Developing an Online Photography Exposition Platform Using Ruby on Rails

Bachelor’s Thesis

Thomas Quaritsch and Michael Unger

Supervisor: Univ.-Prof. Dipl.-Ing. Dr.techn. Frank Kappe

Institute for Information Systems and Computer Media (IICM)

Graz University of Technology

January 22, 2009
Abstract

This Bachelor’s thesis describes the design and implementation of an online platform for presenting and selling professional photographies as digital downloads and prints. Therefore, it gives a brief introduction in how to write web applications using Ruy on Rails and Ajax and then explains the application design as well as various ideas behind the interface design. It also addresses security measures, localization issues and usability concerns. The result of this project, called Archive Direct, is a web application based on modern technologies and techniques which can also be applied to other applications and frameworks beyond Rails.
Zusammenfassung

# Contents

## 1 Introduction
1.1 Archive Direct ........................................... 6  
1.2 Thesis Structure ........................................... 6  
1.3 Authorship ............................................... 7  
1.4 Conventions Used in this Document ......................... 7

## 2 Ruby On Rails
2.1 Rails Principles ........................................... 8  
2.2 Ruby ......................................................... 9  
2.3 Basic Concepts ............................................ 11  
   2.3.1 MVC-Architecture ..................................... 11  
   2.3.2 Database Abstraction ................................ 12  
   2.3.3 REST (Representational State Transfer) ............. 13  
   2.3.4 Testing ............................................... 15  
   2.3.5 Plugins ............................................... 17  
   2.3.6 Environments ......................................... 17  
2.4 Components ............................................... 18  
   2.4.1 Active Record ........................................ 18  
   2.4.2 Action Controller ..................................... 19  
   2.4.3 Action View .......................................... 20  
   2.4.4 Action Mailer ........................................ 20

## 3 AJAX
3.1 The XMLHttpRequest Object ............................... 23  
3.2 Implementations and Browser Support ....................... 23  
3.3 Ruby On Rails and AJAX Integration ......................... 24  
3.4 AJAX Libraries and Incompatibilities ....................... 24

## 4 Application-Design
4.1 Models and Relations ..................................... 25  
   4.1.1 Image Properties .................................... 27  
   4.1.2 Image Files and Prints ............................... 30  
   4.1.3 Users .................................................. 31  
   4.1.4 Orders, Payments and Downloads ....................... 34  
   4.1.5 Pricing Model ........................................ 37  
4.2 Security ................................................... 38  
   4.2.1 User Authentication/Authorization ..................... 38  
   4.2.2 Securing Database Access ............................. 41  
   4.2.3 SQL Injections ........................................ 41  
   4.2.4 Input Validation ....................................... 41  
   4.2.5 Logging Sensitive Data .............................. 42  
   4.2.6 Transmission of Sensitive Data Using Secure Channels (HTTPS) .... 43  
   4.2.7 Cross-Site Scripting (XSS) ........................... 43  
   4.2.8 Cross-Site Request Forgery (CSRF) .................... 44  
   4.2.9 Application Code Leakage ............................. 44
1 Introduction

When the idea came up to develop an online photography exposition platform, offering the ability to buy an image and download it immediately after its purchase, the first thing we were sure about was, that it should be written in Ruby On Rails. This is a modern framework for developing interactive web applications in a structured and enjoyable way. Furthermore, it offers an integrated support of Ajax.

Ajax? That’s an innovative way of using JavaScript, impressively demonstrated by the likes of Google Maps¹ or Apple². Ajax can help to make a site usable and feel like an installed application. This is especially useful as this application should be a highly interactive one.

1.1 Archive Direct

Called ArchiveDirect³, our application provides two main features. First, it works as an exposition platform for a photographer. Images can be presented there, and customers can have a close look at the portfolio.

Second, images from the archive can be bought online. It should offer a main source for regional and international images. Images can be bought either as downloads of files in various resolutions, instantly available after its purchase, or as prints directly shipped to the customer. The possibility of instant downloads should offer web designers, magazine publishers or even private persons a stock of images they can use when they are searching for an image.

By using modern web technologies as well as by creating an intuitive, recognizable and fancy but minimalistic design, but especially by providing an optimal presentation for impressive images, ArchiveDirect should be set apart from the wide range of online photo archives.

1.2 Thesis Structure

This thesis consists of a theoretical part, consisting of Sections 2 (Ruby on Rails) and 3 (Ajax), which give the reader a short introduction into the used techniques Ruby on Rails and Ajax.

Sections 4 (Application Design), 5 (Site Design and Navigation) and 6 (Payment Solutions) present and document the project implementation from a technical as well as an optical- and usability-design point of view.

The thesis is concluded by Section 7 (Lessons Learnt), which tries to document a few experiences we had when planning, designing and implementing the project in order to improve for the next one.

¹http://maps.google.com/
²http://www.apple.com/
³http://archivedirect.com
1.3 Authorship

Thomas Quaritsch wrote Sections 2.1, 2.2, 2.3, 4.1 (except 4.1.5), 4.2, 4.3, 4.4 and 7. Michael Unger wrote Sections 2.4, 3, 4.1.5, 5.1, 5.2, 5.3, 5.4, 5.5 and 6.

1.4 Conventions Used in this Document

A comment sign (#) behind a line marks the output value of a code line, e.g.

```ruby
puts "Hi!"            # Hi!
```

An additional double right arrow marks the return value of a code line, e.g.

```ruby
5.class              # ⇒ Fixnum
```
2 Ruby On Rails

Ruby on Rails is a programming framework for creating web applications. As the name suggests, there are two major parts involved: the framework itself (often only called “Rails”) and the programming language Ruby. Ruby is a scripting language invented by the Japanese programmer Yukihiro “Matz” Matsumoto in 1995. On top of that language, David Heinemeier Hansson created the open source framework Rails. But he did not originally intend to create a framework at all, instead, Rails is an extraction from a live application called “Basecamp”[^4], a project management application from 37signals[^5]. Rails was first released in 2004 and is nowadays developed and maintained by a handful of core developers and many, many contributors.[MO08]

The most recent version of Rails is 2.2.2 at the time of writing. Because Rails is still fairly young and web technologies evolve very quickly, it is still heavily under development and constantly enhancing.

There are a lot of web application frameworks for other languages. But what is so special about Rails? It lets you write well-structured applications with fewer lines of code. “Follow the conventions, and you can write a Rails application using less code than a typical Java web application uses in XML configuration.”[TH07]

Rails is built upon two main principles:

- Don’t Repeat Yourself (DRY)
- Convention over Configuration

2.1 Rails Principles

Don’t Repeat Yourself. DRY means, that every piece of information or functionality should exist only once inside your application. This not only reduces the lines of code, it also helps on changes and refactoring if you do not have to dig around in your files (which can be thousands) and find all occurrences of a code piece. DRY is similar to the “third normal form” technique you may know from database organization.[TH07]

Convention over Configuration. Rails has implicit default configurations for just about everything. That starts with naming schemes (e.g. a model class called Post assumes that its database records are stored in a table called “posts”[^6]) over the way you structure your files (Rails automatically creates a project tree with all necessary directories) and ends with assumptions about additional software you use (e.g. version control). [TH07]

[^4]: http://www.basecamphq.com
[^5]: http://www.37signals.com
[^6]: The functionality for deriving such plural names (pluralization) and the opposite, singularization, has even some “magic”, so that ‘Person’.pluralize gives “People” or ‘Sheep’.singularize retains “Sheep”.

---

8 / 70
2.2 Ruby

However, most of these conventions can be overwritten easily, if you want to. For example, if you have to support a legacy database with different table names, all you need to do is:

```ruby
class Post < ActiveRecord::Base
  set_table_name 'postings'
end
```

2.2 Ruby

One of the main reasons why Rails is so powerful, is the programming language behind it. As already stated above, Ruby is an interpreted language that is older than Java and PHP but is still relatively unknown.

Here are a few peculiarities and exceptional features of Ruby. They will also help you reading our source code (excerpts).[TH02]

**Everything is an object.** Ruby is a purely object-oriented language in the sense that really *everything* in Ruby is an object. Unlike many other languages, there are no primitives. For example, you can ask the constant number 5 which class it has:

```ruby
5.class # ⇒ Fixnum
```

Despite of this, Ruby supports the procedural programming paradigm besides the object-oriented paradigm. You can just throw a few lines of Ruby code into a file and execute it without defining a class.\(^7\)

**Duck-Typing.** Objects in Ruby are strongly but dynamically typed. That means that every variable has a specific type at any time (i.e. is an instance of a specific class), but it may change its type during its lifetime.

```ruby
a = 5
a.class # ⇒ Fixnum
a = "Foobar"
a.class # ⇒ String
```

Any arguments to method calls are not typed. Instead, the method expects the parameter objects to support specific method calls.

```ruby
def printme(argument)
  puts "The argument is " + argument.to_s
end
printme(5) # The argument is 5
printme("Foobar") # The argument is Foobar
```

\(^7\)Behind the curtains, you are automatically put into an object called `main` of type `Object`.\]
Here, both the Fixnum class and the String class support the `to_s` method which returns a string representation of an object.

**Blocks and Iterators.** You can pass a code block to a method which then can be called by it, even with parameters. This enables you to write iterators, for example.

```
[1, 2, 3].map { |i| i+5 } # ⇒ [6, 7, 8]
```

The method `map` iterates through every element of the array and replaces it by the return value of the block, which is delimited with curly braces (or with `do/end` alternatively). The bars inside the block specify the block argument variable.

**Semicolon.** As you might have noticed, statements need not be terminated using a semicolon if there is a new line at the end. But you can put two statements in one line separated by a semicolon.

**Optional braces.** Braces around method arguments can be omitted if there is no ambiguity. The following two calls are completely equivalent:

```
"Foobar".sub(/Foo/, 'Tea')
"Foobar".sub/Foo/, 'Tea'
```

The `sub` method replaces parts of a string matched by a regular expression and takes the expression and a replacement string as arguments. While braces contribute to readability in most cases, methods like `puts` are typically called without them.

**Symbols.** Rails extensively uses so-called Ruby symbols. They are prefixed with colons and are some sort of constants\(^8\) that are used mostly as key values of hashes, e.g.:

```ruby
parameters = Hash.new
parameters[:action] = "commit"
link_to "Click Here", parameters[:action]
```

**Instance and class variables.** Local variables are not prefixed in any way (e.g. `aVariable = 5`). Instance and class variables are prefixed using one and two `@`-symbols, respectively. Examples:

```ruby
@anInstanceVariable = 47
@@aClassVariable = 11
```

**No public attributes.** All class attributes are private. Access is possible only through setters and getters. Example:

---

\(^8\)Strictly speaking, a symbol is the internal representation of a name. The advantage of symbols, beside the fact that `:action` is one key-stroke shorter than `'action'`, is that every occurrence of a symbol returns the same object. The string constant `'action'`, however, creates a new object each time.
class Address
  def firstname
    return @firstname
  end
  def firstname=(new_firstname)
    @firstname = new_firstname
  end
end

This can be abbreviated using:

class Address
  attr :firstname
end

You can now use this class and instantiate objects like this:

me = Address.new
me.firstname = "Thomas"
puts me.firstname  # Thomas

Methods ending with ! and ?. The characters ! and ? can be the last character of a Ruby method name. This feature is used to symbolize the destructive behavior of a method (e.g. destroy!) or a querying method that returns a boolean (e.g. valid?)

Of course, Ruby also supports concepts like Class inheritance and Exceptions. However, it does not support multiple inheritance directly. Including multiple blocks of functionality into different classes can be done using modules and so-called mix-ins. [TH02]

BTW: You can try Ruby here (in your browser!): http://tryruby.hobix.com

2.3 Basic Concepts

2.3.1 MVC-Architecture

The main architecture of Rails is based on the MVC (Model, View, Controller) design pattern. This means you cannot create any Rails application that does not follow this concept, instead, Rails helps you by creating all necessary directories where you can put your models, views and controllers into.[TH07]

The sequence of actions that is needed to handle a request inside the MVC architecture is depicted in Figure 1. It contains four major steps:

1. The request from the browser is routed to a controller.
2.3 Basic Concepts

2.3.1 Controller

Controller

View

Model

Database

Browser

1

2

3

4

Figure 1: Handling of an incoming HTTP request by the components of the MVC architecture.[TH07]

2. The controller works with one or more model objects, which are backed by the database.

3. The controller renders a view, which can also use information from the models.

4. The rendered view is returned to the user’s browser.

In a Rails application, multiple controllers, models and views exist. But usually only one backend database is used.

2.3.2 Database Abstraction

Rails uses ORM (Object-Relational Mapping\textsuperscript{9}) to work with SQL\textsuperscript{10} databases. Each Rails model class represents a database table and each model object represents one line inside this table. Every column in the table automatically creates an attribute in the model. This way you do not need to manually write SQL queries any longer. You just use the find class method of the model to retrieve an object (a line) from the database, read and write the object’s attributes and save any changes back.[TH07]

\begin{verbatim}
billing_address = Address.find_by_lastname("Maier")
billing_address.lastname = "Meier"
billing_address.save
\end{verbatim}

This concept is continued for relations between records. For example, if a table “posts” has a column “forum_id” and you create the following model classes:

\textsuperscript{9}\url{http://de.wikipedia.org/wiki/Object-Relational_Mapping}
\textsuperscript{10}According to the Rails Wiki (\url{http://wiki.rubyonrails.org/rails/pages/DatabaseDrivers}), there is currently support for eleven database systems, including MySQL, PostgreSQL, SQLite, Microsoft SQL Server, IBM DB2 and Oracle.
class Post < ActiveRecord::Base
  belongs_to :forum
end

class Forum < ActiveRecord::Base
  has_many :posts
end

then you can access the associated models as attributes:

```ruby
p = Post.find(47)     # finds an arbitrary posting
f = p.forum           # returns associated forum
puts f.title
```

On the other side, you can also access the list of postings from the forum model and even create and destroy objects (=records, table rows) using this association.[TH07]

### 2.3.3 REST (Representational State Transfer)

A Rails controller consists of one or more actions (= methods) that can be called from the “outside”. Which controller and which action is bounded to a specific URL is decided by the routing mechanism.

**The old way.** In the classical Rails routing mechanism, each URL consisted of three sections, which specify the controller to call, an action inside the controller and one optional parameter.

```
http://www.example.com/<controller>/<action>/<id>
```

So, the URL `http://www.example.com/users/show/23` invokes the “show” action in the “UsersController”, which then displays the user with the ID 23. The controller could possibly look like this:

```ruby
class UsersController < ApplicationController
  def show
    @user = User.find(params[:id])
    render :action => "show"
  end
end
```

Similarly, there are typically actions called `new`, `create`, `edit`, `update` and `destroy` to enable the typical CRUD (create – retrieve – update – delete) actions as well as an `index` action to

---

11 More parameters are possible using the `?param1=value1&param2=value2&...` syntax

12 You might wonder why we need both `new` and `create` which seem to serve the same purpose. But the first only displays the form where the user enters the data for the new record which is then POSTed to the second action to eventually create the record in the database. The same applies to the `edit` and `update` actions.
display a list of records. Those actions are called either by HTTP GET (show, new and edit) or HTTP POST (create, update, destroy). [TH07]

**RESTful Rails.** Since Rails 1.2 there is a new way of thinking. It is not about controllers and actions, instead a URL represents a resource, e.g.

http://www.example.com/forums/4711

represents the record with the ID 4711 from the forums table. The action that should be executed on this resource is decided from the used HTTP verb. [WB07]

<table>
<thead>
<tr>
<th>HTTP verb</th>
<th>URL</th>
<th>Action</th>
<th>non-REST URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/forums</td>
<td>index</td>
<td>GET /forums/index</td>
</tr>
<tr>
<td>POST</td>
<td>/forums</td>
<td>create</td>
<td>POST /forums/create</td>
</tr>
<tr>
<td>GET</td>
<td>/forums/4711</td>
<td>show</td>
<td>GET /forums/show/4711</td>
</tr>
<tr>
<td>PUT</td>
<td>/forums/4711</td>
<td>update</td>
<td>POST /forums/update/4711</td>
</tr>
<tr>
<td>DELETE</td>
<td>/forums/4711</td>
<td>destroy</td>
<td>POST /forums/destroy/4711</td>
</tr>
<tr>
<td>GET</td>
<td>/forums/new</td>
<td>new</td>
<td>GET /forums/new</td>
</tr>
<tr>
<td>GET</td>
<td>/forums/4711/edit</td>
<td>edit</td>
<td>GET /forums/edit/4711</td>
</tr>
</tbody>
</table>

Using REST, you can standardize URLs, use more features provided by HTTP and clean up your controllers (the actions are always the same and can be sourced out). As you can see from the table above, the new and edit actions that present the input forms for a resource require special handling again.

**Nested Resources.** The best thing about RESTful resources is nesting. You can easily work with URLs like

http://www.example.com/forums/4711/postings
http://www.example.com/forums/4711/postings/238
http://www.example.com/forums/4711/postings/238/attachments
http://www.example.com/forums/4711/postings/238/attachments/23
...

which can be continued arbitrarily. In the show action of the AttachmentsController from the last example above, you not only have the ID of the attachment, but also those of the enclosing posting and forum record available.

So RESTful routing gives you meaningful and memorable URLs and simplifies your controllers. [WB07]

---

13In this context a HTTP request method is called “verb”, because it specifies which action should be executed on a resource.
2.3.4 Testing

Rails has a built-in testing framework that enables you to thoroughly test each and every aspect of your application. Ideally, coding a Rails project should be done using TDD (test-driven development) and the classical red – green – refactor cycle. This means that you should first write your test, check that it fails ("red"), then correct the code until it works ("green") and at the end maybe refactor. There are also tools that support you by constantly executing your tests in the background while you are coding, e.g. ZenTest\(^\text{14}\).

Testing in Rails has several aspects:

**Fixtures.** For testing you typically need some test data in your database so that you can verify your code works. In Rails, the test data is provided by so-called fixtures. For each table you create one fixture file containing the records in YAML\(^\text{15}\) format:

```ruby
germany:
  name_en: Germany
  name_de: Deutschland
  code: DE

italy:
  name_en: Italy
  name_de: Italien
  code: IT

...
```

Each fixture record has a name and values for the table columns. Fixtures get loaded into the database automatically when tests are executed. At the end of each test the changes are discarded by either rolling the database back or reloading the fixtures.

**Unit Tests.** Unit tests are used for testing your models. All you have to do is to inherit your test case from the provided standard test class and write a method that starts with `test_`.

```ruby
class CountryTest < ActiveSupport::TestCase
  def test_should_require_name
    c = Country.create(:name => nil)
    assert !c.valid?
    assert !c.save
  end
end
```

\(^{14}\)http://www.zenspider.com/ZSS/Products/ZenTest

\(^{15}\)YAML ("YAML Ain’t Markup Language") is a data serialization language that can be used to represent structured data in textual form, see http://en.wikipedia.org/wiki/YAML.
Inside the test method you can work with your model objects and assert that they behave as intended, using various given assertion methods.[TH07]

In the example above, it is tested that a country object is not valid and cannot be saved to the database without filling the name attribute.

**Functional Tests.** Functional tests are used to test the controllers in your application. You can use the common HTTP request methods (get, post, delete, put) to simulate a browser posting a request to the controller and then assert that the controller responds with the correct answer.

```ruby
def test_should_confirm_email
  user = users(:aaron)
  get :confirm, :id => user.confirmation_code
  assert_redirected_to '/'
  assert_not_nil flash[:notice]
  login_as :aaron
  assert user.reload.confirmed?
end
```

In the example above, it is tested that a user is able to confirm his e-mail address using a special URL containing his confirmation code (e.g. received by e-mail). Then it is verified that he is redirected to the start page after the confirmation is done and seeing a success message. At the end it is asserted that the user's state in the database is really confirmed.

**Integration Tests.** In Rails, integration tests are used to verify that portions of your application work together. Unlike in unit tests and integration tests, where you are focussed on one model or one controller, integration tests can use multiple controllers and models within one test. A typical example of an integration test may be to code down a user story into an executable specification: A user logs into the site, selects an image he likes to buy, puts it into the shopping cart, and checks out via PayPalExpress. Eventually, those are the kinds of tests that really matter – they verify that your site really works. [TH07]

**The Problems.** All this testing is very cool. One might think that writing tests consumes a lot of unnecessary time that you could spend on implementing new features. But tests are a help both while developing an application and when maintaining it. During development they help by forcing you to think about what you want to achieve before writing any code. In the maintenance phase and for refactoring code, they are a relief because they ensure that nothing was damaged by your changes.

The first problem appears as you have hundreds of tests with thousands of assertions and a lot of fixtures. Even if database changes are rolled back after every test (instead of reloading the complete set of fixtures over and over again), they are slow. Even though you can have the test run automatically in the background, waiting a minute for 500 tests to execute can be cumbersome. There are of course approaches to fix this issue by not using fixtures at all, but using fake objects...
(mocks or stubs) instead. These also address the issue that fixtures are often hard to maintain because of their relations.

Another problem is that long stories often produce long integration tests that are boring to write and duplicate code over and over again. So one solution is to provide a so-called Domain Specific Language (DSL) for your integration tests that shortens the tests and makes them more readable.

The continuation of that idea leads to RSpec\textsuperscript{16}, which allows you to specify how your application should behave. RSpec is a tool for Behavior Driven Development (BDD) that eventually allows you to write user stories in plain text! (while retaining special keywords, of course). But all of this stuff is very new in the Rails world and is constantly moving and evolving so think twice before using it for a production application.

2.3.5 Plugins

Rails is highly extensible and customizable using the plugin system. Because of the flexible nature of Ruby\textsuperscript{17}, a plugin can modify every Rails functionality that you can think of. There are thousands of plugins that can be used to extend your models, controllers, views, tests, and whatever you can think of.

2.3.6 Environments

An application typically has different configurations, depending on its current status or where it is executed. For example, during development it is executed on the programmer’s computer while she is constantly changing the code files and trying out the application in the browser. When the application is published, it is run on an internet server and accessed by several users “simultaneously”. To reflect the different needs, Rails features multiple execution environments. By default, the following three environments exist:

Development. In development mode, the application is typically run using the Ruby webserver WEBrick\textsuperscript{18} or Mongrel\textsuperscript{19}. In this mode, changes to most source files are recognized by the framework without restarting the server. This speeds up and eases development a lot. Furthermore, the framework prints extensive backtraces and error messages if something fails. Development mode typically also uses some fake objects or test access data to, e.g., not send real e-mails or do real money transactions when testing the application.

\textsuperscript{16}http://rspec.info/
\textsuperscript{17}Ruby allows you to re-open classes and add methods, overwrite methods, rename methods and even delete methods from it.
\textsuperscript{18}http://www.webrick.org/
\textsuperscript{19}http://mongrel.rubyforge.org/
Production. The production environment is used on the production webserver, where the application is typically run behind a proxying web server (e.g. Apache) using either Mongrel again, by starting a FastCGI service or the new mod_rails plugin\textsuperscript{20} for Apache. As the application code is now static, it is loaded once upon the start of the server and kept in memory afterwards (at least as long as there are requests). This speeds up the request rate as the framework does not need to be reloaded for each request separately. As the user should only see descriptive error messages (if errors ever occur), the framework does not print backtraces but presents her a simple error page. Backtraces can be mailed to the administrator using the “Exception Notification” plugin, see Section 4.3.1 for details.

Test. When running the unit tests, functional tests and integration tests, the “test” environment is used.

As it is often feasible to run the application in different execution environments on the same machine (i.e. using the same MySQL server) with different data, there is a separate database for each environment. These databases are typically called “applicationname\_development”, “applicationname\_production” and “applicationname\_test” and can be configured in config/database.yml. This way, running the unit tests – which loads data from the fixture files into the database – does not destroy the data you use for development and in production mode.

2.4 Components

The Rails framework makes specific tasks simple to work with by providing integrated components. In the following the Active Record, the Action Controller, the Action View and the Action Mailer are explained.

2.4.1 Active Record

For a simple handling of databases, Rails provides Active Record. As object-relational mapping (ORM) describes, data and logic are brought together into a single object. A whole table is covered by a class, a row out of it by an object and the columns are the object’s attributes. Each object is wrapping around a row and contains data, out of the table, as well as the logic to manipulate this data. In Rails, the model component uses Active Record by default. For an average access or manipulation, no SQL is necessary at all, as it enables the developer to work with the data without spending significant treatment to the database. Therefore, the focus can be set on what should be done and not how. Active Record even helps to validate data. If data should be stored in the database, but does not pass the validation, it is refused to save it.

With a minimum of configuration Active Record can be used, as Rails already provides basic defaults. The following example demonstrates how easy the title of an image, found by its ID, can be changed in the database:

\textsuperscript{20}http://www.modrails.com/
2.4 Components

```ruby
class Image < ActiveRecord::Base
end

image = Image.find(521)
image.title = "Eisenberg, Burgenland"
image.save
```

As an application gains complexity, more and more tables are related to each other. Active Record creates high level mappings, out of foreign key relations. This helps to reduce the lines of code significantly. There are three basic relations between tables,

- one-to-one,
- one-to-many and
- many-to-many,

covering all possible cases. For building the database schema, four declarations satisfy the possible relations:

- has_one
- has_many
- belongs_to
- has_and_belongs_to_many

In the case of a many-to-many relation, declared with has_and_belongs_to_many, a join table, containing the related IDs is introduced. This join table is the link between the two related tables. Rails allows to use more than just a pair of foreign keys in this join table when using a model instead. It can then work as a part of the schema containing various data.

The common type of relational databases is expanded by an object-oriented approach. Tables can be inherited receiving the properties and relations. Single-table inheritance, as it is provided by Rails, lets all classes inherited be combined in a single table. The inheritance hierarchy is simply defined in the model classes. [TH07]

2.4.2 Action Controller

For navigating through the site, and interacting with it, incoming requests have to be processed and the reactions created and executed. Therefore, rails provides two components that are working together. The Action Controller and the Action View, together called Action Pack.

The Action Controller processes an request and decides what to do. As the interface between the
model and the view, it provides data from the model and controls the view.

Once a request has to be processed, Rails takes a look at the URL and assigns the appropriate controller for the specific task. A controller can contain actions that are called with additional parameters. Generally, the controller contains no logic manipulating data. It serves only as an interface and manipulations of data are done by the model. [TH07]

2.4.3 Action View

As mentioned above, the Action Controller provides information about what to display. The second part of Rails' Action Pack, the Action View processes this information and is responsible for generating the output.

Usually this is done by rendering a template and providing its output in typical web languages like HTML or XML. A template consists of static code and dynamic content. What kind of dynamic code is used, depends on the template. Rails supports three basic kinds:

- **RHTML** - A combination of static HTML with embedded Ruby commands, handling the dynamic integration of data or control elements, used for generating HTML pages.
- **RXML** - Used for generating output in form of a markup language like XML.
- **RJS** - The kind of template dealing with JavaScript.

[TH07]

Various other kinds of templates can be used, but have to be constructed or integrated separately, like for example the following:

- **PDF** - A template can be used for creating output in form of an PDF file.

Normally, every controller has its subdirectory of the views directory where its templates are stored. As webpages typically share the same elements or content on several pages, Rails provides Layouts. They are templates themselves and are constructions that include other templates, so elements can be written into separate files and just be included when needed.

2.4.4 Action Mailer

Action Mailer is a part of Rails, used for the automatic sending and receiving of e-mails. It can be used for tasks like sending order confirmations or birthday greetings.

The body of an e-mail is typically created by using a view. The parameters concerning the e-mail, like the sender, the time stamp or the receiver are handled in a specific model.
Ajax, is an abbreviation of “Asynchronous JavaScript and XML”, brought a significant change in how modern websites can be built. Unlike in old-fashioned web design, where a page has to be reloaded completely after an user interaction that caused data exchange with the server, Ajax enables data exchange with the server in the background while only parts of the currently displayed page can be updated in place. No refresh is necessary at all.

**Classical Web Request.** Once the user makes an interaction on a classical web page like submitting a form, the data is sent to a server, processed there and the resulting page is sent back to the client, like seen in Figure 2. By sending the new site to the client also parts of the page that have not been changed are transmitted unnecessarily. This causes long waiting times for the user, as he has to wait for the page to be completely reloaded once he clicked on a button, and therefore sent a request. As web applications become more complex and interactive, waiting times increase and become disproportionally long. [Ray07]

![Classical requests](image.png)

Figure 2: Classical requests.
**Ajax Request.** As web applications become more user-interactive, Ajax offers a possibility to make them feel kind of client-sided. Once a request is sent out, the example of a form submission can be taken once again, there is no strict server client communication that leads to a page reload.

As the Ajax-based client is split up into two parts, the *User Interface Layer* handling the direct presentation of the page to the user, and the *Ajax Layer* as an intermediate connection layer. Interactions on the user interface are not obligatorily sent to the server. The *User Interface Layer* is separated from the network by the *Ajax Layer*. This handles requests different, as he can process some of them himself without any server interaction, or he sends and requests data really necessary for the specific task. This causes less overhead than it would have in the classical way. By doing this in the background, the user can continue interacting with the current page, and once the request is completed, only parts of the page have to be renewed. A flowchart of an Ajax-based request can be seen in Figure 3. [Ray07]

![Ajax Request Diagram](image)

Figure 3: Requests using Ajax.

Ajax’ increasing popularity is based on the fact, that pages do not have to be reloaded completely but only parts of them may have to be updated. This makes highly interactive webpages more reactive and therefore feel more like installed client applications, but with the advantages of a server-based site. As JavaScript is available for all platforms and supported by the majority of modern web
browsers, it is easily accessible and does not require any additional plugins.

Since Ajax offers an incredible range of new possibilities, also an adaptation of the design process is necessary. Attention has to be paid to new problems like the difficulty of using a browser’s “Back” button, as typically only static sites are stored, or the lacking optical response to a user’s click that he is used to. As there is no response like the page reload to an interaction, the designer of a page has to take care to give the user feedback and to show him that something is happening in the background. This can be done by waiting symbols like a spinning wheel or messages.

The power of this revolutionary technology can be experienced on a wide range of popular websites. Examples of how Ajax can be used in an very immersive way can be seen at Google’s application Google Maps or at the webpages of Apple\textsuperscript{21}, the NASA\textsuperscript{22} or the one of an Ajax library script.aculo.us\textsuperscript{23} itself.

### 3.1 The XMLHttpRequest Object

Originally a development from Microsoft, the XMLHttpRequest object is supported by a majority of modern web browsers. It forms the base of Ajax and allows the asynchronous exchange of data. The XMLHttpRequest object offers the possibility to send and receive various data using the HTTP protocol. Various methods for transferring are provided. The basic idea is to create an HTTP request and to send it to the server that is processing the request. In the meantime, the client can perform other tasks and does not have to wait for the result. If needed, a result can be fetched from the server. [Wen06]

### 3.2 Implementations and Browser Support

As the XMLHttpRequest object and JavaScript are supported by a majority of modern web browsers, one could easily think, that at least by using Ajax platform issues should not rise. But as web developers are used to handle different interpretations of HTML and CSS in different browsers, they have to deal with the same kind of problem when using Ajax.

This starts while instantiating the XMLHttpRequest object, since not all browsers support it natively, various versions for different browsers have to be written. As standardization and the implementation of them advances, problems become less, but today there are still a huge amount of browser specific tools, features and different interpretations. But once a web application should be accessible over multiple platforms and different versions, attention has to be paid.

Using frameworks can simplify the process of writing multi platform compatible code, as they provide higher level Ajax services. For building this application, the frameworks Prototype and script.aculo.us were used. They provide functions already suitable for the use with different platforms. For example

\textsuperscript{21}http://www.apple.com

\textsuperscript{22}http://www.nasa.gov/

\textsuperscript{23}http://script.aculo.us/
the XMLHttpRequest object does not have to be instantiated by hand as the framework handles this, taking care of the specific platform. They support the best part, but not all web browsers in use today. Although the usage of this library is the best alternative, as writing code working equivalent on all browsers would cause disproportional overhead.

### 3.3 Ruby On Rails and AJAX Integration

Ruby on Rails makes the development of Ajax even more simple, as the Rails framework already supports the use of Ajax. This is done by the integration of Prototype and script.aculo.us into Rails, as well as by Rails helpers generating JavaScript code, so that there can be no need to write any JavaScript code at all.

### 3.4 AJAX Libraries and Incompatibilities

Prototype and script.aculo.us are typical frameworks used for web applications in combination with Rails. The fundamental Prototype provides a wide range of JavaScript functions like handling the XMLHttpRequest or dealing with the DOM. Built on top of it, script.aculo.us focuses on the interface as it provides easy and fast-to-use visual effects and GUI elements.

Due to the seamless integration, and the original intent for the use with Rails, Prototype and script.aculo.us were used to create our webpage. They offer an astonishing amount of possibilities. Attention has to be paid by using any further JavaScript framework, as names often overlap and problems occur.
4 Application-Design

This section describes the technical design of the Archive Direct site, that means:

- Which models have been created, what tasks do they have and how are they related to each other?
- Which security concerns have been taken into account to prevent accessing model objects for unauthorized users?
- Which plugins have been used to provide the functionality and how do they work?
- Which approach has been taken to localize the site for different languages?

4.1 Models and Relations

Figure 4 shows the entity-relationship diagram of the Archive Direct application. The diagram is a mixture between a class diagram and a database layout diagram. The following conventions are used:

- Each box corresponds to a model class. The name of the model class is the singular name of the box title (e.g. an Orders box corresponds to the Order class).
- Arrows with unfilled, triangular heads denote that one class is derived from the class where the arrow points to (like in the UML syntax).
- The simple arrows (→) show the relations between the classes, which are typically one-to-many (1:N) unless otherwise noted.
- Following the Active Record convention, every class that is not derived from another class corresponds to a database table\textsuperscript{24}. From the database point of view, the simple arrows symbolize “references” relations. Thus, the arrow always points from the table containing the foreign key column to the referenced tables’ column (mostly its primary key).
- The lower section of a box lists the table columns together with its data type and thus the model’s database-backed attributes. When the column list is not present, the attributes are either the same as they are in the parent class (since classes saved in the same table always share the same attributes) or are simply left out for the diagram.
- An underlined column is part of the primary key (mostly the id column).

Notes:

\textsuperscript{24}Active Record uses Single-Table Inheritance when deriving classes.
Figure 4: Entity-Relationship diagram of the Archive Direct application.
The entity-relationship model of the *Archive Direct* application was designed to be in third normal form wherever it is possible and feasible.

Some tables contain columns named `column_de` and `column_en`. These columns contain the attribute `column` in German and English language as they are the two languages supported by the *Archive Direct* application for now.

Because the entity-relationship diagram is very large we will split it up and look at various logical sections separately.

### 4.1.1 Image Properties

Figure 5 shows the model `Image` and models that are used to save properties of an image into the database.

- **Categories**
  - `id`: int
  - `name`: text
  - `lft`: integer
  - `rgt`: integer
  - `parent_id`: integer

- **ImageCategories**
  - `image_id`: int
  - `category_id`: int

- **Image**
  - `id`: int
  - `location_id`: int
  - `date`: datetime
  - `image_format_id`: int
  - `description`: text
  - `photographer_id`: int
  - `hdr`: boolean
  - `price_class_id`: integer
  - `blackandwhite`: boolean
  - `associated_image_id`: integer
  - `visible`: boolean

- **ImageTags**
  - `image_id`: int
  - `tag_id`: int

- **ImageFormats**
  - `id`: int
  - `description`: text

- **Locations**
  - `id`: int
  - `name`: text
  - `region`: text
  - `country_id`: int

- **Countries**
  - `id`: int
  - `name`: text
  - `shipping_zone_id`: integer

- **Photographers**
  - `id`: int
  - `name`: string

- **PriceClasses**
  - `id`: int
  - `name`: string

*Image* is the main model that represents a picture in the *Archive Direct* application. It collects all print versions and downloadable files of a picture, independently of their size or type. The models in Figure 5 are explained by looking at the following `Image` attributes and relations:

- **date**: the date and time on which the picture was taken as exact as known
- **description**: a textual information that describes what can be seen on the picture
- **hdr**: if the picture was composed using multiple exposures (dynamic range increase, DRI, also called high dynamic range, HDR)
- **blackandwhite**: if the picture consists of gray-values only (true) or if it is a color picture (false)
  
  So an image may be either black/white or color and thus the user needs to create two images if he wants to provide a picture in both a black/white and a color version. To link these two images, the `associated_image_id` column references the color version of a black/white picture if it exists, and vice versa.
- **visible**: if the image is currently visible on the site (true)
4.1 Models and Relations

- **photographer**: the photographer who has taken the picture
  This enables the site to contain pictures of multiple photographers.

- **location**: describes the place where the picture was taken
  A location always consists of three parts: `<Location Name>/Region Name>/Country>`. An example might be: “Burg Güssing / Burgenland / Austria”

- **categories**: lists the categories the image belongs to
  An image may be assigned to multiple categories (which maintain a tree structure themselves). To resolve the many-to-many (N:M) relation, a so-called “join table” and the associated “join model” named `ImageCategory` is created. So for every association from an image to a category, one entry in this join table is created that represents the existence of that association.

- **tags**: lists the tags that have been assigned to an image
  Similar to the categories, an image may be assigned to multiple tags. Those tags “connect” images on another level and enable a text-search on the images, e.g. a picture may be assigned to the tags *Wine, Sun, Grapes, Landscape,* and *Burgenland.*

- **image_format**: the so-called format of an image
  The format specifies different image attributes like dimension or the technique that was used to create it. A typical format may be “6x17”, which specifies that the image was done using an analog medium format camera that uses a film with $6\text{cm} \times 17\text{cm}$ sized negatives. Others may be “6x6” or “3:2”, which is a typical format of digital cameras.

- **price_class**: specifies which price class the image belongs to
  Every image is assigned to one price class. This enables different pricings for different image “qualities”, e.g. an image that had high production costs or has a higher value because of any other reason is assigned to price class “A”; a low-quality image may be assigned to price class “C” by the uploader/photographer. The price class is eventually used to calculate the price of a specific downloadable file.

The corresponding code that represents the relations mentioned above is

```ruby
class Image < ActiveRecord::Base
    has_many :image_categories, :dependent => :destroy
    has_many :categories, :through => :image_categories
    has_many :image_tags, :dependent => :destroy
    has_many :tags, :through => :image_tags
    belongs_to :color_version, :class_name => 'Image', :foreign_key => 'associated_image_id'
    has_one :bw_version, :class_name => 'Image', :foreign_key => 'associated_image_id'
    belongs_to :location, :dependent => :destroy, :include => :country
    belongs_to :image_format, :dependent => :destroy
    belongs_to :photographer
    belongs_to :price_class
end
```
class ImageCategory < ActiveRecord::Base
  belongs_to :category
  belongs_to :image
end

class Category < ActiveRecord::Base
  has_many :image_categories, :dependent => :destroy
  has_many :images, :through => :image_categories
end

class Location < ActiveRecord::Base
  has_many :images
  belongs_to :country
end

class Country < ActiveRecord::Base
  has_many :locations
  belongs_to :shipping_zone
end

class ImageTag < ActiveRecord::Base
  belongs_to :tag
  belongs_to :image
end

class Tag < ActiveRecord::Base
  has_many :image_tags
  has_many :images, :through => :image_tags
end

class ImageFormat < ActiveRecord::Base
  has_many :images
end

class PriceClass < ActiveRecord::Base
  has_many :images
end

class Photographer < ActiveRecord::Base
  has_many :images
end
4.1.2 Image Files and Prints

Figure 6 again shows the model Image, but now focuses on how the files and prints are managed.

The Image model has two more attributes we want to look at:

- **image_files**: the list of downloadable files
  An image may be available in different sizes (resolutions), e.g. the uploader may decide to provide a version with $8858 \times 2917$ pixels, a version with $2464 \times 811$ pixels and a version with $1743 \times 574$ pixels. To save this information into the database, an Image has one or more associated ImageFiles. An ImageFile saves the width and height of the associated file in pixels, the file size in bytes, the mime type (e.g. image/jpeg or image/tiff). Each ImageFile is associated with a FileType to group similar types of files (e.g. all TIFF images with 20 megapixels).

- **image_prints**: the list of prints that are available for this image
  The uploader may specify that the image is available in different sizes or printing methods for ordering. This fact is reflected using the ImagePrints model. Similar to the image files, the PrintType groups similar types of prints (e.g. 60 $\times$ 60 cm on canvas, Fine Art).

Note that every image needs at least five special image files, that should always be present: two differently sized versions of thumbnails (ImageThumbnailSmall and ImageThumbnailLarge, they are used in the search results), ImagePreviewSmall and ImagePreviewLarge, two differently sized versions of previews (so that the user can take a closer look) and the originally uploaded file (ImageOriginal) that is needed to create new downsized versions, for example. These image files are modeled using their own classes derived from ImageFile, because they need special handling, e.g. regarding watermarks.

The FileType model is needed when creating and deleting image files. The uploader may specify, which versions he wants the Archive Direct application to create from an image (e.g. a 20MP (megapixels) TIFF, a 10MP TIFF and a 3MP JPEG version, which corresponds to specifying a list of FileTypes. Later on, one may decide to no longer provide the 3MP JPEG version, delete the corresponding FileType and remove this version from the list of available downloads for all files. Similarly, it is possible to create new versions from all files by creating a new FileType with a specific MP size and a specific mime type. The application is then able to create the down-sized...
versions it needs by using a scaled version of the originally uploaded file. As the original version can have an arbitrary size and mime type, a special `FileTypeOriginal` exists for handling those files.

The code that expresses the relations stated above is:

```ruby
class Image < ActiveRecord::Base
  has_many :image_files, :dependent => :destroy
  has_many :image_prints, :dependent => :destroy
  has_one :thumbnail_small, :class_name => "ImageThumbnailSmall", :dependent => :destroy
  has_one :thumbnail_large, :class_name => "ImageThumbnailLarge", :dependent => :destroy
  has_one :original, :class_name => "ImageOriginal", :dependent => :destroy
  has_one :preview_small, :class_name => "ImagePreviewSmall", :dependent => :destroy
  has_one :preview_large, :class_name => "ImagePreviewLarge", :dependent => :destroy
end

class ImageFile < ActiveRecord::Base
  belongs_to :image
  belongs_to :file_type
end

class FileType < ActiveRecord::Base
  has_many :image_files, :dependent => :destroy
end

class ImagePrint < ActiveRecord::Base
  belongs_to :image
  belongs_to :print_type
end

class PrintType < ActiveRecord::Base
  has_many :image_prints, :dependent => :destroy
end

4.1.3 Users

Figure 7 shows how a user and all user-related data was modeled. This data consists of addresses, permissions, searches and a user’s shopping cart.

The main model for saving user-related data is the `User` model. It has the following attributes and relations:
• **login**: the login name of the user, her “user name”.

• **crypted_password, salt**: The password chosen by the user is salted and hashed using SHA1 and stored to the database.

• **remember_token, remember_token_expires_at**: The user can select that she wants to stay logged in for a specific amount of time. Therefore, a token is saved inside a browser cookie and to the database. When the cookie is available and the token is not expired, she is logged in automatically if she comes back.

• **confirmation_code, confirmed_at, confirmed_email**: A user must confirm his e-mail address before she can actually buy something on the site. Therefore, an activation code is sent to the e-mail address she entered. When the e-mail address is confirmed using the link inside the mail, the time and the confirmed e-mail address is recorded.

• **gender, birthday**: The users’ gender (‘m’/’f’) and birthday (datetime)

• **phone_*, email_*, web, facsimile**: Additional optional contact data that can be provided by the user.

• **salutation, title, firstname, lastname, company**: A user must enter his name and address before she can buy an image. If the user represents a company, the company name is saved in the corresponding column, first name and last name then refer to a contact person of the company.

• **city, street, zipcode, country**: See above.
• **vat_number**: In order to issue a bill without tax to a company, its VAT number is needed.

• **comment**: Just a comment field that may be used by the administrators.

A user can create multiple addresses that may consecutively be used as billing or shipping addresses for orders. An address entry mainly consists of the same columns that are also available in the User model and meaningful for an address record.

Different access levels are distinguished using so-called **Roles**. There might be special roles for administrators, uploaders/photographers, simple users or guests. As a user may have multiple roles, the many-to-many relation between User and Role is again resolved by introducing a join model, Permission.

As a user searches through the *Archive Direct* site, the keywords/categories/colors he uses to narrow his search are stored to the database. This enables the user to use previously configured searches again when he comes back. Technically, a search is represented by the **Search** model, which itself consists of one or more **Filter** entries. Each type of filter (e.g. CategoryFilter, KeywordFilter or ColorFilter) is configured using some data (i.e. the desired category, keyword or color) and outputs some search conditions. The **Search** model then combines all conditions from its filters using a logical AND thus narrowing the search more and more when adding filters to it.

When the user selects an image she wants to buy, it is added to her shopping cart. A user may have at most one shopping cart at a time, represented by the **Cart** model. It consists of multiple **CartItems** which refer to an image file or image print themselves. Furthermore, a user must select a payment type as well as a billing address (and a shipping address in case she buys prints) during the ordering process.

The code that is needed to reflect the relations mentioned before is:

```ruby
class User < ActiveRecord::Base
  has_many :permissions, :dependent => :destroy
  has_many :roles, :through => :permissions
  has_many :searches, :dependent => :destroy
  has_one :cart, :dependent => :destroy
  belongs_to :country
  has_many :addresses, :dependent => :destroy
  belongs_to :default_shipping_address, :class_name => "Address", :dependent => :destroy
  belongs_to :default_billing_address, :class_name => "Address", :dependent => :destroy
end

class Address < ActiveRecord::Base
  belongs_to :user
  belongs_to :country
end
```
4.1 Models and Relations

```ruby
class Cart < ActiveRecord::Base
  has_many :cart_items, :dependent => :destroy
  belongs_to :user
  belongs_to :shipping_address, :class_name => "Address"
  belongs_to :billing_address, :class_name => "Address"
  belongs_to :payment_type
end

class CartItem < ActiveRecord::Base
  belongs_to :cart
  belongs_to :buyable, :polymorphic => true
end

class Role < ActiveRecord::Base
  has_many :permissions, :dependent => :destroy
  has_many :users, :through => :permissions
end

class Permission < ActiveRecord::Base
  belongs_to :user
  belongs_to :role
end

class Search < ActiveRecord::Base
  belongs_to :user
  has_many :filters, :dependent => :destroy
  has_many :keyword_filters, :dependent => :destroy
  has_many :category_filters, :dependent => :destroy
end

class Filter < ActiveRecord::Base
  belongs_to :search
end
```

4.1.4 Orders, Payments and Downloads

Figure 8 shows all models that cope with orders, payments and downloads.

The main model that deals with orders is called **Order**. At the end of the checkout process a new order record is created from the user’s cart. It has the following attributes and relations:

- **user**: the user this order belongs to
- **order_no**: a unique order number
4.1 Models and Relations

APPLICATION-DESIGN

Figure 8: Entity-Relationship diagram of the Archive Direct application – Orders, Payments and Downloads.

- **sum_price**: the price of the order, including shipping costs

- **invoice_pdf**: the order’s invoice in PDF format saved as a binary object. After an order is created, the corresponding invoice is automatically created and saved to this column for the user to download and the photographer to print.

- **date**: the date when the order was created

- **locked**: if the order is locked (so that it cannot be modified any longer)

- **canceled_at, shipped_at**: when the order was shipped or canceled, including the fact that it was shipped or canceled

- **billing_address, shipping_address**: the billing and shipping addresses
  
The addresses from the user’s cart (which are of type Address) are converted to addresses of the type OrderAddress which are then referenced in the order record. This is necessary to keep the order consistent even if the user decides to change or modify the addresses used for the order afterwards.

- **state**: the current state the order is in
  
  An Order acts like a state machine using the acts_as_state_machine plugin and keeps its current state in the state column. The machine’s states and transitions are depicted in Figure 9. Initially, it is in state pending. Depending on the payment process, it might go through the state prepared. Eventually it should be in state paid after the order was paid. If something goes wrong, the state is changed to failed, which might be a temporary state only.

- **payment_type**: the payment type that is used for this order (e.g. Credit Card, Instant Bank Transfer or Paypal)
The payment type determines, which type of transactions is used in the payment process.

- **transactions**: the list of OrderTransactions that were produced while paying this order
  Each payment request issued to a payment gateway is saved as a database record using the model OrderTransaction. To enable tracability of the payment process, every OrderTransaction stores the request data and complete result of a payment request. For every payment gateway supported by the Archive Direct application, a separate subclass of OrderTransaction exists (e.g. OrderTransactionPaypal)

- **order_items**: the list of items this order contains
  Finally, the order needs to save a list of items that were bought. To retain the data of an order even when images on the site are deleted or modified, specific fields of the bought items are duplicated in the list of ordered items. The OrderItem also needs a relation to the image file or image print that was originally put into the shopping cart, so that the photographer knows which image to print or the system knows which download to create. This relation is created using a polymorphic reference, because the referenced item may be either an ImageFile or ImagePrint.

![State Diagram of the Order model](image)

Figure 9: State Diagram of the Order model.

When an Order object enters the state “paid”, it creates Download records for each ImageFile contained in the order. Afterwards, the user is able to download her images for a configurable amount of time from her account page. To prevent blocking the Rails servers through large downloads, downloading is decoupled using a small CGI script written in Ruby.

---

25 This violates the third normal form requirements but is inevitable for this kind of data.
4.1 Models and Relations

4.1.5 Pricing Model

The price of an image is defined by various of its attributes. On the one hand, an image can be purchased as a high quality print, bringing high production costs with it, or as a download as a digital image file.

Downloads. In opposite to the prints, downloads are easy to classify. Every image is assigned to exactly one price class, for example A, B or C. The pricing class in combination with the resolution of the desired image define its price. An illustration of the pricing model can be seen in Figure 10. The number of price classes and of resolution steps is variable. A price has to be assigned to every element of the grid manually.

![File Resolution](image)

Figure 10: Grid for specifying the price of an image. The number of resolution steps as well as the price for every combination can be set manually. All values in the illustration are fictional and serve only as an example.

Prints. Prints are offered to selected images only. They are defined by their size, kind of paper or canvas and their kind of printing. This multitude of different variations brings a wide range of self costs. Therefore, all printing types, each defined by its size, paper and kind of printing, get a price set manually. They can be defined in the System Settings of the Administration Area.
Shipping Costs. The estimation of the shipping costs is divided into three parts:

1. **Estimating package weight**
   The approximate weight of every available print is stored. Once the order is completed, the weights of all items are added. The result is the weight of the complete package. For example (all values used are fictional):
   
<table>
<thead>
<tr>
<th>Print Type</th>
<th>Size</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canvas Print</td>
<td>(100cm x 100cm)</td>
<td>400g</td>
</tr>
<tr>
<td>Glossy Print</td>
<td>(30cm x 21cm)</td>
<td>50g</td>
</tr>
</tbody>
</table>
   
   $\sum 450g$

2. **Determining postage**
   The weight of a package and its destination determine the postage. Therefore, a table consisting of weight intervals and shipping zones is used. For example (all values used are fictional):
   
<table>
<thead>
<tr>
<th>Weight / Zone</th>
<th>1. EU</th>
<th>2. USA</th>
<th>3. Asia</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 2kg</td>
<td>€2.75</td>
<td>€5.24</td>
<td>€17.75</td>
<td>...</td>
</tr>
<tr>
<td>2 - 5kg</td>
<td>€7.50</td>
<td>€12.09</td>
<td>€27.19</td>
<td>...</td>
</tr>
<tr>
<td>over 5kg</td>
<td>€13.25</td>
<td>€25.75</td>
<td>€45.75</td>
<td>...</td>
</tr>
</tbody>
</table>

3. **Adding flat fee for packaging**
   Once the postage was determined, a flat fee for packaging costs is added. The resulting sum are the shipping costs, the customer is charged. For example (all values used are fictional):
   
   | Postage         | €5.24 |
   | Flat Fee for Packaging | €3.00 |
   
   Charged Shipping Costs  €8.24

4.2 Security

When designing and programming a web application, security is a big issue. This is due to the fact that a web application is publicly accessible from everyone around the world at any time. Security for web applications is a multifold area and not every threat can be taken into account. The following subsections show some of the topics that we dealt with while developing the Archive Direct application.

4.2.1 User Authentication/Authorization

The Archive Direct user authentication is based on a plugin called RESTful authentication. When a user sets his password, it is salted, hashed and stored to the database together with the salt value.
The following block of code shows the corresponding code from the User model.

```ruby
# before save
def encrypt_password
  return if password.blank?
  self.salt = Digest::SHA1.hexdigest("--#{Time.now.to_s}--#{login}") if new_record?
  self.crypted_password = self.class.encrypt(password, salt)
end

class << self
  def encrypt(password, salt)
    Digest::SHA1.hexdigest("--#{salt}--#{password}")
  end
end
```

Now a user can authenticate against the system using her login or email and her password. The following block of code shows the corresponding code from the User model.

```ruby
class << self
  def authenticate(login, password)
    u = find_without_filter :first, :conditions => ['login = ? or email = ?', login, login] # need to get the salt
    if u && u.authenticated?(password)
      User.current = u
    end
  end
end

def authenticated?(password)
  crypted_password == encrypt(password)
end
```

This is then used by the SessionsController to authenticate a user if she wants to log in. After successful authentication, the user’s id is stored to the current session. This is needed to log in the user at consecutive requests, i.e. so that the user “stays logged in”. Additionally, the current user object is stored to current_user, so that it is accessible from the controllers and views.

In order to prevent that a (logged in) user accesses data from another user, the default find method is scoped with the current user’s id:

```ruby
def find(*args)
  conditions = if User.current.is_a?(User)
    if User.current.admin?
      nil
    else
```
4.2 Security

This means that unless the currently logged in user has admin rights (the authorization system is explained below), she can only fetch her own user model from the database. Furthermore, since all user-specific sites are nested inside the user, e.g.

http://www.archivdirekt.at/users/4711/orders/34

it is necessary to fetch the user with the given id (e.g. 4711) from the database first and all other models are automatically protected without using a separate find scope on them\(^{26}\). Of course, sometimes the application needs to issue a user find without the scope (e.g. for user authentication, see the code sample above), so an alternative unscoped find method still has to be provided (called `find_without_filter` for the user model).

User authorization\(^{27}\) is based on the user’s permissions and roles explained in Section 4.1.3. The implementation in the Archive Direct application is based on the Rails plugin `acl_system`. For each controller action one can define conditions on the roles of the current user that are necessary to execute the action, e.g.

```ruby
access_control [:new, :create, :update, :edit] => '(admin | user | moderator)',
:delete => 'admin & (!moderator | !blacklist)'
```

specifies that a user must have the admin, user or moderator role in order to execute new, create, update or edit and that she must be admin but neither a moderator or nor a blacklisted user to execute delete.

In the Archive Direct application, currently the authorization specifications are rather simple and mainly used to protect the admin area, e.g.

```ruby
access_control :DEFAULT => "admin"
```

which is used in all Admin: :+ Controllers to protect all methods from non-admin users.

---

\(^{26}\)This does not prevent accessing user-specific data from the Rails Console, but since a user never gains console access this will not be a problem.

\(^{27}\)The term authorization describes the process of verifying which actions a user is allowed to do. In contrast, authentication means verifying that a user is the same that created the corresponding account, by checking the account password for example.
4.2 Security

4.2.2 Securing Database Access

As nearly all information from the application is stored in the MySQL database, securing access to this database is crucial. This can be done by a few simple measures:

- Bind the MySQL server’s socket only to localhost, if the application is running on the same server as the database. Otherwise, if multiple hosts are deployed, connect them using an encrypted private network (e.g. OpenVPN) connection. The MySQL server does not need to be accessed from the outside.

- Use the MySQL grant possibilities to restrict access to the databases. Use different passwords for production and development systems and do not store production passwords to the version control repository.

4.2.3 SQL Injections

SQL injections are the classical vulnerabilities of web applications. If an application uses user-provided strings to construct SQL queries, e.g.

```ruby
User.find(:first, :conditions => "login='#{params[:username]}' and password='#{params[:password]}'")
```

a user may provide a username like `' or 1=1 --` and gain unauthorized access to the application. [TH07]

To protect your application from SQL injections, you must always use bind variables for your queries. In Rails, this is as simple as passing an array instead of a string whenever you use an SQL snippet, e.g.

```ruby
User.find(:first, :conditions => ['"login=? and password=?", params [:username], params[:password]])
```

Rails then takes care of escaping and quoting all the user input properly. [TH07]

4.2.4 Input Validation

In addition to functional input validation, e.g.

```ruby
validates_numericality_of :price
```

which can be used to check the contents of model attributes, one should also check which model attributes are supplied by the user. When creating or updating a model, a controller mostly uses the `new(...)` or `update_attributes(...)` method and passes the `params` hash containing
all parameters received from the input form. Even if the form does not allow the user to enter and thus overwrite sensitive information, she may use a custom form and submit arbitrary parameters with the request.[TH07]

To ensure that a parameter is never set or updated using one of those mass-assignment functions, Rails provides the `attr_protected` method.

```ruby
class User
  attr_protected :confirmed_at
end
```

Or even better than blacklisting may be to use whitelisting, i.e. specify, which attributes may be mass-assigned, and have all others discarded. The following snippet shows code used in the User model of the Archive Direct application.

```ruby
class User
  attr_accessible :login, :email, :password, :firstname
  attr_accessible :lastname, :city, :zipcode, :company
  attr_accessible :street, :country_id, :country, :title
  attr_accessible :remember_token, :remember_token_expires_at
  attr_accessible :salutation
end
```

### 4.2.5 Logging Sensitive Data

A Rails application usually logs all requests to files located under `log/`. When you take a look at this log file, you might encounter sensitive data logged to those files, e.g.

```
Processing SessionsController#create (for 127.0.0.1 at 2008-09-14 16:42:27) [POST]
Session ID: BAh7CjoSYWN0[....]
Parameters: {"commit":"Log in", "action":"create", "authenticity_token":"17d5f8c8ca8f2afa6866a03304f9eaeaf950003295", "controller":"sessions", "login":"admin", "password":"haXor", "remember_me":"1"}
```

Using the method `filter_parameter_logging`, you can prevent fields from being logged into any file, e.g.

```ruby
class ApplicationController < ActionController::Base
  filter_parameter_logging "password"
end
```

Rails then replaces the value of the parameter "password" with the string "[FILTERED]".[TH07]
4.2.6 Transmission of Sensitive Data Using Secure Channels (HTTPS)

Passwords, addresses, payment information and other sensitive data should only be transmitted over a secure HTTP connection (HTTPS). Setting up the HTTPS certificates is outside the scope of a Rails application. Instead, the frontend balancer (e.g. Apache Webserver) is used to setup HTTPS. The only concern of the Archive Direct application is to make sure that certain pages are always transferred over HTTPS. As the user typically starts from a non-HTTPS site, e.g. http://www.archivdirekt.at, she must be redirected to the HTTPS protocol when viewing a page that contains or submits sensitive data. For example, all requests to http://www.archivdirekt.at/session/new, which is the login-screen, should be redirected to https://www.archivdirekt.at/session/new.

This can easily be done using the Rails plugin “ssl_requirement”. After installing it, you can enable SSL redirection for your actions by specifying

```ruby
class SessionsController < ApplicationController
  ssl_required :create, :destroy
end
```

4.2.7 Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) is the number one vulnerability of web applications. It has even surpassed buffer overflows in the number of reported security vulnerabilities. XSS attacks arise from the ability of users to provide textual input for a web application, which is not an uncommon feature, even if the application is not a web forum. A malicious user may then enter some JavaScript code into some application data field, and trick other users into viewing the page that outputs that data field again. This may be by sending her a link or make the page publicly available (e.g. a forum post). If the application is not secured properly, the user’s browser may then execute the (probably malicious) JavaScript code and, e.g., send some private information to another server.\[Wik08b\]

Prominent sites that were vulnerable to cross-site scripting include Google, Yahoo!, MySpace and Facebook. Rails provides an easy mechanism to protect your applications from such attacks by using an escaping function called h(...). When using this function everytime you print some model field (which potentially contains malicious code), all dangerous characters are automatically escaped. Due to its short name and Ruby’s optional braces for function arguments, using this function is very easy. Instead of

```erb
<%= file.description %>
```

you just write

```erb
<%=h file.description %>
```

to escape the output of the field.

---

4.2.8 Cross-Site Request Forgery (CSRF)

"Contrary to cross-site scripting (XSS), which exploits the trust a user has for a particular site, cross-site request forgery exploits the trust that a site has for a particular user."[Wik08a] When a user is surfing around various internet sites, it is possible that she is still logged in to a web application while opening another page. This page may contain some code that brings the user’s browser to load a specific (destructive) URL of the application. But in contrast to Cross-Site Scripting, this code does not have to be a complete JavaScript snippet, a simple link target or image source may be enough, e.g.

```html
<img src="http://www.example.com/destroy?user_id=1"/>
```

Therefore, it is not enough to strip HTML entities from the field contents. Rails provides request forgery protection by generating a unique token from the session ID that is included in every form as hidden field

```html
<input name="authenticity_token" type="hidden" value="489254aff63b4bda35ad5878521caa80563aed46" />
```

and then checked by a mechanism in the controller. Be aware that this mechanism only works for POST, PUT and DELETE requests. Hence, all destructive operations (deleting data, changing data, ...) should never be executed using GET requests. Anyway, this way of securely using the HTTP request types is encouraged for REST style applications.

The cross-site request forgery protection of Rails is enabled by default through the line

```ruby
class ApplicationController < ActionController::Base
  protect_from_forgery
end
```

in the base application controller.

4.2.9 Application Code Leakage

Breaking a web application is harder for the attacker, when she does not know the source code. Therefore, the code should be hidden and never be sent to the website user. This has been ensured in Rails by the following measures:

- No application code is located inside the web server’s document root. Only the “public” folder that provides images, javascript and CSS files is available for direct access.

- While application stack traces are useful while developing an application, they are turned off by default in production mode. For any errors (i.e. exceptions inside the application) the user

---

29Essentially only for POST requests, because PUT and DELETE requests are “faked” to POST requests by some JavaScript snippet, because browsers are still not able to issue requests other than GET and POST.
is just presented a helpful error message what he can do about the error now. Additionally, the stack trace is sent to the developers using the Rails plugin “exception_notification”.

4.3 Plugins

As mentioned in Section 2.3.5 on page 17, a Rails plugin can modify each and every aspect of the Rails framework. This is due to the flexibility of Ruby, mainly the possibility to re-open classes and rename or overwrite methods dynamically. Because the Rails core code cannot contain all the functionality one might ever need when writing a web application, a lot of things are available as plugins that can be installed on need. Using existing code for a function you need in your application is most probably better than developing it again. Therefore, plugins are essential when writing a Rails application, although there might also be some problems due to outdated code or incompatibilities between plugins (also see Section 7, “Lessons Learned” on page 67). Because sometimes a function is provided by more than one plugin, also the task of selecting the right plugins is important. The following table lists the plugins used in the Archive Direct application together with some information and description.

4.3.1 List of used Plugins

<table>
<thead>
<tr>
<th>Name</th>
<th>Source</th>
<th>Documentation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACL System 2</td>
<td><a href="http://opensvn.csie.org/ezra/rails/plugins/dev/acl_system2/">http://opensvn.csie.org/ezra/rails/plugins/dev/acl_system2/</a></td>
<td><a href="http://aclsystem.rubyforge.org/">http://aclsystem.rubyforge.org/</a></td>
<td>The authorization system of the Archive Direct application is built on this plugin. It allows you to specify the needed access rights for an action and provides some helpers to restrict parts of a view to a specific user group. See Section 4.2.1 on page 38 for details.</td>
</tr>
<tr>
<td>Active Scaffold</td>
<td><a href="http://activescaffold.googlecode.com/svn/tags/active_scaffold">http://activescaffold.googlecode.com/svn/tags/active_scaffold</a></td>
<td><a href="http://activescaffold.com/docs/">http://activescaffold.com/docs/</a></td>
<td>Active Scaffold relieves you from the pain of creating a management interface for the administrators over and over again, just with different fields. With a few lines of code, it allows you to build a complete ajaxified interface to create, edit, update, destroy and search records, including their associations.</td>
</tr>
<tr>
<td>Annotate Models</td>
<td><a href="http://repo.pragprog.com/svn/Public/plugins/annotate_models">http://repo.pragprog.com/svn/Public/plugins/annotate_models</a></td>
<td>none – just run rake annotate_models</td>
<td>Inserts table structure comments at the top of Model and Fixture files, so you do not have to go to the database every time to know which columns your table has.</td>
</tr>
<tr>
<td>Assert Valid Asset</td>
<td><a href="http://www.realityforge.org/svn/code/assert-valid-asset/trunk/">http://www.realityforge.org/svn/code/assert-valid-asset/trunk/</a></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## 4.3 Plugins

### Documentation:
- [http://www.realityforge.org/articles/2006/03/15/rails-plugin-to-validate-x-html-and-css](http://www.realityforge.org/articles/2006/03/15/rails-plugin-to-validate-x-html-and-css)

### Description:
Enables you to run (X)HTML and CSS checks in functional tests to ensure W3C validity. This is done using the online validation services at [http://www.w3.org](http://www.w3.org). The results are cached for speed improvement.

### Name: Awesome Nested Set
### Source: [git://github.com/collectiveidea/awesome_nested_set.git](git://github.com/collectiveidea/awesome_nested_set.git)
### Documentation: README, Source Code
### Description:
A better version of the ActsAsNestedSet Plugin, which allows to store Objects in a tree-like organization, like needed for the Categories in the Archive Direct application.

### Name: Exception Notification
### Source: [http://svn.rubyonrails.org/rails/plugins/exception_notification](http://svn.rubyonrails.org/rails/plugins/exception_notification)
### Documentation: [http://dev.rubyonrails.org/svn/rails/plugins/exception_notification/README](http://dev.rubyonrails.org/svn/rails/plugins/exception_notification/README)
### Description:
Using this plugin you can easily send yourself e-mails when a user sees an exception in production mode (i.e. "something went wrong, ...."). These e-mails contain backtraces, request headers as well as all session and environment information for debugging.

### Name: Rails RCOV
### Source: [http://svn.codahale.com/rails_rcov](http://svn.codahale.com/rails_rcov)
### Documentation: [http://agilewebdevelopment.com/plugins/rails_rcov](http://agilewebdevelopment.com/plugins/rails_rcov)
### Description:
RCOV is a test coverage analysis tool for Ruby. This Rails plugin integrates the RCOV tool into Rails and thus allows you to create pretty HTML files containing your test coverage statistics.

### Name: Resource Controller
### Source: [http://svn.ardes.com/rails_plugins/resources_controller](http://svn.ardes.com/rails_plugins/resources_controller)
### Description:
Because of the REST approach, the controllers in your application will mostly be the same. Resources Controller takes this chance to factor all actions out to a single module. Again, with just a few lines of code you build a full-featured REST controller that is even capable of handling nested resources.

### Name: Restful Authentication
### Source: [http://svn.techno-weenie.net/projects/plugins/restful_authentication](http://svn.techno-weenie.net/projects/plugins/restful_authentication)
### Description:
This is the plugin, on which the authentication system in the Archive Direct application is built. It provides a generator that allows you to create the necessary user model and creates a controller for logging in and out.

### Name: Simple Localization
### 4.3 Plugins

**Documentation:** [http://simple-localization.arkanis.de/docs/index](http://simple-localization.arkanis.de/docs/index)

**Description:** Since *Archive Direct* should be available in at least two languages, this plugin is used to provide the interface in different localizations. See Section 4.4 on page 49 for details.

<table>
<thead>
<tr>
<th>Name</th>
<th>Source</th>
<th>Documentation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asset Packager</td>
<td><a href="http://sbecker.net/shared/plugins/asset_packager">http://sbecker.net/shared/plugins/asset_packager</a></td>
<td><a href="http://synthesis.sbecker.net/pages/asset_packager">http://synthesis.sbecker.net/pages/asset_packager</a></td>
<td>When using multiple CSS and JavaScript files (which can be quite large when using some AJAX libraries), it is usually faster to combine them into a single file for production mode as the browser needs to establish a separate connection to the web server for every referenced file. This plugin contains a rake task that does that job for you (which can be even executed automatically on deployment when using Capistrano). Additionally, the plugin compresses the files by removing insignificant spaces, line-breaks and tabs.</td>
</tr>
<tr>
<td>ActiveRecordDefaults</td>
<td><a href="http://svn.viney.net.nz/things/rails/plugins/active_record_defaults/">http://svn.viney.net.nz/things/rails/plugins/active_record_defaults/</a></td>
<td><a href="http://svn.viney.net.nz/things/rails/plugins/active_record_defaults/lib/active_record_defaults.rb">http://svn.viney.net.nz/things/rails/plugins/active_record_defaults/lib/active_record_defaults.rb</a></td>
<td>Using MySQL defaults for columns is not very flexible, because you would have to write a migration every time you want to change the value. This plugin provides your model with default values using a simple syntax directly in the model class. This also allows you to set up default values for non-database-backed attributes and attributes that have a non-primitive data type.</td>
</tr>
<tr>
<td>ValidatesDateTime</td>
<td><a href="http://svn.viney.net.nz/things/rails/plugins/validates_date_time/">http://svn.viney.net.nz/things/rails/plugins/validates_date_time/</a></td>
<td><a href="http://svn.viney.net.nz/things/rails/plugins/validates_date_time/README">http://svn.viney.net.nz/things/rails/plugins/validates_date_time/README</a></td>
<td>This plugin allows you to validate given dates, times and datetimes in your model. This is not only a syntactical check, but also ensures that the content is valid (e.g. Feb 30, 2009 is invalid).</td>
</tr>
<tr>
<td>BackgroundRB</td>
<td><a href="http://backgroundrb.rubyforge.org/">git://github.com/gnufied/backgroundrb.git</a></td>
<td><a href="http://backgroundrb.rubyforge.org/">http://backgroundrb.rubyforge.org/</a></td>
<td>BackgrounDRb is a ruby tool that allows you to execute tasks asynchronously from your program in the background. This plugin integrates BackgrounDRb with Rails and is used for the image-scaling background tasks in the <em>Archive Direct</em> application.</td>
</tr>
</tbody>
</table>
4.3 Plugins

**Description:** This plugin provides an additional hook that is executed after a model record has been committed to the database. This is necessary for starting asynchronous tasks (see BackgrounDRb above), because the after_save hook is executed too early in this case.

**Name:** Prawnto  
**Source:** git://github.com/thorny-sun/prawnto.git  
**Documentation:** http://cracklabs.com/prawnto

**Description:** Prawn is a library to create PDF files from a Ruby script in a very easy way. Prawnto integrates this functionality into your Rails application and allows you to create views in PDF format.

**Name:** ActiveMerchant  
**Source:** git://github.com/Shopify/active_merchant.git  
**Documentation:** http://www.activemerchant.org, http://activemerchant.rubyforge.org

**Description:** Active Merchant is a payment processing library that provides support for various payment gateways (mostly located in the US, but it also includes PayPal). It is used in Archive Direct to process the payments through PayPal Express and (using a self-developed gateway) MPay24.

**Name:** Acts as State Machine  
**Source:** http://elitists.textdriven.com/svn/plugins/acts_as_state_machine/trunk  
**Documentation:** http://elitists.textdriven.com/svn/plugins/acts_as_state_machine/trunk/README

**Description:** This plugin allows you to turn your model into a state machine. By defining states and transitions between them, you can build an ActiveRecord model that always has a defined state and well-defined state-transitions. This plugin is used for the “Order” Model in the Archive Direct application.

4.3.2 Self-written Plugins

We also had to develop some plugins by ourselves, because the needed functions were not available. Beware that these plugins are not as extensively tested as others. The source code is available in the vendor/plugins directory.

**Name:** ActiveMerchant MPay Gateway  
**Description:** This plugin extends the ActiveMerchant plugin by a gateway that works with the MPay24 SELECTPAYMENT interface.

**Name:** String Extensions  
**Description:** Provides some additional methods for the String class, that were needed for the Archive Direct application and factored out into this plugin.

**Name:** Fixnum Extensions  
**Description:** Provides some additional methods for the Fixnum class, that were needed for the Archive Direct application and factored out into this plugin.

**Name:** Float Extensions
4.4 Localization

4.4.1 Localization Targets

Localization is not "just search for all strings in the code and translate them". Instead, providing a web application in different languages usually includes various aspects, some of them falling into the scope of "internationalization", i.e. adapting to the culture and conventions of different countries.

The following parts need to be somehow controlled by the current language of your application:

The output text. It is quite obvious to translate all text that is printed from your views and is visible to the user.

Document metadata. A bit less obvious are texts that are not directly visible, like HTML meta tag contents (site description, keywords), alternative IMG descriptions or RSS feeds.

Error messages. Active Record error messages that are produced by your validations should be translated to the same language as the rest of the site.

Models and Attributes. When translating the error messages, you should also translate the model's name and their attribute names so that the whole error message is localized, e.g. "Bild konnte nicht gespeichert werden: Beschreibung ist ungültig" instead of "Image konnte nicht gespeichert werden: Description ist ungültig".

Dates. Month names, day names and the date/time format needs translation and adaption, respectively (18.12.2008 vs 2008/12/18 vs 2008/18/12 vs Dec 18, 2008, ...).

Prices. Even if you decide to use the same currency for all languages (which might even be impossible for international websites), the representation of a price may be different from country to country (e.g. EUR 1.000 vs EUR 1,000).

Numbers. Yes, even plain numbers are represented differently in different languages, e.g. 1.000,34 vs 1,000.34.

Database content. Sometimes you also want to translate something that is stored in the database,
4.4 Localization

4.4.2 Rails Plugins

In order to solve this task, various Rails plugins are available, some of them using very different approaches.\(^\text{30}\)

**Globalize.** The Rails plugin Globalize is probably the most widely used localization plugin. It supports a huge number of languages and countries. Original texts from the source code and their translated versions are stored into a database table. Globalize has no ability to translate model names and their attribute’s names.

**Ruby-GetText.** This is the Ruby equivalent of the well-known GNU gettext library. It is a Ruby Gem and uses a similar syntax (i.e. `"this is the english version"`) and the same file format (.po files that are then compiled into binary files).

**SimpleLocalization.** This plugin uses a more rubyish approach to store the translations: YAML files. One YAML file provides settings, formats, model translations as well as the translated strings for the corresponding language. The drawback of this tool is, that it is not able to translate database contents and has no supporting tools (e.g. to extract strings automatically from the code).

4.4.3 SimpleLocalization

Despite the fact that the first two options may provide more features, after some prototyping the SimpleLocalization approach was implemented into the Archive Direct application. After installing it, you can use the provided .yml files as a base for your own versions. The files contain different sections. In the following, some excerpts of the German translation file are shown.

- Translated error messages

  ```yml
  active_record_messages:
    accepted: " attrib muss akzeptiert werden"
    empty: " attrib darf nicht leer sein"
    blank: " attrib darf nicht leer sein"
    ...
  ```

- Settings for translating times and dates

  ```yml
  dates:
  ```

\(^\text{30}\)A more complete overview of available options can be found at [http://wiki.rubyonrails.com/rails/pages/InternationalizationComparison](http://wiki.rubyonrails.com/rails/pages/InternationalizationComparison)
daynames: [Sonntag, Montag, Dienstag, Mittwoch, Donnerstag, Freitag, Samstag]
abbr_daynames: [Son, Mon, Din, Mit, Don, Fri, Sam]
date_formats:
  attributes: '%d.%m.%Y'
  short: '%e. %b'
  long: '%e. %B %Y'

• Translations and settings for various Rails helper methods

helpers:
  ...
  number_to_currency:
    precision: 2
    order: [unit, main, separator, fraction]
  number_to_phone:
    delimiter: ''
    extension:
    country_code:
  date_select:
    order: [:year, :month, :day]
distance_of_time_in_words:
  less than 5 seconds: weniger als 5 Sekunden
  less than 10 seconds: weniger als 10 Sekunden
  less than 20 seconds: weniger als 20 Sekunden
  less than a minute: weniger als eine Minute
  ...

• Translated model names and model attributes

models:
  image:
    name: Das Bild
    attributes:
      name: Name
      photographer_name: Fotograf
      new_tags: Schlagworte
    ...

• And finally, strings used in your application code

app:
  ...
  new_category: "Neue Kategorie"
  new_image: "Neues Bild anlegen"
4.4 Localization

Inside your views, you can use the translated versions of your strings using the helper method called `l(...)`. So, instead of

```ruby
<%= link_to "back", images_path %>
```

you use

```ruby
<%= link_to l(:back), images_path %>
```

which will insert the translated version of “back” into the output as defined in the “app” section of the `de.yml` file.

In order to ease development, a small Ruby tool was written to extract strings from the source code automatically. So you can use

```ruby
<%= link_to l("back"), images_path %>
```

during development, and after running the script all strings inside the helper function are replaced by a key (:back).

### 4.4.4 Translating Database Entries

As SimpleLocalization is not able to translate strings stored in the database, a plugin called Multi-languageColumn was developed (see Section 4.3.2 on page 4.3.2). It enables you to have database columns translated by just appending the language code to the column name, e.g. “name_en” and “name_de”. Depending on the current locale, the model’s attribute “name” then returns the correct version. The same applies for the setter method (“name=”).

### 4.4.5 Localization Features of Rails 2.2

Since version 2.2 – which was released a long time after we had chosen our localization strategy – Rails features a localization mechanism that looks very much like the approach of the SimpleLocalization plugin. It works with YAML files located under “config/locales/” that look very similar to those shown above.
5 Site Design and Navigation

To create an efficient web page, or actually an online shop that is easy and intuitive to handle is a hard task. A stable and highly performative fundament, as described in the sections earlier is essential to build up at. But to help the user to gain a maximum of comfort in handling the site, a coherent concept in designing the interface is needed. The design process to achieve the requirements of combining easy and intuitive handling and an appealing design with quick loading times and a good and stable performance is described in the following section.

5.1 The Process of Creating a Design Concept

Already at a very early stage of the project, we collected various ideas, made notes and sketches and discussed them with a wide range of people. We kept thinking of how to make the site attractive, as well in its appearance as in its handling. The goal was to create a design that is simultaneously functional, good-looking, immediately familiar to handle and allows to integrate innovative features to set the site apart from the mass of online photo sites.

While collecting ideas and discussing them, we tried to discover what makes the difference between an average and an outstanding website. Therefore, we had a look at a multitude of web sites and online shops and noted down functional and optical features we either liked or found annoying. With them in mind, several further versions and adaptations of our concepts were done and evaluated in discussions and interviews. For the interviews, we showed these drawings, and later our actual designs, to people, both familiar with computers and online shopping and some hardly using a computer at all. They were told to perform a task. Their behavior, as well as their feedback, helped to improve the navigation and the usability of the site furthermore.

Figure 11 shows one of our drafts of how thumbnail images can be displayed with an attempt to add as little overhead to each as necessary by still providing all essential information. A first version of the Search Box can be seen on the right side, giving the user the possibility to combine various search methods and edit them.

Refining and improving the design and the navigational features kept going on during the whole process of designing the application and implementing it. It was done in an iterative way that required a lot of discussion and testing, but once we had the most important features extracted, our fundamental requirements were formed into a basic concept that retained valid throughout the project.

5.2 The Interface – Design and Usability

While creating the interfaces for costumers and administrators, is was taken care of the following guidelines. They do represent the main principles extracted from the conception stage, and affect optical and aesthetical as well as usability and navigational issues.
5.2 The Interface – Design and Usability

5.2.1 Simplicity

The elementary principle in designing the interface was to keep it simple. Furthermore, the design had to be done for a website whose primary aim is to show and sell photos. And what can be more important to a good and informative site than its content?

Let the images speak. A strong and colorful image can appear lost or constricted if it does not get the space it would actually need. Therefore, we tried to put the spotlight on them and move the users’ attraction there. The simple white background is not distracting and does not influence the exposition of any image, regardless of its color or content. Already while opening the site, the user should be fascinated by the images.

The first view, as seen in Figure 12, shows just a simple and plain site with an image in a rather inconvenient format of 6x17 as its center, fixing the attraction. Intentionally only one big size image is presented for a few seconds, alternating with a couple of others in form of a cross-fading slide show. The image gains enough space to develop its impression.

In the browse and search view, as seen in Figure 13, the images are presented in a proper size without
5.2 The Interface – Design and Usability

Figure 12: Screenshot of the first view. The big size image in the inconvenient format of 6x17 gains the users’ attraction and sets the center of the page. In the online version, the image alternates with a couple of others in a crossfading slide show to demonstrate some of the stock photos in an immersive way.

any unnecessary overhead at first. Only by moving the mouse over them (as seen in Figure 14), or by clicking on them (as seen in Figure 15) more specific information is revealed. Any content that is currently not relevant can be hidden and the essential one accented.

Reduction to the Essential. As John Maeda\textsuperscript{31} wrote, the simplest way to simplify something is through thoughtful reduction. [Mae06]

By thinking which information and control elements are essential for every single page and which can be left out, we achieved a reduction to the key elements of every page. Thereby, we were able to place all navigational elements really necessary with still enough free space on the site and a simple design to help the user find his way immediately. The reduction of elements does not mean a reduction of functionality. Following the principle of only displaying the tools and information useful for the possible next tasks, we achieved a discreet and plain interface that is easy to understand.

Only when performing a specific task, e.g. browsing through categories or taking a closer look at an image, the necessary controls and information are provided.

The principle of simplicity was impressively demonstrated by the ongoing success and fame of Google\textsuperscript{32}. It is an amazing example of how few elements are necessary to achieve a high per-

\textsuperscript{31}Founder of the SIMPLICITY Consortium at the MIT Media Lab.
\textsuperscript{32}http://www.google.com
5.2 The Interface – Design and Usability

Figure 13: View for searching and browsing through the photo archive. The pictures do have enough space to be able to gain the user’s attraction and attendance. Except for the images, only really essential components are displayed in a very simple and reduced way.

forming und usable interface. It helps users to spend more time with the content than with handling it, while at the same time giving the website some kind of seriousness and reliability, or as John Maeda says:

"In simplicity we trust." [Mae06]

5.2.2 Consistent Site Arrangement

There are not many things more annoying than to spend a lot of time discovering how a system is working when you just want to get something out of it. In the example of an online shop, the criterion of an intuitive and easy-to-handle system is especially important, since there is plenty of competition. To help the user to find his way on the site very easily, a consistent site arrangement, that can be seen in Figure 16 was used.

The page was divided into virtual parts, consisting of the three major ones:

(1) Logo
(2) Navigation Bar
(3) Content
5.2 The Interface – Design and Usability

 SITE DESIGN AND NAVIGATION

Figure 14: While having a look at a page full of thumbnails, they are displayed without any currently distracting information. Only if a picture gains the user’s attraction and the mouse is moved over the picture, additional information as seen on the left thumbnail is displayed.

And the on demand appearing:

(A) Search Box

To make the navigation even more easy, an intuitive and to the experienced internet user familiar arrangement was used. This rather common arrangement is used by the better part of successful and well-known web shops and portals. So once a new user comes to the site, he can navigate just like he is used to.

(1) Logo. The Logo always stays on the upper left side. It gives the user the permanent possibility to return to the start page by clicking on it.

(2) Navigation Bar. Placed on the upper right side of the site, the Navigation Bar forms the central point of navigational and user-related issues. During the normal use of the site, a small line guides the user’s eye and leads him to the elementary controls on the right side. With only three tab buttons, one can navigate between the main sites. By highlighting the specific button, it is clearly visible on which site the user is at the moment. It also offers the possibility for logging in or out and changing the language (see Figure 17).

When a user wants to place an order and continues to the checkout, the page changes, but the basic arrangement remains the same. The Navigation Bar changes to the one seen in Figure 17. It gives the customer information of how far the ordering process is already done and shows what is still to come.

(3) Content. Due to the focus on the content of the site, a lot of room was reserved for it. Having the main navigational element already on the upper right side leaves enough space, even to present images in the very wide format 6x17 on any screen in an appealing and proper way as done on the start page (as seen in Figure 12). Basically, the whole width of the page is available for
Figure 15: A specific info site for any picture can be reached by clicking on its thumbnail. Here, any interesting information concerning the photo (Title, Photographer, Format, Date, Size, Price, ...) can be seen.

content as this was done for example on the starting page, the account setting, the shopping cart or the ordering process.

Only while browsing through the archive, while searching or while taking a close look on a specific picture, the Content area is shortened to the left side, where then the Search Box appears.

(A) Search Box. The search for a photo matching some given criteria can be done by using the Search Box seen in Figure 23. It appears on the very left side of the search and browse, as well as on the image detail view. In opposite to the width of the Logo above, the Search Box is resizing, depending on the user’s browser size. The content therefore shrinks depending on the size of the box.

When viewing the Administration Dashboard, as seen in Figure 21, the search box is replaced by a navigation box for administration purposes. Functions necessary for updating and maintaining the site are accessible through it.
5.2.3 Consistent Color Scheme

To achieve the simplistic layout, even the color scheme was held very minimalistic. The few colors seen in Figure 19 were used as the basic colors for the best part of the layout. By using a single color in combination with one entity like the admin area or image formats like download or print, a relation between color and task is created that helps the user to navigate and gain an overview more easily. The three main colors are:

- **# 40152d** - Used as the main color, as it was used mainly in the logo and for a small text on the navigation bar. Due to the fact that the focus of the site is on downloading images, this color was also to identify the download availability of images.

- **# 2d40a5** - As this color does match very good to the main color of the site, it was used to identify the second image format, the image prints.

- **# 7c0909** - The color red was used at every additional button for administrator users.
5.2 The Interface – Design and Usability

SITE DESIGN AND NAVIGATION

Figure 18: Navigation Bar at the start of a checkout process. The already finished pages are highlighted green, the current one in a dark gray and the gray ones are still to come.

Figure 19: Basic colors used for the design of the interface.

5.2.4 A Few Steps To Everywhere

Chaos and an unnecessary loss of time should be avoided by providing essential navigation features to allow the user to navigate to wherever he wants to. To return to the homescreen, the logo on the upper left of the site is clickable. There, as well as on all other pages except the ones necessary for the checkout process, the Navigation Bar, as seen in Figure 17, is provided. On the right side of the navigation bar links to the three basic functions of the site are provided. The links to

- the search site,
- the personal account administration (when logged in),
- and the shopping cart

cover all essential navigational features to allow fast and intuitive handling.

5.2.5 Optimal Adaptation to the User Screen

To fill the space of different client-resolutions and various browser sizes in an optimal way, the site was made resizeable. The Search Box, as well as the Navigation Bar are flexible and stretch according to the size of the window.

The most interesting part of adaptation to the user screen are the self-resizing images. The size of thumbnails and preview images is adapted according to the browser window size, using JavaScript. A more detailed description can be seen in Section 5.4.
5.2.6 Adequate Administration Area

In order to allow fast, easy and direct updates and maintenance, an administration area was built directly into the site, helping to keep the site interesting. Once an user with administrator rights is logged in, a fourth tab called “Admin” is presented in the Navigation Bar, as seen in Figure 20.

Figure 20: Once a user with administrator privileges is logged in, a fourth tab called “Admin” is presented.

The Administration Dashboard as seen in Figure 21 provides all necessary functions for updating the image archive or for maintaining the site. The Administration Dashboard is the starpage of the whole administration area. It shows untranslated tags, inviting the administrator to translate them, provides recently uploaded images and a navigation box on the left side with links to any administrational functionality. Once again, the interface is built up by following the design criteria mentioned above.

Figure 21: The Administration Dashboard, providing functions for updates and maintenance of the whole site.

On top of every image detail page as seen in Figure 15, an administration bar for the edition of image-related stuff, as seen in Figure 22, is presented allowing fast and direct administration for a single picture.
Figure 22: Presented on every image detail page when logged in as an administrator, these additional functions allow fast and direct administration of a single image-related content.

5.3 Image Search Functions

To help the customer find the image he was looking for, search methods are provided. By building them in a modular way, the system remains flexible and expandable for further developments of the search methods. All search methods are accessible using the Search Box, seen in Figure 23.

A search method can be selected by its related button on the top of the Search Box. The respective input fields and control elements are then presented. Once a criterion is added, it appears in the list below. All available search functions can be combined in arbitrary ways. If a specific criterion is not wanted anymore, it can be removed by clicking on the small “X” symbol on the right side of it.

Figure 23: View of the Search Box, with already two keywords “Burgenland” and “Weinbau” entered. All available search functions can be combined in arbitrary ways. A search criterion can be removed by clicking on the “X” symbol on the right side of it. The buttons on the top of the box are used to select the desired search method.

The ability to combine various search methods helps the user to simply add all ideas he is having of a specific picture to the Search Box, helping to find a matching image easier.
5.4 Image Presentation

Images make up the core of the website. Therefore, a decent presentation is necessary to lead the users’ attention to them. To display the images in an optimal way the following guidelines were defined:

**Image Ranking.** The display order of images is based on a ranking system that is calculated by viewing numbers of images and how often they were bought. This helps to present the most popular images at first, and therefore impress the customer at the beginning of his browsing.

**Reduction of Control and Information Elements.** To reduce the number of disrupting elements around an image, control and information elements are only presented when they are really useful. As seen in Figure 14, information to a specific thumbnail is only available while moving the cursors over it, or by clicking on it as seen in Figure 15.

**Optimal Size.** Images need space to speak for themselves. Therefore, thumbnails are variable in their size. Within a minimum and maximum, they adapt their size according to the size of the browser window using javascript. As the window is resized, the thumbnail size changes.

A very close look can be taken on a specific image in its detail view. Once the image is clicked on, a large watermarked preview version is presented, while the remaining site is overlaid gray as seen in Figure 24. Therefore, the Lightbox plugin was used.

Once again, the size of the presented preview image depends on the browser window size. In opposite to the thumbnails, no continuous resizing is possible, as only a large and a small preview version are available. The adaptation of the image size was done by ourselves using JavaScript.

The presentation of a large preview image using the Lightbox plugin was chosen for several reasons. The presentation of the very wide 6x17 format needs space. To lay the preview image above the site allows to use the necessary space. The gray background additionally brings the image out, letting the customer have a look at it without any parts of the site disturbing it.

5.5 AJAX

For creating a good looking interface that is fun to work with, Ajax is a very appropriate tool. It can make the site more useable, reactive and helps to save time as the number of page reloads can be reduced. Following our design guidelines, we tried to use Ajax only when it was really supporting the task. We tried not to make a show out of it, but used it to make the site more pleasant and its handling more efficient.

http://www.lokeshdhakar.com/projects/lightbox2/
5.6 Technical Usability

Searching and browsing images requires a lot of user interaction. A page reload every time, a keyword is added or deleted to the current search would be very annoying and reduce the sites usability. Therefore, Ajax was used to update just parts of the page, what makes the web application feel like an installed client application. So once the user enters a new keyword into the SearchBox, the form is submitted in the background and only the necessary parts are updated. Deleting a previously added search criterion or switching between search methods is also done by using Ajax, so that no page reloads are required.

Visual effects provided by Ajax are used to display a big size preview of an image, as seen in Figure 24. The therefore used Lightbox plugin displayes the image above the currently displayed page.

For every implementation using Ajax it was taken care of to still provide a non JavaScript fallback version. They substitute the functionality in a very simple case and do page reloads instead of updating just parts or display a big size preview image on a new page instead of the Lightbox display.

5.6 Technical Usability

To keep the site competitive, a wide range of guidelines concerning technical usability has to be followed. These guidelines concern the following topics:
5.6 Technical Usability

W3-Conformance. It was tried to satisfy the criteria of the World Wide Web Consortium, that defines today’s web standards.

Browser Compatibility. By consequent testing, browser and platform compatibility was achieved. Therefore, some browser-specific HTML and JavaScript code was necessary. During the development process, the design and functionality was tested in following browsers:

- Mozilla Firefox 2/3
- Apple Safari 3.2
- Opera 9.63
- Microsoft Internet Explorer 6/7

Image Replacement. To design the site, several images including text were used. An image replacement method using cascading style sheets (CSS) was used to keep the site search engine friendly and accessible.

Therefore, the text written in a raster graphic image is also written as plain text invisible for the user, but indexable for search engines. The following CSS code shows an example:

```css
background: transparent url(image.png) no-repeat scroll 0 0;
text-indent: -9999px;
overflow: hidden;
```

URLs. Due to the Rails routing mechanism, the site uses compact and readable URLs (*Uniform Resource Locators*). For example, instead of

```
http://archivdirekt.at?2710a55oidjowklj31
```

an address like

```
http://archivdirekt.at/images/201
```

is used to view an image. The main reasons for using such URLs:

- They are better read- and memorable.
- As they do not contain any session-info, a bookmark can be set on any page.
- Because they are containing readable information and are in a structured format, they help to optimize for search engines.
6 PAYMENT SOLUTIONS

Multiple Languages. The complete page is available in two languages – English and German. Once a new keyword is added to an image, the administrator is invited to translate it on the Administration Dashboard. The language can be selected at every time with a small button on the upper right side of the Navigation Bar.

6 Payment Solutions

Offering images as instant downloads requires suitable payment methods. As it should be possible to download the image a few seconds after its purchase, a fast validation is required. Therefore, payment methods fulfilling this demand were needed.

6.1 Analysis of Customer Wishes

As the credit card has established as a standard in internet payment, it was a necessity to offer it as a payment method. This secures the desired internationality and covers a wide range of customers. PayPal is another meanwhile well established provider, and it appeared important enough to offer it as a payment method as well.

A special target audience of the site are extended regional tourism, marketing, press and design offices. Especially for these predominantly Austrian customers direct bank transfers are suitable.

6.2 Selected Payment Solutions

As the handling of payment requires a lot of security and bureaucracy, it was outsourced. Two suppliers, each offering at least one standard in modern payment methods were selected.

PayPal. Over the last years the rapidly emerging PayPal\textsuperscript{34} has become a well established and trusted provider for internet payment today. Due to it’s popularity and the simple and free usage, it is an easy-to-integrate solution.

mPAY24. mPAY24\textsuperscript{35} is a well established provider for various online paying methods. As it offers the desired methods of credit card and bank transfer payment, the common interface was integrated. The ArchiveDirect can now decide which methods should be supported by buying the licenses for the desired ones.

\textsuperscript{34}https://www.paypal.com/
\textsuperscript{35}https://www.mpay24.com/
7 Lessons Learned

Lesson 1: Using Changing Frameworks Causes a lot of Update Work.

The current major release of Rails is 2.x.x. While other languages or frameworks often try to retain full compatibility for old code, a lot of methods were removed or changed in an incompatible way when Rails 2.0.0 was introduced. This means, upgrading your application to this new major release may result in a lot of work to incorporate all of these changes.

Why are changes necessary? Besides the fact that developers often simply have new ideas and find ways to make things easier and more powerful, new Internet technologies and evolutions require changes in web application frameworks, too. For example, AJAX (and JavaScript in general) and web services change the way applications are built.

While changes between minor releases are typically backwards-compatible and deprecated methods just produce some warnings on the console, the Rails core developers decided to do some cleanup on the major release. This helps to keep the framework code clean and well-structured.

In addition to updating your application code to a new Rails version, it might also be necessary to update the plugins you are using. When a plugin gears deeply into the Rails code, even a small code change might cause it’s breaking. That means that you might have to fix installed or self-written plugins which may result in a lot of work. Hopefully, this situation will change with the next Rails major release, where the developers want to provide a stable API (application programming interface) for plugins.

Our personal feeling is that Ruby on Rails requires more work to keep your application up-to-date and working with the newest release than with other frameworks, but in return, you get an easy development environment featuring the newest technologies.

Lesson 2: Writing a Rails Application is Finding and Integrating Plugins.

One of the largest Rails plugin directories, http://agilewebdevelopment.com/plugins, currently has over 1,000 plugins available. Why is there such a huge number of plugins? Because the Rails core developer team tries to keep the framework code clean and as small as possible. It is not their intention to include each functionality you might need in a web application. Therefore, lots of plugins can be found on the Internet.

When you need a specific function in your Rails application, it is very likely that someone else has already written a plugin providing it – you just need to find it. Therefore, writing a Rails application typically includes searching the web for plugins, probably comparing different implementations, installing plugins and becoming acquainted with their usage. Even though this looks like a lot of work, it is typically less work than implementing the function by yourself. Chances are high that the plugin has been tested extensively by it’s developer and other users.
Lesson 3: Plugins Are not always as High-Quality as the Rails Core Code.

As mentioned above, using plugins not only brings you additional functionality, but it also might bring you additional problems. This is due to the fact that (especially small) plugins are typically not maintained as good as the Rails core, because there is a much smaller user community. As a result, it might take a long time before a plugin is updated after it broke due to an incompatible Rails changeset.

Another problem which can arise when using plugins is, that some of them are incompatible with each other. Then you either have to remove one of those conflicting or fix it by yourself if possible, which possibly introduces a lot of work again.

Choose your plugins wisely, have a look at the source code and understand how they work. This makes it easier to use them and fix them in case of incompatibilities or bugs.

Lesson 4: Executing Automatic Tests Can Take a lot of Time.

Automatic tests, like unit tests, functional tests and integration tests are great tools when developing an application. As mentioned in Section 2.3.4 (Ruby on Rails, Testing), you can have your tests run automatically in the background. But as your application grows, typically also the number of tests raises (as you want to cover all your application code by appropriate tests) and thus the execution time of your test suite raises.

While developing Archive Direct, about 215 tests were written, some of which deal with image processing (loading, scaling, watermarking, saving). The problem was that those operations took very long to be executed, needed a lot of resources and thus made the test execution painfully slow. Thus, even if you have to test your operations with real images to make sure they work correctly, you sometimes have to work with fake objects and fake methods.

Rails makes this easy by providing you the possibility to use mock objects for testing (i.e. you overwrite classes or methods of classes in the testing environment only). So we ended up with using small or memory-only images and faked resizing methods for most of our unit tests. Even if it seemed fast enough when executed once, it was just too slow for every-day testing.

Lesson 5: JavaScript Is Nice when it Works, but a Pain to Debug.

The new trend of using JavaScript more often nowadays to create websites acting and feeling like real applications, forces web developers to develop and, of course, debug JavaScript code again. Fortunately, there exist a lot of libraries which solve various tasks for you (e.g. Ajax libraries, animation libraries, form libraries, input validation libraries, . . . ).

However, you should not underestimate the time needed for integrating those libraries together in your application, testing all the features you need in all browsers you want to support and debug.
JavaScript code that does not work as intended (maybe just on specific browsers or platforms).

As mentioned in Section 3.4 (AJAX Libraries and Incompatibilities), libraries may be conflicting with each other due to overlapping function names, which means that you have to either fix a library or (probably faster) look for an alternative.

Regarding testing, there are no real usable automatic JavaScript testing frameworks that enable you to test the dynamic features of your website in different browsers. This results in a lot of (maybe repetitive) work.

Debugging JavaScript is fortunately supported by various browser tools like FireBug\(^\text{36}\), which includes a powerful JavaScript debugger and profiler.

**Lesson 6: Think and Discuss until You Find the Obvious Solution.**

When developing an application, you often have to make design decisions. How to save your data? How to organize your methods? How to structure your site?

Our experience during the development of the *Archive Direct* application showed that it is good to have a partner to discuss things with. Different people often have different ideas and think in different directions. If you found a solution, but are not completely satisfied with it, continue thinking and discussing. We found out that for nearly every problem a clean and (afterwards) obvious solution exists. Even if finding this solution takes a lot of time, it is not worth spending time on implementing a non-optimal solution that you might have to change again.

**Lesson 7: Don’t Underestimate Developing a Full-Featured Web Application.**

Even with the power of Rails, developing a full-featured web application is not just “let’s draw some design, code some HTML and CSS and then make the whole thing dynamic”.

The *Archive Direct* application includes 35 database tables we needed to design, about 3800 lines of Ruby application code, 1500 lines of test code, 2900 lines of HTML with embedded Ruby, 1500 lines of CSS and 150 lines of JavaScript. Yes, we underestimated the amount of work needed for such an application.

While it was planned to create the *Archive Direct* website in the summer term 2008, now, half a year later, we are eventually ready to release a first version to the public.

\(^{36}\text{http://getfirebug.com/}\)
References


