



Typo Content Management System Architecture Documentation

Nicholas CHEN

nchen@uiuc.edu

http://softwareengineering.vazexqi.com/files/typo_documentation.pdf

CS527: Advanced Topics in Software Engineering

December 19, 2006

Contents

1	Introduction	7
2	Typo Documentation Roadmap	9
2.1	Overview of Documentation	9
2.2	Conventions in this document	10
3	Typo Overview	11
3.1	Background	11
3.2	Typo look and feel	12
3.3	Resources for Typo	12
4	Typo Module Viewtype	13
4.1	Model-View-Controller in Typo	13
4.2	Models in Typo	13
4.2.1	Typo main models	15
4.2.2	Other notable models	17
4.3	Controllers in Typo	19
4.4	Views in Typos	21
4.5	Web Services in Typo	23
4.6	Other modules	24
5	Typo Component-and-Connector Viewtype	27
5.1	Rails default HTTP request handling	27
5.2	Request for public pages	27
5.3	Requests for administrative pages	30
5.4	Discussion on Typo connection-and-components	32
6	Typo Allocation Viewtype	35
6.1	Typo directory system	35
7	Conclusion	39

List of Figures

3.1	Two different views for Typo: the public view and the administrative view	12
4.1	EmailNotifier (app/models/email_notifier.rb)	14
4.2	WebNotifier (app/models/web_notifier.rb)	15
4.3	Models in Typo	16
4.4	Class diagram for main Typo models (app/models)	17
4.5	Editing categories for articles	18
4.6	Typo sidebar setting	19
4.7	Class diagram for Typo controllers (app/controllers)	20
4.8	Filters for authentication (app/controllers/admin/base_controller.rb)	21
4.9	Routing rules in Typo (app/config/routes.rb)	22
4.10	Find by date (app/controllers/articles_controller.rb)	22
4.11	rxml template in Typo (app/views/cache/list.rhtml)	23
4.12	rhtml template in Typo (app/views/xml/_rss20_item_trackback.rxml)	23
4.13	The Web Services Controller in Typo (app/controllers/backend_controller.rb)	24
4.14	Class diagram for Typo Web Services app/api	25
5.1	A <i>pseudo</i> sequence diagram of article request	28
5.2	Mapping main site to corresponding controllers in (app/config/routes.rb)	29
5.3	Typo theme selection (app/controllers/content_controller.rb)	30
5.4	A <i>pseudo</i> sequence diagram of administrative page request	31
6.1	Typo Directory Structure	36
6.2	Detailed view of the app directory	38

Chapter 1

Introduction

This is an architectural documentation for the Typo Content Management System done as part of the project for CS527: Advanced Topics in Software Engineering at the University of Illinois at Urbana-Champaign.

Usually when we think about documentation, we envision a reference document that we can use to learn about the API and function definitions as well as how to use those functions. While that type of documentation is certainly useful it is not what this document is trying to accomplish. Instead, this document serves as an architectural documentation. An architectural documentation is *different* from the usual document because it captures the components of the system and attempts to describe the relations between those components. From those relationships, we try to infer the design rationale behind that architecture.

What is software architecture? While there are many acceptable definitions, we will use the one given by Garlan and Perry:

The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.

Every system out there has its own architecture. This architecture might not be obvious or it might just be a large *ball of mud*. Nonetheless, a big ball of mud is also a form of architecture — perhaps not the most elegant but definitely the most common. Therefore, Typo itself has its own architecture that needs to be discovered and distilled. That is our goal in this document.

Because Typo itself is constantly evolving¹— the current stable version is 4.03 — any detailed documentation will soon become obsolete and worthless. Instead, this document presents an abstract or conceptual model for Typo. This level of architecture is concerned with the large-scale subsystems within Typo itself and how they interact. This model is most useful for entry-level developers who are interested in modifying the functionalities of Typo.

We believe that concrete examples are the best way to illustrate important concepts. Thus this document will make use of screenshots and code excerpts to illustrate concepts that we are trying to make. Whenever possible, references to the files in the Typo source code or external sites that

¹The next version of Typo will support the **RESTful** features in Ruby on Rails 1.2

describe certain concepts better will be provided. Thus, this document also serves as a *pointer* for learning the more advanced features of Rails and Typo by building upon the contributions of previous contributors and developers.

All examples have been tested on Typo 4.03 on Rails 1.1.6 running on the webrick server and interfacing with a MySQL database back-end².

Since Typo is built on the Ruby on Rails framework (or just Rails), we assume that the reader is familiar with it. The reader should be comfortable with the basic modules in Rails: **Active Record**, **Action Controller** and **Action View**. Detailed and up-to-date information on Rails can be found at [Ruby on Rails](#) official site.

This documentation is written loosely following the style advocated by [2] but with a more pragmatic style of writing to make this work accessible to more readers. Ruby on Rails developers are more familiar with a *feature*-driven description of the framework and we try to mimic that approach here as well. We motivate the discussion of a concept by giving an example of where it is used and then describe how and why it is implemented as such.

²An easier way of running Typo is to install it as a ruby gem and then use the Mongrel server and interface it with the SQLite3 database. However this method still has some bugs so we are sticking with the conventional way of installing and running Typo.

Chapter 2

Typo Documentation Roadmap

2.1 Overview of Documentation

There are three main chapters for this document. Each section examines a particular architectural viewpoint. A viewpoint describes the architecture of a system from a particular perspective. Different architectural viewpoints give us different perspective on how Typo works. Each viewpoint has its own strengths and weaknesses. By presenting three different viewpoints, we get a more comprehensive understanding of the architecture behind Typo.

Each viewpoint is accompanied by one or more views. Views can be figures, tables or even code. Whenever possible, we will describe the view using standard notations such as UML. Moreover, each view is also accompanied by a textual description.

Module Viewtype Shows the different modules in Typo, their functionalities and their relations with one another. The module view shows a *static* snapshot of Typo.

Connector and Connection Viewtype Shows how various components interact at runtime. Understanding the runtime interactions is essential for understanding how Typo works under the hood.

Allocation Viewtype Shows how the Typo directory is organized and where the relevant files reside. The location of files plays an important role in Rails since the framework expects certain files to be in specific locations.

Instead of having a separate chapter as suggested in [2] dedicated to describing the relations between the three views, each chapter will include enough description and cross-references to tie it to the other two chapters.

Our main stakeholders are Rails developers and web designers. Therefore this documentation caters to them and assumes prior knowledge and experience with Rails.

2.2 Conventions in this document

In this document, the relevant file in the source code of Typo will be marked in **bold** text as such: **app/models**. This means that the relevant source file can be found in the app/models directory in the Typo source code.

Whenever possible, links will be embedded into the document itself. Therefore, certain phrases will be in this [color](#) to signify that they link to an external resource. Cross references within this document are also clickable and appear as [Chapter 2](#). Whenever the text refers to a resource in the bibliography, a link will be provided in the form of [\[3\]](#).

To help distinguish between class and method names from the source code, they will be listed as `ClassName` and `method_name`.

Chapter 3

Typo Overview

3.1 Background

Typo is a free, open source blogging engine written using the Ruby on Rails framework. Typo 4.0.3 was released on August 17, 2006 and this documentation will focus on that version. Typo should not be confused with **Typo3** which is another popular content management system written in PHP and uses the MySQL database. To prevent confusion with Typo3, the release number for Typo jumped from version 2 to version 4.

Typo offers the following features¹:

- Uses caching. Typo only creates the served files when needed, and serves static copies to the readers.
- Single level reader comments.
- Spam protection, customizable blacklists and Akismet² support.
- **Textile**, **Markdown** and **SmartyPants** support as well as the options to create your own filters.
- Ping and Trackback.
- Categories and tags for articles.
- **Ta-Da List**, **del.icio.us**, **Flickr**, **43 Things** and **Upcoming.org** syndication can be displayed in the sidebars.
- Ajax support for adding comments and articles.
- Full text search with live preview.
- RSS2 and Atom 1.0 syndication feeds as well as feeds for comments and trackbacks.

¹This list is modified from the one on **Wikipedia**.

²The **Akismet** web service maintains a repository of comments and trackbacks that come from different sites and uses that information to detect spam. When a comment or trackback comes to our site, it can be optionally forwarded to Akismet to flag it as spam or not.

- Supports various databases: [MySQL](#), [SQLite](#), and [PostgreSQL](#).
- Simple URL format for all of the permalinks.
- Web based administration and posting interface as well as support for all 3 major external client APIs (Blogger, MetaWeblog and MovableType Extensions).
- Migration scripts from MovableType 3.x, Textpattern 1.x, WordPress 1.5x-2.0 as well as plain RSS.

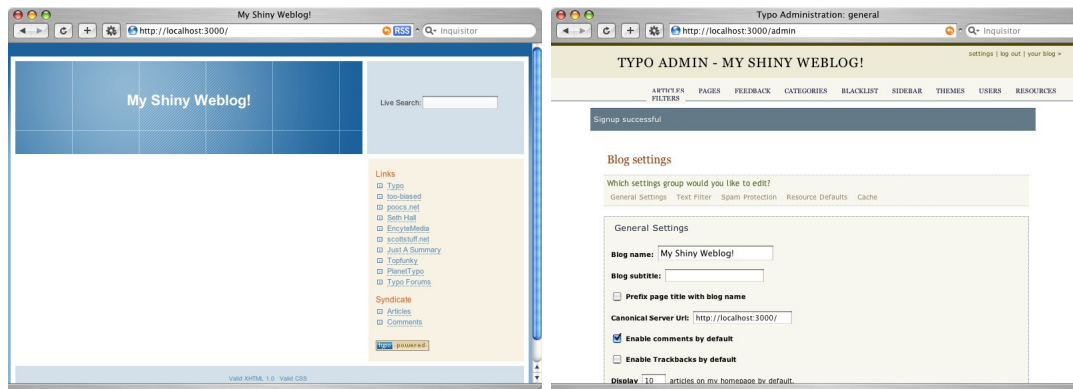
Other popular content management systems written in Ruby are [Hobix](#) and [Mephisto](#).

3.2 Typo look and feel

By default, Typo comes with the *Azure* theme installed (shown in Figure 3.1). The look and feel of our weblog can be easily customized using themes. The ability to create themes is a useful feature for web designers and will be discussed in Section 4.6.

Figure 3.1 Two different views for Typo: the public view and the administrative view

(a) The default weblog interface with no posts yet (b) The administrative view for Typo



3.3 Resources for Typo

For the past few weeks, the official Typo [website](#) has been down due to excessive spam. In the mean time, most of the communication and development has moved over to the [forums](#) and [mailing list](#).

The current version of Typo can be checked out from the Subversion repository at [leetsoft.com](#). Instructions for installing Typo can be found from the README in the distribution or from the [typo forums](#).

Chapter 4

Typo Module Viewtype

4.1 Model-View-Controller in Typo

Rails is a framework for developing web applications following the Model-View-Controller pattern. This pattern maintains a clean separation between the business logic and display logic. Making a conscious effort to preserve this separation leads to code that is easier to maintain. Because the business logic is not tied down to the user interface, either of them can be modified without affecting the other. Model-View-Controller is an example of the Layered Architecture pattern advocated by Evans in [3] for creating a rich domain model for an application.

Typo takes advantage of those features and has well-partitioned layers. In the following sections, we describe each layer in isolation. In Chapter 5 we shall see how these parts interact.

4.2 Models in Typo

Models in Rails are subclasses of the `ActiveRecord` class. `ActiveRecord` wraps a row in a database table, encapsulates the database access and adds domain logic on that data. The `ActiveRecord` class is inspired by a similarly named pattern from [4].

In Rails, the default naming convention for connecting models and database tables is based on pluralization. The database table names are always the pluralized form of the class name. For instance, if we had an `ActiveRecord` class called `Post` then the corresponding table would be called `Posts`. To ensure that common cases of pluralization works, Rails has a fairly complete set of rules for pluralization in its `ActiveSupport` module. Rails is able to pluralize words such as “sheep” (sheep) and “octopus” (octopi)¹.

This tight coupling between the table names and class names has its advantages and disadvantages. It is usually trivial to find the corresponding table given the class name and vice versa. This can help in debugging. However, this tight coupling also makes it harder to refactor the database and the class separately. The developer has to have control over both the class and the

¹This functionality is provided by the `Inflector` module in `ActiveSupport` package. For more information on what other features it provides, refer to the Rails wiki [page](#).

database table. While this is not a problem in smaller development teams, for larger projects sometimes a separate database administrator is in charge of the database. Both developer and database administrator would have to work closely to ensure that the table and class are in sync.

To alleviate this tight coupling, Rails allows `ActiveRecord` classes to configure the name of the corresponding table. This is done using the `set_table_name` method. `set_table_name` takes a string that is the name of the table. This is a useful feature for dealing with legacy tables. Nonetheless, whenever possible developers should name their classes and database tables following the Rails convention.

A diagram² of all the classes that have corresponding database tables is shown in Figure 4.3. Each rectangular box represents a class with the name written on the first row. The fields of each class is described in the second row. Classes are linked together using foreign keys which are annotated using the arrows. A standalone version of the diagram is available from [here](#).

The classes recorded in Figure 4.3 have important information that need to be persisted to the database. There are other classes in `app/models` but since they do not have a corresponding table, they have no persistent values. These other classes provide notification functionalities. For instance, when a comment or a trackback has been added, an e-mail or `jabber` notification can be sent to the author of that article.

Notifications are done by subclasses of `ActiveRecord::Observer` which follows the Observer pattern (see [5]). There are two observers in Typo: `EmailNotifier` and `WebNotifier`. `EmailNotifier` (Figure 4.1) observes the `Article` and `Comment` classes and its `after_save` method gets invoked every time an object of type `Article` or `Comment` is saved. `WebNotifier` (Figure 4.2) observes the `Article` class and its `after_save` method is invoked after an `Article` object is saved. As can be seen from these examples, each model can have multiple observers waiting on it. These *automatic* callbacks are useful but should be used carefully since it is hard to determine the observers of each model when reading the code. The code for registering a class as an observer is not actually written in the *class that is to be observed* but rather in the *class that is observing it*.

Figure 4.1 EmailNotifier (`app/models/email_notifier.rb`)

```

1 class EmailNotifier < ActiveRecord::Observer
2   observe Article, Comment
3
4   def after_save(content)
5     content.send_notifications
6     true
7   end
8 end

```

²This diagram was generated with the help of the `visualizemodels` plugin for Rails. The plugin goes through the classes that have corresponding tables in the database and generates a graph of their relations.

Figure 4.2 WebNotifier (`app/models/web_notifier.rb`)

```
1 class WebNotifier < ActiveRecord::Observer
2   observe Article
3
4   def after_save(article)
5     article.send_pings
6   end
7 end
```

4.2.1 Typo main models

The six main classes of Typo are `Content`, `Article`, `Page`, `Feedback`, `Comment` and `Trackback`. The relationships among these classes are shown in Figure 4.4. Parent classes are indicated by an arrow from the subclasses. Together, these classes implement the main functionality that allows Typo to function as a content management system.

Even though there is a hierarchy between these six classes, it is not immediately obvious from Figure 4.3. The reason being that class inheritance cannot be directly represented in a database. Instead certain patterns have to be implemented to *simulate* this effect. Rails offers the [Single Table Inheritance](#) pattern for representing class hierarchies. Single Table Inheritance represents an inheritance hierarchy of classes as a single table that has columns for all the fields of the various classes. The type of a class is differentiated by the `type` string in the database table. Single Table Inheritance and other patterns for simulating inheritance in databases can be found in [4].

All objects of type `Content` have different states associated with them. They can be *new*, *draft*, *published*, etc. For a list of possible states that `Content` objects can assume, refer to `app/models/content_state`. This is what Typo uses to filter out spam and post notifications when the article changes state from “unpublished” to “just published”.

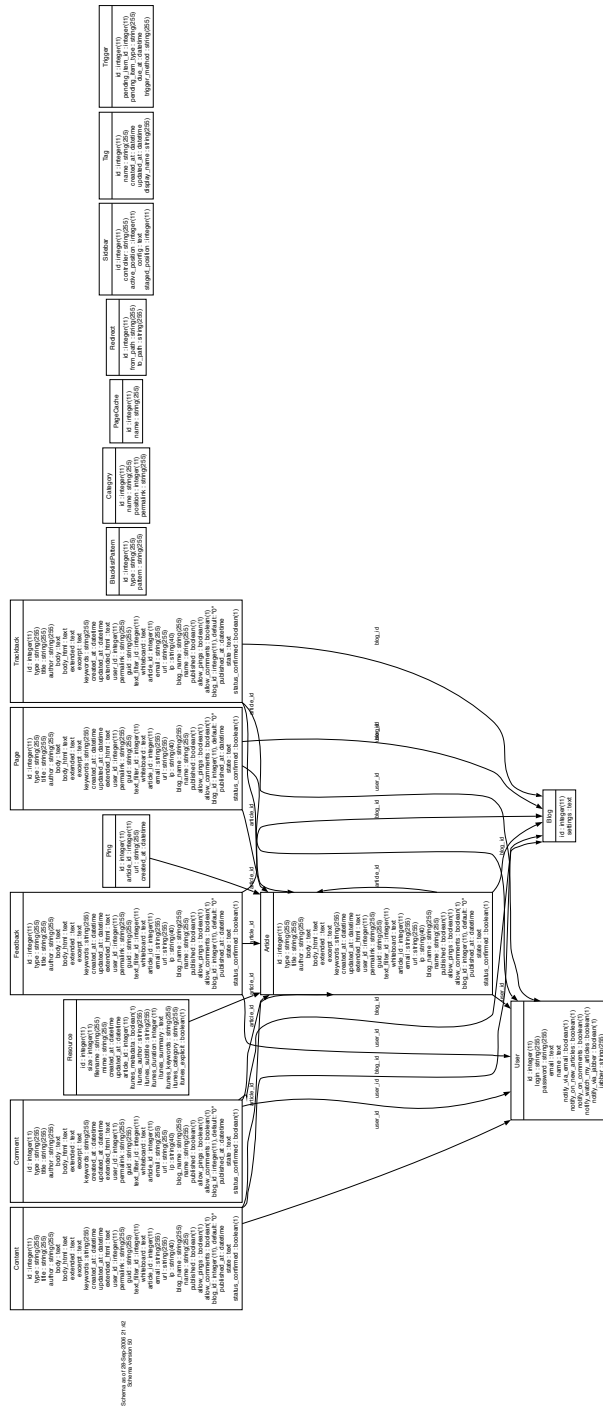
The roles of the six concrete classes are described below.

Article An article is the primary content in a Typo weblog. Articles can have comments and trackbacks. Articles can also be scheduled for immediate posting or future posting at a preset time. Articles can also have individual RSS feeds associated with them.

Page A page is a simplified version of an article. A page does not have comments or trackbacks associated with it. Pages are meant to be static contents with *semi-hidden* URLs. There is no way to navigate to a page without knowing its exact URL or using the admin interface. For a discussion on possible uses of pages in Typo, please refer to [this](#) thread on the mailing list.

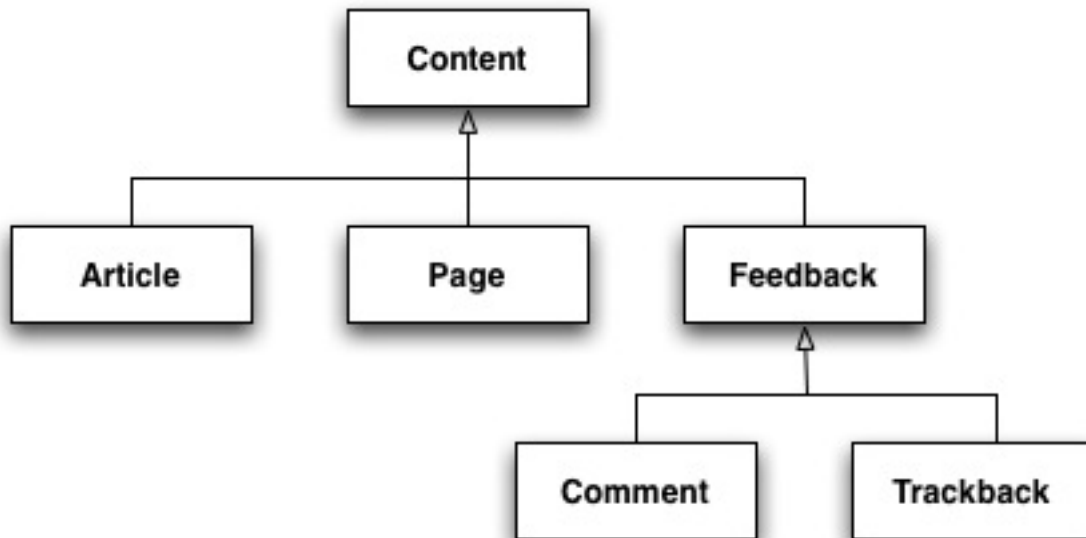
Comment A comment is associated with an article and is left by visitors to an article. Comments contain a visitor’s name, message and optionally his or her e-mail and web site. At the moment, Typo only supports single level comments. Comments flagged as spam or potential spam are not displayed.

Figure 4.3 Models in Typo



Trackback Trackbacks are sent when another site quotes information from our site. It is a type of ping that notifies the author of the article that someone else has used the information from his or her post on a different site. Each article in Typo has a special trackback URI for this. In the current version of Typo, trackbacks are still displayed even when they are flagged as potential spam. Trackbacks are not displayed when they are marked as spam though³.

Figure 4.4 Class diagram for main Typo models (`app/models`)



4.2.2 Other notable models

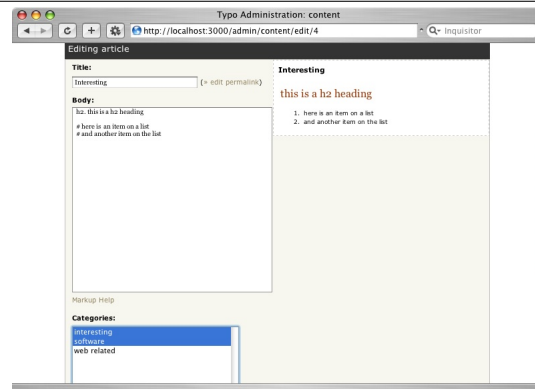
The functionalities of other notable classes are described below.

BlacklistPattern Stores a string or regular expression that is used to filter out spam. These patterns are used by `SpamProtection` (found in `app/lib/spam_protection.rb`) to check against a feedback (comment or trackback) for spam.

Blog Stores the settings of the blog as serialized `YAML` in the database using the `serialize` method of `ActiveRecord`. The settings are recorded from the admin page shown in Figure 3.0b.

Category Each article can have zero or more categories associated with it. Categories can be set via the web service API (see Section 4.5) or via the Typo web interface (see Figure 4.5).

³When we publish an article, Typo automatically sends trackbacks to *other* sites if our article contains references to those sites. Before an article is published, Typo scans through all the links in that article and tries to determine the proper trackback URI for each article to ping. The trackbacks that we send to other sites are not recorded in the `Trackbacks` table.

Figure 4.5 Editing categories for articles

PageCache Typo uses its own cache system instead of relying on the default Rails implementation. This implementation is more conservative in sweeping the cache and will not invalidate all the cache when only superficial changes have occurred. For a more detailed description of the rationale behind this, refer to [this](#) post by a Typo developer.

Ping Typo sends out pings when new articles are posted to external websites that keep track of blogs such as [Technorati](#) and [Blo.gs](#).

Redirect Stores the old and new URL and redirects the user to the new page using HTTP status code 301 when a request comes for the old URL.

Resource A resource is a file associated with each article. This file (usually an podcast) is then sent as an enclosure in the RSS feed. The file is not stored in the database, only the metadata about the file. The file itself is stored in **public/files**

Sidebar The **Sidebar** class keeps track of the sidebars that are activated and their positions. Sidebars can be found on the right hand side of the page in [Figure 3.0a](#). The interface for modifying sidebar settings is done through the admin interface shown in [Figure 4.6](#). For more information on sidebars, refer to [Section 4.6](#).

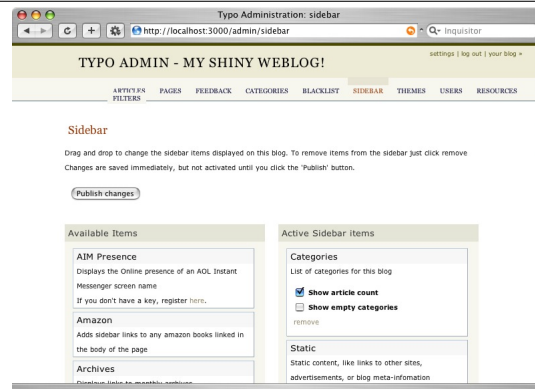
Tag Each article can have zero or more tags associated with it. Tags can only be set through the web interface of Typo when editing an article.

Trigger When an event such as future posting has been schedule, it is recorded in this table. This table is periodically checked every time a HTTP requests comes to perform any pending triggers. This should be used with caution because the time stored with each trigger is the time on the server and this might differ from the local time.

User Stores the information for each author. Typo weblogs can have multiple authors. Each author can edit articles written by other authors.

When Typo needs to be upgraded, migration scripts are provided to migrate the database schemas. These migration scripts can be found in **db/migrate**. The data from the previous version of Typo is preserved when the migration scripts are run.

Figure 4.6 Typo sidebar setting



4.3 Controllers in Typo

Controllers in Rails are subclasses of `ApplicationController`. Incidentally, `ApplicationController` is the name of a pattern in [4]. However, the role of `ApplicationController` is more similar to the Page Controller pattern — also defined in [4]. This discrepancy might lead to some confusion so it is best to mention this here.

Controllers are invoked when an HTTP request comes. The exact controller is determined via a naming convention. An example incoming request could be of the form `http://www.yourdomain.com/mycontroller/myaction/id`. The file `mycontroller_controller.rb` handles the request and calls the method `myaction` which must be a public method or else an exception is raised. The id from the URL is also passed to `myaction`.

By default, a controller is only able to access a model that has the *same* name as it. For example, `BaseController` is supposed to access a model called `Base` which is a subclass of `ActiveRecord`. For a controller to access other models, it needs to declare the dependency using the method `model` and pass the names of the models that it uses (separated by commas): `model :other_model1, :other_model2`.

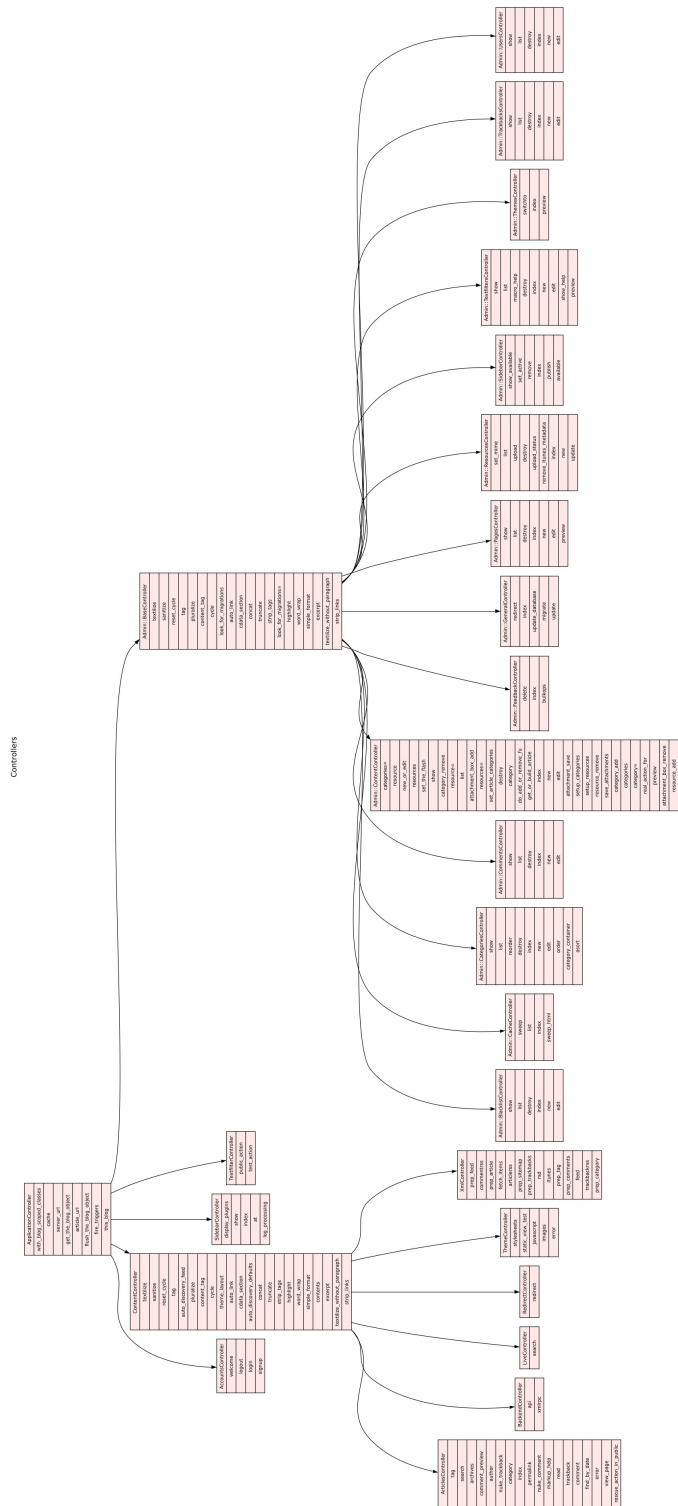
A diagram⁴ of all the controllers in Typo is shown in Figure 4.7. The class name is shown as the first row of each rectangular box. The subsequent rows show the names of the methods (public, protected and private) that each class has. The arrows point from parent class to subclass. A standalone version of the diagram is available from [here](#).

As can be seen from Figure 4.7, the two most important controllers are `ContentController` and `Admin::BaseController`. The name `Admin::BaseController` means that the `BaseController` is located under a directory called `admin` i.e. `app/controllers/admin/base_controller.rb`. This style of grouping controllers into modules is advocated in [7].

`ContentController` and `Admin::BaseController` separate the types of controllers into two groups: the public controllers (take care of requests for public pages) and the administrative controllers (take care of requests for private administrative pages).

⁴This diagram was generated with the help of the [Rails Application Visualizer](#) plugin for Rails.

Figure 4.7 Class diagram for Typo controllers (app/controllers)



Authentication is required before accessing the administrative pages. Login and authentication is taken care of by the `LoginSystem` module defined in `lib/login_system.rb`. Rails has support for *before*, *after* and *around* filters. A filter is akin to an *advice* in Aspect-Oriented Programming. The before filter gets run before each method call; the after filter gets run after each method call; and the around filter gets run before and after each method call. We can restrict the methods that get filtered using extra parameters. On line 3 in Figure 4.8, we run the `login_required` method before each method **except** the login and signup methods (if they exist). Filters makes it convenient to do authentication and verifications. As with observers (see Section 4.2) filters should be used sparingly because they are hard to determine by reading the code.

Figure 4.8 Filters for authentication (`app/controllers/admin/base_controller.rb`)

```

1  ...
2
3  before_filter :login_required , :except => [ :login , :signup ]
4  before_filter :look_for_needed_db_updates ,
5                :except => [ :login , :signup , :update_database , :migrate ]
6  ...

```

Most controllers enable access to the create, read, update and delete (CRUD) of the underlying models. Models create the corresponding objects from the models and relay those objects to the view for displaying. For more information on this, refer to Chapter 5.

The most convenient way of determining what a controller does is to check the URL that invokes that controller. For instance, in the administrative controllers, there is a separate controller for each page: `PageController`, `SidebarController`, etc.

However, just following the normal Rails conventions for handling HTTP requests based on the URL is not sufficient. Typo has support for *pretty* URLs. For instance, a request for all articles written on a particular month comes in the form `http://www.yourdomain.com/articles/2006/11`. If we were to just follow the convention, then we would require a controller for each year (2006 in this case) and a public method for each month!

Instead Rails has a practical way to do this: by specifying the routes as rules in `app/config/routes.rb`. The excerpt shown in Figure 4.9 shows how one controller can handle different URLs. Using regular expressions, the URL is parsed into year, month and day. These values are then passed to the `find_by_date` method in `ArticlesController` as shown in 4.10. So there is the need for only one controller and one method in that controller to handle the different URLs.

Most of the functions in the controllers should be short. The main purpose of a controller is to intercept an HTTP request and delegate the responsibility to a model. It then collects the return value and passes this value to the view.

4.4 Views in Typos

For displaying pages, Rails relies on the `ActionView` module. `ActionView` is an implementation of the Template View pattern in [4]. Template View is a pattern for rendering information into HTML by embedding markers in an HTML page.

Figure 4.9 Routing rules in Typo (**app/config/routes.rb**)

```

1  ...
2
3  map.connect 'articles/:year/:month/:day',
4    :controller => 'articles', :action => 'find_by_date',
5    :year => /\d{4}/, :month => /\d{1,2}/, :day => /\d{1,2}/
6  map.connect 'articles/:year/:month',
7    :controller => 'articles', :action => 'find_by_date',
8    :year => /\d{4}/, :month => /\d{1,2}/
9  map.connect 'articles/:year',
10   :controller => 'articles', :action => 'find_by_date',
11   :year => /\d{4}/
12
13  ...

```

Figure 4.10 Find by date (**app/controllers/articles_controller.rb**)

```

1  ...
2
3  def find_by_date
4    @articles = this_blog.published_articles.find_all_by_date(
5      params[:year], params[:month], params[:day])
6    render_paginated_index
7  end
8
9  ...

```

In Section 4.3 we dissect how a URL was parsed to call the appropriate action in a controller. In our example, `http://www.yourdomain.com/mycontroller/myaction/id` would invoke `mycontroller_controller.rb` and call the method `myaction`. If there is a view associated with that method, then it will be rendered. By default, the view is called `myaction.rhtml` and is found in **app/views/mycontroller**

Rails supports two kinds of templates natively (a plugin can be loaded to support other templates):

- `rxml` templates are used to generate XML responses. This is useful for syndication and web services responses.
- `rhtml` templates are used to generate HTML that can be viewed in web browsers. Ruby code is embedded within the HTML using `<%= ... >` and `<% ... >` markers. The results of the former are displayed in the HTML whereas the results of the latter are suppressed.

Figure 4.11 rxml template in Typo (**app/views/cache/list.rhtml**)

```
1 <% @page_heading = 'Cache' %>
2
3 <% content_for('tasks') do %>
4   <%= task_sweep 'Empty Page Cache' %>
5 <% end %>
6 <div id="users">
7   <p>There are <%= @page_cache_size %> entries in the page cache</p>
8 </div>
```

Figure 4.12 rhtml template in Typo (**app/views/xml/_rss20_item_trackback.rxml**)

```
1 xm.item do
2   xm.title "\"#{item.title}\"_from_#{item.blog_name}_(\"#{item.article.title}\"
3   xm.description item.excerpt
4   xm.pubDate pub_date(item.created_at)
5   xm.guid "urn:uuid:#{item.guid}", "isPermaLink" => "false"
6   xm.link item.url
7 end
```

To avoid duplication, similar code in the views can be abstracted into layouts. Layouts determine the overall look and feel of the page. This is useful for creating consistent headers and footers throughout a website. Layouts are located in **app/views/layouts**.

Sometimes there is some logic that is needed to display the page. It could be something as simple as alternating between table row colors or deciding which graphics to switch between pages. These kinds of logic do not have anything to do with the underlying business logic. However, just embedding the logic into the views makes it hard to read and debug. Instead, such logic should be abstracted into their own helper methods. Views can then easily call these helper methods and reduce clutter in the presentation code. Typo uses helper methods extensively; they can be found in **app/helpers**.

4.5 Web Services in Typo

Support for web services blogging API is also included in Typo. Currently the only use for web services is to enable desktop clients such as **Ecto**, **MarsEdit** and Firefox with the **Performancing** plugin to post new posts or edit current posts without using the Typo web interface.

Typo currently supports three types of blogging web services standards: Blogger, MetaWeblog and MovableType. Only the minimum features of each standard has been implemented in Typo. The implementation of the MovableType API has the most features implemented among the three.

A SOAP or XML-RPC request for <http://www.yourdomain.com/backend/xmlrpc> is handled by the **Backend** controller. Layered dispatching allows the same controller to be used for the three

different blogging standards. This setting is enabled on line 6 in Figure 4.13.

Figure 4.13 The Web Services Controller in Typo (`app/controllers/backend_controller.rb`)

```

1
2 class BackendController < ContentController
3   cache_sweeper :blog_sweeper
4   session :off
5
6   web_service_dispatching_mode :layered
7   web_service_exception_reporting false
8
9   web_service(:metaWeblog) { MetaWeblogService.new(self) }
10  web_service(:mt)         { MovableTypeService.new(self) }
11  web_service(:blogger)   { BloggerService.new(self) }
12
13  alias xmlrpc api
14 end

```

Depending on the type of blogging standard that is requested, a new `TypoWebService` object is constructed in lines 9 – 11 in Figure 4.13. `MetaWeblogService`, `MovableTypeService` and `BloggerService` are all subclasses of `TypoWebService` as shown in Figure 4.14. Each is a part of the Strategy pattern from [5] for handling different blogging standards. The authentication method is defined in the `TypoWebService` base class and shared among its subclasses. The actual authentication is delegated to the `User` class.

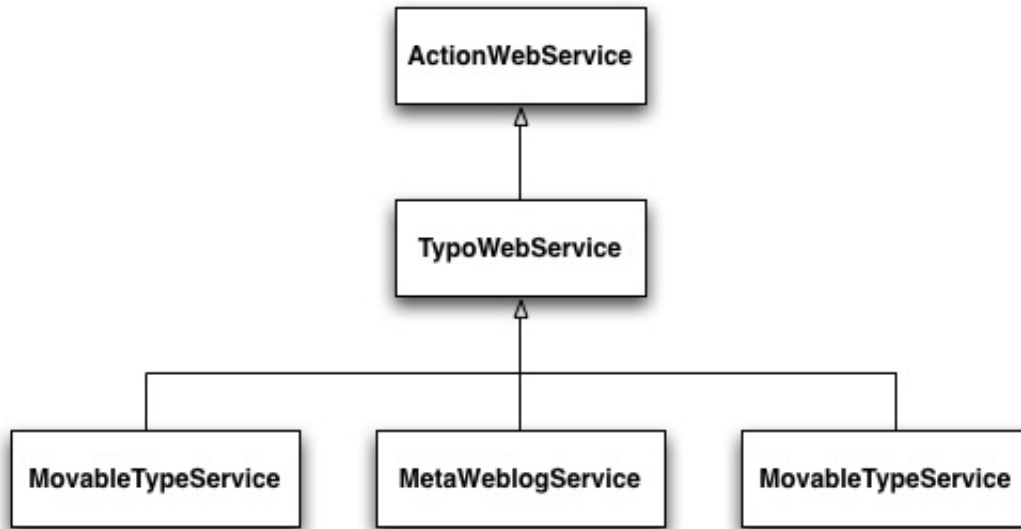
The appropriate `TypoWebService` is constructed and it extracts the information from the SOAP or XML-RPC requests and delegates the calls to the right model. The model then takes care of the request and updates the database as necessary. For more information on how web services work in Rails, refer to [ActionWebService: Web services on Rails](#).

4.6 Other modules

Typo has some extension and customization points that can be enhanced without modifying the main code. These extension points have attracted various contributions from other developers. In fact, the success of Typo is mostly due to its extensibility. These abstractions mean that the developers can easily add new features to Typo without changing the back-end substantially. There are three main points for customization in Typo:

Sidebars Sidebars are extensions that users can write to extend the contents of Typo. They have their own controller and views for displaying tiny bits of information. Some sidebars are updated automatically and display information from other sites. Typo already comes with support for displaying different kinds of information in the sidebars. For instance, there are sidebars to include information from [43things](#), [flickr](#) and [del.icio.us](#). Details on how to create a sidebar can be found in `components/sidebar/README`. A tutorial on creating your own sidebar can be found [here](#).

Figure 4.14 Class diagram for Typo Web Services `app/api`



Themes Themes are user contributed interfaces for Typo. Themes usually do not need to contain their own logic; they might call basic code to include some other contents though. A minimalist theme has been provided in `themes/scribbish`. A tutorial on creating your own themes can be found [here](#).

Filters Filters are text transformations that are applied to articles. There are three kinds of filters: markup filters (convert text into HTML), macro filters (expand embedded tags into full HTML) and post-processing filters (cleans up HTML). Each article has an associated text filter; by default it is *none*. More information on filters can be found [here](#).

These other modules are examples of the Plugin pattern from [4].

Chapter 5

Typo Component-and-Connector Viewtype

5.1 Rails default HTTP request handling

In Section 4.3 we discussed the default HTTP request handling performed by Rails. We also discussed how we can overwrite the default routing convention by modifying `app/config/routes.rb`. However, for Typo, the request handling is usually not that simple. This is because there are hierarchies between different classes. Therefore, methods in the parent class need to be invoked before a request to view an article on our blog can be handled. The situation gets more complicated because filters declared in the parent classes must run as well.

In the following sections we look at how different requests for a page is handled in Typo. We will see the runtime interactions between different components. Fortunately, we have already seen the relationships between classes in Chapter 4. This relationship will come in useful when we determine the filters that need to be run on the controller.

To keep things simple, we will assume that everything runs in one thread and that we do not have to worry about interprocess communications. We will also assume that we are not caching the results from any previous calls¹. The main point here is to see the objects that are involved with each HTTP request.

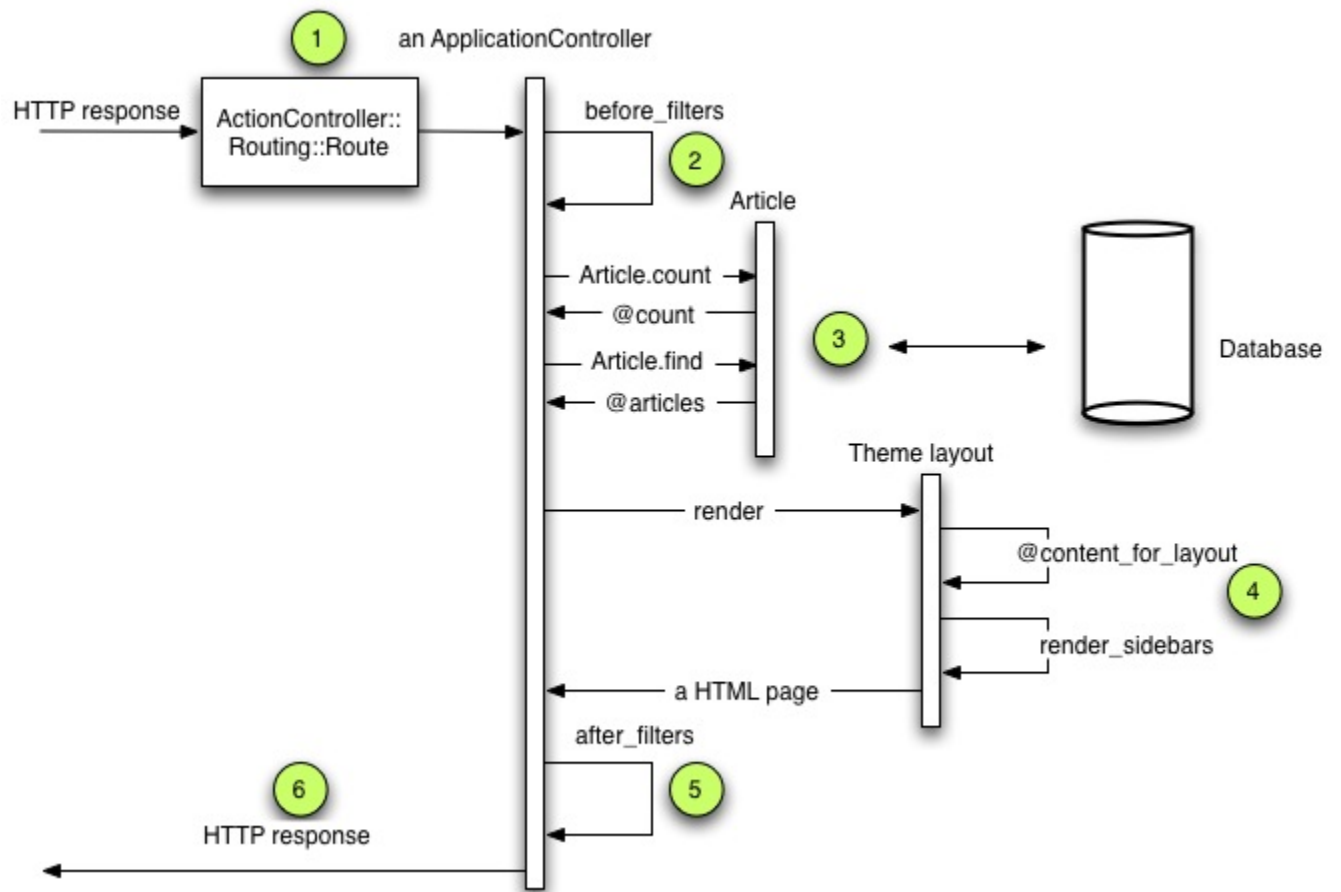
We will consider two scenarios for HTTP request handling: requests for article and requests for administrative pages. Articles are publicly accessible while administrative pages are restricted to registered users of our weblog.

5.2 Request for public pages

Assume that a request comes for `http://www.yourdomain.com/`. A *pseudo* sequence diagram is shown in Figure 5.1. There are six main parts when handling a request for the index of the site as can be seen in Figure 5.1.

¹In fact caching is only turned on by default in *production* mode and turned off in *development* mode. So our assumption that we do not need to check the cache is realistic.

Figure 5.1 A *pseudo* sequence diagram of article request



1. The URL is parse following the mapping rules in **app/config/routes.rb**. In our case, a request for `http://www.yourdomain.com/` will invoke the `index` method on a new instance of `ApplicationController` which is found in **app/controllers/articles_controller.rb**. The relevant line is line 4 in **routes.rb** as shown in Figure 5.2.

Figure 5.2 Mapping main site to corresponding controllers in (**app/config/routes.rb**)

```
1 ActionController::Routing::Routes.draw do |map|
2
3   # default
4   map.index '', :controller => 'articles', :action => 'index'
5   map.admin 'admin', :controller => 'admin/general', :action => 'index'
6
7   ...
```

2. Referring to Figure 4.7 we notice that `ApplicationController` is a subclass of `ContentController` which itself is a subclass of `ApplicationController`. Both the parent and *grandparent* classes define filters: there are `before_filters`, `after_filters` and `around_filters`. Filters are inherited by the descendant classes and must be invoked before the `index` method². The following methods are invoked before the `index` method:

`get_the_blog_post` Returns the current blog and its settings³.

`fire_triggers` Removes any pending triggers (see Section 4.2) that have expired and returns `true`.

`auto_discovery_defaults` Determines and caches the url for the syndication feeds for the current page⁴.

`Blog.filter` Apparently this filter is never called since there is no `filter` class method in `Blog`.

3. After the filters have been invoked, the `ApplicationController` objects calls the `count` and `find` class methods on `Article`. `Article` is an `ActiveRecord` class that maps to the database. This `ApplicationController` object finds all the posts and keeps a count of them. The count is needed for pagination.
4. Our `ApplicationController` object then chooses the layout based on the current theme and sends it to the page template for rendering. The code snippet that chooses the theme is shown in Figure 5.3. Each article might have associated text filters (see Section 4.6). Therefore before each article is displayed, its text filter is run. The most recent articles are displayed and the sidebars are rendered. Rendering the sidebars is done in **app/views/helpers/articles_helpers.rb**.

²In addition, `verify` methods may also be invoked before the actual `index` method. `Verify` methods are similar to filters except that they are used mostly for assertions that should not be false. There are no verification methods that are invoked for this particular request.

³At the moment, there is only **one** blog object since Typo cannot support multiple blogs in the same installation. This might change in the future.

⁴The rationale behind the caching is discussed [here](#). Enabling simple caching speeds up rendering time considerably since the filters get invoked on *all* controllers because they are defined on a parent class.

Figure 5.3 Typo theme selection (`app/controllers/content_controller.rb`)

```

1 def theme_layout
2   this_blog.current_theme.layout
3 end

```

- Before the `index` method finally returns, the `after_filters` and `around_filters` are applied.
 - `flush_the_blog_object` Sets the current blog object to `nil` and returns `true`.
 - `Blog.filter` Apparently this filter is never called since there is no `filter` class method in `Blog`.
- The resulting HTML page that has been populated with all the information is finally returned as an HTTP response.

The resulting HTML page looks like Figure 3.0a. The rest of the requests for public pages follow a similar mechanism. The only parts that differ would be the controller that gets invoked based on the URL scheme.

5.3 Requests for administrative pages

Assume that a request comes for `http://www.yourdomain.com/admin`. A *pseudo* sequence diagram is shown in Figure 5.4. There are six main parts when handling a request for the index of the site as can be seen in Figure 5.4.

- The URL is parse following the mapping rules in `app/config/routes.rb`. In our case, a request for `http://www.yourdomain.com/admin` will invoke the `index` method on a new instance of `Admin::GeneralController` which is found in `app/controllers/admin/general_controller.rb`. The relevant line is line 5 in `routes.rb` as shown in Figure 5.2
- Referring to Figure 4.7 we notice that `Admin::GeneralController` is a subclass of `Admin::BaseController` which itself is a subclass of `ApplicationController`. Both the parent and *grandparent* classes define filters. The following filters are invoked before the `index` method:

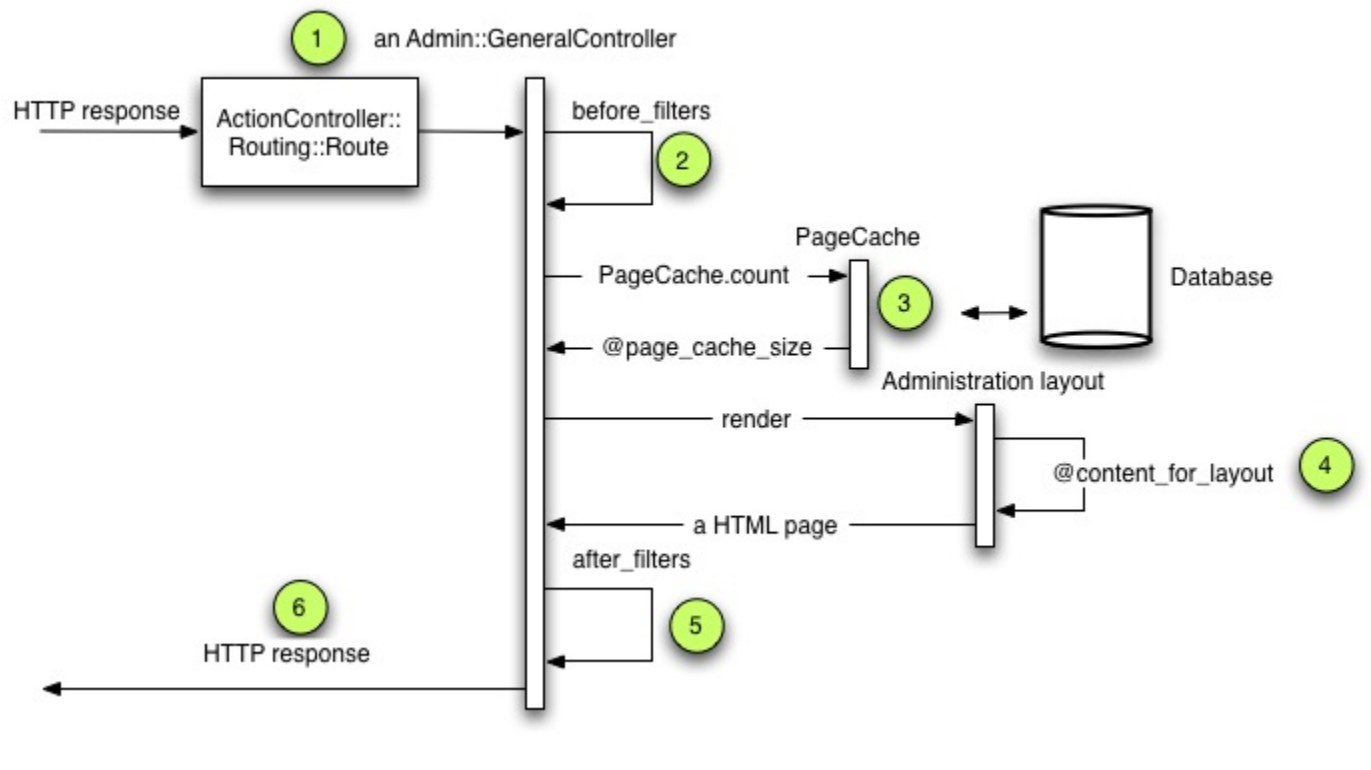
`get_the_blog_post` Returns the current blog and its settings⁵.

`fire_triggers` Removes any pending triggers (see Section 4.2) that have expired and returns `true`.

`login_required` Since this is an administrative page, users must login before they can view or modify the settings. Therefore, the `login_required` filter is run. The code for logins is implemented in `app/lib/login_system.rb` as is mixed-in. For more information on Mixins in Ruby, refer to [6]. The actual authentication is done in the `User` class.

⁵At the moment, there is only **one** blog object since Typo cannot support multiple blogs in the same installation. This might change in the future.

Figure 5.4 A *pseudo* sequence diagram of administrative page request



`look_for_needed_db_updates` Checks to see if the current schema version is up-to-date with the version corresponding to this installation of Typo. If it is not, then perform an upgrade on it first using migration scripts in `app/db/migrate`. Schema updates are usually performed using a rake task (see Section 6.1) but in the event that the user forgets to run the rake task when upgrading to a new version of Typo, calling this method ensures that the database is upgraded to the correct version.

`Blog.filter` Apparently this filter is never called since there is no `filter` class method in `Blog`.

3. After the filters have been invoked, the `Admin::GeneralController` object calls the `count` class method on `PageCache`. `PageCache` is Typo's own implementation of a cache system for web pages (see Section 4.2). `PageCache` gets the information from the database. `Admin::GeneralController` gets the number of pages currently on cache. This number is presented to the user when the final page is rendered so the user can choose to empty the cache if necessary.
4. For administrative pages the theme chosen resides in `app/views/layout/administration.rhtml`. There is no support for loading different themes for the administrative pages. The contents for the page is constructed from `app/view/admin/index.rhtml`.
5. Before the `index` method finally returns, the `after_filters` and `around_filters` are applied.

`flush_the_blog_object` Sets the current blog object to `nil` and returns true.

`Blog.filter` Apparently this filter is never called since there is no `filter` class method in `Blog`.

6. The resulting HTML page that has been populated with all the information is finally returned as an HTTP response.

The resulting HTML page looks like Figure 3.0b. The rest of the requests for public pages follow a similar route. The only parts that differ would be the controller that gets invoked based on the URL scheme.

5.4 Discussion on Typo connection-and-components

As the two examples (Section 5.2 and Section 5.3) illustrate, the handling of different requests to our weblog can be pretty complicated. Even though there is only one controller that is being called to handle the request, that controller might inherit multiple filters from its parent class that needs to be called. These filters might need to construct additional objects as well. All these interactions between classes required careful analysis so that they do not break.

There are also times when the performing those filters, while *unharmful* is completely redundant. Filters defined in the base class for all controllers, `ApplicationController` are run for all subclasses even those that do not need those filters.

In fact, this is the most complex part of the Typo system. Fortunately, there are tests written to verify that the controllers are indeed doing the right thing. These tests help new developers check that they did not break any of the underlying features.

Discussion on Typo connection-and-components

In this chapter, we have also ignored the mechanisms for AJAX calls. AJAX calls are similar to what we have discussed here except the page is updated dynamically without the user being aware of an HTTP request. For more information on how AJAX works in Rails, refer to [7]. Adding AJAX features should only change the way the controllers and views are setup; it should not change the domain logic in the mode.

Chapter 6

Typo Allocation Viewtype

6.1 Typo directory system

There is only one view for the Allocation Viewtype: the Implementation View. In the Implementation View, we describe how the various Typo modules are mapped onto the file system.

Because Typo is built on Rails, it follows the conventions of that framework. These conventions help minimize the configuration files that are needed to get Rails to behave properly. Following these conventions is part of the *convention over configuration* [1] mantra.

Rails uses a standard directory structure for organizing its files. It is important that applications using Rails obey these conventions since Rails uses the directory structure to look for the necessary files; misplaced or missing files will cause the framework to abort its operations.

The directory structure for Typo is shown in Figure 6.1. The blue icons represent directories.

What follows is a description of the files and subdirectories that go into each directory.

app Model, View and Controller files go in subdirectories of **app**.

bin Contains the Typo installer.¹

components Contains reusable components such as plugins and sidebars. Components allow the sharing of code between different views (see Section 4.4).

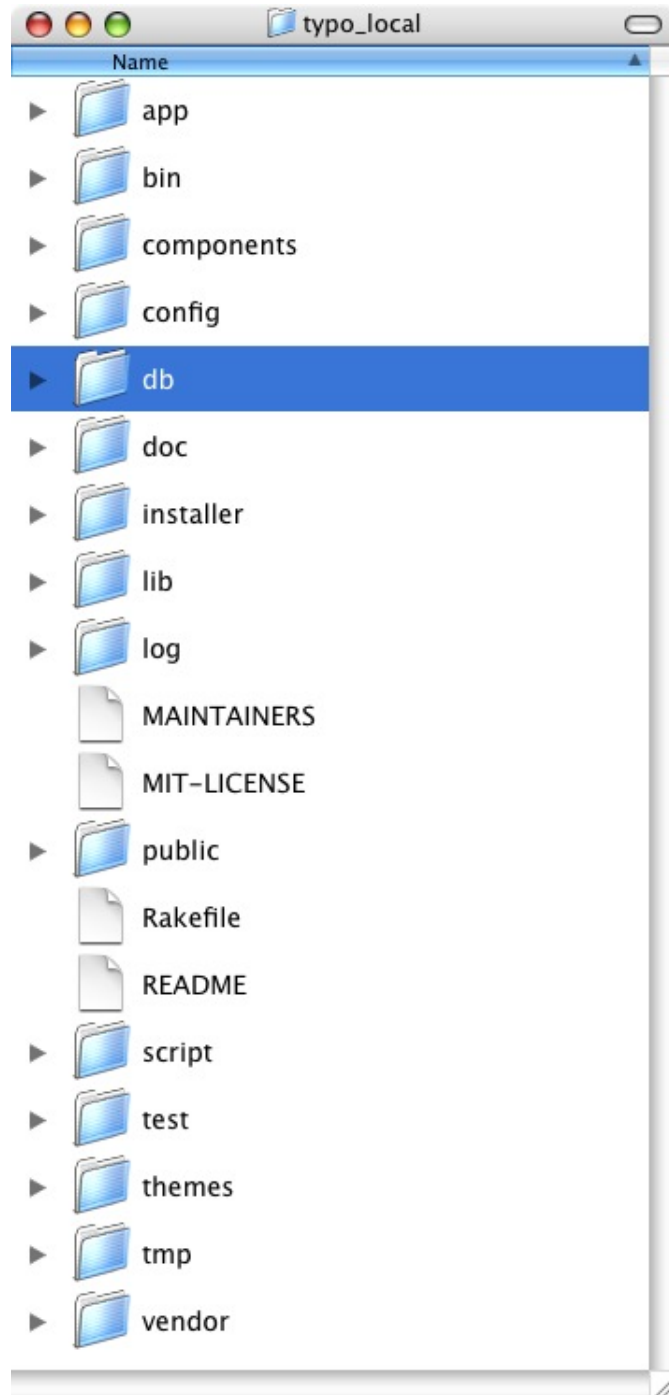
config Contains configuration files. Connection parameters to databases and URL routing information can be found in the directory.

db Contains the schema for various database implementation such as MySQL, PostgreSQL and Sqlite. Migration scripts for upgrading to a different version of Typo also go in this directory. There are also scripts for switching from a different weblog system to Typo. Currently scripts are available for MovableType 3.x, Textpattern 1.x, WordPress 1.5x-2.0 as well as plain RSS.

doc Documentation directory. Currently only contains an installation guide.

¹Uses the rails-installer gem to create an installer for Typo. For more information, refer to [Rails Application Installer](#).

Figure 6.1 Typo Directory Structure



- installer** Contains information for installing Typo using ruby gem². Templates for setting up various web servers are also included.
- lib** Contains shared code written by Typo developers. The files here are used in various other classes.
- log** Contains log files produced by the running Typo in different modes: development, production and test.
- maintainer** Contains contact information of active developers for Typo.
- public** Contains the static contents such as html, css, js and image files. This is the *root* directory of your application i.e. files are served as if they reside in this directory.
- Rakefile** Contains tasks for running tests, upgrading databases, etc.
- Script** Contains utility scripts for benchmarking, debugging, initializing the web server, etc.
- test** Contains unit tests, functional tests, mocks and fixtures.
- themes** Contains the themes for changing the look and feel of a weblog. New themes have to be placed in this directory for Typo to detect them.
- cache** Contains the cache, sessions and sockets that Typo uses. Typo stores its sessions in a database since this is more efficient than relying on the server to store it on a flat file system.
- vendor** Contains reusable code and utilities written by third-party developers. Currently has the third-party text filter utilities for formatting blog posts. A full installation of Rails can be included here to freeze the application with a particular version of Rails.

The **app** directory contains the main files that Typo uses for its Model-View-Controller implementation.

- apis** Contains files that implement Action Web Services in Rails. This is the back-end needed to support posting from desktop clients. Please refer to Section 4.5 for a detailed description that each part plays.
- controllers** Contains the various controllers for Typo. See Section 4.3
- helpers** Contains the helpers for the views in Typo.
- models** Contains the models for Typo. See Section 4.2.
- views** Contains the .rhtml and .rxml files for generating the views. See Section 4.4.

²For more information on how this works refer to [Typo Installer](#).

Figure 6.2 Detailed view of the **app** directory



Chapter 7

Conclusion

We have looked at three different viewtypes for Typo. Hopefully, after reading this document, a developer would be more acquainted with Typo and its architectural design.

This document is not a detailed description of Typo (nor does it attempt to fulfill that role). A lot more information can be gleaned from reading the source code and picking up a book on Rails to find out the advanced features that can be used.

This documentation effort would have been *simpler* if there were more comments in the source code. Currently, comments are sparse and it can be hard to figure out what each method actually does. Certain connection-and-connectors views have been skipped because there was not enough information to verify the validity of that view. Having *wrong* documentation is as bad as having no documentation at all.

Moreover, the Rails framework is constantly evolving. Typo does not always follow the latest version of Rails. Things have been known to break with Typo when Rails gets updated. When Rails upgraded to version 1.1.1, Typo 2.6.0 broke¹. This is not the fault of the developers; rather it is an unfortunate side effect from the fast-paced evolution of Rails. As newer features are added, older features have to be modified (or even removed); open source developers usually spend more time trying to get their own software to work than trying to reconcile it with a beta version of Rails. Nonetheless, Typo and Rails usually get reconciled fairly quickly.

Typo is still under active development and interested readers should sign up for the [mailing list](#) to monitor the future development of Typo.

¹This issue is discussed on [Riding Rails](#). A quick solution for remedying that situation was also suggested.

Bibliography

- [1] Nicholas Chen. Convention over configuration, November 2006.
- [2] Paul Clements, Felix Bachman, Len Bass, David Garlan, James Ivers, Reed Little, Robert Bord, and Judith Stafford. *Documenting Software Architectures Views and Beyond*. Addison-Wesley, 2003.
- [3] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.
- [4] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.
- [5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [6] Dave Thomas. *Programming Ruby*. The Pragmatic Programmers, 2004.
- [7] Dave Thomas and David Heinemeier Hanson. *Agile Web Development with Rails*. The Pragmatic Programmers, 2005.