Rahisi

simple web development

Marlon Baeten

June 21, 2010

Bachelor Thesis Computer Science
Radboud University Nijmegen
Supervisor: Pieter Koopman
Student number: 0609765
Course code: IBI006

Abstract

There are many frameworks and languages designed for the development of web applications. Most of these are intended to be as generic and universal as possible and can be used for any type of web application. This property requires a high expressiveness and the availability of many functionalities. The more expressiveness and functionalities a system has, the more complex and incomprehensive it gets. For simple and small web applications this adds an overhead.
Rahisi offers a solution to this problem. It focusses purely on simple web applications. In *Rahisi* a web application can be defined using three small definition languages. In each languages a clearly separated part of the system is defined. The use of these dedicated languages result in a comprehensible and easy to learn system, in which a simple web application can be defines with a minimal amount of code. The anticipated disadvantage is the limited expressiveness and small set of functionalities of *Rahisi*.

Contents

Chapter 1

Introduction

The internet is a dynamic and fast evolving platform. Together with the constantly changing internet, change the techniques that make it possible to develop new web applications, or to modernise existing ones. There are dozens of programming or scripting languages to choose from. On top of them dozens or even hundreds of frameworks have been created to streamline the development and maintenance of a web applications.
This thesis will describe yet another way to develop web applications. *Rahisi* is designed with a clear goal in mind: to make the development of simple web applications as fast, easy and with the least code as possible. These objectives are achieved by designing three small definition languages. A model-, a controller- and a template language. These will be described in chapters 2, 3 and 4
In this chapter we will describe the aim-, the architecture-, the principles and techniques- and everything out of the scope of *Rahisi*.

1.1 The aim

The aim of *Rahisi* narrows down to be the perfect trade-off between the amount of code needed to develop a web application and the expressiveness of the system. The amount of code that is needed, has an almost direct relation with the comprehensibility, development speed, the ease of maintenance and the learning curve of the system.
The following paragraphs will explains these aims in more detail.

1.1.1 Comprehensible

In order to be of any use a system needs to be comprehensible. If a system is comprehensible each step in the development phase becomes less complicated. For example, when a system is easy to understand less errors will be made during the implementation. Errors will cause extra effort in the testing phase; they need to be detected, traced and repaired. There are also less obvious benefits of a comprehensible system, like the happiness of the programmer.

1.1.2 Fast development

The speed of a development process is a crucial characteristic to the success of certain development process or technique. The time spend on the development of a web application has a direct impact on the money that is spend. Another argument to make the development process as fast as possible is the fact that it makes it easier to cope with changing requirements. The time-to-market is another important factor.

1.1.3 Easy maintenance

The time spend on maintenance is often underestimated [3]. Easy maintenance can lower the time and thereby the costs of maintenance. If a system is complex or has a confusing structure, the time that is spend to correct, update or adjust even a tiny part can be far out of proportion. Small websites, like the website for a student association, are often managed by many different people over time. A lot of time and frustration are saved with a system that is easy to maintain.

1.1.4 Small learning curve

Even with a dedicated *web framework* in a well known scripting language, building a small website can take a lot of time. When the developer is new to a particular framework, most of the time is spend on reading documentation and tutorials. Such a system could be an advantage for a professional web developer that needs a lot of functionalities and flexibility. For a simple website it is easier to have a smaller set of functionalities. Less functionalities and flexibility can achieve a smaller learning curve. A small learning curve has a direct influence on the maintainability and the speed of the development process.

web framework a software framework designed to make common web development tasks easier

1.2 Architecture

1.2.1 The MVC principle

The architecture of *Rahisi* is based on the MVC principle for web applications like demonstrated in [1, 2]. There are many interpretations of the MVC principle for web applications. *Rahisi* uses the one that is shown in figure 1.1.

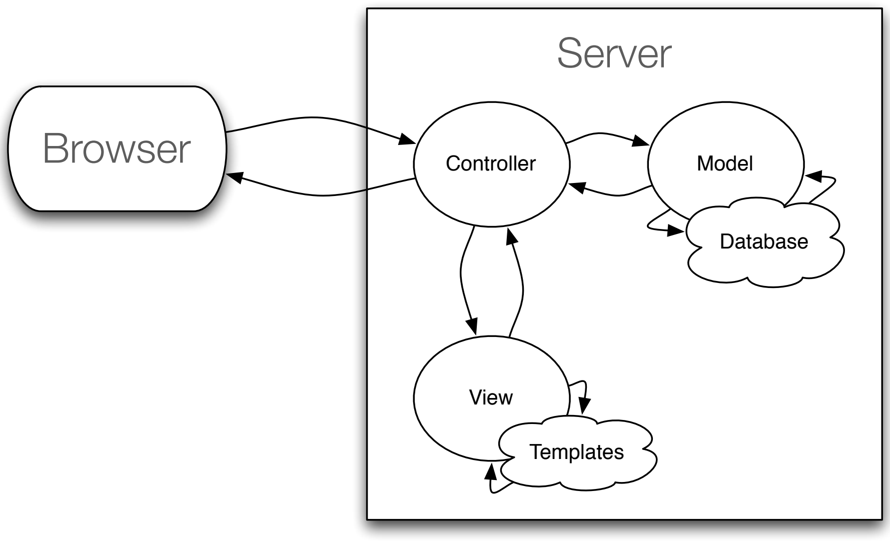


Figure 1.1: MVC architecture

In figure 1.1 a schematic overview of the architecture of *Rahisi* is given. The *server* stands for the *web server* which handles HTTP request and responses for *Rahisi*. In the scheme above *Rahisi* is a part of this server, it generates content that the web server serves.

web server a program that serves content using the HTTP over the World Wide Web

1.2.2 Request handling

A typical request is handled as followed:

- The client (web browser) sends a *HTTP* request to the server. The server deliveries the request to the controller of a certain *Rahisi* web application.
- The controller maps the request to a certain *webpage*. It sends one or more request for data (queries) to the model. The controller only queries the data that is needed on the requested page.
- The model fetches the needed data from the database and passes it to the controller.
- The controller passes the data to a template that is assigned to the specified webpage.
- The template merges the dynamic data and a static webpage layout and returns the result to the controller.
- The controller than passes the resulting webpage using the web server back to the client (web browser).

HTTP a request-response bases application layer protocol often used to request en send HTML pages

webpage a document or other resource of information that is accessible trough a web browser

1.2.3 Responsibilities

Controller
The controller is a function. As input it receives a HTTP request, the output is a HTTP response. Another responsibility is to map a request to a certain *action* or webpage. The controller also has to manipulate the model, based on the request. It is responsible for storing data in the model that is send with a request, and to query the model for data that is needed in the view. The controller is also responsible for the invocation of the needed template(s).

action part of the controller that defines the manipulations on the models and the needed views for a certain request

Model
The model is the definition and representation of the dynamic data in a web application. It implements a uniform access and definition of the data. Before data is stored in the database, the model has the responsibility to validate the data.

View
The view consist of a set of templates. Templates are rendered to create a representation of the data in one ore more formats. A typical example is a HTML template that renders a webpage around the dynamic data that is passed from the controller.

1.2.4 Security

The only way the security of a web application build in *Rahisi* can be compromised, and on which *Rahisi* has any influence, is input data. Input data, is data send with a HTTP request. Malicious data can only influence the web application when it is stored in the database. The only way to prevent malicious data to be stored in the database is by validating it. The model in *Rahisi* has extensive validation capabilities. These validation will be executed for all data that is stored in the database. When decent validations are specified in the model, the security of a *Rahisi* web application is assured.

1.2.5 Dependencies

In order to understand the operation of *Rahisi* it is important to know that the definition of a web application in *Rahisi* is compiled before deployment. In the compiling process Python [9] code is generated. The generated code together with the code that handles the basic operation of *Rahisi*, compose an interpretable web application. This web application operates within a web server. The fact that *Rahisi* code is translated to Python code is important to know for certain details in *Rahisi*, at which we will come back later.

1.3 Principles and techniques

The following subsections will enumerate the most important principles and techniques of *Rahisi*.

1.3.1 The MVC architecture and loose coupling

The MVC architecture was originally implemented in the programming language *Smalltalk*. Nowadays, it is a commonly used pattern both in GUI applications, as in web applications. There are a lot of different flavours of the architecture, but the most important property is the separations between a model, a view and a controller. By giving these components clear responsibilities and have them coupled loosely, the system gains a lot of flexibility.

1.3.2 Relational database

Rahisi uses a database based on the *relational database* model as a storage for data. Naturally there are many other ways to store data, but the flexibility, efficiency and widespread support of relational databases makes them a suitable choice for *Rahisi*.

relational database model of industry standard database software, in which data is organised in tables that often match with real world objects

1.3.3 The semantic web

The semantic web [4] is the development on the World Wide Web, to make information on webpages not only readable by humans, but also easy to parse by computers. As stated in [4] this development will bring a variety of new possibilities, like querying and combining information that can be found on websites.
Rahisi is designed with the semantic web in mind. The loose coupling between data (the model) and the representation of data (the view) makes it easy to create new representations that are machine readable. For example: de posts on a blog can be viewed on a HTML page, but without changing the model or even the controller this information can also be presented in a XML RSS feed [6].

1.3.4 DRY principle

The DRY principle (<u>D</u>o not <u>R</u>epeat <u>Y</u>ourself) [7] states that duplicate code must be avoided. Duplicate code or definitions can introduce a lot of errors, make maintenance harder and cost extra time during development.

1.4 Out of the scope

In order to achieve the goals that are listed section 1.1, trade-offs have to be made. In this section some of these trade-offs and constraints of *Rahisi* are further explained.

1.4.1 Efficiency

Efficiency is not a important characteristic for *Rahisi*. Real optimisations and fine-tuning is vital when a web applications has to serve thousands of visitors every hour. In contrast to small web applications, that often have a small amount of visitors. In addition a lot of resources can be saved by *caching*. There are plenty of caching systems that can save resources if they are low in a certain environment.

caching storing common accessed data, like webpages, to boost performance

1.4.2 The front end

The front end of a web application means everything within the scope of design or usability. The design is by good practice separated from the content of a *HTML* page. For example in a separate *CSS* stylesheet. The front end of a website is often done by different developers (or designers) than the ones that develop the back end. This is not surprising: front-end development is another discipline than back-end development. In addition to possible other aims like aesthetics or style, usability is an important factor for the front end of a web application. Once a system, that also controls the back end, makes assumptions about the front end, a designer or a usability expert can be obstructed. That is why *Rahisi* is not focussed on the front end. Another important reason to separate the front end from the back end, is the need to separate data (or content) and representation, like stated in subsection 1.3.3.

HTML markup language for webpages

CSS language to define the look and formatting of documents written in markup languages

1.4.3 Client side scripting

In *Rahisi* a clear choice is made about the *partitioning* of a web application: client side code is only used to enhance the user interface, thus for usability. In a large part of the websites today client side code is used sole for this purpose. In [1] a system is proposed that makes it possible to make the partitioning decisions as late as possible in de development process. This can have a lot of benefits. A reason to partition more responsibilities to the client, could be performance. For example: it could be more efficient to let a browser search trough a list of names instead of sending a new request to the server and executing a query on the database. As stated before, efficiency is not important for *Rahisi*, therefore the possibility of a late partitioning decision is not needed. Note that the choice could be made to send more data to the browser than the amount that is directly displayed, to make the web application more responsive. This is however a usability choice.
In *Rahisi* there is the ability to add client side code to templates, but *Rahisi* has no special features with respect to client side scripting.

partitioning the separation of the code between the client (web browser) and the server

1.4.4 Complex queries and database schemes

A relational database is managed with *SQL*. Using SQL, a database schema can be defined, a dataset can be queried and many other task can be done, like managing database users and privileges.
With SQL, extensive database schemes can be defined. *Rahisi* limits the possibilities when defining such a scheme. This is done for example in the amount of datatypes as we shall see in chapter 2.
Within SQL there are many techniques to select data from a database. The more complex a database structure, or application, the more complex queries can grow. For a straightforward, simple web applications these are seldom needed. The types of SQL queries most often used are listed below.

SQL standardised language designed to manage data in relational databases

- Select all rows, optionally with an offset, a limit and an order of the result.
- Select rows given the criteria of the item above, and a certain search expression.

- Select statistics like the maximum, minimum, count or average of a certain field in the database, with an optional search expression.

There is no need to specify a *join* in a SQL query because they are done automatically when a *foreign key* is specified in *Rahisi*. Chapter 3 will give a more precise definition of these queries.

join a SQL statement to combine two or more tables in a database
foreign key a referential constraint between two tables

1.4.5 Large scale web applications

One of the aims of *Rahisi* is to implement simple web applications (like a small blog for example). This means that large scale web applications, that are often more complex or need a whole other structure, are out of the scope of *Rahisi*. *Rahisi* limits the scale of the web application not by the size of the dataset that is accessible trough the web application, but by the complexity of the dataset. This is simply limited by the complexity of a database query, which has to be possible to define in *Rahisi*. By focussing only small and simple web applications, a smaller set of functionalities is needed. This has a lot of advantages, like a small learning curve and a more comprehensible system.

1.4.6 Non trivial data processing

A small or mediate website is often primarily focussed on information sharing. The difference between just sharing information and for example the extraction of information out of data, is the data processing involved. In *Rahisi* there are functionalities to render a piece of text to valid HTML or to translate a date to the preferred format. There is no possibility within the system to define new, custom ways to manipulate and modify data. This possibility will make the language, in which the controller or the view is defined, a lot more complex. Therefore it is out of the scope of *Rahisi*.

1.4.7 Custom architectures

The MVC architecture is the basis of *Rahisi*. In order to reach the aims stated in section 1.1 this architecture can not be changed. The possibility to construct a custom or more complex architecture will obstruct important characteristics of *Rahisi*, like a small learning curve or the comprehensibility of the system. For example, an architecture in which the database can be queries from template files, has to support some kind of SQL expression in the templates. This will compromise the loose coupling between the controller en the view that is described in subsection 1.3.1. Therefore the developer is obligated to work according to the MVC principle.

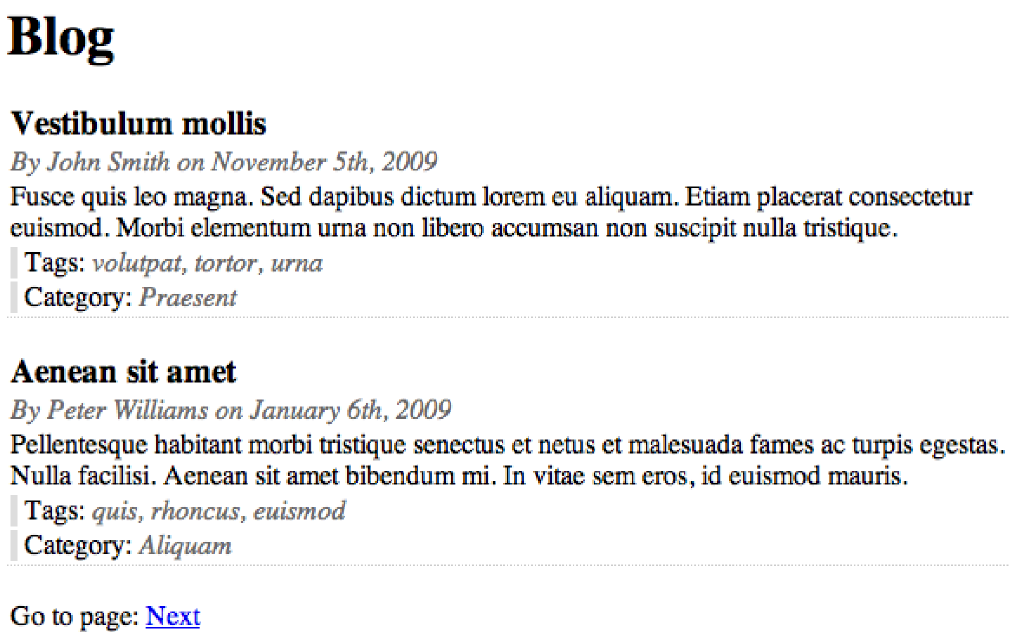
1.5 Example application

In order to describe and demonstrate *Rahisi* from now on, an example web application is used. This application is a basic blog that enables the owner to write new blog entries, and lists these entries (possibly separated over several pages).
Summarised, the blog application consists of the following functionalities.

- Display the most recent articles spread over several pages.
- Display the most recent articles of a certain category, spread over several pages.
- Add a new article.

In the example there are certain, more or less vital parts of the web application left out on purpose:

- **authentication** In order to only allow the owner of the blog to post new articles some way of authentication has to be implemented. Although it is relatively straightforward to implement a basic authentication in *Rahisi*, in the example application this is left out to make it less cluttered.
- **client side scripting** Like stated in subsection 1.4.3 client side scripting is by a convention of *Rahisi* only used for the user interface, therefore it has nothing to do with the operation of *Rahisi*. That is why it is left out of the example.


Blog

Vestibulum mollis

By John Smith on November 5th, 2009

Fusce quis leo magna. Sed dapibus dictum lorem eu aliquam. Etiam placerat consectetur euismod. Morbi elementum urna non libero accumsan non suscipit nulla tristique.

Tags: *volutpat, tortor, urna*

Category: *Praesent*

Aenean sit amet

By Peter Williams on January 6th, 2009

Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Nulla facilisi. Aenean sit amet bibendum mi. In vitae sem eros, id euismod mauris.

Tags: *quis, rhoncus, euismod*

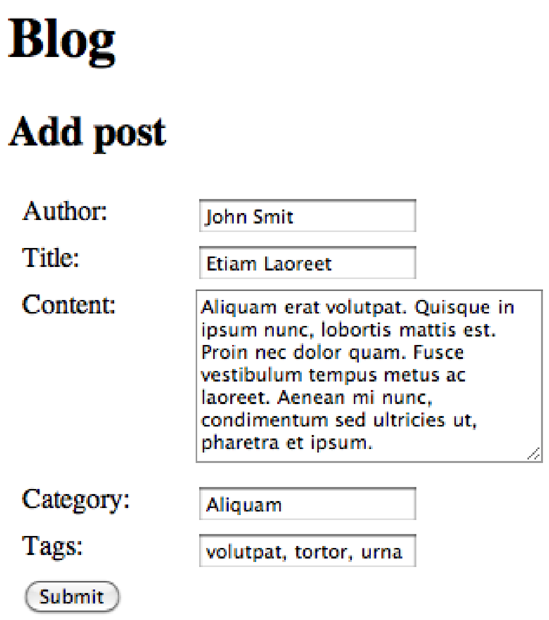
Category: *Aliquam*

Go to page: Next

Figure 1.2: Blog home

In figure 1.2 the *homepage* of the blog is illustrated. On the homepage the latest articles and their meta information, like *tags* or a publication date, are shown. To display this page, *Rahisi* needs to fetch the latest articles from the model. On the page a link is visible that enables the visitor to browse trough older articles.

tags keywords or terms assigned to pieces of information



Blog

Add post

Author:	John Smit
Title:	Etiam Laoreet
Content:	Aliquam erat volutpat. Quisque in ipsum nunc, lobortis mattis est. Proin nec dolor quam. Fusce vestibulum tempus metus ac laoreet. Aenean mi nunc, condimentum sed ultricies ut, pharetra et ipsum.
Category:	Aliquam
Tags:	volutpat, tortor, urna

Figure 1.3: Add blog entry

In figure 1.3 the HTML form is shown that enables the author(s) of the blog to add an article to the blog. This form can be reached by entering the associated *URL* in the address bar of the web browser (`http://www.example.com/add`). Most fields are common text input fields, except for the *tag* field. The tag field enables the author(s) to add or delete multiple tags using a list of comma-separated values.

URL Uniform Resource Locator, a label for a certain information resource on the web

Chapter 2

The model

The model is a definition and representation of the dynamic data that is accessible trough a web application. It implements a uniform access, and definition, of the data. The model can be seen as an abstraction to the database of the application.
In *Rahisi* the name 'model' both identifies the component of the system that implements the abstraction to the database, and the associated definitions of data in the database.
In *Rahisi* the model is only accessible trough the controller. The way model data is selected or queried will be described in chapter 3. In this chapter an overview will be given of the syntax and semantics of the model.

2.1 Structure

In *Rahisi* an abstraction is made on top of SQL. Repetition of information, or code, is a bad thing (see subsection 1.3.4). Therefore a 'database table' is only defined once. With an abstract table definition in *Rahisi*, a class, a class validation and a database table are created.
In order to understand the structure of the model, we will first define the terminology:

- **The model** is the part of *Rahisi* that implements a uniform access, and definition, of the dynamic data of a web application.

- The *model* consist of multiple **classes** that specify the *fields* of a set of *records* that share these properties. A class can be directly mapped to a *table* in the database.

- A **record** is a concrete data instance of a particular *class*. A *record* can be directly mapped to a row in a database table.

- A **field** is the name and type of a particular value in a *record* that is specified in the *class* of that *record*.

In addition to fields, in a class also extra options of that class and relationships to other classes are defined. Classes in *Rahisi* are comparable with classes in a *OOP* language, with the difference that there are no methods or inheritance in a *Rahisi* class. Records could be compared with instances of classes in a OOP language.

OOP stands for Object Oriënted Programming, a programming paradigm that uses data structures consisting of data fields and methods called objects

Classes have two main responsibilities: storing data, and maintaining data integrity. This last responsibility is realised in two ways:

- For each field a datatype is specified. This type will be converted to an appropriate database field type, that also makes the data storage efficient.

- Optionally the keyword '`required`', '`unique`', or a *regular expression* can be added to a field definition. Together with the field type, these options are used to verify data after a *form submission*. If a posted form does not contain data in the required format of the class, it will not be stored in the database.
  For example: when a new article is submitted in the example application, without a title specified, an error is shown, and the new post is not added.

regular expression string of characters written in a formal language that offers a precise and flexible way to match a text by a pattern

form submission the event of sending data in a HTML form via HTTP to a web server

2.2 Definition

The syntax of a class is based on YAML [5]. YAML is a serialisation standard that form a very suitable basis for the syntax of classes in *Rahisi*. This is because it is well readable for both humans and machines, and it has very little syntactical overhead. These characteristics are consistent with the aims of *Rahisi*, stated in section 1.1.

2.2.1 Types

The model of the blog application consists of three classes, namely *post*, *tag* and *category*. The definitions of the fields in these classes is shown below. For the illustration of the functionalities of a class, the classes of the example application will be presented in parts. A complete overview of the example classes will be given after this section.

```
Post:
    author: string(150)
    title: string(150)
    body: text

Tag:
    name: string(100)

Category:
    name: string(100)
```

The *post* class consist of three fields. After the name, the type of the field is specified. The fields *author* and *title* have type `string(150)`. This mean that they are stored as a sequence of characters with a maximum of 150 characters. The field *body* has the type `text`, this means that it can hold a large

sequence of characters, the limit of the amount of characters is dependent of the database implementation.
In subsection 1.4.1 we assumed that efficiency is not important. This is why *Rahisi* does not offer the choice between datatypes like `TINYINT`, `SMALLINT`, `MEDIUMINT`, `INT` or `BIGINT`. These help a database to store integers of a certain size as efficiently as possible. In *Rahisi* only the type `int` is available.
Another choice that is made with respect to table definitions is the fact that all rows are identified with a unique identifier. This identifier is a extra integer field called *id*. The value of this field will be automatically incremented for each new row that is inserted in the database. That makes the definition of a another *primary-key* unnecessary. *Rahisi* still offers the functionality to define a field as *unique*, as we shall see in section 2.4.

primary-key constraint on a database table, used to ensure that each row in a table can be uniquely identified

2.2.2 Form validation

In the following code example, after the types of the fields *author* and *body* the keyword `required` is added. This keyword tells *Rahisi* that in each record of class *post* these field are not allowed to be empty.
After the type of field *title* a regular expression is added. With a regular expression a more complex and precise verification can be specified. In this example the field *title* has to consist of al least 2 and up to 150 characters that either are *alphanumeric*, a underscore, a space or a comma. For a precise definition of the regular expressions that can be used in *Rahisi*, see [10].

alphanumeric a alphabetic or a numeric character

```
Post:
    author: string(150), required
    title: string(150), '[\w, ]{2,150}'
    body: text, required

Tag:
    name: string(100), '\w{2,100}'

Category:
    name: string(100)
```

2.2.3 Options of classes

Besides the field definition, a class can have another special field, called *options*. In this field two possible extra functionalities can be specified (`timed` and `comma-separated`). These options make it faster and easier to add common properties of a class.
The *post* class has the option `timed`. This keyword instructs *Rahisi* to add two extra fields to the class: a *created* and a *updated* field. The *created* field is set to the current date and time when a new records is created. The *updated* field is set to the current date and time every time the record is changed. The *created* and *updated* fields are not shown in the class, but can be addressed either in the controller, when querying a certain class, or in the view, to display them.
The option `comma-separated` adds the functionality that one or more records

can be stored from a single string of comma separated values. In the example the *tag* class has the option `comma-seperated`. When a *post* record is fetched, the tags also can be viewed as a comma separated list. This saves the developer a lot of code.

```
Post:
    options: timed

Tag:
    options: comma-separated
```

2.2.4 Relations between classes

As we shall see in chapter 3, when a record of the class *post* is fetched from the database, also all associated tags, and a category is fetched. These so called relations are specified with two keywords `has_one` and `has_many`. The `has_one` relation specifies that the class can have a relation with zero or one record of the class that is specified after the keyword. The `has_many` relation specifies that the class can have a relation with zero or *more* records of the class that is specified after the keyword. For example, a post can have an arbitrary amount of tags.

```
Post:
    has_one: Category, required
    has_many: Tag

Tag:
    has_many: Post

Category:
    has_many: Post
```

Typically when a post is added, one or more keywords and a category are added to.

2.3 Complete model of the blog example

```
Post:
    options: timed
    author: string(150), required
    title: string(150), '[\w, ]{2,150}'
    body: text, required
    has_a: Category, required
    has_many: Tag

Tag:
    options: comma-seperated
    name: string(100), '\w{2,100}'
    has_many: Post

```

```
Category:
    name: string(200)
    has_many: Post
```

2.4 Structure of a class definition

The model consist of a set of class definitions. Each class is declared with a name (starting with a upper case character). The following attributes can be specified in a class:

- **Field**
  A field consist of a field name and a type. The possible types are:

  a) **int** Integer value.
  b) **float** Real number value.
  c) **string(*n*)** String with maximum length *n*.
  d) **text** Text field.
  e) **date** Date field.
  f) **datetime** Date and time field.
  g) **enum(*values*)** An enum field can only contain one of the specified `int` or `string` values.

  Optionally the following properties can be added to a field:

  a) **required** The field may not be empty.
  b) **unique** The field has to hold a unique value between all records in the database.
  c) ***regex*** A specified regular expression that validates the field.

- **Option**
  Option is a special keyword. After this keyword the following options can be specified to add functionality to the class:

  a) **timed** The class automatically adds the *created* and *updated* fields.
  b) **comma-separated** When a class only consist of one string field, the keyword `comma-separated` adds the possibility to add multiple entries, separated by a comma, at once. When the class plays a `has_many` role in another model, it also can be fetches as a comma separated list.

- **Relation**
  After the keywords `has_one` and `has_many` a relationship with a certain class can be specified.

  a) **has_one** A record has at most one relation with a record of the specified class name.
  b) **has_many** The record has zero or more relations with a record of the specified model name.

2.5 Grammar

All grammars in this thesis will be given in a from resembling the Backus Naur Form [8]. All literal strings are surrounded by double quotes. Some nonterminal symbols have no further production rules in this grammar, these are explained below in natural language.

- `<number>` an integer number, such as -1 or 42.
- `<string>` a string of characters, without spaces or newlines.
- `<regex>` a Python regular expression (surrounded by quotes) [10].
- `<field name>` and `<class name>` strings that identify relatively a field and a class.

```
⟨model⟩ → ⟨class name⟩ ":" ⟨newline⟩ ⟨field⟩ ⟨newline⟩ ⟨newline⟩
⟨field⟩ → ⟨field⟩ ⟨newline⟩ ⟨field⟩ |
    ⟨field name⟩ ":" ⟨space⟩ ⟨type⟩
    ["," ⟨space⟩ ⟨field option⟩] |
    "options:" ⟨space⟩ ⟨option⟩ |
    "has_one:" ⟨space⟩ ⟨class name⟩ |
    "has_many:" ⟨space⟩ ⟨class name⟩
⟨type⟩ → "int"| "real"| "string(" ⟨number⟩ ")"
    "text"| "date"| "datetime"| "enum(" ⟨enum value⟩ ")"
⟨option⟩ → ⟨option⟩ "," ⟨space⟩ ⟨option⟩ |
    "timed" | "comma-separated"
⟨field option⟩ → ⟨field option⟩ "," ⟨space⟩ ⟨field option⟩ |
    "required" | "unique" | ⟨regex⟩
⟨enum value⟩ → ⟨enum value⟩ "," ⟨space⟩ ⟨enum value⟩ | ⟨string⟩
⟨newline⟩ → ⟨opt-space⟩ ⟨newline⟩ | "\n"
⟨space⟩ → ⟨space⟩ ⟨space⟩ | " "
⟨opt-space⟩ → ⟨space⟩ | ""
```

Chapter 3

The controller

The controller is a function. As input it receives a HTTP request and the output in a HTTP response.
When a user pushes a link or a button on a *Rahisi* web application, or when a URL to a *Rahisi* web application is entered in the address bar of the web browser, the browser sends a HTTP request to the web server. The web server passed this request to the controller. The controller manipulates the model, fetches data from the model and invokes templates. In other words, the controller handles all user requests and input of the web application.

3.1 Definition

In this section the functionalities of the controller in *Rahisi* is explained with the use of the example application. The example controller is presented in parts. On the end of this section a complete overview of the example controller is given.

3.1.1 Routing

The routing of the controller, defines what piece of code will be executed after a user pushes a button or enters a URL in the web browser.
In the controller each *route* is assigned with an *action*.

- An **action** is a set of commands that will process the data that is needed to display a page, it defines the manipulations on the model and the needed templates for a certain request.
- A **route** resembles a URL after the (top level) domain name, also called a path. A route differs from a path in the fact that it can contain variables.

A variable in a route is specified with a colon directly after a slash. This way a certain action can be triggered with one or more variables. For example, in the blog application a certain page can be viewed by requesting the URL `http://www.example.com/page/2`. In the route `/page/:page_number` the name starting with a colon matches all values until the next slash, or end of the route, and stores this value in a variable with the same name. In this case a variable named *page_number* is created with the value 2. Examples of request statements are shown below.

```
Request: /page/:page_number

Request: /category/:category_name
```

When a HTTP request is made *Rahisi* will try to match the requested path with a specified route (after the keyword `Request`). When a path matches more than one route, the first is chosen (the first match from the top). This is why the next code example matches all request that are not matched by any of the previous routes:

```
1 Request /:otherwise
2     Response: 404 error
```

3.1.2 Responses

At the bottom of each action a *response* is specified. A response consist of one or more names of templates, optionally preceded with a HTTP status code. Standard the HTTP status code will be 200 *(OK)*, other common responses, like 404 *(Not Found)* or 500 *(Internal Server Error)*, have to be specified explicitly. For example see line 2 of the code above.

3.1.3 Queries

In an action often one or more models are queried. For example:

```
Request: /
    view.posts = Post.getAll(10)
    Response: index
```

In the example above, the last 10 records are fetched from the model *post*. These are assigned to the variable `view.posts`. By preceding the variable with ''`view.`' the variable is automatically accessible in all view templates that are invoked by the action.
Possible queries in *Rahisi* are shown below. Brackets indicate that the enclosed parameter is optional.

- **getAll([*limit*[, *offset*[, *order*]]])**
  Fetch all records.
- **getWhere([*where*[, *limit*[, *offset*]]][, *order*])**
  Fetch all records, given a certain search criterium.
- **getCount([*where*])**
  Return the amount of records selected, given a certain search criterium.
- **getStat(*sum/avg/min/max*, *field*[, *distinct*[, *where*[, *order*]]])**
  Return a statistic.

The parameters that are used in these queries specify the following:

- **limit** An integer value specifying, the maximum amount of records that need to be fetched.

- **offset** An integer value specifying, the offset from where the next records have to be fetched, in the current ordering.
- **order** A field name and a direction to order the results by. For example ‘’`created desc`’ fetches records ordered by the creation date and time descending from new to old ones. Standard, records are ordered by the order that they are stored, but than reversed.
- **where** A statement consisting of the logical combinators (`and, or, not`), field names and variables, equality operators (`=, !=, <, >, <=, >=, %=`) and parentheses to change priorities. These equality operators are used in many programming languages and mathematics, except for `%=`, which matches strings according to a certain definition. This definition can contain a *wildcard* in the form of a percent character. This *wildcard* matches on all characters. The statement `"Arthur Dent" %= "%Dent"` will therefore result in the boolean value true.

The following parameters are used in the `getStat` query:

- **sum/avg/min/max** Respectively select the total sum, the average, the maximum or the minimum from a specified field. This only works on fields of type *integer*.
- **field** The field to calculate the statistic from.
- **distinct** The field to group the elements by.

In *Rahisi* queries can be applied to each others result. In the next code example this is illustrated with the resulting posts of a certain category.

```
view.posts = Category.getWhere(name = category_name).
    Post.getAll(10)
```

The query ‘`Category.getWhere(name = category_name).Post`’ returns a set of *post* records. The query ‘`.getAll(10)`’ selects the last 10 items from this set.

3.1.4 Save and delete

With the functions *save* and *delete* one can add, modify or delete records in the database. They can be applied in the controller to a specified model.
The function *save* adds a new record to the model, given the data in the current HTTP request. This data is always coming from a HTML form. If the identifier of a record is specified in a request, the thereby identified record will be updated with the data available.
The *delete* function expects one parameter, namely the identifier of the record that needs to be deleted.
Both the functions *save* and *delete* return the boolean value true on success and false when an error occurred.

3.1.5 Standard statements

In the controller, common programming statements can be used, like assignment, arithmetic operations and conditional statements. In the next example arithmetics are combines with an assignment.

```
offset = items_per_page * page_number
```

The following code illustrates a conditional statement.

```
if Post.save():
    view.message = 'New post saved.'
    Response: index
else:
    view.message = 'Error in form.'
    Response: add
```

3.1.6 Special functions

Besides normal statements and operations the following special functions can be used in a controller action. Brackets indicate that the enclosed parameter is optional.

- **setCookie(*name*, *value*[, *expire*])** Instruct the browser of the client via a HTTP response to store a *cookie* with the specified name, value and expiration time and date.
- **getCookie(*name*)** Return the value of the specified cookie.
- **mail(*to*, *from*, *subject*, *body*)** Send an email message to the specified recipient.
- **hash(*value*)** Return the *hash* of a certain variable. This function is often used to increase security in the web application.

cookie a text string that is stored by a web browser and returned in a HTTP request every time the browser requests a page of the server that sent the cookie

hash the result of an one-way function that takes an arbitrary block of data and returns a fixed-size bit string

3.2 Complete controller of the blog example

```
items_per_page = 10

Request: /
    view.posts = Post.getAll(items_per_page)
    Response: index

Request: /page/:page_number
    offset = items_per_page * page_number
    view.posts = Post.getAll(items_per_page, offset)
    Response: index

Request: /category/:category_name
    view.posts = Category.getWhere(name = category_name)
        .Post.getAll(items_per_page)
    Response: category
```

```

Request: /category/:category_name/page/:page_numer
    offset = items_per_page * page_number
    view.posts = Category.getWhere(name = category_name)
       .Post.getAll(items_per_page, offset)
    Response: category

Request: /add
    Response: add

Request: post /add
    if Post.save():
        view.message = 'New post saved.'
        Response: index
    else:
        view.message = 'Error in form.'
        Response: add

Request /:otherwise
     Response: 404 error
```

3.3 Custom code

Rahisi deliberately has limited functionalities. In order to still define operations that are not possible by default, *Rahisi* has the possibility to define custom Python code or custom SQL queries. To define custom code, a curly brace has to be inserted, followed by the language that is used (Python or SQL). For example, the following statement could be inserted in an action to transform a string to uppercase: `{python string = string.uppercase()}`. This functionality is not included in the grammar of the controller, because it is an extension of *Rahisi*.

3.4 Grammar

The grammar of the controller language is given in the same syntax as the grammar in section 2.5. The nonterminals explained in section 2.5 are also applicable, in addition to the nonterminal below.

- `<view name>` a string that identifies a view template.

$\langle$controller$\rangle \rightarrow \langle$constant$\rangle$ $\langle$newline$\rangle$ $\langle$newline$\rangle$ $\langle$action$\rangle$
$\langle$constant$\rangle \rightarrow \langle$constant$\rangle$ $\langle$newline$\rangle$ $\langle$constant$\rangle$ |
    $\langle$variable$\rangle$ $\langle$space$\rangle$ `"="` $\langle$space$\rangle$ $\langle$expression$\rangle$
$\langle$action$\rangle \rightarrow \langle$action$\rangle$ $\langle$newline$\rangle$ $\langle$newline$\rangle$ $\langle$action$\rangle$ |
    $\langle$request$\rangle$ $\langle$newline$\rangle$ $\langle$statement$\rangle$ $\langle$newline$\rangle$ $\langle$response$\rangle$
$\langle$request$\rangle \rightarrow$ `"Request:"` $\langle$space$\rangle$ $\langle$route$\rangle$
$\langle$route$\rangle \rightarrow \langle$route$\rangle$ $\langle$route$\rangle$ |
    `"/"` [`":"`] $\langle$string$\rangle$
$\langle$response$\rangle \rightarrow$ `"Response:"` [$\langle$space$\rangle$ $\langle$http response$\rangle$]

```
        ⟨space⟩ ⟨view name⟩
⟨statement⟩ → ⟨variable⟩ ⟨space⟩ "="
        ⟨space⟩ ⟨expression⟩ ⟨newline⟩ |
        "if" ⟨space⟩ ⟨boolean expression⟩ ⟨space⟩ ":" ⟨newline⟩
        ⟨statement⟩
        ["else:" ⟨newline⟩ ⟨statement⟩ "fi" ⟨newline⟩]
⟨expression⟩ → ⟨variable⟩ | ⟨math expression⟩ |
        ⟨string manipulation⟩ | ⟨model expression⟩ |
        "("⟨boolean expression⟩")"
⟨model expression⟩ → ⟨class name⟩ "." ⟨query⟩
⟨query⟩ → ⟨query⟩ "." ⟨class name⟩ "." ⟨query⟩ |
        "getAll(" [⟨number⟩ ["," ⟨space⟩ ⟨number⟩
        ["," ⟨space⟩ ⟨order⟩]]] ")" |
        "getWhere(" ⟨where⟩ ⟨space⟩ ["," ⟨space⟩ ⟨number⟩
        ["," ⟨space⟩ ⟨number⟩ ["," ⟨space⟩ ⟨order⟩]]] ")" |
        "getCount(" [⟨where⟩] ")" |
        "getStat(" ⟨stat⟩ ⟨space⟩ ⟨field name⟩
        ["," ⟨space⟩ ⟨field name⟩ ["," ⟨space⟩ ⟨where⟩
        ["," ⟨space⟩ ⟨order⟩]]]")" |
        "save()" | "delete(" ⟨number⟩ ")"
⟨order⟩ →⟨order⟩ "," ⟨space⟩ ⟨order⟩ |
        ⟨field name⟩ ⟨space⟩ "asc" | ⟨field name⟩ ⟨space⟩ "desc"
⟨where⟩ → "(" ⟨where⟩ ")" |
        ⟨field name⟩ ⟨space⟩ ⟨rel. operator⟩ ⟨space⟩ ":" ⟨variable⟩ |
        ⟨where⟩ ⟨space⟩ "and" ⟨space⟩ ⟨where⟩ |
        ⟨where⟩ ⟨space⟩ "or" ⟨space⟩ ⟨where⟩ |
        "not" ⟨space⟩ ⟨where⟩
⟨boolean expression⟩ → ⟨variable⟩ |
        "(" ⟨boolean expression⟩ ")" |
        ⟨boolean expression⟩ ⟨space⟩ ⟨rel. operator⟩
        ⟨space⟩ ⟨boolean expression⟩ |
        ⟨boolean expression⟩ ⟨space⟩ "and"
        ⟨space⟩ ⟨boolean expression⟩ |
        ⟨boolean expression⟩ ⟨space⟩ "or"
        ⟨space⟩ ⟨boolean expression⟩ |
        "not" ⟨space⟩ ⟨boolean expression⟩ | ⟨math expression⟩
⟨rel. operator⟩ → "=" | "<>" | "<" | ">" | "<=" | ">=" | "%="
⟨math expression⟩ → "(" ⟨math expression⟩ ")" |
        ⟨math expression⟩ ⟨space⟩ ⟨math operator⟩
        ⟨space⟩ ⟨math expression⟩ |
        ⟨number⟩ | ⟨variable⟩
⟨math operator⟩ → "+" | "-" | "*" | "\" | "**" | "%"
⟨variable⟩ → ⟨quote⟩ ⟨string⟩ ⟨quote⟩ | ⟨string⟩ | ⟨number⟩
⟨quote⟩ → """
⟨newline⟩ → ⟨opt-space⟩ ⟨newline⟩ | "\n"
⟨space⟩ → ⟨space⟩ ⟨space⟩ | " "
⟨opt-space⟩ → ⟨space⟩ | ""
```

Chapter 4

The view

The view consist of a set of templates. Templates are rendered to create a representation of the data passed by the controller, in possibly one ore more formats.
A typical template is a HTML file that renders a webpage around the dynamic data that is passed by the controller.

4.1 Syntax

View templates can consist of all possible *unicode* characters, with one exception. Curly braces are 'special' characters used by *Rahisi* to define the location of dynamic content in the document. To get around this restriction, two curly braces can be used. From the point in the text that two curly braces are opened, all curly braces will be left alone, until two closing curly braces are encountered.
Like stated before, between curly braces dynamic content can be specified. Examples of dynamic content are variables, manipulated variables or an other template that needs to be inserted. The following section will give a description of the statements that can be used in a template.

unicode industry standard for the consistent representation of almost all characters of all languages

4.2 Statements

Functions in a template need to be placed between two curly braces.

- **{if *condition*} .. {else} .. {end}**
  Conditional statement. The *else* clause is optional.
- **{*variable*.raw}**
  Return the preceded variable raw, without automatic *HTML escaping*. All other variables will be escaped, so that they are rendered properly on a HTML page.
- **{partial *view name*}**
  Render and insert a view template.
- **{*variable*.format(*date format*)}**
  Format a date or a date and time.

HTML escaping convert characters and structure in text so that it will be displayed correctly on a HTML page

- **{foreach(*set* as *variable*)} .. {end}**
  Loops trough a dataset for each row in that dataset, wile assigning a specified variable to the current value in that dataset.

4.3 Templates of the blog example

In the example below the index template is shown. This template is invoked to display the *home page* of the blog. The only code that is present in this template invoke other templates: a header template, a template to display blog posts, etcetera. The first line is a HTML comment with the filename of the template.

```
<!-- index.tpl -->
{partial header}
{partial post}
{partial pagination}
{partial footer}
```

The following example contains a variable. A variable is a string (without spaces) surrounded by curly braces, and is assigned in the controller. The view will standard escape text variables, so that proper HTML is inserted.

```
<!-- category.tpl -->
{partial header}
<h2>{category}</h2>
{partial post}
{partial pagination}
{partial footer}
```

The next template holds a few new functionalities. On line 2 the `{foreach(..)}` structure will loop trough every element in the dataset *posts*. In every iteration the variable *post* will be assigned to a new row. Everything between the foreach statement and the end of the loop, indicated with `{end}` will be repeated in the resulting webpage.
On line 5 the variable *post.created* that holds a date and time is formatted the the specified date format with the `{format(..)}` function.
On line 8-10 everything between the `{if ..}` and the `{end}` statements are only shown if the variable *category_name* is set.

```
 1 <!-- post.tpl -->
 2 {foreach posts as post}
 3 <div>
 4     <h3>{post.title}</h3>
 5     <p>By {post.autor} on {post.created.format(d-m-Y)}</
          p>
 6     <p>{post.body}</p>
 7     <p>Tags: {post.tags}</p>
 8     {if category_name}<p>Category: {category_name}</p>{
          fi}
 9 </div>
10 {end}
```

The next template combines the *if* control structure and variables with arithmetic operations.

```
<!-- pagination.tpl -->
<p>Go to page:
{if pagenumber > 1}<a href="/{if category_name}category
   /{category_name}/{end}page/{pagenumber - 1}">Previous
   </a>{end}
<a href="/{if category_name}category/{category_name}/{
   end}page/{pagenumber + 1}">Next</a>
</p>
```

The following code holds a HTML form, which is used to add a new entry to the blog.

```
<!-- add.tpl -->
{partial header}
<h2>Add post</h2>
<form action="/add" method="post">
    <p>Author: <input type="text" name="author" /></p>
    <p>Title: <input type="text" name="title" /></p>
    <p><textarea name="body"></textarea></p>
    <p>Category: <input type="text" name="category" /></
       p>
    <p>Tags: <select size="10" name="tags[name]"
       multiple="multiple"></select></p>
    <p><input type="submit" value="Submit" /></p>
</form>
{partial footer}
```

When a page is requested that does not exists, a HTTP 404 response is returned. In addition the following HTML page is returned, that displays the error in the browser of the client.

```
<!-- error.tpl -->
{partial header}
<h2>Error</h2>
</p>Page not found.</p>
{partial footer}
```

The following templates hold the header and the footer of the blog. These are needed to generate a proper HTML page.

```
<!-- header.tpl -->
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
   Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd
   ">
<html>
<head>
        <title>Blog</title>
</head>
<body>
```

```
<h1>Blog</h1>
{if message}<p>{message}</p>{end}
```

```
<!-- footer.tpl -->
</body>
</html>
```

4.4 Grammar

The following grammar uses the same syntax as the model and the controller grammars. The nonterminals explained in section 2.5 and 3.4 are also applicable, in addition to the one below.

- `<date format>` a date format string, given the directives of the Python `strftime` function [11].

```
⟨template⟩ → ⟨text⟩
⟨text⟩ → ⟨text⟩ ⟨text⟩ |
     ⟨string⟩ | "\n" | " " | ⟨statement⟩
⟨statement⟩ → "{foreach" ⟨variable⟩ ⟨space⟩ "as"
     ⟨space⟩ ⟨variable⟩ "}" |
     "{if" ⟨boolean expression⟩ "}" | "{else}" | "{end}" |
     "{" ⟨variable⟩ "}" | "{partial:" ⟨space⟩ ⟨view name⟩ "}"
⟨variable⟩ → ⟨math expression⟩ |
     ⟨variable⟩ "." ⟨manipulation⟩ |
     "range(" ⟨number⟩ ["," ⟨space⟩ ⟨number⟩
     ["," ⟨space⟩ ⟨number⟩]]] ")" |
     ⟨string⟩ | ⟨string⟩ "." ⟨string⟩ | ⟨number⟩
⟨manipulation⟩ → "substr(" ⟨number⟩
     ["," ⟨space⟩ ⟨number⟩]]")" |
     "date(" ⟨date format⟩ ")"
⟨boolean expression⟩ → ⟨variable⟩ |
     "(" ⟨boolean expression⟩ ")" |
     ⟨boolean expression⟩ ⟨space⟩ ⟨rel. operator⟩
     ⟨space⟩ ⟨boolean expression⟩ |
     ⟨boolean expression⟩ ⟨space⟩ "and"
     ⟨space⟩ ⟨boolean expression⟩ |
     ⟨boolean expression⟩ ⟨space⟩ "or"
     ⟨space⟩ ⟨boolean expression⟩ |
     "not" ⟨space⟩ ⟨boolean expression⟩ | ⟨math expression⟩
⟨rel. operator⟩ → "=" | "<>" | "<" | ">" | "<=" | ">=" | "%="
⟨math expression⟩ → "(" ⟨math expression⟩ ")" |
     ⟨math expression⟩ ⟨space⟩ ⟨math operator⟩
     ⟨space⟩ ⟨math expression⟩ |
     ⟨number⟩ | ⟨variable⟩
⟨math operator⟩ → "+" | "-" | "*" | "\" | "**" | "%"
⟨space⟩ → ⟨space⟩ ⟨space⟩ | " "
```

Chapter 5

Implementation

This chapter will give a description off the implementation of *Rahisi*. First, the languages and software that is used to implement *Rahisi* is introduced. Second, an explanation of the implementation of the model, controller and view will be given. The general idea of *Rahisi* is that the files that define the view, the model and the controller are first compiled to a program, separated over several Python scripts. This program is embedded in a web server. The result is a working web application.

5.1 Technologies

5.1.1 Python

Rahisi is implemented using Python to translate *Rahisi* code to new Python code. The language Python is chosen because of the high level of abstraction and the availability of many applicable libraries. Examples of the libraries used in *Rahisi* are the PyYAML library, to parse YAML class definitions, or the mod_python [15] module that embeds *Rahisi* in the Apache web server [16]. Another candidate language that also offers these possibilities is PHP, but the fact that is is weakly typed and the lack of decent parsing libraries, makes Python a better choice.

5.1.2 Apache

Apache is the most used web server [17]. Due to the wide range of functionalities, and the fact that it is open-source, the popularity of Apache keeps growing. The possibility to integrate a Python application seamlessly within the Apache web server makes it an obvious choice for *Rahisi*. In a *Rahisi* website, all the *static content* is directly served to the client by the Apache web server. All requests to non-static content are generated by the Python scripts of *Rahisi*, and served by Apache.

static content all content on a webpage that is not dynamically generated, .i.e. all images, stylesheets and documents used or accessible trough the website

5.1.3 SQLite

The database implementation used by *Rahisi* is SQLite [18]. SQLite is open-source database management system that can be integrated within Python in many ways. SQLite is the suited database for *Rahisi* due to good performance and many functionalities, in addition to the many bindings to Python. In the implementation of *Rahisi* a DB-API 2.0 interface for SQLite databases [13] is used. Also the fact that SQLite does not need any configuration (like setting up database users) makes it fitted for *Rahisi*. But *Rahisi* is not dependent on SQLite. The abstraction to database implementations that Python delivers, makes it fairly easy to switch to an other database.

5.2 The model

The model in *Rahisi* is defined in a syntax that is based on YAML [5]. It is implemented by parsing the class definitions and subsequently generation the table definitions for the SQLite database and Python classes that define each class of the *Rahisi* model. A class is parsed using the PyYAML module [12]. The generation of SQLite table definitions are obvious, except for the relations.
Each class is translated to a table, each field in that class is translated to a database table field. To each table an `id` field is added, like stated in 2.2.1. The SQLite type of the field is derived from the *Rahisi* type. For example `string(200)` is translated to `text`. If there are any options specified, extra fields are added. For example, when the option `timed` is specified, two extra `integer` fields are added that hold timestamps, called `created` and `updated`. If a `has_one` relation is added to a class, a field is added to the table definition with a foreign key to the `id` of the referenced table. A `has_many` relations causes *Rahisi* to generate a extra table. This table is used to define relations between two tables, and consist off two fields that each hold the `id` of one of the tables. The makes it possible that for each row, multiple relations to other rows can be stored in the database.

5.3 The controller

The controller is, like the model and the view, parsed by *Rahisi* and translated to Python code. *Rahisi* uses the Python SimpleParse module [14] together with the grammar of the controller (see section 3.4). The SimpleParse module parses the controller file to create a tree structures describing the structure and statements of the controller file. The resulting tree is than translated to Python statements.
Many statements in the *Rahisi* controller can be translated to Python code directly, like assignments. Queries on the other hand, have to be translated to valid SQL(ite) queries, that are embedded in the Python code.
Rahisi translates each action in the controller to a function. Code is generated that maps a HTTP request to the appropriate action function. In the action function the body of the action (translated to Python code) is placed. The response statement in *Rahisi* is translated to a call to the generated view class, that handles the templates.

5.4 The view

The view part of *Rahisi* consist of:

- A Python class, that passes the variables from the controller to the appropriate templates;
- A set of templates, consisting of pure Python code and text (like HTML or XML).

The templates are translated from *Rahisi* templates. All *Rahisi* statements in the templates are surrounded by curly braces. A Python script, that also uses the SimpleParser module, parses these statements and translates the resulting tree to new Python statements. If a partial template is called, the view class ensures that the referred template is inserted.

5.5 Deployment

A typical *Rahisi* application consist of only a few files. A model file, a controller file and a directory with a set of templates. To deploy a *Rahisi* application, these files are first parsed and translated to Python code and SQL queries and table definitions. The tables are created in a SQLite database. Together with the Apache web server it constitutes a complete working web application.
If the *Rahisi* applications has to be edited, the affected parts can be generated again. If the structure of the classes in the model is edited, the affected tables we be altered accordingly.

Chapter 6

Related work

Like stated before, there are dozens of programming or scripting languages to choose from when developing a web application. On top of them dozens or even hundreds of frameworks have been created to streamline the development- and maintenance precess.
A few that are relevant to mention in relation to *Rahisi* are *Ruby on Rails* [19], *CakePHP* [20] and *Django* [21]. These frameworks all follow the MVC model, like *Rahisi*.

- **Ruby on Rails** is a web framework based on the language Ruby. It is designed to create practical web applications, with less code, simplicity and less configuration files than other frameworks. Ruby on Rails also makes *meta-programming* possible. This makes a higher abstraction level in the code possible. A higher abstraction makes the code more comprehensible.

meta-programming functions in a programming language that can be used like statements

- **CakePHP** is a web framework based on the web scripting language PHP. It aims to *"enable PHP users at all levels to rapidly develop robust web applications"*. In contrast with Ruby on Rails and Django it does not have an extensive abstraction to the database, using models.

- **Django** is a Python based web framework. Between the three web application frameworks mentioned, Django most resembles *Rahisi*. It also has a high abstraction to database tables in the form of models. Django aims to make the creation of complex, database driven web application more easy and faster.

In contrast with the frameworks mentioned above, in *Rahisi* a web application is not written in a existing scripting language. The definition languages used by *Rahisi* are especially designed for their purpose: to make the development of simple web applications as fast, easy and with the least code as possible. *Rahisi* is a research in restricting the possibilities and functionalities when developing a web application, and thereby decreasing the amount of code that is needed while increasing the comprehensibility. This is unlike the frameworks mentioned above, that aim to delivering as much functionalities as possible. They are designed to make the development of complex web applications more simple, *Rahisi* is designed to make the development of simple web applications even simpler.

Chapter 7

Conclusion

In this thesis a profound introduction is given to *Rahisi*. *Rahisi* is a system, or rather an experiment with the aim of making the development of simple web applications, easier and faster.
The *Rahisi* experiment has been successful in decreasing the minimal amount of code needed to define simple dynamic web applications. This is done using three special definition languages, each with a special purpose for every part in the system. The limited set of statements used by *Rahisi* make a web application more comprehensible and thereby ease the development and maintenance.
The inevitable disadvantage of *Rahisi* is the finite expressiveness of the languages used in the system. *Rahisi* only delivers the functionalities of common web application, like storing and querying data from a database and insert the dynamic content in HTML pages. Another price that needs to be payed is the fact that three little definition languages have to be learned. When developing a web application with *Rahisi* these special languages have to be used, and not in an existing language, like PHP, Ruby or Python. This exclude the possibility to use external libraries, or existing pieces of code.

Bibliography

[1] Avraham Leff and James T. Rayfield *Web-Application Development Using the Model/View/Controller Design Pattern* IEEE Computer Society, Los Alamitos, CA, USA, 2001

[2] Alan Knight, Naci Dai *Objects and the Web* IEEE Software, pp. 51-59, March/April, 2002

[3] Richard J. Turver, Malcolm Munro *An early impact analysis technique for software maintenance* Journal of Software Maintenance: Research and Practice, vol. 6, 1994, pag. 35-52

[4] Tim Berners-Lee, James Hendler, Ora Lassila, *The Semantic Web* Scientific American, ISSN 0036-8733, vol. 284, 2001, pag. 34

[5] YAML(AML Ain't Markup Language) `http://www.yaml.org/`

[6] RSS (Really Simple Syndication) `http://validator.w3.org/feed/docs/rss2.html`

[7] DRY principle (Do not Repeat Yourself) `http://c2.com/cgi/wiki?DontRepeatYourself`

[8] Backus Naur Form `http://www.ietf.org/rfc/rfc2234.txt`

[9] The Python programming language `http://www.python.org`

[10] Python regular expressions `http://docs.python.org/library/re.html`

[11] Python time formatting `http://docs.python.org/library/time.html`

[12] Python YAML implementation `http://pyyaml.org/`

[13] Python SQLite module `http://docs.python.org/library/sqlite3.html`

[14] Python SimpleParse `http://simpleparse.sourceforge.net`

[15] Apache/Python Integration `http://www.modpython.org/`

[16] Apache HTTP Server Project `http://httpd.apache.org/`

[17] Netcraft January 2010 Web Server Survey `http://news.netcraft.com/archives/2010/01/`

[18] SQLite `http://www.sqlite.org/`

[19] Ruby on Rails web framework `http://rubyonrails.org/`

[20] CakePHP PHP web framework `http://cakephp.org/`

[21] Django Python web framework `http://www.djangoproject.com/`