To Do’ボタンは機能するようになった。ここで示す画面を構築するために、スタイルシートに次のような変更を追加した:

public\stylesheets\ToDo.css

```
body { background-color: #c6c3c6; color: #333; }
.notice {
color: red;
background-color: white;
}
h1 {
font-family: verdana, arial, helvetica, sans-serif;
font-size: 14pt;
font-weight: bold;
}
table {
background-color:#e7e7e7;
border: outset 1px;
border-collapse: separate;
border-spacing: 1px;
}
td { border: inset 1px; }
.notice {
color: red;
background-color: white;
}
.lt_gray { background-color: #e7e7e7; }
.dk_gray { background-color: #d6d7d6; }
.hightlight_gray { background-color: #4a9284; }
.past_due { color: red }
```

‘Edit To Do’画面

3日目の残りの時間で‘Edit To Do’画面の構築を行う。この画面は‘New To Do’に非常に良く似ている。学生時代、教科書に「これは読者への練習問題として取っておこう」と書いてあるのにはうんざりさせられたものだ。だから、ここで皆さんに同じことができるというのはすばらしいことだ。[9]

それをもって三日目の終わりとしよう。このアプリケーションは、もはやRailsのscaffoldとは似ても似つかぬものになっているが、その裏では、開発を容易にするためのRailsの数々のツールを使っているのだ。

9 しかし、私の大学時代の教科書の著者とは違って、4日目に答えを明かすことにする。:-) 39ページの *app¥views¥items¥edit.html.erb* を参照のこと。

Rails　4日目

‘Notes’画面

‘Notes’を‘Edit To Do’にリンクする

Notes の scaffold コードには、CRUD 機能が全て備わっているが、ユーザーにはどれも直接呼び出してほしくない。その代わりに、Item に付随する Note がない場合、‘Edit To Do’画面の Notes のアイコンをクリックすることで、Note を作成できるようにしたい：

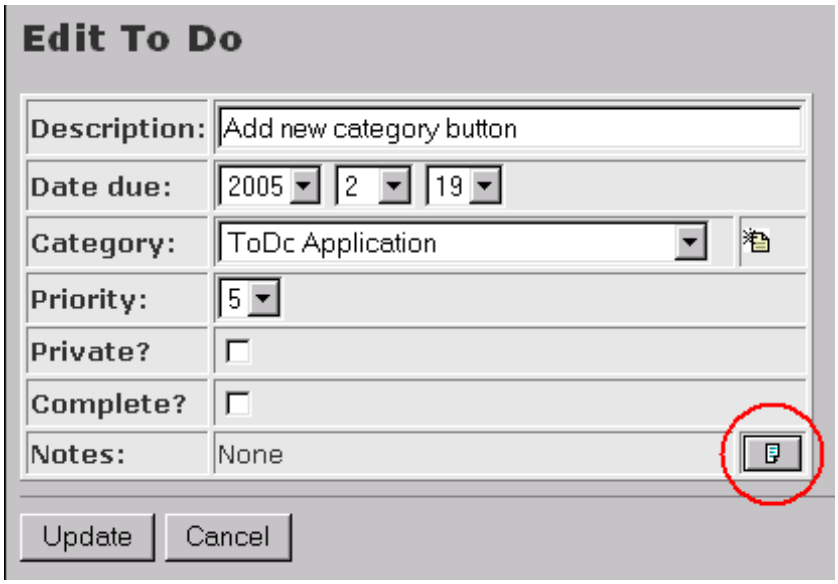


図7: ‘Edit To Do’画面から新しい Note を作成する

Note が既にある場合には、‘Edit To Do’画面の適切なアイコンをクリックすることで、編集や削除を行いたい。:

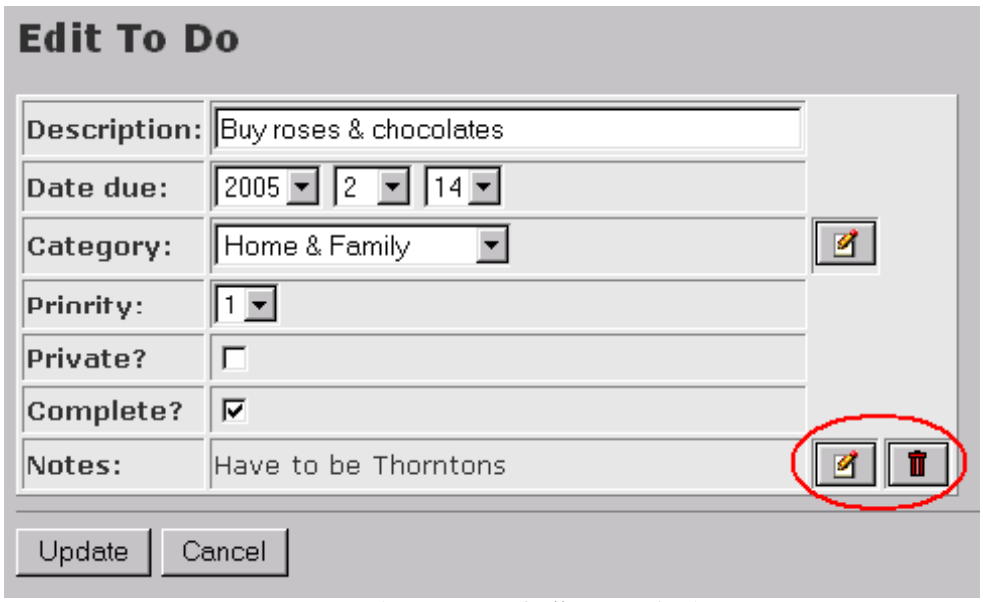


図8：既存の Note の編集または削除

まず最初に、‘Edit To Do’画面のコードを見ておこう。ノートボタンが、既存のノートがあるか否かに応じて変わることと、コントロールがどのようにしてNotesコントロールに移るかに注意。edit.html.erbは変えるところがない。

app\views\items\edit.html.erb

```
<% @heading = "New To Do" %>
<%= render :partial => 'form', :locals => {:button_name => "Update"} %>
```

フォーム中にNote関連の表示項目とリンクを表示するため、formパーシャルを変更する。new.html.erbと兼用しているため、`@item.new_record?`で動作を切り替えている。

app\views\items_form.html.erb

```
<% form_for @item do |f| %>
  <%= f.error_messages %>
  <table>
    <tr>
      <td><%= f.label :description %>:</td>
      <td><%= f.text_field :description, :size => 40, :maxlength => 40 %></td>
    </tr>
    <tr>
      <td><%= f.label :due_date %>:</td>
      <td><%= f.date_select :due_date, :user_month_number => true %></td>
    </tr>
    <tr>
      <td><%= f.label :category %>:</td>
      <td><%= f.collection_select :category_id, @categories, :id, :category %></td>
      <td><%= link_to(image_tag('note.png', :alt => 'Add category'),
                                                    new_category_url) %></td>
    </tr>
    <tr>
      <td><%= f.label :priority %>:</td>
      <td><%= f.select :priority, [1,2,3,4,5] %></td>
    </tr>
    <tr>
      <td><%= f.label :private %>?</td>
      <td><%= f.check_box :private %></td>
    </tr>
    <tr>
      <td><%= f.label :completed %>?</td>
      <td><%= f.check_box :done %></td>
    </tr>
  <% unless @item.new_record? %>
    <tr>
       <td><%= f.label :note_id %>:</td>
    <% if @item.note_id.nil? %>
      <td>None</td>
      <td><%= link_to(image_tag('note.png', :alt => 'Add note'),
             {:controller => 'notes', :action => 'new', :item_id => @item.id}) %></td>
    <% else %>
      <td><%= @item.note.more_notes %></td>
      <td><%= link_to(image_tag('edit_button.png', :alt => 'Edit note'),
                                              edit_note_url(@item.note_id)) %></td>
      <td><%= link_to(image_tag('delete_button.png', :alt => 'Delete note'),
                {:controller => 'notes', :action => 'destroy', :id => @item.note_id},
                 :confirm => 'Are you sure you want to delete this note?',
                 :method => 'delete') %></td>
    <% end %>
    </tr>
  <% end %>
  </table>
  <hr />
  <%= f.submit button_name %>
  <%= f.submit "Cancel", :type => 'button',
                         :onClick => "parent.location='#{items_url}'" %>
<% end %>
```

‘Edit Notes’画面

既存のノートを編集するのは非常にわかりやすい。以下がテンプレートである。

app\views\notes\edit.html.erb

```
<% @heading = "New Note" %>
<%= render :partial => 'form', :locals => {:button_name => "Update"} %>
```

それと、対応するパーシャル:

app\views\notes_form.html.erb

```
<% form_for @note do |f| %>
  <%= f.error_messages %>
  <table>
       <tr>
         <td><%= f.label :more_notes %></td>
         <td><%= f.text_area :more_notes %></td>
       </tr>
  </table>
  <%= f.submit button_name %>
  <%= f.submit "Cancel", {:type => 'button',
                          :onClick => "parent.location='#{items_url}'"} %>
<% end %>
```

一度、ノートの更新あるいは破棄が終われば、‘To Do List’画面に戻りたい。

app\controllers\notes_controller.rb (抜粋)

```
  def update
    @note = Note.find(params[:id])

    respond_to do |format|
      if @note.update_attributes(params[:note])
        flash[:notice] = 'Note was successfully updated.'
        format.html { redirect_to items_url }
        format.xml  { head :ok }
      else
        format.html { render :action => "edit" }
        format.xml  { render :xml => @note.errors, :status => :unprocessable_entity }
      end
    end
  end

  def destroy
    @note = Note.find(params[:id])
    @note.destroy

    respond_to do |format|
      format.html { redirect_to items_url }
      format.xml  { head :ok }
    end
  end
```

我々が作成した参照整合性のルールによって、ノートが削除された場合には、Item 内でそれを指していたいかなる参照も削除されることを思い出していただきたい。(「参照整合性の維持のためにモデルを使用する」を参照すること。

‘New Note’画面

Create はちょっとだけトリッキーだ。我々が行いたいのは、新しい Note を Notes テーブルに保存し、Notes テーブルで新しく作成された `id` を調べる。そして、それを、Items テーブルの関連する Item の `notes_id` フィールドに格納する。

セッション変数は、画面間でデータを永続化する有用な方法を提供してくれる。これによって、われわれは Notes テーブルの中のレコードの id を保存することができる。

セッション変数を使ってデータを保存し取り出す

最初に、まず新しい Notes レコードを作成する。編集している Item の id を渡す:

```
app\views\items\edit.html.erb (抜粋)
<td><%= link_to(image_tag('note.png'),
              {:controller => "notes", :action => "new", :item_id =>@item.id}) %></td>
```

Notes コントローラの新しいメソッドが、セッション変数にこのように格納する。

```
app\controllers\notes_controller.rb (抜粋)
  def new
    session[:item_id] = params[:item_id]
    @note = Note.new

    respond_to do |format|
      format.html # new.html.erb
      format.xml  { render :xml => @note }
    end
  end
```

'New Notes'テンプレートにはびっくりするような点は無い:

```
app\views\notes\new.html.erb
<% @heading = "New Note" %>
<%= render :partial => 'form', :locals => {:button_name => 'Create'} %>
```

`create` メソッドは再びセッション変数を引き出し、Items テーブルから該当するレコードを検索するために使用する。次に create メソッドは、Note テーブルに作成したばかりのレコードの id を使って、Item テーブルの `note_id` を更新する。そして、再び Items コントローラに戻る:

```
app\controllers\notes_controller.rb (抜粋)
  def create
    @note = Note.new(params[:note])

    respond_to do |format|
      if @note.save
        flash[:notice] = 'Note was successfully created.'
        @item = Item.find(session[:item_id])
        @item.update_attribute(:note_id, @note.id)
        format.html { redirect_to(items_url) }
        format.xml  { render :xml => @note, :status => :created, :location => @note }
      else
        format.html { render :action => "new" }
        format.xml  { render :xml => @note.errors, :status => :unprocessable_entity }
      end
    end
  end
```

'Categories'画面を変更する

もう、このシステムではたいしてすることは残っていない。始めの方で作成したテンプレートにちょっと手を加え、同じスタイルのナビゲーションボタンを持つようにすることくらいである。:

```
app\views\categories\index.html.erb
<% @heading = "Categories" %>
<% form_tag "/categories/new", :method => "get" do %>
  <table>
    <tr>
      <th>Category</th>
    </tr>
```

```
    <% @categories.each do |category| %>
      <tr>
        <td><%=h category.category %></td>
        <td><%= link_to(image_tag('edit.png'), edit_category_url(category.id)) %></td>
        <td><%= link_to(image_tag('delete.png'), { :action => 'destroy',
                                                   :id => category.id },
                    :confirm => 'Are you sure you want to delete this category?',
                    :method => :delete) %></td>
      </tr>
    <% end %>
    </table>
    <hr />
    <%= submit_tag "New Category..." %>
    <%= submit_tag "To Dos", {:type => 'button',
                              :onClick => "parent.location='#{items_url}'"} %>
<% end %>
```

app\views\categories\new.html.erb

```
<% @heading = "Add new Category" %>
<% form_for(@category) do |f| %>
  <%= f.error_messages %>
  <p>
    <%= f.label :category %><br />
    <%= f.text_field :category, :size => 20, :maxlength => 20 %>
  </p>
<hr />
    <%= f.submit "Save" %>
    <%= submit_tag "Cancel",
          {:type => 'button', :onClick => "parent.location='#{categories_url}'"} %>
<% end %>
```

app\views\categories\edit.html.erb

```
<% @heading = "Rename Category" %>
<% form_for(@category) do |f| %>
  <%= f.error_messages %>
  <p>
    <%= f.label :category %><br />
    <%= f.text_field :category, :size => 20, :maxlength => 20 %>
  </p>
<hr />
    <%= f.submit "Update" %>
    <%= submit_tag "Cancel",
          {:type => 'button', :onClick => "parent.location='#{categories_url}'"} %>
<% end %>
```

システム全体のナビゲーション

アプリケーションを通じた最終的なナビゲーションパスは下図の通りである。冗長な scaffold コード、たとえば `show.html.erb` などは、ただ単純に削除すればよい。これは、scaffold コードの美点である。最初につくるときには何の苦労もないし、いったん目的を果たした後は単純に捨てることができる。

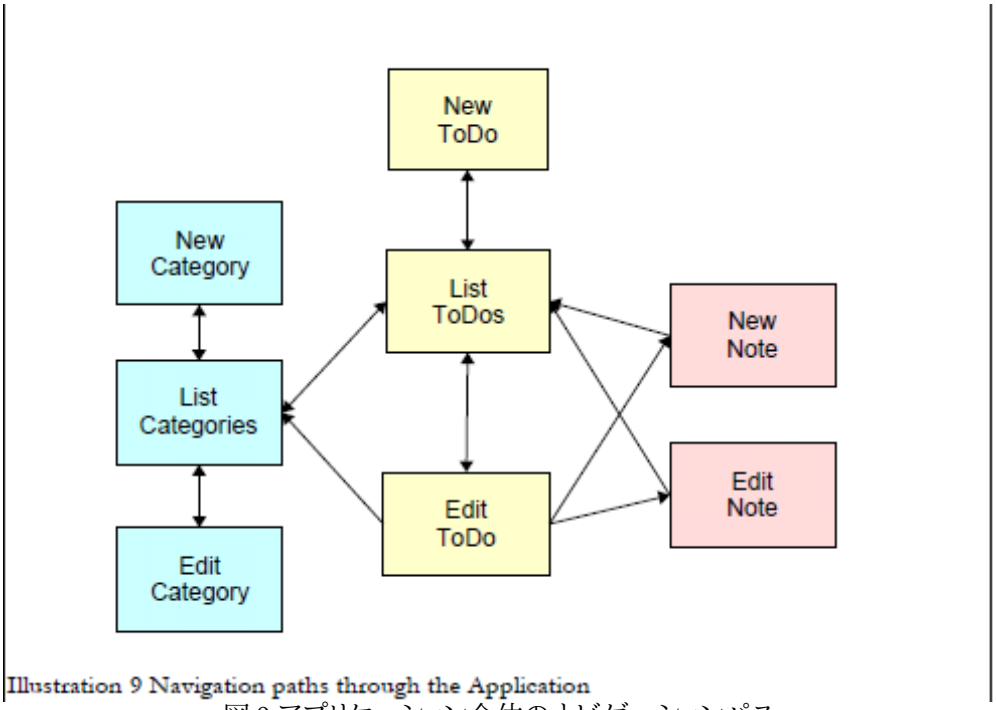


Illustration 9 Navigation paths through the Application

図 9 アプリケーション全体のナビゲーションパス

アプリケーションのホームページを設定する

最後のステップとして、ユーザーがブラウザで `http://todo` を参照した際に表示されるデフォルトの Welcome to Rails?画面を削除しよう。2 つのステップだ:

- Routes ファイルにホームページの定義を加える:

config\routes.rb (抜粋)

```
map.root :controller => 'items'
```

- `public\index.html` のファイル名を `public\index.html.orig` に変更する。

このアプリケーションの（初期のバージョンの）コピーをダウンロードする

この‘To Do’アプリケーションを試すためにのコピーをほしければ、http://rails.homelinux.org にリンクがある。

- Rails を使ってディレクトリ構造を作成する必要がある（7ページの Rails スクリプトを実行するを参照）
- `todo_app.zip` ファイルを新しくできた ToDo ディレクトリにダウンロードする
- ファイルを unzip する `unzip -o todo_app.zip`
- rename public\index.html public\index.html.orig
- サンプルデータベースを使いたければ、 `mysql -u root -p < db/ToDo.sql`

そして最後に・・・

　このドキュメントがあなたのお役に立てばと思っている ほめ言葉でも悪口でも
jpmcc@users.sourceforge.net までフィードバックをいただければうれしい。

　さあ、Rails で楽しいコーディングを！

補足:後で考えたこと

‘Four Days’を書いた後、非常にたくさんのフィードバックを戴いた。それらは、このドキュメントの質を向上させるのに非常に役立った。なかでも、繰り返し同じ質問を受けた 「同じ画面で複数のレコードを更新するにはどうすればよいか」というものである。そこで、このもっとも良く聞かれる質問のための補足を加えた。これは、簡単に理解できる Rails の概念ではない。今後もっとたくさんの Helper が出てきてほしい分野である。

複数同時更新

下記の画面では、ユーザーは一番左端のチェックボックスを使って、複数の ToDo 案件にチェックをつけたり外したりできる。その後、‘Save’をクリックして、その結果をデータベースに保存することができる。

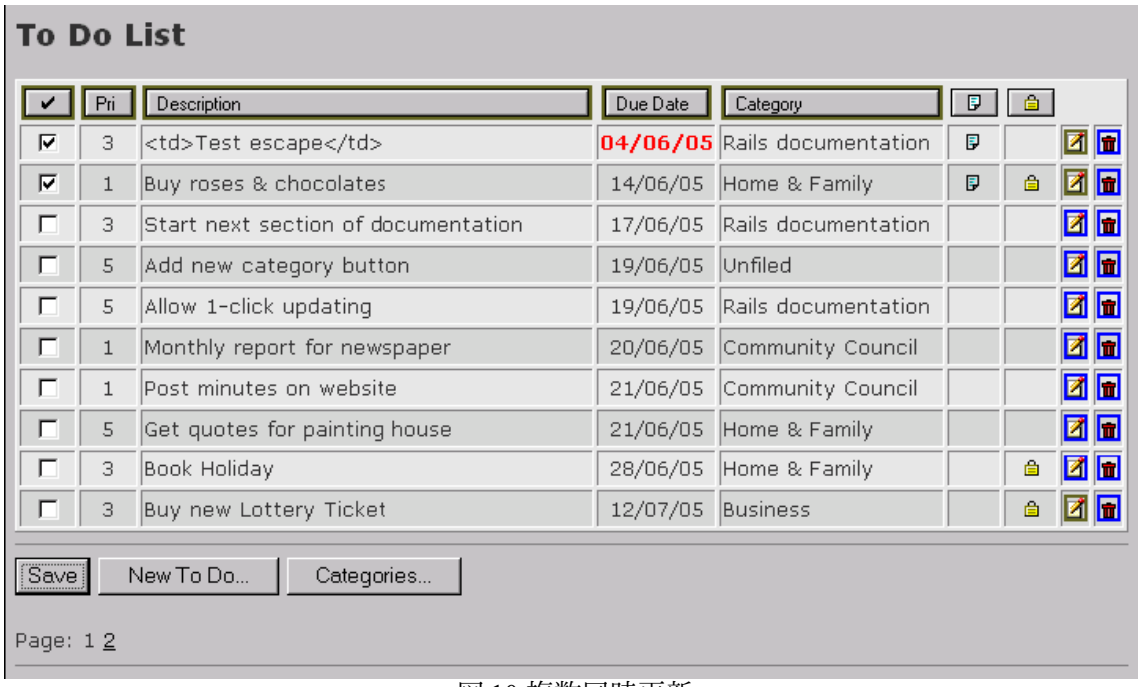


図 10 複数同時更新

ビュー

Rails はもうひとつの命名規約で複数同時更新をサポートする。それは、編集中のレコードの名前の後に角括弧[]で挟まれた id を加えるというものだ。これによって、画面上の複数のレコードから特定のレコードを選び出すことができる。我々が生成しようとしている HTML からさかのぼって見て行こう。これが、id 6のレコードがどのように見えるかだ:

```
<td style="text-align: center">
<input name="item[6][done]" type="hidden" value="false" />
<input type="checkbox" id="item_done" name="item[6][done]" value="true" checked />
</td>
```

(チェックボックスがチェックされていなければ checked は要らない)

このコードを生成する方法のひとつはこれだ:

app\view\items_index_stripes.html.erb (抜粋)

```
  <td style="text-align: center">
    <%= hidden_field_tag("item[#{item.id.to_s}][done]",'false') %>
    <%= check_box_tag("item[#{item.id.to_s}][done]",'true', item.done) %>
  </td>
```

check_box_tag のパラメータは、name, value = "true", checked = false, options =

{};

hidden_field_tag の場合は、name, value = nil, options = {} だ。

上記の例で hidden_field_tag が前になっているのには重要な意味がある。check box がチェックされていないと、その check box はパラメータとして送信されない。check box がチェックされていると、両方のパラメータが送信されるが、次のコントローラのメソッドで、これを順番に処理するため、後から送られた値で上書きが行われる事になるからである。

Documentation: ActionView::Helpers::FormTagHelper

そしてもちろん Save ボタンが必要だ:

app\views\items\index.html.erb (抜粋)

```
<% @heading = "To Do List" %>
<% form_tag :action => 'updater' do %>
<table>
...
</table>
<hr />
<%= submit_tag "Save" %>
<%= submit_tag "New To Do..",  {:type => 'button',
                                :onClick => "parent.location='#{new_item_url}'"} %>
<%= submit_tag "Categories...", {:type => 'button',
                                 :onClick => "parent.location='#{categories_url}'"} %>
<% end %>
```

コントローラ

'Save'ボタンをクリックしたときに、コントローラに戻すのは、下記のようなハッシュである:

```
params: {
:controller=>"items",
:item=> {
"6"=>{"done"=>"false"},
... etc...
"5"=>{"done"=>"true"}
},
:action=>"updater"
}
```

:item ビットに注目してみよう。たとえば、太字の行は、id が 6 のレコードでは、done フィールドの値が false ということを意味している。ここからは、Items テーブルを更新するのは比較的容易である:

app\controller\items_controller.rb (抜粋)

```
  def updater
    params[:item].each do |item_id, attr|
      item = Item.find(item_id)
      item.update_attributes(:done,attr[:done])
    end
    redirect_to items_url
  end
```

each によって変数 item_id には 6 が入り、"done" => "false"が attr に入る

Ruby Documentation: class Array

このコードでも動作はするが、development.log で何が起きているかを見ると、Rails が変更されているかいないかにかかわらず、全てのレコードを引き出し、しかも更新していることがわかるだろう。これは、不必要なデータベース更新を生成しているだけではなく、updated_on も更新されることを意味する。これは望ましいことではない。'done'が変わった場合にのみ更新されるほうがうんと良いが、そのためにはちょっとしたコーディングが必要だ。:-(

app\controller\items_controller.rb (抜粋)

```
  def updater
    params[:item].each do |item_id, attr|
      item = Item.find(item_id)
      if item.done != eval(attr[:done])
        item.update_attributes(:done,attr[:done])
      end
    end
    redirect_to items_url
  end
```

同類のものを比較するためには、string である done を eval メソッドを使って true/false に変換する必要があることに注意。この手のミスは見逃しやすい。Rails があなたの想定したとおりに動いているかどうかを確認するために、development.log を時々チェックするのは意味があることだ。

ユーザーインターフェイスに関する考察

このコードは機能するが、これを応用してさらに画面上のどの列でも編集できるようにすることもできる。(もうひとつの易しい練習問題だ:-)。ただし、それによってユーザーが何を期待するかという問題が持ち上がってくる。もしもユーザーがチェックボックスのいくつかを変更し、'Save' を押さないまま、'New To Do'をクリックしたり、表示の並べ替えを行ったらどうだろう。システムは常に、別のアクションを実行する前に、'Save' すべきだろうか?これもユーザーへの簡単な練習問題だ・・・

まだやらねばならぬこと

(訳注:Four Days on Rails 2.3 では、Rails 0.12.1 をベースにしていたため、Category で Item をソートするのは、困難であった。原著者の John McCreesh 氏は、find_by_SQL による解決策を提示していたが、Rails 1.1.6 では、外部結合がサポートされているため、下記のようにすっきりと記述することが可能である。)

app\controller\items_controller.rb (抜粋)

```
  def index
    @items =  case params[:order]
              when 'priority'     then  Item.all(:order => 'priority,due_date')
              when 'description'  then  Item.all(:order => 'description,due_date')
              when 'category'     then  Item.all(:include => :category,
                                          :order => 'categories.category,due_date')
              else                      Item.all(:order => 'due_date,priority')
              end

    respond_to do |format|
      format.html # index.html.erb
      format.xml  { render :xml => @items }
    end
  end
```

Rails で楽しくコーディングしよう!

この文書の Ruby と Rails 用語の索引