



UNIVERSITY OF APPLIED SCIENCES

DEPARTMENT OF COMPUTER SCIENCE

Master's Thesis

The Hawk Framework

Creating pure functional web applications using Haskell

Submitted on:

October 19th, 2009

Submitted by:

B. Sc. Björn Peemöller

Borgfelde 4a

22869 Schenefeld, Germany

Phone: +49 (40) 84 00 26 90

E-mail: bjoern.peemoeller@gmx.de

B. Sc. Stefan Roggensack

Unzerstraße 1–3

22767 Hamburg, Germany

Phone: +49 (40) 85 10 40 57

E-mail: roggensack@arcor.de

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

The Hawk Framework

Creating pure functional web applications using Haskell

Master's Thesis by Björn Peemöller and Stefan Roggensack

Web applications nowadays constitute a growing part of the World Wide Web. To facilitate the development of web applications, there is a variety of frameworks available, written in different programming languages. Haskell, a non-strict, purely functional programming language, offers both an expressive syntax and an advanced type system. Although these properties make Haskell a language considerable for web application programming, there is currently only few support. This work presents the design and implementation of a framework for developing web applications in Haskell. Based on the model-view-controller pattern, the framework not only provides a flexible controller concept, but also a fully-fledged template system for HTML generation. A database mapping completes these components towards a considerable framework for web application development.



Copyright © 2009 Björn Peemöller and Stefan Roggensack

This work is licensed under the Creative Commons Attribution-Noncommercial-Share Alike 3.0 Germany License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/de/> or send a letter to Creative Commons, 171 2nd Street, Suite 300, San Francisco, California, 94105, USA.

Layout done with the help of Timo B. Hübel's template, L^AT_EX 2_ε, KOMA-Script and B_IB_TE_X.

Contents

List of Figures	VII
List of Listings	IX
1. Introduction	1
1.1. Motivation	2
1.2. Scope	3
1.3. Related Work	3
1.4. Outline	4
1.5. Notice for Examination	5
2. Aspects of Haskell Programming	7
2.1. Type Families	8
2.2. Exception Handling	10
2.2.1. Basic Usage	11
2.2.2. The Exception Hierarchy	12
2.2.3. Extending the Exception Hierarchy	14
2.2.4. Lifted Exception Handling	16
2.3. Meta-programming using Template Haskell	17
2.3.1. Basic Principles	18
2.3.2. Internal Representation	19
2.3.3. The Quotation Monad	20
3. Architecture	23
3.1. Characterization of Web Applications	23
3.2. Application Architecture	25
3.2.1. Model Component	27
3.2.2. View Component	28
3.2.3. Controller Component	30
3.2.4. Controller/View Interaction	31
3.2.5. Web Server Integration	32
3.2.6. Final Architecture	32
4. Database Mapping	35
4.1. Concept	36

4.2. Database Management	39
4.2.1. Interface	39
4.2.2. Exceptions	41
4.3. SQL Construction	42
4.4. Data Type Mapping	45
4.4.1. Basic Mapping	45
4.4.2. Mapping with Identifiers	48
4.4.3. Model Interface	49
4.5. Relationship Mapping	50
4.6. Validation	53
4.7. Attribute Updates	55
4.8. Summary	57
5. Controller	59
5.1. Basic Types	59
5.1.1. Hack Interface	60
5.1.2. EnvController Monad	62
5.1.3. StateController Monad	63
5.1.4. Actions and Views	65
5.2. Action Implementation	67
5.2.1. Request Access	67
5.2.2. Redirects and Error Responses	68
5.2.3. Cookies	69
5.2.4. Session	70
5.2.5. Flash Messages and Errors	72
5.2.6. Action Composition	73
5.2.7. Routing	74
5.3. Front Controller	76
5.3.1. Request Handling	77
5.3.2. Configuration	78
5.3.3. Hack Integration	79
6. Template View	81
6.1. Templates	81
6.1.1. Binding Targets	82
6.1.2. Template Composition	84
6.1.3. State Access	86
6.1.4. Head Merge	87
6.2. TemplateView	88
6.2.1. Value Binding	90
6.2.2. Html Form Construction	91
6.2.3. Template Access	92
6.3. Templates as Algebraic Data Types	93

7. Project Diary Application	97
7.1. Introduction	97
7.2. Application Structure	99
7.3. Model Component	100
7.4. Actions	101
7.4.1. Modification of Areas	101
7.4.2. Authentication	103
7.5. Template Views	103
7.6. Summary	104
8. Conclusion	107
8.1. Summary	107
8.2. Assessment	108
8.3. Future Work	109
8.3.1. Database Mapping	110
8.3.2. Controller Component	112
8.3.3. View Component	112
8.3.4. Additional Features	113
8.4. Outlook	113
A. Source Code of the Customer Application	115
A.1. Model Implementation	116
A.2. Customer Actions	120
A.3. Customer Templates	121
A.4. Customer View	124
A.5. Configuration	126
B. Installation Guide	129
Bibliography	131
Affidavit	137

List of Figures

3.1. Schematic data flow for a web application	24
3.2. Model-view-controller pattern	26
3.3. Layers of the model component	28
3.4. Layers of the view component	29
3.5. Layers of the controller component	30
3.6. Principles for controller/view interaction	31
3.7. Architecture and data flow of a Hawk application	34
4.1. Exemplary database model	36
4.2. Layers of the database mapping	37
4.3. Simulation of hierarchic parameters	56
4.4. Type classes for database mapping	58
5.1. Types of the Hawk controller	60
5.2. Working principle of the EitherT monad	65
5.3. Components of the front controller	76
5.4. Server integration via the Hack web interface	80
6.1. Template combination via embedment	84
6.2. Surrounding a template	85
6.3. Template processing order	89
6.4. Helper functions for HTML form construction	92
7.1. Index page of the project diary	98
7.2. Directory structure of the project diary	99
7.3. Error messages while editing an area	102
7.4. Embedment hierarchy of the templates (extract)	104
7.5. Contribution of the embedded templates to the index page	105

List of Listings

2.1. Exception type class representing Haskell exceptions	12
2.2. MonadIO type class for lifting I/O actions	16
2.3. MonadCatchIO type class for lifting exception handling	17
4.1. Types for customer and category	36
4.2. MonadDB type class for monadic database access	40
4.3. Database access functions for MonadDB	40
4.4. Exceptions defined in the mapping component	41
4.5. Query type representing database queries	42
4.6. Criteria type for SQL construction	43
4.7. MonadDB functions for Criteria	44
4.8. Persistent type class for database table mapping	45
4.9. Example table mapping	46
4.10. Functions derived for Persistents	47
4.11. WithPrimaryKey type class for providing identifiers	48
4.12. Model type class providing a uniform interface	49
4.13. ForeignKeyRelationship type class for representing relationships	51
4.14. BelongsTo type class for relationship access of referenced types	52
4.15. ValidatorT monad transformer for validation	53
4.16. Selected predefined validation functions	54
4.17. Updateable class for enabling attribute updates	56
4.18. UpdaterT monad transformer for attribute updates	57
5.1. Hack interface representing HTTP requests and responses	60
5.2. EnvController monad providing the environment for request handling	62
5.3. StateController monad providing a state for response generation	64
5.4. View type class for rendering and its instance TextView for rendering plain text	66
5.5. Functions for accessing the request parameters	67
5.6. Functions for returning alternative responses	68
5.7. Cookie data type and access functions	69
5.8. Functions for accessing the session	71
5.9. Types representing a session and a session store	71
5.10. Functions for accessing the flash and error store	73
5.11. Function for simple routing based on a module/action pattern	75

List of Listings

5.12. Functions for constructing application URLs	76
5.13. Configuration of a Hawk application	78
5.14. Environment of a Hawk application	79
6.1. TemplateView data type	88
6.2. Bind function for inserting content into a template	90
6.3. Helper functions for XmlTree construction	90
6.4. Functions for accessing the template	92
6.5. Function typedView for converting template data types	95
7.1. Calculation of the current work of a project	100
7.2. Action for editing an area	102
7.3. Action combinator for user authorization	103
A.1. Data type for customers	116
A.2. Data type for categories	118
A.3. Foreign key relationship between customers and categories	119
A.4. Short-cut function for accessing the category of a customer	119
A.5. Actions for customers	120
A.6. Layout template with flash access	121
A.7. Template for listing all customers	122
A.8. Template for showing a single customer	123
A.9. Template for editing a customer	123
A.10. The customer views	125
A.11. Application routing	126
A.12. Application configuration	127
A.13. The main function	128

1

Introduction

Since its invention in the early 1990s, the World Wide Web has dramatically grown in its dimensions as well as its importance for everyday life. While in the beginning of the World Wide Web there existed only a very few web sites, nowadays the web has become an important channel for information distribution in the current information society. According to a research done by Netcraft Ltd. in September 2009, more than 200 million web sites are accessible [[Net09](#)].

Not only did the accessibility and technology evolve, but also did the web sites during the last few years. In the beginning, the web sites were mainly published as static sites without further content computation. Because of the upcoming claim for being up-to-date, the sites subsequently became more and more dynamically calculated. Based on the current technology, it is now possible to provide entire applications with web browsers as the user interface. In order to manage the increasing complexity connected therewith, such applications are typically realized based on *web application frameworks*. These frameworks support the developer by providing library functions for the most common tasks of web development.

1.1. Motivation

Although web frameworks have significantly evolved in the past few years, there still is room for further improvement. Many current frameworks are written in scripting languages which are dynamically typed, as well as interpreted at runtime. The probably most prominent representative of those frameworks is Ruby on Rails [Rai, RTH09], implemented in Ruby. Scripting languages often provide an expressive syntax, which dramatically reduces the amount of code needed for achieving certain functionality. Nonetheless, these languages suffer from the fact of runtime interpretation. Due to the lack of error checking at compile time, many programming errors in interpreted languages can only be encountered if the respective code fragment is evaluated. Furthermore, the lack of static typing makes these languages vulnerable for erroneous type conversions. To cite an example, a security problem in Ruby on Rails was discovered in June 2009, which allowed the authentication mechanism to be bypassed. Because of an incorrect type conversion into a boolean value, an invalid user name in connection with an empty password was accepted as valid.¹ According to this, the dynamic typing and the runtime interpretation may reduce the reliability of the software. To overcome this disadvantage, a common solution is to provide extensive test coverage to ensure robustness, which requires considerable effort.

On the other hand, languages like Java, which are static typed and compiled, allow the programmer to detect these problems at compile time. In consequence, a certain class of runtime errors is prevented. However, many of these languages lack expressiveness compared to scripting languages, and therefore require more source code to achieve the same functionality.

Haskell [Jon03], a non-strict, purely functional language with static, polymorphic typing, not only provides an expressive way of programming, but also has the above-mentioned advantages of compile-time error-checking. In conclusion, Haskell can seriously be considered as a programming language suitable for web application programming. Surprisingly, there is currently no framework available for web application development in Haskell which offers a feature set comparable to those of current frameworks like Ruby on Rails.

¹http://weblog.rubyonrails.org/2009/6/3/security-problem-with-authenticate_with_http_digest

1.2. Scope

The aim of this work is to develop and implement a framework for web application development in Haskell. The framework is named “Hawk”, which is a permuted acronym for “Haskell Web Application Kit”. It ought to support the application programmer by providing library functions for the most common tasks of web development, which basically comprises request handling and response generation, as well as a solution for application data persistence.

The architecture of the applications to be developed using Hawk, which is based on the *model-view-controller* pattern, led to the development of three different framework components. To begin with, the *controller* component supports the developer with functionality necessary for the implementation of the request handling. Additionally, it provides an interface to combine the developed web application with different handlers for serving the application to the web. Secondly, the *template system* allows the generation of HTML pages, while providing a clean separation of the HTML templates and the view logic necessary to insert the content into the templates. Beyond that, it also provides a mechanism to enforce the accordance of the template and the inserted values in respect of their structure, as well as of their types. Finally, a component for *database mapping* adds the functionality of persisting arbitrary types into a database. For data retrieval, the database mapping not only supports raw SQL statements, but also provides a small domain specific language, capable of expressing the most common querying expressions in an elegant way.

The implementation of the controller and the template system currently consider server-side processing only. In consequence, the support of client-side processing, which nowadays is typically realized by Asynchronous JavaScript and XML (AJAX), is not covered. However, the framework is designed to enable the subsequent addition of AJAX functionality, as this is a predictable requirement.

1.3. Related Work

Out of the entire collection of web frameworks currently available, the frameworks Ruby on Rails [Rai, RTH09] and symfony [Symfony, ZP07] primarily inspired the design of the Hawk framework. The framework Lift [Lift, CBDW09], written in the

1. Introduction

functional language Scala, gave the inspiration to the current design of the template system.

Although the Haskell community offers a small amount of libraries for web development as well, the majority of these libraries only cover specific aspects. Out of the set of the more comprehensive libraries, the *Haskell Application Server Stack* (*Happstack*) [Ht] provides a wide range of functionality needed for web application development. However, Happstack notably differs from other frameworks in its persistence concept, as it holds the application data in the main memory during execution. On the contrary, *Turbinado* [Kem] is strikingly similar to Ruby on Rails, adopting its key concepts such as dynamic code loading. Nonetheless, besides the core framework, Turbinado only provides little support, lacking features such as an elaborated database mapping or helper functions for HTML generation. Furthermore, with the adopted concept of dynamic code loading and strings to be embedded into the templates, the robustness may be affected.

1.4. Outline

Prior to the development of the framework, an introduction into selected techniques of Haskell programming will follow in chapter 2. The focus is on concepts the reader is not expected to be familiar with, such as the construction of associated types via type families, the current approach of exception handling provided by the GHC, and compile-time meta-programming using Template Haskell.

Subsequently, the development of the framework is described in the chapters 3 to 6. In chapter 3, the architecture of the Hawk framework is developed. On the basis of a characterization of web applications in general, the architecture according to the model-view-controller pattern is motivated. The individual components are refined afterwards, leading towards the final application architecture. After this, in chapter 4 the design and implementation of the database mapping is illustrated. Starting with the low-level database access, the different abstraction levels from a database to Haskell data types are step-wise introduced. Subsequently, in chapter 5 the functionality for request processing offered by the framework is depicted. This does not only contain the functions usable for processing, but also its configuration and integration to form a self-contained application which can be served to the web. To complete the description of the framework implementation, chapter 6 presents

the template system for HTML generation. Beginning with a conceptual overview, its capabilities as well as its functions are successively introduced and discussed.

Based on the functionality illustrated in the implementation chapters, chapter 7 presents an exemplary application built with the Hawk framework. This chapter consolidates the functionality previously described and demonstrates the intended usage. Furthermore, the most interesting aspects of the implementation are discussed in more detail. Finally, chapter 8 concludes with a reflection of the achieved results. Subsequent to a summary and a brief assessment, possible future improvements are identified, in connection with an outlook on the near future of the Hawk framework.

1.5. Notice for Examination

The present Master's thesis is the result of a cooperation of two students. Because of the dependencies between the individual framework components, it was decided to not divide the topic into separate theses. In consequence, this work has been realized in continuous discussions, for both the implementation and the thesis. Although therefore it is not possible to place the responsibility for specific parts on one of the authors, some parts are influenced by one of them to a greater extent. Björn Peemöller especially contributed to the development of the database mapping component, while Stefan Roggensack had a notable influence on the template system. On the contrary, the framework architecture and the controller component can be considered to be influenced by both of them to the same extent.

2

Aspects of Haskell Programming

The framework to be developed in this work will be implemented using the Haskell 98 standard as defined in [Jon03], in combination with some language extensions. The reader is expected to be familiar with functional programming in Haskell, hence the basic principles of Haskell are assumed to be known. For example, this includes the standard Haskell data types and type classes, the role of functions in Haskell, and the concept of monads and monad transformers.

In addition to these fundamentals, this chapter describes further concepts used for the implementation of the Hawk framework, which are considered to be not covered by basic Haskell lectures or books. Section 2.1 introduces the concept of type families, used for the realization of relationships in the database mapping. The exception handling, as discussed in section 2.2, is generally used throughout the entire framework, but also particularly emphasized in the database mapping. Finally, section 2.3 introduces the basics of the meta-programming extension Template Haskell. Its capabilities are used to provide the HTML templates with a corresponding data type to add compile-time type checking for the content insertion.

2.1. Type Families

Type Families [Cha] are type constructors which represent sets of types, contrarily to single types. Unlike ordinary type constructors, they allow the representation to be varied depending on the type parameters of the constructor. Ordinary constructors like `type` have to provide a uniform representation, regardless of the parameter values passed in. If, for example, the element type of collections should be expressed in dependence of the collection's type, it is not possible to denote the fact that a list might contain arbitrary types, while a bit set is restricted to contain only `Chars`, providing an efficient implementation. Therefore, the approach of using a type synonym parametrized with the type of the collection will result in a uniform element type such as

```
type Elem c = Char
```

This problem also remains if the collection type should be expressed in respect of the element's type. It is still impossible to change the general representation for specific element types:

```
type Collection e    = [e]
type Collection Char = BitSet -- wrong!
```

This limitation of uniform representations not only applies for the type synonym used above, but also for other type constructors offered by Haskell, namely the `data` and the `newtype` constructor. To solve this problem, *type families*, provided by the homonymous language extension `TypeFamilies`, allow the subsumption of different representations under a collective type constructor.

A type family can be declared via the additional keyword `family` and has to specify the number of type parameters the result depends on (the *arity*), as well as the kind¹ of the result:

```
{-# LANGUAGE TypeFamilies #-}
type family Elem c :: *
```

In this example, the type family depends on one single type parameter `c`, therefore its arity is 1. The kind of the result is `*`, representing all nullary type constructors, while the overall kind of `Elem` is `* -> *`. In consequence, `Elem` can be seen as a function on the type level. While ordinary functions calculate a value based on their input

¹Kinds classify types just like types classify values.

values, a type-level function calculates a type based on its inputs types. According to this, type families can be used to express the dependency of one type on other types, which will later be used for expressing database relationships.

Members of the type family are provided using the additional keyword `instance` and have to declare the result of the type-level function:

```
data MyBitSet = MBS Integer
type instance Elem [e]      = e
type instance Elem MyBitSet = Char
```

Here the list is used as a collection for arbitrary element types, while the `MyBitSet` provides an efficient collection for `Chars` only. The `instance` keyword furthermore denotes conceptual similarities to type classes. While in type classes the implementation of the member functions varies depending on the respective instance, in type families the result type varies depending on the respective type parameters. Furthermore, just like type classes may be extended by new instance declarations, type families are open for later extension by providing further family members.

In addition to the general form introduced above, type families can also be used inside the declaration of type classes, resulting in the family constructor to be included in the class declaration. In this context, the family constructor may depend on the type parameters of the type class only, or more precisely on a permutation of a subset of the type class parameters. Consequently, the type family is associated with the surrounding type class, motivating the term of an *associated type family*.

Revisiting the above mentioned example of collections, it may be useful to provide the possible collections with a generic interface for insertion, union and membership test. The collections are expected to provide an instance of the type class, as well as declare the type of the elements they support. This can be achieved using an associated type family like:

```
{-# LANGUAGE TypeFamilies #-}
class Collects c where
  type Elem c :: *
  empty      :: c
  insert     :: Elem c -> c -> c
  member     :: Elem c -> c -> Bool
  toList     :: c -> [Elem c]
```

In this class declaration, the `type` definition indicates that the collection type `c` has an associated type `Elem c` of the kind `*`. The associated type may then be used

2. Aspects of Haskell Programming

in the signatures of the member functions of the type class. Because in standard Haskell type synonyms are not allowed inside type classes, the `family` keyword is dropped. An instance declaration has to provide an implementation for `Elem` just like for the member functions. Again, the keyword `instance` is dropped in the type class declaration for the same reasons:

```
instance Eq e => Collects [e] where
  type Elem [e] = e
  empty          = []
  insert         = (:)
  member         = elem
  toList         = id
```

In addition to type synonyms, the concept of type families is also applied to the other type constructors for user-defined types, namely `data` for algebraic data types and `newtype` for type renaming. These *data type families* follow the same rules as the *type synonym families* described above, for both the general and the associated form.

2.2. Exception Handling

The handling of erroneous conditions is a substantial part of building reliable applications. In object-oriented languages like Java, exceptions are used to cover a wide range of potential errors such as exceptions related to I/O or to missing or illegal arguments. On the contrary, in Haskell the latter category of errors can be handled using standard Haskell types like `Maybe` or `Either`, which is also referred to as *pure error handling* [OGS08]. The term denotes the fact that the error handling is done without requiring functionality offered by the `IO` monad, resulting in pure functions. The most common example may be the usage of the type constructor `Maybe`, offering the value constructors `Just` to denote valid results and `Nothing` to indicate an error.

Nevertheless, there are situations in which the pure error handling is not sufficient. Examples are the disaster recovery in case of unexpected conditions such as interrupted network transmissions, or exceptions thrown by foreign libraries. Like in object-oriented languages, these errors are represented in Haskell as exceptions as well. The current Haskell 98 standard [Jon03] includes exceptions in form of the abstract type `IOError`. However, this type is not extensible by the user or by

libraries, which led to several refinements of the exception concept in the past. Hence, this section introduces the exception mechanism currently implemented in the GHC [JRH⁺99, MJMR01, Mar06]. Reflecting the abilities of exception systems provided in object-oriented languages such as Java, the current interface provides

- a *hierarchy of exceptions*, enabling it to catch exceptions of a particular “sub-class” (like in the object-oriented world) and to re-throw other exceptions not contained in this class,
- a way to *extend the exception hierarchy* at all levels by adding new exceptions, as well as new exception sub-hierarchies.

2.2.1. Basic Usage

Exceptions in Haskell are represented as instances of the type class `Exception`. Using special functions, exceptions can be thrown in pure code as well as in monadic code. A common example of such a function is `error`, which is also included in the Haskell standard. This function takes a `String` as a parameter and throws an exception containing the passed-in message. However, the usage of `error` leads to a special exception representation being thrown, denoting the usage of this function. In contrast, arbitrary exceptions can be thrown using the function:

```
throw :: (Exception e) => e -> a
```

For example, a division by zero is represented in Haskell as a `DivideByZero` exception. Naturally, this exception is thrown if a division by zero is detected, but it may in addition be thrown manually:

```
ghci> 1 `div` 0
*** Exception: divide by zero
ghci> throw DivideByZero
*** Exception: divide by zero
```

As long as there is no corresponding exception handler present, exceptions cause the program to be halted and a message to be printed to the standard error channel. In contrast to the throwing of exceptions, the handling of exceptions can only be done inside the `IO` monad. This is caused by the fact that in pure code the order of expression evaluation might be changed for optimization reasons. Therefore, potential exceptions may change the order in which they would occur. While this non-determinism is not allowed in pure code, it can be dealt with inside the `IO`

2. Aspects of Haskell Programming

monad. For catching arbitrary exceptions, the module `Control.Exception` provides the `catch` function:

```
catch :: (Exception e) => IO a -> (e -> IO a) -> IO a
```

The `catch` function takes a computation to be evaluated as its first argument, as well as an exception handler as its second argument. The exception handler is then executed if an exception was thrown during the computation. It is worth noting that the Haskell Prelude defines a similar function `catch`. However, the Prelude version differs in the type signature and is restricted to catch `IOErrors` only. To avoid ambiguity when working with exceptions, it is useful to either exclude the standard `catch` by hiding it, or to use a qualified import of the `Control.Exception` module. Using the more general `catch` function, it is for example possible to catch a `DivideByZero` exception:

```
ghci> throw DivideByZero 'catch' \DivideByZero
      -> putStrLn "caught!"
caught!
```

2.2.2. The Exception Hierarchy

Exceptions are represented as a type class to make the set of exceptions extensible, so that new exceptions can be added later. These new exceptions can then be thrown and caught using the standard functions. However, introducing a type class leads only to an extensible *set* of exceptions, in contrast to the intended *hierarchy*. In consequence, there has to be an additional technique to support hierarchical ordering. A first step is to provide a root node, called `SomeException`, which constitutes the top of the entire exception hierarchy. As this type is the root node of all exceptions, every exception can be caught as a `SomeException`. Additionally, this type is used inside the declaration of the class `Exception`, shown in listing 2.1.

```
class (Typeable e, Show e) => Exception e where
  toException    :: e -> SomeException
  fromException  :: SomeException -> Maybe e
```

Listing 2.1: Exception type class representing Haskell exceptions

The instances of this type class have to be an instance of `Show`, which is needed to print uncaught exceptions to the standard error channel. The instance declaration of the type class `Typeable` is used for the internal down-casting of exceptions to check whether they belong to a particular exception class. These conversions are used in the member functions responsible for wrapping exceptions. Of these, the function `toException` wraps specific exceptions into the root node of the exception hierarchy, `SomeException`. Assuming congruent types, `fromException` is used to unwrap the exceptions afterwards. The type `SomeException` thereby relies on the language extension of existentially quantified types [HHJW07]. This allows the exclusion of the internal exception from the wrapper type to provide a uniform interface:

```
data SomeException = forall a. (Exception a) => SomeException a
```

On the basis of these declarations, it is possible to determine the exact type of the exceptions thrown in the previous examples. Internally, the `DivideByZero` exception belongs to a set of arithmetic exceptions, which is represented as an algebraic data type:

```
data ArithException = ... | DivideByZero | ...
```

In the exception hierarchy, an arithmetic exception is located directly under the root of all exceptions. In conclusion, the exact value of a `DivideByZero` as a thrown exception can be derived as:

```
SomeException DivideByZero :: SomeException ArithException
```

Therefore, the `DivideByZero` exception can be seen as a member of the set of arithmetic exceptions (compared to a sub-class in the object-oriented world), as well as a member of the set of all exceptions. According to this, it is possible to catch a division by zero on three different levels: as a `DivideByZero` exception, an `ArithException`, and a `SomeException`. The exception handler can be adjusted to handle specific kinds of exceptions by explicitly providing the type of the exception to be caught:

```
ghci> throw DivideByZero 'catch' \(e::ArithException)
      -> putStrLn ("caught: " ++ show e)
caught: divide by zero
ghci> throw DivideByZero 'catch' \(e::SomeException)
      -> putStrLn ("caught: " ++ show e)
caught: divide by zero
ghci> throw DivideByZero 'catch' \(e::IOException)
      -> putStrLn ("caught: " ++ show e)
*** Exception : divide by zero
```

2. Aspects of Haskell Programming

In the first statement, the exception is caught as a member of the set of arithmetic exceptions. As a `DivideByZero` also is a `SomeException`, it can additionally be caught on this most general level, as demonstrated in the second statement. Finally, in the third statement the exception is *not* caught. This is because a `DivideByZero` does not belong to the `IOExceptions`, which are located beneath arithmetic exceptions, but not above.

2.2.3. Extending the Exception Hierarchy

To create a new kind of exception, it is sufficient to declare the respective type as an instance of the type class `Exception`. Using the compiler extension `DeriveDataTypeable`, the instance declaration of the type class `Typeable` can be derived, comparable to the standard Haskell type classes like `Show`. Providing an instance of both `Typeable` and `Show`, the new exception type can be declared as an instance of `Exception`:

```
{-# LANGUAGE DeriveDataTypeable #-}
import Data.Typeable
import Control.Exception

data AppException = AppException deriving (Typeable, Show)
instance Exception AppException
```

Because the instance declaration does not implement the member functions `toException` and `fromException`, the default implementations are chosen. This causes the `NewException` to be located directly under `SomeException`, and therefore on the same level in the exception hierarchy as exceptions like `IOExceptions` or `ArithExceptions`. Just like the exceptions already provided, the new exception can be thrown and caught just like any other:

```
ghci> throw AppException 'catch' \(e::AppException) -> print e
AppException
```

If the exceptions are supposed to be further divided, there are two alternatives. The first and nearby solution is to summarize different exceptions in an algebraic data type, such as:

```
data AppException = ThisException | ThatException
```


This approach is currently used for the standard Haskell exceptions, such as arithmetic exceptions, or exceptions related to I/O. As already demonstrated, this solution provides enough flexibility to catch exceptions based on the exception hierarchy. Nonetheless, it is limited in its flexibility, as the data declaration of `AppException` has to enumerate every foreseeable exception. In consequence, there is no way to add new exceptions later. Compared to the object-oriented language Java, this can be seen as an equivalent to final classes.

The second solution allows it to establish a new exception hierarchy, which can be extended later. To achieve this, an additional exception class has to be implemented. This is realized in a way similar to the `SomeException` already introduced. To begin with, the new exception hierarchy has to be declared as:

```
data SomeAppException
  = forall a . (Exception a) => SomeAppException a
  deriving Typeable

instance Show SomeAppException where
  show (SomeAppException e) = show e

instance Exception SomeAppException
```

To facilitate the later addition of exceptions on a lower level, it is reasonable to define additional helper functions for sub-classing new exceptions:

```
appToException :: (Exception a) => a -> SomeException
appToException = toException . SomeAppException

appFromException :: (Exception a) => SomeException -> Maybe a
appFromException x = do
  SomeAppException a <- fromException x
  cast a
```

New exceptions which shall be treated as `AppExceptions` can then be defined as follows:

```
data NewException = NewException deriving (Typeable, Show)

instance Exception NewException where
  toException    = appToException
  fromException  = appFromException
```

Comparable to the previous solution based on an algebraic data type, it is possible to catch a `NewException` as a special case of an `AppException`. Furthermore, the

2. Aspects of Haskell Programming

hierarchy is extensible because the user may add new exceptions later by making them a sub-class of `AppException`. A further sub-hierarchy of `AppException` is also possible by introducing new hierarchy levels. This can be achieved in the same way as an `AppException` is declared as a sub-hierarchy of `SomeException`.

2.2.4. Lifted Exception Handling

The functions described so far allow the handling of exceptions inside the `IO` monad. In consequence, this implies that throwing an exception will result in the program being transferred to the `IO` monad, where exception handling is available. However, many applications do not only rely on the ability to perform I/O actions, but also on additional functionality like an internal state or an application environment. This functionality usually is achieved by combining the `IO` monad with the respective monad transformers such as in

```
type StateAndIOMonad = StateT MyState IO
```

As the majority of the code may rely on both state access and I/O, it would be desirable to be able to handle exceptions inside the monad transformer. This would lead to a more precise exception handling of selected functions, for example inside the `StateAndIOMonad`, in addition to the general exception handling of the `IO` monad outside. For instance, when calling `readFile` with a `FilePath` referencing a nonexistent file, the content may be set to the empty input `""` without leaving the `StateAndIOMonad`.

For *performing* I/O actions inside other monads, the I/O actions have to be lifted from the `IO` monad into the target monad. This is done by declaring the respective monad as an instance of the type class `MonadIO`, shown in listing 2.2.

```
class (Monad m) => MonadIO m where
  liftIO :: IO a -> m a
```

Listing 2.2: `MonadIO` type class for lifting I/O actions

Providing the instance declaration of `MonadIO`, I/O actions can then be lifted into the new monad using the lifting function, like `readFile' = liftIO . readFile`. However, although this approach is sufficient for the majority of I/O actions, it cannot be used to lift exception handling. The reason becomes obvious by looking at the type signature of the `catch` function and its lifted version:

```
catch :: Exception e => IO a -> (e -> IO a) -> IO a
\x -> liftIO . catch x :: (Exception e, MonadIO m)
    => IO a -> (e -> IO a) -> m a
```

Although the result of the `catch` function can be lifted into other monads, the lifted function would still be restricted to catch exceptions inside `IO` computations only. To provide exception handling for other monads capable of I/O functionality, the `catch` function has to be generalized to:

```
catchM :: (Exception e, MonadIO m) => m a -> (e -> m a) -> m a
```

This generalization is available by declaring the respective monad as an instance of the type class `MonadCatchIO`, shown in listing 2.3.

```
class MonadIO m => MonadCatchIO m where
    catch :: Exception e => m a -> (e -> m a) -> m a

    -- functions used for asynchronous exceptions
    block  :: m a -> m a
    unblock :: m a -> m a
```

Listing 2.3: `MonadCatchIO` type class for lifting exception handling

Based on the member function `catch` of the type class, additional functions like those offered by the `Control.Exception` module are available. The functions `block` and `unblock` are necessary for supporting asynchronous exceptions such as system interrupts, or exceptions between concurrent threads. In addition, the corresponding library [\[Mon\]](#) provides instance declarations for the most common monad transformers, such as the `StateT` transformer.

2.3. Meta-programming using Template Haskell

Meta-programming denotes the calculation of programs by program code, instead of writing it manually. Meta-programming can be used to reduce the effort for creating source code of the same structure, or to compute code in respect to additional information. Depending on the moment when the meta-program is executed, meta-programming can be divided into two kinds: run-time and compile-time meta-programming. *Run-time meta-programming* comprises the creation or modification of program fragments while the program itself is executed. A common example is

2. Aspects of Haskell Programming

reflection, to some extent also provided for Haskell via the “Scrap Your Boilerplate” library [LJ]. Furthermore, in many scripting languages it is possible to generate code at run-time as a string and execute it later. On the contrary, *compile-time meta-programming* refers to the generation of source code at compile time, which is integrated and compiled in combination with other source code after its generation.

Template Haskell [SJ02, SJ03] is a compile-time meta-programming extension for Haskell, allowing the algorithmic construction of Haskell code at compile time by providing respective Haskell functions. This extension generally can be seen as a template system for Haskell, just like the templates available at C++, as well as a type-safe macro system. Template Haskell covers almost the entire program structures Haskell provides, except for some language extensions. The template functions themselves are written in Haskell and therefore type-checked, as well as compiled using the same techniques like for ordinary Haskell functions.

2.3.1. Basic Principles

The program fragments generated by template functions can be inserted into regular modules using so-called *splice expressions*:

```
const2 = $(constN 2 "42")
const3 = $(constN 3 "42")
```

A splice expression is denoted by a `$` and initiates the execution of the referenced template function. The result of the template function, a Haskell program fragment, is inserted at the position of the splice expression. After the insertion, the module is compiled as usual. In the example stated above, the template function takes two parameters, requiring the function call to be put in parentheses. To distinguish splice expressions from the Haskell infix operator `$`, no space is allowed between the `$` and the opening parenthesis. For template functions without parameters, the parentheses may be omitted; in this case the `$` has to be directly followed by the function name, like in `$thfun`. The parameters of template functions have to be computable at compile time, as well as the template function. In consequence, the function has to be imported from another module.

The splice expression shown above results in a generalization of the standard `const` function to be inserted, which ignores a customizable number of parameters and returns the second parameter passed to the template function. The splice expressions therefore lead to the following code:

```
const2 = \_ -> \_ -> "42"
const3 = \_ -> \_ -> \_ -> "42"
```

As the `constN` function produces a program fragment as its result, it therefore can be seen as a *meta-program*. A simple way to construct such template functions is the usage of *expression quotations*. Expression quotations allow the construction of template functions out of fragments of regular Haskell code. Therefore, these quotations can be seen as the “templates” of Template Haskell. An expression quotation consists of a Haskell fragment put in Oxford brackets `[| ... |]`. Using this notation, the `constN` function can be implemented as:

```
constN 0 x = [| x |]
constN n x = [| \_ -> $(constN (n-1) x) |]
```

This definition calculates the resulting function recursively. If the number of the parameters to be ignored equals zero, the final result is returned. The expression `[| x |]` thereby denotes that the parameter `x` is lifted into an expression returning the value of `x`. In the second case, an additional wild card is introduced before the template function itself is invoked.

2.3.2. Internal Representation

The program fragments constructed by Template Haskell internally are represented as an algebraic data type, the abstract syntax tree of the respective fragment. Code that is to be inserted in general can be divided into two kinds, expressions and declarations. *Expressions* of the type `Exp` result in Haskell expressions such as literals, pattern matches or lambda expressions. *Declarations* of the type `Dec` subsume class or type declarations, type signatures or function declarations. The construction of program fragments is done inside the `Q` monad, as some kinds of functions require monadic functionality. For example, the template function introduced above can be determined to be of the type:

```
constN :: Int -> String -> Q Exp
```

To investigate the internal structure of the program fragment representation, it is possible to evaluate template functions using the `runQ` function. Evaluated in the GHCi, this function returns the syntax tree of the result. The result may then be printed out in either its internal representation or in a pretty-printed form, which

2. Aspects of Haskell Programming

complies with the Haskell syntax that would have been used to implement this function:

```
ghci> runQ (constN 1 "0")
LamE [WildP] (ListE [LitE (CharL '0')])
ghci> runQ (constN 1 "0") >>= putStrLn . pprint
\_ -> ['0']
```

The first output illustrates the internal representation of the result, a lambda expression containing a pattern with one wild card parameter. The result of the lambda function is a list expression, containing a literal expression with the character literal '0'. The corresponding Haskell code is shown in the second output.

Alternatively, the `constN` may also have been constructed without the Oxford brackets by a combination of constructors of the abstract syntax tree. In fact, many expressions cannot be expressed using the Oxford bracket notation. For example, the lambda expression introduced above may have been expressed as a single lambda function with multiple parameters. While the replication of wild card patterns is not possible using Oxford brackets, it can be achieved via functions for constructing the syntax tree:

```
constN2 :: Int -> String -> Q Exp
constN2 n x = return $ LamE (replicate n WildP)
                        (LitE $ StringL x)
```

In contrast to the previous version with nested lambda expressions, this function replicates the wild card pattern:

```
ghci> runQ (constN 2 "0") >>= putStrLn . pprint -- old version
\_ -> \_ -> ['0']
ghci> runQ (constN2 2 "0") >>= putStrLn . pprint -- new version
\_ _ -> "0"
```

2.3.3. The Quotation Monad

Until now, the example functions relied only on the capabilities of the syntax tree, without using the surrounding quotation monad, or the `Q` for short. However, the quotation monad becomes necessary for more complex template functions, as it among others provides the following features:

- Generation of fresh names

- Performing of I/O actions

The *generation of fresh names* is necessary if the template functions internally use variable names. As those names in principle may conflict with names outside of the body of the function, the `Q` monad provides fresh names via the function `newName :: String -> Q Name`. For example, it may be desirable to create functions to access the m -th element of an n -ary tuple, comparable to the standard functions `fst` and `snd` for pairs. The intended function `sel` take the parameters m and n and should behave like the following:

```
sel 1 3 ==> \(x,_,_) -> x
sel 2 3 ==> \(_,x,_) -> x
```

To identify the m -th component of the tuple, a variable name is necessary. Once computed, it can be used in the remaining code afterwards:

```
sel :: Int -> Int -> Q Exp
sel m n = do
  x <- newName "x"
  return $ LamE [TupP (replicate (m-1) WildP ++ (VarP x) :
    replicate (n-m) WildP)] (VarE x)
```

Another important feature of the `Q` monad is its capability to perform I/O actions. This feature enables Template Haskell to construct program code based not only on parameters directly passed to the function, but also on additional information. It is, for example, possible to read arbitrary files from the file system and derive data types based on their content. This approach will be used later in this thesis for deriving data types from XHTML templates.

3

Architecture

Web applications involve a variety of different aspects, as already foreshadowed in the introduction. In consequence, the application architecture has to be carefully considered, which is the topic of this chapter. It introduces the structure of Hawk applications, as well as the considerations which led towards it. To clarify the working principle of a web application, the fundamental characteristics are illustrated in section 3.1 first. Reflecting these characteristics, section 3.2 then describes the derived architecture of Hawk applications.

3.1. Characterization of Web Applications

The aim of this thesis is the development of a framework for implementing *web applications* in Haskell. The term *web* refers to the fact that the applications is provided to the internet, and in consequence has to be based on the standards used therein. This implies HTTP [[RFC2616](#)] as the underlying communication protocol and HTML [[RLHJ99](#), [Mis02](#)] as the format used for representing the user interface of the application. In addition to this, features such as cascading style sheets (CSS)

3. Architecture

[BCHL09] and JavaScript [Int99] may be used for enhancing the user interface in both usability and elegance.

The term *application* in this context refers to the bundling of multiple HTML pages and resources [Mor07]. These are logically related and accessible to the user as a whole by a web server under a certain root path. In contrast to static web sites with a fixed content, web applications generate the responses dynamically in respect to the incoming request and internal application data.

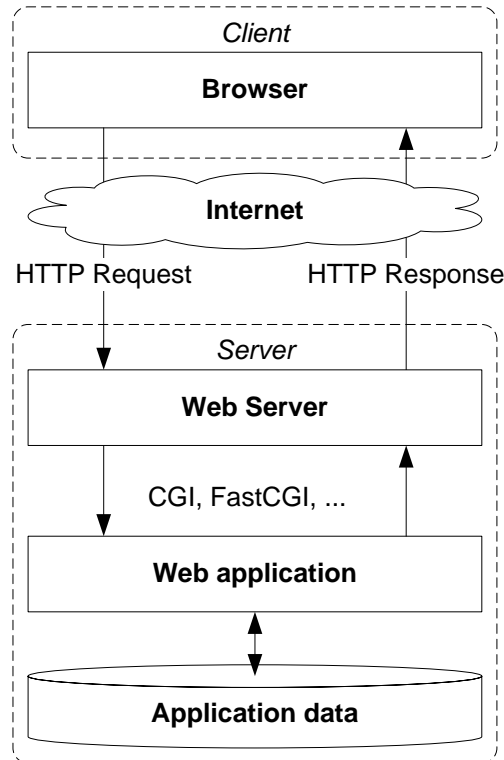


Figure 3.1.: Schematic data flow for a web application

Web applications are invoked by the user by requesting the URL of the application via the browser, resulting in a HTTP request send to the web server (cf. figure 3.1). The server accepts the request and forwards it to the web application. Assuming an HTML page as output, the application then generates the page, which is handed over to the server in the shape of a HTTP response. Consequently, the HTML page constitutes the application interface to the user. In addition to HTML as the standard format for the user interface, it is nowadays common to provide a range of different output formats for web applications, such as the JavaScript Object Notation

(JSON) [RFC4627] used in connection with JavaScript. Finally, the web server sends the computed response to the web browser, which in turn shows it to the user.

The connection of the web application to the web server in general can be realized based on a variety of techniques. If the application is provided as a binary program, one possible solution is the Common Gateway Interface (CGI) [RFC3875]. This technique uses the standard input of the invoked application and environment variables to provide the request, while the application in turn provides the response via the standard output. Because a new instance of the application is created for every request, the CGI suffers from performance issues, due to the overhead for setting up the application environment. This issue is addressed by the FastCGI technique [OM96], in which the application is started only once. After that, the server and the application communicate via network sockets. Furthermore, there are even other techniques like the direct integration of the application and server functionality into one executable. In conclusion, there is a variety of non-uniform interfaces for the communication of the web server and the application, which has to be considered for the architecture of the Hawk framework.

Finally, web applications typically require additional data to realize their functionality, such as the list of products in a web shop. Such data, fundamental to the application, has to be stored on (or at least be accessible by) the server computer. While the file system may be sufficient for a limited set of data, major data sets are typically stored inside a database.

3.2. Application Architecture

Based on the interaction pattern described in the previous characterization of web applications, three different tasks for a web application can be identified:

- Request Handling
- Data Access
- Response Generation

The *request handling* refers to the acceptance of the request passed in, and the initiation of respective actions. For example, this may be the addition of a product to a shopping cart. The term *data access* denotes the access of the web application to the underlying data store, such as a database. Subsequent to the request handling

3. Architecture

and an eventual data access, the *response generation* is responsible for creating a response based on the request, in the shape of one of several supported formats.

Reflecting these general tasks, it is reasonable to choose an application architecture supporting a separation of them. This does not only clarify their responsibilities, but also let them focus on particular parts of the application. While there are several architectural patterns established for structuring web applications, the *model-view-controller* (*MVC*) pattern seems to be the best choice, as its component structure reflects the different tasks identified above. This pattern has not only been proven to

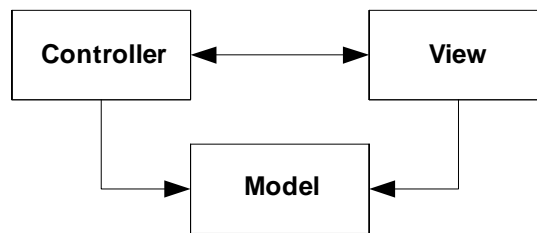


Figure 3.2.: Model-view-controller pattern

be useful for desktop application, but also it is used for many web applications. The MVC pattern, shown in figure 3.2, identifies three components with the following responsibilities [Fow03]:

Model: The model component is responsible for representing the domain of the respective application, both in data and logic. For a web shop, this may be the product catalogue, or transaction logic for processing an order.

View: The view component is responsible for rendering the information to be presented to the user. For a web shop, this may be a certain product page, or a page displaying the content of the user's cart.

Controller: The controller component is responsible for the processing of the incoming request. In addition, it generates the information to be displayed, typically by means of the functionality the model provides. For a web shop, this may be adding a product to the user's cart.

The MVC pattern serves two purposes: It encourages the separation of the data (model) from its presentation, as well as the division of the presentation into its processing (controller) and its visual representation (view). Usually, the controller alters the model and calculates the information to be displayed, while the view may

also access the model to retrieve further information for rendering. The interaction of the controller and the view generally is not specified by the pattern.

Although the separation of controller and view is not necessary, it offers more flexibility as it allows the combination of a single controller with different views, as well as vice versa. The former may for example be useful to provide multiple forms to the user, sharing the same model they operate on. Thus, changes on the model can be restricted to different attributes, depending on the respective form. In the opposite direction, a uniform view may be combined with multiple controllers, which realize different projections on the model. For example, the same product list layout may be used to display the hot offers of a web shop, as well as the products highly ranked.

As the MVC pattern only constitutes a basic architectural pattern, its implementation in the Hawk framework has to be further specified. Therefore, the different components and their responsibilities are investigated in more detail in the following.

3.2.1. Model Component

The model component generally implements the application domain by means of the application data. Besides the *application data logic*, a major task of the model therefore is to provide the access to the application data. As already mentioned, for data storage there are several possible solutions, for example a binary file or a database. However, Hawk is intended to support the development of larger applications, which are considered to require a large amount of application data. Therefore, it was decided to provide access to a relational database in the framework. This does not prohibit the usage of alternative concepts such as file storage, which in some situations may even be superior. However, it is expected that a database will be the appropriate solution for the majority of applications.

While the data logic operates on the data types defined in the application, the data access in addition has to deal with the data types provided by the database. In consequence, it is reasonable to divide the model component into two layers, the *data logic layer* and the *database access layer* (cf. figure 3.3). The data logic has to be written by the developer, as this part is specific to the application's domain. On the contrary, the database access is provided by the Hawk framework in the form of a database mapping, which will be described in chapter 4.

3. Architecture

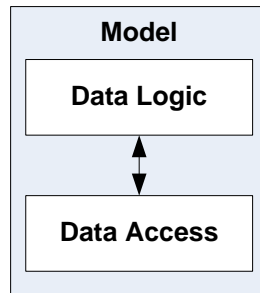


Figure 3.3.: Layers of the model component

Using a database as the data store, the types of the model component are expected to reflect the database layout. That is, the database mapping is designed to provide a bi-directional mapping from a data type of the model to a specific table inside the database. In consequence, it is reasonable to adopt this structure inside the model by according modules. However, this is not mandatory for the data logic, so the developer is free to structure it to best fit the needs of the application domain.

3.2.2. View Component

The view component is responsible for rendering the response, eventually based on information provided by the controller or by the model. A response in general can be generated in one of a wide range of different formats, such as HTML, JSON or plain text. As the set of supported formats shall not be restricted, the framework provides an extension mechanism which allows the implementation of additional view concepts in the future. However, the rendering of HTML responses is considered to be an important task of the view component and therefore particularly covered.

For rendering information to be displayed into an HTML page, there are two common approaches [Fow03]:

- The template view pattern
- The transform view pattern

In the *template view pattern*, the resulting HTML page is represented as a template in which the dynamic content is integrated later. The template contains markers to indicate places where the parts of dynamic content is inserted, which is done during rendering. To form valid HTML, the information has to be converted into the respective HTML elements before insertion. A common example are Java

Server Pages, also available for Haskell in the form of Haskell Server Pages [Bro05]. Contrarily, in the *transform view pattern*, the information to be displayed is the start of the output generation. Following certain transformation rules, the output is transformed step-wise into HTML.

The main advantage of the template view pattern is the fact that the structure of the resulting HTML pages is available in form of the template, and therefore can easily be edited and displayed. This considerably simplifies the design of the web site, as the design can be done by a web designer using special tools. In contrast, the transformation view pattern requires transformation rules implemented by a programmer for realizing the web site design. However, the main advantage of the transform view pattern is the fact that the construction can be verified to produce valid HTML only. For the Hawk framework, the advantages of the *template view* are higher weighted, leading to the adoption of the template view pattern. Nonetheless, due to the extensibility mentioned above, it is also possible to add other concepts of HTML generation later.

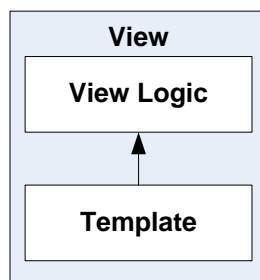


Figure 3.4.: Layers of the view component

With the decision to support templates the question arises, about where to put the view logic for converting data into HTML fragments. Many template systems use scripting tags included in the HTML template, allowing it to put the view logic into the template. Unfortunately, it can often be observed, that the view logic is infiltrated by complex processing logic, which in turn decreases the maintainability. In consequence, a different approach is chosen for Hawk templates. The templates are designed to define so-called *binding targets* only, in which content is inserted. The formatting of the content, as well as the insertion, is done using Haskell. In conclusion, the view logic and the template are separated, as depicted in figure 3.4. The details of the template system and its usage are illustrated in chapter 6.

3.2.3. Controller Component

The controller component of a web application is responsible for handling the request. This may include extraction of the request parameters, access to the model, and provision of data to be rendered by the view. Although it is possible to realize the request handling for the entire application in a single controller, this would lead to unnecessarily complex code. Instead, the controller component is structured into smaller controllers, each responsible for handling the request of a specific URL. For example, there may be a controller only responsible for listing the products of a web shop.

Although the controllers realize different logic, they share a considerable amount of processing. For example, every controller may need to access the application environment, or may extract the parameters of a request. To ease the implementation of new controllers, these common tasks will be factored out into a *front controller* [Fow03], as shown in figure 3.5. The front controller is responsible for setting up the environment necessary for handling the respective request, which is done by the controllers the developer provides. To provide an unambiguous terminology, the controllers provided by the developer shall be named *actions*, as the term “controller” is already used in a different context.

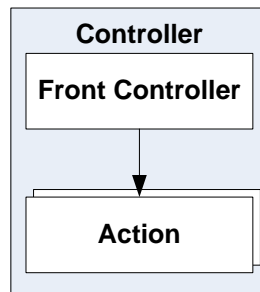


Figure 3.5.: Layers of the controller component

Beside the provision of the environment for request handling, the front controller is also responsible for choosing the action to be invoked in respect of the request. In consequence, the different actions are logically composed by the front controller towards a single controller, providing a unique entry point to the entire application. The configuration of the front controller and the environment provided for implementing actions are presented in chapter 5.

3.2.4. Controller/View Interaction

Encouraging the separation of controller and view, there arises the question about their relationship towards each other. It is both possible that the controller references the view as a renderer for the results of its computation, as well as the opposite, the view referencing the controller as the data source for the rendering. The decision mainly depends on what a URL references: an action to be performed, which may be rendered, or a resource which may require some actions to be performed during its rendering.

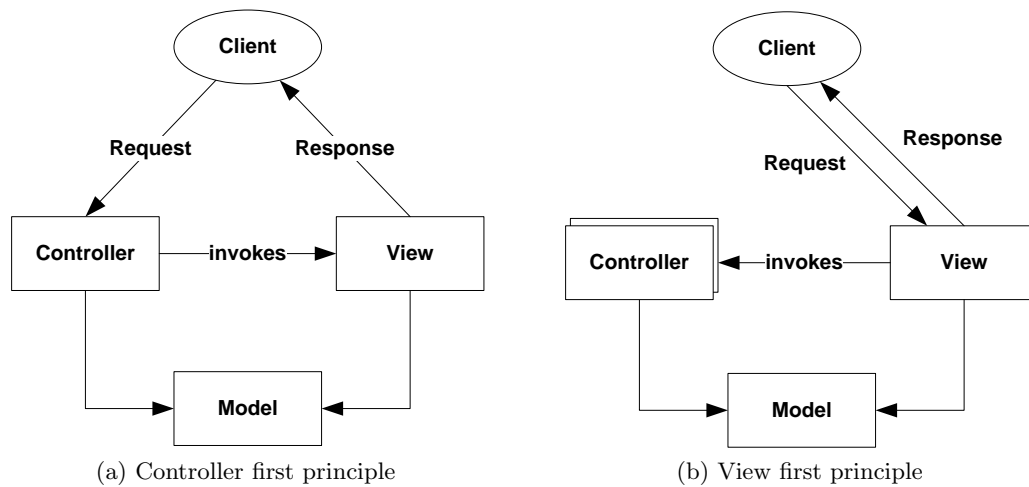


Figure 3.6.: Principles for controller/view interaction

The so-called *controller-first* approach (cf. figure 3.6a) is the approach predominantly used in web frameworks. Here the URL refers to a controller which is then performed. Afterwards, it is decided whether the response to be returned has already been computed, which may be the case if the user shall be redirected to another URL. Otherwise, the response will be generated by rendering the associated view. If the performed action calculated information to be displayed, this information will also be passed to the view.

In contrast, in the *view-first* approach (cf. figure 3.6b) a URL refers to a specific view to be rendered to calculate the response. The view may reference one or more controllers, which then are invoked to calculate the dynamic content to be displayed in the view.

3. Architecture

Although both approaches are applicable in general, the Hawk framework follows the *controller-first* approach. The reason for this decision is to provide a clearer separation of processing and visual representation, as the view first approach implies invocation logic inside the view. Consequently, a request will first be handed over to the controller, or more precisely the front controller which will invoke an action. However, this decision only has little influence: In Hawk, the view component has full access to the request during rendering, and the user is free to provide different views. Therefore, it is perfectly possible to create a view invoking controllers to emulate a view-first behaviour.

3.2.5. Web Server Integration

As already mentioned in the beginning of this chapter, there are several ways to connect a web application to a web server. Consequently, Hawk shall be able to be used with at least the most common variants. Because the front controller is responsible for routing the requests to the internal controllers, no further routing functionality has to be provided by the server. More precisely, an application built with the Hawk framework will accept all requests located under the root location of the entire application. However, although there are multiple server implementations available for Haskell, they generally do not share the same interface. Furthermore, the same is also true for alternative approaches such as CGI handlers.

Fortunately, in mid 2009 the Hack interface [Wan] was published to the Haskell community. This interface unifies the data structures used for the invocation of web applications. Furthermore, a variety of so-called Hack handlers has been developed, covering concepts such as CGI or simple web servers. Unlike the implementations previously available, these handlers all share the same interface. Therefore, the serving of Hawk applications is based upon the Hack interface. This does not only supersede the implementation of an own server, but also dramatically increases the usability of the framework in different environments.

3.2.6. Final Architecture

The final architecture of the web applications to be implemented using Hawk, as well as the schematic data flow, is depicted in figure 3.7. In this implementation, an incoming request is first handed over to the front controller, which analyses the

request and invokes the respective action. The action then handles the request, generally using the logic provided by the model, and calculates the content to be rendered by the view. Assuming an HTML page to be returned, the view then converts this content into HTML fragments, which are embedded into a template afterwards. Furthermore, the view logic is free to access the model to retrieve additional information to be displayed. The resulting HTML page is handed over to the front controller, which converts the page into a complete response and returns the response to the Hack handler.

To encourage the structuring of the application, the actions and views are grouped into so-called *modules*. A module subsumes a set of logically related actions and views, like those regarding a specific type of the domain model like for listing, editing, creation or deletion. Consequently, it is recommended to reflect the structure of actions and views in the Haskell modules as well. Whether a module contains both the action and the view, or if these are separated in two modules, is up to the developer.

3. Architecture

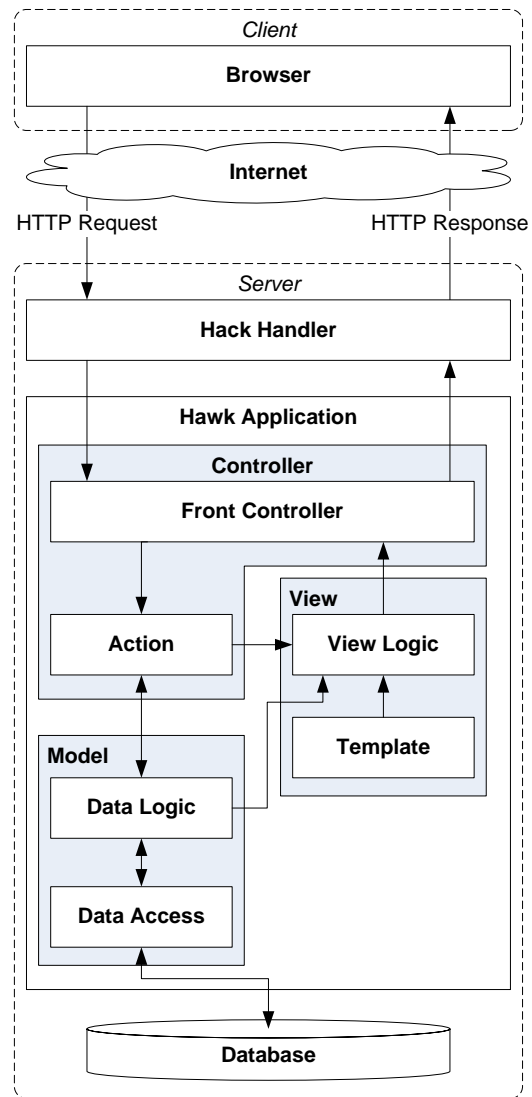


Figure 3.7.: Architecture and data flow of a Hawk application

4

Database Mapping

This chapter presents the database mapping provided by the Hawk framework. As the task of mapping Haskell data types to a database involves several steps, the mapping is divided into different layers. These layers are subsequently described, beginning with the database access and followed by the mapping of types and relationships. After the description of additional functionality beyond simple mapping, the chapter concludes with a brief summary of the different concepts and their relationships.

To demonstrate the concepts in this chapter, they will be described with the help of a continuous example. Figure 4.1 shows an extract of a simple database schema for a web shop, containing the two tables *customers* and *categories*. While the former represents the customers of the web shop, the latter is used to classify them. A customer is specified by his name, consisting of the *first name*, the optional *initials* and the *last name*, as well as his *date of birth*. In addition, the corresponding table contains a column *updated_at*, containing the timestamp when the record was updated. A customer category consists of a unique *name* for identifying the category and *discount* information, containing the discount assigned to the respective category in percentage. Customers and categories are connected by a 1 : *n* relationship: Every customer belongs to one category, while a category naturally has many customers.

4. Database Mapping

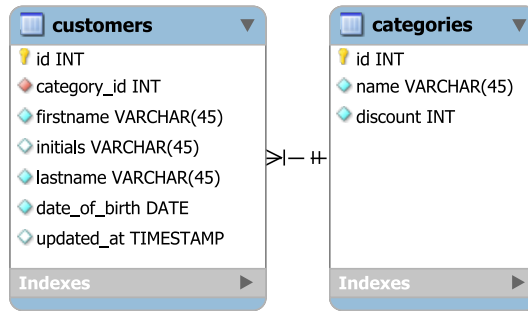


Figure 4.1.: Exemplary database model

The customers and categories are represented in Haskell as records, containing a field for every of the above mentioned properties. The structures of the types are shown in listing 4.1.

```
data Customer = Customer
  { _id      :: Integer
  , categoryId :: Maybe Integer
  , firstName :: String
  , initials  :: Maybe String
  , lastName  :: String
  , dateOfBirth :: Day
  , updatedAt  :: Maybe UTCTime
  } deriving (Eq, Read, Show)

data Category = Category
  { _id      :: Integer
  , name     :: String
  , discount :: Int
  } deriving (Eq, Read, Show)
```

Listing 4.1: Types for customer and category

The types and tables of this example will be used to illustrate the different steps, and furthermore will be picked up in the subsequent chapters. Therefore, appendix A additionally presents the source code of a Hawk application based upon this example.

4.1. Concept

The database mapping in general is responsible for saving types of the domain model into a relational database, as well as retrieving the data. While the database is accessible by means of SQL, the mapping component should provide an interface capable of handling the types defined in the domain model. In consequence, a

mapping from a user-defined type to the data structure provided by the database becomes necessary. Furthermore, it is reasonable to provide the mapped types with a uniform interface to ease the usage of the storage. These requirements, regarding different levels of abstraction, are reflected by the structure of the database mapping, leading to different layers (cf. figure 4.2).

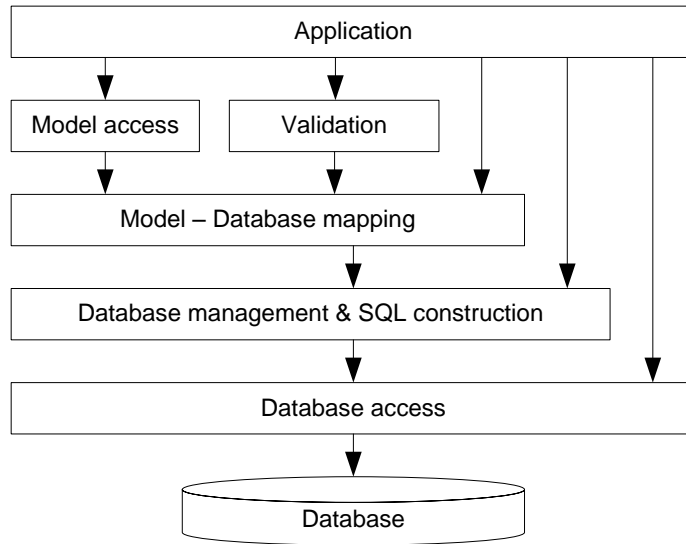


Figure 4.2.: Layers of the database mapping

Database access: The *database access* is responsible for the communication with the underlying database. As database access is a common part of software development, the existing libraries available for Haskell were investigated. In general, there are currently three notable projects providing database access. Haskell Database Connectivity (HDBC) [Goea] and HSQL [Ang] both are libraries which provide access to databases on the level of SQL [OGS08, p. 493]. As HDBC was developed to fix some problems with HSQL, it seems to be more stable. In addition, HDBC provides useful features such as an automatic escaping mechanism, which HSQL currently does not. Contrarily, HaskellDB [BAH04] is a library based on either HDBC or HSQL, which uses the concept of extensible records [JJ99] for representing data stored in the database, resulting in an API of a higher level. Unfortunately, HaskellDB introduces a lot of additional complexity as the types to be mapped to the database are restricted to be extensible records only. Furthermore, there is no support of raw SQL statements as HaskellDB operates on an internal query mechanism only. In conclusion, the Hawk database mapping is based on the HDBC library.

4. Database Mapping

Database management: To identify the database, JDBC uses an additional connection parameter in the access functions. To release the user from the need to manually provide the connection, it is reasonable to provide a mechanism for managing the database connection. Therefore, the *database management* encapsulates the connection and provides a simplified interface for accessing the database. As the web applications are expected to use only one single database, the database abstraction therefore is currently limited to manage one connection only.

SQL construction: As the basic access is realized on the level of SQL, there has to be a way to consistently construct SQL statements. Although the majority of the statements will be constructed internally, the developer will also have to construct queries specific to the application. In conclusion, the construction of at least the most common SQL statements should be supported, which is done by introducing a small domain specific language.

Model-Database Mapping: The mapping between types of the domain model and the underlying database schema, which constitutes the core part of the database mapping, is done by the *model-database mapping* layer. The mapping is provided by a set of type classes, which specify the mapping of a type of the domain model to a database table. Using this information, it is furthermore possible to map foreign key relationships between the tables to relationships between the types. For both, mapped types and relationships, the mapping layer offers common functions, such as *create*, *read*, *update* and *delete* functions for types, as well as functions for relationship traversal.

Model Access: Although the mapping information enables the framework to supply functions for database access on the level of the domain model types, only default implementations can be provided. Nevertheless, it may be necessary to change these functions to realize additional functionality, such as updating of timestamps whenever a value is updated in the database. To provide the developer with the ability to customize the behaviour of the mapped types, the *model access* layer provides a uniform interface which enables the user to overwrite the default implementation.

Validation: Finally, the validation component offers functions for checking whether some constraints of the model's types are satisfied or not. In web applications, validation is often done after the domain model has been updated by the

parameter of a request. Therefore, the validation component in addition supports the validation of updates of selected attributes, based on `String` values.

In general, each layer is accessible by the user's code. Although the usage of functions of lower levels like direct JDBC access is not intended, it is still possible and enhances flexibility, as the user is free to avoid dispensable abstraction overhead if necessary.

4.2. Database Management

The database management lifts the access functionality provided by JDBC to a higher level, managing the connection internally. In JDBC, the SQL statements are represented as string literals, which may contain parameters denoted by a question mark. This allows the usage of prepared statements, as well as releases the user of manually escaping values. This in turn facilitates the usage by superseding manual escaping, as well as reduces the risk of SQL injection. The parameters to be inserted and the results returned from the database are represented as `SqlValues`. To retrieve the values contained therein, the `SqlValues` can be converted from and to other Haskell types using the functions `fromSql` and `toSql` respectively. Database connections are represented as an instance of the `IConnection` type class. As the database access results in a communication with the “outside world”, the result is computed in the `IO` monad. In conclusion, a hypothetical function for row selection using parameters would result in the type:

```
select :: IConnection conn => conn -- the connection
      -> String                    -- the sql string
      -> [SqlValue]               -- the parameters
      -> IO [[SqlValue]]          -- the result
```

Given an instance of `IConnection` previously established, the function may then be used in the following manner:

```
select aConn "SELECT * FROM customers WHERE id = ?" [toSql 1]
```

4.2.1. Interface

To avoid the need for manually providing the connection, the functions for database access are provided by a type class called `MonadDB`, shown in listing 4.2.

4. Database Mapping

```
class MonadCatchIO m => MonadDB m where
  getConnection :: m HDBC.ConnWrapper
```

Listing 4.2: MonadDB type class for monadic database access

Every instance of `MonadDB` has to implement a function `getConnection`, which provides the connection used for database access. To avoid the need for higher order type inference, the connection is represented by the type `ConnWrapper`, which wraps other connections to reduce type complexity.

As the original functions provided by HDBC are computed in the `IO` monad, the desired type class requires each instance to be an instance of `MonadIO`, enabling the lifting of HDBC's functions into the target monad. Unfortunately, HDBC's function for wrapping database access into a transaction cannot be lifted, as it relies on the ability to catch exceptions in the `IO` monad. To provide database transactions, the ability to catch these exceptions is inevitable. Therefore, the type class in addition is required to provide an instance of the type class `MonadCatchIO` (cf. section 2.2).

It is worth noting that the usage of this type class does not restrict the application to use only one database. Depending on the monad instantiating the type class, it is perfectly possible to provide different connections to certain computations. For example, this could be done by using a `ReaderT` monad transformer in combination with the `local` function of the `MonadReader` type class. This allows the change of the connection locally for a single computation.

Using the introduced type class, the HDBC functionality needed for database access can be lifted as shown in listing 4.3.

```
sqlSelect :: MonadDB m => String -> [SqlValue] -> m [[SqlValue]]
sqlExecute :: MonadDB m => String -> [SqlValue] -> m Integer
sqlExecuteMany :: MonadDB m => String -> [[SqlValue]] -> m ()
commit :: MonadDB m => m ()
rollback :: MonadDB m => m ()
inTransaction :: MonadDB m => m a -> m a
```

Listing 4.3: Database access functions for MonadDB

Similar to the hypothetical function introduced above, the function `sqlSelect` is used for querying the database. The result list is entirely evaluated before being returned, to avoid potential problems caused by multiple querying in combination with lazy database access. The function `sqlExecute` is used for executing statements

manipulating the database, such as statements for inserting or updating values. The result value thereby denotes the number of the changed rows in the database. The function `sqlExecuteMany` behaves in a similar manner, but operates on a list of rows. As it internally re-uses the SQL statement, it is considered to be faster than the multiple usage of `sqlExecute`.

The functions `commit` as well as `rollback` provide access to managing database transactions. Finally, the function `inTransaction` takes a computation and wraps it into a new transaction. If the computation passed in succeeds (no exceptions are thrown), the changes made so far are committed to the database, while otherwise the transaction will be rolled back.

4.2.2. Exceptions

For database access there is a variety of exceptions which may occur, such as non-established connections, incorrect SQL statements or invalid conversions. To support the handling of these exceptions, the database mapping component contains an extensible hierarchy of exceptions as depicted in listing 4.4.

```
data SomeModelException
  = forall a . (Exception a) => SomeModelException a
data RecordNotFound      = RecordNotFound String
data SQLException        = SQLException String
data UnmarshalException = UnmarshalException String
```

Listing 4.4: Exceptions defined in the mapping component

The type `SomeModelException` builds the top of the hierarchy and subsumes the exceptions which may be thrown inside the mapping component. The hierarchy is located under the general root of exceptions, namely `SomeException`, and currently contains three specific exceptions:

- The `RecordNotFound` exception indicates that a specific database record could not be retrieved, although its existence was assumed. The `String` parameter is used to specify the desired record, for example by containing the respective SQL statement.
- The `UnmarshalException` is used in the context of converting the database representation into the types of the domain model. More precisely, it will be

4. Database Mapping

thrown if the result returned by the database is expected to be convertible into the desired type, but the conversion fails. The `String` again is used to specify the exact reason, like the desired type and the results returned by the database.

- The `SQLException` is a general exception thrown by JDBC if an error was encountered during database access. The string contains the underlying error as reported by JDBC.

4.3. SQL Construction

The functions introduced so far provide database access by means of SQL only. To facilitate the construction of commonly used SQL statements, a small domain specific language is provided. The language is based on an algebraic data type representing the SQL statements, as well as convenience functions for creating them. The statements to be supported can be divided into two groups:

- Statements for querying data (`SELECT`),
- Statements for manipulating data (`INSERT`, `UPDATE`, `DELETE`).

Both provide an algebraic data type representing the respective statements. As the type `Manipulation` of the latter category is rather trivial and expected to be rarely used by the user, its description should be skipped. Instead of that, the type `Query` for `SELECT` statements, as shown in listing 4.5, will be focused. Not only does this type provide more features, but also is it intended to be frequently used by developers.

```
type ColumnName = String
type TableName  = String

data Query = Select
  { option      :: Maybe String      -- ^ ALL, DISTINCT, ...
  , projection  :: [Projection]      -- ^ SELECT
  , tables      :: [TableName]       -- ^ FROM
  , grouping    :: [ColumnName]     -- ^ GROUP BY
  , having      :: Maybe Restriction -- ^ HAVING
  , criteria    :: Criteria          -- ^ WHERE, ORDER, ...
  }
```

Listing 4.5: Query type representing database queries

This type contains the common fields used for a single **SELECT** and follows the structure of SQL statements. Therefore, the purpose of the fields **option**, **tables** and **grouping** should be self-explanatory. The **projection** field contains a list of column **Projections** specifying the database columns to be returned, whereas an empty projection will return the entire row. A **Projection** consists of the name of the respective column and an optional aggregation operation to be performed on the values, as well as an optional alias for renaming the column. For example, a simple query for fetching the number of all customers in the database can be expressed as:

```
setProjection [rowCountP] $ setTables ["customers"] newSelect
==> SELECT COUNT(*) FROM customers;
```

The expression **rowCountP** is the projection of all rows to their quantity. The parts of a **SELECT** statement restricting the set of returned rows are combined in the type **Criteria**, depicted in listing 4.6.

```
data Criteria = Criteria
  { restriction :: Maybe Restriction -- ^ WHERE
  , ordering    :: [Order]          -- ^ ORDER BY
  , extra      :: [String]          -- ^ LIMIT, OFFSET, ...
  }

data Restriction
  = Simple      SimpleRestriction
  | NotExpr     Restriction
  | LogicExpr   LogicOp [Restriction]

data SimpleRestriction
  = UnExpr      UnOp          CompareValue
  | BinExpr     BinOp          CompareValue CompareValue
  | Between     CompareValue   CompareValue CompareValue
  | In          CompareValue   [SqlValue]

data LogicOp      = And | Or
data CompareValue = Col ColumnName | Val SqlValue
data UnOp         = IsNull | IsNotNull
data BinOp        = Equal  | NotEqual | ...
```

Listing 4.6: Criteria type for SQL construction

Besides the **Restriction**, the **Criteria** consists of a field for determining the order of the result, as well as a restriction based on the internal position of a row. The **ordering** combines a column name with a sorting direction (ascending, descending) like in `[asc "firstName", asc "lastName"]`. The **extra** part is used for providing

4. Database Mapping

optional arguments for limiting the amount of rows to be returned (e.g. "LIMIT 1"), or for specifying the first row to be returned (e.g. "OFFSET 10"). Both are typically used together for splitting the result into different pages (pagination).

The `Restriction` type allows the result set to be restricted by providing SQL predicates the rows have to satisfy. These predicates are expressed by functions on `CompareValues`, representing either a certain column of the database using the `Col` constructor or a parameter value indicated by `Val`. The values can be combined using functions like `eqExpr` for equality or `neExpr` for inequality, as well as their corresponding infix operators `(.==.)`, `(./=.)`, which are also provided. Furthermore, simple restrictions can be composed using the logic operators *and*, *or* and *not*.

For example, the following criteria restricts the result set to all customers where the values of the `id` is less than 5 and which are assigned to the category with the `id` 1:

```
setTables ["customers"]
$ modifyCriteria (setRestriction r) newSelect
where r = (col "id" .<. val 5) .&&.
         (col "category_id" .==. val 1)
==> SELECT * FROM customers WHERE id <= 5 AND category_id = 1;
```

To use the data types representing SQL statements for working with the database, the database access interface provides the two functions shown in listing 4.7.

```
querySelect :: MonadDB m => Query -> m [[SqlValue]]
executeManipulation :: MonadDB m => Manipulation -> m Integer
```

Listing 4.7: MonadDB functions for Criteria

The function `querySelect` takes a single `Query` and executes it against the database, whereas the function `executeManipulation` does the same for a `Manipulation`. Both functions first calculate the respective SQL string and collect the parameters used therein. In the SQL string the parameter positions are escaped using question marks, allowing the usage of the functions offered by `MonadDB`.

It is worth noting that the criteria type does not provide a way to construct type-safe statements. At querying, the statements will be converted into a plain SQL string and a list of `SqlValues`. Therefore, the result returned by a `Query` is of the type `[[SqlValue]]`. In conclusion, the main advantage is the simplified SQL construction, as mistyping will be encountered at compile time. Furthermore, a `Criteria` does not affect the structure of the returned result, so it becomes possible to re-use it when

the resulting type can be determined otherwise. This will be the case when functions for mapped domain model types are introduced.

4.4. Data Type Mapping

To map user defined data types to tables in the database, the information how to map the individual attributes of the type to the table columns is necessary. To provide this information, the database mapping contains a set of type classes, introduced beneath.

4.4.1. Basic Mapping

The type class `Persistent`, as shown in listing 4.8, is responsible for providing the basic mapping information for types without requiring the type to provide a unique identifier.

```
class Persistent p where
  persistentType :: p -> String
  tableName      :: p -> TableName
  tableColumns   :: p -> [ColumnName]
  toSqlList      :: p -> [SqlValue]
  fromSqlList    :: [SqlValue] -> p
  toSqlAL       :: p -> [(ColumnName, SqlValue)]
```

Listing 4.8: Persistent type class for database table mapping

The first three functions provide meta information about the type, instead of the value. Therefore, the results are expected to be independent of the values the functions are applied to. Consequently, it should be possible to obtain valid results by applying the functions to `undefined`. The function `persistentType` identifies the mapped type by returning the name of the type. Although not necessary for the persistence functions, the function is used to provide helpful information in case of erroneous conversions. The function `tableName` specifies the table of the database the type is mapped to, while the function `tableColumns` specifies the respective table columns.

The functions `toSqlList` and `fromSqlList` perform the actual conversion between the type and a list of `SqlValues`, which will typically be done using the `toSql` and

4. Database Mapping

`fromSql` conversion functions of HDBC for a single `SqlValue`. The list returned by `toSqlList` is expected to be of the same length as the list of table columns, as well as share the same order. The same applies to the list passed to `fromSqlList`, as the columns of a query result are calculated by `tableColumns`. To make this intrinsic relationship between the conversion functions more apparent, the type class additionally contains a function `toSqlAL`. This function returns an association list of table columns to `SqlValues`, instead of only a list of `SqlValues`. In consequence, `toSqlAL` realizes the same functionality as the combination of `fromSqlList` and `tableColumns`. Therefore, it is sufficient to provide just either of them.

To cite an example, listing 4.9 shows the mapping of the above mentioned type `Customer` to the according database table. Although in most cases all attributes will be mapped as in the example, this is not necessary. It is also perfectly possible to map just the subset needed in the particular application, for instance to store additional information in the type which is not mapped to the database.

```
instance Persistent Customer where
  persistentType _ = "Customer"
  tableName _     = "customers"
  fromSqlList (l0:l1:l2:l3:l4:l5:l6:[])
    = Customer (fromSql l0) (fromSql l1) (fromSql l2) (fromSql
      l3)
      (fromSql l4) (fromSql l5) (fromSql l6)
  fromSqlList _ = error "wrong list length"
  toSqlAL x = [ ("_id"           , toSql $ _id           x)
    , ("category_id"         , toSql $ categoryId x)
    , ("firstname"           , toSql $ firstname x)
    , ("initials"            , toSql $ initials x)
    , ("lastname"            , toSql $ lastname x)
    , ("date_of_birth"       , toSql $ dateOfBirth x)
    , ("updated_at"          , toSql $ updatedAt x)
  ]
```

Listing 4.9: Example table mapping

Although the conversions are expected to be safe, this cannot be guaranteed as the list length is variable in general. An erroneous conversion into a list of `SqlValues` will result in an illegal SQL statement and can therefore be detected. However, at the conversion *from* a list of `SqlValues`, exceptions may only contain little information non-specific to the conversion, such as “wrong list length”. To provide the user with additional information if an error occurred during conversion, exceptions thrown when converting from `[SqlValue]` are caught and re-thrown as an `UnmarshalException`.

This exception provides the target type and the list to be converted as additional information.

Based on the mapping information declared by `Persistent`, common functions for retrieving and altering can be derived, as shown in listing 4.10.

```

countPersistents      :: (MonadDB m, Persistent p)
                      => p -> Criteria -> m Integer
selectPersistents     :: (MonadDB m, Persistent p)
                      => Criteria -> m [p]
selectMaybePersistent :: (MonadDB m, Persistent p)
                      => Criteria -> m (Maybe p)
selectOnePersistent   :: (MonadDB m, Persistent p)
                      => Criteria -> m p
deletePersistentsByCriteria :: (MonadDB m, Persistent p)
                      => p -> Criteria -> m Integer
insertPersistent      :: (MonadDB m, Persistent p)
                      => p -> m Integer

```

Listing 4.10: Functions derived for Persistents

The function `countPersistents` takes a `Persistent` as well as a `Criteria` to retrieve the number of rows in the table satisfying the `Criteria`. As the `Persistent` parameter is only needed for calculation of the corresponding database table, the value itself is not necessary and may be `undefined`. For example, the number of all customers stored in the database can be retrieved by:

```
cCount <- countPersistents (undefined :: Customer) newCriteria
```

In contrast, `selectPersistents` retrieves the `Persistents` satisfying the `Criteria` themselves. If the restriction contained in the `Criteria` is supposed to limit the result to a single element, the retrieval is additionally supported by `selectMaybePersistent`. The result of this function is limited to one `Persistent`, even if more than one satisfies the restriction, which would lead to the first element being returned. If no element can be retrieved, then `Nothing` is returned.

Furthermore, it may be possible that the presence of at least one value is preconditioned. In these cases, the unwrapping of the result out of `Maybe` would be cumbersome and tempting into using the `fromJust` function. As the application of `fromJust Nothing` would lead to a general and therefore little helpful exception, this special task is focused by `selectOnePersistent`. In case of a missing value, a `RecordNotFound` exception is thrown, indicating the desired type as well as the

4. Database Mapping

given **Criteria**. Otherwise, the value is returned. However, this function reduces robustness, so its usage should be well considered.

Finally, the functions `deletePersistentByCriteria` and `insertPersistent` are used for deleting values with a **Criteria**, and inserting values respectively. In both cases, the result denotes the number of altered rows in the database.

4.4.2. Mapping with Identifiers

The functions introduced above focus on lists of **Persistents**, eventually restricted by a **Criteria**. To additionally be able to work with a single **Persistent**, it has to be uniquely identified. The presence of an identifier is denoted by providing an instance of the type class `WithPrimaryKey`. The type class as well as the derived functions are presented in listing 4.11.

```
type PrimaryKey = Integer

class Persistent p => WithPrimaryKey p where
  pkColumn      :: p -> ColumnName
  primaryKey    :: p -> PrimaryKey
  setPrimaryKey :: PrimaryKey -> p -> p

findByPKs      :: (MonadDB m, WithPrimaryKey p)
=> [PrimaryKey] -> m [p]
findOneByPK    :: (MonadDB m, WithPrimaryKey p)
=> PrimaryKey -> m p
findMaybeByPK :: (MonadDB m, WithPrimaryKey p)
=> PrimaryKey -> m (Maybe p)
insertByPK     :: (MonadDB m, WithPrimaryKey p) => p -> m p
updateByPK     :: (MonadDB m, WithPrimaryKey p) => p -> m Bool
deleteByPK     :: (MonadDB m, WithPrimaryKey p) => p -> m Bool
```

Listing 4.11: `WithPrimaryKey` type class for providing identifiers

Instances of the class are expected to be identified by an **Integer** value. For identifier retrieval, the type has to provide the value of the identifier as well as the name of the identifier column. The `setPrimaryKey` is used to update the key after insertion, as described below in more detail. In consequence, the instance declaration for a customer would look like:

```
instance WithPrimaryKey Customer where
  primaryKey = _id
  pkColumn  = head . tableColumns
  setPrimaryKey pk c = c {_id = pk}
```

The derived functions beginning with the prefix `find` follow the same principle as the functions introduced for `Persistents`, with the main difference that for restriction they use the primary key instead of a `Criteria`. As the values can be uniquely identified, it now becomes possible to update and delete single values. In both cases, the result indicates whether a change to the database has taken place.

For insertion, the `insertByPK` functions considers the attributes only, but not the identifier, as new primary keys are expected to be provided by the database after insertion. Therefore, the database is queried for the newly assigned key, which is then used to update the initial value to be inserted via `setPrimaryKey`. The inserted and altered value is returned afterwards.

4.4.3. Model Interface

The type classes introduced so far are sufficient for the major requirements in persisting data types into a database. However, they are designed to provide the necessary functions by a minimal set of information. This naturally limits the flexibility, as there are only small possibilities to change the default behaviour. For example, it may be necessary to update a field containing a time stamp every time a value is inserted or updated. Although this may be realized by introducing additional functions, this would lead to confusing code as every type may then provide a different update function. Therefore, the interface for data types persisted in a database is consolidated by the type class `Model`, shown in listing 4.12.

```
class WithPrimaryKey a => Model a where
  new          :: MonadDB m => m a
  count        :: MonadDB m => a -> Criteria -> m Integer
  select       :: MonadDB m => Criteria -> m [a]
  selectOne    :: MonadDB m => Criteria -> m a
  selectMaybe :: MonadDB m => Criteria -> m (Maybe a)
  deleteByCriteria :: MonadDB m => a -> Criteria -> m Integer
  find         :: MonadDB m => [PrimaryKey] -> m [a]
  findOne      :: MonadDB m => PrimaryKey -> m a
  findMaybe   :: MonadDB m => PrimaryKey -> m (Maybe a)
  delete       :: MonadDB m => a -> m Bool
  insert       :: MonadDB m => a -> m a
  update       :: MonadDB m => a -> m a
```

Listing 4.12: Model type class providing a uniform interface

4. Database Mapping

This class provides a default implementation which is basically a re-export of the functionality provided by the type classes introduced above. In addition, the member function `new` is added to provide a uniform way to create new values. The particular attributes of the value to be created may be initialized to arbitrary values, appropriate to the respective domain. This also applies to the primary key, which is updated at insertion anyway. Furthermore, the functions changing the database, such as `delete` or `insert`, are executed inside a transaction, providing per-operation-transaction. However, this behaviour might change in the future with a more elaborated transaction support. The different functions wrapped in a transaction are also accessible and can be used by the developer, such as

```
updateInTransaction p = inTransaction $ updateByPK p >>
  return p
```

Instances of the type class are allowed to overwrite the default implementations. For example, refreshing a time stamp of a customer before updating is possible via:

```
instance Record Customer where
  update c = do
    now <- liftIO getCurrentTime
    updateInTransaction $ c { updatedAt = Just now }
```

4.5. Relationship Mapping

The concept of a foreign key relationship between tables in a database can be transferred to a relationship between the corresponding types the tables are mapped to. In Haskell, a relationship between two types can be expressed as a type class involving the mapped types. As the referenced type has to be uniquely identified, it has to provide an instance of `WithPrimaryKey`. On the contrary, for the referencing type an instance of `Persistent` is sufficient. In consequence, a first approach might lead to

```
class (Persistent referencing, WithPrimaryKey referenced) =>
  Relationship referencing referenced where ...
```

Unfortunately, this approach restricts the two incorporated types to share only one relationship, as there is only one type class instance allowed for each combination. However, the ability to express multiple relationships between two types is desirable,

so a relationship has to be expressed using an additional third type. The purpose of this type is the determination of the relationship only, which allows it to be an arbitrary type, although a new data declaration indicating the purpose is more adequate. As this new type denotes a relationship between the two other types, these can be said to be associated to the type determining the relationship. This fact can be expressed in an elegant way using an *associated type family* (cf. section 2.1). The type class for denoting relationships is shown in listing 4.13.

```

type ForeignKey = Maybe Integer

class (Persistent (Referencing r),
      WithPrimaryKey (Referenced r)) => ForeignKeyRelationship r
  where
    type Referencing r :: *
    type Referenced r  :: *
    relationshipName   :: r -> String
    fkColumn           :: r -> ColumnName
    foreignKey         :: r -> Referencing r -> ForeignKey
    setForeignKey      :: r -> ForeignKey -> Referencing r ->
                        Referencing r

```

Listing 4.13: ForeignKeyRelationship type class for representing relationships

The associated type synonym `Referencing r` denotes the type referencing another type, which is in turn denoted by `Referenced r`. In the database, the table of the former type has a foreign key relationship to the table of the latter type. Like the function `persistentType` of the class `Persistent`, the function `relationshipName` is used to identify the relationship. Again, this is used for providing additional information in case of error conditions. While the function `fkColumn` determines the table column containing the foreign key, the functions `foreignKey` and `setForeignKey` are used to manipulate the relationship.

For example, the relationship between customers and categories can be expressed via:

```

data Customer2Category = Customer2Category

instance ForeignKeyRelationship Customer2Category where
  type Referencing Customer2Category = Customer
  type Referenced  Customer2Category = Category
  relationshipName _ = "customer2category"
  fkColumn _       = "category_id"
  foreignKey _     = categoryId
  setForeignKey _ c key = c {categoryId = key}

```

4. Database Mapping

Similar to the type classes introduced before, functions for retrieving the values involved in a certain relationship are derived. As the functions in their general form will not provide additional insights, they are omitted at this place. In fact, these functions are further adjusted to reflect specific kinds of relationships, which all share the same database implementation of foreign keys. These relationship kinds are distinguished based on their cardinality, such as a one-to-one or a one-to-many relationship.

As the different kinds of relationships require different sets of functions, they are denoted by additional type classes offering the particular functions. More precisely, the type classes are oriented on a specific *direction* of a relationship. For example, the one-to-one and the one-to-many share a common “fromOne” direction. That is, for both relationships the referencing part is considered to *belong to* one referenced value. In the opposite direction, a value may be referenced by *one* or *many* value. Consequently, this leads to three type classes: **BelongsTo**, **HasOne** and **HasMany**. The referencing value (the table containing the foreign key) is always mapped to a **BelongsTo** relationship. In contrast, a referenced value may be referenced by one (**HasOne**) or by many (**HasMany**) other values.

To cite an example, listing 4.14 presents the functions offered by the type class **BelongsTo**.

```
class ForeignKeyRelationship r => BelongsTo r where
  getMaybeParent :: MonadDB m => r -> Referencing r
  -> m (Maybe (Referenced r))
  getParent      :: MonadDB m => r -> Referencing r
  -> m (Referenced r)
  setParent      :: MonadDB m => r -> Referencing r
  -> (Maybe (Referenced r)) -> m (Referencing r)
```

Listing 4.14: **BelongsTo** type class for relationship access of referenced types

These functions can be used to express “one-to-*” relationships. Although they are used for changing the relationship by adding or removing elements, the functions do not automatically lead to changes in the database. This allows the developer to perform additional changes before updating the database, and is considered more flexible as the developer can decide when to perform changes. Similar to the **Model** type class, it becomes possible to change the default behaviour by overwriting selected functions.

Furthermore, as the functions have unspecific names, it is reasonable to provide alias functions expressing the relationship they operate on, such as:

```
instance HasMany Customer2Category

getCategory :: MonadDB m => Customer -> m Category
getCategory = getParent Customer2Category
```

4.6. Validation

Depending on the specific application, the types used in the domain model of the application may have to satisfy certain constraints. The process of checking whether or not these constraints are satisfied is known as *validation* [BBP09, Red08]. Constraints in general can be expressed as predicates of the form `a -> Bool` for a certain type `a`. However, as constraints are expected to be combined to form composite constraints, there arises the need for an additional context in which the single constraints are validated. This context has to provide a mechanism to announce constraint violations, as well as collect such violations. In Hawk, this context is provided in form of the `ValidatorT` monad transformer for validations, illustrated in listing 4.15.

```
type ValidationError = String
type AttributeName  = String
type ValidationErrors = [(AttributeName, ValidationError)]

newtype ValidatorT m a
  = ValidatorT { unwrap :: WriterT ValidationErrors m a }
deriving (Functor, Monad, MonadIO, MonadTrans, MonadWriter
  ValidationErrors)

addError      :: Monad m
  => AttributeName -> ValidationError -> ValidatorT m ()
silent        :: Monad m
  => ValidatorT m a -> ValidatorT m a
execValidatorT :: Monad m
  => ValidatorT m a -> m ValidationErrors
runValidatorT  :: Monad m
  => ValidatorT m a -> m (a, ValidationErrors)

class Validatable v where
  validator :: MonadDB m => v -> ValidatorT m ()
```

Listing 4.15: `ValidatorT` monad transformer for validation

4. Database Mapping

The monad transformer contains two type parameters, the monad the violation should be executed in, as well as an arbitrary result type. Inside the applied monad transformer, constraint violations, or *validation errors*, can be announced using the function `addError`, which assigns the violation description to the respective attribute to be validated. The `AttributeName` denotes the particular part in which the violation occurred, for example a field in a record. An empty string may be used to denote validation errors regarding non-structured values as well as errors regarding a composite value in general. The function `silent` can be used to retrieve the result of a single validation while discarding possible error messages. This function therefore allows the composition of basic validations while providing a single error message.

To express that a certain type can be validated using the validation monad, it has to provide an instance of the type class `Validatable`, which requires an implementation of a `validator` function. The result of the `validator`, the collected errors, can be accessed by the functions `execValidatorT` or `runValidatorT` in form of an association list of `AttributeNames` to `ValidationErrors`. These functions un-wrap the monad transformer and return the collected results, whereas the second function additionally returns the results of the monadic computation.

Based upon these basic functions, there are predefined validation functions for common tasks such as list length validation as well as combinator functions, of which a subset is shown in listing 4.16.

```
check      :: Monad m => Bool -> ValidationError
  -> AttributeName -> ValidatorT m Bool
validate :: Monad m => (a -> Bool) -> ValidationError
  -> AttributeName -> a -> ValidatorT m Bool

validatePast      :: MonadIO m
  => AttributeName -> UTCTime -> ValidatorT m Bool
validateNotNull :: Monad m
  => AttributeName -> [a] -> ValidatorT m Bool
```

Listing 4.16: Selected predefined validation functions

Validation functions in general are expected to not only propagate errors, but also to return whether the validation was successful. The first function `check` tests whether a boolean value is true and announces a validation error if it is not, providing a simple way of constraint checking. On the contrary, the function `validate` can be used to construct more specific validation functions by partial application. An

example is given by `validatePast`, which validates if a given date is in the past. As `ValidatorT` is a monad transformer, is it possible to access functions offered by the underlying monad, such as the `IO` monad which is used for retrieving the current date for comparison. Finally, combinator functions like `(<&>)` and `(<?>)` allow the construction of composite validation functions for a single attribute. While the former applies both validations on the values passed in, the latter only applies the second validation if the first was successful.

For an example of validations, the validator checking that each customer has non-empty first and last names, as well as a date of birth in the past, can be written as:

```
instance Validatable Customer where
  validator c = do
    validateNotNull "firstName"    $ firstName    c
    validateNotNull "lastName"     $ lastName     c
    validatePast      "dateOfBirth" $ dateOfBirth c
```

4.7. Attribute Updates

The domain model of a web application typically is altered by means of the information passed in with the user request. More precisely, the different attributes of a type of the domain model have to be updated by the different parameters. The parameters contained in a HTTP request are encoded as strings and made accessible via a `Map String String`, so they first have to be converted into the appropriate types before updating the attributes. In addition, the parameters may be erroneous; therefore it cannot be guaranteed that they contain valid information only. In consequence, the update process has to be aware of possible conversion errors.

As a first approach, it might be promising to extract and convert the parameters first and update the model type afterwards. However, as a model type may be updated in different subsets of its attributes, this would lead to duplicated code. Therefore, the process is inverted: The parameters are supplied to the model type as a whole, and each attribute with a present parameter is updated. To denote the ability of a type to be updated by parameters, it has to provide an instance of the type class `Updateable`, shown in listing [4.17](#).

```
class Updateable u where
  updater :: MonadDB m => u -> String -> UpdaterT m u
  updater u _ = return u
```

Listing 4.17: Updateable class for enabling attribute updates

The first parameter of the `updater` function is the single value to be updated and afterwards returned, of course in the updated form. The second parameter denotes an optional prefix; it can be used to access the parameters if they are represented in the `Map` in a hierarchic structure. For example, in some circumstances it might be necessary to access the first name of a person by the key `"firstname"` as provided by one form, while a different form provides the same information using the key `"customer.firstname"`. Whereas in the former case the person may be updated by `updater p ""`, the latter requires an additional prefix in the form of `updater p "customer"`. In conclusion, this allows structured information to be expressed in a non-structured map like illustrated in figure 4.3.

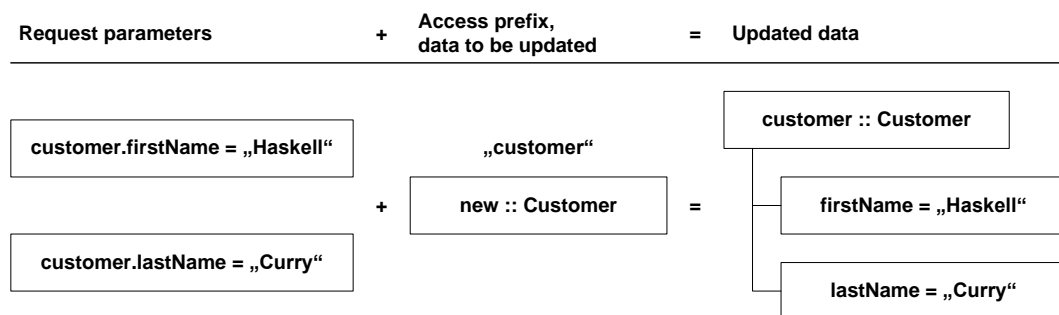


Figure 4.3.: Simulation of hierarchic parameters

The parameters for updating are provided by the `UpdaterT` monad transformer, presented in listing 4.18. The access to the parameter map is realized using the `ask` function of `MonadReader`. As the update process includes the potential erroneous parsing of strings, errors can be announced using the functionality inherited by the included `ValidatorT` monad transformer mentioned before. If there is no parameter present for a certain attribute of a value to be updated, the value is expected to remain unaltered. In general, missing parameters are not considered as errors to allow the updating of attribute subsets. If a parameter is present though, any conversion error should be announced while the value should remain unaltered.

```

type Params = M.Map AttributeName String

newtype UpdaterT m a
  = UpdaterT { unwrap :: ReaderT Params (ValidatorT m) a }
deriving (Functor, Monad, MonadIO, MonadWriter ValidationErrors
         , MonadReader Params)

updateByParams :: (MonadDB m, Updateable u)
  => u -> String -> Params -> m (u, ValidationErrors)
updateAndValidate :: (MonadDB m, Updateable u, Validatable u)
  => u -> String -> Params -> m (u, ValidationErrors)

```

Listing 4.18: UpdaterT monad transformer for attribute updates

The update can be executed by using either `updateByParams` or `updateAndValidate`. Both expect the value to be updated, as well as the parameter map. Although the `Map` is expected to be provided by the respective HTTP request, it is of course possible to manually provide the values. In addition to the update, the `updateAndValidate` function also validates the value after the update, combining both lists of error messages. Currently, there are some instance declarations for frequently used type such as `String` or `Integer`, so their `updater` function can be re-used.

To conclude with an example, a customer may be updated in the components of his name and his date of birth:

```

instance Updateable Customer where
  update c key = do
    f <- updater (firstName  c) $ subParam key "firstname"
    l <- updater (lastName   c) $ subParam key "lastname"
    b <- updater (dateOfBirth c) $ subParam key "dateofbirth"
    return $ c { firstName = f, lastName = l, dateOfBirth = b }

```

The helper function `subParam` is used to construct the keys for the sub parameters by eventually inserting a period. As the timestamp as well as the relationship are expected to be altered differently, they are excluded.

4.8. Summary

With the aid of the type classes as well as the functions introduced, all common tasks of the data access foreseen for a web application can be implemented. While HDBC as the fundamental database access library is still accessible, `MonadDB` allows the

4. Database Mapping

lifted functions to be used in arbitrary monads fulfilling the respective preconditions. The `Criteria` in addition facilitates the construction of common database queries, as well as simplifies the type class functions for mapped types. By means of the various type classes, it is possible to gain the desired functionality though preserving the flexibility of custom behaviour. Finally, the `ValidatorT` as well as the `UpdaterT` in combination with the respective type classes focus on the tasks of validation and failure-aware altering of model data, which furthermore enhances the usability of the database mapping. To conclude with, figure 4.4 provides an overview of the type classes for mapping introduced in this section, as well as their relationships.

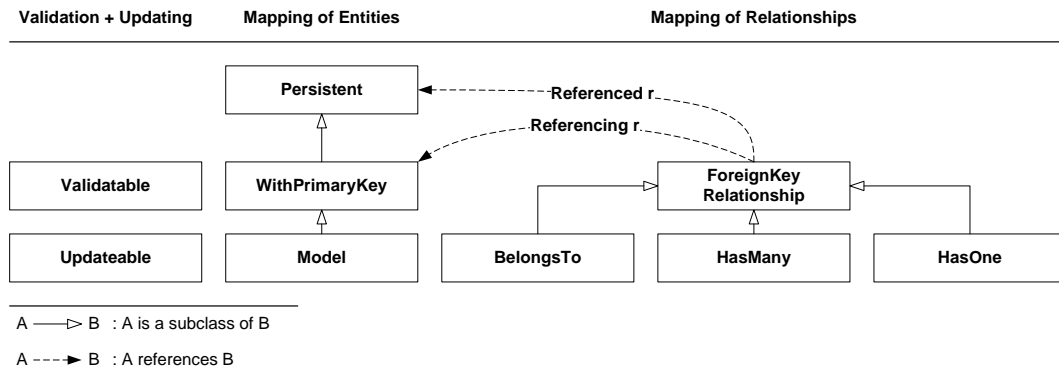


Figure 4.4.: Type classes for database mapping

5

Controller

The controller component constitutes the core of the Hawk framework, as it provides the implementation environment for actions and integrates the different components into a complete application. This chapter begins with a description of the basic types used in the controller, before the features provided for action implementation are discussed. Subsequently, the integration of the actions into the front controller is illustrated. The chapter concludes with a description of the configuration needed to create a complete application, and depicts a possible way to integrate the application into a simple web server.

5.1. Basic Types

As already mentioned in chapter 3, the controller component is divided into the front controller and different actions. The *front controller* is responsible for the initial request handling and is invoked by the Hack handler. Based upon the incoming request, it decides which action should be invoked, and provides the environment necessary for the request processing. The *action* then handles the request, realizing

5. Controller

the actual processing logic. The result of the action, which may be of an arbitrary type, is passed to a view capable of rendering the type passed in. Finally, the result of the view is returned to the front controller, which converts it into a complete response. The relationships between the different parts of the controller component are depicted in figure 5.1, as well as the fundamental types, which will be described next.

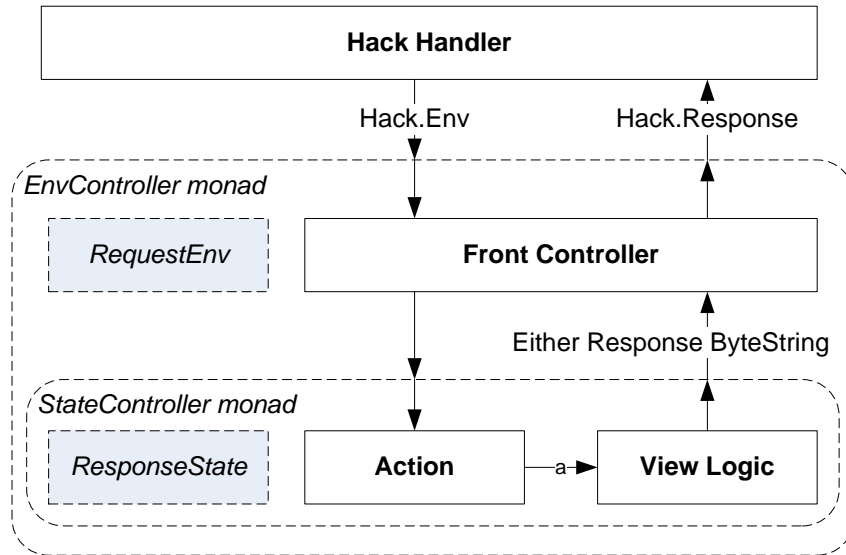


Figure 5.1.: Types of the Hawk controller

5.1.1. Hack Interface

The Hack types constitute the interface of the application as a whole to a Hack handler, allowing the combination of a web application with multiple web serving strategies. The main advantage of this interface is its simplicity, as it only defines two types. These types, representing an HTTP request and an HTTP response, are shown in listing 5.1.

```
data Env = Env
  { requestMethod :: RequestMethod
  , scriptName    :: String
  , pathInfo      :: String
  , queryString   :: String
  , serverName    :: String
  , serverPort    :: Int
  , http          :: [(String, String)]
```

```

    , hackUrlScheme :: Hack_UrlScheme
    , hackInput      :: ByteString
    -- other, hack-specific fields ...
  } deriving (Show)

data Response = Response
  { status    :: Int
  , headers   :: [(String, String)]
  , body      :: ByteString
  } deriving (Show)

```

Listing 5.1: Hack interface representing HTTP requests and responses

The data type `Env` represents an incoming HTTP request. The field `requestMethod` contains the request method, such as GET or POST, while the URL included in the request is already split into several parts. The server part of the URL is contained in the fields `serverName` and `serverPort`, while the remaining part, the path within the server, is even further divided. The part `scriptName` contains the virtual location of the application on the server, which in principle allows the serving of multiple applications in one server application, using only one port. If the application is located in the root path of the server, this `String` will be empty. The `pathInfo` contains the request part inside this virtual location, in this context the action to be performed. If the request additionally contains request parameters included in the URL, these are contained in the `queryString`. Finally, the field `hackUrlScheme` contains the requested protocol (either HTTP or HTTPS), whereas the request body is provided in the field `hackInput`. The HTTP headers delivered with the request are given in `hackHeaders`, containing additional information such as the accepted encodings or the name of the web browser.

If, for example, a web application for managing customers is served at `localhost:3000/myapp`, then a possible request for showing a single customer may look like:

```
GET http://localhost:3000/myapp/customer/show?id=1
```

According to the information given above, this request is represented as the following `Env` value:

```

Env { requestMethod = GET, scriptName = "/myapp"
    , pathInfo = "/customer/show", queryString = "id=1"
    , serverName = "localhost", serverPort = 3000
    , http = [{- some headers -}]
    , hackUrlScheme = HTTP, hackInput = empty, ...
  }

```

5. Controller

The counterpart of the `Env` is the `Response`. This type represents an HTTP response and consists of three different fields. The `status` field indicates the status of the request handling, for example a regular result (status 200, OK) or a request for a non-existent resource (status 404, not found). The `headers` contain additional information of the response, such as the encoding of the result. Finally, the `body` contains the actual answer sent to the user, such as an HTML page to be displayed. For example, the answer to the request mentioned above could look like:

```
Response { status = 200, headers = [{- some headers -}]
          , body   = {- html page showing the customer -}
          }
```

5.1.2. EnvController Monad

Beside the request provided by the Hack handler, a web application has to operate on additional information, such as the database connection. Furthermore, the application may rely on several configuration options, necessary in different parts of the application. Although it would be possible to pass all this information into the respective functions as parameters, this would be very cumbersome. Therefore, this information is provided in form of a monad, responsible for encapsulating the information passing. This monad `EnvController` and the contained information data type are shown in listing 5.2.

```
data RequestEnv = RequestEnv
  { databaseConnection :: ConnWrapper
  , configuration      :: AppConfiguration
  , request            :: Hack.Env
  , environmentOptions :: [(String, String)]
  }

newtype EnvController a
  = EnvController { runController :: ReaderT ResponseEnv IO a }
  deriving (Functor, Monad, MonadIO, MonadCatchIO, MonadReader ResponseEnv)

instance MonadDB EnvController where
  getConnection = asks databaseConnection
```

Listing 5.2: `EnvController` monad providing the environment for request handling

The type `RequestEnv` subsumes the information necessary to handle a request. As the first field, the `databaseConnection` denotes the database containing the data of the domain model. As the monad provides an instance of `MonadDB`, both the actions and views can access the model component, providing the database access.

The `AppConfiguration` forms a substantial part of the `RequestEnv`, as it not only configures particular aspects of the front controller, but also is used for integrating the actions and views into the front controller. This type will be discussed in its entire form later, as it deals with different aspects and is therefore subsequently introduced.

The `environmentOptions` provide the ability to add custom application options later without requiring a change of the data structure. They may in addition be used by the developer, as well. Finally, the passed-in request is put into the `RequestEnv`. Although it would have been possible to explicitly provide the request as a parameter, the monadic access facilitates the combination of functions, as the request is automatically provided.

The `RequestEnv` is accessible by means of the `ReaderT` monad transformer, allowing the environment to be accessed by the `ask` function. Beside the above mentioned instance of `MonadDB`, it also offers instances of type classes allowing I/O actions as well as exception handling.

5.1.3. StateController Monad

In principle, a web application can be implemented using the above mentioned `EnvController` monad, as this provides full access to the request as well as the environment. Nevertheless, for developing larger applications, there arises the need for a state shared between different actions. A typical example is the need to share information between subsequent requests, also known as a session. For instance, this may be necessary at a web shop to allow the customer to subsequently fill his cart.

Furthermore, actions may not only provide the actual return type, but also provide additional information to the view, such as a status message. An explicit passing of such messages complicates the combination of actions and views, especially if the messages are specific to a certain view.

5. Controller

Therefore, these issues are addressed by a monad specific for the implementation of actions and views, offering the required features by an internal state. The type of this state and the corresponding monad are depicted in listing 5.3.

```
data ResponseState = ResponseState
  { session      :: Session
  , cookies      :: [Cookie]
  , responseHeaders :: Map String String
  , flash        :: Map String String
  , errors       :: Map String [(String, String)]
  }

type StateController = EitherT Response (StateT ResponseState
  EnvController)
```

Listing 5.3: StateController monad providing a state for response generation

The `ResponseState` is the state shared between actions and views, consisting of five fields. The session basically provides the functionality of a `Map String String`, but is automatically shared between different requests. The cookies furthermore allow the developer to store small portions of data in the client's browser manually, without the need for an entire `Session`. Both cookies and sessions will be introduced in detail later.

The `responseHeaders` allow the developer to set additional headers of the response inside an action. These headers will be included into the final response afterwards. Finally, the `flash` and `errors` maps allow the actions to provide additional information to the view. While the former is intended for short status messages, the latter should be used for providing validation errors. The access and usage of these fields will be described in more detail in the following section as well.

The state is provided to the actions and view by the `StateController`, realized as a `StateT` transformer applied to the `EnvController`. In consequence, the functionality of the `EnvController` is also made accessible. Furthermore, an `EitherT Response` transformer is applied on top, allowing it to directly return a response. Usually, an action returns an intermediary result to be rendered by a view, which converts it to the body of the response. The body is then extended to a regular response with the status 200 and returned to the browser. However, some actions may require the ability to compute the response directly, for example to provide a redirect to another action, or to indicate an error, such as a non-existent resource (status 404, not found). This is expressed using the `EitherT` monad transformer. In conclusion,

the actions and views can be focused to handle the normal case only, whereas other constellations are handled internally.

The working principle of the `EitherT` monad is additionally depicted in figure 5.2. In this illustration, the `Right` value can be associated to the regular result, such as the HTML page for showing a customer. In contrast, the `Left` value can be associated to a special response, such as a response denoting an invalid request parameter, resulting in a “page not found” result.

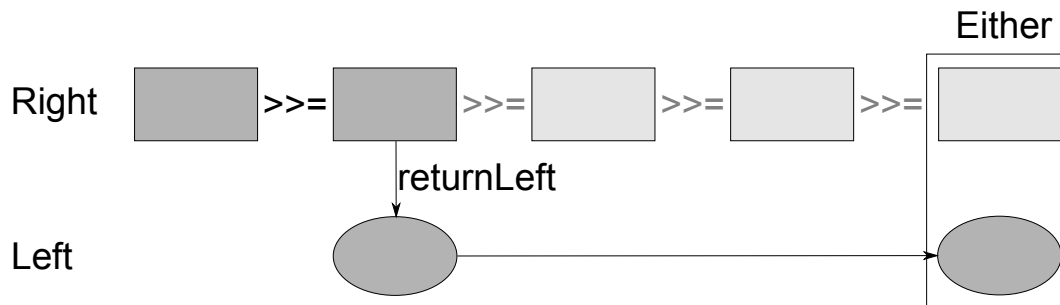


Figure 5.2.: Working principle of the `EitherT` monad

5.1.4. Actions and Views

As mentioned above, the `EitherT` transformer allows two kinds of return types. First, complete responses can be provided using special functions, such as for redirects. In contrast, regular results will be converted in a response with the status code 200 (ok). Therefore, the rendered action only has to provide the body of the response, which leads to the type:

```
type Controller = StateController ByteString
```

The term `Controller` refers to the fact that the action and the view are already combined. Prior to the combination, the actions are free to return arbitrary types, whereas the views specify the type they can render. In consequence, the type of a single action in general is:

```
anAction :: StateController a
```

To be able to be rendered, a corresponding view is required. As views may realize multiple formats such as XHTML or plain text, they cannot be represented as a

5. Controller

single type. Instead, they are denoted by a type class `View`, shown in listing 5.4. The listing also shows the instance declaration for plain text.

```
{-# LANGUAGE TypeFamilies #-}
class View a where
  type Target a :: *
  render :: a -> Target a -> StateController ByteString

data TextView a = TextView
  { toText :: a -> StateController String }

instance View (TextView a) where
  type Target (TextView a) = a
  render tv = liftM fromString . toText tv
```

Listing 5.4: View type class for rendering and its instance `TextView` for rendering plain text

While different formats are expected to be realized by different data types, they are consolidated using the type class `View`. This class requires its instances to denote the type of input they can render into a response, by means of the associated type synonym `Target a`. Furthermore, the instances have to provide a function `render`, implementing the actual rendering process.

Below the type class, an exemplary instance is provided in form of the `TextView`. A `TextView` converts its input into a `String`, offering a default implementation for `Show` instances. While the function `toText` can be used to customize the conversion of the input into a `String`, the instance declaration is responsible for the conversion into a `ByteString`.

To cite an example, an action retrieving a single customer may be combined with a text view for showing it, based on the `show` function of the `Customer` type. This may be realized as:

```
showController :: Controller
showController = showAction >>= render showText

showAction :: StateController Customer
showAction = {- code for retrieval -}

showText :: TextView Customer
showText = TextView show -- simple rendering
```

5.2. Action Implementation

While the previous section introduced the fundamental types of the controller component, this section focuses on the implementation of actions. Although the implementation may directly access the monad, there is a variety of functions already provided, hiding the complexity from the developer.

5.2.1. Request Access

The information passed in with the request can be accessed by a variety of functions. For example, the method of a request (GET, POST, ...) may be accessed using the function `getRequestMethod`:

```
getRequestMethod :: StateController RequestMethod
```

This may be useful to separate the processing of a GET request for showing a form for editing from the processing of the POST request, which will contain the changes. Like the request method, the other fields are also accessible by comparable functions. Furthermore, the access to the request parameters is further simplified. Although they are provided in the request body or in the query string, depending on the request method, the parameters can be accessed by uniform functions, shown in listing 5.5.

```
getParam      :: String -> StateController String
lookupParam  :: String -> StateController (Maybe String)
readParam     :: Read v => String -> StateController (Maybe v)
```

Listing 5.5: Functions for accessing the request parameters

While `getParam` retrieves either the parameter or an empty `String`, the function `lookupParam` additionally provides the information if the parameter is present. Although parameters are represented as `Strings`, they often contain the `String` representation of another type. This type can be retrieved back using the function `readParam`, whereas the result `Nothing` denotes either a missing parameter or a failed conversion.

In conclusion, the retrieval of a single customer can now be implemented as:

5. Controller

```
showAction :: StateController Customer
showAction = do
  customer <- readParam "id" >>= liftMaybe findMaybe
  case customer of
    Nothing -> error404response
    Just c   -> return c
  where liftMaybe = maybe (return Nothing)
```

5.2.2. Redirects and Error Responses

The regular results computed in the actions represent the normal response content, which is automatically wrapped into a response after rendering. Beside these results, it is sometimes necessary to provide additional responses, like for the following cases:

- The user should be redirected to another URL
- An error was detected and should be indicated

These cases are handled by functions for returning specific functions, shown in listing 5.6, which internally rely on the `EitherT` functionality described above.

```
redirectToUrl      :: String -> StateController a
error404Response  :: StateController a
error500Response  :: StateController a
customResponse    :: Response -> StateController a
```

Listing 5.6: Functions for returning alternative responses

The `redirectToUrl` function is used for delivering the URL, which will subsequently handle the current request, to the browser. The browser then sends a new request to the specified address afterwards. For example, this may be useful to redirect the user to a page showing a customer he just edited. Because of the redirect, the display page is then associated to the URL for showing customers, instead of to the URL for editing customers. This is an often used pattern to prevent the repeated invocation of the editing action. For example, without the redirect, the user may have bookmarked the result page under the edit URL, leading to an edit action although the displaying was intended.

Furthermore, errors can be denoted using the function `error404Response` and `error500response`, resulting in the corresponding status codes. Finally, the developer is also free to provide custom responses using the respective function.

Whenever one of the above mentioned functions is used, the following computations will be by-passed, as well as the view. This can be compared to a return statement in imperative languages, which provides multiple exit points of a function or procedure. In consequence, the developer can check some preconditions first and eventually return a redirect or an error response if these are not fulfilled, like for a missing parameter. The subsequent code can then handle the normal case. As this part does not have to deal with errors, it requires less expression nesting, resulting in cleaner code.

5.2.3. Cookies

Cookies [RFC2109] are key-value pairs of strings, which are saved in the client's browser and transferred inside the HTTP headers. As cookies are sent to the application with incoming requests, they can be used to share common data between different requests. While the HTTP protocol itself is stateless, cookies allow the sharing of small data portions to create a common context.

```
-- definition from Network.CGI.Cookie
data Cookie = Cookie
  { cookieName      :: String
  , cookieValue     :: String
  , cookieExpires   :: Maybe CalendarTime
  , cookieDomain    :: Maybe String
  , cookiePath      :: Maybe String
  , cookieSecure    :: Bool
  } deriving (Show, Read, Eq, Ord)

-- functions for working with cookies
newCookie      :: String -> String -> Cookie
getCookie      :: String -> StateController (Maybe Cookie)
setCookie      :: Cookie -> StateController ()
deleteCookie   :: String -> StateController ()
getCookieValue :: String -> StateController (Maybe String)
setCookieValue :: String -> String -> StateController ()
```

Listing 5.7: Cookie data type and access functions

5. Controller

Cookies in Hawk are based on the cookie implementation of the CGI library [Bri]. A `Cookie` (cf. listing 5.7) contains a name for its identification and the associated value, both of type `String`. In addition, it can be modified to expire after a specific amount of time, set via the `cookieExpires`. An expired cookie is discarded by the browser and therefore no longer contained in the requests.

Furthermore, a cookie can be restricted to be sent only if the URL of the request matches a specific domain and a specific path. More precisely, the cookie is only sent if the `cookieDomain` matches the tail of the request domain, and the `cookiePath` matches the beginning of the request path. Finally, a cookie can be restricted to be sent by the browser only if the connection is a secure HTTPS connection (`cookieSecure`).

To ease the access of the cookies inside actions, there are several functions encapsulating the cookie handling. These functions can be used to retrieve (`getCookie`, `getCookieValue`) and to alter (`setCookie`, `setCookieValue`) the values stored in a cookie. While the functions suffixed by `Value` directly operate on the `String` value, the other two functions operate on the cookie type. These are for example needed to set additional properties of the cookie, such as the security restriction. This may be done with:

```
setCookie $ newCookie "key" "secret" {cookieSecure = Just True}
```

Although cookies are an established and widely used technique, they also have some disadvantages. Cookies are not only limited in size (a maximum of 4 KByte), but also are transported unencrypted. Therefore, they can in principle be changed by the user. In consequence, it is not recommended to use them for sensitive information, such as login information. Finally, cookies may also be disabled by web browsers.

5.2.4. Session

Similar to the cookie interface introduced above, the session is used to store string values under a certain name and to share them between different requests. In contrast to cookies, sessions do not suffer from the size limitations of cookies and provide a higher level of security as their content cannot be easily changed. Consequently, the session is considered superior for sharing state information between different parts of the application. The session can be accessed via the functions shown in listing 5.8.

```

getSessionValue  :: String -> StateController (Maybe String)
setSessionValue  :: String -> String -> StateController ()
readSessionValue :: Read v
                  => String -> StateController (Maybe v)
deleteSessionKey :: String -> StateController ()
clearSession     :: StateController ()

setSessionExpiry :: Maybe UTCTime -> StateController ()
getSessionExpiry  :: StateController (Maybe UTCTime)
sessionExpired    :: StateController Bool

```

Listing 5.8: Functions for accessing the session

The first group of functions is used to store values in the session, as well as to retrieve or delete them. The function `clearSession` resets the entire session, resulting in all values to be deleted. Like cookies, sessions in addition may contain an expiry timestamp, which can be modified and retrieved using the second group of functions.

To share sessions between different requests, they have to be stored in the meantime and loaded for subsequent requests. In general, sessions can be stored in different locations such as in a database, in the file system or even on the client's browser. Therefore, the session storage has to be configurable. Session stores are represented by the type `SessionStore`, which allows the implementation of different storage strategies. The session store has to be provided in the application configuration, in combination with options specific for the particular store. The type of session stores and the internal representation of sessions are shown in listing 5.9.

```

data Session = Session
  { sessionId :: String
  , expiry    :: Maybe UTCTime
  , dataStore :: Map String String
  } deriving (Read, Show)

type SessionOpts = [(String, String)]
data SessionStore = SessionStore
  { readSession    :: (MonadIO m, HasState m)
                    => SessionOpts -> m Session
  , saveSession    :: (MonadIO m, HasState m)
                    => SessionOpts -> Session -> m ()
  }

```

Listing 5.9: Types representing a session and a session store

5. Controller

A session is identified by its `sessionId`, allowing the retrieval of a session from stores like a database by means of its key. In addition, a session contains a `Map` storing the values, as well as an optional timestamp for expiry. A `sessionStore` is represented by two functions for loading and saving the session, which are used before and after a request is handled by an action. The function `loadSession` may either retrieve an existent session or create an empty one. The type class `HasState` is only used internally and denotes the presence of the `RequestEnv` and the `ResponseState`. Based upon these two functions, the session stores may realize behaviour specific to their storage location. In addition, they may rely on additional parameters in form of the `sessionOpts`, provided in the `AppConfiguration`.

Currently, there are two different session stores implemented: `noSession` and `cookieSession`. The former store can be seen as a dummy, as it discards the session after every request and creates an empty session for every new request. Therefore, this store can be used to disable the session. Secondly, the `cookieStore` saves the entire session in a cookie. To detect possible changes of the content, the session is additionally signed by a keyed-hash message authentication code (HMAC) function before it is stored in the cookie. For this function, the application configuration has to provide a secret key in form of a `String`, containing 32 characters. Using this key, the session is also verified to be consistent when delivered with the request. In consequence, the session can be assumed to be not altered by the user. Like the cookies in general, this session storage also limits the size of the session, due to its storage in a cookie. Nonetheless, the session is generally intended to contain identifiers and small data portions only, so this is considered to be sufficient.

5.2.5. Flash Messages and Errors

Beside the communication between different requests using the session concept, actions may provide additional information to the view, which should be discarded after the response calculation. Examples are a confirmation message to be shown after an action was performed, or validation errors to be shown in combination with forms. The former, the exchange of status messages, is realized by the concept of the *flash*, an internal map capable of storing a message under a given name. The latter, the storage of validation errors, is covered by an additional map, for storing validation errors. Both can be accessed via the corresponding access functions shown in listing 5.10.

```
setFlash    :: String -> String -> StateController ()
getFlash   :: String -> StateController (Maybe String)
setErrors  :: String -> [(String, String)] -> StateController ()
getErrors  :: String -> StateController [(String, String)]
```

Listing 5.10: Functions for accessing the flash and error store

In general, the information contained in the flash and the error map, both will be discarded after the response calculation. Nevertheless, this would cause loss of information if the action returns a redirect. In this case, the view for which the status message was intended is not processed until the subsequent request. As this would lead to the status message being lost in combination with redirects, this special case is separately handled. In case of redirects, the flash containing the status messages is made accessible to subsequent requests by temporarily storing it in the session.

5.2.6. Action Composition

In general, the actions are intended to realize self-contained functionality, such as one action for editing a customer, and another action for displaying them. Nevertheless, there may be the need for several actions to share a common functionality. For instance, a set of actions may be restricted to be only accessible if the user previously logged in with his user name and his password. Although it would be possible to introduce this functionality in the respective actions, it is reasonable to extract this behaviour into a common function to prevent code duplication. For instance, this may be sensible for a function performing a login check.

In Haskell, this can be expressed using a higher-order function, which can be combined with the respective actions. This concept of action composition allows a simple, but powerful way of decorating actions with additional functionality. For example, the login check can be realized as:

```
requireLogin :: StateController a -> StateController a
requireLogin c = do
  u <- getSessionValue "logged_in"
  if (isJust u)
    then c
    else redirectToUrl "/main/login"

safeShow = requireLogin showAction
```

5. Controller

This function takes an arbitrary action and checks whether the user is logged in, which is denoted by an entry in the session. If the entry is present, the passed-in action is performed. Otherwise, the user is redirected to the login page. This function is then applied to a `showAction`, resulting in a safer version accessible to logged in users only.

5.2.7. Routing

Routing in the context of Hawk applications is the process of passing the incoming request to the corresponding actions. These actions are then responsible for handling the request and generate the response. Although the routing is performed by the front controller, it has to be configured by the developer, to determine which combination of action and view should be invoked for a incoming request. This is expressed by a partial function for determining the respective action-view combination:

```
type Controller = StateController ByteString
routing :: Hack.Env -> Maybe Controller
```

In this function, the type synonym `Controller` is used for the result type of an action-view combination (cf. section 5.1.4). As this function has to reference the different actions and views in their implementation, it is furthermore the central mechanism to provide them to the front controller. Consequently, the front controller can be *configured* to serve the respective application, avoiding code dependency to the application's code.

The routing function has to be provided by the developer as a part of the `AppConfiguration`. Because of the general type signature, the function can be used to express different approaches of determining the controllers. For example, the routing may require a static URL pattern like `"/modulename/actionname"` to identify an action. Another example may be a routing corresponding to the technique of *representational state transfers* [Fie00], where the request method is additionally used to denote the kind of an action (GET for retrieval, DELETE for deletion, ...). In consequence, the routing can be configured to allow arbitrary routing strategies.

In addition to this general interface, the framework also provides a simple routing approach based on the above mentioned URL pattern `"/modulename/actionname"`. To use this routing strategy, the actions of a module have to be collected in a `Map String Controller`, whereas the modules in turn are collected in another `Map` again. In consequence, the controller map is of the type:

```
controllers :: Map String (Map String Controller)
```

In this approach, the outer `Map` is used to determine the corresponding module, whereas the inner `Map` contains the requested action. A routing map may for example be constructed as:

```
customerControllers = fromList [("show", showController)]

controllers :: Map String (Map String Controller)
controllers = fromList [ ("customer", customerControllers) ]
```

This routing map can then be converted to a routing function using the function `simpleRouting`, defined in `Hawk.Controller.Routes` (cf. listing 5.11).

```
simpleRouting :: Map String (Map String Controller)
              -> Env -> Maybe Controller
```

Listing 5.11: Function for simple routing based on a module/action pattern

This function takes an incoming request and extracts the module and the action name from the `pathInfo` provided in the request. As the matching is case-sensitive, it is recommended to identify the module and the action by lower-case letters only. The `pathInfo` is split at the first slash, such that the first part determines the module, whereas the second part determines the action. If the `pathInfo` only contains one part, the action defaults to `"index"`, whereas the complete path determines the module. If the `pathInfo` refers to the root of the application `"/"`, it is assigned to the controller identified by the empty string. Consequently, the controller associated to the application's root path, the *root controller*, can be specified via the empty string in the controller `Map`:

```
controllers :: Map String (Map String Controller)
controllers = fromList
  [ ("customer", customerControllers)
  , ("", customerControllers) -- root controller
  ]
```

Based upon this simple routing approach, a few functions for constructing links referencing a specific action are also provided as shown in listing 5.12. The function `urlFor` takes a path referencing a module or an action inside the application and prepends the virtual location of the application. This allows the application's path to be changed without damaging the internal links. The function `actionUrl` basically behaves the same, but allows the module and the action to be specified separately,

5. Controller

and in combination with additional request parameters to be included. Finally, the function `redirectTo` can be used to redirect the user to a specific action, providing both the action and additional parameters.

```
urlFor      :: HasState m => String -> m String
actionUrl   :: HasState m => String -> String
            -> [(String, String)] -> m String
redirectTo  :: String -> String
            -> [(String, String)] -> StateController a
```

Listing 5.12: Functions for constructing application URLs

5.3. Front Controller

The front controller is the central part of the Hawk framework, as it integrates the actions implemented by the developer into an entire application. This application can be served to the web by means of different Hack handlers. Nonetheless, the front controller is not restricted to accept only request handled by the actions. As web applications typically contain at least some static files, such as images or cascading style sheets, the front controller additionally provides them to the browser as well. Consequently, the front controller is structured into several parts, as depicted in figure 5.3.

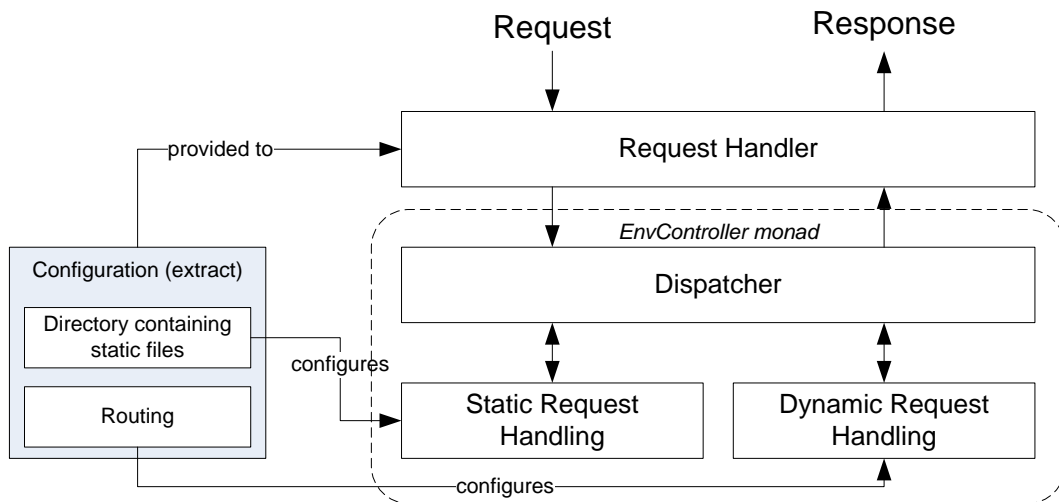


Figure 5.3.: Components of the front controller

The *request handler* constitutes the interface to the application, as it complies with the Hack interface. The request handler is pre-initialized with the application configuration and then handed over to the Hack handler. If a request is provided to the handler, it basically provides both the configuration and the request to the **EnvController** monad and invokes the *dispatcher*. The dispatcher is responsible for handling the request, combining the functionality of handling request to static files (static request handling), as well as the invocation of the provided actions (dynamic request handling).

The internal request handling process, as well as the entire application configuration and the retrieval of the request handler are described in the following.

5.3.1. Request Handling

The dispatcher function, responsible for the request handling and response generation in Hawk, is implemented as:

```
dispatcher :: EnvController Response
dispatcher = liftM addDefaultHeaders
              $ handleExceptions
              $ staticRequest
              $ dynamicRequest
              error404
```

The **addDefaultHeaders** function, which is applied on top, takes the **Response** calculated by the passed-in computation and adds default HTTP headers, such as the header denoting the length of the content. The **handleExceptions** is responsible for handling the exceptions thrown inside the application. Exceptions are mapped to a response indicating an internal server error (status 500). The remaining functions are responsible for the response generation.

The **staticRequest** function is responsible for handling requests for static files, such as images or style sheets. The folder containing the static files to be served to the web has to be specified inside the **AppConfiguration** as the **publicDir**. Supplied with a request, the **staticRequest** searches inside this folder for a corresponding file. If the file is present, it will be returned, while otherwise the request is handed over to the **dynamicRequest** function.

In contrast to static files, the **dynamicRequest** function is responsible for invoking the controllers (rendered actions) provided by the developer. Based on the routing

5. Controller

information, the function searches for the controller to be invoked. If there is no controller provided to the respective request, the `error404` function is executed, resulting in a response with the status 404 (page not found). Otherwise, the `dynamicRequest` loads the session and initializes the `ResponseState` to be provided to the controller, which is invoked afterwards. The results returned by the controller is extracted out of the `EitherT` monad by un-applying the transformer, leading to an intermediary result of the type `Either Response ByteString`. While complete responses remain unchanged, the results of the type `ByteString` are converted into a response with the status 200. Finally, the session is saved into the `SessionStore` and the headers and cookies of the `ResponseState` are integrated into the response.

5.3.2. Configuration

To construct the `EnvController` monad, in which the dispatcher is implemented, the application configuration and the database connection have to be provided. The configuration bundles the different options necessary for particular parts of the application, as shown in listing 5.13.

```
data AppConfiguration = AppConfiguration
  { sessionStore :: SessionStore
  , sessionOpts  :: [(ByteString, ByteString)]
  , routing      :: Hack.Env -> Maybe Controller
  , templatePath :: String
  , publicDir    :: String
  , error404file :: String
  , error500file :: String
  , confOptions  :: [(String, String)]
  }
```

Listing 5.13: Configuration of a Hawk application

The `sessionStore` and `sessionOpts` fields contain the options necessary for session handling. While the former denotes the type of the store that should be used in the application, the latter contains additional options specific to this store. The `cookieStore`, for example, requires this options to contain the secret key for session verification. The `routing` integrates the actions and views developed by the user, in form of the general routing function. The template path is used inside the template view to denote the directory containing the templates, and will be described in the next chapter. While the `publicDir` denotes the folder containing the files to be

served as static files, the fields `error404file` and `error500file` denote two file names expected inside the `publicDir`, for example `"404.xhtml"`. These files should provide the page to be shown if the request could not be handled, or if an internal server error occurred. Using these files, the appearance of the respective response can be customized. Finally, the `confOptions` provide a list for additional configuration options.

The database connection finally is provided in form of the `AppEnvironment`, shown in listing 5.14.

```
data AppEnvironment = AppEnvironment
  { connectToDB :: IO ConnWrapper
  , logLevels   :: [(String, Priority)]
  , envOptions  :: Options
  }
```

Listing 5.14: Environment of a Hawk application

The intention of the `AppEnvironment` is to allow the execution of an application, represented via the configuration, in different environment. This type does not only contain the database to be used, but also additional settings to set the level for logging. The logging in Hawk is realized using the `hslogger` library [Goeb], and may even be used inside the actions and views. For example, for testing of the application, the database connection might be changed to a special database. Furthermore, during development, the application may use extensive logging, which is not acceptable in a production environment.

5.3.3. Hack Integration

To be integrated into a Hack handler, the front controller finally has to comply with the type of a Hack application. Hack applications are defined as:

```
type Application = Env -> IO Response
```

For retrieving a Hack application, the module `Hawk.Controller.Initializer` contains the function `getApplication`, which takes the application's environment and configuration as its parameters:

```
getApplication :: AppEnvironment -> AppConfiguration
               -> Application
```

5. Controller

The application environment is processed first, leading to the logging being configured and the database connection being established. Afterwards, the configuration is provided to the `EnvController` monad. Using this function, the application can be integrated into a Hack handler, for example a Hack web server. Furthermore, the application may also be combined using Hack middleware, as depicted in figure 5.4.

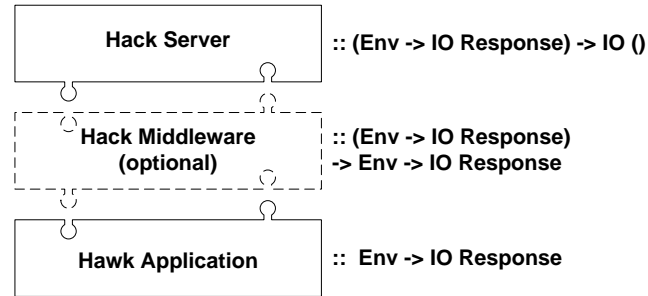


Figure 5.4.: Server integration via the Hack web interface

6

Template View

The purpose of the template view is to provide a way to render data calculated by the action into an HTML page. As described in section 3.2.2, the approach is based on HTML templates, defining the structure of the resulting page, and enforces the separation of the template and the view logic. First, in section 6.1 the templates are described in detail, in connection with the additional features they provide for template composition. The following section 6.2 discusses the view instance to be used in combination with the templates, as well as the functions provided to implement the view logic. Finally, an approach for improving the robustness of the rendering process is illustrated in section 6.3.

6.1. Templates

The templates used inside the template view are XML files with the file extension ".xhtml". They are expected to be located in the file system folder `templateDir`, as provided by the `AppConfiguration`, whereas the path may either be absolute or relative to the application's root path. The template folder is expected to be

6. Template View

further structured according to the modules the corresponding views belong to, resulting in the name of the module to be included into the search path. For example, the view used for rendering a list of customers may be located in the module `Customer` and may use the `list` template, so the template is expected to be located in `<templateDir>/customer/list.xhtml`.

XHTML is used as the XML language in the templates, enriched with several tags specific to the template system. To be distinguishable from XHTML tags, the template system tags use the prefix `hawk` and the name space `http://fh-wedel.de/hawk`. An exemplary template, containing both the name space information and specific tags, may therefore look like:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:hawk="http://fh-wedel.de/hawk">
    <head><title>Hawk application</title></head>
    <body><hawk:bind name="content"/></body>
</html>
```

The main purpose of the template system tags is to define so-called *binding targets*, locations inside the template where dynamic content should be bound to. The view logic is responsible for replacing the target by the content, which is typically provided by an action. To enable the re-usage of templates, templates furthermore can be combined by composition tags. For instance, a template may reference another template to be embedded prior to rendering, or may be surrounded by a layout template. Finally, the template system offers tags for the common task of inserting the flash messages and validation errors stored in the `ResponseState`. These different tags are explained in more detail in the following.

6.1.1. Binding Targets

Binding targets are identified by a logical name and can be defined using the `bind` tag. This tag has to provide an attribute `name`, containing the logical name of the target as its value. Although the uniqueness of the name is not checked, it is strongly recommended to use different names only, in order to prevent unpredictable behaviour during content insertion. A simple example for the definition of binding targets is shown in the following template for rendering a customer:

```
<hawk:bind name="firstName"/>
<hawk:bind name="lastName" maxLength="20">Curry</hawk:bind>
```

In the first line, an empty `bind` element is defined, which is the most common case. In addition, a `bind` may also contain attributes or child elements, like in the second line. This element has a child element, serving as a default value for displaying the template during editing. In principle, both examples are handled in the same way, as they are replaced with dynamic content. However, if a target provides an attribute or child elements, these will be accessible to the view logic, such as the attribute `maxLength`, allowing the content to be calculated by means of them.

Beside the insertion of scalar values, the binding targets can also be used to express optional content as well as structured content. Optional content is realized in the view logic by either binding the content to be inserted to the target, or by binding empty content, resulting in the target to be removed. Consequently, it is also possible to bind a list structure to a target. To support structured content, the `bind` tags have to be nested. For instance, the previous example can be extended to represent lists of customers via:

```
<hawk:bind name="customer">
  <hawk:bind name="firstName"/>
  <hawk:bind name="lastName" maxLength="20">Curry</hawk:bind>
</hawk:bind name="customer">
```

As the child elements of the outer `bind` are accessible to the view logic, the logic may render each customer using the inner targets. The result, a list of rendered customers, can then be bound to the outer `bind`.

In addition to elements, it is also possible to insert attributes into the templates. Like elements, the attributes are identified by a logical name, though in a different name space. These targets are provided via an additional `bind` attribute inside regular XHTML tags, containing the logical name of the target as its value. For instance, the adjustment of the color of a `div` tag by a style attribute may be realized as:

```
<div hawk:bind="color">Answer: <hawk:bind name="answer"/></div>
```

Providing a value for the attribute and an XML element for the answer, the rendered template may then look like:

```
<div style="green">Answer: 42</div>
```

6.1.2. Template Composition

To re-use common template parts in different contexts, templates may be composed by embedding sub-templates, or by being surrounded with another template. This allows the extraction of common template parts, as well as the sharing of a uniform layout.

Embedding Sub-templates

The embedment of sub-templates into a template is provided in form of the `embed` tag. The tag references the template to be embedded via the attribute `what`, specifying the template by means of its file path. The template is retrieved based on the file system location of the embedding one, so it is both possible to provide absolute or relative paths. The content of the embedded template is inserted into the embedding

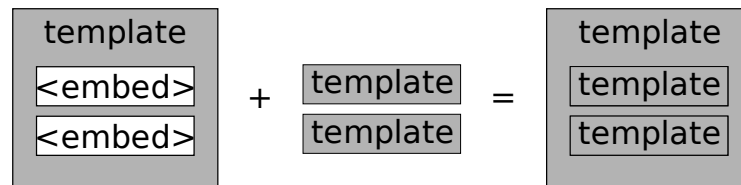


Figure 6.1.: Template combination via embedment

template as a replacement for the `embed` tag (cf. figure 6.1). For example, the header of a XHTML page may be separated into a global header used for all pages, as well as a local header for the respective module. A corresponding embedding may look like:

```
<div class="header">
  <hawk:embed what="../header/superheader.xhtml"/>
  <hawk:embed what="local_header.xhtml"/>
</div>
```

Surrounding Templates

The surrounding of templates is the counterpart of the embedding mechanism. In this case, a template is surrounded by another template (cf. figure 6.2), which may be the global layout, for instance. The template to be surrounded has to specify the

path to the surrounding template in the file system via the attribute `with`, in the same manner as for template embedding. The surrounding template has to contain a binding target denoting the position where the inner template will be inserted. This name is referenced by the inner template in the attribute `at`. For example, a

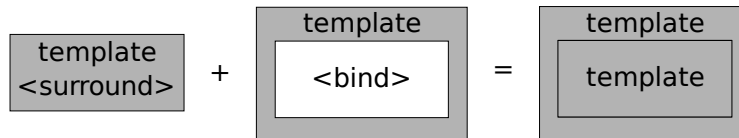


Figure 6.2.: Surrounding a template

template may define the content of a page to be inserted into the HTML body, while the layout template contains the head of the page. This may be realized as:

```
<!-- surrounded template -->
<hawk:surround with="layout.xhtml" at="content">
  <h1>The content!</h1>
</hawk:surround>

<!-- surrounding template -->
<html>
  <head><title>The title!</title>
  <body><hawk:bind name="content"/></body>
</html>
```

Ignore Tag

To form valid XML, templates are required to contain a single node as their root node. However, for templates to be embedded in other templates, it may be necessary to define several elements on the same hierarchic level, without an additional root node. To support those templates, the tag `ignore` can be used as a transient root node, containing a list of child elements. At embedding, the `ignore` element is replaced by its children. The following example shows a template grouping some sub-templates to be embedded as a single template:

```
<hawk:ignore>
  <hawk:embed what="./sub1.xhtml"/>
  <hawk:embed what="./sub2.xhtml"/>
</hawk:ignore>
```

6.1.3. State Access

Two fields of the **ResponseState**, the **flash** map for messages and the **error** map for validation errors, contain content which is supposed to be inserted into the templates. This insertion is supported in the form of two corresponding tags, providing basic formatting. Nonetheless, for more ambitious cases the customized insertion via binding targets is of course possible as well.

Flash Messages

The **message** tag is used to insert the message strings saved in the **flash** of the **ResponseState**. The string to be inserted is identified by the attribute **type**, of which the value is used as the lookup key for the **flash** map. If a message is present for the key, the message is inserted; otherwise the **message** element is removed. To customize the appearance of the messages, the element may contain arbitrary HTML tags in its body. In this case, the message will be inserted at the place indicated by a special **content** tag. For instance, the following template shows an example for displaying notices:

```
<hawk:message type="notice">
  <div class="notice"><hawk:content/></div>
</hawk:message>
```

Validation Errors

The **error** tag behaves similarly to the **message** tag, but is used for displaying error messages previously collected in the **ResponseState** and is typically used in combination with an input form. As the errors are assigned to a logical name, this name has to be provided in the attribute **for**. Again, the value of the attribute is used as the lookup key for retrieving the assigned errors. If there are no errors present, the tag is removed. Otherwise, the errors will be inserted into the body of the **error** element, at the place denoted by a **content** tag. Errors are displayed as an unordered list, whereas each entry is of the form **attribute : error**. An exemplary template may look like:


```
<hawk:error for="customer">
  <div class="errors">
    <h1>The following errors occurred</h1>
    <hawk:content/>
  </div>
</hawk:error>
```

Assuming that the user inserted an invalid date of birth (date in the future) during editing a customer, as well as a name too long to be printed on billings, these errors can be saved in the `ResponseState` during request handling. While the invalid date can be assigned to the `dateOfBirth` attribute, the printed name consists of three component (first name, initials, last name), leading the error to be assigned to the entire customer. As the result, the template will be extended to:

```
<div class="errors">
  <h1>The following errors occurred</h1>
  <ul>
    <li>date of birth : must be in the past</li>
    <li>the full name must be shorter than 40 characters</li>
  </ul>
</div>
```

6.1.4. Head Merge

For applications consisting of multiple sites, it is desirable to provide a uniform layout throughout the application. The intended way is to surround the particular templates with another template defining the general layout. The inner templates will be inserted into the surrounding template, typically somewhere inside the `body` element. However, this would require the inner templates to share the same `head` element defined in the surrounding template. Nevertheless, templates may need to alter the content of the `head` element specific to the respective page, for example to provide an individual page title or to include additional cascading style sheets.

Therefore, the template view provides the concept of a *head merge*, adopted from the Lift framework [CBDW09]. Unlike valid HTML, templates may additionally contain a head element inside the body. This element is recognized by the template view and automatically merged into the head element located under the HTML element. In conclusion, it is both possible to keep a uniform layout template and change elements belonging to the page head. For example, to set the title of a page, it is possible to add a title element in the template of the respective page via:

6. Template View

```
<hawk:surround with="layout" at="content">
  <head><title>My page title</title></head>
  <h1>The content</h1>
</hawk:surround>
```

6.2. TemplateView

The `TemplateView`, shown in listing 6.1, is the view type to be used in combination with the templates described above. As it is an instance of the `View` type class, it can be used in the same manner like the other instances.

```
data TemplateView a = TemplateView
  { templateName :: String
  , toXhtml      :: XmlTree -> a -> StateController [XmlTree]
  }
```

Listing 6.1: `TemplateView` data type

The `templateName` denotes the file name of the template associated to the view (without the file extension), allowing the combination of a rendering function with different templates, as well as the opposite. The rendering function, named `toXhtml`, is responsible for rendering the template of the view, in respect of the dynamic content, represented by the type parameter `a`. The templates are represented by the type `XmlTree`, originating from the Haskell XML Toolbox (HXT) [Sch]. The result of the rendering is the template enriched with the dynamic content. The result is provided in form of a list, to be able to provide a list of templates of intermediary results, while the final template is expected to be returned as a singleton. The calculation of the results is performed inside the `StateController` monad, allowing the function to access the `ResponseState` or the `RequestEnv`. Furthermore, it may retrieve additional information from the model. For example, for rendering a customer, the view may access the customer's category as well.

The entire processing of a `TemplateView` is divided into three steps (cf. figure 6.3):

1. Preprocessing
2. Rendering
3. Postprocessing

At first, the template to be rendered is calculated in its entire form by resolving all embedments and surroundings during *preprocessing*. Furthermore, the **ignore** tags are replaced by their child elements. In the second step, the content is inserted (bound) into the template by means of the rendering function `toXhtml`. Finally, the remaining tags are examined during *postprocessing*. This includes the insertion of eventually messages and errors, as well as the removal of remaining binding targets. Afterwards, the name space is removed and the **head** elements are merged into one element at the top to form valid XHTML.

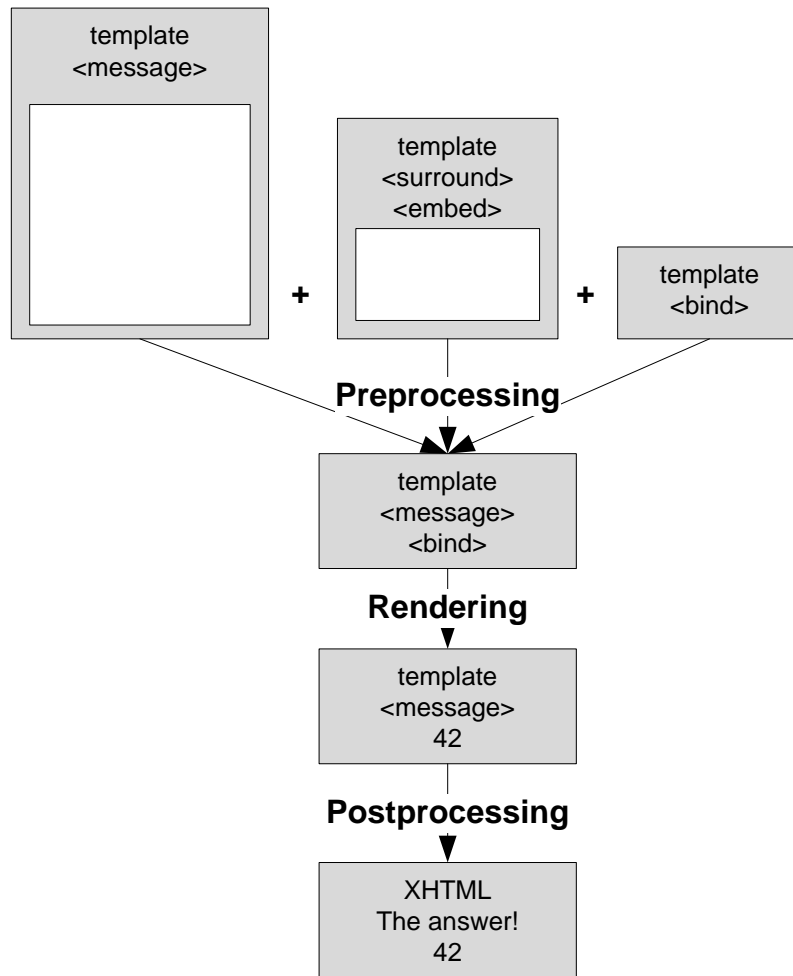


Figure 6.3.: Template processing order

While the first and the third step are provided by the framework, the rendering function has to be implemented by the developer. This can be achieved using a set of supporting functions, which should be described next.

6.2.1. Value Binding

To insert content into a template, the template view provides a function `bind`, shown in listing 6.2, binding the content to the respective positions in the template. The content thereby is supplied with the name of the target it should be bound to.

```
bind :: XmlTree                                -- The template
    -> [(String, [XmlTree])]                  -- XML elements to be bound
    -> [(String, [(String, String)])]         -- Attributes to be bound
    -> [XmlTree]                              -- the result
```

Listing 6.2: Bind function for inserting content into a template

The first parameter of the function is the template containing the binding targets, whereas the second parameter contains the content to be bound, in the form of an association list. While the key specifies the binding target, the value is the content to be bound to this target. The attributes to be bound are given as the third parameter, also as an association list. In this case, the attributes are given as a list of key-value pairs, whereas the key denotes the attribute's name and the value, consequently, the attribute's value. If the content contains names which are not present in the template, the respective content will be silently ignored. To match the type of the `toXhtml` function, the result of the `bind` function is a list of `XmlTrees`.

While the attributes are represented as `Strings`, the elements have to be provided in the form of `XmlTrees`. In conclusion, the data to be inserted has to be converted prior to insertion. This task is supported by a set of conversion functions capable of constructing HTML elements as well as plain text, shown in listing 6.3.

```
type Attributes = [(String, String)]
text           :: String -> XmlTree
showtext      :: Show a => a -> XmlTree
tag           :: String -> Attributes -> XmlTree
contentTag    :: String -> Attributes -> [XmlTree] -> XmlTree
```

Listing 6.3: Helper functions for `XmlTree` construction

The function `text` converts a `String` into a corresponding text in the HTML, additionally escaping the content. That is, character such as umlauts are converted into their corresponding escaping sequences. For instances of `Show`, the function `showtext` first converts them into a `String`, before converting them into an `XmlTree`

afterwards. In addition to simple text, it is furthermore possible to construct entire HTML elements using the functions `tag` and `contentTag`, respectively. Both accept the name of the tag as its first argument (such as `"h1"`), followed by optional attributes. While the former function constructs an empty element, the latter also may contain child elements, provided in the third argument. Finally, it is of course possible to construct the content using the HXT directly.

To cite an example, for showing all customers in a table there might exist a template of a single row:

```
<tr>
  <td><hawk:bind name="name"/></td>
  <td><hawk:bind name="dob"/></td>
</tr>
```

Using the helper functions, the rendering function of a single customer may then be given as:

```
customerXhtml :: XmlTree -> Customer
               -> StateController [XmlTree]
customerXhtml template c = return $ bind template
  [ ("name", [Html.show $ firstName c ++ ' ' : lastName c])
  , ("dob" , [Html.showtext $ dateOfBirth c])
  ]
  [] -- no attributes
```

Assuming that the template file is named `"customer.xhtml"`, the template name and the rendering function may be combined to form a `TemplateView` via:

```
customerView :: TemplateView Customer
customerView = TemplateView "customer" customerXhtml
```

6.2.2. **Html Form Construction**

Beside the conversion of the content into `XmlTrees`, a common task in combination with input forms is the construction of elements contained in the form. To ease the creation of these elements, Hawk provides a set of functions for creating the common variants, such as text fields, or select boxes for date selection. For example, the text field of an edit form for the customer's first name may be realized as:

```
("firstname", [H.textfield "firstname" (firstName c) []])
```

6. Template View

These functions are to be used in the same manner as the basic `HtmlHelpers`. Hence, a detailed description is omitted. To provide a brief overview, figure 6.4 enumerates some helpers as well as their purpose.

Purpose	Provided helpers (extract)
Selection	select box, check box, radio button, select group for date and/or time selection
User input	text field, text area, password field, hidden field, button, file upload
Other	label, link, image

Figure 6.4.: Helper functions for HTML form construction

6.2.3. Template Access

In the example stated above, the template was passed directly into the `bind` function, without further investigations of its structure. While this may be sufficient for simple templates, the rendering of templates containing complex structures such as nested binding targets require access to the template structure. Therefore, the functions shown in listing 6.4 provide access to selected aspects of the template.

```
getAttribute    :: XmlTree -> String -> String
lookupAttribute :: XmlTree -> String -> Maybe String
subTemplate     :: XmlTree -> String -> [XmlTree]
```

Listing 6.4: Functions for accessing the template

The functions `getAttribute` and `lookupAttribute` can be used to retrieve the attributes of an `XmlTree`, for example to retrieve an optional attribute of a `bind` tag. The function `subTemplate` allows the retrieval of certain `bind` tags of the template. To achieve this, it searches in a given template (first parameter) for all `bind` tags of a given name (second parameter). The found tags are then returned, whereas unique names will result in a singleton list. Beside to access the attributes, this function in particular is intended to access nested binding targets in the templates. For instance, a template for listing all customers in a table may embed the template defining a single row of customer via:

```
<table>
  <hawk:bind name="customer">
    <hawk:embed what="customer.xhtml"/>
  </hawk:bind>
</table>
```

In this template, the inner template is wrapped into a `bind` tag and can therefore be extracted using the `subTemplate` function. The extracted template may then be used to render the list of customers, before the results are concatenated to form the rendered list of all customers:

```
listXhtml :: XmlTree -> [Customer] -> StateController [XmlTree]
listXhtml t cs = do
  let renderFunction = customerXhtml
      $ head $ subTemplate template "customer"
  rendered <- concat 'liftM' mapM renderFunction customers
  return $ bind template [("customer", c)] []
```

In this example, the template part for a single customer row is extracted first. The above introduced function for rendering a single customer, `customerXhtml`, is afterwards partially applied to the template part, resulting in the `renderFunction`. This function can then be used to render the list of customers, whereas the results are concatenated afterwards. The final result, the list of customer rows, can be bound to the template as usual.

6.3. Templates as Algebraic Data Types

Represented as an `XmlTree`, the templates to be filled with content provide no information about their internal structure on the type level. Neither can the nesting of the binding targets be derived without further ado, nor can the intended type of the content. The structure of the targets is only assumed in the rendering function, based on the developer's knowledge, while the content to be inserted is represented as `[XmlTree]`. In conclusion, it is neither possible to guarantee that content is bound to every target, nor that the content is of the intended type. This may in principle cause errors which are hard to find, such as unbound targets and content not bound at all, for example because of a mistyping. To solve this problem, the templates can be represented as algebraic data type, which are derived from the template structure, using Template Haskell. In consequence, the templates provide additional type information such as their structure, enabling the above stated errors to be encountered at compile time.

To represent the type of the content to be bound to a certain target, the binding targets in the templates may be extended with additional information. This information both specifies the intended content type, as well as a function to be used for conversion

6. Template View

into HTML. The type of the content is denoted by the attribute `type` and may principally reference an arbitrary Haskell type. The attribute `format` denotes the name of a function for converting this type into a `[XmlTree]`. For example, the row of a single customer may be extended to:

```
<tr>
  <td><hawk:bind name="name" type="String"/></td>
  <td><hawk:bind name="dob" type="Day"
              format="formatDay"/></td>
</tr>
```

Based upon this information, it is possible to derive the algebraic data type representing the template. This type will contain record fields for the different binding targets of the template, which are of the type specified in the template. More precisely, the derivation mechanism behaves like the following:

- `surround` tags are resolved before conversion and the template is surrounded
- `embed` tags are converted into a field containing a list of the type which is derived for the embedded template, allowing repetition
- `bind` tags are converted into a field containing either a `[XmlTree]` if no `type` attribute is present, or the respective type otherwise
- Other elements such as `messages` are ignored as they do not affect the binding targets

The type can be calculated and inserted using a splice expression in combination with the function `viewDataType`, taking the module name and the name of the template as the parameters. For the template of a customer row, the splice expression `$(viewDataType "Customer" "customer")` leads to the creation of the following type:

```
data CustomerCustomer = CustomerCustomer
  { name :: String
  , dob  :: Day
  }
```

To be convertible into a list of `XmlTrees`, the types used in the template and in the resulting data type either have to specify a conversion function in the template (like `formatDay`), or have to provide an instance of `Show`. Based upon this information, the template data type can be bound to a template afterwards, which is indicated by

an instance of the type class `Bindable`. As the binding is internally calculated, it should not be discussed any further.

Using the template's data type, the result for a single customer row can now be represented as:

```
formatDay :: Day -> [XmlTree]
formatDay d = [Html.showtext d]

$(viewDataType "Customer" "customer")

customerXhtml' :: Customer -> StateController CustomerCustomer
customerXhtml' c = return CustomerCustomer
  { name = firstname c ++ " " ++ lastname c
  , dob  = dateOfBirth c
  }

customerView' :: TemplateView Customer
customerView' = typedView "customer" customerXhtml'
```

In this example, the `formatDay` function converts a date value by using `showtext`, although other formats are of course possible. As the function is referenced in the template and therefore used for binding the data type to the template, it has to be stated prior to the splice expression. To additionally create a `TemplateView`, the function `typedView`, as shown in listing 6.5, integrates the binding based on the template's name and the corresponding data type into an ordinary `TemplateView`.

```
typedView :: Bindable b => String
          -> (a -> StateController b) -> TemplateView a
```

Listing 6.5: Function `typedView` for converting template data types

Furthermore, the data types can also be used to express embedded templates in a simple way. Revisiting the example of listing all customers, the type can be derived to and used as:

```
data CustomerShow = CustomerShow
  { customerCustomer :: [CustomerCustomer] }

listXhtml' :: [Customer] -> StateController CustomerList
listXhtml' cs = do
  rendered <- mapM customerXhtml' cs
  return $ CustomerList rendered
```

6. *Template View*

It should be remarked that this type is derived from a slightly modified template. While in the original template the `bind` element was necessary to allow the repetition of a structure (the customer row), this feature is included in the data type for embedded templates by default. Therefore, the `bind` element has to be removed, resulting in the simplified template:

```
<table><hawk:embed what="customer.xhtml"/></table>
```

7

Project Diary Application

In the previous chapters, the fundamental concepts of the Hawk framework have been introduced, based upon a small example. During development of the framework, a slightly more complex application has been implemented, to gain additional experiences with the Hawk framework. This application deals with aspects such as user authentication, traversal of the domain model, and a more complex template nesting. To provide further insights into the capabilities of the Hawk framework, these aspects are described in this chapter.

7.1. Introduction

Students at the University of Applied Sciences in Wedel at the field of computer science have to realize a software project during their study at bachelor degree. To provide guidance in managing these projects and to gain experience in estimating the needed time, there exists a *software project diary*. This diary is written using the framework Ruby on Rails and has been re-implemented in its major parts using Hawk.

7. Project Diary Application

The objective of the diary is to provide a system to trace the advances of software projects. Each project contains some attributes describing the project, such as the *project title*, the *members* working at the project as well as the available range of time, denoted by a *start* and an *end date*. To be able to manage the different tasks, a project is divided into multiple *areas*, which in turn contain multiple *packages*. While the areas represent the major building blocks of the project, such as problem analysis or implementation, the packages provide a structure for the areas. To encourage the reasoning about the total realization time, each package has to provide the time needed for its realization. As precise estimations are impossible, the time has to be provided in a range by providing the minimum and the maximum time estimated.

While this structure is used for planning the project, the advance is denoted by different *steps*. A step is the smallest unit and represents a certain work for realizing a particular package. It contains both the afforded time and the percentage of the step completion. In consequence, based upon the steps the afforded time can be aggregated for packages, areas and the entire project. Furthermore, based on the completion of the steps, the expected time can be forecast and compared to the time estimated before.

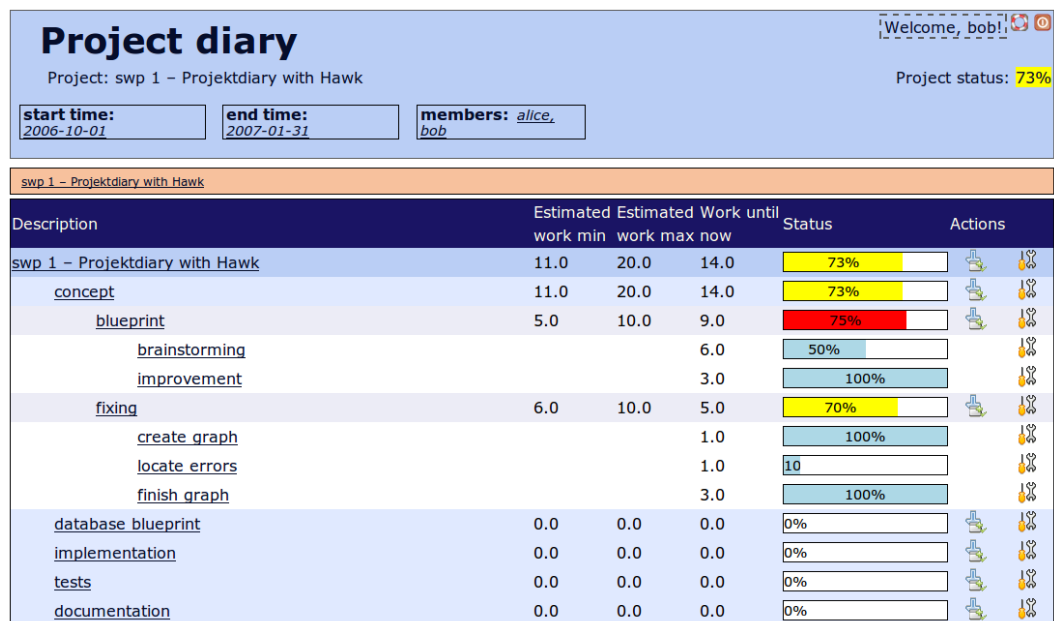


Figure 7.1.: Index page of the project diary

The index page of a project is depicted in figure 7.1. This screenshot shows the overview of the project “Projectdiary with Hawk”, as well as its structure of areas,

packages and steps. For example, the first area “concept” has two packages, containing five steps in total. To the right of the area title are the minimum and maximum of the estimated time, as well as the number of hours afforded so far. The statuses of the particular items are presented as a progress bar, with the color indicating the severity of the discrepancy.

7.2. Application Structure

The directory structure of the application is split into two Haskell modules and three additional folders (cf. figure 7.2). While the **App** module contains the main application, the **Config** module contains the routing, as well as the configuration of the entire application. Files which are served as static files, such as the XHTML files for error conditions, images and style sheets, are contained in the **public** folder. The **log** folder Hawk contains the log files produced by the Hawk framework, while the SQLite database is put in the **db** folder.

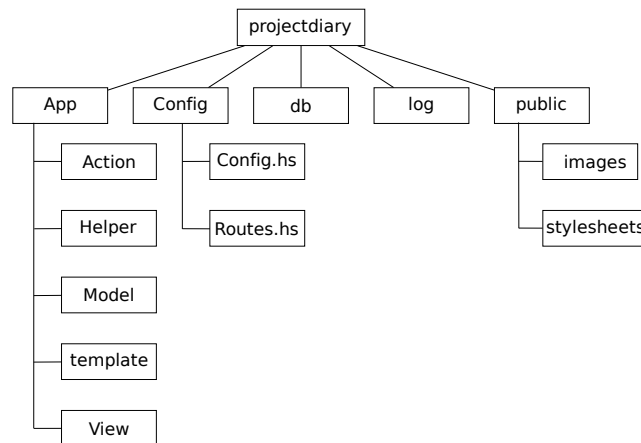


Figure 7.2.: Directory structure of the project diary

Inside the **App** module, the application is structured according to Hawk’s model-view-controller implementation. The **Action** module contains the different actions of the application, which are structured according to the underlying model types. For example, there is an action for managing areas and one for package, as well as for the other types. The **template** directory contains the XHTML templates, which are used by the **View** module, containing the view logic. Inside the **Model** module,

the mapped types of the application are defined. Finally, the `Helper` folder contains functions used in multiple places.

7.3. Model Component

The types of the domain model, like areas or packages, are mapped to corresponding tables in the database. Based upon the type mappings, the relationships are established and further specified using the `BelongsTo` and `HasMany` type classes, as there are only one-to-many relationships present. In addition, instances for the type classes for validation and attribute updates are provided. For example, for an area it is checked that the description is not empty, as well as unique in the scope of the respective project.

Some information, needed for displaying the project overview, requires the relationships to be traversed, for example the aggregation of the time afforded so far. As this is a part of the domain logic, it is in consequence also implemented in the model component. The current time afforded in the project, `workNow`, is the sum of the afforded time of the areas, which in turn relies on the time of the packages and their steps. In consequence, the value has to be aggregated, like shown in listing 7.1.

```
Project.workNow :: MonadDB m => Project -> m Double
Project.workNow = getAreas >=> liftM sum . mapM Area.workNow

Area.workNow :: MonadDB m => Area -> m Double
Area.workNow = getPackages >=> liftM sum . mapM Package.workNow

Package.workNow :: MonadDB m => Package -> m Double
Package.workNow = liftM (sum . map Step.duration) . getSteps
```

Listing 7.1: Calculation of the current work of a project

The first function retrieves all areas of the project and calculates their `workNow`, which in turn require the same for their packages. Although this implementation is straight forward, it suffers from two problems. First, the same traversal is also applied for retrieving different attributes of the project, for instance the minimum and maximum time estimated. As the database mapping currently provides no form of caching, this results in multiple retrievals of the same values. Fortunately, this can be easily avoided using the technique of accumulator types for retrieving the values simultaneously:

```

nowMinMax :: MonadDB m => Project -> m (Double, Double, Double)
nowMinMax = getAreas >=> liftM sum3 . mapM Area.nowMinMax
  where
    sum3 xs = (sum a, sum b, sum c)
    where (a,b,c) = unzip3 xs

```

Nevertheless, due to the traversal, there arises an additional problem. As the function `Area.nowMinMax` is evaluated for every area of the project, this will lead to n evaluations, where n is the number of associated areas. Furthermore, for the packages again a similar function is evaluated, resulting in a total of $n*m$ evaluations of the function for packages. As the function for packages retrieves the steps from the database, this may result in a significant number of database accesses. In conclusion, the step-wise traversal has to be avoided. A first approach may be the use of a custom SQL query for retrieving all steps of a project. But as the values have to be calculated and provided on the different levels of the hierarchy, this only solves a part of the problem. Consequently, out of this problem the demand for a caching mechanism can be derived.

7.4. Actions

The actions of the project diary mainly deal with the creation of new items and the modification of existing items. Furthermore, the diary is used by multiple students realizing different projects, so the user of the diary has to be authenticated before the access is granted. While the creation and modification is combined in a single action, the authentication is realized as an action combinator. Both are explained in more detail in the following.

7.4.1. Modification of Areas

The modifications of the different items basically behave the same; therefore the modification of an area is picked as a representative example. The action for editing an area is depicted in listing 7.2, responsible for the updating and creating of the `Area` data type. First, the function `findOrCreate` searches for an existing entry. If an `id` parameter is provided in the request parameters and an entry is existent in the database, the respective area is returned. Otherwise, an empty area is created with the `new` function of the `Model` instance. Whether the value could be found in the database or not, is denoted by the additional boolean value, where `True` denotes

7. Project Diary Application

a new area. After the `findOrCreate` function, it is checked if the user is allowed to edit this project. If this is not the case, the remaining part of the action is skipped due to the `EitherT` functionality. In this case, the user is logged out and redirected to the login page as an invalid access was encountered.

```
areaEditAction :: StateController (Area, Bool)
areaEditAction = do
  (area, isNew) <- findOrCreate'
  getProject area >=> projectAuthorize
  method <- getRequestMethod
  case method of
    POST -> do
      (a, errs) <- getParams >=> updateAndValidate area ""
      if null errs then do
        if isNew then insert a else update a
        setFlash "notice" "Your changes have been saved"
        redirectToAction "diary" "index"
      else do
        setErrors "area" errs
        return (a, isNew)
    _ -> return (area, isNew)
```

Listing 7.2: Action for editing an area

If the user is associated to the project, it is decided whether the request contains changes (POST request), or if the value should be displayed for editing (GET request). While in the latter case the area is returned to be rendered, in the former case the area is updated, using the functionality of its `Updateable` and `Validatable` instances. If the area is valid, the changes are saved and the user is redirected to the start page, whereas otherwise the errors are saved in the state. These can then be displayed to the user, as shown in figure 7.3.

The screenshot shows a web interface for editing an area. At the top, a red error message box states: "The following errors prohibited the area from being saved". Below this, a grey box contains the text: "There were problems with the following attributes:" followed by a bulleted list: "■ description : must be unique". The main form area has a light blue background and contains two input fields: "Area description:" with the value "database blueprint" and "Comment:" which is empty. At the bottom of the form are two buttons: "Save" and "Reset".

Figure 7.3.: Error messages while editing an area

7.4.2. Authentication

To be able to access the project diary, a user has to login to a specific project, using his user name and his password. If the user name and the password are valid and the user has access to the project, its name and the current project are saved in the session, both with their identifier (users are also saved in the database). Based upon this information, the access of the user to a specific action can be checked. For instance, the `indexAction` for displaying the start page shall only be accessible by members of the respective project. Therefore, the action is combined with an action combinator for user authorization (cf. listing 7.3). This function first checks whether the session has expired or not, using `prepareSession`. Afterwards, `userAuthorize` retrieves the user and project id from the session via the functions `currentUser` and `currentProject`. If one of these fails, the client is sent back to the login page using the `redirectToLogin`. Finally, it has to be checked if the user is permitted to access the respective project, before executing the passed-in action. This check queries the database, retrieving if the user is associated to the project.

```

userAuthorize :: StateController a -> StateController a
userAuthorize contr = prepareSession "diary" $ do
  u <- currentUser
  p <- currentProject
  if isNothing u || isNothing p
  then redirectToLogin "diary" Nothing
  else do
    permitted <- isAtProject (fromJust u) (fromJust p)
    if permitted then contr
    else redirectToLogin "diary" Nothing

```

Listing 7.3: Action combinator for user authorization

7.5. Template Views

As the project diary is accessible in the form of HTML pages, these pages naturally are implemented using the template view. As depicted in the screenshot of the index page showing the project overview, the different areas, packages and steps are also included in this page. Furthermore, the titles of the respective items are links to separate pages, showing only the corresponding content. For example, by clicking on

7. Project Diary Application

a link to a package, only the package is shown, as well as its steps. As this feature is provided on all levels of the hierarchy, this naturally leads to a hierarchy of embedded templates to avoid both template and code duplication.

The hierarchy of the template embedment is shown in figure 7.4. For every level of the four levels of the hierarchy, there is a respective template which is rendered to the response. These are the **index** template for the diary module and the different **show** templates. These templates constitute the entire response page, and therefore also include the respective header templates to show additional information. The rendering of the hierarchic table is realized by the different templates prefixed with an underscore. Furthermore, the templates are surrounded by a layout template

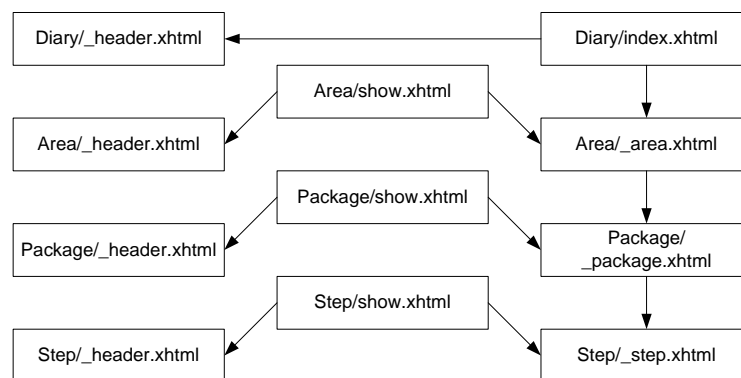


Figure 7.4.: Embedment hierarchy of the templates (extract)

containing a common header and reducing the redundancy of the templates (not shown in the figure).

The rendering function of the index page naturally reflects the structure of the template embedment by re-using the rendering functions of the sub-templates. For instance, the areas are rendered using their rendering function `showXhtml`, while this in turn relies on the rendering of packages. The origin of the content of the main page in respect of the templates and views is depicted in figure 7.5.

7.6. Summary

This chapter only discovered the surface of the project diary implementation, as the entire presentation would be out of scope. However, the diary has been successfully re-implemented, as the described functionality suggests. Furthermore, the concepts

















Description		Estimated work min	Estimated work max	Work until now	Status	Actions
concept	Area	11.0	20.0	14.0	<div><div></div></div> 73%	 
blueprint	Package	5.0	10.0	9.0	<div><div></div></div> 75%	 
brainstorming	Step			6.0	<div><div></div></div> 50%	 
improvement				3.0	<div><div></div></div> 100%	 
fixing		6.0	10.0	5.0	<div><div></div></div> 70%	 
create graph	Step			1.0	<div><div></div></div> 100%	 
locate errors				1.0	<div><div></div></div> 10	 
finish graph				3.0	<div><div></div></div> 100%	 

Figure 7.5.: Contribution of the embedded templates to the index page

and functions provided by Hawk proved their usefulness. The database mapping, for example, is used for a complex traversal of relationships. Although some limitations could be discovered regarding efficiency, these are not specific to the implementation, but to the problem. In fact, some limitations can already be solved using standard Haskell techniques, while other provide useful hints for further improvements. For the controller component the action composition has to be accentuated, providing a simple, but powerful way to combine functionality. Finally, the template composition dramatically improved the template handling as it allows the re-usage of significant amount of both templates and view logic.

8

Conclusion

To conclude the thesis, a discussion of the achieved results should follow in this chapter. To begin with, this chapter provides a summary of the developed framework as well as an evaluation of the results. Subsequently, an overview of future work is presented, which is based upon the prior assessment, the experiences gained during development and the inspirations found at comparable frameworks. Finally, an outlook of the near future of the Hawk framework is sketched, which hopefully will lead to further improvements.

8.1. Summary

In this thesis a framework for developing dynamic web applications in Haskell has been developed and implemented. The framework, named Hawk, encourages the developer to structure his web application according to the model-view-controller pattern and supports the development of each of the three components by appropriate library functions.

8. Conclusion

The monad for implementing actions provides access to the incoming request in general, and to specific information contained therein in particular, such as the automatically parsed request parameters. Furthermore, a response state is provided, which introduces the concept of sessions between different requests and allows the communication between actions by sharing a common state. Finally, the routing of incoming requests to the according action can be easily and comprehensively configured.

To realize the persistence of the model component, the framework offers a library for mapping the data to an underlying database. By providing the mapping information about how a specific type should be mapped to a database table, the developer gains the functionality for storing and querying the data. Not only does the mapping provide a uniform interface for accessing the data, but also is the developer free to change the default behaviour according to his needs. While the common task of querying is possible using the full range of SQL, it is in addition supported by query constructions via a criteria. Furthermore, the task of data validation to improve data quality is assisted by validation functions, which can be applied to arbitrary data types.

The view component in general is designed to be extensible, allowing the user to provide different views. In addition, the generation of HTML pages is covered by a template system. Based upon the principle of binding dynamic contents to logical names defined, this system encourages the separation of the template and the necessary logic for inserting the dynamic content. Furthermore, the templates can be converted to corresponding data types at compile time. This data type can be used inside the view logic to ensure the correspondence of the view logic and the template, both in structure and types.

In conclusion, the framework does not only cover the typical parts of web application development, but is also extensible, for example by providing additional techniques of response generating.

8.2. Assessment

The practical usefulness of this framework has been proven during the implementation of the project diary, illustrated in chapter 7. The original diary, which was used as a guideline for the implementation presented in this thesis, could easily be translated

without further adaptation. Furthermore, the type system of Haskell both clarified the code and made side effects become apparent. In Rails, for example, I/O actions such as the access to the database cannot be discovered without further ado. In the Hawk implementation, they are indicated by a computation requiring an instance of `MonadDB`. In consequence, the type system not only facilitates the reasoning about the code, but also avoids some common sources of errors.

The database mapping and the validation provide the functionality necessary for web applications. Their range of features can roughly be compared to those of other frameworks. Nevertheless, it would be appreciable if the mapping component would further evolve towards a more feature-complete mapping library.

The request environment, provided to the developer for implementing actions, is comparable to those of other frameworks and sufficient for the intended tasks. The routing concept, on the contrary, may be further extended. Currently the URLs are strictly oriented to the internal structure of modules and controllers. To be able to provide more readable and therefore user-friendlier URLs, it is desirable to include a mechanism for rewriting the internal URLs to an external representation.

The template system, which is inspired by Lift, has proven to be perfectly usable, although it follows a rather unconventional approach. Furthermore, it encourages the strict separation of layout and code, and is therefore considered as a valuable contribution to solve this fundamental problem. The concept of generating data types out of the template enforces the structural identity of the template and of the content to be filled in, preventing a common source of errors.

8.3. Future Work

Although the framework already has been proven to be useful, it is still in an early stage of development. During the implementation phase, a lot of desirable features have been encountered, which would further improve the framework. In addition, the comparison of Hawk to other frameworks shed light on features which are currently not present, but are worth it to be included.

8.3.1. Database Mapping

Although the database mapping is sufficient for web application development, it would be reasonable to separate the component to allow a greater variety of applications. This in turn implies additional requirements to form a more general mapping library.

Multiple Database Connections

The current implementation is designed to work with a single database, which is provided to the individual functions performing database access. The connection currently is supplied to the code by means of a type class, making no assumptions about its origin. In consequence, it may also be changed to another database connection for particular parts of the application by the user. Nonetheless, the handling of multiple connections should rather be done by the database abstraction, so this generalization to multiple connections may become necessary in the future.

Nested Transactions

Another task to be investigated is the support for nested transactions. The HDBC library currently provides no mechanism for nesting transactions nor does every database support them. However, the functionality of nested transactions may be useful to roll back only specific parts of the database changes. To solve this problem, the mapping library therefore may provide nested transactions itself on the application level. As some database systems support the technique of save points inside transactions, this may be a potential way of implementation. Nonetheless, dependencies to specific database systems in general should be avoided. In consequence, further investigations have to be made.

Improved SQL Capabilities

Currently, the Criteria type is directly converted into a SQL string without any intermediary data types. While this is sufficient for the tasks encountered so far, the introduction of an additional type representing SQL statements would increase the flexibility. A separate type would allow the construction of more sophisticated SQL statements, as well as a traversal of the statement structure. This in turn opens a door for supporting different SQL dialects, and for SQL optimization.

Typed Criteria

At the moment, the Criteria functions do not conserve the types of their arguments, as these are internally converted into `SqlValues`. Unfortunately, this allows the construction of ill-typed restrictions such as `val True ==. val 42`. A possible solution is the introduction of *phantom types* [Hin03]. The function

```
val' :: Convertible a SqlValue => a -> CompareValue a
val' a = Val . toSql
```

conserves the type information, though converting the value into `SqlValue` as well. This additional type information may then be used to restrict comparison expressions to work only on values of the same type. For example, the signature for the equality function may be changed to

```
(.==.) :: CompareValue a -> CompareValue a -> Restriction
```

requiring both `CompareValues` to contain arguments of the same type. However, without further ado this approach is not applicable for column expressions using `col`. This is caused by the fact that it is not possible to derive the type information from the column name. HaskellDB [LAA⁺] solves this problem by introducing first-class labels for record field access, which are data types instantiating a type class for retrieving the name of the associated column, as well as its type. Nonetheless, this solution would significantly increase the complexity of the mapping. As the current implementation is designed to value simplicity higher than extensive type safety, adaptations therefore have to be carefully considered.

Record Identifiers

The current implementation for identifying records assumes that every value mapped to the database provides an identifier. For values to be inserted, the developer has to provide an initial value, which later will be replaced by the value assigned by the database. In consequence, by means of its identifier it is not possible to decide whether a certain value is already present in the database. While this additional information may be provided somewhere else, it seriously complicates the uniqueness validation. Although a carefully chosen initial value is expected to be not assigned in the database, this cannot be guaranteed. To solve this problem, it would be possible to make the identifier optional by using `Maybe`, or to force the value construction to simultaneously insert the value into the database.

8.3.2. Controller Component

Routing

As the dispatcher for retrieving the dynamic controllers is free to be entirely implemented by the user, the routing in principle provides enough flexibility. However, the default routing mechanism may be enhanced, as it currently operates with a module/action pattern of the request's `pathInfo` only. To support additional routing concepts, it would be desirable to provide respective routing functions. For instance, the routing for the concept of *representational state transfer* [Fie00] may be realized by a dispatcher also considering the request method.

URL Rewriting

The current approach requires each URL to exactly follow the pattern `/module/action?params`, which can lead to incomprehensible and hard to read URLs. Therefore, it would be sensible to distinguish an URL into its *internal* and its *external* representation. While the internal representation follows the already familiar pattern, the external representation can be designed to be human readable. This would allow the rewriting of an internal URL like `/customer/search?firstname=Haskell` into a more readable form like `/customer/with_firstname/Haskell`. The assignment between internal and external representations has to be bi-directional, as the external representations may also be calculated by means of the internal representation while generating HTML responses. For example, the creation of a link using the above mentioned internal form should result in a link to the external representation.

8.3.3. View Component

The most interesting enhancement of the view component may be the integration of AJAX, to allow the framework to support client-side processing. The JavaScript functionality is already present in form of different libraries such as jQuery [jqu]. In consequence, it is reasonable to adopt them to be used in connection with the template view. The template view therefore has to be extended by helper functions, which generate the necessary JavaScript functionality, freeing the developer from being forced to write JavaScript himself. Furthermore, AJAX may require additional functionality such as an additional view for generating JSON responses.

8.3.4. Additional Features

Scaffolding

When starting to develop a new application from scratch, the developer has to set up an appropriate file system/module structure. Furthermore, a considerable amount of code is repetitive, such as the code needed for mapping types to the database. Therefore, it would be helpful to unburden the developer from these tasks by providing an executable for creating the project structure in the file system, as well as the default implementations for actions such as edit, show, delete and list.

Test Support

When developing larger applications, at a certain point it will be necessary to check the implementation by automated testing. Pure functions can easily be tested using test frameworks such as *QuickCheck* [Cla] or *HUnit* [Her]. Nonetheless, the main part of the code is expected to be written using functionality offered by a surrounding monad, such as database access or access to the response state. Furthermore, the different test cases may have side effects and hence have to be isolated in general. In consequence, it is desirable to provide appropriate test monads as well as functions to support automated unit testing in Hawk.

8.4. Outlook

The next steps for improving the framework are already planned. The first and most important step will be to provide the framework to the Haskell community to gain further experiences and ideas in which direction the framework might evolve. Therefore, the framework should be published as open source on the Hackage [hac] platform.

Secondly, the ideas described in the previous section will subsequently be considered to be integrated into the framework. While many of the ideas will result in rather small changes, the integration of client-side processing by adding AJAX support to the framework is a major project. Fortunately, Alexander Treptow will deal with this task in his Master's Thesis, enriching the framework with JavaScript and JSON support.



Source Code of the Customer Application

This chapter contains the source code of an application based on the database model and types introduced in the implementation chapters. The application declares the data types of a *customer* and a *category*, which are connected by an $n : 1$ relationship. The application provides a single controller available via the URL `/customer`, offering the functionality for

- listing of all customers via `/customer/list`,
- deletion of a single customer via `/customer/delete`,
- insertion of a new customer (assigned to a category) via `/customer/edit`,
- updating of a customer, also via `/customer/edit`.

The application is divided into two sub-modules, `Config` and `App`. While the former contains the basic configuration and the routing information, the latter contains the application itself. The `App` module is further divided into three sub-modules. The `Model` module contains the domain model and its mapping to the database, the `Controller` contains the request handling, and the `View` module contains the view responsible for rendering the templates. The templates are also located in the `App` folder, inside a separate folder `template`.

A.1. Model Implementation

The model component mainly consists of the declaration of the `Customer` and the `Category` data type. To allow the usage of the same record field labels for different records (like `_id` for the identifier), the type declarations are put into different sub-modules. Both the customer and the category functionality are consolidated and re-exported by the modules `Model.Customer` and `Model.Category`, respectively.

Listing A.1 shows the type for representing customers. The functionality for database mapping is provided by declaring the necessary type class instances. Inside the `Record` type class, the timestamp of the customer is updated before accessing the database, using the ability to override the default implementation. The validation process checks whether a customer has a non-empty name (both first name and last name), as well as a valid date of birth. The `Updateable` instance finally updates the name attributes and the date of birth, but does not consider the category id. A change of this id would change the relationship and therefore is done in the controller for safety reasons.

```

module App.Model.Customer.Type (Customer (..)) where

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

-- the customer type
data Customer = Customer
  { _id          :: PrimaryKey
  , categoryId   :: ForeignKey
  , firstName    :: String
  , initials     :: Maybe String
  , lastName     :: String
  , dateOfBirth  :: Day
  , updatedAt    :: Maybe UTCTime
  } deriving (Eq, Read, Show)

-- persistence information
instance Persistent Customer where
  persistentType _ = "Customer"
  fromSqlList (l0:l1:l2:l3:l4:l5:l6:[])
    = Customer (fromSql l0) (fromSql l1) (fromSql l2)
      (fromSql l3) (fromSql l4) (fromSql l5) (fromSql l6)
  fromSqlList _ = error "wrong list length"
  toSqlAL x = [ ("_id"          , toSql $ _id          x)

```

```

        , ("category_id" , toSql $ categoryId x)
        , ("firstname"   , toSql $ firstName  x)
        , ("initials"    , toSql $ initials   x)
        , ("lastname"    , toSql $ lastName   x)
        , ("date_of_birth", toSql $ dateOfBirth x)
        , ("updated_at"  , toSql $ updatedAt  x)
    ]
    tableName = const "customers"

-- primary key functionality
instance WithPrimaryKey Customer where
    primaryKey = _id
    pkColumn = head . tableColumns
    setPrimaryKey pk c = c {_id = pk}

-- model behaviour
instance Model Customer where
    new = do
        t <- liftIO getCurrentTime
        return $ Customer 0 Nothing "" Nothing "" (utctDay t) Nothing
    insert c = do
        now <- liftIO getCurrentTime
        insertInTransaction $ c { updatedAt = Just now }
    update c = do
        now <- liftIO getCurrentTime
        updateInTransaction $ c { updatedAt = Just now }

-- validation
instance Validatable Customer where
    validator c = do
        validateNotNull "firstname" $ firstName c
        validateNotNull "lastname"  $ lastName  c
        validatePast      "dateOfBirth" $ UTCTime (dateOfBirth c) 0
    return ()

-- update
instance Updateable Customer where
    updater c name = do
        fn <- updater (firstName c) $ subParam name "firstname"
        ini <- updater (initials c) $ subParam name "initials"
        ln <- updater (lastName c) $ subParam name "lastname"
        dob <- updater (dateOfBirth c) $ subParam name "dateofbirth"
        return $ c { firstName = fn, initials = ini
                    , lastName = ln , dateOfBirth = dob }

```

Listing A.1: Data type for customers

Listing A.2 shows the type and declarations for the `Category`, similar to those of the customers. While the validation checks the non-emptiness of the name and

A. Source Code of the Customer Application

the discount information, the name is additionally checked for uniqueness. As the category name logically identifies the category, it has to be ensured that there are no two categories sharing the same name.

```
module App.Model.Category.Type (Category (..)) where

import Hawk.Model

data Category = Category
  { _id      :: PrimaryKey
  , name     :: String
  , discount :: Int
  } deriving (Eq, Read, Show)

instance Persistent Category where
  persistentType _ = "Category"
  fromSqlList (l0:l1:l2:[]) = Category (fromSql l0) (fromSql l1
    ) (fromSql l2)
  fromSqlList _ = error "wrong list length"
  toSqlAL x = [ ("_id"      , toSql $ _id      x)
    , ("name"      , toSql $ name      x)
    , ("discount" , toSql $ discount x)
    ]
  tableName = const "categories"

instance WithPrimaryKey Category where
  primaryKey      = _id
  pkColumn        = head . tableColumns
  setPrimaryKey pk p = p {_id = pk}

instance Model Category where
  new = return $ Category 0 "" 0

instance Validatable Category where
  validator c = do
    validateNotNull "name"      $ name      c
    validateUniqueness [] name "name"      c
    return ()

instance Updateable Category where
  updater c _name = do
    name' <- updater (name      c) $ subParam _name "name"
    disc' <- updater (discount c) $ subParam _name "discount"
    return $ c { name = name', discount = disc' }
```

Listing A.2: Data type for categories

Based upon the types previously defined, in listing A.3 the foreign key relationship between customers and categories is declared. Identified by the type `Customer2Category`, the functions for accessing the foreign key are provided. Furthermore, the relationship is declared as an $n : 1$ relationship, using the type classes `BelongsTo` and `HasMany`. These declarations allow the usage of additional functions specific for this kind of relationship, such as `getParent`.

```
{-# LANGUAGE TypeFamilies #-}
module App.Model.Customer.ForeignKeys where

import App.Model.Customer.Type (Customer(..))
import App.Model.Category.Type (Category)
import Hawk.Model

data Customer2Category = Customer2Category

instance ForeignKeyRelationship Customer2Category where
  type Referencing Customer2Category = Customer
  type Referenced  Customer2Category = Category
  relationshipName _ = "customer2category"
  fkColumn _ = "category_id"
  foreignKey _ = categoryId
  setForeignKey _ package key = package {categoryId = key}

instance BelongsTo Customer2Category
instance HasMany    Customer2Category
```

Listing A.3: Foreign key relationship between customers and categories

Finally, as shown in listing A.4, a short-cut function for accessing the category of a customer is implemented.

```
module App.Model.Customer.Functions where

import App.Model.Customer.Type
import App.Model.Customer.ForeignKeys
import App.Model.Category.Type (Category)
import Hawk.Model

getCategory :: MonadDB m => Customer -> m Category
getCategory = getParent Customer2Category
```

Listing A.4: Short-cut function for accessing the category of a customer

A.2. Customer Actions

Based on the model component introduced in the previous section, the controller realizes the functions for accessing the customers. The functions for creating new customers and for updating existent customers are merged into a single function, `editAction`. Instead of providing two separate functions, the distinction is based on the presence of the eventually passed in customer identifier. While a present parameter would lead to an update, a missing parameter denotes the creation of a new customer. Based upon the request method, it is then decided how the request should be processed. A GET request leads to the form being initially displayed, allowing the user to edit the customer, while for a POST request the changes will be saved in the database. The `isNew` flag is used to distinguish between insertion and updating in the database, as well as for the later rendering in the view component.

```
{-# LANGUAGE TemplateHaskell #-}
module App.Controller.CustomerController
  ( listAction
  , editAction
  , deleteAction
  ) where

import App.Model.Customer
import Hawk.Controller
import Hawk.Model

-- list all customers
listAction :: StateController [Customer]
listAction = select newCriteria

-- edit a single customer
editAction :: StateController (Customer, Bool)
editAction = do
  (customer, isNew) <- findOrCreate
  method <- getRequestMethod
  case method of
    POST -> do
      (c, errs) <- getParams >>= updateAndValidate customer ""
      if null errs then do
        catId <- readParam "category"
        let c' = c { categoryId = catId }
        if isNew then insert c' else update c'
        setFlash "notice" "Your changes have been saved"
        redirectToAction "customer" "list"
      else do
```

```

        setErrors "customer" errs
        return (c, isNew)
    _ -> return (customer, isNew)

-- delete a customer
deleteAction :: StateController ()
deleteAction = do
    (customer, isNew) <- findOrCreate
    if isNew then setFlash "error" "The requested customer does
        not exist"
    else do
        delete customer
        setFlash "notice" "The customer has been deleted"
    redirectToAction "customer" "list"

-- retrieve the customer to be edited or deleted
findOrCreate :: StateController (Customer, Bool)
findOrCreate = do
    customer <- readParam "id" >>= liftMaybe findMaybe
    case customer of
        Nothing -> do
            newCustomer <- new
            return (newCustomer, True)
        Just c -> return (c, False)

```

Listing A.5: Actions for customers

A.3. Customer Templates

The HTML pages for displaying the customer list as well as the edit page for customers are surrounded by a common layout template. This template, shown in listing A.6, contains the HTML head and integrates the flash messages eventually set by the controller.

```

<?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 -->

```

A. Source Code of the Customer Application

```
<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.6: Layout template with flash access

The template for listing all customers defines the table structure for displaying them (cf. listing A.7). The structure of the rows to be included is moved into another template, easing the construction of a list containing the rendered customers.

```
<?xml version="1.0" encoding="utf-8" ?>
<hawk:surround with="../../../Layouts/header.xhtml" at="content"
  xmlns:hawk="http://fh-wedel.de/hawk">
  <h1>Customer list</h1>
  <table>
    <tr>
      <th>Name</th>
      <th>Date of Birth</th>
      <th>Customer Category</th>
      <th>Actions</th>
    </tr>
    <hawk:embed what="customer.xhtml"/>
  </table>
  <br/>
  <hawk:bind name="newlink"/>
</hawk:surround>
```

Listing A.7: Template for listing all customers

The template for rendering a single customer is shown in listing A.8. It defines a binding target for the name of the customer, as well as the date of birth and the category. Furthermore, an additional target “links” is included for inserting links to

the delete and edit actions. The name of the customer is expected to be of the type `String`, while the date of birth is expected to be of the type `Day`. A passed in date will be rendered in to an `XmlTree` using the function `formatDay`. The construction of the data type representing this template and the declaration of `formatDay` will be done later, inside the View component.

```
<tr>
  <td><hawk:bind name="name" type="String"/></td>
  <td><hawk:bind name="dob" type="Day" format="formatDay"/></td>
  <td><hawk:bind name="category"/></td>
  <td><hawk:bind name="links"/></td>
</tr>
```

Listing A.8: Template for showing a single customer

The template for editing customers, as shown in listing A.9, finally contains a form for the respective attributes. Errors encountered during the validation process are provided to the user by means of the `hawk:error` element.

```
<?xml version="1.0" encoding="utf-8" ?>
<hawk:surround with="../../Layouts/header.xhtml" at="content"
  xmlns:hawk="http://fh-wedel.de/hawk" xmlns="http://www.w3.
  org/1999/xhtml">
  <head>
    <title>Edit Customer</title>
  </head>

  <h1>Edit Customer</h1>

  <hawk:error for="customer">
    <div class="errorExplanation" id="errorExplanation">
      <h2>The following errors prohibited the customer from
        being saved</h2>
      <p>There were problems with the following attributes:</p>
      <hawk:content/>
    </div>
  </hawk:error>

  <form action="/customer/edit" method="post">
    <p>
      <label for="firstname">First Name:</label><br/>
      <hawk:bind name="firstname"/>
    </p>
    <p>
```

A. Source Code of the Customer Application

```
<label for="initials">Initials:</label><br/>
<hawk:bind name="initials"/>
</p>
<p>
  <label for="lastname">Last Name:</label><br/>
  <hawk:bind name="lastname"/>
</p>
<p>
  <label for="dateofbirth">Date of Birth:</label><br/>
  <hawk:bind name="dateofbirth"/>
</p>
<p>
  <label for="category">Category:</label><br/>
  <hawk:bind name="category"/>
</p>
<hawk:bind name="id"/>
<input type="Submit" value="Save"/>
<input type="reset" value="Reset"/>
</form>
<br/>
<hawk:bind name="listlink"/>
</hawk:surround>
```

Listing A.9: Template for editing a customer

A.4. Customer View

The customer view shown in listing [A.10](#) is responsible for rendering the results computed by the controller, using the templates listed above. First of all, the view defines the function `formatDay` for rendering a `Day` into a list of `XmlTrees`. As this function is referenced in the customer template, it will be used inside the data type of this template, generated by the splice expression

```
$(viewDataType "Customer" "customer")
```

Below the generation of the template data types, the routing targets are defined by combining an action with the respective rendering function. Because the `editAction` ends with a redirection to the list of customers, it is combined with an `emptyView` as there are no results expected to be rendered. In contrast, the `listXhtml` and `editXhtml` functions are responsible for converting their input and the HTML template into the complete HTML page. While the former function uses the functionality of data types for template representation, the latter operates on the raw template.

```

{-# LANGUAGE TemplateHaskell #-}
module App.View.CustomerView (routes) where

import App.Controller.CustomerController
import App.Model.Customer
import qualified App.Model.Category as Cat

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

import Data.Maybe (fromMaybe)
import Data.Time.Calendar (Day, toGregorian)

formatDay :: Day -> [XmlTree]
formatDay d = [H.showtext d]

$(viewDataType "Customer" "customer")
$(viewDataType "Customer" "list")

routes :: [Routing]
routes =
  [ ("list",listAction >>= render (typedView "list" listXhtml))
  , ("edit",editAction >>= render (TemplateView "edit"
    editXhtml))
  , ("delete",deleteAction >>= render emptyView) ]

-- version with data type
listXhtml :: [Customer] -> StateController CustomerList
listXhtml cs = do
  cc <- mapM customerXhtml cs
  return CustomerList
    { customerCustomer = cc
    , title = [H.text "Customers"]
    , newlink = [H.textlink "/customer/edit" "New customer"]
    }

customerXhtml :: Customer -> StateController CustomerCustomer
customerXhtml c = do
  cat <- getCategory c
  return CustomerCustomer
    { name      = firstName c ++ ' ' : lastName c
    , dob       = dateOfBirth c
    , category  = [H.text $ Cat.name cat]
    , links     = [ H.textlink ("/customer/edit?id=" ++ show (
      _id c)) "Edit"

```

A. Source Code of the Customer Application

```
        , H.text " | ", H.textlink ("/customer/delete?id=" ++
          show (_id c)) "Delete" ]
    }

-- version without data type
editXhtml :: XmlTree -> (Customer, Bool)
          -> StateController [XmlTree]
editXhtml template (c, isNew) = do
  cats <- select newCriteria
  let (y, m, d) = toGregorian $ dateOfBirth c
  return $ bind template
    [ ( "firstname", [H.textfield "firstname" (firstName c)
      []])
    , ( "initials", [H.textfield "initials" (fromMaybe "" $
      initials c) []])
    , ( "lastname", [H.textfield "lastname" (lastName c) []])
    , ( "dateofbirth"
      , [ H.selectDay "dateofbirth.day" d []
        , H.selectMonth "dateofbirth.month" m H.monthNames []
        , H.selectYear "dateofbirth.year" y 1900 2009 [] ])
    , ( "category", [H.select "category" (H.options (show . Cat
      ._id) Cat.name cats) []])
    , ( "listlink", [H.textlink "/customer/list" "Customer list"
      []])
    , ("id", [H.hidden "id" (show $ _id c) [] | not isNew])
    ] []
```

Listing A.10: The customer views

A.5. Configuration

The routings defined for the customer controller only map the action's name to the corresponding action. Therefore, the identification of the module has to be added in a second step. In listing A.11, the module is registered under the URL `/customer`.

```
module Config.Routes where

import qualified App.View.CustomerView as CV
import Hawk.Controller.Types (Controller)
import qualified Data.Map as M

routing :: M.Map String (M.Map String Controller)
routing = M.singleton "customer" $ M.fromList CV.routes
```

Listing A.11: Application routing

The entire routing is integrated into the application configuration. In addition to the configuration, listing A.12 also shows the definition of an environment the application should be executed in. The environment specifies the database to be used, as well as the severity levels for logging.

```

module Config.Config (development, configuration) where

import qualified Config.Routes as Routes (routing)

import Hawk.Controller.Initializer (AppEnvironment (..))
import Hawk.Controller.Routes (simpleRouting)
import Hawk.Controller.Session.CookieSession (cookieStore)
import Hawk.Controller.Types (AppConfiguration (..))

import Control.Monad (liftM)
import Data.ByteString.Lazy.UTF8 (fromString)
import Database.HDBC.Sqlite3
import Database.HDBC (ConnWrapper(..))
import System.Log.Logger

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

configuration :: AppConfiguration
configuration = AppConfiguration
  { sessionStore = cookieStore
  , sessionOpts  =
    [("secret", "12345678901234567890123456789012")]
  , routing      = simpleRouting Routes.routing
  , templatePath = "./App/template"
  , publicDir    = "./public"
  , confOptions  = []
  , error404file = "404.html"
  , error500file = "500.html"
  }

```

Listing A.12: Application configuration

Providing an environment and the configuration, the Hack application function can then be retrieved. In listing A.13 this application is passed to a server function as the Hack handler.

A. Source Code of the Customer Application

```
module Main (main) where

import Config.Config (configuration, development)
import Hawk.Controller.Initializer (getApplication)
import Hack.Handler.SimpleServer as Server (run)

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

Listing A.13: The main function



Installation Guide

A CD is attached to this report, containing a PDF version of the report as well as the source code of the developed Hawk framework and two applications developed with Hawk. The content of the CD is structured as follows.

Libraries

The **Libraries** folder contains the source code of the Haskell libraries developed for the thesis. These are:

eitherT The **EitherT** monad transformer. This library contains the transformer as well as type class instances for other common monad transformers.

stringable The **Stringable** type class for uniform **String** representations. This type class provides a function which converts values into **Strings** like the **show** function, but avoids the escaping of **String** values.

B. Installation Guide

hawk The Hawk framework. This library contains the current status of the Hawk framework, including the controller components, the template view and the database mapping. The Hawk library depends on both the **EitherT** and the **Stringable** libraries.

The libraries are provided as cabal packages and can therefore be installed using the standard cabal installation process. Because of the library dependencies, the Hawk framework has to be installed at last. To install a library, the following commands have to be executed inside the respective library folder after copying them to the hard disk:

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

If the cabal install program is available, the installation commands for a single library may even be abbreviated to:

```
$ cabal install
```

Applications

The **Applications** folder contains the web applications developed using the Hawk framework. These are:

customer This application is based upon the database model introduced in the implementation chapter and listed in the previous appendix chapter. It contains a simple controller for listing, editing, creation and deletion of customers. This application focuses on simplicity and provides a good start for investigating the structure of Hawk applications.

projectdiary This application contains the project diary described in chapter 7, containing the user functionality described therein as well as an additional administration section for project and user management.

Both projects follow the same structure as described in this thesis. They may be started after the installation of the above mentioned libraries via executing the **main** function in the **Main** module.

Bibliography

- [Ang] ANGELOV, Krasimir: *hsqL: Simple library for database access from Haskell*. <http://hackage.haskell.org/package/hsqL>, last checked: 2009-08-13
- [BAH04] BRINGERT, Björn ; ANDERS HÖCKERSTEN, Anders: Student Paper: HaskellDB Improved. In: *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*. New York, NY, USA : ACM Press, 2004 <http://haskelldb.sourceforge.net/haskelldb.pdf>, 108–115
- [BBP09] BEAN VALIDATION EXPERT GROUP ; BERNARD, Emmanuel ; PETERSON, Steve: *JSR-000303 Bean Validation 1.0.CR1 Proposed Final Draft Specification*. http://docs.jboss.org/hibernate/stable/validator/reference/en/pdf/hibernate_validator.pdf. Version: 2009, last checked: 2009-08-20
- [BCHL09] BOS, Bert ; CELIK, Tantek ; HICKSON, Ian ; LIE, Håkon W.: *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification*. <http://www.w3.org/TR/CSS21/>. Version: 2009
- [Bri] BRINGERT, Bjorn: *cgi: A library for writing CGI programs*. <http://hackage.haskell.org/package/cgi>, last checked: 2009-09-30
- [Bro05] BROBERG, Niklas: *Haskell Server Pages through Dynamic Loading*, 2005 <http://www.cs.chalmers.se/~d00nibro/hsp/hsp-hw05.pdf>
- [CBDW09] CHEN-BECKER, Derek ; DANCUI, Marius ; WEIR, Tyler: *Exploring Lift: Scala-based Web Framework*. Apress, 2009
- [Cha] CHAKRAVARTY, Manuel: *Type families*. http://www.haskell.org/haskellwiki/GHC/Type_families, last checked: 2009-10-01

Bibliography

- [Cla] CLAESSEN, Koen: *QuickCheck: Automatic testing of Haskell programs*. <http://hackage.haskell.org/package/QuickCheck>, last checked: 2009-09-28
- [Fie00] FIELDING, Roy T.: *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, PhD thesis, 2000. <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>
- [Fow03] FOWLER, Martin: *Patterns of enterprise application architecture*. Addison-Wesley, 2003
- [Goea] GOERZEN, John: *HDBC: Haskell Database Connectivity*. <http://software.complete.org/hdbc>, last checked: 2009-08-13
- [Goeb] GOERZEN, John: *hslogger: Versatile logging framework*. <http://software.complete.org/hslogger>, last checked: 2009-10-17
- [hac] *HackageDB*. <http://hackage.haskell.org/packages/hackage.html>, last checked: 2009-09-29
- [Her] HERINGTON, Dean: *HUnit: A unit testing framework for Haskell*. <http://hunit.sourceforge.net/>, last checked: 2009-09-28
- [HHJW07] HUDAK, Paul ; HUGHES, John ; JONES, Simon P. ; WADLER, Philip: A history of Haskell: being lazy with class. In: *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*. New York, NY, USA : ACM, 2007. – ISBN 978–1–59593–766–X, pages 12–1–12–55
- [Hin03] HINZE, Ralf: Fun with phantom types. In: GIBBONS, Jeremy (Hrsg.) ; DE MOOR, Oege (Hrsg.): *The Fun of Programming*. Palgrave Macmillan, 2003, pages 245–262
- [Ht] HAPPSTACK TEAM, HAppS L.: *Happstack – Haskell application server stack*. <http://happstack.com/>, last checked: 2009-09-30
- [Int99] INTERNATIONAL, ECMA: *ECMAScript Language Specification*. <http://www.ecma-international.org/publications/standards/Ecma-262.htm>. Version: 1999

- [JJ99] JONES, Mark P. ; JONES, Simon P.: Lightweight extensible records for Haskell. In: *In Haskell Workshop*, 1999 <http://research.microsoft.com/en-us/um/people/simonpj/Papers/recpro.ps.gz>
- [Jon03] JONES, Simon P.: *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003
- [jqu] *jQuery: The Write Less, Do More, JavaScript Library*. <http://jquery.com/>, last checked: 2009-09-30
- [JRH⁺99] JONES, Simon P. ; REID, Alastair ; HENDERSON, Fergus ; HOARE, Tony ; MARLOW, Simon: A semantics for imprecise exceptions. In: *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*. New York, NY, USA : ACM, 1999. – ISBN 1-58113-094-5, pages 25–36
- [Kem] KEMP, Alson: *Turbinado : MVC Web Framework for Haskell*. <http://wiki.github.com/turbinado/turbinado>, last checked: 2009-09-30
- [LAA⁺] LEIJEN, Daan ; ANDERSSON, Conny ; ANDERSSON, Martin ; BERGMAN, Mary ; BLOMQVIST, Victor ; BRINGERT, Bjorn ; HOCKERSTEN, Anders ; MARTIN, Torbjorn ; SHAW, Jeremy: *haskelldb: SQL unwrapper for Haskell*. <http://haskelldb.sourceforge.net/>, last checked: 2009-09-30
- [Lift] *Lift: The Simply Functional Web Framework*. <http://liftweb.net/>, last checked: 2009-09-26
- [LJ] LÄMMEL, Ralf ; JONES, Simon P.: *Scrap your boilerplate ... in Haskell*. <http://www.cs.vu.nl/boilerplate/>, last checked: 2009-10-11
- [Mar06] MARLOW, Simon: An extensible dynamically-typed hierarchy of exceptions. In: *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*. New York, NY, USA : ACM, 2006. – ISBN 1-59593-489-8, pages 96–106
- [Mis02] MISC.: *XHTMLTM 1.0 The Extensible HyperText Markup Language (Second Edition)*. <http://www.w3.org/TR/xhtml1/>. Version: 2002
- [MJMR01] MARLOW, Simon ; JONES, Simon P. ; MORAN, Andrew ; REPPY, John: Asynchronous exceptions in Haskell. In: *SIGPLAN Not.* 36 (2001), Nr. 5, pages 274–285. – ISSN 0362-1340

Bibliography

- [Mon] *MonadCatchIO-mtl: Monad-transformer version of the Control.Exception module.* <http://code.haskell.org/~jcpetruzza/MonadCatchIO-mtl>, last checked: 2009-08-13
- [Mor07] MORDANI, Rajiv: *Java Servlet Specification Version 2.5 MR6.* 2007
- [Net09] NETCRAFT LTD.: *Web Server Survey.* Version: September 2009. http://news.netcraft.com/archives/2009/09/23/september_2009_web_server_survey.html, last checked: 2009-09-28
- [OGS08] O’SULLIVAN, Bryan ; GOERZEN, John ; STEWART, Don: *Real Worlds Haskell.* O’Reilly, 2008
- [OM96] OPEN MARKET, Inc.: *FastCGI Specification.* <http://www.fastcgi.com/devkit/doc/fcgi-spec.html>. Version: 1996
- [Rai] *Ruby On Rails.* <http://rubyonrails.org/>, last checked: 2009-09-26
- [Red08] RED HAT MIDDLEWARE LLC: *Hibernate Validator Reference Guide. Version: 3.1.0.GA.* http://docs.jboss.org/hibernate/stable/validator/reference/en/pdf/hibernate_validator.pdf. Version: 2008, last checked: 2009-09-20
- [RFC2109] KRISTOL, D. ; MONTULLI, L.: *HTTP State Management Mechanism.* RFC 2109 (Proposed Standard). <http://www.ietf.org/rfc/rfc2109.txt>. Version: February 1997 (Request for Comments). – Obsoleted by RFC 2965
- [RFC2616] FIELDING, R. ; GETTYS, J. ; MOGUL, J. ; FRYSTYK, H. ; MASINTER, L. ; LEACH, P. ; BERNERS-LEE, T.: *Hypertext Transfer Protocol – HTTP/1.1.* RFC 2616 (Draft Standard). <http://www.ietf.org/rfc/rfc2616.txt>. Version: June 1999 (Request for Comments). – Updated by RFC 2817
- [RFC3875] ROBINSON, D. ; COAR, K.: *The Common Gateway Interface (CGI) Version 1.1.* RFC 3875 (Informational). <http://www.ietf.org/rfc/rfc3875.txt>. Version: October 2004 (Request for Comments)
- [RFC4627] CROCKFORD, D.: *The application/json Media Type for JavaScript Object Notation (JSON).* RFC 4627 (Informational). <http://www.ietf.org/rfc/rfc4627.txt>. Version: July 2006 (Request for Comments)

- [RLHJ99] RAGGETT, Dave ; LE HORS, Arnaud ; JACOBS, Ian: *HTML 4.01 Specification*. <http://www.w3.org/TR/html4/>. Version: 1999
- [RTH09] RUBY, Sam ; THOMAS, Dave ; HANSSON, David H.: *Agile Web Development with Rails*. Pragmatic Bookshelf, 2009
- [Sch] SCHMIDT, Uwe: *hxt: A collection of tools for processing XML with Haskell*. <http://www.fh-wedel.de/~si/HXmlToolbox/index.html>, last checked: 2009-10-15
- [SJ02] SHEARD, Tim ; JONES, Simon P.: Template metaprogramming for Haskell. In: CHAKRAVARTY, Manuel M. T. (Hrsg.): *ACM SIGPLAN Haskell Workshop 02*, ACM Press, October 2002, pages 1–16
- [SJ03] SHEARD, Tim ; JONES, Simon P.: *Notes on Template Haskell Version 2*. <http://research.microsoft.com/~simonpj/papers/meta-haskell/notes2.ps>. Version: November 2003
- [Symfony] *symfony: Web PHP framework*. <http://www.symfony-project.org/>, last checked: 2009-26-09
- [Wan] WANG, Jinjing: *hack: a Haskell Webserver Interface*. <http://github.com/nfjinjing/hack/tree/master>, last checked: 2009-09-21
- [ZP07] ZANINOTTO, Francois ; POTENCIER, Fabien: *The Definitive Guide to symfony*. Apress, 2007

Affidavit

We hereby declare that this thesis has been written independently by us, 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, October 19th, 2009

(Björn Peemöller)

(Stefan Roggensack)