Lời nói đầu

Trong xu thế công nghệ thông tin đang trên đà phát triển vô cùng mạnh mẽ. Thông tin được trao đổi qua khắp thế giới, trong đó website là một trong những công nghệ ngày càng rất phát triển. Theo khảo sát của công ty kiểm soát thị trường Netcraft. Thì tính đến tháng 4 năm 2008, thì đã có tới 165.719.150 websites trên toàn thế giới. Như vậy chỉ trong vòng 4 tháng, đã tăng tới 10 triệu các websites. Bên cạnh đó, công nghệ web cũng đang dần thay đổi, xu hướng web 2.0 đang dần chiếm vị trí. Tốc độ cũng như đòi hỏi Điều này đòi hỏi cần có những framework mới đáp ứng được nhu cầu về công nghệ cũng như đáp ứng được xã hội thông tin.

Đã có rất nhiều các framework ra đời để hỗ trợ việc phát triển ứng dụng web như Zend, Cake cho ngôn ngữ PHP, Struts và Strping cho Java... Nhưng trong vài năm gần đây, một framework mới ra đời đã nhanh chóng nổi lên và cạnh tranh lại với các framework trên và đôi khi còn có những đánh giá là vượt trội hơn, đó là Ruby on Rails.

Hôm nay, em muốn giới thiệu cùng với các thầy và các bạn về Ruby on Rails. Chủ yếu mục tiêu của luận văn này là nêu cố gắng trình bày tổng quan về Ruby on Rails và những điều được cho là tâm đắc của framework này. Từ đó giúp chúng ta hiểu và có thể có những đánh giá tốt nhất, khách quan nhất về framework Ruby on Rails.

Cùng với sự giúp đỡ của thầy giáo, Tiến Sĩ Nguyễn Hải Châu, em đã tìm hiểu, nghiên cứu về Ruby on Rails và trình bầy trong khóa luận này những vấn đề sau:

Chương 1: Giới thiệu tổng quan một số các framework hiện nay, làm một số phép so sánh các framework với nhau.

Chương 2: Giới thiệu tổng quan về Ruby on Rails, lịch sử ra đời, kiến trúc và triết lý của nó.

Chương 3: Giới thiệu các thành phần quan trọng tạo nên sức mạnh của Ruby on Rails.

Chương 4: Trình bày mô tả lại dự án website bán hàng (BookShop) mà em đã viết dựa trên framework Ruby on Rails.

Chương 1
Tổng quan các framework

Các frameworks ra đời nhằm mục đích hỗ trợ cho các nhà phát triển triển khai và bảo trì ứng dụng được dễ dàng hơn. Ruby on Rails là một web framework, tuy nó ra đời sau, nhưng nó vẫn có những đặc điểm giống và khác, điểm mạnh và yếu hơn so với các web framework khác. Bởi vậy, trong chương sẽ giới thiệu tới các bạn một bức tranh cơ bản về framework.

1.1. Khái niệm framework

Framework là một thư viện code, được thiết kế để giúp đỡ cho việc phát triển phần mềm. Trong đó những chi tiết ở mức độ thấp khi tạo ra một ứng dụng sẽ được framework định nghĩa sẵn trong các gói thư viện của mình, và nó dễ đàng được sử dụng khi cần. Điều này giúp cho một nhà phát triển tích kiệm được thời gian, công sức trong việc giải quyết các vấn đề liên quan đến ứng dụng.

Một framework nói chung thường bao gồm hai phần: “frozen spots” và “hot spots”. Trong đó “frozen spots” sẽ xác định kiến trúc tổng thể của hệ thống (Hay nó sẽ xác định các thành phần cơ bản và mối quan hệ giữa các thành phần đó). Còn “hot spots” sẽ trình bày các phần của kiến trúc trên, nơi mà các nhà lập trình sử dụng framework để thêm vào code của họ, từ đó thêm các chức năng cho dự án.

Có rất nhiều framework, nhưng một framework tốt sẽ giúp cho các nhà phát triển tích kiệm được nhiều thời gian và sự tập trung hơn vào giải quyết các vấn đề mang tính chất logic thay vì phải tập trung hơn là việc phải chú trọng vào code. Một số framework sẽ giới hạn các lựa chọn trong quá trình phát triển để mà làm tăng tính hiệu quả.

Framework ko chỉ giúp cho việc phát triển một ứng dụng cụ thể nào, có nhiều kiểu ứng dụng framework khác nhau, đó có thể là framework cho việc xây dựng biên dịch các ngôn ngữ lập trình, framework cho dịch vụ đa phương tiện(multimedia), hay framework cho phát triển ứng dụng web.

Một framework là một phần mềm được viết bởi ngôn ngữ lập trình, bởi vậy nó cũng tuân theo quy tắc vòng đời của một phần mềm. Tức là luôn có những framework mới được sinh ra và cũng có những cái phải chết đi. Số phận của mỗi framework còn tùy thuộc vào từng giai đoạn yêu cầu cũng như xu thế mới trong việc phát triển công nghệ. Ruby on Rails là một web framework, nó ra đời cho việc đáp ứng lại nhu cầu của công web 2.0.

1.2. Giới thiệu các web framework

Hiện nay có rất nhiều web framework khác nhau nhằm hỗ trợ cho các nhà lập trình web. Ví dụ Java có Spring , Apache Struts, Tapsetry, ASP.NET có ASP.NET MVC Framework, DotNetNuke, PHP có CakePHP, Zend, và Ruby có Ruby on Rails.

Ngoài mục đính chung của các framework là giúp các nhà phát triển ứng dụng web dễ dàng phát triển, bảo trì hơn. Các framework còn một số điểm chung khác. Như về mô hình kiến trúc, đa số các framework là sử dụng mô hình ba lớp (Presentation, Business-Logic, Data Access) hay mô hình Model-View-Controller(MVC). Như framework Spring sử dụng mô hình MVC và Ruby on Rails cũng vậy. Còn ASP.NET và CakePHP sử dụng mô hình ba lớp. Nói chung cả hai mô hình này đều nhằm mục đích là phân tách giữa các phần: hiển thị, phần xử lý logic và truy vấn dữ liệu.

Các web framework sẽ cung cấp cho các nhà lập trình những chức năng chung sau:

- Trừu tượng cơ sở dữ liệu. Ví dụ như đảm bảo rằng các truy vấn làm việc tốt mà không quan tâm đến cơ sở dữ liệu được đặt trong MySQL, MS SQL Server, Oracle hay một hệ quản trị cơ sở dữ liệu nào đó.
- Cung cấp các khuôn mẫu template.
- Quản lý session người dùng hay tạo dựng các URL.

1.2.1. Đánh giá các framework

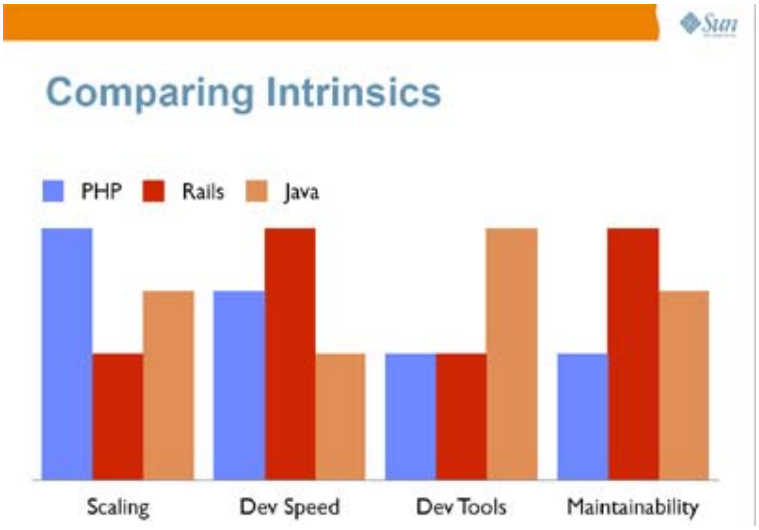
Cách đây không lâu, Java đã nổi lên như một hiện tượng với khẩu hiệu *"Viết một lần, chạy khắp nơi"*, và với công nghệ Applet nổi tiếng phục vụ cho các ứng dụng web, và chính điều này đã làm cho Java trở nên nổi tiếng cùng với sự phát triển như vũ bão của mạng lưới thông tin toàn cầu, Internet. Tuy nhiên, Java cùng với sự phát triển của nó đã trở nên quá rườm rà, và cồng kềnh. Trên tạp chí BusinessWeek có đăng tải bài **"Java? It's so nineties"**, tạm dịch là, "Java ư? Sao mà quê thế!" Bài báo này đã làm dấy lên một cuộc tranh luận gay gắt giữa các Java-fans và những người yêu chuộc sự đổi mới. Và phe đổi mới có đưa ra kiến trúc Ruby on Rails để so sánh với Java, và qua đó để khẳng định nhận xét của mình. Vậy sự thật Ruby on Rails có đúng vậy không, tôi không thể khẳng định vì sự so sánh các framework với nhau luôn có những điểm hạn chế. Tuy nhiên, sau đây tôi sẽ cố gắng nêu ra những nhận xét mang tính chất khách quan, cũng như cố gắng nêu

bật những ưu điểm mà Ruby on Rails đem lại cho các nhà phát triển ứng dụng web. Từ đó các bạn có thể tự rút ra nhận xét riêng cho mình.

Điều đầu tiên khi tôi chuyển từ Java sang Ruby on Rails, đó là tôi cảm nhận được sự đơn giản trong quá trình phát triển. Trong khi đó, theo tôi cũng như nhiều người nhận xét cho rằng Java là một ngôn ngữ khá là cồng kềnh, đặc biệt trong việc cấu hình. Điều này thì Ruby on Rails vượt trội, bởi triết lý mà Rails đem lại luôn là cấu hình một cách đơn giản và ít nhất.

Trong khi đó php ra đời là một ngôn ngữ lập trình web, đó là một dạng của mã nguồn mở và được sử dụng nhiều nhất, tuy nhiên nó được đánh giá là chỉ thích hợp nhất cho việc phát triển các website vừa và nhỏ. Còn Ruby on Rails được xem như là công nghệ web tương lai. Một điểm rất mạnh nữa mà Ruby on Rails đem lại cho người lập trình đó là tốc độ phát triển. Với Ruby on Rails việc phát triển ứng dụng web trở nên vô cùng nhanh chóng.

Thông thường để so sánh đánh giá một framework, người ta thường căn cứ vào bốn yếu tố chính sau: tốc độ phát triển, khả năng bảo trì, công cụ hỗ trợ, khả năng thay đổi mở rộng. Sau đây xin trích dẫn tài liệu của Tim Bray, giám đốc hãng công nghệ web (Web Technologies at Sun Microsystems). Dựa trên các nghiên cứu đánh giá, anh ấy đã đưa ra biểu đồ so sánh sau:



Như theo lược đồ đánh giá trên, sẽ thấy cả 3 frameworks trên có những điểm mạnh và yếu riêng. Về tốc độ phát triển và khả năng bảo trì thì Rails đứng ở vị trí đầu tiên. Vậy theo các bạn thì yếu tố nào là quan trọng nhất: khả năng mở rộng, tốc độ, công cụ, hay bảo trì! Theo như Tim Bray cũng như nhiều nhà chuyên môn thì tính bảo trì(Maintainability) là quan trọng nhất, nguyên nhân bởi vì một ứng dụng khi được xây dựng, thì đa số là nó sẽ được sử dụng trong một thời gian dài. Điều

này đồng nghĩa với việc trong suốt thời gian đó nó phải được bảo trì. Vấn đề là những cái gì làm ảnh hưởng đến vấn đề bảo trì, tôi cho rằng đó là kiến trúc, do khả năng dễ hiểu của code, và độ dài của code. Đây cũng chính là những thành phần đem lại sức mạnh cho Ruby on Rails với kiến trúc mô hình MVC, với ngôn ngữ linh hoạt Ruby. Tất cả điều này sẽ được giới thiệu với các bạn trong các chương sau.

Chương 2
Framework Rails

Như đã giới thiệu ở chương một, một web framework ra đời nhằm hỗ trợ các nhà phát triển trong việc triển khai và bảo trì ứng dụng web. Tuy nhiên sức mạnh của mỗi framework sẽ khác nhau do kiến trúc và ngôn ngữ mà nó dùng là khác nhau. Ruby on Rails tuy mới ra đời nhưng nó luôn được đánh giá là một framework mạnh cũng nhờ kiến trúc và ngôn ngữ mà nó dùng. Sau đây xin giới thiệu tới các bạn chi tiết về framework này, lịch sử và kiến trúc mà nó sử dụng, những ưu điểm, sức mạnh mà Rails sẽ mang tới cho một nhà phát triển ứng dụng web.

2.1. Giới thiệu về Ruby on Rails(Rails)

2.1.1. Nguồn gốc của Rails

Đầu tiên, tôi muốn giới thiệu với các bạn nguồn gốc hình thành framework Ruby on Rails. Rails ra mắt công chúng lần đầu tiên vào năm 2004, Rails thoạt đầu được dùng như là nền tảng cho một công cụ quản lý dự án được đặt tên là Basecamp và được tạo ra bởi nhà phát triển web David Heinemeier Hansson, một nhân viên của công ty phát triển web 37signals (Mỹ). Ban đầu họ xây dựng Rails không phải với mục đích là xây dựng ra một framework riêng, chủ tâm ban đầu là dùng nó để xây dựng các ứng dụng khác của 37signals. Sau đó Heinemeier Hansson thấy tiềm năng của nó giúp cho anh ấy làm các công việc dễ dàng hơn bằng cách rút ra các tính năng phổ biến như trừu tượng cơ sở dữ liệu và khuôn mẫu(template) bên trong, và sau đó nó trở thành phiên bản đầu tiên được tung ra của Ruby on Rails.

Trên thực tế Rails được rút trích từ Basecamp, một chương trình mà được cộng đồng Rails cho là trong quá trình làm chương trình đó thì đã gặp và giải quyết được rất nhiều các vấn đề phức tạp. Và Rails, một framework bắt nguồn từ chương trình đó thì nó đã thừa kế được những sức mạnh từ dự án đó. Vì vậy Rails luôn sẵn sàng có thể giải quyết được các vấn đề thực tế tồn tại trong quá trình phát triển web. Ngoài ra, ban đầu Rails được xây dựng ra còn với mục đích là để xây dựng các ứng dụng khác từ nó, bởi vậy Rails trước khi xuất hiện trên thế giới thì nó đã chứng minh được bản thân nó là một framework rất hữu dụng, chặt chẽ và toàn diện.

Sau khi phiên bản đầu tiên được tung ra thì cộng đồng Rails cũng đã đóng góp bổ sung hàng mở rộng nó, và sửa các lỗi được tìm thấy. Và phiên bản mới nhất của Rails bây giờ là phiên bản 2.0.2.

2.1.2. Triết lý của Rails

Rails ra đời cũng như một số framework khác, nó cũng có triết lý riêng của mình. Bạn cũng có thể phần nào thấy sự khác biệt của Rails thông qua triết lý của nó. Triết lý của Rails được định hướng bằng cặp khái niệm:

"DRY and convention over configuration" có thể dịch là *đừng lặp lại chính mình* và *sự ngầm định thay cho cấu hình*.

1. DRY

DRY viết tắt từ "Don't Repeate Yourself", tức là mỗi phần được biết trong hệ thống nên sẽ được diễn tả chỉ ở một nơi duy nhất hay hiểu đơn giản là các đoạn code, đoạn hàm chức năng, các định nghĩa cơ sở dữ liệu... sẽ được đặt tại đâu đó trong dự án, và thật dễ dàng, từ mọi vị trí trong dự án bạn có thể triệu gọi nó ra và sử dụng. Rails sử dụng sức mạnh của Ruby để mang lại điều đó. Khi bạn quyết định thay đổi hoạt động của một ứng dụng mà dựa trên nguyên tắc DRY, bạn không cần sửa đổi code của ứng dụng nhiều hơn một vị trí quan trọng. Trong khi điều này có lẽ là một điều phức tạp trước đây, đối với rails nó thực sự đơn giản. Ví dụ, thay vì copy và dán đoạn code với một tính năng tương đương nào đó, bạn phát triển ứng dụng web của bạn theo cách là cho hàm chức năng này được lưu giữ một lần, tại một vị trí trung tâm, và sẽ được chuyển đến nơi mà cần sử dụng nó trong mỗi phần của ứng dụng. Và bằng cách này, nếu hành động gốc cần thay đổi, bạn chỉ cần sửa đổi nó một lần tại một vị trí, thay cho việc phải sửa lại tại các nơi khác nhau trong ứng dụng.

Một ví dụ mà Rails hỗ trợ nguyên tắc DRY, không giống như Java, nó không ép bạn lặp việc định nghĩa giản đồ cơ sở dữ liệu trong ứng dụng. Rails thì coi dữ liệu của bạn là một nguồn thông tin cho việc lưu trữ dữ liệu, và nó yêu cầu rõ ràng thông tin về cơ sở dữ liệu, điều này đảm bảo cho nó hành động đúng với dữ liệu.

Rails cũng bám chặt nguyên tắc DRY khi nó thực thiện kỹ thuật như Ajax. Các nhà phát triển thường thây việc lặp lại code trong khi tạo ứng dụng với Ajax. Xét cho cùng các website cũng nên hoạt động trên trình duyệt mà không hỗ trợ Ajax, và code đòi hỏi hiển thị tới cả hai kiểu trình duyêt. Rails làm việc này một cách dễ dàng để xem mỗi trình duyệt sinh ra một cách thích hợp mà không cần lặp lại bất kỳ code nào.

2. Sự ngầm định thay cho cấu hình(Convention over configuration)

Khái niệm "sự ngầm định thay cấu hình" để cập tới thực tế rằng Rails thừa nhận một số thứ mặc định khi xây dựng một ứng dụng web điển hình. Không giống như một số framework khác, yêu cầu bạn phải từng bước cấu hình để xử lý trước khi cho bạn chạy một ứng dụng dù là đơn giản nhất. Các thông tin cấu hình đó thường được lưu giữ trong một file XML, và các file đó ngày một lớn lên và rất phiền phức cho công việc bảo trì, cụ thể bạn có thể thấy điều này rất rõ ở Java. Trong nhiều trường hợp, bạn bị ép phải lặp lại toàn bộ việc cấu hình khi bắt đầu một dự án mới. Trong khi đó, Rails ban đầu được triết xuất ra từ một ứng dụng có sẵn, kiến trúc của nó là quá đủ để làm việc bên trong các khung sườn framework về sau. Heinemeier Hansson chủ tâm tạo ra Rails theo một cách mà nó không cần quá tốn sức cho việc cấu hình, miễn là tuân theo một chuẩn ngầm định. Kết quả là nó không yêu cầu một file cấu hình dài dòng nào cả. Trên thực tế, nếu bạn không có nhu cầu thay đổi những chuẩn ngầm định này, Rails thực sự chỉ cần một file cấu hình ngắn và đơn giản để mà bạn chạy ứng dụng của bạn. File đó mục đích là tạo kết nối tới cơ sở dữ liệu.

Khi làm việc với Raills, bạn sẽ thấy rất rõ sự ngầm định của Rails được thể hiện ở rất nhiều trường hợp. Ví dụ ngay khi tạo một dự án với Rails. Rails đã mặc định tạo ra một dự án theo mô hình MVC. Hay đơn giản là khi đặt tên một mô hình dữ liệu(Model), thì Rails cũng sẽ ngầm định tên của Model đấy là một chữ in hoa, số ít và bảng dữ liệu mà nó liên kết là một chữ thường số nhiều. Ví dụ Model là Book và tên bảng dữ liệu sẽ được nghĩ là books.

Việc ngầm định này ban đầu có thể sẽ khiến người lập trình cảm thấy hơi khó chịu vì Rails sẽ không khuyến khích bạn đặt tên tùy ý. Tuy nhiên khi quen nó, bạn sẽ thấy nó rất thú vị, vì giúp bạn tập trung vào viết code, cũng như giúp bạn đặt tên theo chuẩn điều này giúp ích khi bảo trì hay đọc lại code sẽ giúp bạn dễ dàng hiểu và nhận ra mỗi file là gì. Ngoài ra cấu hình ngầm định còn giúp người lập trình tích kiệm khá nhiều thời gian cho việc cấu hình.

2.1.3. Tự động sinh code trong Rails

Ta cũng có thể nói đến Rails với việc sinh code tự động. Khi bạn phát triển trên Rails, có các thư mục hay file là nơi để bạn điền code vào, và tất cả các phần trong ứng dụng của bạn tương tác với nhau trên một cách thức chuẩn. Hay nó đã chuẩn bị sẵn một bộ khung chương trình trước cho bạn, điều này giúp bạn đơn giản là chỉ việc điền các đoạn code login vào trong ứng dụng và tích kiệm và giảm tải công việc của bạn. Và nhờ có bộ khung đã định nghĩa sẵn này, Rails cung cấp các generate, rake, script cho phép bạn nhanh chóng tạo, xóa ra một số template để bạn định nghĩa model, view ,controller, hay ngay cả database trong cơ sở dữ liệu.

2.1.4. Vòng đời phát triển ứng dụng trong Rails

Rails động viên sử dụng các điều kiện khác nhau cho mỗi giai đoạn trong chu trình vòng đời của ứng dụng(phát triển,kiểm tra, và sản phẩm). Nếu bạn đã hoàn thành việc phát triển ứng dụng web trong một khoảng thời gian, đây có lẽ là cách để bạn thực hiện. Rails chỉ hợp thức hóa các môi trường này.

development trong điều kiện development, thay đổi tới code nguồn của ứng dụng ngay lập tức; tất cả chúng ta cần làm là load lại trang tương ứng trong trình duyệt. Tốc độ không phải là yếu tố quyết định trong điều kiện này; Thay vì vậy, mục tiêu là cung cấp cho người phát triển nhìn xuyết suốt các thành phần có liên quan đến việc hiển thị trên mỗi trang web. Khi một lỗi xảy ra trong điều kiện phát triển, các nhà phát triển có thể nhìn thoáng qua để biết dòng code nào gây ra lỗi, và như thế nào dòng đó được gọi tới. Khả năng này được cung cấp bởi **stack trace**(bao gồm đầy đủ các danh sách các phương thức gọi để chuẩn bị cho lỗi), cái này được hiển thị khi một lỗi không trông đợi xảy ra.

test Trong điều kiện kiểm tra, chúng ta thường làm mới(refresh) cơ sở dữ liệu với một ranh giới của dữ liệu giả cho mỗi lần kiểm tra được lặp lại-điều này đảm bảo rằng kết quả của cuộc kiểm tra là nhất quán, và cách xử lý đó là có khả năng tăng thêm. Unit và các thủ tục kiểm tra là tự động hóa hoàn toàn trong Rails. Khi chúng ta kiểm tra ứng dụng Rails, chúng ta không xem nó sử dụng một trình duyệt web truyền thống. Thay vì vậy, các cuộc kiểm tra được gọi từ các dòng lệnh, và có thể chạy dưới background. Điều kiện kiểm tra cung cấp một môi trường chuyên dụng để xử lý các hoạt động đem lại hiệu quả.

production Giai đoạn mà ứng dụng của bạn đã hoàn thành, khi mà các kết quả kiểm tra là tốt, các lỗi đã được loại ra. Khi đó việc thay đổi code là rất hiếm, điều này có nghĩa là môi trường sản phẩm có thể lạc quan để tập trung vào việc sử dụng. Nhiệm vụ như viết số lượng lớn các log cho mục đích gỡ rồi là trở nên không cần thiết khi này. Tuy nhiên, nếu một lỗi xảy ra, bạn không muốn làm người dùng sợ hãi với những thông báo dấu vết khó hiểu(nó chỉ tốt trong giai đoạn phát triển)

Với các nhu cầu khác nhau cho mỗi điều kiện,giai đoạn, Rails lưu trữ dữ liệu cho mỗi điều kiện trong các dữ liệu được tách biệt hoàn toàn. Bởi vậy, tại bất kỳ thời điểm nào, bạn có thể có:

- Dữ liệu trực tiếp với người dùng thực sự tương tác trong môi trường điều kiện sản phẩm.
- Một bản copy của dữ liệu trực tiếp mà bạn sử dụng để gở rối lỗi hay phát triển tính năng mới trong môi trường điều kiện phát triển.

- Một bộ dữ liệu kiểm tra để mà bạn có thể liên tục làm việc load lại trong môi trường điều kiện kiểm tra.

2.2. Kiến trúc Model-View-Controller

Kiến trúc mà Rails sử dụng là kiến trúc Model-View-Controller(MVC), một kiến trúc mà có rất nhiều các framework đang sử dụng. Tuy nhiên, Rails có điểm khác biết của nó.

2.2.1. Mô hình Models, Views, and Controllers(MVC)

Năm 1979, Trygve Reenskaug mang đến một kiến trúc mới cho việc tương tác và phát triển ứng dụng. Trong thiết kế của anh ấy, ứng dụng được chia ra làm ba dạng thành phần: model, view, và controller. Mô hình này ban đầu chủ yếu được sử dụng cho các dự án ứng dụng mà không được các nhà phát triển web khai thác. Mãi 20 năm sau khi nó ra đời, ý tưởng sử dụng mô hình này trong việc phát triển web mới được khai thác. Kết quả là ra đời các framework như WebObjects, Struts, JavaServer Faces và Rails.

MVC ra đời nhằm phần chia các thành phần có sự liên quan với nhau, và khi cần thì dễ dàng móc nối lại. Từ đó làm cho việc viết code và bảo trì dế dàng hơn.

- Model là phần chủ yếu chú trọng tới dữ liệu. Nó cho phép bạn đặt ra các quy tắc với dữ liệu hay quy tắc trong kinh doanh để mà thực hiện với dữ liệu. Ví dụ, có một trường dữ liệu là giá bán, yêu cầu của trường này là mức giá tối thiểu khi bán không thể nhỏ hơn 100 nghìn đồng chẳng hạn, khi này Model sẽ tạo ra một ràng buộc(trigger) không cho phép sửa đối hay nhập dữ liệu nhỏ hơn điều kiện này. Điều này tạo nên cảm giác, bằng cách bổ sung thêm các quy tắc vào model, chúng ta có thể đảm bảo rằng không có gì khác trong ứng dụng có thể làm sai lệch dữ liệu. Model hành động như một người bảo vệ và một nơi để lưu trữ dữ liệu.

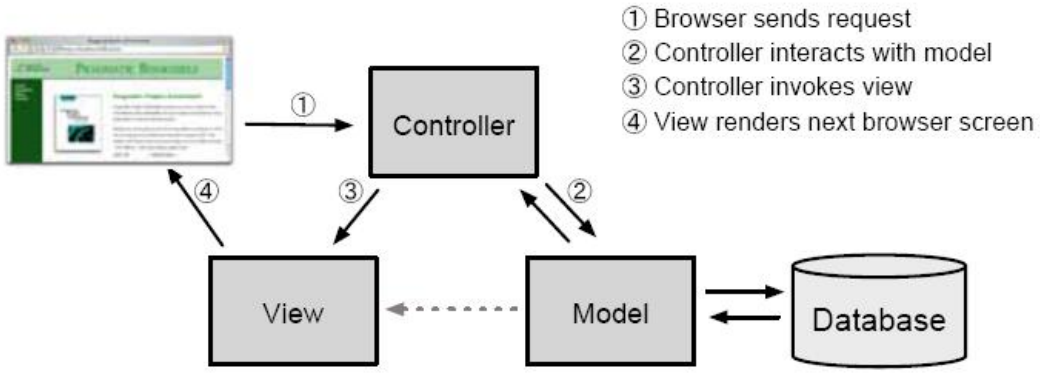
- View là phần chịu trách nhiệm để mà sinh ra giao diện người dùng, thông thường được dựa trên dữ liệu trong model. Ví dụ, một kho dữ liệu được lưu trữ trên mạng và có một danh sách các mặt hàng được hiển thị ra trên trình duyệt. Danh sách này được truy xuất thông qua Model tuy nhiên View sẽ chịu tránh nhiệm lấy danh sách ra từ Model, và định dạng cho người dùng.

- Controllers chịu tránh nhiệm phối hợp ứng dụng. Controller nhận sự kiện từ bên ngoài(thường là trình duyệt), tương tác với Model, và hiện thị một cách thích hợp với người xem.

Việc chia một ứng dụng phần mềm ra làm ba thành phần là sẽ rất có lợi, bạn có thể thấy vì một số lý do sau đây:

- Nó làm tăng khả năng phát triển mở rộng của ứng dụng: ví dụ nếu ứng dụng của bạn khi chạy có các vấn đề về cơ sở dữ liệu truy xuất chậm, bạn có thể nâng cấp phần cứng đang chạy cơ sở dữ liệu mà không tác động tới các thành phần khác.
- Nó khiến cho việc bảo trì dễ dàng hơn: vì các thành phần phụ thuộc thấp với nhau, vì vậy khi thay đổi một cái thì không ảnh hưởng tới cái khác
- Nó tăng khả năng sử dụng lại: một Model có thể được sử dụng lại cho nhiều phần hiển thị và ngược lại cũng vậy.
- Nó khiến ứng dụng có thể phân tán: sự chia rẽ tách biệt về code giữa các thành phần có ý nghĩa là mỗi thành phần trong chúng có thể ở trên một máy tính riêng, nếu thấy cần thiết.

Hình sau mô tả trừu tượng hoạt động của mô hình gồm ba phần chính MVC.



Kiến trúc Model – View – Controller

Rails cố gắng kết cấu cho ứng dụng của bạn theo mô hình phát triển model, view, controller và chia ra các chức năng riêng. Nó kết nối chúng lại với nhau khi

chương trình của bạn thực hiện. Một trong những điều mà tôi khi học về Rails cảm thấy rất thú vị, đó là việc liên kết trong Rails trở nên rất đơn giản và bạn không cần phải cấu hình gì cả. Điều này Rails thực hiện được chính là dựa trên triết lý của nó "*sự ngầm định thay cho cấu hình*".

Cách đặt tên của model, view, controller là đều theo một quy ước chung. Và nhờ có vậy mà bộ ba trên được liên kết với nhau một cách dễ dàng. Người lập trình không phải lo lắng về sự liện kết này.

2.2.2. Phân chia các modun

Mỗi thành phần của mô hình model-view-controller được đặt trong thư mục con của thư mục app với cái tên tương ứng là models, views, controllers.

Sự phân chia trong rails còn tiếp tục trong phạm vi code mà bao gồm bên trong cả bản thân framework. Các lớp tạo thành từ các hàm chức năng của Rails cũng có thể chia thành ba modun sau:

1. ActiveRecord

ActiveRecord là modun cho việc xử lý các thông tin dữ liệu và các thao tác logic trên dữ liệu. Nó đóng vai trò của Model trong kiến trúc MVC. Mọi đối tượng Model trong Rails đều thừa kế từ nó. Sau khi thừa kế nó, các model đó đóng vai trò như các đối tượng mà chứa dữ liệu trong các bảng tương ứng trong cơ sở dữ liệu. Và đối tượng đó có thể có các phương thức mà nó thừa kế từ ActiveRecord như xem, xen, xóa, sửa, tìm kiếm... như thao tác trực tiếp trên bảng dữ liệu.

```
class Book < ActiveRecord::Base
end
```

Tóm lại ActiveRecord được thiết kế để xử lý tất cả các nhiệm vụ của một ứng dụng mà quan hệ với cơ sở dữ liệu, bao gồm:

- Việc thiết lập một kết nối tới server chứa cơ sở dữ liệu
- Nhận dữ liệu từ bảng
- Lưu giữ dữ liệu mới vào cơ sở dữ liệu

ActiveRecord cho phép một lượng lớn các adapter cơ sở dữ liệu để kết nối tới các gói server dữ liệu khác nhau, như MySql, PostgreSQL, MySqlite, Oracle, và Microsoft SQL Server. Chúng ta sẽ có thể hiểu thêm về ActiveRecord trong phần sau: "Trừu tượng hóa cơ sở dữ liệu với Active Record".

2. ActionController

ActionController là thành phần xử lý logic yêu cầu trình duyệt và khả năng thông tin giữa Model và View. Các controller của bạn đều sẽ thừa kế từ lớp này. Nó được tạo từ thành phần của thư viện ActionPack(ActionPack là tên của thư viện bao gồm ActionController and ActionView trong kiến trúc MVC). Có thể chia ra chức năng của nó ra làm ba chức năng chính sau:

- Quyết định xử lý một yêu cầu cụ thể, ví dụ có hay không việc xuất ra đầy đủ toàn bộ trang web hay chỉ một phần của trang, hoặc có thể là kiểm tra quyền truy xuất của yều của người dùng tới trang web đó.
- Nhận dữ liệu từ model và gửi nó tới view. Model đóng vai trò như một đối tượng chứa dữ liệu trong cơ sở dữ liệu.
- Tập hợp thông tin yêu cầu từ trình duyệt và sử dụng nó để tạo hay sửa dữ liệu trong model.

  Một controller sẽ được thừa kế và xây dựng các hành động(action) trông như sau:

```
class StoryController < ActionController::Base
#xử lý hiện thị
def index
end
#xử lý hiện thị dữ liệu cho
def show
end
end
```

Nếu để ý, chúng ta một lần nữa lại thấy Rails thực hiện triết lý về việc ngầm định cấu hình của mình. Đó là cách đặt tên lớp, nó bắt đầu bằng một chứ in hoa và không có khoảng trống giữa các từ. Còn về tên file thì được đặt là chữ thường và có dấu gạch dưới mỗi từ. Ví dụ lớp StoryController sẽ có tên file tương ứng là story_controller.rb và được đặt trong thư mục app/controllers. Trong đặt tên file cho model cũng đều được Rails tuân thủ theo những quy tắc tương tự như vậy. Kết quả của sự thống nhất cách đặt các tên file giúp cho sự liên kết mô hình MVC trở nên dễ dàng và tự nhiên.

3. ActionView

ActionView::Base là lớp cha cho tất cả các views, sự trừu tượng một views được xử lý hoàn toàn bởi ActionView::Base.

Như đã nói ở trên, quy tắc trong mô hình MVC là view chỉ nên bao gồm phần trình bày về mặt logic. Điều này có nghĩa là code trong một view chỉ nên thực hiện các hành động có quan hệ về mặt hiện thị trong ứng dụng, không code nào trong view là nên thực hiện mặt logic phức tạp của ứng dung, và cũng không nên lưu trữ hay nhận dữ liệu từ cơ sở dữ liệu. Trong Rails, mọi thứ được gửi từ trình duyệt web được xử lý bởi một view. Các file hiện thị được lưu trong thư mục aap/view của ứng dụng. Một trang view có thể đơn giản là một trang HTML, nhưng nó cũng có thể là một trang mà được kết hợp giữa HTML và code của Ruby, tạo thành một trang web động và có đuôi cú pháp là ERb(Embedded Ruby).

Cũng giống như một trang web JSP hay PHP, nó cũng cho phép vừa gõ các thẻ HTML và vừa gõ các câu lệnh trong các thẻ đặc biệt.

Ví dụ PHP có đoạn code là:

```
<strong><?php echo 'Hello World from PHP!' ?></strong>
```

Thì Ruby cũng có cách viết gần tương đương:

```
<strong><%= 'Hello World from Ruby!' %></strong>
```

Chương 3
Những ưu điểm nổi bật của Rails

Như đã giới thiệu ở trên, Rails có rất nhiều ưu điểm giúp cho các nhà lập trình nhanh chóng phát triển một ứng dụng web. Giúp cho việc bảo trì ứng dụng cũng trở nên dễ dàng hơn. Để làm được điều đó là sự hợp nhất của một tập hợp nhiều thành phần nhỏ trong kiến trúc cũng như là ngôn ngữ mà Rails sử dụng. Điều này cũng có thể thấy ngay trong chương 2, khi tôi giới thiệu về triết lý của Rails.

Với triết lý "sự ngầm định thay cho cấu hình" cũng đã góp phần nhỏ cho tốc độ phát triển ứng dụng Rails.

Trong chương này, tôi xin phép giới thiệu một số thành phần khác đã đóng góp vào sự thành công của framework Ruby on Rails. Đó là:

- Sự linh hoạt của ngôn ngữ Ruby mà Rails sử dụng.
- Sự trừu tượng hóa cơ sở dữ liệu trong Rails.
- Sự hỗ trợ rất lớn khi phát triển trang web với Ajax.

3.1. Giới Thiệu Về Ngôn ngữ Ruby

Đầu tiên xin giới thiệu với các bạn về ngôn ngữ Ruby, một phần không thể thiếu trong framework Rails.

Thay vì định nghĩa về Ruby tôi xin trích nguyên văn lời giới thiệu về Ruby của cộng đồng Ruby như sau: "Ruby là một ngôn lập trình mã nguồn mở, linh hoạt, với một sự nổi bật về sự đơn giản dễ dùng và hữu ích. Nó có cú pháp "tao nhã" và tự nhiên dễ đọc và dễ dàng để viết".

Và theo như tác giả của ngôn ngữ Ruby, anh Yukihiro "matz" Matsumoto người Nhật, thì tác giả là một người rất ưa thích các ngôn ngữ như: Perl, Smalltalk, Eiffel, Ada, và Lisp. Tác giả đã cố gắng kết hợp một cách thận trọng các ngôn ngữ này với nhau từ đó tạo ra Ruby. Có lẽ chính vì vậy mà Ruby là một ngôn ngữ với các câu lệnh và cú pháp khá quen thuộc khi được học.

Một vài đặc điểm nhận thấy ở ngôn ngữ Ruby là:

- Ruby có cú pháp ngắn nhưng có thể làm được nhiều việc.
- Ruby là ngôn ngữ rất hướng đối tượng.

3.1.1 Xem mọi thứ như một đối tượng

Với Ruby, mọi thứ đều là một đối tượng. Mọi bit thông tin hay code đều có thể được đưa ra cho chính chúng việc sở hữu các thuộc tính hay hành động. Trong lập trình hướng đối tượng thì các thuộc tính gọi là các biến thực thể (instance variables) và các hành động gọi là các phương thức. Ruby là một hướng đối tượng "tinh khiết", điều này có thể thấy bằng một bit của code mà thực hiện một hành động với một con số. Số 5 cũng được coi là một đối tượng và có phương thức làm times. Đoạn code dưới sẽ in ra 5 lần dòng "We *love* Ruby":

```
5.times { print "We *love* Ruby" }
```

Đa số các ngôn ngữ khác, con số và một số kiểu dữ liệu nguyên thủy khác thì không được coi là hướng đối tượng. Nhưng Ruby theo sự ảnh hưởng lớn của ngôn ngữ Smalltalk bằng việc đưa ra các phương thức và thực thể biến tới tất cả các kiểu của nó. Điều này tạo nên sự thống nhất và dễ dàng khi sử dụng Ruby với tất cả các đối tượng.

3.1.2. Ruby là ngôn ngữ mềm dẻo và linh hoạt

Ruby được coi là một ngôn ngữ mềm dẻo và linh hoạt, từ khi nó cho phép người dùng tự do sửa đổi cả phần cốt lõi của nó. Các phần cốt lõi của Ruby có thể được rời đi hoặc định nghĩa lại như ý muốn. Các phần hiện hữu có thể được thêm vào. Ruby cố gắng không hạn chế các nhà lập trình.

Ví dụ, bổ sung việc thực hiện phép cộng thay vì phải dùng toán tử(+). Nhưng nếu bạn muốn sử dụng khả năng đọc từ "plus", bạn có thể thêm một phương thức tới lớp Numberic. Ban đầu khi đọc đến đây, có thể có ai đó sẽ cho rằng nó giống nạpchồng toán tử trong C++, tuy nhiên thực sự thì không phải vậy. Vì Ruby coi cả các con số là một đối tượng và đối tượng đó thuộc lớp Numeric. Đây là một lớp sẵn có trong thư viện của Ruby. Ví dụ:

```
class Numeric
   def plus(x)
     self.+(x)
   end
end

y=5.plus 6
# y = 11
```

Có thể nói toán tử trong Ruby là một cái đặc biệt cho các phương thức, bạn có thể dễ dàng định nghĩa lại chúng như bạn muốn.

3.1.3. Khái niệm Modules và Mixin trong Ruby

Modun là cách để mà nhóm các phương thức, các lớp và các hằng lại cùng nhau, modun đem lại 2 lợi ích chính cho bạn.

- Đầu tiên là cung cấp một namespace và ngăn cản sự va chạm các tên gọi(phương thức, biến...)
- Thứ hai, là giúp thực hiện khả năng mixin(sẽ được giải thích ở dưới).

Namespaces

Khi bạn bắt đầu viết một dự án lớn với Ruby, tất nhiên bạn sẽ muốn gói lại các đoạn code hay sản phẩm để sau này có thể sử dụng lại, hay như tạo nó thành thư viện riêng của mình. Và để làm như vậy, bạn sẽ chia các phần code vào những file riêng biệt để mà nội dung có thể được chia sẻ giữa những chương trình Ruby khác nhau.

Thường thì code trong các ngôn ngữ lập trình hướng đối tượng như Java, C# thương đặt chúng trong các class, để mà từ đó bạn có dế dàng lấy lại chúng từ một class trong một file nào đó. Tuy nhiên, hạn chế của điều này là đôi khi bạn muốn nhóm những thứ cùng nhau mà không phải tuân theo quy tắc của một class.

Ruby cho phép bạn có thể đặt tất cả những thứ như các hàm, các hằng số... trong một file và khi bạn cần dùng lại nó, đơn giản là chỉ cần gọi file đó từ trong một chương trình bất kỳ. Đây là cách mà ngôn ngữ C cho phép bạn làm. Tuy nhiên, trong ngôn ngữ C vẫn bị 1 lỗi trong trường hợp có hai file khác nhau cùng định nghĩa 1 phương thức có cùng tên gọi. Giả sử file đầu tiên là test1.c và file thứ hai là test2.c, cả 2 file đều có hàm void in(). Một chương trình Main.c chứa hàm main gọi tới hai file để sử dụng các hàm được định nghĩa trong cả 2 file. Lúc này sẽ dẫn đến lỗi nhập nhằng khi cả hai file đều có hàm in.

```
//Test1.c
void in(){
//......
}
//Test2.c
void in(){
//......
}
//Main.c
#include “Test1.c”
#include “Test2.c”
void main(){
in(); // error
```

```
}
```

Khác với C, Ruby có cơ chế tốt hơn trong việc xử lý này. Modules định nghĩa một namspace(Chú ý là không giống namespace của các ngôn ngữ khác là các hàm và biến là phải đặt trong một class), như một cái hộp mà bên trong nó có chứa các phương thức và hằng số của bạn. Các hàm có thể được đặt vào trong một modun như một phương thức static trong class. Trường hợp lỗi trên được giải quyết như sau trong Ruby:

```
#File Test1.rb
module Test1
PI=3.14 # const
def  Test1.in()
puts “test1”
end
end
#File Test2.rb
module Test2
Hangso = 100 #const
def  Test2.in()
puts “test2”
end
end
#File chạy Test
require 'Test1'
require 'Test2'
Test1.in   #test1
Test2.in   #tes2
#Tham chiếu đến 1 hằng số
puts Test1::PI  #3.14
```

Mixin

Modun còn có một ưu điểm nữa rất hay, có thể coi là sức mạnh của Ruby đó là cung cấp khả năng mixin. Vậy mixin là gì?

Trước tiên chúng ta hãy so sánh với một số ngôn ngữ lập trình khác. C++ cho phép đa thừa kế, đây là sức mạnh của C++, tuy nhiên đa thừa kế cũng có thể dẫn đến sự nguy hiểm, nhập nhằng. Còn Java, C# thì cho phép đơn thừa kế. Còn Ruby? Ruby rất linh hoạt và cho phép bạn sử dụng cả đơn thừa kế và sức mạnh của đa thừa kế. Chú ý rằng Ruby có sức mạnh của đa thừa kế nhưng nó vẫn chỉ là ngôn ngữ đơn thừa kế. Cái gì giúp nó có sức mạnh của đa thừa kế, đó chính là khả năng mixin. Mixin cung cấp 1 khả năng như đa thừa kế nhưng lại bỏ đi mặt hạn chế của nó.

Trong các ví dụ trước. Chúng ta đã định nghĩa các phương thức trong modun(có vẻ giống như một phương thức static trong class), các phương thức sử dụng tên mà được định nghĩa với tiền tố là tên của modun(Ví dụ: Test1.in). Nếu điều này làm bạn nghĩ đến các phương thức của lớp, có lẽ suy nghĩ tiếp theo của bạn là "điều gì xảy ra nếu tôi định nghĩa các phương thức thực thể trong modun?".

Câu trả lời đơn giản là một modun không thể có các thực thể, vì nó không phải là lớp. Tuy nhiên, vào lúc bạn include một module vào định nghĩa của lớp, khi đó tất cả các phương thức trong modun đó sẽ chính xác trở thành các phương thức, hay thức thể của lớp đó. Và chúng ta gọi hành động đó là đã được "mixed in", tức là nó đã trở thành các thực thể instance của 1 lớp. Thực tế, modun mixed-in có hiệu quả như một siêu lớp. Hãy xem ví dụ sau để thấy điều đó:

```
/*-------------------------Modun A ------------------------------------*/
module A
def in
    puts  "Đây là #{self.class.name}"
end
end
/*-------------------------Lớp B ------------------------------------*/
class B
def test
     puts "Test"
end
end
/*-------------------------Lớp C ------------------------------------*/
class C < B
include A
```

```
#...
end
#Chạy thử
A.in # Đây là Module
c = C.new
c.in   #Đây là C
c.test #Test
```

Trong lập trình C, "include" được tiền xử lý để thêm nội dung của 1 file vào trong một file được biên dịch khác. Còn khai báo này của Ruby đơn giản là làm tham chiếu tới một tên của một module. Nếu module đặt trong một file khác, bạn phải sử dụng câu lệnh "require" để tham chiếu đến module.

Với Ruby "include" không đơn giản chỉ là copy các phương thức trong module vào trong lớp đó, nó làm việc tham chiếu từ lớp tới modun được include. Nếu nhiều lớp được include từ module, tất cả chúng sẽ trỏ tới modun được include.

Nếu bạn định nghĩa một phương thức trong một modun, ngay cả trong chương trình của bạn đang chạy, tất cả các lớp mà include modun sẽ có cách xử lý riêng đối với các đối tượng trong lớp của nó. Về cơ chế gọi phương thức và biến thì, mặc định ban đầu nó sẽ gọi phương thức được định nghĩa trong lớp, nếu không có thì sẽ gọi tới phương thức được mixin từ Module, cuối cùng là lớp cha của nó.

Mixin cung cấp cho bạn nhiều cách để thêm vào các phương thức mới vào trong các lớp. Tuy nhiên, sức mạnh thực sự được nhận thấy khi các đoạn code trong mixin(trong Module được thêm vào) tương tác với đoạn code trong lớp của nó. Và đặc biệt là ta có thể thêm các phương thức đã định nghĩa sẵn trong các thư viện của Ruby để xây dựng vào trong lớp của ta. Ví dụ, trong thư viện sẵn có có lớp Comparable, bạn có thể sử dụng "Comparable mixin" để thêm các toán tử so sánh(<,<=,==,>=,>). Để làm việc này, Compare thừa nhận rằng nếu bạn định nghĩa toán tử <==> thì bất kỳ các toán từ khác ở trên đều coi như là được định nghĩa. Bởi vậy, khi môt khi bạn viết một lớp, bạn định nghĩa một phương thức toán tử này <=>, và include Comparable, thì sẽ nhận 6 chức năng so sánh một cách tự do. Hay thử với lớp sau đây, được định nghĩa sau:

```
class Song
  include Comparable
  attr_reader :number
```

```
  def initialize(number)
    @number = number
  end
  def <=>(other)
    self.number <=> other.number
  end
end
n1 = Song.new(10)
n2 = Song.new(100)
puts n1 <=> n2 #  -1
puts n1 <= n2   # true
puts n1 >= n2   #false
puts n1 == n2   #false
```

3.2 Trừu tượng hóa cơ sở dữ liệu với Active Record

Nói chung, chúng ta sẽ muốn ứng dụng web của chúng ta lưu giữ thông tin của chúng trong các mối quan hệ về cơ sở dữ liệu. Tuy nhiên để kết hợp giữa việc lập trình hướng đối tượng với các quan hệ trong cơ sở dữ liệu lại không phải là chuyện đơn giản. Các đối tượng là tất cả gồm dữ liệu và các phép toán, còn cơ sở dữ liệu là tất cả các bộ giá trị. Ta có thể dễ dàng mô tả nó bằng thuật ngữ quan hệ tuy nhiên nó lại trở nên khó cho việc code trong lập trình hướng đối tượng. Qua thời gian, cộng đồng đã tìm ra cách để làm cho sự hòa hợp giữa các quan hệ dữ liệu và hướng đối tượng trở nên thống nhất hơn. Rails đã biết cách trừu tượng cơ sở dữ liệu, giúp cho các nhà lập trình có thể cảm thấy việc tương tác với các quan hệ dữ liệu trở nên đơn giản hơn trong lập trình.

3.2.1. ActiveRecord

Active Record là một tầng ánh xạ quan hệ với đối tượng(object-relational mapping hay ORM) được hỗ trợ bởi Rails. ActiveRecord được xây dưng gần giống tiêu chuẩn mô hình ORM. Mô hình ORM cho phép các bảng ánh xạ tới các lớp, các hàng tới các đối tượng, và các cột tới các thuộc tính. Nếu một cơ sở dữ liệu có một bảng được gọi là orders, chương trình của chúng ta sẽ có một lớp được đặt tên là Order. Tuy nhiên nó không giống hầu hết các thư viện ORM chuẩn bởi cách nó được cấu hình. Bằng cách sử dụng bộ mặc định nhạy bén, Active Record làm giảm tải lượng cấu hình mà nhà phát triển bình thường sẽ phải thực hiện.

1. Tạo quan hệ trong cơ sở dữ liệu với ActiveRecord

Hầu hết các ứng dụng đều làm việc với rất nhiều bảng trong cơ sở dữ liệu, và thông thường đều có các mối quan hệ giữa các bảng với nhau, có thể là kiểu quan hệ môt – một, quan hệ một – nhiều hay quan hệ nhiều – nhiều.

Hay ví dụ như mỗi Đơn_Đặt_Hàng sẽ có rất nhiều danh mục sản phẩm, mà mỗi danh mục sản phẩm đó sẽ tham chiếu tới một Sản_phẩm cụ thể nào đó. Và ngược lại, mỗi sản phẩm cụ thể nào đó chỉ nằm trong một Đơn_đặt_hàng. Đây là quan hệ một nhiều.

Thông thường để giải quyết điều này, trong phạm vi cơ sở dữ liệu, các mối quan hệ được diễn tả bằng các liên kết các bản dựa trên các khóa chính, khóa ngoại. Và nếu như ví dụ trên thì ta ta sẽ có một bảng trung gian là bản Danh_Mục_Sản_Phẩm, bảng này sẽ chưa một khóa chính cho phù hợp với bảng Sản_Phẩm. Đồng thời bảng Danh_Mục_Sản_Phẩm phải nói rằng nó có có một khóa ngoại được tham chiếu tới bảng Đơn_Đặt_Hàng.

Đơn_Đặt_Hàng(**id_order**,.....)

Sản_Phẩm(**id_product**,.....)

Danh_Muc_Sản_Phẩm(**id_product**[primary key], id_order[foreign key],.....)

Nhưng đó là ngôn ngữ ở mức thấp. Rails cho phép chúng ta sử dụng các đối tượng model để tạo nên quan hệ dữ liệu, mà không cần quan tâm đến các cột khóa. Và hơn thế nữa, nếu một Sản_Phẩm cụ thể luôn thuộc một Đơn_Đặt_Hàng, ta cũng muốn có một cách nào đó để nhanh chóng từ đối tượng Sản_Phẩm có thể thao tác với dữ liệu nằm tại Đơn_Đặt_Hàng, ví dụ như:

id = product.order.id_order

ActiveRecord đã giúp ta việc này, hay với khả năng ORM, cho phép biến đổi từ mối quan hệ khóa ngoại ở mức độ thấp trong cơ sở dữ liệu thành việc ánh xạ đối tượng ở mức độ cao. Và nó xử lý theo ba trường hợp:

- Một hàng trong bảng A được kết hợp với rỗng một hàng trong bảng B.
- Một hàng trong bảng A được kết hợp với một số các hàng trong bảng B.
- Một số các hàng trong bản A kết hợp với một số các hàng trong bảng B.

Với sự hỗ trợ của ActiveRecord, chúng ta có thể giảm tải việc thao tác trên cơ sở dữ liệu, bằng cách thêm vào một ít code.

2. Xác định mối quan hệ

ActiveRecord hỗ trợ ba loại kiểu quan hệ giữa các bảng: môt-môt, môt-nhiều và nhiều-nhiều. Và bạn chỉ ra mối quan hệ đấy bằng cách định nghĩa trong các model các khai báo:

has_one, has_many, belongs_to and has_and_belongs_to_many.

Quan hệ môt-một có lẽ là tồn tại giữa một đơn đặt hàng với một hóa đơn. Và khi đó ta định nghĩa trong cả hai lớp Order(Đơn_Đặt_Hàng) và Invoice(Hóa_Đơn) như sau:

```
class Order < ActiveRecord::Base
has_one :invoice
. . .
class Invoice < ActiveRecord::Base
belongs_to :order
. . .
```

Đơn_Đặt_Hàng và Danh_Mục_Sản_Phẩm(LineItem) có mối quan hệ một nhiều: hay một đơn đặt hàng có nhiều danh mục sản phẩm. Rails sẽ code như sau:

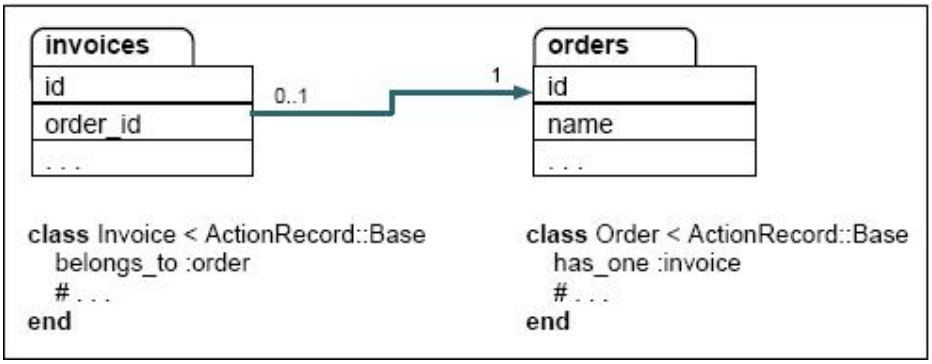
```
class Order < ActiveRecord::Base
has_many :line_items
. . .
class LineItem < ActiveRecord::Base
belongs_to :order
. . .
```

Xét trường hợp quan hệ nhiều-nhiều. Ta giả sử ta phân loại mặt hàng các sản phẩm, thì có lẽ ta có: một sản phẩm có thể thuộc nhiều loại mặt hàng(ví dụ "Vở" có thể thuộc cả loại mặt hàng đồ dùng học tập cũng thuộc loại mặt hàng tạp phẩm). Và ngược lại một loại mặt hàng thì có thể có nhiều sản phẩm hàng trong đó. Đây là một quan hệ nhiều-nhiều. Rails sẽ diễn tả nó như sau:

```
class Product < ActiveRecord::Base
has_and_belongs_to_many :categories
. . .
class Category < ActiveRecord::Base
```

```
has_and_belongs_to_many :products
. . .
```

Đi vào chi tiết việc Rails định nghĩa các mối quan hệ này. Ta hãy trở lại việc xác định mối quan hệ môt-một.



Quan hệ một-một, thông thường sử dụng một thuộc tính là khóa ngoại trong một bảng để chỉ ra mối quan hệ giữa một hàng của bảng này tham chiếu tới ít nhất một hàng trong bảng kia. Tương tư như vây, hình vẽ trên cũng chỉ ra rằng một đơn đặt hàng có thể có một hóa đơn hoặc chẳng có hóa đơn nào.

Chúng ta thêm belong_to vào lớp Invoice(Hóa_Đơn) là tương tự như ta thêm khóa ngoại, và chỉ ra khóa ngoại đó là nằm trong lớp Order.

Việc đổi vị trí has_one và belongs_to về cơ bản là không khác nhau lắm, tuy nhiên khi save thì nó có khác nhau. Tức là lớp mà có has_one thường phải tạo ra trước, rồi sau đó lớp chứa phụ thuộc belongs_to sẽ được save sau. Điều này cũng giống với việc thứ tự mà ta insert dữ liệu vào bảng khi có ràng buộc về khóa ngoại trong cơ sở dữ liệu.

Theo mặc định thì định dạng khai báo thì thuộc tính tên lớp và trường khóa ngoại là một danh từ số ít và được thêm vào sau (_id)

Bạn cũng có thể viết đè cho mặc định với một cái tên khác ví dụ đặt tên là paid_order.

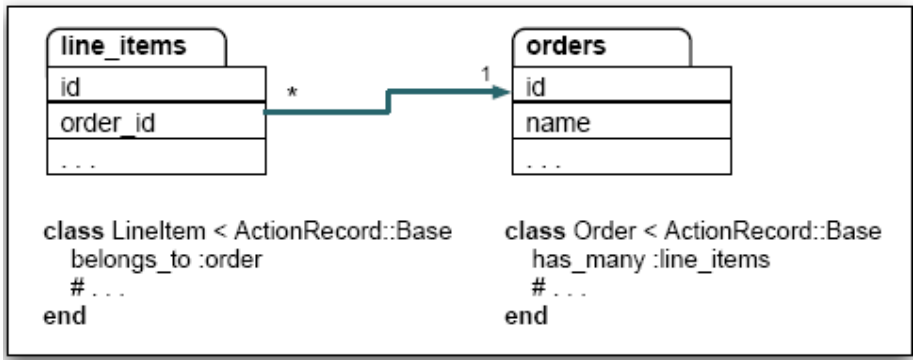
```
class LineItem < ActiveRecord::Base
        belongs_to :paid_order,
            :class_name => "Order",
            :foreign_key => "order_id",
            :conditions => "paid_on is not null"
```

```
end
```

Đoạn code chỉ ra tên mà nó kết hợp là "paid_order" và nó được tham chiếu tới lớp có tên là Order, thông qua khóa ngoại là order_id, và thêm vào đó có điều kiện là cột paid_on là không được phép null.

Chúng ta cũng có thể sử dụng khai báo ":dependent" để thay thế. Khai báo này sẽ nói rằng các hàng trong bảng con không thể tồn tại độc lập với cá hàng trong bảng cha. Điều này có nghĩa là nếu bạn xóa đi một hàng trong bảng cha, và bạn đã định nghĩa :dependent=>true, ActiveRecord sẽ giúp bạn xóa luôn hàng được kết hợp từ bảng con.

Quay lại trường hợp một-nhiều



Một quan hệ môt-nhiều cho phép bạn miêu tả một bộ các đối tượng. Ví dụ, một order có thể có một số các danh mục line_item. Và trong cơ sở dữ liệu, tất cả các hàng danh mục line_item có một order cụ thể chứa một cột khóa ngoại tham chiếu tới bảng orders.

Ở đây bảng cha ordes có khai báo "has_many :line_items" và bảng con có khai báo là "belongs_to :order". Nếu để ý ta thấy Rails là rất nhạy cảm với các danh từ số ít, số nhiều.

Khai báo belongs_to ở đây có ý nghĩa giống trong quan hệ một-một, nghĩa là bảng này luôn phụ thuộc vào một bảng khác.

Riêng đối với khai báo "has_many" thì có thêm một vài chức năng tới model của nó.

Has_many

Has_many định nghĩa một thuộc tính, và thuộc tính này giống như một bộ collection chứa các đối tượng con chính là các hàng trong bảng con, mà như ví dụ trên là bộ các đối tượng line_items. Từ đó bạn có thể dễ dàng truy xuất các đối

tượng đó như truy xuất trong mảng, và thao tác các thuộc tính(các cột trong bảng) của đối tượng đó.

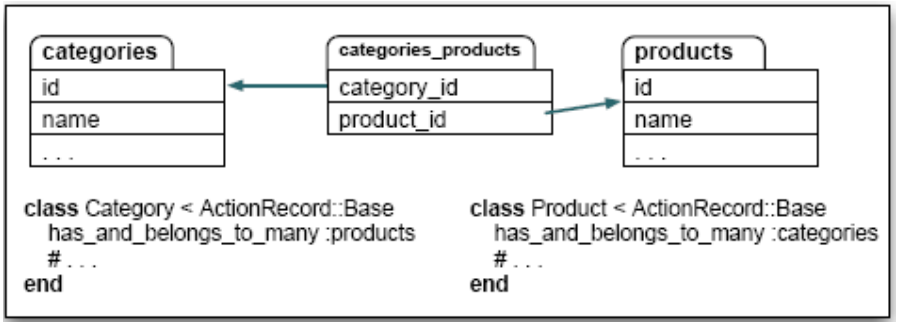
Lây ví dụ với mô tả quan hệ như trên, ta có thể code như sau:

```
order = Order.new
#với một order ở đây bây giờ có thể thao tác với các đối tượng trong danh mục line_items
order.line_items.find(:all) # trả về một mảng danh mục line_items
```

Cũng giống như Has_one, bạn có thể sử dụng khai báo :dependent để thay thế. Hay thay đổi tên lớp kết hợp bằng các khai báo thêm vào :class_name, :foreign_key, :conditions hay :order.

ActiveRecord cũng hỗ trợ cho bạn khả năng viết đè các hàm SQL thao tác của nó. Đôi khi chúng ta cần sử dụng lênh SQL để làm một việc gì đó đặc biệt.

Cuối cùng ta xét tới quan hệ nhiều-nhiều.



Quan hệ nhiều-nhiều được khai báo có tính chất đối xứng, cả hai bảng đều khai báo “has_and_belongs_to_many :products”.

Trong cơ sở dữ liệu, kết hợp nhiều-nhiều được thực hiện bằng cách join bảng, mà bảng này thì chứa một cặp khóa ngoại trỏ tới hai bảng khác nhau. ActiveRecord thừa nhân tên của bảng trung gian này được đặt theo tên của hai bảng kết hợp. Ví dụ bảng trung gian trong quan hệ trên sẽ được đặt tên là “categories_products”.

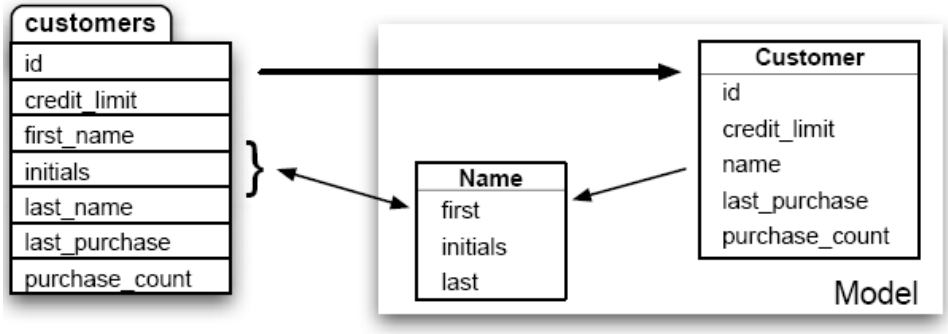
Khai báo “has_and_belongs_to_many” tạo một thuộc tính tương tự như khai báo “has_many” tức là tạo ra một bộ collection các đối tượng. Ngoài ra nó cũng cho phép bạn thêm các thông tin mơi vào bảng join(categories_products).

3.2.2 Trừu tượng ở mức cao(Aggregation)

Các cột dữ liệu có môt bộ các kiểu giới hạn là: integers, strings, dates... Ta muôn cho ứng dụng của chúng ta phong phú hơn hay chúng ta muôn định nghĩa các lớp để miêu tả trừu tượng code của chúng ta. Nó sẽ là thật tuyệt với nếu chúng ta có thẻ như thế nào đó ánh xạ một vài cột thông tin trong cơ sở dữ liệu vào trong các lớp trừu tượng ở mức cao hơn, hay giống như chúng ta đóng gói bản thân các hàng trong một đối tượng model.

Ví dụ, một bảng dữ liệu khách hàng(customer) có lẽ bao gồm các cột được sử dụng để lưu trữ thông tin: firstname, middle name,surname. Và trong chương trình, chúng ta muôn đóng gói các cột mà liên quan đến đối tượng tên gọi(Name). Ba cột nhận việc ánh xạ tới một đối tượng Ruby và đối tượng này cũng được định nghĩa trong mô hình(Model) khách hàng cùng với các trường khác của customer.

Hình dưới đây mô tả ba thuộc tính trong bảng customers sẽ được trừu tượng trong đối tượng có tên là Name



Tiện ích này được gọi là khối tập hợp(*aggregation)* và một số thì gọi nó là kết cấu(*composition).* Nó tùy thuộc cách mà bạn nhìn nó từ trên nhìn xuống hay từ dưới nhìn lên. ActiveRecord hỗ trợ làm viêc này một cách dễ dàng. Bạn sẽ định nghĩa một lớp để giữ dữ liệu, và bạn thêm một khai báo với lớp model và nói cho nó biết các cột dữ liệu nào được ánh xạ và ánh xạ vào trong lớp nào(lớp sẽ chứa dữ liệu thông tin các cột, ở trên là lớp Name).

Lấy ví dụ với mô hình dữ liệu trên, để tạo ra một aggregation, ban đầu ta phải khởi tạo một lớp đối tượng là Name, trong đó định nghĩa việc khởi tạo các thuộc tính first, initials, last. Và định nghĩa cho phép các thuộc tính trong lớp này trả về các dữ liệu. Vi dụ:

```
class Name
    attr_reader :first, :initials, :last
```

```
    def initialize(first, initials, last)
      @first = first
      @initials = initials
      @last = last
    End
   def to_s
     [ @first, @initials, @last ].compact.join(" ")
   end
end
```

Tiếp theo trong model Customer, ta phải khai báo rằng ta có ba trường dữ liệu nên được ánh xạ vào trong đối tượng Name, ta dùng khai báo sau:

```
composed_of : tên_thuộc_tính,
:class_name =>tên_lớp_nào_đó,
:mapping => các_thuộc_tính_cần_ánh_xạ
```

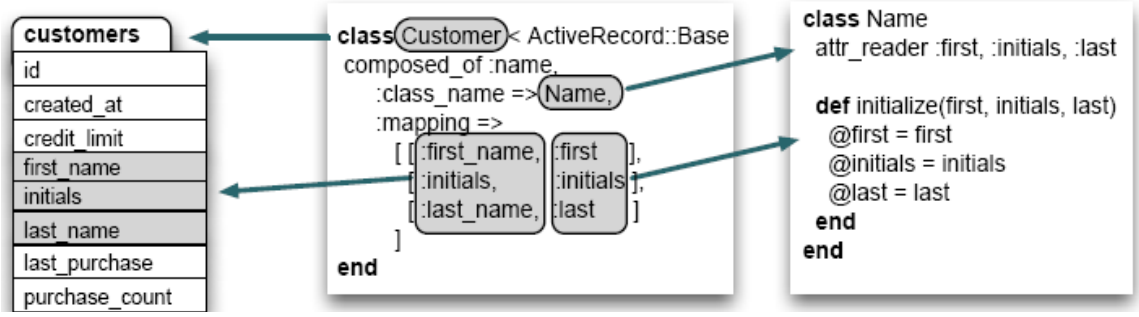
Thực tế, có những lúc ta không cần phải khai báo đầy đủ trên. Nhưng việc khai báo trên cũng rất dễ hiểu. "tên_thuộc_tính" là tên mà ta sẽ dùng để đi từ một đối tượng model(Customer) đến các thuộc tính, hay phương thức trong đối tượng được đóng gói(Name). Và :class_name nhằm chỉ tới tên của lớp mà ta định nghĩa, nếu tên thuộc tính là trùng với tên lớp và viết bằng chữ thường thì ta có thể bỏ qua khai báo này.Ví dụ:

```
class Customer < ActiveRecord::Base
   composed_of :name, :class_name => Name, ...
end
customer = Customer.find(123)
puts customer.name.first
```

Đoạn code trên chỉ là ví dụ về cách đi lại trong aggregation, còn mục đích của đóng gói đối tượng không phải chỉ dùng để lấy các thuộc tính, mà nó có tác dụng khi ta cần xây dựng thêm một số các phương thức được dùng cho việc xử lý một đối tượng nào đó.

Tham số :mapping nói cho Active Record các cột nào trong bản ánh xạ tới các thuộc tính và khởi tạo các tham số trong đối tượng đó. Tham số của :mapping là

một mảng của mảng. Trong đó, mảng bên trong là một mảng có hai thành phần. Thành phần đầu tiên là tên của cột dữ liệu, và thành phần tiếp theo là tên thuộc tính trong đối tượng được khởi tạo. Hình dưới đây sẽ mô tả chi tiết:



3.2.3 Transactions với ActiveRecord

Một transaction trong cơ sở dữ liệu có tác dụng nhóm các thao tác trên cơ sở dữ liệu lại với nhau.

Active Record sử dụng một phương thức transaction() để thực hiện thanh một khối transaction trong cơ sở dữ liệu. Kết thúc của một khối thì transaction được thực hiện(commit), thay đổi cơ sở dữ liêu. Khi có một ngoại lệ xảy ra trong khối, thì tất cả các thay đổi trong khối sẽ được phục hồi lại(roll back) và cơ sở dữ liệu là không có gì bị thay đổi nữa. Chúng ta sử dụng transaction với lớp của ActiveRecord như sau:

```
Account.transaction do
account1.deposit(100)
account2.withdraw(100)
end
```

3.2.4 Thừa kế một bảng đơn lẻ(*single table inheritance)*

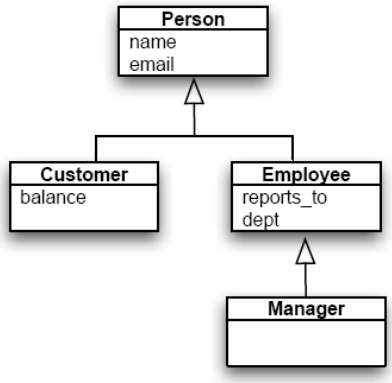
Khi chúng ta lập trình với các đối tượng và các lớp, đôi khi chúng ta sử dụng thừa kế để mô tả quan hệ giữa các các lớp trừu tượng. Ví dụ đối tượng người có thể đóng nhiều vai trò khác nhau như khách hàng, nhân viên, quản lý.... Tất cả các vai trò đó có một vài thuộc tính chung(ví dụ tên, tuổi, giới tính) và một số thuộc tính riêng mà chỉ đối tượng thừa kế mới có.
Đôi khi chúng ta muôn mô hình model bằng cách nào đó nói rằng lớp Nhân viên và lớp Khách hàng là hai lớp con của lớp Người và lớp Manager là lớp con của lớp Nhân Viên. Lớp con thừa kế các thuộc tính và trách nhiệm của lớp cha của nó.

Đối với quan hệ cơ sở dữ liệu, chúng ta không có khái niệm thừa kế: các quan hệ là mô tả chủ yếu bằng các thuật ngữ kết hợp(môt-một,một-nhiều,nhiều-nhiều). Tuy

nhiên, chúng ta cũng có lúc cần lưu giữ một mô hình đối tượng vào bên trong một dữ liệu quan hệ. Có nhiều cách để ánh xạ môt cái vào trong một cái khác. Khả năng đơn giản nhất là cơ chế được gọi là thừa kế một bảng(*single table inheritance*). Cơ chế này cho chúng ta ánh xạ tất cả các lớp thừa kế thứ cấp vào trong một bảng dữ liệu đơn. Bảng này bao gồm các cột cho mỗi thuộc tính của tất cả các lớp thứ cấp. Và nó thêm vào môt cột mà ngầm định thì được gọi là "type", cột này định danh cho lớp cụ thể của đối tượng.
Giả sử bảng dữ liệu như thế này:

```
create table people (
id int not null auto_increment,
type varchar(20) not null,

/* thuộc tính dùng chung cho các lớp thừa kế */
name varchar(100) not null,
email varchar(100) not null,

/* thuộc tính cho đối tượng là Khác hàng có type=Customer */
balance decimal(10,2),

/* thuộc tính cho đối tượng là Nhân viên có type=Employee */
reports_to int,
dept int,
/* thuộc tính cho đối tượng là Quản lý có type=Manager */

/* khai báo khóa */
constraint fk_reports_to foreign key (reports_to) references people(id),
primary key (id)
);
```

Chúng ta định nghĩa lớp dẫn xuất và lớp thừa kế như sau:




```
class Person < ActiveRecord::Base
end
class Customer < Person
end
class Employee < Person
end
class Manager < Employee
end
```

Bây giờ chúng ta thử khai báo và xem kết quả của việc thừa kế

```
Manager.create(:name => 'Bob', :email => "bob@some.add",
                        :dept => 12, :reports_to => nil)
Customer.create(:name => 'Sally', :email => "sally@other.add",
                         :balance => 123.45)
person = Person.find(:first)
puts person.class    #=> Manager
puts person.name     #=> Bob
puts person.dept     #=> 12
person = Person.find_by_name("Sally")
puts person.class     #=> Customer
puts person.email     #=> sally@other.add
puts person.balance   #=> 123.45
```

Để ý đoạn code trên, khi person.class nó sẽ trả về tên lớp là Manager hay Customer bởi vì ActiveRecord ngầm định cột thuộc tính kiểu lớp là cột "type".

3.3. Bộ các công cụ hỗ trợ

3.3.1. Công cụ Rake.

Rails sử dụng công cụ Rake để thực hiện một các nhiệm vụ xác định trong dự án. Một số nhiệm vụ của như là: Tạo, sửa, xóa bảng hay cơ sở dữ liệu. rollback các phiên bản trước của cơ sở dữ liệu. Tạo và chuẩn bị việc kiểm tra cơ sở dữ liệu. Tạo tài liệu cho dự án. Chạy các chức năng test trong giai đoạn kiểm tra. Tạo các file HTML. Xóa các file hay thư mục trong dự án. Xóa các session, cache, hay socket. Update việc cấu hình. Và còn nhiều nhiệm vụ khác nữa.

Việc sử dụng công cụ rake, giúp bạn tích kiệm khá nhiều thời gian. Đơn giản như khi muốn thay đổi cơ sở dữ liệu, tao hay rollback. Rake sẻ giúp bạn làm việc này với chỉ một câu lệnh. Ví dụ để tạo ra các cơ sở dữ liệu đã được định nghĩa trong file cấu hình. Chỉ với câu lệnh **rake db:create:all** thì ba cơ sở dữ liệu sẽ được tạo ra. Trong chương cuối, phần tạo dự án sẽ nói rõ hơn về điều này.

3.3.2. Bộ sinh Generator

Generatator cũng là một công cụ cho phép sinh ra nhiều thành phần trong dự án. Một số các thành phần quan trọng như Model, Controller, Webserivce, Mailer. Ngoài ra nó còn cho phép tự động sinh code với scaffolding.

Khả năng sinh code với Scaffolding

Ngày xưa, khi Rails mới ra đời, "scaffolding" là một đặc tính mà cộng đồng Rails đã coi như một điểm mạnh của sản phẩm vàca ngợi nó. Nhưng lại có một số người chê bai nó, có lẽ là do họ chưa thật sự hiểu được cách sử dụng nó. Có thể định nghĩa đơn giản scaffolding là một công cụ để tạo ra giao diện một trang web nhanh chóng đồng thời vẫn cho phép tương tác với dữ liệu model. Có hai cách tiếp cận scaffolding:

- Scaffolding tạm thời: Cái này đơn giản chỉ cần thêm vào một dòng code tới controllers. Kỹ thuật này sinh ra code tạm thời khi chạy, vì khi cấu trúc chính đã hoàn thành, thì cấu trúc tạm thời sẽ bị bỏ đi.

- Scaffolding vĩnh cửu: Nó sử dụng công cụ generate script để sinh ra các file và code. Dựa vào đó mà chúng ta có thể thử nghiệm và sửa đổi chương trình đó.

Ví dụ sau chỉ ra việc khởi tạo một scaffolding tạm thời, câu lệnh khai báo có thể như sau:

```
class StoryController < ApplicationController
    scaffold :story
end
```

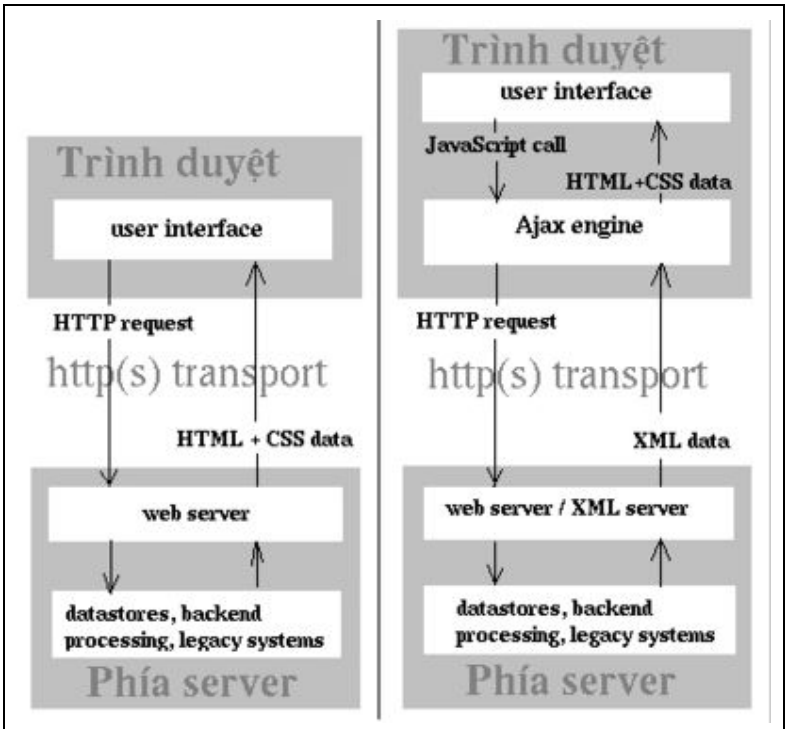
Với câu lênh scaffold trên, thì nó đã đóng vai trò thay thế một loạt các actions khác trong controller để ra một giao diện có khá nhiều các chức năng. Với giao diện này, bạn có thể thoải mái dịch các câu truyện(story), tạo một vài story mới bằng cách kích vào link "New story", hay xóa đi một story. Nói chung là các thao tác cơ bản trên cơ sở dữ liệu. Và như vậy, với scaffold, bạn có thể nhanh chóng theo con đường có sẵn đó để làm việc với model dữ liệu. Hay thúc đẩy việc phát triển giao diện để tạo một vài bảng ghi giả trong cơ sở dữ liệu, làm cho việc phát triển trở nên dễ dàng hơn.

Tuy nhiên, scaffolding cũng có mặt giới hạn của nó, đó là nó không có sự kết hợp của sức mạnh quan hệ mà ActiveRecord mà ở phần trên ta đã nói.

3.4. Kết hợp công nghệ Ajax

3.4.1 Khái niệm Ajax

Để làm rõ về công nghệ Ajax, sau đây sẽ làm một phép so sánh giữa công nghệ web truyền thống và công nghệ web sử dụng Ajax.



Website truyền thống	Website sử dụng Ajax

Các website truyền thống về bản chất là gửi dữ liệu từ các form, được nhập bởi người sử dụng, tới một máy phục vụ web. Máy phục vụ web sẽ trả lời bằng việc gửi về một trang web mới. Do máy phục vụ phải tạo ra một trang web mới mỗi lần như vậy nên các ứng dụng chạy chậm hơn.

Trong khi đó, với các website sử dụng Ajax có thể gửi các yêu cầu tới máy phục vụ web để nhận về chỉ những dữ liệu cần thiết, thông qua việc dùng SOAP hoặc một vài dịch vụ web dựa trên nền tảng XML cục bộ khác. Trên máy Client, JavaScript sẽ xử lý các đáp ứng của máy chủ. Kết quả là trang web được hiển thị nhanh hơn vì lượng dữ liệu trao đổi giữa máy chủ và trình duyệt web giảm đi rất nhiều. Thời gian xử lý của máy chủ web cũng vì thế mà được giảm theo vì phần lớn thời gian xử lý được thực hiện trên máy khách của người dùng.

3.4.2 Ajax và Rails

Việc Ajax có mặt trong Rails có phần bắt nguồn từ lịch sử của Rails. Như đã giới thiệu ở chương 2, Rails ban đầu được xây dựng để phát triển các ứng dụng sản phẩm của công ty 37signals, khi đó các nhà phát triển cần sử dụng các chức năng Ajax. Việc viết Ajax trực tiếp bằng Javascript gây ra nhiều khó khăn, mà mất nhiều công sức. Trong quá trình đó, công ty đã quyết định xây dựng Ajax trong Rails, các

hàm chức năng của Ajax được đưa vào trong framework. Kết quả là Ruby on Rails trở thành một trong những framework đầu tiên đưa công nghệ Ajax vào framework.

Có hai khía cạnh trong vấn đề Ajax/Rails. Đầu tiên đó là framework Rails đã sử dụng cả hai framework Javascript là Prototype và script.aculo.us, gói chúng lại trong Rails. Từ đây Rails xây dựng các helper, cái mà cho phép chúng ta dễ dàng sử dụng công nghệ Ajax mà không cần phải gõ Javascript. Thay vào đó là những câu lệnh đơn giản, dễ nhớ sẽ tự động sinh ra javascript khi chạy ứng dụng.

Chương 4
Ứng dụng triển khai

Để minh họa cho mô hình và tính năng của framework Ruby on Rails, sau đây tôi xin trình bày tóm tắt các bước phát triển một dự án với framework này.

4.1. Mô tả ứng dụng:

Dự án tôi xây dựng ở đây là một dự án phát triển web. Trong đó mục tiêu của dự án là làm một trang web có tính chất thương mại, cụ thể là một trang web bán sách. Một trang web thương mại nói chung, thông thường có hai phần chính:

- Phần hiển thị cho khác hàng: bao gồm các thao tác cho đơn giản cho khác hàng như đăng ký, đăng nhập, đặt mua, thông tin cá nhân, nhận xét...
- Phần quản lý nội dung: cho phép người quản lý, thống kê, giám sát mặt hàng, khách hàng và các nội dung khác thông qua trang web quản lý.

4.2. Hướng dẫn cài đặt

Ruby on Rails là một framework cũng giống như Java, cho phép bạn chạy trên nhiều hệ điều hàng khác nhau như Unix hay Window. Không có nhiều sự khác nhau khi chạy trên các hệ điều hành. Trong dự án dưới đây tôi sử dụng hệ điều hành WindowXP để triển khai.

Đầu tiên, chúng ta cần cài đặt Ruby, và framwork Rails. Sau đó cài hệ quản trị dữ liệu MySQL.

Để cài ngôn ngữ Ruby, ta có thể tải file cài đặt ruby186-25.exe cho window trên tại trang http://rubyinstaller.rubyforge.org/.

Tiếp theo sử dụng RubyGem, một công cụ quản lý việc cài đặt của Ruby để cài đặt Ruby on Rails bằng câu lênh sau từ command dos:

C:\> gem install rails --include-dependencies

Việc cài đặt rất đơn giản, ngay bây giờ bạn có thể bắt đầu tạo dự án mà không phải cấu hình gì thêm. Trên thực tế, bạn có thể sử dụng thêm AptanaStudio, một IDE hỗ trợ để phát triển Rails rất tốt. Tuy nhiên, để cảm nhận được sức mạnh của Rails, chúng ta sẽ không sử dụng nó để tạo dự án.

4.3. Tạo dự án

Sau khi cài đặt, bây giờ chúng ta có thể ngay lập tức tạo một dự án với Rails.

Tôi đặt tên dự án này là BookShop, nó là cho một trang web bán sách trực tuyến. Để tạo dự án, từ cửa sổ dos, bạn gõ lệnh: **rails bookshop**

Trong đó bookshop là tên dự án cần tạo ở đây.

Rail sinh ra sẵn kiến trúc MVC của dự án. Bao gồm nhiều thư mục, trong đó ta sẽ quan tâm chính tới một số thư mục sau:

- app là thư mục này là một thư mục chính cho việc tạo các phần trong dự án theo mô hình model, view, controller.
- db là thư mục cho việc tạo và định nghĩa dữ liệu bằng rails.
- config là thư mục dành cho việc cấu hình

4.3.1 Tạo cơ sở dữ liệu dự án với Rails

Phân tích dự án và mô hình dữ liệu

Ban đầu ta cần tạo ra một bảng dữ liệu về các cuốn sách(book). Tuy nhiên, do mỗi sách có thể thuộc một chủ đề nào đó nên ta sẽ tạo một bảng dữ liệu danh mục(categorie). Và mỗi cuốn sách bây giờ bắt buộc phải thuộc một danh mục nào đó, và một danh mục có thể có rất nhiều các cuốn sách khác nhau. Đây là quan hệ một nhiều.

Để giúp cho mỗi khách hàng(account) vào xem sách có thông tin tốt nhất về một cuốn sách. Chúng ta cho phép các khách hàng ghi lại các nhận xét đó đối với các cuốn sách. Đây là một quan hệ một nhiều. Một cuốn sách có thể có nhiều nhận xét.

Sau khi khách hàng vào xem sách, họ có thể đặt mua qua mạng một số cuốn sách nào đó. Để cho người quản trị có thể quản lý được các cuốn sách đã bán, cũng như việc khách hàng có thể lưu lại thông tin lịch sử mua hàng của mình. Ta cần có thêm đối tượng là đơn đặt hàng(order). Quan hệ giữa một quyển sách với đơn đặt hàng là quan hệ nhiều-nhiều. Bởi vì một đơn đặt hàng có thể có nhiều cuốn sách. Và mỗi một cuốn sách nào đó có một mã sách và có thể được nhiều người khác nhau mua nên sẽ có thể thuộc nhiều đơn đặt hàng.

Với các phân tích dự án này, thì dữ liệu có thể được xây dựng như sau:

- categories(**id**, name, desciption)
- books(**id**, name, description, author, publisher, price, link, category_id, created_at)
- accounts(**id**, user, password, age, address, email**)**
- comments(**id**, comment, book_id, account_id, username, date)
- orders_books(**order_id, book_id**, quantity)
- orders(id, user_name, address, phonenumber, email, age)

Triển khai

Trước tiên, chúng ta phải cấu hình cho dự án biết được server và hệ quản trị dữ liệu nằm ở đâu. Có thể nói, đây là phần cấu hình duy nhất cho toàn bộ dự án này.

Dự án Rails khi tạo ra, có tạo ra sẵn môt file câu hình đặt tên là database.yml nằm trong thư mục config. File cấu hình nhắc tới vòng đời phát triển của một dự án như chương 1 có nói tới. Đó là development, test, production.

```
#Cấu hình giai đoạn phát triển.
development:
adapter: mysql
database: book_development
username: root
password: abc123
host: localhost
#Cấu hình giai đoạn kiểm tra.
test:
```

```
adapter: mysql
database: book_test
username: root
password: abc123
host: localhost
#Cấu hình giai đoạn thành sản phẩm.
production:
adapter: mysql
database: book_production
username: root
password: abc123
host: localhost
```

Như đã định nghĩa trong file cấu hình như trên, ta sửa nội dung file sao cho phù hợp. Với hệ quản trị dữ liệu được dùng ở đây là mysql, tên cơ sở dữ liệu cho việc phát triển có tên là book_development. Tương tự cho các giai đoạn kiểm tra và sản phẩm.

Như đã giới thiêu trong chương 3 về công cụ rake, sau khi cấu hình, để nhanh chóng tạo ra ba cơ sở dữ liệu. Ta gõ lệnh: **`rake db:create:all`**

Với mô hình quan hệ dữ liệu như phân tích ở trên. Để tạo dữ liệu trong Ruby on Rails, cũng như để đơn giản cho việc quản lý dữ liệu trong Model. Ta sẽ sinh ra các model tương với các đối tượng dữ liệu trên bằng công cụ generator của Rails.

Lượt lượt gõ các lệnh:

```
ruby script/generate model category
ruby script/generate model book
ruby script/generate model account
ruby script/generate model comment
ruby script/generate model order
```

Sau khi gõ các lệnh trên, Rails tự động sinh ra các file tương ứng cho model và database trong các thư mục model, và db.

Thông thường khi làm việc với cơ sở dữ liệu, ta thường trực tiếp phải tác trên hệ quản trị dữ liệu để tạo ra các bảng hay định nghĩa bảng. Đối với Rails, việc tạo ra cơ sở dữ liệu có thể trực tiếp trên hệ thống. Thay vào đó, tại các file được sinh ra

trong thư mục db\migrate: Các file 001_create_books.rb, 002_create_categories.rb, 003_create_accounts.rb, 004_create_comments.rb, 005_create_orders_books.rb. Ta thêm các dòng code để tạo ra bảng dữ liệu tương ứng trong mysql.

Sau khi hoàn thành việc tạo bảng dữ liệu. Ta migration nó vào cơ sở dữ liệu, bằng cách sử dụng câu lệnh của rake: **rake db:create**

4.3.2 Xây dựng các controller, view

Model đã sẵn sàng được tạo ở phần trên, việc của controller bây giờ là lấy dữ liệu từ model rồi gửi cho view.

Ta sẽ tạo ra 4 controller:

- customer cho việc xử lý các hàng động của khách hàng khi duyệt web.
- manage_book cho việc quản lý đối với sách
- manage_comment cho việc quản lý đối với việc ghi chú.
- manage_order cho việc quản lý các đơn đặt hàng.
- manage_accout cho việc quản lý khách hàng.

Để tạo các controller, ta có thể nhanh chóng tạo ra bằng lệnh trong bộ generator. Sử dụng lệnh: **ruby script/generate controller [tên của controller]**

Đối với controller của customer, dự án này sẽ tạo ra các action như index, login, logout, registry, write_comment, shoppingcart tương ứng với việc khác hàng đăng nhập, đăng ký, viết nhận xét hay đặt hàng, các phần này sẽ xử lý logic như việc phân tích các tham số được submit lên rồi từ đó sử dụng Model để lấy ra các dữ liệu phù hợp gửi tới View.

Riêng với các phần quản lý manage, để nhanh chóng tạo ra phần quản lý, trong dự án này sử dụng khả năng sinh code với scaffold của Rails. Sau đó ta có thể thay đổi một số thứ về giao diện hay xử lý với các code được sinh ra.

Ta sử dụng câu lệnh: ruby script/generate scaffold [tên của model]

Đối với phần view, ngoài việc viết định dạng code với HTML, ta có thể kết hợp sử dụng Ajax vào. Các câu lệnh Ajax trở nên thân thiện mà gần gũi hơn khi sử dụng Rails. Việc sử dụng Ajax sẽ làm tăng tốc việc xử lý trên trình duyệt cũng như trên website, giúp khách hàng thao tác cũng nhanh hơn.

Kết quả trang giao diện của trang web ta như sau:

1. Xây dựng phần quản lý

Để thuận tiện cho việc thông kê và nhập liệu sách, quản lý các comments, khách hàng đăng ký và các đơn đặt hàng, ta sẽ tạo ra các controller, view có giao diện như sau:

Kết luận

Sau thời gian nghiên cứu, và tìm hiểu framework Ruby on Rails, cũng như trong giai đoạn thử nghiệm phát triển ứng dụng web với Ruby on Rails. Khóa luận đã đem lại một cái nhìn tổng thể về các framework cũng như làm rõ hơn về sức mạnh và các tính năng mà framework Ruby on Rails đem lại. Từ đó giúp chúng ta có những nhận định đánh giá khách quan về framework này. Đặc biệt, qua quá trình thực nghiệm nhanh chóng trong việc phát triển ứng dụng web BookShop, chúng ta có thể khẳng định rằng Ruby on Rails là một trong những framework mạnh, và nó sẽ còn tiếp tục phát triển mạnh trong những năm tiếp theo. Tôi tin rằng, số lượng các nhà phát triển ứng dụng web sử dụng Ruby on Rails sẽ ngày một đông hơn.

Do thời gian cũng như điều kiện chưa cho phép, trong ứng dụng thực nghiệm bookshop, chúng ta mới chỉ phát triển được những chức năng chính của một trang web bán hàng. Tuy nhiên, một website thực sự sẽ còn phải có rất nhiều các chức năng khác, để đem lại cho những khách hàng cảm giác thoải mái, và thú vị khi tham gia duyệt web. Trong tương lai, chúng tôi sẽ bổ sung một số các tính năng khác nữa cho trang web như khả năng đặt mua hàng thông qua thẻ tín dụng hay phát triển thêm một diễn đàn sách, để từ đó giúp cho người yêu sách có những thông tin tốt nhất về các loại sách và sản phẩm sách khi đặt mua.

Trong phạm vi của một khóa luận tốt nghiệp, luân văn này không tránh khỏi những thiếu sót trong nhiều mặt. Em mong nhận được sự phê bình, chỉ bảo tận tình của các thầy cô và các bạn, để từ đó làm rõ hơn về framework này. Em xin chân thành cảm ơn!

Tài liệu tham khảo

[1] Ajax on Rails_ Build Dynamic Web Applications with Ruby

[2] Agile Web Development with Rails,

http://www.pragmaticprogrammer.com/titles/rails/index.html

[3] wikipedia, http://en.wikipedia.org/wiki/Software_framework
[4] website chính thức về Ruby http://www.ruby-lang.org/
[5] website chính thức về Ruby on Rails http://www.rubyonrails.org/docs