fhwedel
UNIVERSITY OF APPLIED SCIENCES

Department of Computer Science

Master's Thesis

Extending the Hawk Framework

Creating real world web applications using Haskell

Submitted on:

May 19th, 2010

Submitted by:

B. Sc. Alexander Treptow
Tannenallee 43
22844 Norderstedt, Germany
Phone: +49 (40) 522 57 51
E-mail: alexander.treptow@googlemail.com

Supervised by:

Prof. Dr. Ulrich Hoffmann
Fachhochschule Wedel
Feldstraße 143
22880 Wedel, Germany
Phone: +49 (41 03) 80 48-41
E-mail: uh@fh-wedel.de

Prof. Dr. Uwe Schmidt
Fachhochschule Wedel
Feldstraße 143
22880 Wedel, Germany
Phone: +49 (41 03) 80 48-45
E-mail: si@fh-wedel.de

Extending the Hawk Framework

Creating real world web applications using Haskell

Master's Thesis by Alexander Treptow

Nowadays interactive web applications become a growing part in every day life. There is a huge variety of frameworks for supporting development of such web applications. The Hawk framework is written in the non-strict pure functional programming language Haskell, which offers an expressive syntax and an extended type system. It is optimal for creation of interactive web applications. This thesis will give an example architecture for a real world web application using the Hawk framework as well as an abstraction layer for the search engine framework Holumbus to uncouple web application from search engine. Therefore the search web application Hayoo! API Search is reimplemented using the Hawk framework, an abstraction layer is developed and Hawk is extended within the specifications.

The template [RP09] of Björn Peemöller and Stefan Roggensack was used for the layout.

Contents

List of Figures

List of Listings

1 Introduction

Since the World Wide Web was made public in the early 1990s, its usage highly increased as well as its importance in our everyday life and its dimensions. While the number of web sites growth exponential until its beginning from just a few in 1991 to more than 200 million in February 2010 [Net10], the web becomes an important channel of information distribution and retrieval in our information society.

Not only did accessibility and technology evolve, but also did the web sites during the last twenty years. From mainly static sites without further content computation in the beginning over mostly server side dynamically calculated sites a few years ago to mixed calculated web sites, where the main content is produced by a server or the server only sends a script or application which produces the content on client side. Based on the current technology, it is now possible to provide entire applications with web browsers as user interface. In order to manage the increasing complexity, such applications are typically realized based on *web application frameworks*. These frameworks support developers by providing the most common tasks of web development. This eases the creation of such interactive web applications [PR09, Ch. 3].

A huge amount of frameworks is available, for example client-side, server-side, database and search engine frameworks. Some of them covering nearly the same or complement one another. Collaboration of frameworks need to be done through a common data format or one of the frameworks have to encapsulate the other by offering an interface to it.

1.1. Motivation

Some web frameworks appeared in the last few years or have been extended, mainly in functionality and performance, but there is still room for future development. Many of these currently used frameworks are based on scripting languages, which are mainly typed dynamically and have to be interpreted in runtime. So they are slow, vulnerable to errors and need more resources than languages with a strong type checking system and compilation. Pros of scripting languages are an expressive syntax, that reduces lines of code and they support fast testing, because of no prior compilation time needed.

The Hawk framework uses the non-strict functional language Haskell, which creates really fast web applications and got a more expressive syntax than most web frameworks, comparable to Ruby on Rails [Ruby]. Haskell has a strong type checking, so there are much less errors due to runtime.

The Hawk framework lacks support for fast interactive web applications. Contemporary web applications often use *Asynchronous JavaScript and XML* (Ajax) to update parts of the web application without reloading the whole page. This saves bandwidth costs and time to transfer data. Additionally Ajax can use a much less verbose data exchange format than XML . So web applications appear to users more and more like desktop applications. Sproutcore [Sproutcore], a JavaScript framework, shows that the trend is up to a mix of desktop and web applications [Jol]. It will result in an architecture where data is accessible from everywhere and applications appear as fast and interactive as desktop applications. The HTML Specification number 5 is another huge step in this direction [HH09].

Another motivation behind this thesis is the concepts of uncoupling and design architectures on applications. Frameworks are the biggest challenge in designing a library, because they will be used in many different contexts and should be flexible and easily extensible [GHJV94, p. 26ff].

1.2. Scope

The objective of this work is to develop an example architecture for a web application using the web application framework Hawk [PR09] and the search engine framework Holumbus [Hüb08].

Both frameworks are written in the functional programming language Haskell. Hawk is shorthand for the permuted acronym *Haskell Web Application Kit*. It is a framework for creating web applications using the *Model-View-Controller* pattern (MVC). Holumbus is a search engine framework to create highly specialized and configurable search engines.

Additionally the example web application should be a real world web application that will be used later. So this thesis shows up a reimplementation of the *Hayoo! API Search* (Hayoo) using the Hawk framework. Hayoo is a search engine, that searches the Haskell API on hackage [hackage]. It uses the Holumbus framework for searching functionality. Hayoo will be extended by search customization and a structure for search engine web applications will be presented, that wraps Holumbus specific data and configuration from web applications. Also it presents how to reuse this architecture for other web applications.

Hawk will be extended by authentication and *JavaScript Object Notation* (JSON) support for exchanging with client side Ajax [JSON] as well as an application storage type to keep application data in main memory.

This work does not cover an Indexer for a Holumbus search engine.

1.3. Related Work

A lot of web frameworks are available. *Ruby on Rails* and *Zend Framework* primarily influenced extensions of the Hawk framework [Ruby, Zend]. Additionally the Haskell community supports some useful specific libraries, e.g. a JSON representation in Haskell. Also a set of more comprehensive libraries is offered. The *Haskell Application Server Stack* (Happstack) provides a wide range of functionality for developing web applications [Ht], but it does not support databases. Only files and the main memory are used to store data.

Hayoo's current implementation is closely related to the one developed in this thesis as well as the *Hoogle* search engine [Mit04]. Both currently do not support user defined search settings or advanced search configurations.

1.4. Outline

To give a short overview for the following chapters, chapter 2 provides some fundamental information about increasing web application interactivity, individualization in web applications and authentication. Prior to the implementation chapter 3 analyzes scope and problems of this work. Chapter 4 describes extensions that were implemented to the Hawk framework. After this, the reimplementation of Hayoo is pictured in chapter 5. The subsequent chapter 6 illustrates how to reuse the Holumbus Wrapper, which uncouples web application from search engine framework Holumbus.

Chapter 7 gives a short conclusion of this work, which analyzes and compares reached goals and discusses future development of the Hayoo page and Hawk framework.

In Appendix A a script for creating a base structure for web applications with hawk is presented as well as a simple guestbook web application.

2 Fundamentals

For the following chapters the reader is expected to be familiar with the functional programming language Haskell and basic Haskell principles like data types, type classes and functions. It is also recommended to read Timo B. Hübel's thesis about *The Holumbus Framework* [Hüb08] and Björn Peemöller's and Stefan Roggensack's thesis about *The Hawk Framework* [PR09], as this thesis heavily is based on them.

This chapter covers common fundamentals in web application development in addition to the upward mentioned fundamentals. It introduces the reader to follow the architectural and structural decisions made in the implementation. Therefore a short introduction to web frameworks in section 2.1 is given to spot the differences to the previous implemented features. Section 2.2 describes interactivity improvements in nowadays web applications to behave like desktop applications. Authentication and its forms of appearance are analyzed in section 2.3. Subsequently section 2.4 describes concepts of individualization in web applications in general as well as search engine individualization in special.

2.1. Web Frameworks

This paragraph gives a short description of web frameworks in general and a little more specific description of the Hawk framework, with a focus on the architecture of web frameworks.

2.1.1. Web Frameworks in General

Most web frameworks are implemented using the MVC pattern. Web pages in general are handled like shown in figure 2.1. A client requests a web server through HTTP.

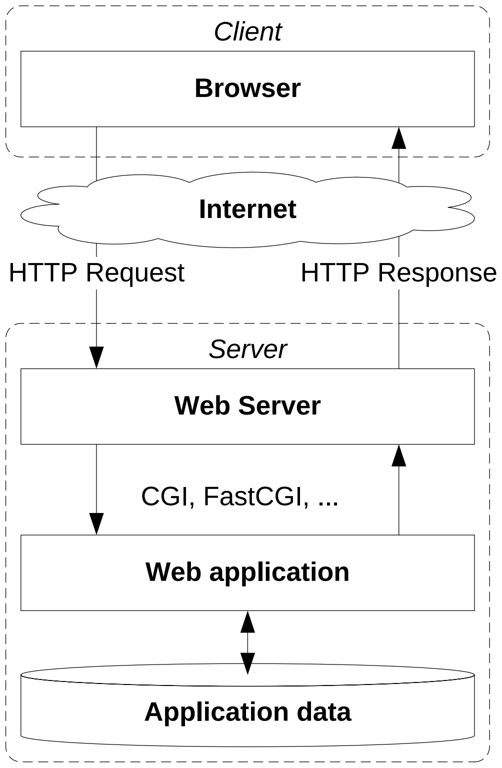


Figure 2.1.: Client-Server interaction on web page requests

The web server processes the request and generates a response. Often data from a database is used for generating the response. If the client receives it, the response may be further formatted by client side processing e.g. on Ajax requests.

To ensure higher reusability and maintainability most web frameworks recommend to implement web applications using the Model-View-Controller pattern (2.2). This

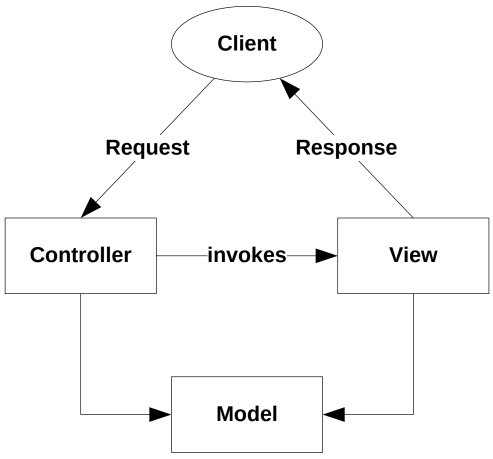


Figure 2.2.: Model-View-Controller pattern

pattern classifies an application to the three elements model, view and controller. The model contains functions working on permanent data, a bidirectional mapping between database data and an application internal representation as well as interaction with databases. The view formats data for the response. Often a template engine is used to merge data from the controller and a given template to a response that makes sense to the client. The controller receives forwarded requests from the web server and coordinates their processing. Additionally the controller defines functions which not depend on model or database.

The framework is either bound into the web server or communicates with it e.g. over a fast common gateway interface (FastCGI) and offers different components for commonly used web application functionality. So it is much faster and easier to create a web application using a framework. The following figure (2.3) depicts how web frameworks are structured in general. A client also requests the web server, which forwards the request to a controller. A controller will process a request by querying the model and pushing data to the view for formatting the output. A model often uses a database or cache for offering a state to an application.

2.1.2. Hawk's Architecture

The Hawk framework roughly works like most web frameworks. Figure 2.4 depicts Hawk's architecture and data flow. The *Front Controller* receives requests and calls the requested action, which may interact with the model and receives data from

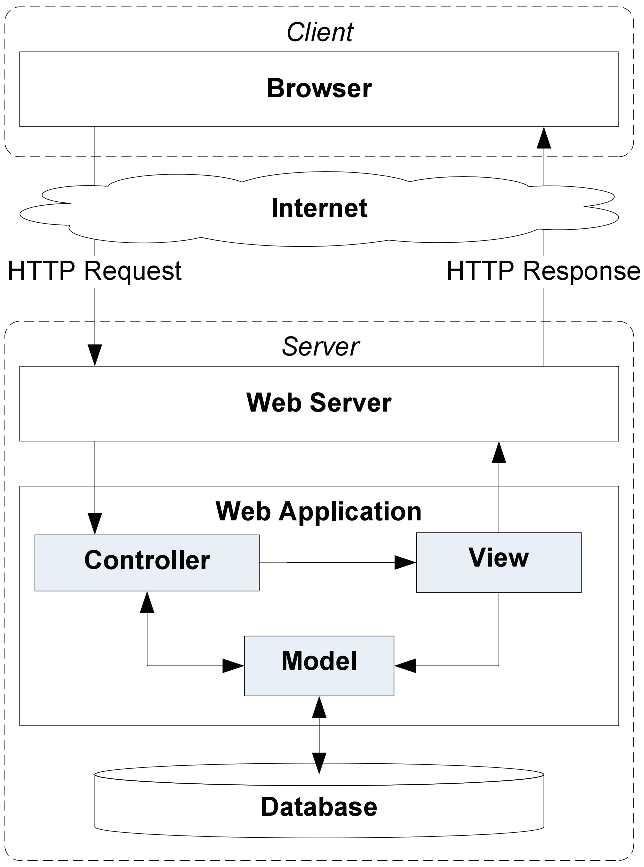


Figure 2.3.: Architecture and data flow for web frameworks in general

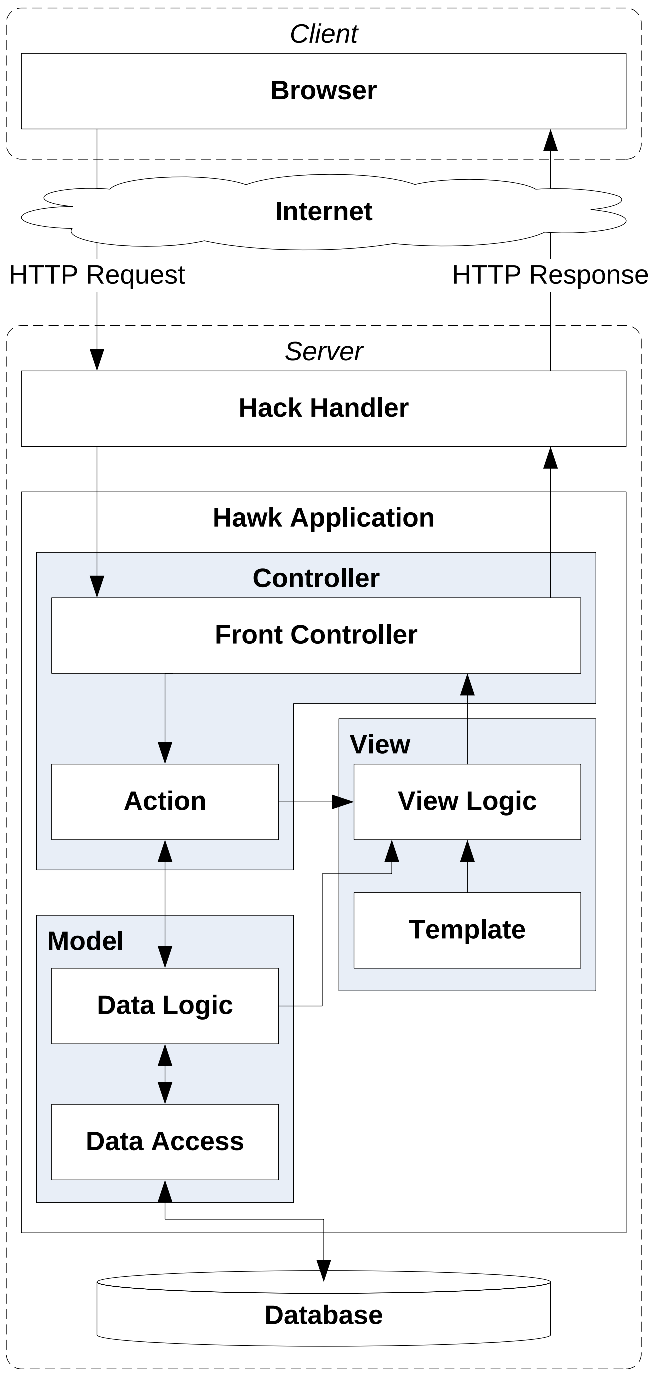


Figure 2.4.: Architecture and data flow of a Hawk application

the database. After the action is processed it triggers the view for generating a response. The view generates the response from data given by the action, a template and maybe some additional data from the model. The formatted response is passed through *Front Controller*, hack handler and network to the client [Wan09]. Whereas the hack handler is a simple web server application like Apache Tomcat or Yaws [Fou, yaw].

To spot the differences between this and the previous thesis as well as setting the new components in context. Figure 2.5 shows the previous components structure of Hawk.

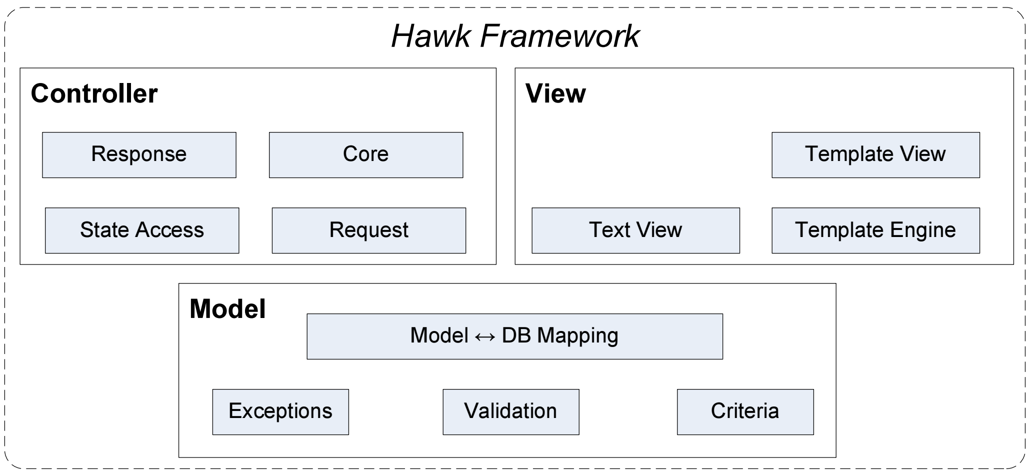


Figure 2.5.: Components of the Hawk framework

The Hawk framework is a collection of functionalities, except of the *Core*, which additionally encapsulates a web application. A detailed work flow of the *Core* is given in figure 4.2. The *Controller* also offers functionalities for accessing request data, generating responses and providing access to cookies and session data as well as the data types, which are part of the framework core.

The *View* provides different viewer modules for viewing dynamically generated text or template pages and the template engine for filling a template with values and assuring type safety. The *Model* provides a mapping between application models and database, as well as the exceptions that may occur on database querying. It also offers query generation via *Criteria* and validation of inserted or updated data.

2.2. Improvements on Web Applications

There are several approaches and architectures in current web technology that try to realize web applications which act like desktop applications to the user. Desktop applications perform well with nearly no latency, low usage of memory and CPU resources in comparison to current web applications. Animated visual information is displayed in real-time by hardware acceleration.

Web applications poorly lack these advantages. There is a high latency by transferring information via network. Most markup languages like XHTML [IM07] or XML are human readable, but therefore too verbose [BPM+08]. They increase the amount of information transferred, the amount of memory used as well as they need more CPU resources to parse that redundant information.

Another disadvantage is, that all document information is transferred on each request and the user has to wait on every request until the server responds, because normal HTTP requests are handled synchronous [KL00].

As described in *Maximizing Human Performance* an application should respond in $\frac{1}{10}$th second [Tog01], if an application do not respond in half a second it has to offer a time indicator e.g. a hour glass. An application that shows up a hour glass on every user action is inapplicable.

Many web applications lack these restrictions. Mainly because of the time needed for transferring data. Fortunately there are some approaches that increase performance of web applications. So paragraph 2.2.1 depicts approaches for improving interactivity, especially asynchronous communication between client and server and subsection 2.2.2 depicts performance improvements by lowering the needed resources, e.g. bandwidth, memory and CPU.

2.2.1. Interactivity

There are several technologies that bring more interactivity to web applications, than synchronous HTTP requests with HTML and *Cascading Style Sheets* (CSS) [JRH99, cBHL09]. In the beginning there were only Java Applets, that can load code dynamically and JavaScript on page load [Int09]. Nowadays there are many other ways, like Flash with ActionScript, Shockwave, Visual Basic Script or Ajax.

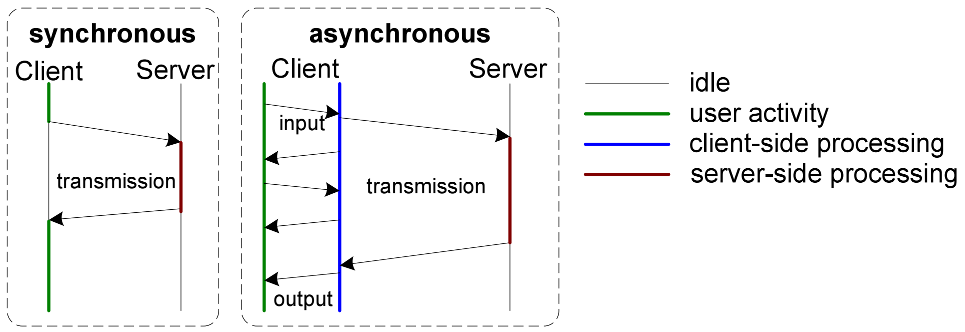


Figure 2.6.: Synchronous compared with asynchronous interaction

All these approaches have in common that the code is processed at runtime on the client and they can load new information from a server without reloading the whole web page. So they all acting asynchronously. This gives an impact on usability by not waiting with an empty page on the server response. This difference is depicted in Figure 2.6.

From the above mentioned approaches Ajax is most commonly used, as it is available in nearly every client. The others need extra plug-in's installed on client side.

The naming Ajax was first used by *adaptive path* and described as "Ajax is a set of technologies being used together in a particular way" [Gar05]. So Ajax is a technology and an architecture. It combines the use of JavaScript, whereas every other *Document Object Model* (DOM) manipulating scripting language can be used, CSS, DOM and *XMLHttpRequest* (XHR) [NCH$^+$04, Kes09]. For DOM or HTML manipulation also XML with *Extensible Stylesheet Language Transformation* (XSLT) can be used. (X)HTML and CSS is used for mark up and style information. DOM starting with Level 3 can be accessed by JavaScript to dynamically manipulate and therefore display information or e.g. read form data. This allows the user to directly interact with the web application. The XMLHttpRequest object and JavaScript provide a method for exchanging data asynchronously between server and client, which is a web browser in most cases. JavaScript is also used to connect all mentioned standards together.

Ajax represents a way, trying to realize web applications with a level of interactivity likely to desktop applications. Because of its asynchronous behavior the web page seems to be one application, not several pages.

XHR is a DOM API that can be used by JavaScript to send HTTP requests to a web server asynchronously. The response data is available as plain text and XML document, if it is valid XML markup. Although the plain text can be only HTML and replace the current page or it may be JSON [Int09, p. 9] and therefore evaluable by JavaScript or it is just plain text that needs to be processed in a custom way. Each major web browser support XHR and it is used for web service communication, too.

There are several frameworks that support developers on using Ajax. For example the most popular ones are jQuery, prototype, Dojo, *Google Web Toolkit* (GWT) and JsHttpRequest [jQu06, pro06, Doj, GWT06, JsH06]. Whereas jQuery, Dojo and GWT are web frameworks and not just JavaScript frameworks as prototype. JsHttpRequest is restricted to XHR functionality, so it is the tiniest in lines of code.

In conclusion Ajax is a general term for already delivered web page manipulation using asynchronous request to load additional data.

2.2.2. Performance

Ajax can be used instead of synchronous page request, because Ajax offers some strong performance impacts and for users only the browsers hour glass is changed to one from the web application. On first sight the user can interact with the web application while something is loading in the background and therefore the usability is increased, but Ajax digs much deeper.

Most of the information to display a web page is loaded once, e.g. CSS files, JavaScript files, images, etc., and cached. But on each normal page request the whole HTML file is loaded. With Ajax or other such techniques only the information that need to change is loaded from the server. That means no reload of HTML head, page header, menu, footer, and so on, on each request. *Web Performance Inc.* did a little performance test about it and achieved a cut of bandwidth used by over 60%, just for reloading changed information instead of reloading the whole web page [Mer06]. With each per cent of saved bandwidth comes a huge cost saving on consumer and producer side. Especially in large-scale web applications or ones with very tight bandwidth considerations, e.g. for mobile devices. Although there comes time saving as well as memory and CPU resource saving with it, which brings us closer to the goal of deskop like web applications.

In the case of JavaScript plain text is formatted as JSON, it can be evaluated within JavaScript to create an object for use on the current DOM. The edge of JSON is that it is less verbose than XML, what directly improves performance, because much less bandwidth is required as well es less memory and computation time to parse it [Lac06].

If a lot of JavaScript code is used in a web application you can boost its initial load by dynamically loading JavaScript code [Mah06]. This approach is provided by some JavaScript Frameworks e.g. Ajile, JSAN Import System or Dojo Package System.

2.3. Authentication

Authentication can cover different things. In subsequent it is only used by the meaning of confirming that a client or user is who he pretends to be.

Authentication is closely related to authorization, often it is used as a synonym or authorization is used instead of authentication because in most cases a user that is authorized, have to be authenticated first. The difference between those concepts is, that authorization only verify if someone has the authority to perform an operation or to access a certain source. Authorization do not need authentication. The subject, which accesses a source, can be unknown, while it is allowed to access the source by knowing the password or answering a *Completely Automated Public Turing test to tell Computers and Humans Apart* (CAPTCHA). Captcha's authorizes users as humans, for example to let them create user accounts. Authentication only verifies whether a subject is who it pretends to be, therefore it has to transmit its identity, e.g. name, location, id, biometrics, etc.

In the first paragraph an overview of the most common types of authentication is given and the second depicts some often used ways of authentication in web applications.

2.3.1. Types of Authentication

There exists several types of authentication that can be subsumized to five groups: Password login, *Single Sign-On* (SSO), biometrics, digital certificates and *Public Key Infrastructure* (PKI). Biometrics authentication is rarely used in web context and PKI is nearly only used to authorize servers not clients. So they are not relevant in this context. Only password login, SSO and digital certificates are depicted below. All this types of authentication can be combined, for example SSO requires a login password in most cases.

Passwords login

Password logins almost always require an user ID and a password, whereas the user ID can be public and is visible on input, the password is visually hidden. But most types of password login poorly lack hidden transmission. The password is then submitted in plain text or Base64 encoded which makes no difference.

If the sever receives such a login request it will usually search its database, an authentication file or something like this for the ID and compares the corresponding password with the one submitted by the client. Values of that type of authentication are Basic or Digest HTTP authentication or form-based authentication, which are described in subsection 2.3.2.

Single Sign-On (SSO)

SSO is what it says. An user logging in once and get access to all systems related to his account, without being prompted again. There are some centralized and decentralized services supporting this type of authentication. They also have to offer different data for each page the SSO ID is used at. So users are able to logging in to that system. Some well known SSO services are for example Microsoft Passport, OpenID and Kerberos.

Digital Certificates

A digital certificate is structured data that confirms a public key and further information. For a certificate system a *Certification Authority* (CA) or a public registry, that holds the public keys, is needed. If a user tries to authenticate himself to a system, he encrypts its ID, name or a public string with its private key. The receiving system can decrypt it with the users public key [CSF+08]. This type of authentication includes SSL/TLS, where HTTPS is the part used for authenticating a user to a server [Res00].

2.3.2. Methods of Authentication

The authentication methods described are all in sight of HTTP use only, because its the most valuable protocol for web applications. The most popular authentication methods including basic and digest access authentication [FHBH+99], as well as OpenID [RF06], HTTPS [Res00] and form based authentication.

Basic Access Authentication

Basic access authentication is a basic authentication method supported by all popular web browsers. It is rarely used on public web pages, but may be used for some parts of a web page e.g. an *Administrator Control Panel* (ACP). This is restricted to small systems, because this method does not use a secure channel for password transmission, as do *digest access authentication* described later in this section.

This method is not capable for use on normal public web pages too, because it annoys the user with an extra window popping up.

Basic access authentication is an easy to implement method. If a client tries to access a certain web page that requires authentication the server will send back a `401 Authorization Required` code and a `WWW-Authenticate` with the first word "Basic" and a key value pair with the key `realm`.

A client has to respond to that by adding an `Authorization` HTTP header to the request also starting with `Basic` and a Base64 encoded string of the type `<user ID>:<password>` [Jos03]. Base64 does not encrypt. So an attacker easily can figure out the password as if its transmitted in plain text.

Client request 1:

```
GET /dir/index.html HTTP/1.1
```

Server response 1:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic real="Secure Area"
```

Client request 2:

```
GET /dir/index.html HTTP/1.1
Authorization: Basic SGF5b286SGF3aw==
```

Now the server decide whether to transmit an `OK` or `Unauthorized` response.

Form Based Authentication

Form based authentication is like basic access authentication. It is just the smarter way and without using HTTP response codes and headers. It is maybe the most commonly used method on web pages, although it also lacks security, as all data is passed in plain text.

On a client request the server will return a normal public web page. Full functionality is only accessible for authorized users. This page contain a standard HTML form normally with two input fields for user ID and password. So if the user submits the form, all data is passed in plain text as HTTP post parameters to the server.

Pros of this method are, that the user is not bothered by an input dialog and it is fast and easy to implement. Contra is that it is totally insecure, if its not used via a *Secure Socket Layer / Transport Layer Security* (SSL/TLS) encrypted connection [DR06].

Digest Access Authentication

Digest access authentication intends to supersede the unencrypted basic access authentication method by not sending a password in plain text over a network. Basically digest access authentication is applies the cryptographic *Message-Digest algorithm 5* (MD5) hashing function to avoid cryptanalysis [Riv92].

This method works like basic access authentication, it uses the `401` HTTP response code and the dialog window, as well as the same HTTP headers for transferring that information. The only difference is in the headers content. The keyword clearly will be `Digest` instead of `Basic` as depicted below.

Client request 1:

```
GET /dir/index.html HTTP/1.1
```

Server response 1:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Digest
        realm="testrealm@host.com",
        qop="auth,auth-int",
        nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
        opaque="5ccc069c403ebaf9f0171e9517f40e41"
```

Client request 2:

```
GET /dir/index.html HTTP/1.1
Authorization: Digest username="Mufasa",
        realm="testrealm@host.com",
        nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
        uri="/dir/index.html",
        qop=auth,
        nc=00000001,
        cnonce="0a4f113b",
        response="6629fae49393a05397450978507c4ef1",
        opaque="5ccc069c403ebaf9f0171e9517f40e41"
```

The meanings of the values of the `WWW-Authenticate` and Authorization parameters can be looked up at RFC 2617 subsections 3.2.1 and 3.2.2 [FHBH+99]. The `nonce` parameter combined with MD5 assures the security of HTTP Digest authentication.

Essential is how the response hash is generated with MD5, to encrypt the password. A hash can not be decrypted, so the server have to generate a hash the same way with the users password from its DB and compare the hashes for verification.

```
Part1 = MD5 ( "Mufasa:testrealm@host.com:Circle Of Life" )
      = 939e7578ed9e3c518a452acee763bce9

Part2 = MD5 ( "GET:/dir/index.html" )
      = 39aff3a2bab6126f332b942af96d3366

Response = MD5 ( Part1 ++
                 ":dcd98b7102dd2f0e8b11d0f600bfb0c093" ++
                 ":00000001:0a4f113b:auth:" ++
                 Part2 )
         = 6629fae49393a05397450978507c4ef1
```

OpenID

OpenID is shorthand for open identification and its a decentralized system for web pages and other web services. A user using this system only needs to login once to his OpenID provider and can from now on login to each web page that supports this functionality, only by send its so called OpenID to the web page. No user ID or password is required to transmit over the network. An OpenID is an *Uniform Resource Locator* (URL) which will be passed from user through the web page to an OpenID provider that will return the users information to the web page after an optional confirmation of trustworthiness of the web page.

Pages using this technique can stop using form based authentication on their web page. This comes with some benefits, as the web page provider do not need to care about security leaks on user data or remember password functionality.

It eases the users administration of their data, because they only need to change the data at the OpenID provider. The web page will automatically get this data next time the user visits it.

HTTPS

This is the secure version of the Hypertext Transfer Protocol, it sets the HTTP on top of a SSL/TLS secured connection. HTTPS uses standard port 443 instead of 80 and is mostly used to authenticate servers and for secure transmission of data. Client authentication via HTTPS is very rare.

A network independent secure connection is established by the SSL handshake protocols where a secure identification an authentication is done between client and server. After that a symmetric session key is exchanged through an asymmetric encryption e.g. Diffie-Hellman. The symmetric session key then is used to encrypt and decrypt all information transmitted between client and server.

This is a very common used authentication method. It is like form based authentication just summed up with security. Only the web server has the extra charge to provide HTTPS, need to have a digital certificate to authenticate itself to the user and needs one IP address per host name in the current SSL/TLS version. Most browsers do not require the one IP per host name, they use *Server Name Identification* (SNI) instead [BWNH+03] and it is possible to authenticate a user to a server using a digital certificate.

2.4. Personalization and Customization

In this section the different types of personalization and customization are discussed, as well as the differences between these nearly equal terms. In subsection 2.4.1 they are discussed in general purpose in information technology scope. In 2.4.2 a special eye is put on personalization and customization with the scope on search engines for an example architecture.

From now on personalization means implicit personalization and customization means explicit personalization. A more detailed description is given at the end of the "Personalization and Customization in General" part.

2.4.1. Personalization and Customization in General

Personalization and customization have a lot in common, in most cases they are not distinguished. But there *are* little differences in scope of information technology, they will be depicted in the following subsections.

Personalization (implicit personalization)

Personalization in information technology means to adjust programs, services or information to the users skills, needs and favors. Precondition for successful personalization is identification of unique users, e.g. if there are several users that uses the same account, it is nearly impossible to determine the preferences of the user that currently uses it.

Personalization can be differed into three techniques.

1. **Rule based** personalization is a common technique to personalize contents with user profiles by a predefined, mostly fixed rule base, e.g. for printed catalogs.
2. **Community based** personalization, also called collaborative filtering, observes user groups to do determine their characteristics and to map them to single users to get their preferences. Based on observed data there are created similarity matrices between users and/or between elements. To present elements to users that are either connected to their acquaintances or to similar elements the user chose.
3. **Content based** personalization defines similarities based on meta data between content elements. Therefore the content have to be indexed to suggest other elements fitting the users preferences. The meta data is created manually or automatically, e.g. by tagging or by search engines respectively.

Customization (explicit personalization)

Customization in information technology is the adjustment of hardware, software or information to changed conditions.

Rule based personalization is mostly based on user entered data and the rule base. So this is a customization technique, because the users affects knowingly how the information is presented to him.

Personalization versus Customization

As described above there is an obvious difference between customization and personalization. Two ways lead to the users preferences and these ways are combinable. First there is explicit data entered by the user, called explicit personalization or customization, and second to observe the users actions, so called implicit personalization or just personalization. Personalization most time is unknown to the user and commonly used for both terms. So in most cases people do not differ between. Customization is configured by the user and personalization is done by the system mostly without the user knowing it. He can only guess.

2.4.2. Search Personalization and Customization

This paragraph is about how search engines can be personalized or customized. Some examples will be given.

Search personalization and customization in this context does not mean to adjust the engine itself. It only affects the search context and settings that affects the result, as well as ranking the result in an individual way.

Personalized Search

Personalized search adjusts the search with search habits of users by analyzing their behavior, search history and based on collected data about groups, meta data and other such information e.g. location, system or browser.

In other words it is prediction of what users want to search for based on heuristics. So it is above ranking of results. Maybe the most prominent example for personalized search is the widely used search engine Google.

As described in “The Future of Google’s Search Personalization” [Jac08], a company can not any longer rely on checking its search rankings, because there are no rankings. Google personalizes users search, based on the search history as well as the users location. So the results can differ between different users, using the same search query and this happens even if they are not logged in to their Google account.

Customized Search

In contrast to personalized search, customized search is adjusted by users, either per request or through persistent account data. The search in this connection does not adapt search habits itself, so users have to be proactive and set the data them self.

As mentioned in the former paragraph Google personalizes its search, but they also let the users customize it by using the *advanced search* options which may differ the combination of results, language, file type searched domain or how recent the page is.

Conclusion on Search

So in result of the example, current non-specialized public search engine combine customized and personalized search.

Whether personalized search or customized search is better differs by search engine purpose and its users. Common public search engines may better use both techniques, because there is too much data available, so users can not configure all on their own and users are hardly to influent. If it is a specialized search engine and there is a known type of users, using this engine, they may have a deeper knowing of the information based. In this case it might be a lot easier for them to find relevant results by customizing instead of personalizing the search results.

3 Analysis

The constraints of this work are given by reimplementing Hayoo using Hawk, Holumbus and a wrapper for Holumbus to use it in web applications generated with Hawk. All code needs to be written in the pure functional language Haskell, as Hayoo, Holumbus and Hawk are.

This chapter is split into two parts. Section 3.1 analyzes Hayoo and its changes resulting from the task of this thesis e.g. customization and wrapping the Holumbus specific components. The analysis of Hayoo results in some needed changes of the Hawk framework which are analyzed in section 3.2.

3.1. Hayoo! API Search

Hayoo is an API search engine for Haskell, which uses a search index generated from hackage [hackage]. The existing implementation written by Timo B. Hübel and Sebastian M. Schlatt will be analyzed in 3.1.1. As most of the parts of the current implementation are specific to Holumbus they will be picked up in the encapsulation of Holumbus in 3.1.2. Therefore an analysis is done for what parts of Hayoo can be reused in the new implementation.

The reimplementation of Hayoo will be divided into two main parts to create a maximum in reusability. One part will enclose the Holumbus framework so programmers may easily adopt it for their own search engine web application. It is analyzed in 3.1.2. In paragraph 3.1.3 the above mentioned analysis for customized search is done.

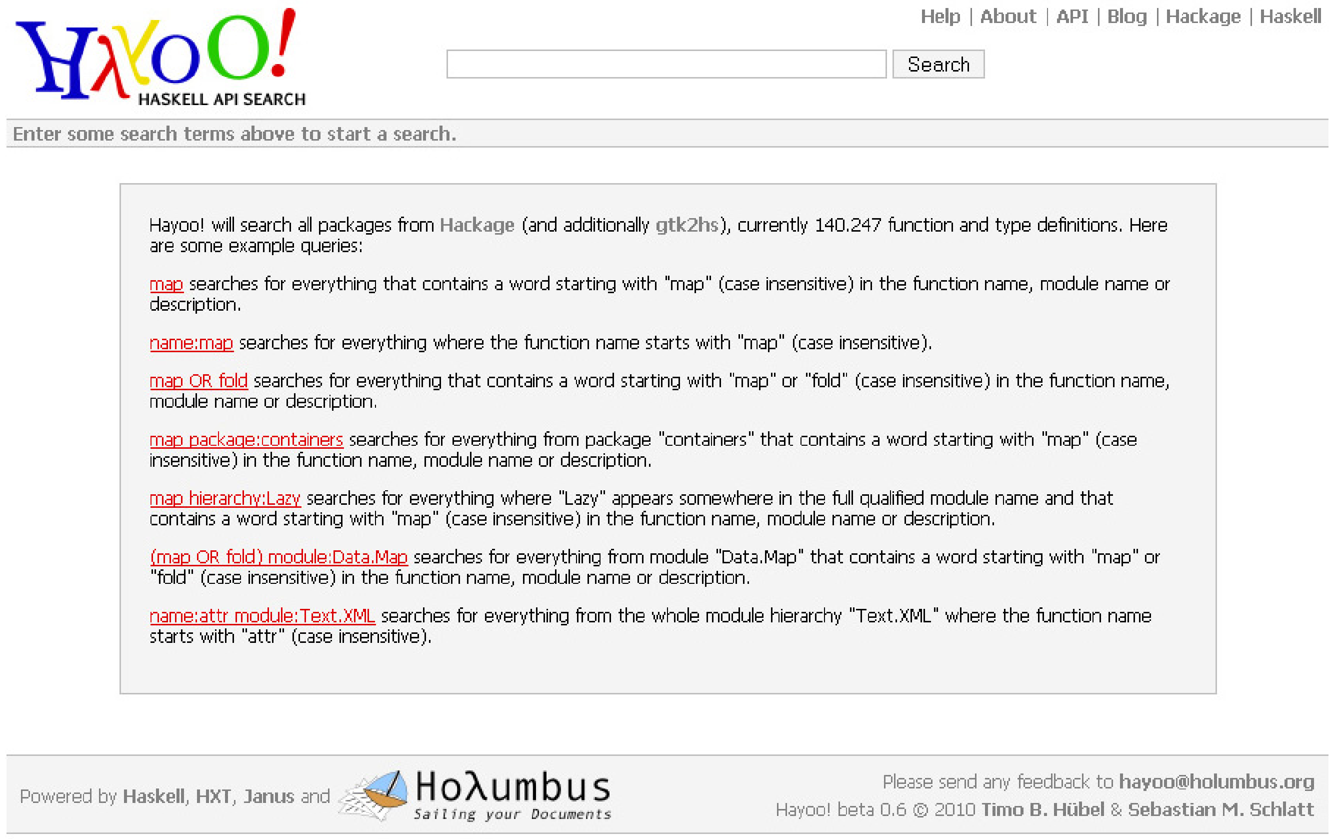
The other part is the architecture for web applications using Hawk. This topic is described in detail in chapter three of *The Hawk Framework* [PR09, p. 23 ff.]. The recommended folder structure is described in appendix A.1.

3.1.1. Existing Implementation

A brief description of Hayoo can be found in *The Holumbus Framework* [Hüb08, p. 55-59]. As mentioned there Hayoo is based on *Hoogle API Search* [Mit04], which is another web search engine for the Haskell API. The current Hayoo web interface is shown in figure 3.1. The indexer part of Hayoo is excluded because this work is not affected by it and does not offer another way to use or to change how indexers can be written.

Hayoo! API Search written by Timo B. Hübel and Sebastian M. Schlatt can be departed in nine different fractions:

1. A Web Server used for providing HTTP access,
2. Handling of requests and search queries,
3. Used data types,
4. Used application data, e.g. search index,
5. Parsing a search query string,
6. Ranking of a search result,
7. Methods for formatting HTML output and
8. formatting Ajax output, as well as
9. the Web Service API to include the Hayoo search functionality in other web applications.


HASKELL API SEARCH
Help | About | API | Blog | Hackage | Haskell
Search
Enter some search terms above to start a search.

Hayoo! will search all packages from Hackage (and additionally gtk2hs), currently 140.247 function and type definitions. Here are some example queries:

map searches for everything that contains a word starting with "map" (case insensitive) in the function name, module name or description.

name:map searches for everything where the function name starts with "map" (case insensitive).

map OR fold searches for everything that contains a word starting with "map" or "fold" (case insensitive) in the function name, module name or description.

map package:containers searches for everything from package "containers" that contains a word starting with "map" (case insensitive) in the function name, module name or description.

map hierarchy:Lazy searches for everything where "Lazy" appears somewhere in the full qualified module name and that contains a word starting with "map" (case insensitive) in the function name, module name or description.

(map OR fold) module:Data.Map searches for everything from module "Data.Map" that contains a word starting with "map" or "fold" (case insensitive) in the function name, module name or description.

name:attr module:Text.XML searches for everything from the whole module hierarchy "Text.XML" where the function name starts with "attr" (case insensitive).

Powered by Haskell, HXT, Janus and Hoλumbus Sailing your Documents

Please send any feedback to hayoo@holumbus.org
Hayoo! beta 0.6 © 2010 Timo B. Hübel & Sebastian M. Schlatt

Figure 3.1.: Current web interface of the *Hayoo! API Search*

Parts of the current Implementation

In the current implementation the chosen web server was Janus [Uhl06a, Uhl06b]. It is handles HTTP requests and passes them to the request function for handling search queries, static page requests, calling the parser and ranking functions as well as formatting the output. The data types only describe one type for passing around the data through the components of Hayoo, a type for query response informations and a custom data type for the Hayoo document information. Hayoo configures Janus to load search index, documents data and cache database at startup. This happens without reloading them on every search request.

The Parser, for incoming query strings to parse, is Hayoo specific. It is needed to normalize the function signature to the index data and for fuzzy search. The Holumbus standard function `parseQuery` lacks these features. Ranking is hard coded in the current implementation and ranks documents by counting them. The word context for searched terms is also rated e.g. a result with the word found in the function name will be higher rated as if the word was found in the description. Exact matches are higher rated than non-exact and results from `Prelude` or `Base` packages are higher rated too.

Hayoo used Pickler's for formatting HTML output. Pickler's are a part of the Haskell XML Toolbox (HXT) [Sch]. The client side processing supports Ajax functionality as well as normal requests for users that do not have JavaScript enabled. Hayoo also offers a Web Service API for handling search requests and sending results as JSON encoded string.

Changes and Adopts to the new Implementation

The web interface of the current implementation will be adopted, with only minor changes resulting from the extra functionality for customized search. This will be a link to a customization form in the page header, a login form and a link to register a new account or to change the users search settings.

The use of Janus will be dropped for the new implementation, because Hawk uses the Hack interface as web server [Wan09]. Therefore data types for passing search data through the application or search responses will change. Only the data type for custom document information of Hayoo will not change, because it is specific to Hayoo's main task, to search *hackage*.

Single load of index, document and cache database files at system start up will be adopted, as loading them on every request would result in a really slow live system. This is one point to extend the Hawk framework. It is analyzed in detail in paragraph 3.2.1.

The parser will be adopted with no changes, because it works for the existing indexer that will not be changed in any way. It only will be integrated to the Holumbus wrapper. Ranking becomes a more relevant part in the new version of Hayoo, because the users will be able to configure the ranking of result. The users addressed by Hayoo are a well known group with some basic knowledge of how search engines work and about the libraries searched. So they may know best how they want the search to be rated to get relevant results. Though there might be little changes to made but this is discussed in subsection 3.1.3.

No Pickler's will be used in the new implementation, as Hawk offers functionality for formatting HTML with templates or Haskell functions using another part of HXT. Additionally the Hawk functions or template system is less confusing for Haskell beginners, so this may help them to start developing web applications with Hawk. Ajax is supported by Hayoo, so it will stay in the new version too. Hawk needs to be extended by supporting JSON to offer Ajax functionality on client side as well as offering the Web Service API. Analysis for JSON support is done in paragraph 3.2.2. The normal HTML view will also be available, so that users with JavaScript disabled can also use Hayoo.

3.1.2. Holumbus Wrapper

As depicted in the introduction of this chapter Hayoo will be split into two main parts which are nearly independent, for reuse reason. In the following the analysis for encapsulation or wrapping of the Holumbus search engine framework is done. The other part is the MVC web application.

A detailed description of how to reuse the wrapper implementation, in section 5.1, is given in chapter 6.

All specific Holumbus types, data and functions have to be encapsulated to uncouple them from the Hayoo web application. This is done using the adapter or wrapper design pattern. It "Convert the interface of a class into another interface clients expect." [GHJV94, p. 139 ff.].

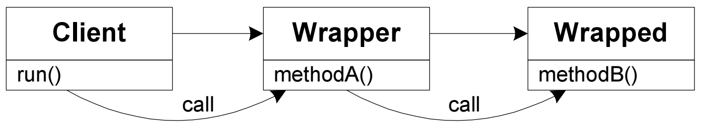


Figure 3.2.: Scheme of the adapter design pattern

The adapter or wrapper design pattern hides interface specific data of an implementation to a client. In figure 3.2 the *Wrapper* hides `methodB()` from the *Client*. A client only needs to instantiate and use the adapter without worrying about implementation specific details of the wrapped interface.

Splitting all different areas of using Holumbus, like querying, formating results, ranking, parsing and holding data does not lead to an application which is easy to change, adapt or reuse. So the mentioned design pattern is used to encapsulate all data in one module or behind one interface and the web application will act as a client to it. Then only a call to one interface for search configuration, querying, parsing, loading data, formatting or printing the results is needed. In that way the whole search engine can be replaced with no or only minor changes to the web application. Or the other way around, replacement of the web application or web interface results in nearly no changes in the search engine.

Another relevant part of the wrapper is to give a reasonable break down to modules e.g. query handling, configuration generation, parsing of queries, printing functions for the results, etc. This wrapper offers a single interface for clients and so merges the special modules in its interface.

The interface only needs four parts.

1. A query function with configuration parameters,
2. functions for generating the search query configuration,
3. functions for formatting the search result and
4. a public type for passing around the search result in an application.

The public type has to be returned by the query function, containing an error string or a search result and the corresponding search configuration. This type is needed because normally a search request is done from a controller which needs to pass the result to a view. The query function will need a parser, a ranking function, the index, document and cache data, the search string and maybe some other custom configuration information. So modules for data types, parser, ranking, creating query configuration and printing are needed in addition to the interface.

The functionality of a search engine web application for example may be extended by translating results, so the wrapper will be easily reusable or changeable. Additionally the processing of search results need to be in an exchangeable format, like plain text, XHTML or JSON.

3.1.3. Personalization or Customization

As depicted in paragraph 2.4.2 there are two major ways of personalization for search engines: Search personalization and search customization. To analyze how users can access Hayoo in an optimal way, the following questions need to be answered.

What kind of Information does Hayoo access?

The Hayoo search index is based on highly specialized information. Only the hackage page is searched as source. The structure of the information is fixed, as well as possible query parameters and contexts. The information base is also limited to some Megabyte of information in text or indexed binary format.

So it is a highly structured, highly specialized search engine. Which leads to nothing, because the second component is unknown: the users.

Who are the Users of Hayoo?

This is a search engine for programmers, so they know at least a little about how search engines or the techniques behind them work. They also know how the information base is structured and they know what information they want to access. So it can be assumed that the user is known to the material and this leads to the following decision.

Decision

Highly structured and highly specialized search engines do not lead to prefer search customization nor to search personalization, but knowledge of users in this context does.

If there is highly unstructured and no special information, but users known to the material, they may prefer a personalized search instead of a customized one. In this case with highly structured and specialized information as well as knowingly users, a customized search approach will be preferred. Personalization in this special context may only annoy or confuse users, because they maybe do not want to search based on their recent queries if they just started a new Haskell project.

Customization data

The users are known to the information based and the search engine is highly specialized, so they get full customization access. It can be assumed that users of Hayoo know exactly what they have to ask the search engine, so it is maybe the best way to offer them the whole range of configuration options for queries, that Holumbus and the Hayoo index support. This includes configuration of the ranking parameters, to get more relevant results on top.

3.2. Hawk

As described in the former section the Hawk framework needs to be extended. In its current version it lacks an application wide storage, so the programmer has to implement it on its own or load that data on every request. Also there is no support for JSON to implement Ajax and to lower bandwidth usage as well as authentication, which nearly all web applications with user accounts need.

Paragraph 3.2.1 of this section analyses how an application wide storage structure can be integrated to Hawk. The subsequent paragraph 3.2.2 discusses different ways to extend Hawk by JSON support. The paragraphs 3.2.3 and 3.2.4 analyze how to implement an authentication system with different authentication methods and how to apply them best to single actions.

3.2.1. Application-Wide Storage

Application-wide storage means that a data type defined in the web application is loaded into the Hawk framework due to startup and accessible by the application on every client request. It is like the Hawk data types `AppEnvironment` and `BasicConfiguration` with the little difference, that only the web application knows the type, but Hawk needs to provide it to the application in a request. This may work easily in a dynamically typed language, but in Haskell it is a little more complex. In addition the type for application-wide storage has to be monadic, because in most cases programmers will store file handles or data in this type e.g. a search index or a DB handle.

The naive approach is to use a type parameter which contains the application data type. This does not make any sense, because it would need to be encapsulated in the `StateController` type which is used in nearly every Hawk module, as well as all applications use it massively. So type parameters are not applicable in this case.

Five different approaches for an application-wide storage are were analyzed, whereas the first three of them were dropped, because they all need type parameters. *Type Families*, *Template Haskell* and *MVar or IORef* are those three approaches [Cha, SPJ02]. The two remaining approaches are *Extensible Records* and *Rank-N Types* [JPJ99, PJVWS07].

Extensible Records

Extensible Records is a language extension to the Haskell programming language [KLS04, PJ03], like *Type Families* and *Template Haskell.* These extensions are available in the Glasgow Haskell Compiler (GHC) [GHC].

With this approach a programmer can extend a data type in the web application, which is defined in the Hawk framework. So the data type is known while compilation of Hawk and can be extended by web application specific data.

A short example from *Strongly Typed Heterogeneous Collections* [KLS04].

```
-- a namespace
data FootNMouth = FootNMouth

-- constructing labels of the record
key = firstLabel FootNMouth "key"
```

```
name = nextLabel key "name"
breed = nextLabel name "breed"
price = nextLabel breed "price"

-- build an example record
unpricedAngus = key .=. (42::Integer)
            .*. name .=. "Angus"
            .*. breed .=. Cow
            .*. emptyRecord
```

The data type is declared first. After that the record labels are created by declaring functions with the constructor functions `firstLabel` and `nextLabel`. `firstLabel` expects a data type with a single constructor and a string as parameter. The string will be the name of the first record element. `nextLabel` expects a function that returns a `FootNMouth` data type and a string. The first input parameter of `nextLabel` always is the function for creating the previous record element. `unpricedAngus` is a function that creates an example `FootNMouth` data type by assigning values to the record elements (`.=.`). The elements are created by functions and are concatenated by the `.*.` operator.

Rank-N Types

Rank-N Types is a language extension, too. The proposal of *Rank-N Types* is to allow explicit universal quantification, so types can have the forms,

```
type -> type
```

```
forall vars. [context =>] type
```

```
monotype
```

where `monotype` is a normal Haskell '98 type [PJ03].

The main attraction on this is that it can hide a type parameter from a data type. The following two code fragments spot the difference between parameterized data types and *Rank-N Types*.

1. Parameterized code

   ```
   data Foo c = Foo {f :: c}

   createF :: c -> Foo c
   createF e = Foo e
   ```

```
showF :: Foo Int -> Int
showF e = f e
```

2. Code with *Rank-N Types* and classes

```
data Bar = Bar {b :: (Class c) => c}

class Class c where get :: c

createB :: (forall c. (Class c) => c) -> Bar
createB e = Bar e

instance Class Int where get = 5

showB :: Bar -> Int
showB e = b e
```

From this code it might be easier to use parameterized data types. On the inputs "'`> showF $ createF 5`"' and "'`> showB $ createB get`"' both return the same output. The parameterized version is tinier in code. This was expected, because it is the more common way to use a data type. The attraction of *Rank-N Types* lies in the signatures of functions and where the data types is used in other functions and types.

For example an application wide state is created with `Foo` as a part of it and every function in that application operates on that state, the parameter type needs to be known in every function, even if only two functions in that application *uses* `Foo`. Additionally the application becomes hard to reuse and to maintain, because the whole application needs to change if the type is changed.

Choice

Some approaches were discussed. A combination of *Rank-N Types* and classes was chosen as the best way to supply an application specific data type to users of the Hawk framework. As the analysis shows only *Extensible Records* and *Rank-N Types* can bring this functionality to Hawk, because they avoid changing the whole framework by using a type parameter. With *Extensible Records* programmers do not have a normal Haskell data type in their application, because the data type will be defined with an empty constructor in Hawk and will be extended in the web application. The programmers instead should use their application data type like a normal data type, they have defined. Another pro of *Rank-N Types* is that they are minimal invasive, so there will be only three changes and one addition to the framework. That is only five new lines of code. So this approach is easy to appreciate and clear.

Precise of why *Rank-N Types* were chosen to implement to application wide storage data type: The approach is minimal invasive, nice to extend and easy to use.

3.2.2. Ajax and JSON

To support Ajax on server side JSON is the most efficient way, because it easily converts to JavaScript objects. Although JSON is not needed for supporting Ajax, because Ajax also can be used with XML. There are four approaches to integrate JSON to Hawk. They are depicted in the following.

AjaxController

An `AjaxController` instead of a `StateController` with another response type can not be used. This will not work as depicted in paragraph 3.2.4, because this will exponential increase the number of controllers, the whole framework and its applications will change, it lacks extensibility and the *hack* interface expects a `ByteString` as response anyway.

Extended Routing

Another way to integrate JSON to Hawk is to extend the routing function. For example if a request URL looks like `http://host/ajax/controller/action` the framework will generate a JSON response, whereas it will generate a normal response on calling `http://host/controller/action`. Disadvantage of this approach is that it only adds an abstraction which make the framework less flexible without solving the problem that a JSON encoded string needs to be generated.

JsonView

Hawk in its current implementation uses three different types of views, a text based, one that uses templates and an empty one. Another view can be added to generate JSON from an application internal data representation. The problem is that the programmer will need to create a view for each output format. For example JSON needs another formatting than an XHTML response, because JSON is translated to the XHTML representation on client side.

Choice

It seemed to be the easiest way to just extend the view part of Hawk by another view type, that generates JSON encoded strings. So the *JsonView* approach was chosen for implementation, because of the above mentioned problems the other two approaches have.

3.2.3. Authentication

Authentication is a main task in most web applications. In this paragraph the analysis for an authentication interface and some basics is done.

Authentication on every request is not useful, so a storage is needed. This storage will be the session data and therefore every authentication application will need to use session handling. There are three basic actions to do on the session storage, which are setting, getting and deleting data or in other words login, check if the user is authenticated and logout. For the later implementation of this hawk extension not all of the authentication methods mentioned in the fundamentals will be implemented.

Authentication will need to be applicable per action. A connection between authentication function and action will need to be given. How the interface will be integrated into Hawk and how to apply it in a web application is discussed in the next paragraph 3.2.4.

3.2.4. Combinator

This paragraph discusses how to applicate a special functionality to a set of actions, e.g. authentication or JSON response. There are three different ways to reach this goal.

1. The above mentioned extension of the controller data type, e.g. there will be a `StateController`, `AjaxController`, `AuthController`, `AjaxAuthController`, and so on. As depicted the number of controllers will explode and nearly the whole Hawk framework will need to change to implement this approach.

2. The second way is to change the routing data type.

   ```
   type Routing =
     (String
     ,(StateController ByteString
      ,StateController ByteString -> StateController
         ByteString))
   ```

   In the inner tuple the function of the second argument will be applied to the first element. This approach is not as worse as the first one, but it lacks extensibility. If another change needs to be made, for example methods to run after the action is proceeded, all applications and some structures in Hawk need to change.

3. The last approach is a combinator function which is applicable at a list of routing tuples, so it can be applied to a set of routes with one function call in the route generating functions of a controller. This do not need any changes in current code of Hawk and is easily extensible. It is also applicable as a post combine function, that proceeds on `ByteString` data after the view created it.

4 Extending Hawk

To guarantee an easy implementation and full functionality for the example web application Hayoo, this chapter describes changes and extension to the Hawk framework. Hawk is based on the MVC architectural pattern. MVC is a software architecture to isolate application logic from input and presentation. It is split to the three parts of data model, presentation and controller. This flexible program design eases later changes, extensions and reuse of each component as well as independent development and testing.

As depicted in figure 4.1 controllers typically handle requests and coordinate the application flow. All not model based calculations and querying of model data will be done in a controller. The results will be passed to the presentation component to format the data in an interchangeable format. So that for example the client side can display it. The presentation component can also query the model for some missing data, e.g. translations. The model component is an interface to the data model which normally is represented as a database. It also support some basic functions for doing model based calculations.

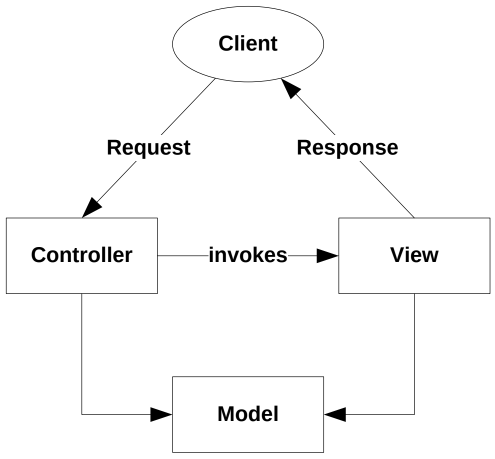


Figure 4.1.: Model-View-Controller pattern

Hawk uses the MVC pattern and is structured to the above mentioned three components. This chapter also is structured in the same way. Section 4.1 shows how the extensions to the controller component of Hawk framework were implemented. The subsequent section 4.2 will show new implementations to the view component. There were no changes in the model, because all model functionality needed for implementing Hayoo was available.

4.1. Controller

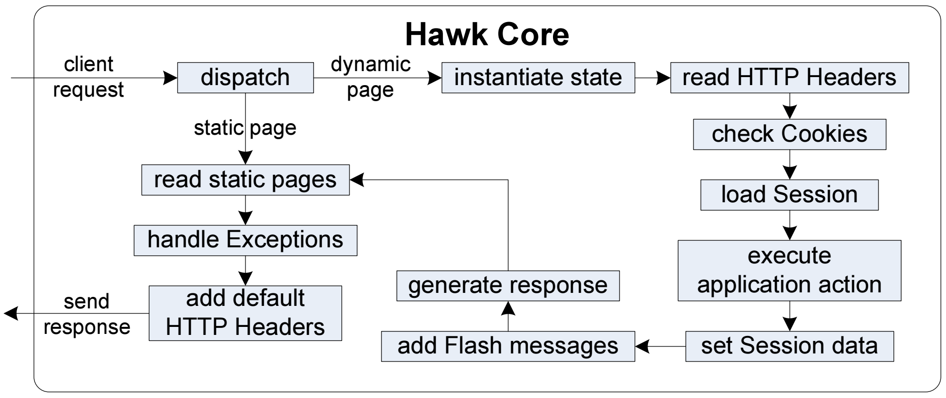


Figure 4.2.: The *Hawk* controller workflow

Figure 4.2 shows how a request is handled in Hawk. The controller core dispatches the request. This will result in either loading a static page or generating the web page dynamically. First the state provided to the application is instantiated and filled with HTTP header data and checking for Cookies. After that the application action invoked by the request is executed, encapsulated by loading and updating session data. Subsequently flash messages are added to the response and the dynamic response is generated. If any error occurred, e.g. the action does not exists, a static page will be loaded and exceptions that may occurred on database queries are caught. Finally the HTTP default headers are added to the response and it is send back to the client.

This section contains extensions made to the controller. Paragraph 4.1.1 describes how the implementation of an application wide storage type was implemented. That way the web application can define its own type and Hawk makes it available to the application on every request, without recreating it. In 4.1.2 the implementation of authentication functionality and its interface is explained. In its subsequent paragraph 4.1.3 the implementation and usage of the combinator for applying the authentication method per action is shown.

4.1.1. Application Storage Type

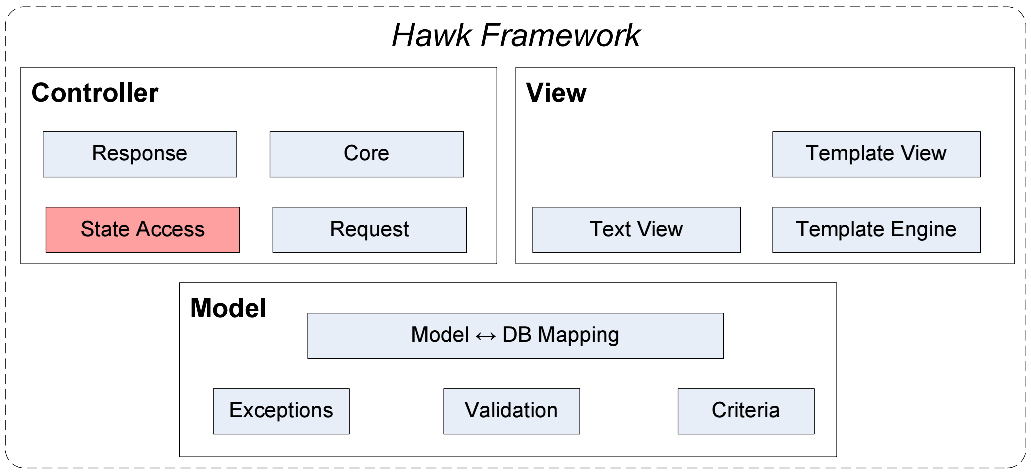


Figure 4.3.: Application storage type in Hawk

The application storage data type extends the *State Access* part of the Hawk framework by a new type of state that is not known to the framework in detail.

The data type for saving and making application specific data available through the life time of a web application uses a Haskell extension called *Rank-N Types* and normal Haskell classes [PJVWS07].

As mentioned ranked types can not provide this functionality on their own. An abstract class is needed to hide the type parameter from the Hawk `RequestEnv` type in combination of the ranked parameter `forall` in some function signatures of Hawk.

In the following all the changes made to Hawk are depicted in detail. First the `RequestEnv` data type was changed, because it holds all valuable data the web application can access.

```
data RequestEnv = RequestEnv
  { databaseConnection :: ConnWrapper
  , configuration      :: BasicConfiguration
  , request            :: Hack.Env
  , environmentOptions :: Options
  , appConfiguration   ::
       (AppConfiguration a, MonadIO m) => m a
  }
```

`MonadIO` i a class for monadic encapsulation of the outside world via IO. It is needed to save for example DB or file handles in the data type without using an unsafe IO operation.

The abstract `AppConfiguration` class only supports one method to get the data type of a concrete class implementation out of it. So the abstract class is really simple.

```
class AppConfiguration a where
  getInstance :: MonadIO m => m a
```

Therefore the `RequestEnv` parameter have to be passed into the framework until the data type is constructed. So all function signatures until the construction have to be changed. Luckily there are only two, the application initializer and the server module. The signatures have to be extended by the following term:

```
(forall a m. (AppConfiguration a, MonadIO m) => m a)
```

So the `requestHandler` function in the `Server.hs` module changes to:

```
requestHandler ::
     (forall a m. (AppConfiguration a, MonadIO m) => m a)
  -> ConnWrapper
  -> BasicConfiguration
  -> Options
  -> Application
requestHandler app conn conf opts env =
   runReaderT
     (runController dispatch)
     (RequestEnv conn conf env opts app)
```

The interesting thing for programmers is what they have to do in the code to use their own web application type, accessible from everywhere in the application. The changes are limited to two locations, the main module and the configuration module. The main module additionally has to call the `getInstance` function and the configuration module need to define the `AppConfiguration` concrete class.

The `getApplication` function from the `Initializer.hs` module now needs three parameters, with the first as the application type.

```
main = run 80
  $ getApplication getInstance environment configuration
```

The following two examples show how to create a simple data type and the corresponding concrete class

```
data MyAppData = MyAppData
  { text :: String }

instance AppConfiguration MyAppData where
  getInstance = return $ MyAppData "some text"
```

respectively a minimal implementation for an `AppConfiguration` concrete class, which is implemented in Hawk's `Initializer` module.

```
instance AppConfiguration () where
  getInstance = return ()
```

So `getInstance` just returns a data type of class `AppConfiguration`, wrapped in a monad.

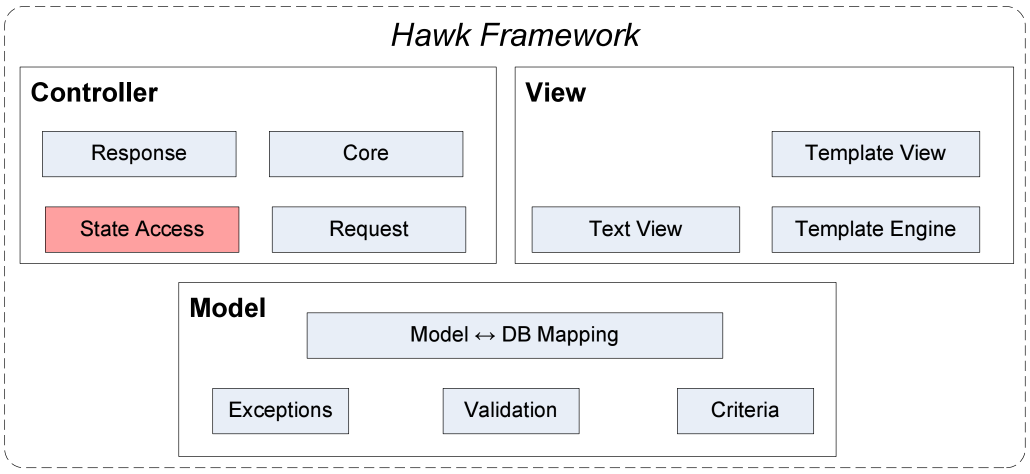


Figure 4.4.: Authentication in Hawk

4.1.2. Authentication

Authentication also affects the *State Access*, because it is staged on the session handling of Hawk as depicted in figure 4.4.

Authentication is a common task in nowadays web applications and nearly every web page created with a web framework supports authentication. Even though no user login exists, there is access for administration and maintenance.

From the different authentication methods discussed previously in 2.3.2 for the present only two are implemented to the Hawk framework. This is to show that the interface is applicable and the methods support all needed tasks. The two authentication methods implemented are *Form-Based* and *HTTP Basic* authentication. The form based implementation will query a database and for demonstration purpose the HTTP basic implementation will differ here and queries a *comma separated values* (CVS) file. Whereas HTTP basic and digest authentication in most cases is only applied in intranets or when only a small number of users has access to this area, e.g. an ACP.

Interface

For the different authentication methods one common interface is needed to independently change the different methods without changing code. Only a minor change in configuration will be needed to replace the concrete method.

As depicted in the analysis authentication only needs a storage, what will be the session storage, and three basic methods. Therefore session handling is assumed for web applications that use authentication. The three methods are login, logout and checking of login data. Additionally a return type for the login function is helpful for web application developer, to offer detailed error messages to users. This data type contains five constructors, which differentiate between success, not existent user ID, wrong password, multiple fitting user ID's and one constructor for custom error, e.g. SQL parsing errors or DB connection errors.

```
data AuthResult = AuthSuccess
                | AuthFailureUnknown String
                | AuthFailureIdNotFound
                | AuthFailureAmbiguousId
                | AuthFailureInvalidCredential
```

The following three basic methods are public, but it is recommended not to use them in web applications. They are not that descriptive as the later described methods for public use. Only if a web application uses a special form of authentication these methods can be used to set, get and delete session data.

```
getSessionAuth :: (HasState m) => m (Maybe String)
getSessionAuth = getSessionValue getAuthKey

setSessionAuth :: (HasState m) => String -> m ()
setSessionAuth = setSessionValue getAuthKey

delSessionAuth :: (HasState m) => m ()
delSessionAuth = deleteSessionKey getAuthKey

getAuthKey :: String
getAuthKey = "user_auth"
```

The login method normally checks if a user is logged in. If not it will try to verify the login data against an information base and set session data. This is not done in the interface. The interface methods will only read the authentication method from `BasicConfiguration` and run it (see section 5.2). So in this case `authType` can either be `dbAuth` or `httpAuth`. `tryLogin` is only a synonym for the `auth` method.

```
auth :: AuthType
auth = asks configuration >>= authType

tryLogin :: AuthType
tryLogin = auth
```

The `auth` method do not need any parameters, but it reads data, so a state is needed to transfer it. `AuthType` is a monadic type, that is wrapped into a DB, IO and state monad.

```
type AuthType =
   (MonadDB m, MonadIO m, HasState m) => m AuthResult
```

The access of authentication data is handled through the following methods. Developers have the option to get a boolean result whether a user is authenticated or they can try to access the users name. The user name result will be `Nothing` if the user is not authenticated.

```
isAuthedAs :: (HasState m) => m (Maybe String)
isAuthedAs = getSessionAuth

isAuthed :: (HasState m) => m Bool
isAuthed = isJust `liftM` getSessionAuth
```

The `logout` function is only a synonym for `delSessionAuth`, so the authentication key just will be deleted from session data.

```
logout :: (HasState m) => m ()
logout = delSessionAuth
```

Form-Based Authentication

The `dbAuth` authentication method of corresponding module handles form based authentication and checks against a database.

```
dbAuth :: AuthType
dbAuth = do
  sess <- isAuthed
  case sess of
    False -> do
      (t, idCol, credCol, crypto, user, pass) <- getOpts
      u <- getParam user
      p <- getParam pass
      rows <- querySelect
          $ setProjection [colP idCol, colP credCol]
          $ setTables [t]
          $ setCriteria
             (restrictionCriteria ((val u) .==. (col idCol)))
          $ newSelect
      case rows of
        (x:[])    ->
          case lookupAuthDbRes (encrypt crypto p) x of
            Nothing ->
              return AuthFailureInvalidCredential
            Just _   ->
              setSessionAuth u >>
              return AuthSuccess
        (_:_:_) ->
              return AuthFailureAmbiguousId
        _         ->
              return AuthFailureIdNotFound
    True ->   return AuthSuccess
```

The method first checks if the user is authenticated. When he is then there will be no further processing. If he is not authenticated the configuration options are read by the `getOpts` function and the `getParam` function will return user ID and password from the request parameters. After this a SQL query on the user ID is executed. If the number of rows found is not one, an authentication failure result is generated. If it is one, the typed password is encrypted with the configured encryption method and compared with the password row from the database. When the password is correct, the users ID will be added to the session data and an authentication success result is generated, otherwise an invalid credential result will be returned.

HTTP Basic Authentication

The `HttpAuth` module implements the HTTP basic authentication method. It is structured similar to the `dbAuth` function. It just do not uses a DB connection. Instead a CSV file is used for data comparison.

```
httpAuth :: AuthType
httpAuth = do
  sess <- isAuthed
  if sess then return AuthSuccess
  else do
    a <- getRequestHeader "Authorization"
    case a of
      Nothing -> return AuthFailureIdNotFound
      Just v ->
        case decode (snd (splitWhere (== ' ') v)) of
          Nothing -> return AuthFailureIdNotFound
          Just v' -> uncurry httpAuth' $
                     splitWhere (== ':') (UTF8.decode v')
  where
```

```
httpAuth' u p = do
  path <- getOpts
  fc <- liftIO (readFile path)
  let t = splitAll (== ',') fc
      l = map (splitWhere (== ':')) t
  case (lookup u l) of
    Nothing -> return AuthFailureIdNotFound
    Just v -> if v == p
      then setSessionAuth u >> return AuthSuccess
      else return AuthFailureInvalidCredential
```

Like `dbAuth` it is checked first if the user is already logged in. Then the function reads the HTTP request header value of the key `Authorization` and splits it. The `decode` function decodes a Base64 encoded string to its plain text representation. This representation is parsed and compared with the file content. Whereas the file contains key value pairs of user ID and password. This method only looks up the user ID, entered to the authentication field from the standard dialog, in the CSV file and compares the password. So this function do not support password encryption. All passwords are in plain text. On success the user ID is saved to session data too and an `AuthSuccess` result is generated.

4.1.3. Action Combinator

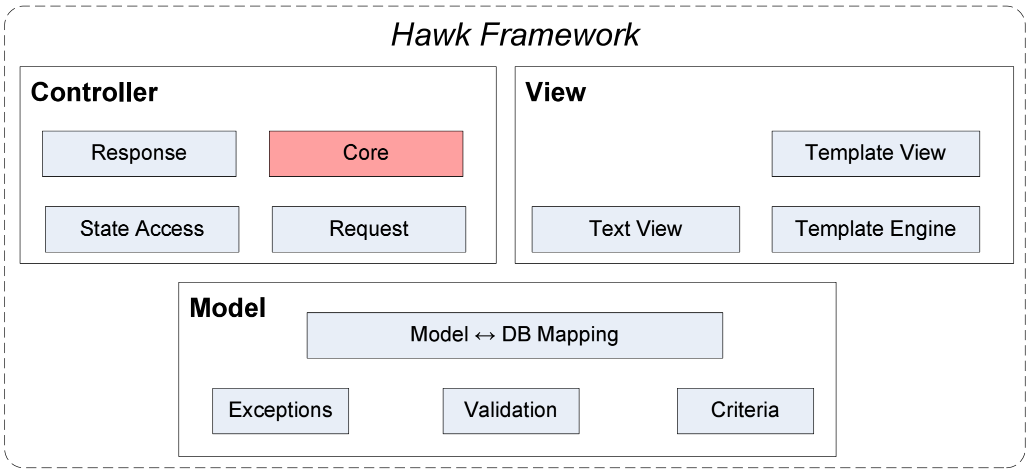


Figure 4.5.: Action combinator in Hawk

The action combinator extends the Hawk framework core, because it provides a method for combining functions with the actions that are invoked from the core.

In the analysis was depicted that in greater web applications it is hard to survey what actions for example are allowed to achieved by known or unknown users. This is especially hard if the check for it is somewhere in the code and not bound to the routing information. This combinator should avoid this problem and will offer a possibility to couple a function to a single or a group of actions. Therefore the combinator has to integrate itself into the existing routing type structure.

```
type Routing = (String, Controller)
type Controller = StateController ByteString
```

In precise a function is needed to manipulate the `Controller`. That leads to a function that takes a function on controllers and a routing list and applies the function to each controller in the list. The function passed to the combinator function can do some pre or post processing or both of them. It can manipulate request or response data, so that the controller action will act in a completely different way.

```
combine :: (Controller -> Controller) -> [Routing] -> [Routing]
combine _ []     = []
combine f (x:xs) = fx : combine f xs
        where fx = (fst x, f $ snd x)
```

Until now a combinator function is only implemented for authentication with two to three additional parameters.

```
authF :: String
      -> String
      -> StateController a
      -> StateController a
authF = authF' ""

authF' :: String
       -> String
       -> String
       -> StateController a
       -> StateController a
authF' e c a contr = do
  b <- isAuthed
  if b
    then contr
    else
      case e of
        "" -> redirectToAction c a
        _ -> do
          setFlash "error" e
          redirectToAction c a
```

The parameters are an optional error string, a controller and an action name, as well as the controller passed by the `combine` function. The `authF'` function checks whether users are authenticated or not. If they are the normal action is run, else they are redirected to the given controller and action,displaying the optional error message. This method is not able to authenticate users, because the user does not send his authentication data on every request.

That is what an example routing list looks like, with a public action and a restricted action. If users try to access the restricted action without being logged in, they will be redirected to the index action and the error message *"You are not logged in."* is displayed.

```
routes :: [Routing]
routes =
  [ ("index",
     indexAction >>= render (typedView "index" indexXhtml))
  ] ++
   combine (authF' "You are not logged in." "index" "index")
  [ ("restricted",
     restrictAction >>= render (typedView "restrict"
         restrictXhtml))
  ]
```

Another example of a routing list can be found in Hayoo's `UserController` in 5.3.2.

4.2. View

This section describes extensions to the presentation component of the Hawk framework. In this case there is only one extension. The implementation of JSON encoded string support and hence an interface for communication through JavaScript objects is shown. This mainly allows Hawk to generate fast interactive web applications with Ajax functionality.

JSON View is a new view component that complements the existing components by JSON support.

In addition to the views for plain text and templates, Hawk was extended by a JSON view. The `JsonView` module supports converting of different data types to an internal JSON representation, which encodes to a JSON string for transmission. Listing 4.1 shows a shortened version of the `JsonView` module.

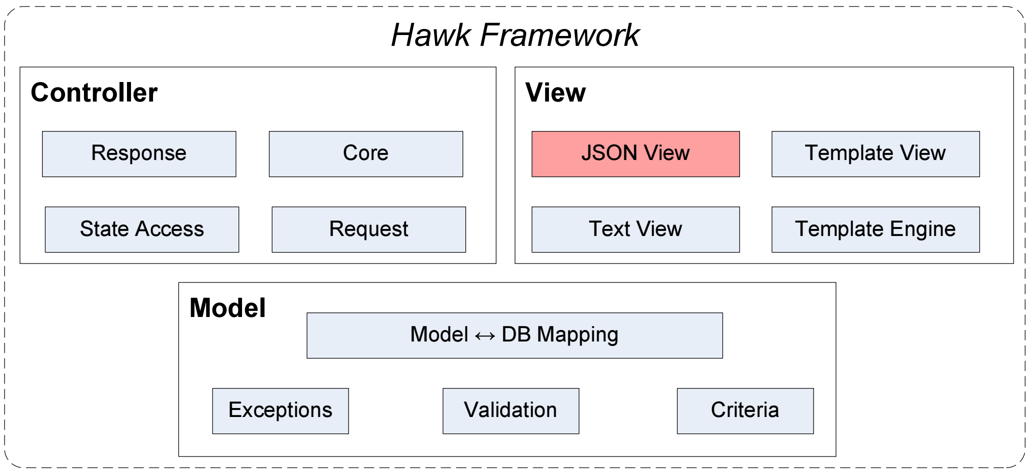


Figure 4.6.: JsonView in Hawk

```
{-# LANGUAGE TypeFamilies #-}
module Hawk.View.JsonView
  ( JsonView (..)
  , jsonView
  , jsonDecode
  , jsonEncode
  , JSON (..)
  , jObject, jArray, jString, jXml, jNumber, jInt
  , jInteger, jDouble, jBool, jNull
  ) where

...
imports
...

data JsonView a =
  JsonView {toJson :: a -> StateController JSON}

jsonView :: (a -> StateController JSON) -> JsonView a
jsonView = JsonView

instance View (JsonView a) where
  type Target (JsonView a) = a
  -- :: a -> Target a -> StateController ByteString
  render jv = liftM jsonEncode . toJson jv

jsonEncode :: JSON -> ByteString
jsonEncode = encode Compact

jsonDecode :: ByteString -> (String, JSON)
jsonDecode s = either l r $ decode s
  where l = (\(e,_) -> (e, Null))
        r = (\j -> ("", j))

jObject :: [(String, JSON)] -> JSON
jObject [] = Object empty
jObject l = Object $ fromList $ ol l
  where
  ol :: [(String, JSON)] -> [(KeyString, JSON)]
  ol [] = []
  ol ((s,j):xs) = (U.fromString s, j) : (ol xs)

...
```

Listing 4.1: View module for JSON encoded strings

This view is analogue to the `TextView`. It got a data type with one constructor and a single function to convert a given type to `StateController JSON`. Furthermore the view function `jsonView` is implemented using the constructor. This function will be used in the routing list of a web application, to render the view function with this view. A concrete instance of the `View` class is needed for rendering. Here the `toJson` function returns the result of a view function of a web application and the `jsonEncode` function converts it to a `ByteString` representation.

This presentation module uses the JSONb package for decoding, encoding and constructor methods to generate the internal representation of JSON [Dus10]. It is to the developer of an application to use JSON constructors, but it is recommended to use these functions for creating JSON instead. This is to uncouple the implemented JSON representation in Hawk from a web application. The `jsonDecode` function is predicted to be used rarely, but if a web application uses another web service which encodes its data as JSON, this function might be used to work on that data in that web application, e.g. save parts of it to DB.

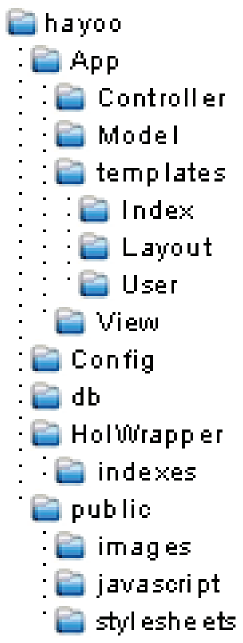
An example of how `JsonView` can be used in web applications is given in the `AjaxView` module of Hayoo 5.5.3.

5 Hayoo! API Search

The *Hayoo! API Search* is an existing web application written by Timo B. Hübel and Sebastian M. Schlatt. This application was reimplemented in this thesis by using the Hawk framework, to give an example implementation of a web application and to improve Hayoo by uncouple search engine specific data from web application and customized search.

Hayoo offers a web interface and web service for searching the Haskell API in on hackage [hackage]. Therefore the pages are indicated by a web crawler to a predefined structure. This generates an index and a documents file. The index file contains mapping from word to document ID, which in the case of Hayoo is a function and the document file contains document information consisting of function, package and module name, signature and an URL to the module on hackage. With the Holumbus framework the index and document data can be queried for a search term. The later mentioned optional cache component contains additional data. In the case of Hayoo it is only used for supplying a detailed description of each function.

The main parts of this web application are the three ways of interaction via HTML, HXR respectively Ajax and as web service, the configuration of search query parameters, parsing of a search string and ranking of the results. These parts are structured by the MVC pattern. This is the recommended structure for a web application using Hawk. Only the Holumbus wrapper is excluded from this structure. In figure 5.2 a scheme for the structure of Hayoo is shown. It leads to the following directory structure, which is described in detail in Appendix A.1.



```
hayoo
  App
    Controller
    Model
    templates
      Index
      Layout
      User
    View
  Config
  db
  HolWrapper
    indexes
  public
    images
    javascript
    stylesheets
```

Figure 5.1.: The Hayoo web application folder structure

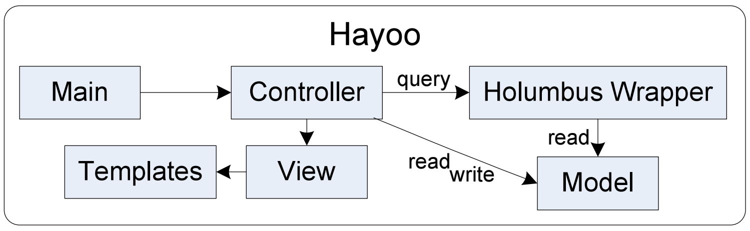


Figure 5.2.: Components of the Hayoo web application

The Hayoo web application has a main part that consists of an application constructor, a configuration and a routing for controllers. The routing leads the Hawk framework to call the controllers on a client request. A controller can query the Holumbus wrapper for a search query or invoke the model to load or save user configuration data. The Holumbus wrapper can query the model for configuration data, to do a customized search and return a result to the controller. The controller then will pass it to a view and the view generates a response with help of a template.

Configuration and Holumbus wrapper are independent of the special MVC parts. In the case of routing they can be used in the controller as well as in the view component of the application. The public directory in this application is configured as the root web directory. So a client can not request source files from other directories directly.

The first section of this chapter describes the Holumbus wrapper. An explanation of how to reuse the Holumbus wrapper for another specialized search engine web application using Holumbus and Hawk is given in chapter 6. Section 5.2 describes the configuration of Hayoo and the sections 5.3, 5.4, 5.5 and 5.6 describe controller, model and view and template in detail.

5.1. Wrapper

The Holumbus wrapper encapsulate the functionality from the Holumbus framework. This gives the advantage of changing the underlying search engine of a web application, to change the specialization of the search engine or to exchange the web interface easily without having some unexpected side effects. The following figure depicts the structure of the Holumbus wrapper. The *Holumbus Wrapper* has an interface that

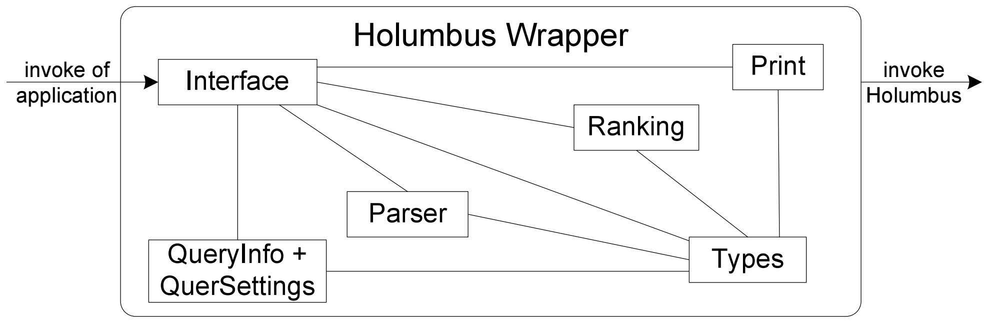


Figure 5.3.: The Holumbus-Wrapper architecture

can be used by the Hayoo web application part. It encapsulates all other parts of the wrapper and the *Types* part provides types and classes for them. All parts using types and functions of the Holumbus framework.

The public interface of the `HolWrapper` module reexports most of the modules, two public types, one query function and four common printing functions.

```
module HolWrapper.HolWrapper
  ( module HolWrapper.Common
  , module HolWrapper.Parser
  , module HolWrapper.Print
  , module HolWrapper.QueryInfo
  , module HolWrapper.QuerySettings
  , module HolWrapper.Ranking
-- Types
  , SearchResult
  , ResultTuple

-- Query
  , query

-- Print
  , numResults
  , numWordCompletions
  , getSearchString
  , getOffset )
```

The types will be described in paragraph 5.1.1 in detail. The query method got three configuration parameters also described in the subsequent paragraphs. It parses the query string using the `QueryParser` function, then running the query with the information of `QueryInfo` and optionally ranking the result with the `Ranking` function.

```
query :: QueryInfo
      -> QueryParser
      -> Ranking
      -> SearchResult
query qi qp r = either Left right parse
  where parse = qp $ qs qi
        right q =
          Right (maybe proc
                   (\x -> rank x proc)
                   (r $ qs qi)
                ,qi)
          where proc = processQuery
                        (processConfig $ qs qi)
                        (index qi)
                        (documents qi)
                         q

qs :: QueryInfo -> QuerySettings
qs = querySettings
```

The shown print methods, offered by this interface, are common methods. They will be used by nearly every search engine web application. Custom methods for formatting the search results are described in paragraph 5.1.3 . The offered methods return the number of results found, the number of word completions found, the used search string and the offset for displaying results. The offset mainly is used for navigating in a search result.

```
numResults :: ResultTuple -> Int
numResults (r, _) = sizeDocHits r

numWordCompletions :: ResultTuple -> Int
numWordCompletions (r, _) = sizeWordHits r

getSearchString :: ResultTuple -> String
getSearchString (_, i) = searchString $ qs i

getOffset :: ResultTuple -> Int
getOffset (_, i) = offset $ qs i
```

5.1.1. Types

The types of Holumbus wrapper are divided into four categories:

1. Public Types, that are known to the Hayoo web application part,
2. Internal Type Representation, for the search engine, query settings and query information,
3. Application Configuration Type, for holding the search index and documents in the main memory and
4. Instances for loading the Holumbus index.

Public Types

For transmission of query results through a web application, for example from controller to view, there are two public types. A web application should never use more than these to get not coupled to tightly to the search engine implementation.

```
type SearchResult = Either String ResultTuple
type ResultTuple = (Result FunctionInfo, QueryInfo)
```

The `SearchResult` type returns either an error string or an `ResultTuple`, whereas the tuple contains the search result and the query information. That the result is needed for formatting is clear. The query information is also needed, because it contains an offset, the search string and the handle to the cache for loading a function description.

Internal Type Representation

The internal type representation contains types and data types for parser and ranking signatures, the index's document custom information and a data type for query settings, query string and the information base.

```
type QueryParser = QuerySettings -> Either String Query
type Ranking = QuerySettings -> Maybe (RankConfig FunctionInfo)
```

The both types for parsing and ranking only exist to have tinier and understandable type signatures. They both expect a `QuerySettings` type as input. The `QueryParser` function returns either an error string or the parsed query. The `Ranking` function returns an optional ranking configuration working on the index's document information described below.

```
data FunctionInfo = FunctionInfo
  { moduleName :: ByteString
  , signature  :: ByteString
  , package    :: ByteString
  , sourceURI  :: Maybe ByteString
  } deriving (Show, Eq)
```

A normal Holumbus document contains a title, an *Uniform Resource Identifier* (URI) and an optional custom data type, which is `FunctionInfo` in the case of Hayoo. The title of a document is the function name and the additional information about the module name, signature, package and the optional source URI is packed in this data type. Though there was no change in the indexer, this data type was directly taken from the old implementation.

```
data QueryInfo = QueryInfo
  { querySettings :: QuerySettings
  , index         :: Persistent
  , documents     :: SmallDocuments FunctionInfo
  , cache         :: Cache
  }
```

```
data QuerySettings = QuerySettings
  { searchString  :: String
  , offset        :: Int
  , caseSensitive :: Bool
  , processConfig :: ProcessConfig
  , modules       :: [RConfig]
  , packages      :: [RConfig]
  }

data RConfig = Rank (String, Score)
             | Name String
```

These three data types contain all the information that is needed for a search query out of the parsing and ranking function. It contains an index, documents and cache handle, as well as the search string, an offset and some configuration parameters. The `RConfig` type is an additional not natively supported type for configuring the search and ranking function. It can either contain a root module or package name or a tuple of it with a scoring parameter for ranking as second argument. `ProcessConfig` is a data type defined by Holumbus for configuring a search. A more detailed description is given in the paragraph about the Holumbus wrapper query mechanism.

Application Configuration Type

Thus Hayoo has no other data that is used application wide in addition to the configuration data, the `AppConfiguration` instance is defined in this module. If other data needs to be available, then the wrapper can only provide the `loadHayooConfig` function to generate the `HayooConfig` data type.

```
data HayooConfig = HayooConfig
  { indexHandler :: Persistent
  , docsHandler  :: SmallDocuments FunctionInfo
  , cacheHandler :: Cache
  }

instance AppConfiguration HayooConfig where
-- getInstance :: (AppConfiguration a, MonadIO m) => m a
  getInstance = loadHayooConfig
```

`HayooConfig` contains the three parts of `QueryInfo` that should be loaded once on application start. The instance of `AppConfiguration` just uses the `loadHayooConfig` function to generate the data type and load the data. The path to the index, documents and cache file are hard coded. They are read by the Holumbus function `loadFromFile` and `createCache`. The first function uses a `XmlPickler` or `Binary` instance to parse these files. The instances are given in the next paragraph.

```
loadIndex :: MonadIO m => m Persistent
loadIndex = liftIO
  $ loadFromFile "./HolWrapper/indexes/hayoo-index.bin"

loadDocs :: MonadIO m => m (SmallDocuments FunctionInfo)
loadDocs = liftIO
  $ loadFromFile "./HolWrapper/indexes/docs-small.bin"

loadCache :: MonadIO m => m Cache
loadCache = liftIO
  $ createCache "./HolWrapper/indexes/cache.db"

loadHayooConfig :: MonadIO m => m HayooConfig
loadHayooConfig = do
  i <- loadIndex
  d <- loadDocs
  c <- loadCache
  return $ HayooConfig
    { indexHandler = i
    , docsHandler  = d
    , cacheHandler = c
    }
```

Instances for loading the Holumbus index

The following instance to load the Holumbus index and documents are taken from the old implementation. XML and binary format are supported.

```
instance XmlPickler FunctionInfo where
  xpickle = xpWrap (fromTuple, toTuple) xpFunction
    where
    fromTuple (m, s, p, r) = FunctionInfo
                               (B.fromString m)
                               (B.fromString s)
                               (B.fromString p)
                               (liftM B.fromString $ r)
    toTuple (FunctionInfo m s p r) = ( B.toString m
                                     , B.toString s
                                     , B.toString p
                                     , liftM B.toString $ r)
    xpFunction = xp4Tuple xpModule xpSignature xpPackage
       xpSource
    where
    xpModule = xpAttr "module" xpText0
    xpSignature = xpAttr "signature" xpText0
    xpPackage = xpAttr "package" xpText0
    xpSource = xpOption (xpAttr "source" xpText0)

instance Binary FunctionInfo where
  put (FunctionInfo m s p r) = put m >> put s >> put p >> put r
  get = do
        !m <- get
        !s <- get
        !p <- get
        !r <- get
        return $! FunctionInfo m s p r
```

5.1.2. Query

The Hayoo web application part will use the `query` function described in the introduction part to the Holumbus wrapper. This function needs the three parameters *QueryInfo*, *QueryParser* and *Ranking*, which will be explained in detail in this paragraph. They all can be created using constructor methods.

QueryInfo

As seen in Holumbus wrapper types `QueryInfo` is composed out of three handlers and the `QuerySettings` data type. The three handlers are two for the search index file and the search documents file and one for the cache database, which holds additional full text information.

```
mkQueryInfo :: StateController (Maybe QueryInfo)
mkQueryInfo = do
  acfg <- asks appConfiguration
  cfg <- acfg
  q <- lookupParam "q"
  case q of
    Nothing -> return Nothing
    Just qr -> do
      qs <- mkQuerySettings qr
      return $ Just $ QueryInfo
        { querySettings = qs
        , index         = indexHandler cfg
        , documents     = docsHandler cfg
        , cache          = cacheHandler cfg
        }
```

The method `mkQueryInfo` tries to create a `QueryInfo` data type. It reads the search string from the client request and the three handlers from the application type. If no query string is available this method will return `Nothing` and it is up to the application to chose what to do. Hayoo in that case will redirect to the index action.

```
module App.HolWrapper.QuerySettings
  ( mkQuerySettings
  ) where

import App.HolWrapper.Types (QuerySettings (..))
import App.HolWrapper.Common
import qualified App.Model.User as U
import qualified App.Controller.UserController as UC

import Hawk.Controller
import Holumbus.Query.Fuzzy (FuzzyConfig (..))
import Holumbus.Query.Processor (ProcessConfig (..))

import qualified Data.List as L
import qualified Data.Map as M

mkQuerySettings :: String -> StateController QuerySettings
mkQuerySettings q = do
  o <- getParam "o"
  querySettingsByRequest [q,o]

querySettingsByRequest :: [String]
                       -> StateController QuerySettings
querySettingsByRequest l = do
  m <- getParams
  if not $ toBool $ M.findWithDefault "" "singleConfig" m
    then querySettingsBySession l
    else return $ listToQuerySettings $ l ++ (tLL m)
      where tLL m = toLookupList m settingElems

querySettingsBySession :: [String]
                       -> StateController QuerySettings
querySettingsBySession l = do
  username <- isAuthedAs
  if null $ maybe "" id username
    then querySettingsByDefault l
    else do
      qrs <- getQuerySettingsFromSession settingElems
      let qs = filter (not . null) qrs
      if settingLength == (length qs)
        then return $ listToQuerySettings $ l ++ qs
        else querySettingsByDatabase l

querySettingsByDatabase :: [String]
                        -> StateController QuerySettings
querySettingsByDatabase l = do
  username <- isAuthedAs
  if null $ maybe "" id username
    then querySettingsByDefault l
    else do
      user <- UC.getCurUser -- TODO catch monadic exception
      return $ listToQuerySettings $ l ++ (userToQSList user)

querySettingsByDefault :: [String]
                       -> StateController QuerySettings
querySettingsByDefault l = return $ listToQuerySttings l

getQuerySettingsFromSession :: [String]
                            -> StateController [String]
getQuerySettingsFromSession = mapM f
  where f x = do
          v <- getSessionValue x
          return $ maybe [] id v

toLookupList :: M.Map String String -> [String] -> [String]
toLookupList m = L.map (\s -> M.findWithDefault "" s m)

listToQuerySettings :: [String] -> QuerySettings
listToQuerySettings [] =
  QuerySettings "" 0 False defaultProcessConfig [] []
listToQuerySettings (q:o:c:oq:w:r:s:f:re:m:p:_) = QuerySettings
  { searchString  = q
  , offset        = toInt o
  , caseSensitive = toBool c
  , processConfig = ProcessConfig
    { fuzzyConfig   = FuzzyConfig
      { applyReplacements  = toBool r
      , applySwappings     = toBool s
      , maxFuzziness       = toFloat f
      , customReplacements = toReplacements re
      }
    , optimizeQuery = toBool oq
    , wordLimit     = toInt w
    }
  , modules       = toRConfig m
  , packages      = toRConfig p
  }
listToQuerySettings (q:o:_) =
  QuerySettings q (toInt o) False defaultProcessConfig [] []
listToQuerySettings (q:_) =
  QuerySettings q 0 False defaultProcessConfig [] []

userToQSList :: U.User -> [String]
userToQSList u = filter (not . null)
               $ L.map (getList $ U.toList u) settingElems
  where getList l s = maybe "" id $ L.lookup s l
```

Listing 5.1: The Wrappers `QuerySettings` module

As depicted in listing 5.1 the constructor method for the `QuerySettings` type is a little more complex. That is because it first tries to get the configuration parameters from request, then from session data, then from DB and at last it uses default settings if no other were found. Additionally this constructor method reads the optional offset parameter from the request.

QueryParser

The parser module has two public constructor methods for creating a `QueryParser`.

```
customParser :: QueryParser
customParser =
  let s = searchString qs
  in caseSense $ customParseQuery $ addr "module" $ addr "
     package" s
  where addr r s = s ++ (concat $ map (rCTC r) $ modules qs)
        caseSense (Left s) = Left s
        caseSense (Right q) | caseSensitive qs = Right $ toCase
            q
                            | otherwise = Right q

defaultParser :: QueryParser
defaultParser = parseQuery . searchString
```

The `defaultParser` method uses the native parsing method from Holumbus to parse the search string. The `customParser` uses a custom parsing function instead. This parsing function is taken from the old implementation of Hayoo and supports parsing for fuzzy search as well as a correct parsing of signatures. So Hayoo uses the `customParser` function and adds module and package restrictions to it as well as case sensitivity if configured.

Ranking

Like the parser the ranking module has more than one public constructor. In this case there are three, because a programmer can choose between custom, default and no ranking.

```
customRanking :: Ranking
customRanking qs = Just $ RankConfig (docsRanking qs tms)
                                     (wordsRanking tms)
  where tms = splitAll (== ' ') $ searchString qs

defaultRanking :: Ranking
defaultRanking _ = Just
  $ RankConfig docRankByCount wordRankByCount

emptyRanking :: Ranking
emptyRanking _ = Nothing
```

The `emptyRanking` function just returns `Nothing`, so the result will not be ranked in any way. `defaultRanking` uses a really simple ranking algorithm. It just ranks the documents by the occurrence of words in it and the words by their occurrence in documents.

The `customRanking` function uses a more complex ranking algorithm, which depends on the searched words to rank the results. For example a higher ranking is done for exact matches. The ranking of documents uses the terms of the search string too and additionally the `RConfig` lists from `QuerySettings` to let users influence the ranking algorithm.

5.1.3. Print

In the introduction to the Holumbus wrapper some basic print function were shown. The print module extends these by specialized functions for printing application specific data. All the functions work on the search Result, therefore they are encapsulated in this wrapper. All print methods in this module expect a `ResultTuple` as input and will return an XML tree or JSON data to the view, so this data can be rendered and transmitted to the client.

The Hayoo print module publishes nine methods for formatting a result. Five of them are for feeding the HTML interface with content. The remaining four formatting as JSON for the web service API. The methods are:

1. **formatCloud** extracts the list of found words, if fuzzy search is activated, and assigns CSS classes to them for showing them in different size. This is based on the frequency of occurrence.
2. **formatOffsetList** creates a list of ten result functions, based on query result and offset. Each function is formatted by module tree, function name, signature, package and description. This function formats the search result content. For the description the cache DB is requested.
3. **formatStatus** just generates the status message displayed in the gray bar with the number of documents and word completions found.
4. **formatPages** counts the number of results and uses the offset to determine what page is currently viewed. It generates the page index at the bottom for changing between result pages.
5. **formatPM** formats the top lists of modules and packages on the right side of the web interface. Therefore the results are counted by their root module, respectively the package.
6. **formatApiFunctions** creates a JSON array of the function information. It is analogue to `formatOffsetList` except that it formats all found functions and not restricts to ten of them.
7. **formatApiCompletions** returns an JSON array of word completions and their occurrences in the documents.
8. **formatApiModules** generates a JSON array of root module names and the occurrences of this root module in the search result.
9. **formatApiPackages** is analogue to `formatApiModules`, it just returns packages instead of modules.

As example `formatCloud` is depicted in the following. `wordHits` generates a list of words and their occurrence in a document, `toSortedScoreList` converts this into a list where all occurrences of a word are summed up. The `cloud` function generates a list of HTML span tags, each containing a link with the special word as content. The `href` and `onclick` will cause a new search request with the word as query string. The CSS class of the link is calculated by normalizing the word score from zero to nine. A higher score so results in a bigger font size. If no ranking is done all words will have size three.

```
formatCloud :: ResultTuple -> H.XmlTrees
formatCloud (r, _) = let max = maxScoreWordHits r
                in cloud max $ toSortedScoreList $ wordHits r

cloud :: Float -> [(Word,Score)] -> H.XmlTrees
cloud m [] = []
cloud m ((w,s):xs) = (spanClass "clouds" cloudLink)
                   : (H.text " ")
                   : cloud m xs
  where cloudLink = [H.contentTag "a" (cAttr w) [H.text w]]
        where cAttr w =
                [ ("class","cloud"++cloudScore)
                , ("href","/index/search?q=" ++ w)
                , ("onclick",
                   "return processQuery(" ++ w ++ ",0)")]
        cloudScore | m < 0.1 = show 3
                   | otherwise = show
                       $ round (9 - ((m - s) / m) * 8)
```

5.2. Configuration

Hayoo's configuration of Hawk consists of two functions generating the types `AppEnvironment` and `BasicConfiguration` as well as the `AppConfiguration` instance in the Holumbus wrapper, the controller routing list and the main module as web application entrance point.

```
development :: AppEnvironment
development = AppEnvironment
  { connectToDB = ConnWrapper ‘liftM‘
                    connectSqlite3 "./db/database.db"
  , logLevels   = []
  , envOptions  = []
  }
```

The `development` method only loads the database connection handler. The database contains the user table for saving custom search configuration.

```
configuration :: BasicConfiguration
configuration = BasicConfiguration
  { sessionStore = cookieStore
  , sessionOpts  =
      [("secret", ****)]
  , authType     = dbAuth
  , authOpts     = ["user", "username", "password"
```

```
                    , "", "username", "password"]
  , routing       = simpleRouting Routes.routing
  , templatePath = "./App/template"
  , publicDir    = "./public"
  , confOptions  = []
  , error401file = "401.html"
  , error404file = "404.html"
  , error500file = "500.html"
  }
```

The `configuration` method defines what type of session handling to use, the sessions secret key, the authentication method and options as well as that `simpleRouting` is used with the routing list generated by the `Routes` module. Furthermore a path to the base web directory and the template directory is defined as well as where to find the files to return if errors occurred. These files are relative to the `publicDir` path.

```
module Config.Routes where

import qualified App.Controller.IndexController as IC
import qualified App.Controller.UserController as UC
import qualified App.Controller.AjaxController as AC

import Hawk.Controller.Types (Controller)
import qualified Data.Map as M

routing :: M.Map String (M.Map String Controller)
routing =         M.singleton "hayoo" (M.fromList IC.routes)
        `M.union` M.singleton "index" (M.fromList IC.routes)
        `M.union` M.singleton "user"  (M.fromList UC.routes)
        `M.union` M.singleton "ajax"  (M.fromList AC.routes)
```

Listing 5.2: Hayoo's controller routing module

As shown in listing 5.2 the path `/hayoo/` and `/index/` are mapped to the `IndexController`. So `/index/search` and `/hayoo/search` will result in the same output.

```
module Main (main) where

import Config.Config (configuration, development)
import App.HolWrapper.Types --import Config.Types

import Hawk.Controller.Initializer (getApplication)
import Hawk.Controller.Types (AppConfiguration (..))
import Hack.Handler.SimpleServer as Server (run)
import Control.Monad.Trans

main :: IO ()
main = run 80
  $ getApplication getInstance development configuration
```

Listing 5.3: Instantiating point of the Hayoo web application

In the main function of the main module the three types `AppEnvironment`, `BasicConfiguration` and `AppConfiguration` are passed to initially call the web application. The `getApplication` function will return an `Application` type to the hack handler server.

5.3. Controller

The controller component of Hayoo is split into three controllers

1. **IndexController** handles standard requests like static pages and non Ajax search request.
2. **UserController** handles all user specific actions like registration, login and logout as well as changing configuration data or deleting the account.
3. **AjaxController** treats Ajax search requests and provides the web service API.

The following three paragraphs give a more detailed description of the three controllers.

5.3.1. IndexController

The easiest way to show how a controller in Hawk is working is to look at its routing list. The list links a string to a defined action and its view.

```
routes :: [Routing]
routes =
  [ ("index"
    ,indexAction >>= render (typedView "index" indexXhtml))
  , ("search"
    ,searchAction >>= render (typedView "search" searchXhtml))
  , ("config"
    ,showConfigAction >>=
      render (typedView "config" configXhtml))
  , ("help",getUser >>= render (typedView "help" helpXhtml))
  , ("about",getUser >>= render (typedView "about" aboutXhtml))
  ]
```

This list of the `IndexController` contains five entries and results in displaying five different pages. The first part of each tuple is the actions name, which is compared to a path requested by a client. The second element is an action that is bound to a view with the bind operator (>>=). So the result of the action will be passed to the rendering function as parameter.

In this controller all actions have one thing in common. They all pick the user ID if a user is logged in and pass it to the `IndexView` methods. The intention is to display the login form if the user is not logged in and to display the logout link if he is.

The two actions "help" and "about" only show static pages with a customized header for logged in users. Whereas `showConfigAction` picks the search string parameter from input additionally to the user ID and runs the `configXhtml` method for generating a configuration form. Subsequently the both actions `indexAction` and `searchAction` are shown.

```
indexAction :: StateController String
indexAction = do
  q <- lookupParam "q"
  case q of
    Nothing -> getUser
    Just v  -> redirectWithParams "index" "search"

searchAction :: StateController (SearchResult, String)
searchAction = do
  qi <- mkQueryInfo
  case qi of
    Nothing -> redirectToAction "index" "index"
    Just v -> do
      u <- getUser
      return (query v customParser customRanking, u)

getUser :: StateController String
getUser = (maybe "" id) `liftM` isAuthedAs
```

These both actions interact with each other. If `indexAction` is accessed with a search string parameter it redirects users to the search action. Otherwise the user ID is picked and a static page is displayed. The `searchAction` acts the other way around. If there is no search string users is redirected to the index action. `mkQueryInfo` returns `Nothing` if there is no search string available. Otherwise the generated `QueryInfo` type is used to call the `query` method of the Holumbus wrapper with `customParser` and `customRanking`.

5.3.2. UserController

The `UserController` is an ideal example for applying the `combine` function, which was described in section 4.1.3. The `UserController`'s routing list is the following.

```
routes :: [Routing]
routes =
  [ ("register"
    ,registerAction >>= render (typedView "register"
       registerXhtml))
  , ("login",authAction >> redirectToAction "index" "index")
  ] ++
  combine (authF' "You are not logged in." "index" "index")
  [ ("index"
    ,indexAction >>= render (typedView "index" indexXhtml))
  , ("show"
    ,indexAction >>= render (typedView "index" indexXhtml))
  , ("edit",editAction >> redirectToAction "user" "index")
  , ("logout",logoutAction >> redirectToAction "index" "index")
  , ("delete",deleteAction >> redirectToAction "user" "logout")
  ]
```

The first two actions can be accessed without being logged in, for the subsequent five actions users need to be logged in. This routing list is constructed of two lists concatenated by the (++) operator and the `combine` function with the `authF'` function as parameter is applied to the second list. So each not logged in user who wants to access one of these actions will be redirected to the `IndexController` and its `indexAction` and receives an the error message *"You are not logged in."*.

registerAction either returns nothing and let the view show a registration form or it reads the passed parameters and tries to insert a new user to the database. If the user was set successfully a flash message will be set and he is redirected to the `authAction` with his user ID and password as parameters.

authAction runs the `tryLogin` method from the authentication interface, creates a flash message from the response and redirects the user to the `/index/index` path. The redirection is defined in the routes function above.

indexAction can be triggered by calling `/user/index` or `/user/show`. It reads the users configuration from DB and passes it to the view to show the configuration editing form.

editAction takes the request parameters passed to it and tries to make the users configuration permanent by writing them to DB. After this users are redirected to the editing form.

logoutAction just calls `logout` from the authentication interface and redirects users to the fron page `/index/index`

deleteAction tries to delete the currently logged in user, sets a flash message and redirects to the `logoutAction`.

This routing list is interesting, because there are so many different ways of routing. "register" is the normal way, "index" and "show" show how many names can result in the same action and "login" and "edit" show how to do a redirect in the routing list. This is much better than doing it in the function on every branch, it helps to scrap the boilerplate. A really special case is shown by "logout" and "delete", redirection can cascade over different actions. In this case a user who is deleted should not stay logged in, so he is logged out by redirecting him.

If a user was redirected he can hardly figure out if his first called action was successfully executed. Therefore Hawk supports so called *flash messages*. These messages are passed through redirects until the next response to the client is returned. The below shown `authAction` is an example of it. A flash message is generated when the user calls the function. After the function was executed the user is redirected to the `/index/index` path and if the authentication or login was successful he gets an *"Authentication succeeded"* message displayed.

```
authAction :: StateController ()
authAction = tryLogin >>= flashAuth

logoutAction :: StateController ()
logoutAction = logout

flashAuth :: AuthResult -> StateController ()
flashAuth AuthSuccess =
  setFlash "success" "Authentication succeeded."
flashAuth AuthFailureIdNotFound =
  setFlash "error" "Username not found."
flashAuth AuthFailureInvalidCredential = ...
flashAuth _ = ...
```

5.3.3. AjaxController

The AjaxController only contains two actions, “search” and “api”.

```
module App.Controller.AjaxController where

import App.View.AjaxView
import App.HolWrapper

import Hawk.Controller
import Hawk.View

routes :: [Routing]
routes =
  [ ("api", searchAction >>= render (jsonView apiJson))
  , ("search", searchAction >>= render (jsonView searchJson))
  ]

searchAction :: StateController SearchResult
searchAction = do
  qi <- mkQueryInfo
  case qi of
    Nothing -> return $ Left "No query to parse."
    Just v -> return $ query v customParser customRanking
```

Listing 5.4: Hayoo’s controller module for Ajax requests

In listing 5.4 it is obvious that the two actions are identical and they only differ in the used view. The searchAction here is nearly equal to the one in the IndexController with the little difference that an error is passed to the view instead of redirecting the client if no search string is given.

5.4. Model

Hayoo’s database model is really simple, because each user has exactly one search configuration and therefore all data is contained by one table. A Hayoo user account does not make any sense if there is no configuration assigned to it. How the configuration parameters were chosen is easy to explain. They consist of ProcessConfig type parameters, then a parameter for case sensitive search and two parameters for a list of root modules and packages were chosen. Hayoo is a highly specialized search engine with a limited information base and users that know much about the material.

So they get a maximum in configuration possibilities. The configuration options can also be seen in the `QuerySettings` data type.

If a user or a group of users work on a project they may want to create an account and restrict the search to some packages and module, so they only get results from packages they use in their project while they are logged in. This may speed up searching for the right functions and working on that project. When users do not find the function they searched for, they only have to log out to get the search result without any restriction.

The user module only exports an `User` data type and a `toList` function. These two are depicted below.

```
data User = User
  { _uid            :: PrimaryKey
  , username        :: String
  , password        :: String
  , email           :: Maybe String
  , caseSensitive   :: Maybe Bool
  , optimizeQuery   :: Bool
  , wordLimit       :: Int
  , replace         :: Bool
  , swapChars       :: Bool
  , replacements    :: Maybe String
  , maxFuzzy        :: Double
  , modules         :: Maybe String
  , packages        :: Maybe String
  } deriving (Eq, Read, Show)

toList :: User -> [(String, String)]
toList u = map (\(x,y) -> (x, show y)) $ toSqlAL u
```

The `toList` function converts a user date to a key value list. This is used to fill a configuration form with the current configuration data of that user.

The `User` data type has all configuration options mentioned in previous sections and one additional parameter of type `PrimaryKey`. It is an auto incrementing key in the SQL representation of this data type and necessary in Hawk's current state of development. All parameters with type `Maybe` can be `NULL` in the database.

The user module also has instances for the classes `Persistent`, `WithPrimaryKey`, `Model`, `Updateable` and `Validatable` from the user data type. `Persistent` configures the table and column names as well as converting a `User` date to and from SQL. `WithPrimaryKey` configures the primary key column and `Model` offers a customized function for creating a new empty user date with the function new. Updateable updates a data type for example generated with new and works in combination of Validatable which checks if the updated data is valid. The validation configuration for User only checks if username and password are not empty and if username is unique in this table.

5.5. View

Each controller has a corresponding view and so the view is divided into three parts for index, user and Ajax. But the view has the two additional modules HtmlCommon and Utils for common functionality.

Utils offers different functions for example for showing page links in the upper right corner of the interface, rendering the login form based on an input user name or filling the current search string into the search input field. In addition to that functions for formatting forms like the advanced search form or user configuration form are supplied.

HtmlCommon offers often used HTML tags in simplified form. These tags only extend the ones offered by Hawk. For example

```
contentTag "div" [("id",attribute)] content
```

is shortened to

```
divId attribute content
```

whereas content is a list of XML trees.

To give an example of how the functions in Utils look like the showLogin function is shown below.

```
showLogin :: String -> XmlTrees
showLogin [] = [form "loginform" "post" "/user/login" []
                 [(textfield "username" "Username" []),
                  (password "password" "Password"
                     [("id", "password")]),
                  (formButton "authbutton" "Login" [])
                 ]
               ]
```

form generates a new HTML form with id attribute "loginform" and calling the /user/login action on submit. textfield builds an input element for normal text with name and id attribute "username" and content "Username". password does the same, but it needs to add the id explicitly. formButton generates a button, here with label "Login" and id "authbutton".

5.5.1. IndexView

IndexView formats action results given by the IndexController. The three functions indexXhtml, helpXhtml and aboutXhtml are completely identical. The function are only given separately because the Hawk template engine will choose a different template file for all of them and there is different static content in each file. The parameters used by this three functions are only for manipulating the page header which is common to all Hayoo pages. A more detailed description is given in the subsequent section 5.6.

A main part of the IndexView is the *Template Haskell* function viewDataType. The different view data types have to be created before the using function is defined and this data types assure that all parameters of the data type are in the used template and the other way around. If there is any incongruity between the XHTML bindings and the generated view data type, then the compiler will show an error.

The remaining two functions configXhtml and searchXhtml use another data type as the three depicted above. This is visible in listing 5.5. configXhtml has one more parameter for the generated form, which is generated in the Utils module. searchXhtml has four more parameters for the print methods from the Holumbus wrapper for creating the word cloud, result list, top list for modules and packages and for printing the navigation from search result data.

```
{-# LANGUAGE TemplateHaskell #-}
module App.View.IndexView where

import Hawk.Controller
import Hawk.View.Template.DataType
import qualified Hawk.View.Template.HtmlHelper as H

import App.HolWrapper
import App.View.Util
```

```
$(viewDataType "Index" "index")
$(viewDataTypeWithPrefix "Search" "Index" "search")
$(viewDataTypeWithPrefix "Config" "Index" "config")

indexXhtml :: String -> StateController IndexIndex
indexXhtml = defaultIndexPage

searchXhtml :: (SearchResult, String)
            -> StateController IndexSearch
searchXhtml (Right r, user) = return IndexSearch
  { searchTitle = pageTitle
  , searchMystatus = formatStatus r
  , searchCloud = formatCloud r
  , searchList = formatOffsetList r
  , searchLogin = showLogin user
  , searchSettings = showSettings user
  , searchQuerytext = mkQueryText $ getSearchString r
  , searchToppm = formatPM r
  , searchPages = formatPages r
  }

configXhtml :: (String, String) -> StateController IndexConfig
configXhtml (q, user) =
  return IndexConfig
    { configTitle = pageTitle
    , configMystatus = statusDefaultText
    , configLogin = showLogin user
    , configSettings = showSettings user
    , configForm = singleRequestConfig q
    , configQuerytext = mkQueryText q
    }

helpXhtml :: String -> StateController IndexIndex
helpXhtml = defaultIndexPage

aboutXhtml :: String -> StateController IndexIndex
aboutXhtml = defaultIndexPage

defaultIndexPage :: String -> StateController IndexIndex
defaultIndexPage user =
  return IndexIndex
    { title = pageTitle
    , mystatus = statusDefaultText
    , login = showLogin user
    , settings = showSettings user
    , querytext = mkQueryText ""
    }
```

Listing 5.5: Hayoo's presentation module for basic pages

5.5.2. UserView

The `UserView` is tightly coupled to its controller. It has two methods for showing the user configuration and the registration form. There is only one difference between these two methods. The `indexXhtml` for showing a user configuration form has a content parameter, because the form is generated in Haskell and is not available as template to fill in existing user data.

```
indexXhtml :: User -> StateController UserIndex
indexXhtml u =
  return UserIndex
        { title = pageTitle
        , mystatus = statusDefaultText
        , login = showLogin $ username u
        , settings = showSettings $ username u
        , querytext = mkQuerytext ""
        , content = showUser u
        }
```

All the above shown functions are from the `Utils` module. `showUser` generates a user form and inserts the users data to the corresponding fields.

5.5.3. AjaxView

There is no huge difference between `AjaxView` and `IndexView` out of formatting as JSON respectively as XML. The `AjaxView` needs no `viewDataType`, because the `JsonView` of Hawk do not need it if there is no template. In return `AjaxView` has to handle an error case that the `IndexController` solves by redirecting, but is not possible for Ajax requests. `searchJson` and `apiJson` are nearly identical. They only differ in the Holumbus wrapper print methods they use.

```
searchJson :: SearchResult -> StateController JSON
searchJson (Left s) = return $ jObject
  [ ("q", jString "")
  , ("offset", jInt 0)
  , ("status", jString s)
  , ("cloud", jString "")
  , ("documents", jString "")
  , ("pages", jString "")
  , ("toppm", jString "")
  ]
searchJson (Right r) = return $ jObject
  [ ("q", jString $ getSearchString r)
  , ("offset", jInt $ getOffset r)
  , ("status", jXml $ formatStatus r)
  , ("cloud", jXml $ formatCloud r)
  , ("documents", jXml $ formatOffsetList r)
  , ("pages", jXml $ formatPages r)
  , ("toppm", jXml $ formatPM r)
  ]
```

As visible Hayoo's `searchJson` method puts XML to the JSON object values in its current version. This is not that efficient, but if they are interchanged by the API print methods this view will also work. Only the JavaScript on client side has to process a little more.

Following structure is generated by `apiJson` using the corresponding methods from `HolWrapper.Print` module.

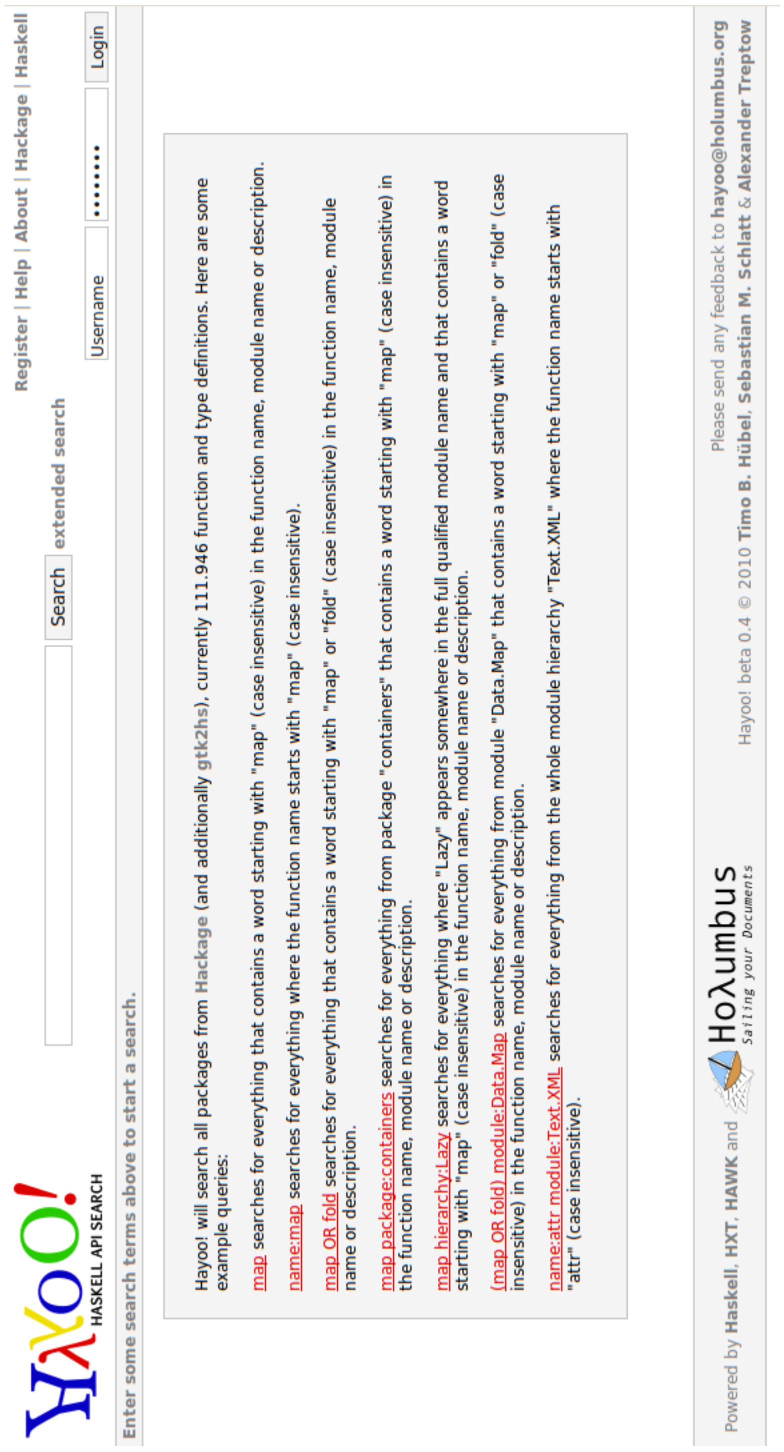
```
{
  "message":"Found 12 results and 17 completions.",
  "hits":12,
  "functions":[ {
    "name:"map",
    "uri":"http://hackage.haskell.org/...",
    "module":"Data.Map",
    "signature":"(a->b)->[a]->[b]",
    "package":"containers"
  }, ... ],
  "completions":[ {
    "word":"MapM",
    "count":11
  }, ... ],
  "modules":[ {
    "name":"Data",
    "count":19
  }, ... ],
  "packages":[ {
    "name":"containers",
    "count":13
  }, ... ]
}
```

5.6. Templates

As might be expected, all pages of Hayoo have the same layout, header and footer. So this section will only give a representative example how templates are used in Hayoo.

For this example the template for showing search results is given. This includes the files `/template/Index/search.xhtml` `/template/Layouts/header.xhtml` and `/template/Layouts/pagefooter.xhtml`.

In listing B.1 `header.xhtml` is displayed. It contains most of the page structure. The HTML header, the page header, binding the parameters “settings”, “querytext” and “login”. The page body with bindings for “mystatus”, flash message bindings and “content” binding. At last `header.xhtml` embeds another template shown in listing B.2. This `pagefooter.xhtml` has no bindings for Hayoo’s application specific content. The `search.xhtml` shown in listing B.3 has a `hawk:surround` tag, that means it is surrounded by another template at a specific position. In this case `search.xhtml` replaces the “content” binding of `header.xhtml` with four more bindings in the resulting merged template: “toppm”, “cloud”, “list” and “pages”, which are related to the `IndexView`.


HAYOO! HASKELL API SEARCH
Register | Help | About | Hackage | Haskell
Search
extended search
Username
Login

Enter some search terms above to start a search.

Hayoo! will search all packages from **Hackage** (and additionally **gtk2hs**), currently 111.946 function and type definitions. Here are some example queries:

map searches for everything that contains a word starting with "map" (case insensitive) in the function name, module name or description.

name:map searches for everything where the function name starts with "map" (case insensitive).

map OR fold searches for everything that contains a word starting with "map" or "fold" (case insensitive) in the function name, module name or description.

map package:containers searches for everything from package "containers" that contains a word starting with "map" (case insensitive) in the function name, module name or description.

map hierarchy:Lazy searches for everything where "Lazy" appears somewhere in the full qualified module name and that contains a word starting with "map" (case insensitive) in the function name, module name or description.

(map OR fold) module:Data.Map searches for everything from module "Data.Map" that contains a word starting with "map" or "fold" (case insensitive) in the function name, module name or description.

name:attr module:Text.XML searches for everything from the whole module hierarchy "Text.XML" where the function name starts with "attr" (case insensitive).

Powered by **Haskell**, **HXT**, **HAWK** and Hoλumbus *Sailing your Documents*

Please send any feedback to **hayoo@holumbus.org**

Hayoo! beta 0.4 © 2010 **Timo B. Hübel**, **Sebastian M. Schlatt** & **Alexander Treptow**

Figure 5.4.: New web interface of the *Hayoo! API Search*

6 Reusing Holumbus Wrapper

The Holumbus wrapper used in Hayoo was developed to provide an overall architecture for wrapping Holumbus search engine functionality, to separate it from the rest of a web application written with Hawk. This results in a loose coupling between Holumbus and the specific web application. Loose coupling of applications and frameworks is essential, because if there are minor changes in the framework then there have to be major repercussions in the application, if tightly coupled [GHJV94, p. 27].

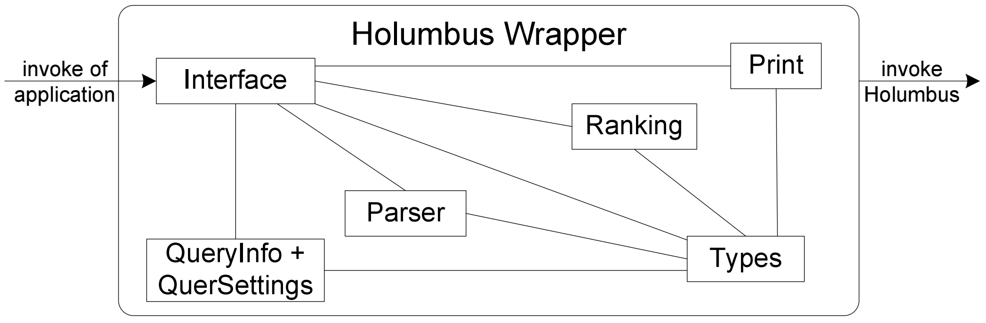


Figure 6.1.: The Holumbus-Wrapper architecture

As visible in figure 6.1 there are three functional areas the wrapper is divided into. *Types* define the basic data structures, *QueryInfo + QuerySettings* is used to create query information data types, *Parser* parses a query string, *Ranking* ranks a query result and *Print* prepares the query results for transmission in exchangeable data formats like plain text, XHTML or JSON.

This chapter describes how the developed Holumbus wrapper architecture can be reused for another specialized search engine written with Holumbus. It is recommended to use Hawk to write the web application part, but also any other framework can be used. The reuse instructions assume that there are no structural changes or type name changes for reuse. Everyone is free to also change these, but this may causes some other changes in wrapper. In addition to that it is assumed that the reader is familiar with the Holumbus wrapper or has read section 5.1.

For the `HolWrapper` interface module there are no changes needed. The subsequent section 6.1 describes what changes a developer need to make in the `Types` module, section 6.2 does that for the query modules and the last section 6.3 of this chapter describes what to change in the `Print` module for reuse.

6.1. Types

It is recommended to keep the basic data type structure, to restrict the needed changes to a minimum. The needed changes can be split into the four areas:

1. FunctionInfo,
2. QuerySettings,
3. AppConfiguration and
4. Index and Documents file loading

FunctionInfo

The `FunctionInfo` data type is the custom parameter in the document data type of Holumbus. Therefore an adopter needs to change this data types parameters to the ones used in his search engine. This data type may only be reused for another specialized API search engine, for a programming language which has nearly the same structure as Haskell.

The changes in this data type causes changes in *Index and Documents file loading* as well as in the *Ranking* and *Print* modules.

QuerySettings

`QueryInfo` is recommended to stay as it is, also if an adopter just do not use the Holumbus cache. Then the database file will stay empty. `QuerySettings` can mainly be kept, too. At least the two parameters `searchString` and `offset` should stay in this data type and `processConfig` is also recommended, because each Holumbus query will need that configuration. If it is generated on building `QuerySettings` or at the `processQuery` call does not make any difference. The other three data types were Hayoo specific so here an adopter can set its own additional configuration parameters. Clearly the `RConfig` data type needs to be removed.

```
data QuerySettings = QuerySettings
        { searchString  :: String
        , offset        :: Int
        , processConfig :: ProcessConfig
        ...
        }
```

AppConfiguration

The `AppConfiguration` instance normally will be implemented in the `Config.Config` module. For another web application that needs additional global and load on startup data this instance declaration has to move to this module and just use the creator function that here is named `loadHayooConfig`. Otherwise only the paths for index, documents and cache file will need to be adjusted.

An example for another `AppConfiguration` instance outside of the Holumbus wrapper is the following.

```
getInstance = do
    wrapper <- loadWrapperConfig
    mydata  <- loadMyData
    return $ MyAppConfig
```

```
            { myappdata1      = func1
            , myappdata2      = mydata
            , mysearchwrapper = wrapper  }
```

loadWrapperConfig is the equivalent to loadHayooConfig. loadMyData is another function for loading some other files and func1 just puts non-monadic data to the application wide data type.

Index and Documents file loading

Both of the instances of XmlPickler and Binary will need to change completely. So that the loadFromFile method can be used to load index and documents files in binary or XML format.

6.2. Query

Query involves the five modules Parser, QueryInfo, QuerySettings, Ranking and Common. The parser processes the search string and converts it to a well-defined format that the Holumbus framework understands. QueryInfo and QuerySettings supply the data handles for searching as well as the configuration parameters and the search string itself. Ranking is done after the results have been retrieved. The Common module offers some additional functions, e.g. parsing of single form data types like floating point values or booleans.

Common

The Common module only needs to be adjusted if users of the search application are able to customize it. Then the function settingsElems needs to be changed. It is also recommended to remove the toRConfig function, because in a new search application this data type will not exist.

In addition to this only the two methods defaultProcessConfig and default-FuzzyConfig may be changed by an adopter, if he wants another default query configuration. How these can be changed and what their parameters configure can be looked up in the API documentation of the Holumbus search engine [Hüb09].

Parser

As the name says only the `customParser` method needs to be changed if it is used. If the default parser is used instead, there will be no changes in this module. The Holumbus `parseQuery` method unfortunately leaks fuzzy search functionality so in most cases it is the better way to implement a new custom parsing method.

QueryInfo

This module only constructs a `QueryInfo` type by querying the application state and the `mkQuerySettings` method. So if there are no changes in the data type only one line of code may have to be customized.

```
q <- lookupParam "q"
```

This line reads the search string from the client request. If the search string parameter has another naming it has to be changed here.

QuerySettings

As in the `QueryInfo` module the parameter name of the offset read from the client request may be adjusted. When users can not customize the search, just change `mkQuerySettings` so that it generates a `QuerySettings` type by default and delete all other methods in this module, otherwise the cascading three functions `querySettingsByRequest`, `querySettingsBySession` `querySettingsByDatabase` need to change.

`querySettingsByRequest` may not need an parameter indicating whether there are settings for a single request or not as well as the `settingElems` from the `Common` module need to change. `settingElems` also need to be adjusted for `querySettings-BySession` to query the right keys from session data. `querySettingsByDatabase` needs another method for getting the configuration data and converting it to a `Query-Setting` data type. All these functions use `listToQuerySettings` which converts a list of strings to a `QuerySettings` type by parsing the parameters. This function need to be adjusted or removed, because it at least uses the `RConfig` type.

Ranking

This module exports three functions. `emptyRanking` and `defaultRanking` can stay as they are. Only `customRanking` needs to be changed, because it uses the `QuerySettings` parameters `modules` and `packages` of type `RConfig`.

6.3. Print

As mentioned in previous chapters, the print module of the Holumbus wrapper is tightly coupled to the application. So all functions here need to be replaced. Maybe some lines of code can be reused, but this depends on the web application and the specific search engine. Print functions nearly can return every data type not only XML, JSON and simple data types. Also they can generate custom types for example to store parts of a search request to the application database.

7 Conclusion

To summarize this thesis, this chapter includes a review of the achieved results where they are compared to the original requirements and further works. Subsequently an overview of possible future work is presented, which is based on experiences made during this work and inspirations found in other web frameworks as well as gained from discussions. Finally a short outlook to the near future of Hawk and Hayoo is given.

7.1. Summary

This thesis presented some fundamentals for improving web application interactivity, authentication and individualization, especially for search engines. The original goals were reached. So the*Hayoo! API Search* has been reimplemented using the web application framework Hawk and extended by customized search functionality. Furthermore the web application and search engine part of Hayoo were uncoupled to easily exchange each part.

For implementing Hayoo using Hawk, the web framework has been extended by authentication, a web application specific data type and JSON support. It was shown in this work, that the Hawk framework is a qualified full functional, usable framework for creating web applications.

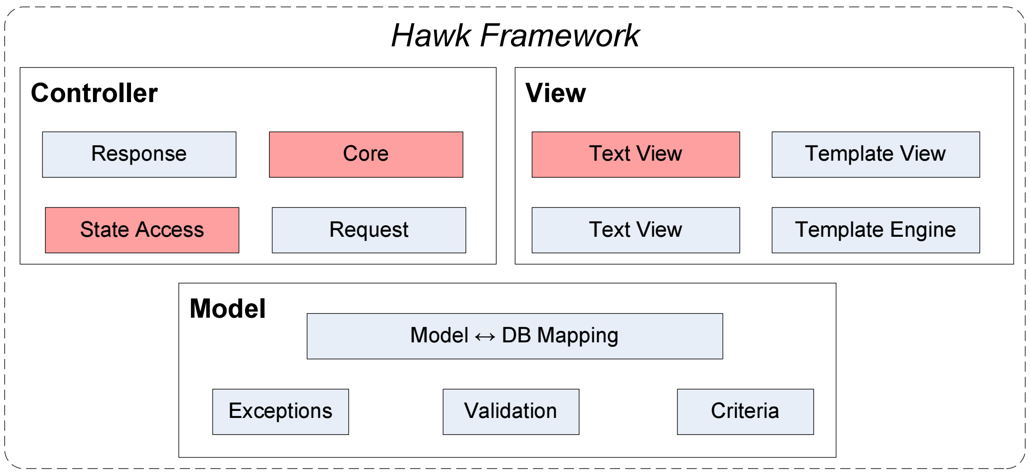


Figure 7.1.: Changes to the Hawk framework

Figure 7.1 shows what parts of the Hawk framework were affected by this work.

In comparison to other web frameworks available nowadays, Hawk is in its early stages. Basic functionality is supported by the current version of Hawk, but it is not suited for huge complex web applications, because it leaks caching support for example. In the subsequent chapter possible and reasonable improvements are discussed in detail. One major advantage of Hawk is that is it based on the strictly typed pure functional programming language Haskell. Therefore there are nearly no run time errors in a web application created with Hawk. Another advantage is that Hawk uses much less memory and time resources than most web frameworks that use scripting languages, because it does not need to be interpreted, is much smaller in code and so there is much less overhead.

Hopefully the *Hayoo! API Search* now has an additional benefit in comparison to the prior implementation, by its extended functionality. Compared to other search engines Hayoo is a highly specialized search engine that has a well defined user group. So an individualization is much easier than in huge common search engines, that have to search highly unstructured data.

7.2. Future Work

As mentioned in the prior section Hawk has proven to be useful, but it is still in its early stages of development. During extending the framework some possible further improvements came to mind. All in the thesis *The Hawk Framework* mentioned extensions are still up to date, out of the expansion of Hawk to support JSON [PR09, p. 109 ff]. The following paragraphs describe some improvements, which are not present but might be useful to be included in the future.

7.2.1. Controller

Mail Support

The business model of much web applications today is partly based on sending mails to users, to sell these mail addresses to advertising companies or just to send them newsletters. At least nearly each web application that has user accounts and maybe user generated content will need mailing functionality to support registration confirmation mail. So that it is not possible to generates thousands of accounts automatically. Another application of mailing is the upwards mentioned sending of newsletters, changes in terms and conditions or something similar.

Caching

Huge web applications are often under high database load, if they do not use caching mechanisms to hold pages, actions, fragments, files or session data in main memory of a web server. Caching decreases the load and response time of web applications enormously. No big, highly used web application today works without caching.

Additional Authentication Methods

As described in section 2.3 there are some more authentication methods that are not currently supported by Hawk, for example OpenID and HTTP Digest authentication. Another improvement on the authentication part of Hawk is to uncouple the data retrieval from the authentication methods. So that each method can for example use a file or database.

7.2.2. View

Localization / Internationalization

Many nowadays web services are personalized. Localization (L10N) and Internationalization (I18N) customizes web applications regarding to language, text, formatting and cultural character of the user. In the application a handler is created for a special region on client request and this handler is used to generate customized data by keys, e.g. a text will be displayed in the users preferred language, a date will be formatted differently for American or German users and for an Arabic page view another CSS file will be returned.

7.2.3. Hayoo

Extension of Hayoo's web interface functionality does not seem to be reasonable at the moment, because all valuable requirements were achieved.

7.3. Outlook

Further steps for Hawk and Hayoo are planed in the nearer future. The framework web page will be available online and Hawk sources will be available on hackage, to get some further impressions and ideas from the Haskell community. In addition the improvements of the previous section will be integrated to Hawk subsequently.

A

Creating Web Applications

Creating web applications with Hawk is really simple. This appendix describes how to structure and build a web application using Hawk. It is recommended to use Linux for developing and compiling Hawk applications, because of the library dependencies even though the *Glasgow Haskell Compiler* (GHC) is cross-platform. GHC needs *curl* and *sqlite3* header files to compile the Hawk framework. If GHC 6.10 or higher is already installed on the system as well as a current version of Hawk was downloaded and unpacked previously, the following commands can be used to install Hawk from the directory containing the files `Setup.hs` respectively `Hawk.cabal`.

```
> runhaskell Setup.hs configure
> runhaskell Setup.hs build
> runhaskell Setup.hs install
```

It is much easier to use the cabal package manager if `cabal install` is available, then only the following line needs to be achieved.

```
> cabal install
```

Alternatively a current version of Hawk can be downloaded from the hackage repository using `cabal install`.

```
> cabal install hawk
```

It is recommended to use cabal for installing Hawk, because it automatically resolves package dependencies and installs missing packages.

An example application and how to compile and start Hawk web applications is depicted in section A.2 of this appendix. Section A.1 presents a shell script for creating the recommended basic structure for a web application and adding some common data.

A.1. Basic Web Application Structure

The further described shell script creates a minimal web application with a custom name, an optional custom directory, the basic folder structure and has a welcome page with no content.

To run `make-project.sh` do

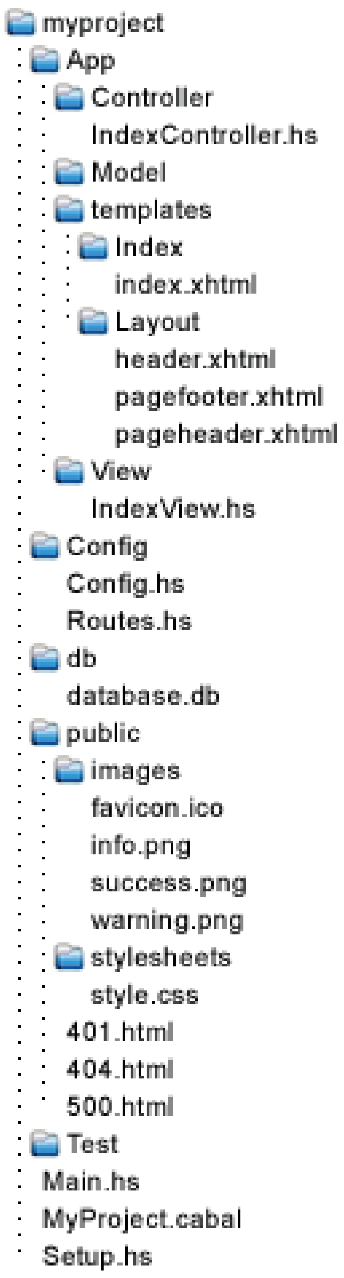
```
./make-project.sh myproject MyProject "Hawk/WebPage1"
```

whereas the usage is

```
./make-project.sh <projectname> <Projectname> [target_dir]
```

- `<projectname>` the project name in lower case letters
- `<Projectname>` the project name in correct notation, often with a beginning capital letter.
- `target_dir` the target directory for the web application, relative to the home folder of the current user.

`make-project.sh` creates folders and files shown in figure A.1. `App` contains application data, i.e. directories for controllers, models, views and templates. `Config` includes configuration and routing data, `db` holds all database covering files, i.e. database and SQL files and `public` contains data directly accessible from the web, like HTML files, images, CSS and JavaScript files. Files for unit tests should be put into the `test` folder, the files `myproject.cabal` and `Setup.hs` are for compiling the web application and `Main.hs` is the application entrance point.



```
myproject
  App
    Controller
      IndexController.hs
    Model
    templates
      Index
        index.xhtml
      Layout
        header.xhtml
        pagefooter.xhtml
        pageheader.xhtml
    View
      IndexView.hs
  Config
    Config.hs
    Routes.hs
  db
    database.db
  public
    images
      favicon.ico
      info.png
      success.png
      warning.png
    stylesheets
      style.css
    401.html
    404.html
    500.html
  Test
  Main.hs
  MyProject.cabal
  Setup.hs
```

Figure A.1.: Basic folder structure and files

To start this first simple web application just install it with cabal from the projects root directory (`$HOME/Hawk/WebPage1/`)

```
> cabal install
```

and start it with this call

```
> ./$HOME/.cabal/bin/myproject
```

Normally all applications installed with cabal are in the `$HOME/.cabal/bin/` directory.

Alternatively GHC's Haskell interpreter `ghci` can be used. Most notably this makes sense during development. Therefore `Main.hs` needs to be run with `ghci`.

```
> ghci Main.hs
```

If it has interpreted the code, the application can be started by running the `main` method.

```
Main> main
```

Due to development the application runs on port `3000`. So for starting its welcome page call `http://localhost:3000/` in a browser. This will show an empty page. Via another URL, for example `http://localhost:3000/foo/`, the web application will try to start the `foo` controller which does not exists, so a 404 error page will be displayed.

A.2. Sample Application: Guestbook

This sections explains how a simple guestbook web application can be implemented. Each user is allowed to create or delete entries, which are saved to a database. The guestbook application is constructed to get started with developing web applications using Hawk and it describes basic functionalities of the framework. Another simple web application with Hawk was shown in *The Hawk Framework* [PR09, App. A]. The functionality of the guestbook application will be restricted to

- show,
- add and
- delete entries.

Though the web application will have one model, view and controller module as well as three XHTML templates. The subsequent paragraphs describe how these modules are implemented and brought together.

Model

The guestbook's model has to provide four dates per entry:

- unique ID
- author name
- content
- creation date and time

Listing A.1 shows the `GuestbookEntry` model with data type and the different model class instances.

```
module App.Model.GuestbookEntry (GuestbookEntry (..)) where

import Hawk.Model
import Control.Monad.Trans (liftIO)
import Data.Time (UTCTime, getCurrentTime)

data GuestbookEntry = GuestbookEntry
  { _id          :: PrimaryKey
  , name         :: String
  , message      :: String
  , createdAt    :: UTCTime
  } deriving (Eq, Read, Show)

instance Persistent GuestbookEntry where
  persistentType _ = "GuestbookEntry"
  fromSqlList (l0:l1:l2:l3:[])
    = GuestbookEntry (fromSql l0) (fromSql l1)
                     (fromSql l2) (fromSql l3)
  fromSqlList _ = error "wrong list length"
  toSqlAL x = [ ("_id"       , toSql $ _id       x)
              , ("name"      , toSql $ name      x)
              , ("message"   , toSql $ message   x)
              , ("createdAt" , toSql $ createdAt x)
              ]
  tableName = const "guestbook"

instance WithPrimaryKey GuestbookEntry where
```

```
  primaryKey = _id
  pkColumn = head . tableColumns
  setPrimaryKey pk ge = ge {_id = pk}

instance Model GuestbookEntry where
  new = do
    t <- liftIO getCurrentTime
    return $ GuestbookEntry 0 "" "" t
  insert ge = do
    now <- liftIO getCurrentTime
    insertInTransaction $ ge { createdAt = now }

instance Validatable GuestbookEntry where
  validator ge = do
    validateNotNull "name"           $ name    ge
    validateLength  5 400 "message" $ message ge
    return ()

instance Updateable GuestbookEntry where
  updater ge s = do
    n <- updater (name    ge) $ subParam s "name"
    m <- updater (message ge) $ subParam s "message"
    return $ ge { name = n, message = m }
```

Listing A.1: Model module for guestbook entries

The `Persistent` instance converts the `GuestbookEntry` data type from and to SQL data and holds the corresponding table name for setting entries. `WithPrimaryKey` sets links between `GuestbookEntry`'s `PrimaryKey` parameter and the primary key in the database table. The `Model` instance supports CRUD functionality. Often only the `new` method is implemented. `GuestbookEntry` also needs a custom implementation of the `insert` method to determine the creation date and time. `Validatable` is used for verifying entries before updating an entry. In this case `name` have to be not empty and the content length have to be between five and 400 characters. At last the `Updateable` class is used for updating parameters of a `GuestbookEntry` date. The creation date and primary key can not be updated, because they are not defined by the user. The primary key is an auto incrementing SQL field and the date is generated on calling the `insert` method for making a `GuestbookEntry` persistent.

Actions

Actions in the `GuestbookController` module only support showing, creating and deleting of entries. Listing A.2 shows the `GuestbookController` module.

```
module App.Controller.GuestbookController (routes) where

import App.Model.GuestbookEntry
import qualified App.View.GuestbookView as GV

import Hawk.Controller
import Hawk.Model
import Hawk.View.TemplateView

routes :: [Routing]
routes =
  [ ("show",indexAction >>=
            render (typedView "show" GV.showXhtml))
  , ("index",indexAction >>=
             render (typedView "show" GV.showXhtml))
  , ("insert",insertAction >>
              redirectToAction "guestbook" "index")
  , ("delete",deleteAction >>
              redirectToAction "guestbook" "index") ]

indexAction :: StateController [GuestbookEntry]
indexAction = select (setOrder [desc "_id"] newCriteria)

insertAction :: StateController ()
insertAction = do
  method <- getRequestMethod
  case method of
    POST -> do
      n <- new :: StateController GuestbookEntry
      (ge, errs) <- getParams >>= updateAndValidate n ""
      if null errs then do
        insert ge
        setFlash "notice" "Successfully added your message!"
        else
          setErrors "guestbookEntry" errs
      return ()
    _ -> return ()

deleteAction :: StateController ()
deleteAction = do
  guestbookEntry <- readParam "id" >>= liftMaybe findMaybe
      :: StateController (Maybe GuestbookEntry)
  case guestbookEntry of
    Nothing ->
      setFlash "error" "The requested customer does not exist"
    Just ge -> do
      delete ge
      setFlash "notice" "The guestbook entry has been deleted"
  return ()
```

Listing A.2: Controller for requests to the guestbook application

As visible from the routing list, there is only one action that results in an output. The `indexAction` only reads all data from database in descending order by `_id` and passes them to the `showXhtml` method in the view. `insertAction` and `deleteAction` redirect to the `indexAction` after they have proceeded. `insertAction` figures out the clients request method, creates a new empty `GuestbookEntry` and tries to update this date with the given request parameters. If there was no error, the `insert` function is called to write the entry to DB, else an error message is printed. `deleteAction` reads the request parameter "id" and tries to load the `GuestbookEntry` with this ID. If an entry was found it will be deleted permanently using the `delete` method of the `Model` class. Otherwise an error will be displayed.

Templates

The initially mentioned three XHTML templates are `header.xhtml`, `show.xhtml` and `singleentry.xhtml`. Listing A.3 shows the framing XHTML template with the HTML header, flash message elements and an `hawk:bind` tag for binding content at this position. The binding reference name is called "content".

```
<?xml version="1.0" encoding="utf-8" ?>
<html>
<head>
  <title><hawk:bind name="title"/></title>
  <link href="/stylesheets/style.css" media="screen"
        rel="Stylesheet" type="text/css"/>
</head>
<body>
  <!-- error message -->
  <hawk:message type="error">
    <div class="error">
    <image alt="Warning" title="Warning"
           src="/images/warning.png"/>
    <hawk:content/>
    </div>
  </hawk:message>
  <!-- info message -->
  <hawk:message type="notice">
    <div class="notice">
    <image alt="info" title="Info" src="/images/info.png"/>
    <hawk:content/>
    </div>
  </hawk:message>
  <!-- content -->
  <hawk:bind name="content"/>
</body>
</html>
```

Listing A.3: Page frameing template for the guestbook application

In listing A.4 the template for displaying the input form is shown. It uses two other `hawk` tags. `hawk:surround` surrounds the containing code with another template given as "with" parameter at binding position "at". So `show.xhtml` is included in `header.xhtml`. `hawk:embed` includes another template at a defined position. Here it is the `singleEntry.xhtml` template.

```
<?xml version="1.0" encoding="utf-8" ?>
<hawk:surround with="../Layouts/header.xhtml"
  at="content" xmlns:hawk="http://fh-wedel.de/hawk">
  <form method="POST" action="/guestbook/insert">
    <p>
      <label for="name">Name</label><br/>
      <input type="text" name="name"/>
    </p>
    <p>
      <label for="name">Message</label><br/>
      <textarea cols="50" rows="5" name="message"/>
    </p>
    <input type="Submit" value="Post"/>
  </form>
  <p>Guestbook Entries:
     <hawk:bind name="posts" type="Int"/></p>
  <hawk:embed what="singleEntry.xhtml"/>
</hawk:surround>
```

Listing A.4: Template for showing the input form and counter

`singleEntry.xhtml` formats a single entry. The template is visible in listing A.5 and just has different bindings for the parts of an entry.

```
<div class="entry">
  <div class="head">
    <span class="l">
      <hawk:bind name="username" type="String"/>
      <hawk:bind name="delete"/>
    </span>
    <span class="r">
      <hawk:bind name="at" type="UTCTime" format="formatTi"/>
    </span>
  </div>
  <div class="body">
    <hawk:bind name="msg" type="String"/>
  </div>
</div>
```

Listing A.5: Guestbook template for showing a single entry

View

The view module support three methods and creates two data type with the `viewDataType` *Template Haskell* method. The `showXhtml` method gets a list of `GuestbookEntry`'s from the controller, applies `entryXhtml` to each `GuestbookEntry` and sets the resulting list of `GuestbookSingleEntry`'s to its parameter in the `GuestbookShow` type. The `entryXhtml` method just fills the parameters of a `GuestbookEntry` into the template. `formatTi` is the last method of the `GuestbookView`. It is bound to the `GuestbookSingleEntry` by `viewDataType` and so every value that "at" is set to, will be formated using this method. The binding is defined by the "format" parameter in listing A.5.

```
{-# LANGUAGE TemplateHaskell #-}
module App.View.GuestbookView (showXhtml) where

import App.Model.GuestbookEntry

import Hawk.Controller
import Hawk.View.TemplateView
import Hawk.View.Template.DataType
import qualified Hawk.View.Template.HtmlHelper as H

import Data.Time
import System.Locale

formatTi :: UTCTime -> [XmlTree]
formatTi t = [H.text (formatTime defaultTimeLocale "%R %D" t)]

$(viewDataType "Guestbook" "singleEntry")
$(viewDataType "Guestbook" "show")
```

```
showXhtml :: [GuestbookEntry] -> StateController GuestbookShow
showXhtml gs = do
  ge <- mapM entryXhtml gs
  return GuestbookShow
    { guestbookSingleEntry = ge
    , title = [H.text "My first Guestbook"]
    , posts = length gs
    }

entryXhtml :: GuestbookEntry -> StateController
   GuestbookSingleEntry
entryXhtml ge = return GuestbookSingleEntry
    { username = name ge
    , at = createdAt ge
    , msg = message ge
    , delete = [H.textlink
                ("/guestbook/delete?id=" ++ show (_id ge))
                "[Delete]"]
    }
```

Listing A.6: View module of the guestbook application

Configuration

Configuring the guestbook web application is not a huge task. Though there only exists one controller, the `routing` function only has one line.

```
routing = M.singleton "guestbook" $ M.fromList GC.routes
```

The `Config` module creates environment settings, e.g. database handler and logging levels, as well as it sets application configuration, such as session handling, routing, root directories and paths to error files.

```
environment :: AppEnvironment
environment = AppEnvironment
  { connectToDB = ConnWrapper `liftM`
                  connectSqlite3 "./db/database.db"
  , logLevels   = [(rootLoggerName, DEBUG)]
  , envOptions  = []
  }

configuration :: BasicConfiguration
configuration = BasicConfiguration
  { sessionStore = cookieStore
  , sessionOpts  = [("secret", "
     01234567890123456789012345678901")]
  , authType     = emptyAuth
  , authOpts     = []
  , routing      = simpleRouting Routes.routing
  , templatePath = "./App/template"
  , publicDir    = "./public"
  , confOptions  = []
  , error401file = "401.html"
  , error404file = "404.html"
  , error500file = "500.html"
  }
```

Now all application data, templates and configuration is written, the guestbook web application can be run. Listing A.7 shows how the configuration methods are passed to the application constructing function `getApplication`. `getInstance` is an application defined data type, that is left empty here, so the standard implementation can be used. For testing purpose the port was set to `3000`.

```
module Main (main) where

import Config.Config
import Hawk.Controller.Initializer
import Hawk.Controller.Types (AppConfiguration (..))
import Hack.Handler.SimpleServer as Server (run)

main :: IO ()
main = run 3000 $ getApplication getInstance environment
   configuration
```

Listing A.7: Instantiating point for the web application

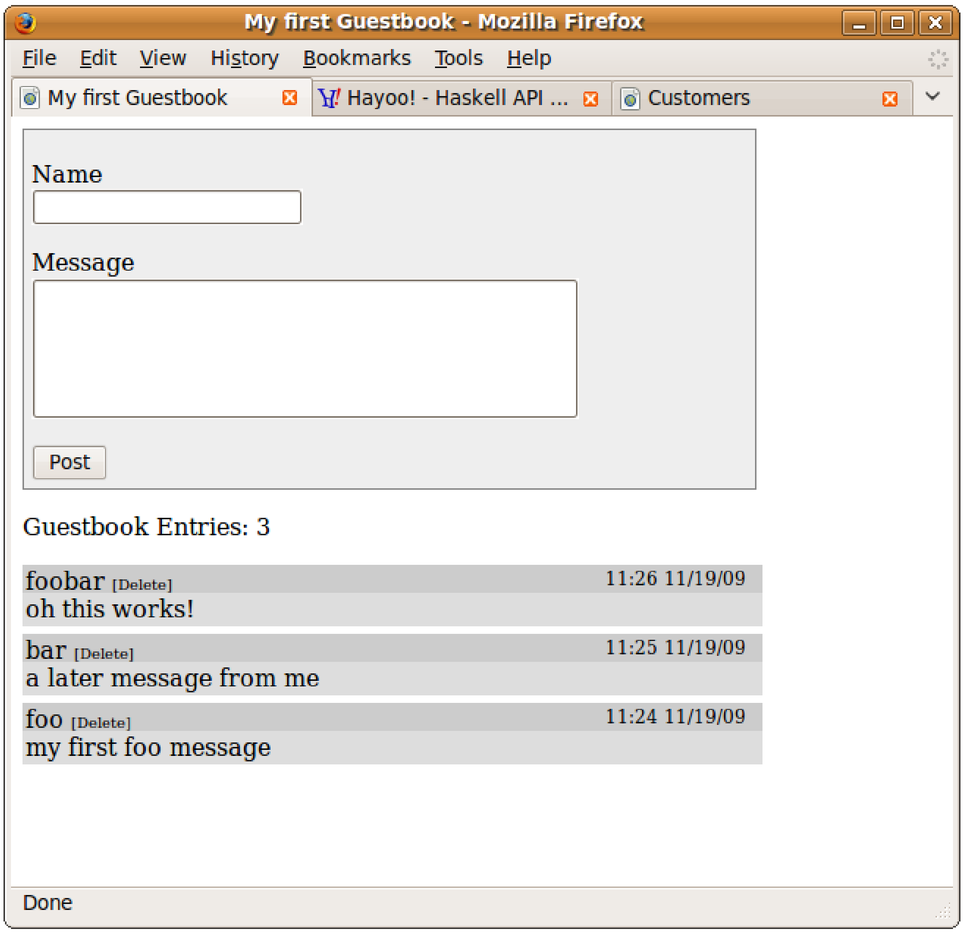
Based on the routing configuration the application can be viewed on local host calling `http://localhost:3000/guestbook/`. But first the application needs to be started. To do that use the following commands:

```
> ghci Main.hs
```

wait till the prompt is back and call the `main` method

```
Main> main
```

It now should look like displayed in figure A.2


My first Guestbook - Mozilla Firefox
File Edit View History Bookmarks Tools Help
My first Guestbook
Hayoo! - Haskell API ...
Customers
Name
Message
Post
Done

Guestbook Entries: 3

foobar [Delete] 11:26 11/19/09
oh this works!

bar [Delete] 11:25 11/19/09
a later message from me

foo [Delete] 11:24 11/19/09
my first foo message

Figure A.2.: Guestbook interface with some entries

B
Hayoo Template Files

```
<?xml version="1.0" encoding="utf-8" ?>
<html>
<head>
  <title><hawk:bind name="title"/></title>
  <meta http-equiv="content-type"
        content="text/html; charset=UTF-8"/>
  <meta content="nofollow" name="robots"/>
  <meta name="keywords" content="Haskell, API, Search, Hackage,
      Functions, Types, Packages"/>
  <meta name="description" lang="en" content="A Haskell API
     search engine with find-as-you-type and suggestions.
     Searches for function and type defintions in all Haskell
     packages from Hackage."/>
  <link href="/images/favicon.ico" type="image/ico"
        rel="Shortcut icon"/>
  <link href="/stylesheets/hayoo.css" media="screen"
        rel="Stylesheet" type="text/css"/>
  <script src="/javascript/prototype.js"
          type="text/javascript"/>
  <script src="/javascript/hayoo.js" type="text/javascript"/>
</head>
<body>
```

```
<div id="container">
  <!-- page header -->
  <div id="info">
    <hawk:bind name="settings"/>
    <a href="/index/help">Help</a> |
    <a href="/index/about">About</a> |
    <a href="http://hackage.haskell.org">Hackage</a> |
    <a href="http://www.haskell.org">Haskell</a>
  </div>
  <div id="query">
    <div id="logo">
      <a href="/index/index">
        <img class="logo" src="/images/hayoo.png"
                     alt="Hayoo logo"/>
      </a>
    </div>
    <form id="queryform" method="get" action="/index/search"
              onsubmit="return submitQuery()">
      <div id="queryinterface">
        <hawk:bind name="querytext"/>
        <input class="formbutton" id="querybutton"
                       type="submit" value="Search"/>
        <img id="throbber" src="/images/loader.gif"
                     alt="Throbber" style="display:none;"/>
        <a href="/index/config">extended search</a>
      </div>
    </form>
  </div>
  <hawk:bind name="login"/>
  <!-- page body -->
  <div id="result">
    <div id="status"><hawk:bind name="mystatus"/></div>
    <!-- error message -->
    <hawk:message type="error">
      <div class="error">
        <image class="message" alt="Warning"
               title="Warning" src="/images/warning.png"/>
        <hawk:content/>
      </div>
    </hawk:message>
    <!-- info message -->
    <hawk:message type="notice">
      <div class="notice">
        <image class="message" alt="info"
               title="Info" src="/images/info.png"/>
        <hawk:content/>
      </div>
    </hawk:message>
    <!-- success message -->
```

```
      <hawk:message type="success">
        <div class="success">
          <image class="message" alt="success" title="Success"
                 src="/images/success.png"/>
          <hawk:content/>
        </div>
      </hawk:message>
      <!-- content -->
      <hawk:bind name="content"/>
    </div>
    <!-- page footer -->
    <hawk:embed what="../Layouts/pagefooter.xhtml"/>
  </div>
</body>
</html>
```

Listing B.1: Hayoo's framing template with page header

```
<div id="credits">
  <div id="powered">
    <div id="libs">
      Powered by <a class="credits"
          href="http://www.haskell.org">Haskell</a>,
      <a class="credits"
          href="http://www.fh-wedel.de/~si/HXmlToolbox/">
          HXT</a>,
      <a class="credits"
          href="" alt="HAskell Webapplication Kit">HAWK</a> and
    </div>
    <a href="http://holumbus.fh-wedel.de">
      <img class="logo" src="/images/holumbus.png"
             alt="Holumbus logo"/>
    </a>
  </div>
  <div id="authors">
    <div id="feedback">
      Please send any feedback to
          <a href="mailto:hayoo@holumbus.org">
          hayoo@holumbus.org</a>
    </div>
    <div id="copyright">
      Hayoo! beta 0.4 &copy; 2010
      <span class="author">Timo B. H&uuml;bel</span>,
      <span class="author">Sebastian M. Schlatt</span> &amp;
      <span class="author">Alexander Treptow</span>
    </div>
  </div>
</div>
```

Listing B.2: Footer of the Hayoo web page

```
<?xml version="1.0" encoding="utf-8" ?>
<hawk:surround with="../Layouts/header.xhtml"
  at="content" xmlns:hawk="http://fh-wedel.de/hawk">
  <div id="toppm">
    <hawk:bind name="toppm"/>
  </div>
  <div id="cloud">
    <hawk:bind name="cloud"/>
  </div>
  <div id="documents">
    <hawk:bind name="list"/>
  </div>
  <div id="pages">
    <hawk:bind name="pages"/>
  </div>
</hawk:surround>
```

Listing B.3: Template for displaying Hayoo search results

Bibliography

[BPM$^+$08] Bray, Tim ; Paoli, Jean ; Maler, Eve ; Yergeau, François ; Sperberg-McQueen, C. M.: Extensible Markup Language (XML) 1.0 (Fifth Edition) / W3C. 2008. – W3C Recommendation. – http://www.w3.org/TR/2008/REC-xml-20081126/

[BWNH$^+$03] Blake-Wilson, S. ; Nystrom, M. ; Hopwood, D. ; Mikkelsen, J. ; Wright, T.: *Transport Layer Security (TLS) Extensions*. RFC 3546 (Proposed Standard). http://www.ietf.org/rfc/rfc3546.txt. Version: June 2003 (Request for Comments). – Obsoleted by RFC 4366

[cBHL09] Çelik, Tantek ; Bos, Bert ; Hickson, Ian ; Lie, Håkon W.: Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification / W3C. 2009. – Candidate Recommendation. – http://www.w3.org/TR/2009/CR-CSS2-20090908

[Cha] Chakravarty, Manuel: *Type families*. http://www.haskell.org/haskellwiki/GHC/Type_families, last checked: 2009-10-01

[CSF$^+$08] Cooper, D. ; Santesson, S. ; Farrell, S. ; Boeyen, S. ; Housley, R. ; Polk, W.: *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 5280 (Proposed Standard). http://www.ietf.org/rfc/rfc5280.txt. Version: May 2008 (Request for Comments)

[Doj] *The Dojo Toolkit: Unbeatable JavaScript Tools*

[DR06] Dierks, T. ; Rescorla, E.: *The Transport Layer Security (TLS) Protocol Version 1.1*. RFC 4346 (Proposed Standard). http://www.ietf.org/rfc/rfc4346.txt. Version: April 2006 (Request for Comments). – Obsoleted by RFC 5246, updated by RFCs 4366, 4680, 4681

[Dus10] DUSEK, Jason: *JSONb: JSON parser that uses byte strings*. http://hackage.haskell.org/package/JSONb. Version: 2010, last checked: 2010-04-23

[FHBH+99] FRANKS, J. ; HALLAM-BAKER, P. ; HOSTETLER, J. ; LAWRENCE, S. ; LEACH, P. ; LUOTONEN, A. ; STEWART, L.: *HTTP Authentication: Basic and Digest Access Authentication*. RFC 2617 (Draft Standard). http://www.ietf.org/rfc/rfc2617.txt. Version: June 1999 (Request for Comments)

[Fou] FOUNDATION, The Apache S.: *Apache Tomcat*. http://tomcat.apache.org/, last checked: 2010-05-05

[Gar05] GARRET, Jesse J.: *Ajax: A New Approach to Web Applications*. Version: February 2005. http://www.adaptivepath.com/ideas/essays/archives/000385.php, last checked: 2010-03-16

[GHC] *The Glasgow Haskell Compiler*. http://haskell.org/ghc, last checked: 2009-09-26

[GHJV94] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns*. Addison Wesley, 1994

[GWT06] *Google Web Toolkit*. http://code.google.com/webtoolkit/. Version: May 2006, last checked: 2010-03-17

[hackage] *hackageDB*. http://hackage.haskell.org/, last checked: 2010-04-21

[Hüb08] HÜBEL, Timo B.: *The Holumbus Framework: Creating fast, flexible and highly customizable search engines with Haskell*. Version: April 2008. http://holumbus.fh-wedel.de/src/doc/thesis-searching.pdf

[Hüb09] HÜBEL, Timo B.: *Holumbus-Searchengine-0.0.8: A distributed search and indexing engine*. http://holumbus.fh-wedel.de/docs/Holumbus-Searchengine/. Version: 2009

[HH09] HYATT, David ; HICKSON, Ian: HTML 5 / W3C. 2009. – W3C Working Draft. – http://www.w3.org/TR/2009/WD-html5-20090825/

[Ht] HAPPSTACK TEAM, HAppS L.: *Happstack – Haskell application server stack*. http://happstack.com/, last checked: 2009-09-30

[IM07] Ishikawa, Masayasu ; McCarron, Shane: XHTML^{TM} 1.1 - Module-based XHTML - Second Edition / W3C. 2007. – W3C Working Draft. – http://www.w3.org/TR/2007/WD-xhtml11-20070216

[Int09] International, ECMA: *ECMAScript Lanugage Specification.* Version: December 2009. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf

[Jac08] Jackson, Mark: *The Future of Google's Search Personalization.* http://searchenginewatch.com/3631746. Version: November 2008, last checked: 2010-03-30

[Jol] Jolley, Charles: *What is a Cloud Application.* http://wiki.sproutcore.com/What-is-a-Cloud-Application, last checked: 2010-04-17

[Jos03] Josefsson, S.: *The Base16, Base32, and Base64 Data Encodings.* RFC 3548 (Informational). http://www.ietf.org/rfc/rfc3548.txt. Version: July 2003 (Request for Comments). – Obsoleted by RFC 4648

[JPJ99] Jones, Mark P. ; Peyton Jones, Simon: Lightweight extensible records for Haskell. In: *In Haskell Workshop*, 1999 http://research.microsoft.com/en-us/um/people/simonpj/Papers/recpro.ps.gz

[jQu06] *jQuery: The Write Less, Do More, JavaScript Library.* http://jquery.org/. Version: 2006, last checked: 2010-03-17

[JRH99] Jacobs, Ian ; Raggett, David ; Hors, Arnaud L.: HTML 4.01 Specification / W3C. 1999. – W3C Recommendation. – http://www.w3.org/TR/1999/REC-html401-19991224

[JsH06] *JsHttpRequest 5: cross-browser AJAX + file uploading.* http://en.dklab.ru/lib/JsHttpRequest/. Version: July 2006, last checked: 2010-03-17

[JSON] *JSON: JavaScript Object Notation.* http://www.json.org/, last checked: 2010-04-20

[Kes09] Kesteren, Anne van: XMLHttpRequest / W3C. 2009. – Last Call WD. – http://www.w3.org/TR/2009/WD-XMLHttpRequest-20091119/

[KL00] KHARE, R. ; LAWRENCE, S.: *Upgrading to TLS Within HTTP/1.1.* RFC 2817 (Proposed Standard). http://www.ietf.org/rfc/rfc2817.txt. Version: May 2000 (Request for Comments)

[KLS04] KISELYOV, Oleg ; LÄMMEL, Ralf ; SCHUPKE, Keean: *Strongly Typed Heterogeneous Collections.* August 2004

[Lac06] LACAVA, Alessandro: *Speed Up Your AJAX-based Apps with JSON.* Version: October 2006. http://www.devx.com/webdev/Article/32651, last checked: 2010-03-16

[Mah06] MAHEMOFF, Michael: *Ajax Design Patterns.* O'Reilly, 2006

[Mer06] MERRILL, Christopher L.: *Performance Impacts of AJAX Development.* Version: January 2006. http://www.webperformanceinc.com/library/reports/AjaxBandwidth/, last checked: 2010-03-16

[Mit04] MITCHELL, Neil: *Hoogle: Haskell API Search.* http://haskell.org/hoogle/. Version: 2004, last checked: 2010-03-24

[NCH+04] NICOL, Gavin ; CHAMPION, Mike ; HÉGARET, Philippe L. ; ROBIE, Jonathan ; WOOD, Lauren ; HORS, Arnaud L. ; BYRNE, Steve: Document Object Model (DOM) Level 3 Core Specification / W3C. 2004. – W3C Recommendation. – http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407

[Net10] NETCRAFT LTD.: *Web Server Survey.* Version: February 2010. http://news.netcraft.com/archives/2010/02/22/february_2010_web_server_survey.html, last checked: 2009-09-28

[PJ03] PEYTON JONES, Simon: *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, 2003

[PJVWS07] PEYTON JONES, Simon ; VYTINIOTIS, Dimitrios ; WEIRICH, Stephanie ; SHIELDS, Mark: *Practical type inference for arbitrary-rank types.* Version: July 2007. http://research.microsoft.com/en-us/um/people/simonpj/papers/higher-rank/, last checked: 2010-04-21

[PR09] PEEMÖLLER, Björn ; ROGGENSACK, Stefan: *The Hawk Framework: Creating pure functional web applications using Haskell.* October 2009 [pro06] *Prototype JavaScript Framework: Easy Ajax and DOM manipulation for dynamic web applications*. http://www.prototypejs.org/. Version: 2006, last checked: 2010-03-17

[Res00] RESCORLA, E.: *HTTP Over TLS*. RFC 2818 (Informational). http://www.ietf.org/rfc/rfc2818.txt. Version: May 2000 (Request for Comments)

[RF06] RECORDON, D. ; FITZPATRICK, B.: *OpenID Authentication 1.1*. Version: May 2006. http://openid.net/specs/openid-authentication-1_1.html, last checked: 2010-03-18

[Riv92] RIVEST, R.: *The MD5 Message-Digest Algorithm*. RFC 1321 (Informational). http://www.ietf.org/rfc/rfc1321.txt. Version: April 1992 (Request for Comments)

[RP09] ROGGENSACK, Stefan ; PEEMÖLLER, Björn: *FH-Wedel Latex Thesis Template*. http://fh-wedel.de. Version: 2009

[Ruby] *Ruby On Rails*. http://rubyonrails.org/, last checked: 2010-04-05

[Sch] SCHMIDT, Uwe: *Haskell XML Toolbox: A collection of tools for processing XML with Haskell*

[SPJ02] SHEARD, Tim ; PEYTON JONES, Simon: Template metaprogramming for Haskell. In: CHAKRAVARTY, Manuel M. T. (Hrsg.): *ACM SIGPLAN Haskell Workshop 02*, ACM Press, October 2002, pages 1–16

[Sproutcore] *Sproutcore*. http://www.sproutcore.com/, last checked: 2010-04-17

[Tog01] TOGNAZZINI, Bruce: *Maximizing Human Performance*. Version: 2001. http://www.asktog.com/basics/03Performance.html, last checked: 2010-03-16

[Uhl06a] http://darcs2.fh-wedel.de/repos/janus/

[Uhl06b] UHLIG, Christian: *A Dynamic Webserver with Servlet Functionality in Haskell representing all internal data by means of XML*. Version: October 2006. http://darcs2.fh-wedel.de/repos/janus/thesis/Janus_Thesis_081106.pdf

[Wan09] WANG, Jinjing: *hack: a Haskell Webserver Interface*. http://hackage.haskell.org/package/hack. Version: 2009, last checked: 2010-03-25

[yaw] *Yaws - Yet another webserver*. http://yaws.hyber.org/, last checked: 2010-05-05

[Zend] *Zend Framework*. http://framework.zend.com/, last checked: 2010-04-05

Affidavit

I hereby declare that this thesis has been written independently by me, solely based on the specified literature and resources. All ideas that have been adopted directly or indirectly from other works are denoted appropriately. This thesis has not been submitted to any other board of examiners in its present or a similar form and was not yet published in any other way.

Wedel, May 19th, 2010

(Alexander Treptow)