Masaryk University
Faculty of Informatics


# Web applications in Haskell

Master's thesis

**Bc. Pavel Dvořák**

Bratislava, autumn 2011

## Declaration

I hereby declare that this thesis is my original work and that, to the best of my knowledge and belief, it contains no material previously published or written by another author except where the citation has been made in the text.

.....

signature

**Adviser:** RNDr. Libor Škarvada

# Acknowledgements

I am grateful to my adviser Libor Škarvada for his time, patience, worthwhile lectures, and also for enabling me and Peter Molnár, who is no longer with us, to teach a course about functional programming in the last two years.

I am also very grateful to my family and friends for their toleration of my constant unavailability, which they must endure during writing of this thesis.

Thank you all.

## Abstract

The main goal of this thesis has been to provide a description of web development methods in the programming language Haskell. That includes an exploration of web frameworks written in Haskell and their comparison.

Furthermore, one of the frameworks, named Yesod, has been extended to intercommunicate with CouchDB, a document-oriented database system.

## Keywords

Haskell, functional programming, HTTP, web development, web frameworks, web services, CouchDB

# Contents


**1 Introduction** **7**

**2 Web development using Haskell** **9**

2.1 Hypertext Transfer Protocol . . . . . 9

2.1.1 HTTP requests . . . . . 10

2.1.2 HTTP responses . . . . . 11

2.1.3 HTTP client library in Haskell . . . . . 12

2.2 Serving content by Haskell . . . . . 15

2.2.1 CGI and FastCGI . . . . . 15

2.2.2 WAI and Warp . . . . . 18

2.2.3 Other implementations . . . . . 19

**3 Haskell web frameworks** **21**

3.1 A brief overview of Haskell web frameworks . . . . . 21

3.1.1 Happstack . . . . . 22

3.1.2 Snap . . . . . 22

3.1.3 Yesod . . . . . 22

3.1.4 Other active projects . . . . . 23

3.2 Comparing Happstack, Snap and Yesod . . . . . 24

3.2.1 URL mapping . . . . . 24
3.2.2 Database handling . . . . . 25
3.2.3 Markup . . . . . 26

**4 CouchDB and Persistent** **29**

4.1 CouchDB: a document-oriented database system . . . . . 29
4.1.1 JavaScript Object Notation . . . . . 30
4.1.2 Views . . . . . 31
4.1.3 CouchDB in Haskell . . . . . 31
4.2 Extending Yesod for CouchDB . . . . . 32
4.2.1 Implementing the Persistent back end . . . . . 32

**5 Conclusions and future work** **34**

# Chapter 1

# Introduction

> I just had to take the hypertext idea and connect it to the TCP and DNS ideas and — ta-da! — the World Wide Web.
>
> — SIR TIM BERNERS-LEE[I]

The World Wide Web (or *the Web* for short) has been the most widely used internet service over the past decade and a half (the detailed history of the Web and the Internet is described in [2]). The merits of its gradual expansion are indisputable: the Web has facilitated not only communication and information access, but also changed for the better the lives of many people. Regardless of the initial overoptimistic enthusiasm, which has been topped off with the dot-com bubble in the beginning of 2000s, the more realistic predictions look promising. It does seem that the Web is going to be with us for a long time.

The service was originally designed for exchange of hypertext documents only, but as the time went and the number of connected users rose, the requirements increased as well. The simple static content ceased to be enough and the users wanted it to be more complex and dynamic. Therefore Sir Tim Berners-Lee founded the World Wide Web Consortium[II] (or *W3C* for short), an international standard organization for the Web. The main goal of W3C is to extend the original Web specifications according to the needs of the industry. Prior to the formation of W3C, the web browser vendors had been adding proprietary features to their software, which led to incompatibility problems and confusion.

However, the specifications issued by W3C relate mainly to the client (i.e. web browser) side of the service. These specifications include, for example, HyperText Markup Language (or *HTML* for short) and Cascading Style Sheets (or *CSS*

[I] The original creator of the Web; for his complete biography, see [1].
[II] Homepage: `http://www.w3.org/`.

for short). The former standard is used for creating structured documents, the latter for shaping their visual aspect. With such tools, web developers are able to form a complex content. But what about the dynamic part? Are there any specifications for writing web applications that respond to the user input? Well, partially. The client side scripting is mostly done with ECMAScript (commonly known as JavaScript) developed by Sun Microsystems and afterwards standardized by Ecma International. The server side, which functions for processing requests and serving the corresponding responses, is based on a few rare changing standards (for example, W3C introduced the Common Gateway Interface, or *CGI* for short, that specifies the way in which the code is executed by a web server). It means that for a web development (on the server side), virtually any programming language can be used.

In the meantime, functional programming boomed. New languages, such as Scala and F#, emerged. The declarative paradigm became more popular than ever and started to be employed outside of academia. One of these successful languages is Haskell,[III] a strongly typed purely functional language. Despite being designed by an academic committee, Haskell found its way to general public and currently, the language is maintained by community, whose members provided many libraries. Haskell is even being used for commercial purposes.[IV] For more information about programming in Haskell, we recommend sources [4, 5, 6, 7].

This thesis is about utilizing the programming language Haskell for development of web applications. Currently, there has been a rapid progress in this area, particularly caused by growing interest in web development in general. Also, the growth has been accelerated by introducing new ideas, such as enumerators, a concept that improves effectiveness of monadic computation dramatically.

The following chapter contains a brief description of the Hypertext Transfer Protocol and the basics of deploying web applications in Haskell.

In the third chapter, we introduce web frameworks, which make web development much easier. We also compare three major Haskell web frameworks.

The fourth chapter describes our improvement of the framework Yesod. We have created a Haskell module that creates a link between the Yesod database interface and the document-oriented database system CouchDB.

And finally, the last chapter summarizes our text and proposes some of the next steps that can be done in the presented subject-matter.

[III] Homepage: `http://haskell.org/`; for a comprehensive language definition, see [3].
[IV] See the Industrial Haskell Group: `http://industry.haskell.org/`.

# Chapter 2

# Web development using Haskell

Our language is state-free and lazy
The rest of the world says we're crazy

But yet we feel sure
That the pure will endure

As it makes derivation so easy.
— JOY GOODMAN[I]

The Web is basically an internet service that is provided by web servers. A web client communicates with one of these servers through the Hypertext Transfer Protocol (or *HTTP* for short), alternatively through its secure version called HTTPS.

The exact specification of HTTP is defined in the document [8].

## 2.1 Hypertext Transfer Protocol

On the application level, HTTP works on the principle that the client sends a request, to which the server answers with a response. The protocol itself is stateless, so in Haskell, it would be natural to use a function with the following type signature: `communicate :: Request → IO Response`. As we connect to the outside world, the `IO` type constructor denotes the input/output monad; the types `Request` and `Response` would be data structures defined here later.

What should a proper HTTP communication look like?

[I] A researcher at University of Cambridge.

### 2.1.1 HTTP requests

An HTTP request consists of a request-line and optional request headers and a request body. The request-line includes a request method and a path name. All the data are separated by the `<CR><LF>` sequence (hexadecimal codes `0x0D` and `0x0A` in the ASCII character set).

#### HTTP request methods

A request method determines the way in which the request is going to be handled. The current version of the protocol defines eight different request methods (see [8, section 5.1.1]), but in practice, mainly these three methods are used:

**The `GET` method**
: The method is the commonest one. It is for example used when the user enters an address of a web site to the browser or when they traverse to another document through a link on a web page. This method is usually cached (both on the client side and on the server side), in order to save some bandwidth.

**The `HEAD` method**
: The same as the previous method, except that the response body is omitted and only the response headers are returned. This is useful in situations where we are not interested in the content and we just want the metadata.

**The `POST` method**
: With this method, the client can send some nontrivial data to the server. Apart from transferring binary files, the method is used for submitting HTML forms. As a rule of thumb, we choose this method when we need to modify a state of the server; for the read-only access `GET` mostly suffice.

#### HTTP path names

A path name is a part of the Uniform Resource Locator (or *URL* for short) that serves for an unambiguous identification of the objects on the Web. The path name begins with a slash symbol and can be seen as a relative path to a file or a directory. For example, the URL `http://example.com/dir/image.png` contains a path name `/dir/image.png`.

Since HTTP/1.1,[II] the initial part of the URL has gained in significance, because the domain name (`example.com` in the above mentioned URL) is sent as a part

[II] The version of the HTTP standard proposed in 1996. Its revised variant is used nowadays.

of a request in order to distinguish between web sites (with different domain names) located on a web server. Before that, there was no easy way to serve multiple web sites using one web server only; the domains had to be separated by an IP address or by a port number. The technique for accessing multiple domains on a single host is called virtual hosting.

#### HTTP request headers and body

A header is a key-value case-insensitive pair separated by a colon. For instance, the `Accept-Charset` header specifies one or more character sets (such as UTF-8), in which the client is able to communicate. Also, the above introduced domain name is sent in a `Host` header.

Some of the request methods (namely `POST`) usually need to carry additional data. It can be stored in the request body and indicated by a `Content-Length` or `Transfer-Encoding` header. If the request body is present, it is preceded by an empty line, in order to distinguish it from the headers.

### 2.1.2 HTTP responses

An HTTP request consists of a status-line and optional response headers and a response body. The status-line includes a response status code. Again, the data are separated by the `<CR><LF>` sequence.

#### HTTP response status codes

A status code symbolizes a state of the server and is represented by a three-digit number. The codes are divided into five categories according to their leftmost digit. Furthermore, in order to provide a human-readable representation, a textual description is assigned to each such code.

**The `1xx` status codes**
: The *informational* codes stand for provisional requests. Namely, the server confirms partially processed request (`100 Continue`) or indicate a need to change a protocol (`101 Switching Protocols`).

**The `2xx` status codes**
: The *success* codes are hopefully the commonest ones. When an operation goes smoothly, the server responds with a simple `200 OK` or in a more specific manner.

**The `3xx` status codes**
: The *redirection* codes signal to the client that further action is required. For example, when a page is moved to another location, it is a good idea to redirect it by returning the status code `301 Moved Permanently` (or temporarily by the code `302 Found`) together with the new URL. The client then resends the request, this time to the corrected location.

**The `4xx` status codes**
: The *client error* codes cover various responses to the problems in the request. In particular, these codes are used to handle an abstruse request (`400 Bad Request`), to prevent access (`401 Unauthorized` and `403 Forbidden`) or to inform about non-existent location (`404 Not Found`).

**The `5xx` status codes**
: The *server error* codes are reserved for server-side failures, such as a bug in the web application (`500 Internal Server Error`) or overload (`503 Service Unavailable`).

#### HTTP response headers and body

Similarly to the request, the response can also contain headers and a body. The difference is that the response body represents a content, e.g., an HTML page. Many headers[III] are possible to use both in the requests and in the responses.

### 2.1.3 HTTP client library in Haskell

The Haskell `HTTP` package[IV] implements communication on the client side. It consists of several submodules and we are going to look briefly at the two most fundamental ones.

#### Network.HTTP

As one would expect, the module defines data structures for the HTTP request:

```
data Request a = Request { rqURI :: URI
                         , rqMethod :: RequestMethod
                         , rqHeaders :: [Header]
                         , rqBody :: a
                         }
```

[III] For instance the `Date` header which denotes the time of the creation of the HTTP message.
[IV] Available at the URL: `http://hackage.haskell.org/package/HTTP`.

... and also for the HTTP response:

```
data Response a = Response { rspCode :: ResponseCode
                           , rspReason :: String
                           , rspHeaders :: [Header]
                           , rspBody :: a
                           }
```

The `URI` type is a data structure defined in the `Network.URI`[V] package and stands for Uniform Resource Identifier, a little more general concept than a URL. Besides, while the `Request` contains an enumeration `RequestMethod` (with data constructors such as `GET`, `POST` etc.), the `Response` includes a triple of integers `ResponseCode` together with a string representing the status message. The data structure `Header` occurs both in the `Request` and `Response`, just as the body, which is usually a `String` type.

We can generate our request by ourselves, assembling the URI using the `parseURI` function located in the `Network.URI` module. The function returns a `Maybe` value which we extract by a function from the module `Data.Maybe`:

```
> let fiURI = fromJust $ parseURI "http://www.fi.muni.cz/";
  fiHeaders = [Header HdrHost "www.fi.muni.cz"];
         fi = Request {rqURI=fiURI, rqHeaders=fiHeaders,
                       rqMethod=GET, rqBody=""}
> fi
GET http://www.fi.muni.cz/ HTTP/1.1
Host: www.fi.muni.cz
```

... or we can take advantage of the shortcut function `getRequest`:

```
> let fi = getRequest "http://www.fi.muni.cz/"
> fi
GET http://www.fi.muni.cz/ HTTP/1.1
Content-Length: 0
User-Agent: haskell-HTTP/4000.2.1
```

Anyway, for establishing the HTTP communication, we use a basic function that is called `simpleHTTP`. It works on a same principle as the above defined hypothetical function `communicate`. As we can see on the following example, it takes a `Request` and returns a value, that is in fact of the type `Either Response`:

[V] Available at the URL: `http://hackage.haskell.org/package/network`.

```
> simpleHTTP fi
Right HTTP/1.1 200 OK
Date: Sat, 24 Dec 2011 14:22:56 GMT
Server: Apache
Content-Location: index.xhtml.cs
Vary: negotiate,accept-language
TCN: choice
Last-Modified: Sat, 24 Dec 2011 00:30:58 GMT
ETag: "104032-3520-4b4cba5b05480"
Accept-Ranges: bytes
Content-Length: 13600
Connection: close
Content-Type: text/html; charset=UTF-8
Content-Language: cs
```

Since the returning value is within the `IO` monad, we are able to apply functors, monadic functions and operators to the response, for example:

```
> simpleHTTP fi >>= fmap (take 42) . getResponseBody
"<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n<!D"
```

### Network.Browser

If we are interested in more complex features of HTTP, such as authentication, handling proxy connections or storing cookies, the `Network.HTTP` module ceases to be enough. For these purposes, it is better to use the `Network.Browser` module that introduces states. That corresponds more with a real web browser.

The module also automatizes some processes; for illustration, let us see a redirect from the URL `http://fi.muni.cz/` to the URL `http://www.fi.muni.cz/`:

```
> browse . request $ getRequest "http://fi.muni.cz/"
Sending:
GET / HTTP/1.1
Host: fi.muni.cz

[...]

301 - redirect
Redirecting to http://www.fi.muni.cz/ ...

[...]
```

## 2.2 Serving content by Haskell

Until now, we have presented mainly the client side of the web development — it is vital for understanding of the server side, that concerns us primarily. In other words, from now on, we are going to focus on the server side only.

There are two ways to operate a web application: connect it with a stand-alone web server through an interface or use a native web server. The former choice takes advantage of established procedures, the latter is essentially a more straightforward method, because we do not need to install and configure another piece of software. Nevertheless, in Haskell, both options are possible.

### 2.2.1 CGI and FastCGI

The classic way of deploying a web application is to use the Common Gate Interface. We have already mentioned it briefly in the introduction — CGI is a web standard defined in the document [9]. As the name suggests, it belongs to the interface category and extends capabilities of a web server (for example the Apache HTTP Server[VI] or lighttpd[VII]).

#### Network.CGI

For employing CGI in Haskell, we need to write a web application which has the `Network.CGI`[VIII] module included, compile it and configure the web server to execute the application by CGI. When the web server receives a corresponding HTTP request, it passes the environment variables (such as the request headers) on through CGI to the application. Then CGI waits for the result of the computation and delivers it back to the server. The following example is a simple CGI script in Haskell that displays a "Hello, world!" text for every HTTP request:

```
import Network.CGI

main :: IO ()
main = runCGI . handleErrors $ output "Hello, world!"
```

The function `runCGI` just runs the web service; the function `handleErrors`, which could be in this particular case safely omitted, returns the `500` status code in case of application failure; the function `output` turns a string to a monad.

[VI] Homepage: `http://httpd.apache.org/`.
[VII] Homepage: `http://www.lighttpd.net/`.
[VIII] Available at the URL: `http://hackage.haskell.org/package/cgi`.

Considering that we have got the Apache server installed and our little CGI script is saved in the path `/var/www/hello_world.hs`, we compile it by invoking the `ghc --make -o hello_world.cgi hello_world.hs` command. Then, we configure the web server in this manner:

```
<VirtualHost *:80>
ServerName example.com
Options +ExecCGI
AddHandler cgi-script .cgi
DirectoryIndex hello_world.cgi
DocumentRoot /var/www/
</VirtualHost>
```

The `VirtualHost` directive denotes a domain-based hosting running on the default HTTP port number. It contains entries that arrange execution of the desired CGI script. Provided that the DNS record `example.com` points to our machine and the web service is running, after entering the correct URL, the web server should return the expected response.

We can also obtain the environment variables using one of the few convenient functions, such as `requestMethod`, `requestHeader` and `pathInfo` to utilize their output it in our code. Or we can alter the HTTP response using functions such as `setStatus` and `setHeader`. Yet again, all the functions are monadic ones.

### Network.FastCGI

The original CGI approach works, but it is not very efficient, because it spawns a new process for every request. That is the reason for introducing FastCGI,[IX] a faster and more secure variant of CGI. It reduces the overhead by capability to process several requests at once.

The Haskell FastCGI module `Network.FastCGI`[X] works on similar principles as the regular CGI. The only difference is that we replace the function `runCGI` with one of the functions located in the FastCGI module. These functions are compatible with the regular CGI, so the rest of the code remains the same.

The transition to FastCGI takes place in this manner: firstly, we need to have the FastCGI web server module installed. Secondly, our little CGI example must be modified to use the other interface. The rewritten code would look like this:

[IX] Homepage: `http://www.fastcgi.com/`.
[X] Available at the URL: `http://hackage.haskell.org/package/fastcgi`.

```
import Network.FastCGI

main :: IO ()
main = runFastCGI $ output "Hello, world!"
```

To distinguish it from a regular CGI, it is better to compile it with a different suffix (using the `ghc --make -o hello_world.fcgi hello_world.hs` command). Apart from that, the Apache configuration also undergoes a slight change:

```
<VirtualHost *:80>
ServerName example.com
Options +ExecCGI
FastCGIExternalServer hello_world.fcgi -socket hello_world.sock
DirectoryIndex hello_world.fcgi
DocumentRoot /var/www/
</VirtualHost>
```

In the configuration, we have specified a socket file that the FastCGI process employs for communication with the web server. The other possibility is to use the `-host` option, which orders the process to communicate through a TCP port. The former way is simpler, the latter is better when we operate the service on several servers.

The both examples run on a single thread only. To achieve a full potential, it is possible to compel FastCGI to take advantage of multiple threads. In particular, when running a multi-core system, the multithreaded program is more efficient. If we wanted to spread the previous example to eight threads, we would write the following code:

```
import Network.FastCGI

main :: IO ()
main = runFastCGIConcurrent 8 $ output "Hello, world!"
```

Moreover, the GHC compiler needs to have the `-threaded` option enabled. It means that in this case, we initialize the compilation by invoking the `ghc --make -threaded -o hello_world.fcgi hello_world.hs` command.

### 2.2.2 WAI and Warp

The other typical way to operate a Haskell web application is to use the Web Application Interface[XI] (or *WAI* for short) that is written solely in Haskell. Similarly to CGI, it is an interface for communication between a web application and a web server. WAI works best in connection with a native Haskell web server named Warp,[XII] but other back ends (e.g. CGI), which are called handlers, are supported as well. Both WAI and Warp are written by the same developer and together, they constitute the fastest Haskell platform for running web applications (as described in the article [10]).

A "Hello, world!" application is not as straightforward as with the CGI modules:

```
{-# LANGUAGE OverloadedStrings #-}

import Network.Wai
import Network.Wai.Handler.Warp
import Network.HTTP.Types

main :: IO ()
main = run 8000 (const . return $
                 responseLBS statusOK [] "Hello, world!")
```

The first line of the code is a language extension that implicitly converts ordinary strings to other forms (the instances of the `IsString` type class). WAI utilizes the data type `ByteString` for the sake of effectiveness. So, instead of converting our string manually, it is much easier to just load the extension.

The function `run`, that is imported from the Warp module, runs the web server with a default configuration on a specified port number. It also takes a value, which has a type of `Application`. It is a type synonym for a familiar type `Request → Iteratee ByteString IO Response`. Except for the enumerator `Iteratee` and the `ByteString`, the type remind us the above presented HTTP principles.

To return an `IO` response for every request, the `const` and `return` functions are employed. Furthermore, the function `responseLBS` returns a lazy byte string response that consists of a status code,[XIII] a list of headers and a body.

[XI] Available at the URL: `http://hackage.haskell.org/package/wai`.
[XII] Available at the URL: `http://hackage.haskell.org/package/warp`.
[XIII] The HTTP `200 OK` status returning function `statusOK` is located in the `Network.HTTP.Types` module.

When we run the code using the `runhaskell` command or by compiling it first and executing the output binary thereafter, the application should be accessible at the URL `http://example.com:8000/`.

It is worth mentioning that the `Network.Wai` module does not contain any helper functions. To work with the interface effectively, it is better to use the modules located in the `wai-extra` package.[XIV] They offer functionality such as request parsing or redirecting.

### 2.2.3 Other implementations

There are many other projects that provide facilities for serving web applications written in Haskell, even though they are not as powerful and common as FastCGI or Warp. But they could come in useful nonetheless.

#### Hack2

The second version of the web interface Hack[XV] is very easy to use and provides a few back ends (e.g. for Warp). In many ways, its modules resemble the WAI project. Hack was initially inspired by a similar project called Rack[XVI] that is a simplistic web interface for applications written in the Ruby programming language.

#### Hyena

The Hyena project[XVII] could be called a rather experimental web server. It is probably the first announced (see [11]) implementation of a web server that is based on the `Iteratee` data structure. Accidentally, one of its modules is called `Network.Wai`, which could lead to possible conflicts with the WAI module.

#### Lucu

The web server Lucu[XVIII] was designed mainly with regard to RESTful applications. REST is an abbreviation for the Representational State Transfer and

[XIV] Available at the URL: `http://hackage.haskell.org/package/wai-extra`.
[XV] Available at the URL: `http://hackage.haskell.org/package/hack2`.
[XVI] Homepage: `http://rack.rubyforge.org/`.
[XVII] Available at the URL: `http://hackage.haskell.org/package/hyena`.
[XVIII] Available at the URL: `http://hackage.haskell.org/package/Lucu`.

denotes a kind of web-based application programming interface. Since Lucu is intended to run behind a reverse proxy server, it does not have implemented advanced features such as logging or client filtering, although it is a rather large project when compared with the others.

### Webserver

And the last project in question is simply named Webserver.[XIX] Its easy structure and plainness makes it look like a good candidate for educational purposes, but there are currently better options, namely the Hack2 project.

[XIX] Available at the URL: `http://hackage.haskell.org/package/webserver`.

# Chapter 3

# Haskell web frameworks

> You don't need a framework. You need a painting, not a frame.
> — KLAUS KINSKI[I]

A software framework is a collection of libraries designated for a specified task. It introduces best practices that try to prevent a programmer from code repetition and monotonous work. In contrast to an ordinary software library, a software framework determines a control flow of a program, i.e., it specifies the overall architecture of an application, frequently by employing one of architectural patterns, such as Model-view-controller (or *MVC* for short).

A web framework[II] is a software framework intended for development of web applications. It usually contains facilities for communication with a database management system, template processing, URL mapping and others.

## 3.1 A brief overview of Haskell web frameworks

In Haskell, there is more than a dozen web frameworks available, but not all are quite usable. Some of them ceased to be maintained, some are still immature. For instance, Alson Kemp, author of the Turbinado web framework, abandoned the one-man development in the beginning of 2010 after he had found out that Haskell does not satisfy his needs (for his full announcement, see [12]).

Nevertheless, let us take a brief look at the currently active Haskell web frameworks. We start with the three major ones.

[I] A famous German actor and director.
[II] Sometimes called a web application framework.

### 3.1.1 Happstack

The Haskell application server stack[III] (or *Happstack* for short) is originally based on a now discontinued project called HAppS.[IV] It comes with its own web server, but it is possible to replace it with the FastCGI or Hack2 back end. Also, the developer of the WAI project planned to add Happstack support in a near future (as mentioned in [10, page 85]).

For a long time, Happstack lacked a proper manual, fortunately one of the developers began to write a tutorial [13]. Since Happstack is a large piece of software that has been maintained for years, there exists its lite version,[V] which could help with the initial understanding.

### 3.1.2 Snap

As described in [14], Snap[VI] is a web framework that provides reusable web components called snaplets, which are similar to Java applets. Similarly to Happstack, a developer in Snap can utilize its built-in web server or optionally, to choose between the FastCGI or Hack2 back end. But the Snap server is usually sufficient, since it was designed with efficiency in mind.

There is no official comprehensible manual for Snap, only a collection of documents [15]. In comparison with Happstack, Snap is much younger project and its first version has been published in 2010.

### 3.1.3 Yesod

The third described web framework is called Yesod.[VII] Its name, which comes from Hebrew, means "foundation" in an ecclesiastic sense; it points out that the framework lays firm foundations for web development. The project emerged around a same time as the Snap framework and since then, it has been very active.

Originally, the author of Yesod had maintained a built-in web server as well, but as the time went, he decided to improve it to such an extent that the server could be employed in other configurations. He released it as separate packages that we today know as the WAI and Warp projects.

[III] Homepage: `http://happstack.com/`.
[IV] Homepage: `http://happs.org/`.
[V] Available at the URL: `http://happstack.com/docs/happstack-lite/happstack-lite.html`.
[VI] Homepage: `http://snapframework.com/`.
[VII] Homepage: `http://www.yesodweb.com/`.

Yesod comes with the excellent documentation [16] that fully covers the topic in easy to understand style. Thus the framework can be recommended even for beginners in web developing in Haskell.

### 3.1.4 Other active projects

These three above described projects are the most active and polished ones. But there are other web frameworks that are worth mentioning as their development still have not completely faded away.

#### Miku and Loli

Miku[VIII] is a very simple framework based on the Hack2 interface that can be employed particularly for prototyping and developing small web applications. Its author had written another project, called Loli,[IX] but it was superseded by Miku. Loli is a similarly simple framework which provides everything important in one tiny package.

#### Scotty

The design of Scotty[X] as well as Miku has been inspired by a Ruby web framework named Sinatra. Both has been created with minimalist approach in mind. The main difference lies in the back end — Scotty takes advantage of the WAI interface.

#### Salvia

Project Salvia[XI] contains a modular web server that provides framework features. Its modularity consists in establishing program interfaces together with handlers that specify restrictions for implementations. The developer then chooses the right implementation or write his or her own.

[VIII] Available at the URL: `http://hackage.haskell.org/package/miku`.
[IX] Available at the URL: `http://hackage.haskell.org/package/loli`.
[X] Available at the URL: `http://hackage.haskell.org/package/scotty`.
[XI] Available at the URL: `http://hackage.haskell.org/package/salvia`.

## 3.2 Comparing Happstack, Snap and Yesod

We are going to see main features of the three major Haskell web frameworks. In many ways, they are similar, yet there are things in which they differ. Every description of the framework feature contains a little example, in order to get the picture.

### 3.2.1 URL mapping

When a user sends a request to a web server, the application must decide what content to serve according to the provided URL and possibly by the HTTP method. This is the work of the URL mapper — it calls the right function together with the right parameters by some specified rules.

#### URL mapping in Happstack

In Happstack, the URL mapper is just a collection of monads that prescribes the dispatching of the requests. It tries to find a constraint sequentially until it encounters the one that fits to the request.

```
urls :: ServerPart Response
urls = msum [ nullDir >> index
            , dirs "test" $ ok "Hello, world!"
            ]

index :: ServerPart Response
```

#### URL mapping in Snap

The URL mapper in Snap is very similar to the Happstack one, the difference is that it employs custom monadic functions.

```
urls :: Application ()
urls = route [ ("/", index)
             , ("/test", writeText "Hello, world!")
             ]

index :: Snap ()
```

#### URL mapping in Yesod

Whereas in Yesod, the URL mapper employs the Template Haskell extension, so it is not just a few monads connected together, but a code that generates another code. The URL definitions can also be placed in a separate file.

```
mkYesod "Links" [parseRoutes|
/ Index GET
/test Test GET
|]

getTest :: Handler RepHtml
getTest = defaultLayout [whamlet|Hello, world!|]
```

### 3.2.2 Database handling

When the request is dispatched, the right function is called and the application code is executed, the program needs to manipulate with some data. Normally, the data are stored in a conventional relational database, but that is not the only approach available.

#### Database handling in Happstack

The application model of Happstack is based on a state monad MACID[XII] and separates application logic from the rest of the code. It is a custom Haskell storage for preserving the state of the application.

```
data Person = Person {pid :: Int, name :: String}
newtype People = People [Person] deriving (Typeable, Data)
instance Component People where
    type Dependencies People = End
    initialValue = People []

$(mkMethods ''People ['readPeople])

readPeople :: Query People People
readPeople = ask
```

[XII] In the abbreviation, the M stands for "monad", the rest for "atomicity, consistency, isolation and durability" — the essential attributes of transactional systems.

```
getName :: (MonadIO m) => Int -> m String
getName x = do
    (People ps) <- query ReadPeople
    return . name . head $ filter (\p -> pid p == x) ps
```

#### Database handling in Snap

As we mentioned before, Snap is based on reusable components called snaplets, so it is possible to choose between many database components. For example, it is possible to employ the basic SQL library HDBC through the Snaplet-HDBC package.[XIII] In this case, we just write the SQL queries right into the application.

```
getName :: (HasHdbc m c s) => ByteString -> m String
getName x = do
    r <- query "SELECT name FROM people WHERE pid = ?" [toSql x]
    return . fromSql $ (head r) ! "name"
```

#### Database handling in Yesod

Yesod supports many database management systems through a unified interface called Persistent.[XIV] Similarly to the object-relational mapping, that is known from the world of object-oriented languages, it encapsulates the access to various database back ends. If we write a code for database manipulation, it should uniformly work for completely different database systems.

```
getName :: (PersistBackend b m) => Int -> b m String
getName x = do
    Just (_, p) <- selectFirst [PId ==. x] []
    return $ personName p
```

### 3.2.3 Markup

And finally, when the data are processed, the application must present them to the user by sending back an assembled response, usually in a form of markup language, such as HTML. There are many Haskell libraries that arrange the assembling process for us and they are universally applicable, but we are going to describe the most natural way for each of the frameworks.

[XIII] Available at the URL: `http://hackage.haskell.org/package/snaplet-hdbc`.
[XIV] Available at the URL: `http://hackage.haskell.org/package/persistent`.

### Markup in Happstack

A typical Haskell approach for generating a marked-up output is to use combinators — a set of functions that can be combined together using operators. In Happstack, a preferable solution is to employ the BlazeHtml library.[XV] It supports both fourth and fifth version of HTML and even XHTML.

```
page :: String -> String -> Html
page heading content =
    H.html $ do
        H.head $ do
            H.title $ toHtml heading
        H.body $ do
            if null content
               then H.p $ toHtml content
               else H.p $ toHtml "Hello, world!"
```

### Markup in Snap

It is universally considered as a good practice to separate the markup from the rest of the application. Since some HTML coders do not even know how to program, as they are sometimes specialized in the markup languages exclusively, it is common to introduce a template system. Basically, the templates are the same as static HTML pages. The difference is that they are processed first and a set of selected tags denote places that should be replaced by a generated code. Snap supports Heist,[XVI] a template engine that is inspired by the Lift web framework.[XVII]

```
<html>
    <head>
        <title><heading/></title>
    </head>
    <body>
        <apply template="content"/>
    </body>
</html>
```

[XV] Available at the URL: `http://hackage.haskell.org/package/blaze-html`.
[XVI] Available at the URL: `http://hackage.haskell.org/package/heist`.
[XVII] Homepage: `http://liftweb.net/`. It is written in the Scala language.

### Markup in Yesod

And finally, Yesod employs its own template language called Hamlet.[XVIII] It is not the only available Yesod module for code generation — in addition, we can produce a CSS code with template languages Cassius and Lucius and some JavaScript transformation with Julius.[XIX]

```
!!!
<html>
    <head>
        <title>#{heading}
    <body>
        $if null content
            <p>#{content}
        $else
            <p>Hello, world!
```

[XVIII] Available at the URL: `http://hackage.haskell.org/package/hamlet`.
[XIX] Together, they are known as "Shakespearean templates".

# Chapter 4

# CouchDB and Persistent

> Django may be built for the Web, but CouchDB is built of the Web. I've never seen software that so completely embraces the philosophies behind HTTP. CouchDB makes Django look old-school in the same way that Django makes ASP look outdated.
>
> — JACOB KAPLAN-MOSS[I]

The most frequently used kind of database management system is a relational one. These systems, which are commonly based on a relational algebra and SQL, has been employed for data storage for decades. But that is not the only way of database management. Let us take a look at one particular approach, that has been inspired by the Web.

## 4.1 CouchDB: a document-oriented database system

In the last decade, there has emerged a drift towards abandoning of SQL, that has given rise to structured storages, familiarly known as NoSQL.[II] CouchDB is one of these storages. It takes a database as a loosely organized collection of documents, that can be transformed by views.

CouchDB is written in Erlang, a functional programming language that is fault-tolerant and highly concurrent, i.e., the service can be easily distributed and scaled up. Every aspect of CouchDB is web-centric: from application's point of view, the data access is provided through a RESTful web service; from user's point of view, the databases are administered using a web interface called Futon.

[I] One of the main developers of Django, a web framework written in Python.
[II] The abbreviation is sometimes interpreted as "not only SQL".

### 4.1.1 JavaScript Object Notation

In CouchDB, the data are stored in the JavaScript Object Notation (or *JSON* for short), a lightweight data-interchange format. On top of that, the whole communication resides in sending JSON documents back and forth between the client and the server.

JSON emerged from JavaScript. The format definition is really simple — values are encoded in the following basic data types:

- an empty value: `null`;
- a boolean: `true` or `false`;
- a number: e.g. `42` or `3.1415`;
- a string: e.g. `"foo bar"`;
- an array: e.g. `["one", 2, "three", true]`;
- an object: e.g. `{"foo": "bar", "baz": 2.71}`.

In case a JSON string contains a double quote character, it must be escaped by a backslash. Furthermore, white space is not significant and the arrays and the objects are heterogeneous collections. Complex data structures are possible to create by combining the objects and the other data types.

When we want to work with JSON in Haskell, we can use the `Text.JSON`[III] package, which represents the format as the following data type:

```
data JSValue = JSNull
             | JSBool Bool
             | JSRational Bool Rational
             | JSString JSString
             | JSArray [JSValue]
             | JSObject (JSObject JSValue)
```

The data type definition should be self-explanatory, maybe except for the `Bool` part of the `JSRational` constructor, that signifies a floating point value. As one would expect, the module provides us functions for converting `JSValue` from and to the JSON document.

[III] Available at the URL: `http://hackage.haskell.org/package/json`.

### 4.1.2 Views

Since the data are not stored in relational tables, but in JSON collections, their querying is not straightforward. As explained in [17, chapter 6], they must be transformed first by a view, which is basically a JavaScript code that is stored in a database and that is executed by a view call in order to transform the data.

### 4.1.3 CouchDB in Haskell

To interconnect a Haskell program with a CouchDB database, we could communicate directly through HTTP by sending requests and processing responses, but it is much easier to use the `Database.CouchDB`[IV] library, that has been created for this purpose. The library provides functions for establishing a connection with the database system, for management of documents and for creation and querying of views. Every interaction with a database is encapsulated in a custom monad, which is defined as follows:

```
data CouchMonad a = CouchMonad (CouchConn -> IO (a, CouchConn))
```

The `CouchConn` data type represents a connection to a CouchDB database, so `CouchMonad` actually executes an exchange of JSON documents between the program and the database.

As an illustrative example, we are going to store information about a person to a CouchDB database named `south_park`:

```
> conn <- createCouchConn "localhost" 5984
> let sp = db "south_park"
> let eric = JSObject $ toJSObject
             [("name", JSString $ toJSString "Eric Cartman"),
              ("age", JSRational False 9)]
> (doc, rev) <- runCouchDBWith conn $ newDoc sp eric
```

We have established a connection with a local CouchDB server, created a JSON document and sent it to the database. The interaction then returned a pair that contains a document identifier and its revision. We can use this information later, for example to get the document back from the database:

[IV] Available at the URL: `http://hackage.haskell.org/package/CouchDB`.

```
> do {(Just (_, _, x)) <- runCouchDBWith conn $ getDoc sp doc;
      putStrLn . render $ pp_value x}
{"_id": "7d4ffcae98cdba9a7f6992470a00115e",
 "_rev": "1-28be4e4fca34e9811b4fbc85eb7aaea4",
 "name": "Eric Cartman", "age": 9}
```

This time, the interaction has returned a `Maybe` triple, since it is not certain whether a document with the given identifier occurs in the database. The first two components of the triple again signify the identifier and the revision, so they are just omitted and only the third value is passed on. Furthermore, we have printed a human-readable representation of the document using the `pp_value` and `render` functions from the `Text.JSON.Pretty` module. The `_id` and `_rev` keys points to textual forms of the identifier and the revision, respectively.

## 4.2 Extending Yesod for CouchDB

Why we have dealt with the CouchDB database system in the foregoing text? CouchDB is not even written in Haskell, let alone it does not relate to the web development directly, although it is an example of a successful web application.

The reason is that we had created an interface between it and the Yesod web framework, in order the Yesod users could store their data in CouchDB databases through the Persistent module.

### 4.2.1 Implementing the Persistent back end

To implement own database back end for the Persistent module, it is necessary to write an instance of the `PersistBackend` type class. Currently, it contains 13 functions that select, insert, update and delete the data. For this purpose, we have employed the above described `Database.CouchDB` module. The most difficult part was to filter the data — with a relational database system, we would normally form an SQL query that would choose only the items we are interested in. But with CouchDB, it is necessary to write a view first, so we had to generate a proper code in JavaScript, save it as a view and only then we could execute it and get the desired data.

The further step is to convert the JSON documents to a Persistent internal data type. This has been done by writing the `PersistValue` instance of the `JSON` type class, which means that the values are converted implicitly.

The last part is to provide functions for establishing a database connection and a configuration structure, together with an instance of the `PersistConfig` type class. Persistent introduces a connection pool that maintains the connections for us and the biggest problem lies in choosing the right pile of monads that allows us to utilize the connection within the back end functions. In our case, we had written a reader monad named `CouchReader`. It instantiates the `MonadBase`, `MonadBaseControl` and `MonadTransControl` classes. Afterwards, we were able to work with the established database connection in our implementation.

# Chapter 5

# Conclusions and future work

> The trouble with programmers is that you can never tell what
> a programmer is doing until it's too late.
> — SEYMOUR ROGER CRAY[I]

The theoretical goal of this thesis has been to present the fundamentals of development applications for the Web with the Haskell programming language. Namely, we have presented the principles of HTTP, a protocol on that the Web is built; furthermore, we have shown a few examples of serving content by web servers written in Haskell; we have also introduced Haskell web frameworks, that are convenient for building nontrivial web applications, and compared them; and finally, we have become acquainted with the CouchDB database system and looked under a hood of Persistent, a database interface used by the Yesod web framework. And the most importantly, we have seen that there is no single way to write a web application in Haskell.

The practical part of our thesis resides in writing a module that implements the CouchDB back end for Persistent. The outcome is far from perfect, but the code works and we have sent it to the Persistent developers that are going to integrate it into their codebase. During development of the module, we have also made minor changes to the `Database.CouchDB` library that are vital for error-free running of the CouchDB back end. All the code is available online under a permissive license.

Before we proclaim our code to be stable, we must test it properly in order to enhance the overall robustness. Also, there is much room for improvements of its efficiency.

[I] A supercomputer architect, the so-called father of supercomputing.

As for the future work in general, there are many ways to extend the Haskell libraries for web development. We can inspire ourselves by progress of the popular web frameworks written in other languages, such as Django, Nette or Ruby on Rails. The biggest pain of writing web application in Haskell is the volatility of the packages — almost everything is in an experimental phase and the projects are rapidly changing all the time, so there is no assurance that our application is going to work with a newer version of a chosen framework. Moreover, the learning curve is usually placed very high and the benefits of developing web applications in Haskell become evident in the long term. To conclude with a more positive fact, there is no problem with the code performance, which is outstanding in overall (see [10, page 82] and [14, page 87]).

Figure 1: A froggie.[18, page 21]

# Bibliography


[1] Tim Berners-Lee. Longer biography, 2011. Available at the URL `http://www.w3.org/People/Berners-Lee/Longer.html` (November 2011).

[2] Robert H'obbes' Zakon. Hobbes' internet timeline 10.1, December 2010. Available at the URL `http://www.zakon.org/robert/internet/timeline/` (November 2011).

[3] Simon Peyton Jones. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003. Also available at the URL `http://haskell.org/onlinereport/` (November 2011).

[4] Miran Lipovača. *Learn You a Haskell for Great Good!* No Starch Press, 2011. Also available at the URL `http://learnyouahaskell.com/` (November 2011).

[5] Paul Hudak, John Peterson, and Joseph Fasel. A gentle introduction to Haskell. *SIGPLAN Notices*, 27(5):1–52, May 1992. The updated version available at the URL `http://haskell.org/tutorial/` (November 2011).

[6] Simon Thompson. *Haskell: The Craft of Functional Programming*. Pearson Education Limited, 2nd edition, 1999.

[7] Bryan O'Sullivan, John Goerzen, and Donald Stewart. *Real World Haskell*. O'Reilly Media, 2009. Also available at the URL `http://book.realworldhaskell.org/read/` (November 2011).

[8] R. Fielding et al. RFC 2616: Hypertext Transfer Protocol — HTTP/1.1, June 1999. Available at the URL `http://www.faqs.org/rfcs/rfc2616.html` (December 2011).

[9] D. Robinson et al. RFC 3875: The Common Gateway Interface (CGI) version 1.1, October 2004. Available at the URL `http://www.faqs.org/rfcs/rfc3875.html` (December 2011).

[10] Michael Snoyman. Warp: A Haskell web server. *IEEE Internet Computing Magazine*, 15(3):81–85, May–June 2011.

[11] Johan Tibell. Haskell-Cafe mailing list: Hyena announcement, June 2009. Available at the URL http://www.haskell.org/pipermail/haskell-cafe/2009-June/063058.html (December 2011).

[12] Alson Kemp. Reflections on leaving Haskell, March 2010. Available at the URL http://www.alsonkemp.com/haskell/reflections-on-leaving-haskell/ (December 2011).

[13] Jeremy Shaw. Happstack crashcourse, 2011. Available at the URL http://happstack.com/docs/crashcourse/index.html (December 2011).

[14] Gregory Collins and Doug Beardsley. The Snap Framework: A web toolkit for Haskell. *IEEE Internet Computing Magazine*, 15(1):84–87, January–February 2011.

[15] Gregory Collins et al. Snap docs, 2011. Available at the URL http://snapframework.com/docs (December 2011).

[16] Michael Snoyman. Yesod web framework book, 2011. Available at the URL http://www.yesodweb.com/book (December 2011).

[17] J. Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide*. O'Reilly Media, 2010. Also available at the URL http://guide.couchdb.org/editions/1/en/ (January 2012).

[18] Antonín Pavelka. Funkční anotace proteinových segmentů. Bachelor's thesis, Masarykova univerzita, 2006. Available at the URL http://is.muni.cz/th/99207/fi_b/ (January 2012).

# Appendix — on the Haskell package system

The easiest way to begin programming with Haskell is to install the Haskell Platform,[I] a software suite containing the up-to-date version of the Glasgow Haskell Compiler and a few fundamental programs. In particular, these programs are intended for software development and for installing Haskell modules.

Haskell community produced a lot of modules that extend the basic language functionality. The amount of work that is devoted to the ecosystem around the language increases steadily. In order to manage such an effort, a package system called Cabal[II] was created. The system is able to build module packages that can be easily compiled on other computers. Furthermore, a package repository called Hackage[III] emerged. The repository is an internet service in which all the published packages are located.

To simplify the installation even more, the packages can be downloaded and installed by a command line tool named `cabal-install`.[IV] Its biggest advantage lies in a dependence handling, i.e., if we want to installed a package `X` that depends on a package `Y`, the package `Y` is automatically downloaded together with all its dependencies and so forth.

## Installing Haskell packages

To install a package, we have to know the exact name of the package first. We can search for it by the search engine Hayoo![V] or look through the package list on the Hackage web site. It is better to use the `cabal` command directly, though.

[I] Available at the URL: `http://hackage.haskell.org/platform/`.
[II] Homepage: `http://haskell.org/cabal/`.
[III] Homepage: `http://hackage.haskell.org/`.
[IV] Available at the URL: `http://hackage.haskell.org/trac/hackage/wiki/CabalInstall`.
[V] Available at the URL: `http://holumbus.fh-wedel.de/hayoo/hayoo.html`.

Whenever we use the `cabal` command, we have to work with the latest package changes. This can be done by invoking the `cabal update` command that downloads the up-to-date package list. Then we are able to search through the list by the `cabal list` command and finally, after finding the right package, to install it by the `cabal install` command.

In case we do not have access to the `cabal-install` tool (or when we lack the direct internet access), the packages can be built manually. It is usually a very tedious process that consists of downloading the package, unpacking it and then invoking the following series of commands:

```
cabal configure
cabal build
cabal copy
cabal register
```

On rare occasions, we would not have even the package system installed. In these awkward situations, these commands would be rather utilized:

```
runhaskell Setup configure --user --prefix=$HOME
runhaskell Setup build
runhaskell Setup install
```

For the sake of completeness, we must mention that the packages are installed into the `~/.cabal` directory and their metadata dwell in the `~/.ghc` directory by default.