Ruby on *Rails*

Ausarbeitung zum Vortrag

präsentiert am 14. Januar 2008

im Rahmen des Seminars

„Secure Web-Application Engineering“

am Hasso-Plattner-Institut für Softwaresystemtechnik

von Christoph Barth (727625)

1. Einleitung

In der Zeit des sogenannten Web 2.0 haben Web-Frameworks[1] einen besonderen Stellenwert. Schon längst werden Webapplikationen nur noch selten „from scratch“ – also von Grund auf erstellt, vielmehr werden sie nun oft – einem Baukastensystem gleich – aus einer Vielzahl bestehender, flexibler Funktionselementen zusammengesetzt und ergeben in Kombination mit eigens entwickelten Ansätzen, Komponenten oder Anpassungen eine individuelle Webanwendung.

Diese Herangehensweise wird durch Web-Frameworks unterstützt. Auf einer bestimmten Programmiersprache aufgesetzt bieten sie u.a. Templates, Bibliotheken und Erweiterungen für die eigene Webanwendung. Derer Frameworks gibt es viele und für die unterschiedlichsten Programmiersprachen.

Ein Beispiel ist *Rails*. *Rails* steht kurz für *Ruby on Rails* und ist ein Web-Framework auf der Programmiersprache Ruby. Auf den nun folgenden Seiten soll *Rails* kurz vorgestellt werden – auch und insbesondere seine Sicherheitsmerkmale.

Vorangestellt wird eine kurze Einführung zu *Ruby* und *Rails*. Hiernach wird auf bestimmte Merkmale einer ‚sicheren‘ Webanwendung eingegangen, wie z.B. Validierung und Protokollierung und wie sie in *Rails* realisiert werden. Das darauf folgende Kapitel beschäftigt sich mit Angriffszenarien auf eine *Rails*-Anwendung und deren Prävention. Diese Arbeit schließt mit einem Ausblick auf *Rails 2.0*, der momentan aktuellen Version.

[1] Framework: Englisch: „Gestell, Rahmen“

1.1 Was ist Ruby?

Wer mit *Rails* umgehen möchte, sollte zuerst einen Blick auf die Programmiersprache *Ruby* werfen, denn sie ist Grundlage von *Rails* und in ihr wird der Entwickler einer *Rails*-Anwendung schreiben.

Ruby ist eine vollständig objektorientierte Programmiersprache aus Japan. Dort wurde sie erstmals 1993 von ihrem Entwickler Yukuhiro („Matz“) Matsumoto der Öffentlichkeit vorgestellt. Lange Zeit blieb Ruby aufgrund fehlender englischsprachiger Dokumentation in unseren Breiten unbekannt, erfreut sich jedoch etwa seit dem Jahr 2000 auch in Europa und den USA einer stetig wachsenden Community.

Ruby besticht durch eine für den Menschen besonders gut lesbare Syntax und ist deshalb recht einfach zu erlernen.

Wie viele Scriptsprachen wird Ruby interpretiert, nicht kompiliert. Dafür stehen für viele POSIX-Betriebssysteme, sowie für Windows Interpreter bereit.

1.2 Was ist *Rails*?

Rails ist ein Web-Framework, das auf besonders schnelle und einfache Erstellung von Web-Applikationen abzielt.

Dafür setzt *Rails* auf das MVC-Pattern[2], das zwingend verwendet wird. Eine Anwendung wird von *Rails* also dreigeteilt:

Models definieren Objekte, die Einträge in der Datenbank der Applikation widerspiegeln. Eine Instanz einer solchen Klasse beschreibt genau einen Datensatz in der Datenbank. *Rails* unterstützt dabei gleich eine ganze Reihe von Datenbankmanagementsystemen, wie z.B. MSSQL, Oracle, SQLite und natürlich MySQL.

Views definieren das Aussehen der Applikation, d.h. sie beschreiben das Benutzerinterface. Es ist wichtig, dass Views Funktionalitäten nicht selbst implementieren, sondern stets auf Controler verweisen. So ist es beispielsweise möglich, verschiedene Views für verschiedene Endgeräte oder Browser zu definieren (z.B. für mobile Geräte), dafür jedoch nicht in den eigentlichen Code der Anwendung eingreifen zu müssen.

Controler bieten die Geschäftslogik der Applikation. Hier wird beschrieben, wie die Applikation arbeitet.

Mit *Rails* können besonders schnell sog. CRUD[3]-Applikationen produziert werden. Es geht davon aus, dass in einer interaktiven Web-Applikation Inhalte erstellt, gelesen, verändert und gelöscht werden können und baut (auf Wunsch) durch „Scaffolding“ gleich entsprechende Models, Views und Controler, die dann das bereits lauffähige Gerüst der Anwendung darstellen.

In *Rails* gilt weiterhin der Grundsatz „Convention over Configuration“, der meint, dass *Rails* bestimmte Konventionen vorgibt, anstatt Standardeinstellungen in Konfigurationsdateien vorzugeben. So ist es beispielsweise Konvention, dass ein Model „User“ in der Datenbank auf die Tabelle „users“ gemappt wird. Es ist nicht nötig, dies in *Rails* extra zu konfigurieren, da es Konvention ist.

Natürlich darf in Zeiten des Web 2.0 die Ajax-Technologie nicht vernachlässigt werden. Zu diesem Zweck liefert *Rails* die beiden JavaScript-Frameworks „Prototype“ und „Scriptaculous“ gleich mit. Diese bieten neben vielen Funktionen, die das Leben mit Ajax stark vereinfachen auch großartige Möglichkeiten, eine Web-Anwendung optisch etwas aufzupeppen.

Rails liegt aktuell in Version 2.0.2 vor und ist freie Software. Dem Entwickler wird die Einrichtung einer Entwicklungsumgebung durch fertige Installationspakete mit eigenem Web- und Datenbankserver sehr einfach (z.B „InstantRails“ für Windows). Leider wird *Rails* noch von wenigen Webhostern unterstützt, was den Produktiveinsatz noch etwas erschwert, jedoch ist zu erwarten, dass sich dies aufgrund der wachsenden Beliebtheit des Framworks sehr bald ändern wird.

[2] Model, View, Controler
[3] Create, Read, Update, Delete – die vier Grundfunktionen einer Datenbank

2. Sicher auf Schienen

Applikationssicherheit in *Rails*

Dieses Kapitel beschäftigt sich mit einigen Bereichen, die für eine gesicherte Web-Anwendung von großer Wichtigkeit sind und wie sie im Rahmen von *Rails* umgesetzt werden.

2.1 Validierung

Mit der Validierung werden Benutzereingaben auf ihre Gültigkeit geprüft. Hierbei könnte es von Bedeutung für die weitere Verwendung dieser Daten in der Applikation sein, ob es sich um eine Zahl (z.B. einen PIN-Code), einen String (wie z.B. ein Passwort) handelt. Oft ist es aber auch von Nöten zu prüfen, ob ein Wert in einem bestimmten Bereich liegt (z.B. eine Zahl zwischen 1 und 10) oder überhaupt vorhanden ist.
Eine Validierung bestimmter Benutzereingaben sollte spätestens vor der Persistierung in die Datenbank erfolgen, um eine konsistent gültige Datenhaltung zu gewährleisten. Im Falle von *Rails* geschieht eine Validierung also direkt vor dem Speichern eines Models in die Datenbank (z.B. durch den Befehl `@[model_name].save`). Folglich wird auch im Model definiert, wie welche Daten validiert werden.

Das folgende Listing zeigt den Umgang mit Validierungen in *Rails.*

```
2  class User < ActiveRecord::Base
3    # Virtual attribute for the unencrypted password
4    attr_accessor :password
5
6    validates_presence_of     :login, :email
7    validates_presence_of     :password,                   :if => :password_required?
8    validates_presence_of     :password_confirmation,      :if => :password_required?
9    validates_length_of       :password, :within => 4..40, :if => :password_required?
10   validates_confirmation_of :password,                   :if => :password_required?
11   validates_length_of       :login,    :within => 3..40
12   validates_length_of       :email,    :within => 3..100
13   validates_uniqueness_of   :login, :email, :case_sensitive => false
14   before_save :encrypt_password
```

Listing 1: Validierungen im Model User

In Listing 1 ist z.B. zu erkennen, dass `login` und `password` auf Vorhandensein und eine bestimmte Länge (zwischen 3 bzw. 4 und 40) geprüft werden. Auch die Einzigartigkeit kann in *Rails* durch `validates_uniqueness_off` geprüft werden.

Weitere[4] Validatoren in *Rails* sind:

- `validates_format_of` – validiert ein Datum gegen einen regulären Ausdruck
- `validates_inclusion_of` – validiert ein Datum gegen eine Liste von Ausdrücken
- `validates_acceptance_of` – validiert ein Datum gegen eine Liste von Checkboxen (z.B. EndUser License Agreement)
- `validates_numericality_of` – validiert, ob ein Datum eine Zahl ist

Grundsätzlich könne auch eigene Validierungmöglichkeiten implementiert werden, dies ist jedoch in den meisten Fällen nicht nötig, da *Rails* ein recht breites Spektrum an Validatoren von Haus aus anbietet.

In manchen Sitationen mag es sinnvoll sein, auf die Validierung zu verzichten (z.B. zu Testzwecken oder im Administrationsbereich). In diesem Fall kann *Rails* angewiesen werden, auch ungültige Datensätze zu persistieren. Der `save()`-Methode wird dazu `false` als Argument übergeben.
Nach obigem Beispiel also `@[model_name].save(false)`.

2.2 Ausnahmebehandlung

Die Ausnahmebehandlung in *Rails* wird durch *Ruby Exceptions* bereitgestellt. Diese ist den Ausnahmebehandlungen anderer „großer“ Programmiersprachen wie *C#* oder *Java* sehr ähnlich. Dabei erbt jede Exception von der Klasse `Exception`, wobei *Ruby* eine Reihe vordefinierter Exceptions anbietet, jedoch sehr wohl eigene definiert werden können (die natürlich auch von `Exception` abgeleitet sein müssen). Eine entsprechende Exception-Hierarchie zeigt Listing 2.

```
1   Exception
2       NoMemoryError
3       ScriptError
4           LoadError
5       NotImplementedError
6       SyntaxError
7   SignalException
8       Interrupt
9   StandardError
10      ArgumentError
11      IOError
12          EOFError
13      IndexError
14      LocalJumpError
15      NameError
16          NoMethodError
17      RangeError
18          FloatDomainError
19      RegexpError
20      RuntimeError
21      SecurityError
22      SystemCallError
23      SystemStackError
24      ThreadError
25      TypeError
26      ZeroDivisionError
```

Listing 2: Exception-Hierarchie in Ruby

Im Listing 3 wird der Umgang mit Ruby Exceptions verdeutlich. Es ist zu erkennen, dass Code, der eine oder mehrere Exceptions ‚werfen' kann, von `begin` und `end` umschlossen werden. Die `rescue`-Anweisungen ‚fangen' die Exceptions bestimmter Typen und verarbeiten weiterführenden Code. Durch Aneinanderreihung von `rescue`-Anweisungen ist es möglich, verschiedenartigen Code für unterschiedliche Exceptions ausführen zu lassen. Mittels der `Ensure`-Anweisung wird Code bearbeitet, der auch im Falle einer Exception ausgeführt wird.

[4] eine vollständige Liste gibt es auf
http://*Rails*.rubyon*Rails*.com/classes/ActiveRecord/Validations/ClassMethods.html

```ruby
begin
  # Anweisungen, die eine Exception hervorrufen können

rescue SyntaxError, NameError => exc_var
  print "String doesn't compile: " + exc_var
rescue StandardError => exc_std_var
  print "Error running script: " + exc_std_var
end

```

Listing 3: Ausnahmebehandlung mit Ruby

2.3 Protokollierung

Bei der Protokollierung wird der Umgang mit der Applikation quasi ‚mitgeschrieben'. Vollständige Logfiles können sehr hilfreich sein, wenn es darum geht, einen Angriff auf die Applikation oder den Server nachzuweisen, bzw. zurückzuverfolgen. Gleichzeitig sind gerade diese Logfiles durchaus gefährdet, weil sie potentiell kritische Daten enthalten können und so Angreifern vielleicht erst den Weg ebnen. Es sollte also penibel genau darauf geachtet werden *was* geloggt wird, nicht nur *dass* geloggt wird.

Rails schreibt sämtliche Ereignisse, Requests an die Applikation und Datenbankanfragen in Logfiles. Dabei wird jeweils ein Logfile für die Development-, die Produktiv- und die Testumgebung geschrieben. Zu finden sind diese Dateien im Verzeichnis der Web-Applikation unter *log\development.log* (als Beispiel für das Logfile der Developmentumgebung).

Rails nimmt jedoch standardmäßig keinerlei Filterungen vor, sodass mancher Eintrag in ein Logfile einer *Rails*-Applikation aussehen könnte wie Listing 4.

```
Processing AccountController#signup (for 127.0.0.1 at 2008-01-13 18:35:35) [POST]
  Session ID: 36e7013cd1eddebaa1416dffc3975c3d
  Parameters: {"user"=>{"password_confirmation"=>"abc123", "login"=>"alice", "password"=>
"abc123", "email"=>"alice@alice123490.ti"}, "commit"=>"Sign up", "action"=>"signup",
"controller"=>"account"}
```

Listing 4: Logfile einer *Rails*-Beispielapplikation

Dieses Logfile zeigt den Request an eine Webseite, auf der sich ein Benutzer (‚alice') registriert hat. Zu der Registrierung gehörte natürlich auch die entsprechende Email-Adresse, sowie das Passwort zum Account. All dies speichert *Rails* standardmäßig als ‚plain text', also weder gefiltert noch verschlüsselt, was natürlich ein Sicherheitsrisiko darstellt.

Die Lösung lautet: Filterung der betreffenden Eingaben im entsprechenden Controller.

Übergreifend kann dies auch im *Application Controller* erfolgen. Eine Filterung geschieht wie in Listing 5 beschrieben.

```
1  class ApplicationController < ActionController::Base
2    filter_parameter_logging "password"
```

Listing 5: Logfile Filterung im ApplicationController

Durch die Anweisung in Zeile 2 werden alle Felder in deren Name der string ‚*password*' vorkommt im Logfile gefiltert. Im Falle des obigen Logfiles wird also sowohl das Feld *password,* als auch das Feld *password_confirmation* gefiltert.

```
1  Processing AccountController#signup (for 127.0.0.1 at 2008-01-13 18:35:35) [POST]
2    Session ID: 36e7013cd1eddebaa1416dffc3975c3d
3    Parameters: {"user"=>{"password_confirmation"=>"[FILTERED]", "login"=>"alice", "password"=>"
   [FILTERED]", "email"=>"alice@alice123490.ti"}, "commit"=>"Sign up", "action"=>"signup",
   "controller"=>"account"}
```

Listing 6: Gefilterte Inhalte im Logfile

Jedoch ist im Bereich Protokollierung in *Rails* weiterhin Vorsicht geboten, denn durch die oben beschriebene Methode ist es nicht möglich, Datenbankanfragen zu filtern.

```
1  [0mINSERT INTO users (`salt`, `updated_at`, `crypted_password`, `remember_token_expires_at`,
2   `remember_token`, `login`, `created_at`, `email`)
3   VALUES('ad1378c43c9856a66850d0276362537ca1c5a1b7', '2008-01-13 18:35:35',
4   '5feafad2e64eeac1e768e74f4f3a80d8d64a3c32', NULL, NULL, 'alice', '2008-01-13 18:35:35',
5   'alice@alice123490.ti')
```

Listing 7: Ungefilterte Datenbankanfragen mit verschlüsselten Werten

Die Lösung hierfür ist, empfindliche Inhalte zu verschlüsseln bevor sie in die Datenbank geraten (und entsprechend gelogt werden). Listing 7 zeigt, dass die Beispielapplikation die Passwörter verschlüsselt und dass trotz Filterung diese verschlüsselten Werte offen im Logfile sichtbar sind.

2.4 Authentifizierung

„There are about over nine thousand ways to do authentication, […]"
- rubyon*Rails* wiki

Der obigen Aussage zum Trotz bietet *Rails* vor Version 2.0 keine eigene Authentifizierungsmöglichkeit. Diese Funktionalität wird bis dahin ausschließlich von Plugins angeboten, die aus der *Rails*-Community stammen. Derer gibt es sehr viele und für die verschiedensten Authentifizierungsmöglichkeiten wie z.B. HTTPD (ab *Rails* Version 2.0 integriert), Ldap, OpenID oder WindowsDomainAuthentication.
Einige oft genutzte Plugins sind recht flexibel in ihrer Arbeitsweise und auch relativ einfach in der Handhabung bzw. Einbindung in die eigene Applikation.

Leider ist es oft nur sehr schwer möglich, Plugins für verschiedene Authentifizierungsmöglichkeiten gleichzeitig in eine einzelne Applikation einzubinden.

Die Plugins *Acts as Authenticated* und *Restful Authentication* sind zwei der bekanntesten Plugins.

3. Szenarien

Im Folgenden werden zwei Szenarien betrachtet, denen Web-Applikationen mit am häufigsten ausgesetzt sind.

- SQL-Injection
- Cross Site Scripting (XSS)

Diese Attacken sind relativ einfach auszuführen und können für ungeschützte Applikationen katastrophale Auswirkungen haben.

3.1 SQL-Injection

Mittels SQL-Injection werden der betroffenen Anwendung schadhafte Datenbankanweisungen ‚injiziert', d.h. durch ungesicherte Eingabemöglichkeiten in die Anwendung gebracht und ausgeführt. Dies kann zur Folge haben, dass benutzerkritische Daten in fremde Hände gelangen, oder gar gelöscht werden könnten. Es könnten auch ‚falsche' Daten in die Datenbank gelangen, wie z.B. gefälschte Accountdaten, die dem Angreifer scheinbar legalen Zugang zur Anwendung ermöglichen.

Der geneigte Leser möge sich zur Verdeutlichung eine Online-Email Anwendung ersinnen. Die folgende Anweisung würde in dieser Anwendung die Emails eines Nutzers mit der ID ‚123' mit einem bestimmten Betreff anzeigen. Wäre die Anwendung ungeschützt könnte man durch geschicktes Wählen des `subject`-Parameters die Anwendung dazu bringen sämtliche Emails aller Nutzer anzuzeigen.

```
Email.find_all "owner_id = 123 AND subject = '#{@params['subject']}'"

   → wähle subject =  „' OR 1 –'"

   → Statement ist immer TRUE, führt zur Anzeige aller Emails aller
     User
```

In *Rails* werden SQL-Anweisungen in der Regel durch das *ActiveRecord* gekapselt, sodass ein gewisser Schutz bereits besteht, falls die methoden `save, find` oder `attributes` verwendet werden. In diesem Falle übernimmt *ActiveRecord* das Escapen der SQL-Anweisung, also das Entfernen von Code aus der Anweisung.

Alternativ kann die `quote`-Methode verwendet werden, wie im folgenden Listing beschrieben.

```
subject = @params['subject']

Email.find_by_sql "SELECT * FROM email WHERE owner_id = 123 AND
subject = #{Email.quote(subject)}"
```

Es muss hier kritisch bemerkt werden, dass gerade *Rails*-Applikationen relativ anfällig auf diese Art Angriffe sind, denn aufgrund des *„Convention over Configuration“*-Konzepts ist die Datenbankstruktur vieler Applikationen schlicht vorgegeben, oder zumindest recht einfach abzuleiten.
Davon abgesehen ist auch die Einfachheit von *Rails* ein Problem. Ist eine Anwendung schnell gebaut, geraten Sicherheitsaspekte womöglich in den Hintergrund, zumal die Enwickler bei jeder (nicht trivialen) SQL-Anweisung doch wieder die `quote`-Methode anfügen müssen, was Mehr- und Handarbeit ist, die wiederum fehleranfällig ist.

3.2 Cross-Site-Scripting (XSS)

Auch XSS-Attacken zielen auf interaktive Applikationen, die oft und viel mit Benutzereingaben zu tun haben (zum Beispiel in Form von Kommentaren in Newssystemen) und entsprechend viele Möglichkeiten dazu bieten. Bei einer solchen Attacke wird fremder Script-Code im Kontext der betroffenen Applikation ausgeführt.

Ein beliebtes Beispiel einer XSS-Attacke ist der Diebstahl von Cookies. Diese werden in *Rails*-Applikationen häufig verwendet, zumal ab Version 2.0 sämtliche Session-Informationen per default als Cookies im Filesystem des Benutzers abgelegt werden. Dabei wird eine kurze Zeile JavaScript-Code ausgeführt, der den Inhalt des Cookies, auf den die Anwendung laut Sicherheitsrichtline des Browsers Zugriff hat, an eine fremde Webseite (die des Angreifers) geschickt.

Im obigen Beispiel einer Newsseite oder eines Blogs hätte ein Benutzer die Möglichkeit einen Kommentar abzugeben. Ein Kommentar ist zumeist recht kurzer Text, der meist unverarbeitet direkt als Content auf der Newsseite oder im Blog angezeigt wird. Dieser Text könnte nun jedoch auch JavaScript Code sein, der direkt durch den Browser des Benutzers interpretiert wird. Grundsätzlich gilt es, dies zu verhindern.

Die Lösung wäre also, den vermeintlichen Code als reinen Text ausgeben zu lassen, oder ganz zu filtern. *Rails* bietet hierfür die `html_escape()`-Methode bzw. die Kurzform `h()`. Diese Methode muss immer dann verwendet werden, wenn Inhalt der von Usern stammt ausgegeben bzw. verwertet wird.

Eine weitere Möglichkeit ist die Methode `TextHelper#sanitize`, die Tags bzw. Scriptbefehle direkt filtert. Ab *Rails* 2.0 ist diese Möglichkeit durchaus empfehlenswert, da sie anhand einer WhiteList arbeitet (pre 2.0 wird eine BlackList verwendet), d.h. der Entwickler kann (muss) explizit angeben, welche Tags bzw. Befehle erlaubt sind (z.B. solche, die der Textformatierung dienen) und welche nicht.

4. Ausblick: *Rails* 2.0

Zum Zeitpunkt dieser Arbeit ist *Rails* aktuell in Version 2.0.2 und hat mit Einführung von *Rails* 2.0 im November 2007 einige Änderungen erfahren. Einige davon sind auch für die Sicherheit von *Rails*-Applikationen interessant und sollen hier aufgezeigt werden.

- HTTP Basic Authentication integriert

  Eine einfache Art der Authentifizierung ist nun im ActionPack von *Rails* integriert und macht damit Authentifizierung auch ohne Plugins möglich. Darüber hinaus ist diese Art recht einfach in der Handhabung.

- Sessions per default nun Cookie-basiert

  Wie bereits erwähnt, werden Session-Informationen nun standardmäßig clientseitig als Cookie abgelegt. Hierbei wird der Inhalt des Cookies anhand eines Schlüssels, der nur dem Server bekannt ist, verschlüsselt. Es wird jedoch vermutet, dass Art und Weise wie dieser Schlüssel erstellt wird bereits gecracked[5] worden ist.

- Die Methode **`Base.protect_from_forgery`** bietet nun die Möglichkeit, einen speziellen Token an alle Serverrequests zu hängen, was sog. Cross Site Reference Forging erschweren/verhindern soll.

5. Fazit

Rails ist ein Webframework, das eher auf Leichtigkeit in der Handhabung und schnelle Ergebnisse zielt, als auf sichere Anwendungen. Dies bedeutet nicht, dass es nicht möglich ist ‚sichere' Anwendungen mit *Rails* zu schreiben, nur wird dem Entwickler von Seiten des Frameworks wenig geholfen, obschon die nötigen Werkzeuge vorhanden sind.

Mitunter unsichere Standardeinstellungen und Konventionen bringen enorme Mehrarbeit mit sich, um die Sicherheit der Applikation zu gewährleisten, womit der eigentliche „Sweet Spot", schnell lauffähige Applikationen zu erstellen, doch ziemlich leidet.

Rails ist dennoch ein aufstrebender Stern am Himmel der Web-Frameworks und seine Sicherheitsaspekte werden von einer wachsamen Community kritisch betrachtet und bearbeitet. Eine Vielzahl an Blogs, Plugins und Boards macht es dem sicherheitsorientierten Entwickler leicht, die nötigen Informationen zu finden, die nötig sind um die eigene Anwendung sicherer zu gestalten.

[5] http://izumi.plan99.net/blog/index.php/2007/11/25/*Rails*-20-cookie-session-store-and-security/

Quellen

- why's (poignant) guide to ruby – http://qa.poignantguide.net
- www.ruby-lang.org
- www.rubyon*Rails*.com
- Wikipedia
- www.ruby-learning.com
- Rolling with Ruby on *Rails* Revisited – www.onlamp.com
- Acts as Authenticated Plugin
  - http://technoweenie.stikipad.com/plugins/show/Acts+as+Authenticated
- http://home.vrweb.de/~juergen.katins/ruby/buch/tut_exceptions.html
- http://*Rails*casts.com
- http://izumi.plan99.net/blog/index.php/2007/11/25/*Rails*-20-cookie-session-store-and-security/