technische universität dortmund

Universität Rostock

Traditio et Innovatio

Diploma Thesis

A Thin Web Application Framework based on Server-side JavaScript

Jan Varwig

July 20th, 2009

Supervisor: Prof. Dr. Clemens Cap
Lehrstuhl für Informations- und Kommunikationsdienste
Fakultät für Informatik und Elektrotechnik
Universität Rostock

2nd Examiner: Prof. Dr. Dietmar Jannach
Lehrstuhl für Dienstleistungsinformatik
Fakultät für Informatik
Technische Universität Dortmund

Contents

1 Motivation **1**
1.1 A Short History of the Web 1
1.2 Current Problems in Web Development 3
1.2.1 Heterogenous Languages 4
1.2.2 Redundancy in Code 5
1.2.3 Glue Code 5
1.2.4 Object-Relational Mapping 6
1.3 Use Cases 7
1.3.1 Model Validation 7
1.3.2 Data Transformation 8
1.3.3 Exposure of Objects Through APIs 9
1.3.4 Dynamic Object Properties 9
1.3.5 Storage and Retrieval of Compound Objects 10
1.4 Goals 11
1.4.1 A Single Language 11
1.4.2 Code Reuse between Server and Client 11
1.4.3 Elimination of Glue Code 11
1.4.4 Tight Integration of a JSON Database 12
1.5 Existing Single-Language Frameworks 12
1.6 Key Ideas 13
1.6.1 A Design Based on JavaScript 13
1.6.2 Key Ideas Put to Work 15

2 Background **19**
2.1 Existing Approaches to Web Frameworks 19
2.1.1 Request-/Action-Based MVC 19
2.1.2 Component Based 20
2.1.3 Free Form 21
2.2 Web-Applications vs. Web-Services 21
2.3 JavaScript 22
2.4 Changes in Recent Years 24
2.4.1 Web 2.0 and AJAX 24
2.4.2 JavaScript Engines 25
2.4.3 Approaches to Database Systems 26
2.5 Protocols for Web Services 27
2.5.1 SOAP 28
2.5.2 REST 28
2.5.3 XML-RPC / JSON-RPC 29
2.6 Existing JavaScript Based Frameworks 29

2.6.1 Helma 29
2.6.2 Persevere 30
2.6.3 Aptana Jaxer 31
2.6.4 Others 31

3 Design 35
3.1 Key Ideas 35
3.1.1 Avoiding Data Transformation and Lightweight Creation of Models 35
3.1.2 A Thin Controller and Uniform Models 36
3.1.3 Code Sharing, Code Splitting and Delegation 39
3.2 Object Infrastructure Design 41
3.2.1 Controllers 41
3.2.2 Models 43

4 Implementation 47
4.1 Choice of Back-End Software 47
4.1.1 JSON Databases Compared 47
4.1.2 The ServerJS Group 49
4.1.3 JavaScript Engines Compared 50
4.2 Client-Server Scenarios 53
4.2.1 Server-side 54
4.2.2 Client-Side 57
4.2.3 Unimplemented Scenarios 59
4.3 Implementation of the Server-side 60
4.3.1 The Underlying Platform: Rhino, Narwhal and Jack 61
4.3.2 Dispatcher 63
4.3.3 Models 64
4.3.4 Controllers 68
4.4 Implementation of the Client-side 68
4.4.1 Client-side Module Loading 69
4.4.2 Splitting 69
4.4.3 Support for Nested Structures 73
4.4.4 Automated Delegation 74
4.4.5 Example Application 74

5 Evaluation 75
5.1 Goals and Design 75
5.1.1 A Single Language 76
5.1.2 Code-Reuse Between Server and Client 76
5.1.3 Elimination of Glue Code 77
5.1.4 Tight Integration of a JSON Database 78
5.2 Use Cases Revisited 78
5.2.1 Model Validation 79
5.2.2 Data Transformation 79
5.2.3 Exposure of Objects Through APIs 80
5.2.4 Dynamic Object Properties 81
5.2.5 Storage and Retrieval of Compound Objects 81

5.3 General Architecture Evaluation 82
5.4 Perspectives and Problems 83
5.4.1 Remaining Tasks 84
5.4.2 Future Work 86

6 Conclusion **89**

A Kupo Information **91**
A.1 Requirements 91
A.2 Installation 91
A.3 Startup 92
A.4 Tests 93
A.5 License 93

Bibliography **102**

Preface

This thesis deals with a subject that is an important problem in modern, practical web application development. It is, however, only sparsely recognized in computer science.

Although a lot of sound and intelligent publications and discussions around the topic exist, they often occur outside academically accepted platforms. This leads to a lack of citable sources from established journals and proceedings, since many relevant or interesting publications take the form of articles in online magazines, weblogs or knowledge bases.

This thesis uses and references technologies that are very young and not well established yet. Unlike software systems devised at universities, which are usually introduced at scientific conferences or in scientific journals, most of these technologies were created in commercial or open-source environments and are documented informally on the web. As an example, even the initial specifications of established technologies like HTTP and HTML were never formally published.

Ignoring the existence of these sources would drastically inhibit exploration and innovation in an important area of modern software development. It was therefore decided to use sources for this thesis even if they have not been formally published. To account for the volatile nature of the web, cited sources that are referenced as URLs only, are attached to this thesis, stored on a DVD-ROM in HTML or PDF format.

The fact that the subject has its roots in web application development practice leads to an emphasis on the prototype implementation described in chapter 4. The practical aspects of this thesis are necessary, however, to illustrate how the presented concepts work together.

Chapter 1 Motivation

Recent developments in JavaScript interpreter technology are opening up new possibilities to use the language for server-side web development.

In this thesis, the design and implementation of a web application framework based on server-side JavaScript will be presented that provides a new approach to web application architecture on a lean stack of proven web technologies. This design aims to provide an answer to current problems in web development.

This chapter provides an overview of these problems. Afterwards, the design for a new framework, derived from these problems, will be presented.

1.1 A Short History of the Web

Before the *World Wide Web* reached its current level of popularity, it began its life in the beginning of the 90s at CERN as a minor protocol alongside the older eMail system, the Usenet, Gopher, and FTP, intended for the exchange of documents between scientists. The commercialization of the Gopher system in 1993 led to a rise of popularity for the World Wide Web as a method of distributing information [Yen93].

In its first published version 0.9, the *HTTP* protocol (*HyperText Transfer Protocol*) already mentioned a way for “hypertext nodes” (entities corresponding to todays web pages or *resources*) to provide dynamic content, based on a string appended to the URL for querying document indexes. The accompanying first version of *HTML* (*HyperText Markup Language*) did not specify forms; instead, the `<ISINDEX>` tag was used to indicate that a document was an index document, leaving it up to the browser to generate the appended string [W3C92]. The *HTML+* standard first mentioned form elements for HTML documents and the manual extension of URLs with search strings to enable a more varied way of generating queries to be send to a server [Rag93].

In the mid-nineties, several important technological developments and a huge increase in public interest moved the web beyond being a simple delivery platform for static documents, towards the dynamic landscape of applications and services it has become since:

In 1995, JavaScript was introduced to the public when Netscape released its *Navigator 2.0* Browser [Cha01]. The *Common Gateway Interface* (CGI) [W3C99] was specified, and *PHP* and *Perl 5* were released. CGI, Perl and PHP made the generation of web pages on the server more dynamic, while JavaScript made the display and interaction of generated pages in the browser more dynamic. The *MySQL* database was released in Version 1.0 and quickly became a popular for sites driven by Perl and PHP. In 1996, the HTTP/1.0 protocol was published, and introduced the POST method, enabling clients to not only query data, but send data to the server as well [BLFN96]. In the late 1990s, public attention to the internet grew and more and more businesses went online.

The next leap happened in the early 2000s when again several important technologies appeared simultaneously. In 1999, Microsoft introduced the *XMLHttpRequest* API in Internet Explorer 5 as a way to exchange data with a server without reloading the entire displayed page. By 2005, Opera, Mozilla Firefox and Safari all had adopted the API, enabling web developers to develop website interaction in a whole new way [Ope09, Moz05, App05]. One of the most prominent early uses of this technology was Google's *Gmail* service.[1] During 2004 and 2005 the terms *AJAX* and *Web 2.0* were coined [Gar05, O'R05]. AJAX, the abbreviation for *Asynchronous JavaScript and XML*, nicely summarizes what the new kind of web applications had in common: the use of asynchronous JavaScript (implemented by the XMLHttpRequest API) on the one hand and *XML* (*Extensible Markup Language*) or other forms of structured data on the other hand.

The use of structured and well-formatted data enabled the building of *web services*, targeting machines instead of human users. Protocols like *XML-RPC* and *SOAP* were established (see Section 2.5), turning the web into an interactive network of services communicating with each other [ACKM04, RR07]. The development of these sites and services was widespread and complicated enough to nourish a large open-source ecosystem of frameworks and development tools for both the client- and server-side of web applications.

The rise in popularity of the *Web 2.0* raised the need for rapid, yet structured development platforms, something neither the existing enterprise frameworks (too complex) or custom scripting (not structured and pow-

[1] `http://www.gmail.com/`

erful enough) could provide. Newly developed frameworks, written in dynamic, interpreted languages, were designed to address these issues, the most prominent examples being the *Zend* Framework[2] (PHP), *Django*[3] (Python) and *Ruby on Rails*[4] (Ruby). At the same time, client-side JavaScript frameworks were coming up (or gaining new attention) that made it easier to use the power of browser scripting. These frameworks provide simple APIs around the most used features and an abstraction layer from the lower level APIs of the browsers, whose incompatibility in certain details (like event propagation and DOM navigation) prevented JavaScript from being used to its full potential in the past. Popular examples are *jQuery*,[5] *Prototype*,[6] or *mootools*[7].

These frameworks and techniques made way for web applications that feel like desktop applications, with fluid user interaction in the browser, communicating with the server through remote procedure calls (*RPC*) or REST interfaces (*Representational State Transfer*, see Section 2.5), in the same way web services exchange information, and enable the user to create and edit complex information, no longer limited to simple database queries.

1.2 Current Problems in Web Development

Although the boost in attention to web development in recent years made the tools and frameworks that software developers rely on easier to work with, a few problems still persist. A common cause for these problems is, that the core protocols of the web and many of the web's components were intended for simple content delivery and not for the complex communication requirements of an application platform.

The development of the components, databases, JavaScript, XML and HTTP, was disjoint and methods for cooperation between them were developed by need during the boom of the web in the late nineties.

Although many view this organic growth as one of the web's strengths (see [RR07]), it also has severe drawbacks like

- Isolated applications due to a lack of standards. Although many problems in web development are reoccurring in almost identical forms (e.g. authentication systems), developers solving these problems find themselves reinventing the wheel over and over.

[2]`http://framework.zend.com/`
[3]`http://www.djangoproject.com/`
[4]`http://www.rubyonrails.com/`
[5]`http://jquery.com/`
[6]`http://prototypejs.org/`
[7]`http://mootools.net/`

- As a result, many solutions are incompatible with each other, exacerbating the problem further.
- Where standards exist, their implementations are not always complete or consistent. A lot of the standards in web development, most importantly CSS (*Cascading Style Sheets*), the DOM (*Document Object Model*) and their JavaScript APIs, lack reference implementations. This led browser vendors to come up with their own, incompatible solutions to unclear details. These incompatibilities became one of the largest problems in web development [Koc09].
- The high overhead of opening stateless HTTP connections was no problem when the web was used to retrieve simple files, but became an issue when web applications and web services began exchanging data in smaller packets with higher frequencies [Zyp08a].

Many tools have been combined over the years to enhance the capabilities of servers and clients and to circumvent these drawbacks of the web architecture and the HTTP protocol. A modern web application includes a web server, an application server, the server-side part of the application, a database, gateways from its back-end to other services, the HTTP protocol, a scriptable browser, markup languages like HTML and XML and possibly even *Flash* or *Java* applets.

This mix of technologies promotes the problems outlined in the following sections.

1.2.1 Heterogenous Languages

The different parts that make up a web application have to communicate with each other, but unlike the stacks that developers of desktop applications are usually using (like the *Java* class library, the Microsoft *.NET* environment or the *Qt* library), these parts are all developed in different languages:

- Documents are written or generated in *HTML / XML* and styled with *CSS* and / or *XSLT*.
- *JavaScript* is used to give the application a more sophisticated interface.
- *SQL* is used to access the database.
- The server-side part of the application is written in *Java*, *Python*, *Ruby*, *PHP*, *ASP* or any other programming language.
- Transmitting data from the server to the client requires knowledge about the details of the *HTTP protocol* for most nontrivial cases.

The heterogenous nature of the web has been identified as a problem for web service interoperability in [CNW01].

Although this mix of technologies is a lot to learn and keep in mind, for application development it is more of an inconvenience than a fundamental problem.

However, it leads to more serious issues.

1.2.2 Redundancy in Code

The use of different languages in the layers of an application means that code can not be shared between these layers. This leads to redundancy and potential duplication of (sometimes critical) parts of the application. A prominent example for this problem is that of data validation [Lho06a].

Web applications usually accept data from the user through HTML forms. To ensure that the entered data is valid, it has to pass checks for consistency, value ranges or adherence to other criteria. If these *validations* are implemented multiple times, effort has to be spent to ensure all implementations are semantically equal.

1.2.3 Glue Code

Code that is needed to combine loosely coupled components together to form a working system is commonly referred to as *glue code* [DGHK05]. Glue Code does not carry meaning in regard to the problem an application is designed to solve, yet has to be written to make parts of the application interoperate.

Approaches exist to prevent developers from having to write glue code and to deal with the associated problems. These approaches either involve reducing the flexibility of the components involved (thus reducing the amount of necessary glue code) or to generate glue code automatically [vGB01, DGHK05].

Glue code is usually a symptom of a language that lacks expressive abilities or mechanisms for reflection and self-adjusting but can also be a result of a library or framework whose design is not completely congruent with the way it is actually used.

Glue code adds visual and structural "noise" to the program code, making it more laborious to write, harder to read, and bugs more difficult to discover. In [vGB01], Bosch and Gurp point out that "flexibility comes at the price of increased complexity" and that "glue code tying together the small components is not reusable". A good example for glue code in web development are the extensive configuration files needed to set up the *Hibernate* ORM framework [KBA+09].

1.2.4 Object-Relational Mapping

The term *Object-Relational Mapping* (*ORM*) describes the mapping of object-oriented data structures to tables of a relational database system (*RDBMS*) [Fow02]. Although object-orientation has been a dominant paradigm in software engineering for many years, the use of relational databases is still very common for the implementation of persistence layers in web development [Bar01]. The fundamental differences between the paradigms of object-oriented and relational technologies that make ORM necessary are called the *Object-Relational Impedance Mismatch* [IBNW09].

There is a wide selection of tools and frameworks available that implement ORM, such as *Hibernate* (Java), *ADO .NET* (.NET), *SQLAlchemy* (Python) or *ActiveRecord* (Ruby). Despite the existence of these tools, ORM is still considered a hard, multi-faceted problem whose solutions are complicated, error-prone and full of compromises.

Good examples for this are inheritance and associations. Both are essential features of object-orientation, can not be easily represented in relational databases and have to be modeled explicitly [New06, IBNW09]. Mapping associations to relational schemas can be used well to illustrate the problems caused by the impedance mismatch.

One-to-many associations in object graphs are usually formed by pointers from a collection to its members, while the members themselves can be completely oblivious to this link. In relational databases, these links run in the opposite direction: Members of a collection contain a pointer to the collection and can not be members of other collections at the same time without modifications to the schema (Case *a* in Fig. 1.1).

Another issue with ORM is a result of the modeling difficulties. Although problems in data modeling can be worked around with some effort, these workarounds can lead to serious degradation of database performance. This degradation can be illustrated in the given example: The usual procedure to solve the problem described in the previous paragraph is to use a third table to store the associations (Case *b* in Fig. 1.1). Now, the members can easily be associated with multiple collections but looking up these associations involves joins over three tables. It becomes possible for the association table to contain references to deleted members or collections, an inconsistency that can be avoided reliably only by additional constraint-checking upon each update to one of the three tables [ME92, SMA+07].

Lastly, two of the benefits of relational data models, namely non-redundant data through normalized schemas and arbitrary queries containing lots of joins, are not an issue in abstract relational logic but infeasible in common practical scenarios when it comes to processing large amounts of data with reasonable performance. In [Atw08], Jeff Atwood provides experience and references descriptions of several architectures of large web applications that used systematic denormalization and introduced redundancy to be able to process large datasets (*YouTube* [Do07] and *Flickr* [O'R06], among others).

In [New06] Ted Neward gives a comprehensive overview of the problems accompanying object-relational mapping.

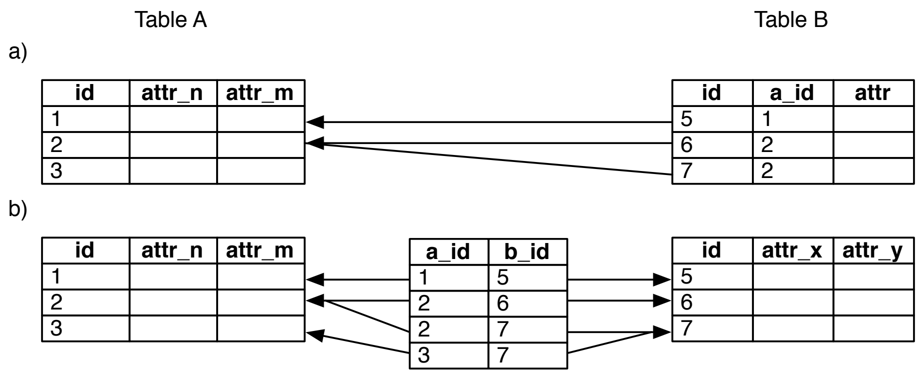

Figure 1.1: *1:n and n:n associations in a relational database. a) A 1:n relation where multiple records in table B are associated with a single record in table A. b) To enable n:n associations from multiple records in B to multiple records in A, an additional table storing the associations is used.*

1.3 Use Cases

The abstract problems outlined in the previous section can become real issues in multiple common use cases. Some exemplary issues will be examined in this section.

1.3.1 Model Validation

Problems: Redundant implementations

Web applications or services accepting data from untrusted sources have to ensure that no invalid data enters the system. In MVC frameworks (*Model-View-Controller*, see Section 2.1.1), this is usually done in the model layer by defining valid value ranges for the models' properties. These validations are checked before a model instance is persisted to the database. In web applications with rich, JavaScript-driven client interfaces, scenarios are imaginable in which validations have to be implemented multiple times.

One validation might exist, written in JavaScript, to provide immediate feedback to the user while he is editing a form in the browser and prevent him from submitting the form until it is valid. A server-side validation has to be implemented in the language of the application server to validate data from the client during request processing. This step is mandatory for almost all applications with untrusted users, otherwise malicious clients could easily pass in invalid data. At last, the database used in the application's persistence layer might apply constraints to ensure integrity at the lowest level.

Three implementations of the same ruleset result in three times the effort required to maintain, develop and test the ruleset. Most importantly, it becomes very easy to introduce bugs into an application if differences in the implementations sneak in.

1.3.2 Data Transformation

Problems: Heterogenous languages, object-relational mapping, glue code required for serialization, and conversion of data

In traditional web application stacks, data passes through many stations on its way from the database to the DOM in the browser, each step requiring a reformatting or re-encoding. Data in a typical application built with MySQL and PHP might traverse the following steps (example taken from [Cap08]):

1. Through the query, binary data is fetched from the **database** via a socket connection.
2. The PHP MySQL module transforms the raw data into a MySQL **rowset**.
3. The rows in the rowset are made available to PHP as **arrays** by the MySQL module.
4. The array is cast into a PHP **object**, processed by the controller and passed to the renderer.
5. The renderer transforms the object to **plaintext, HTML or JSON** (*JavaScript Object Notation*) before it is sent to the client.
6. The client parses the stream coming from the server and constructs the **DOM or other JavaScript objects** from it.

This traversal brings with it multiple problems [Cap08]. Each of these steps involves expensive string processing, most of it on the server-side, leading to bad scalability. The many interfaces involved cause impedance mismatches and offer potential weak spots for injection attacks.

1.3.3 Exposure of Objects Through APIs

Problems: Glue Code

In an MVC based web application framework, exposing an application's models to clients as a service requires writing controller actions that pass HTTP requests to the models and their responses back to the client. In an application that provides strict rules and paradigms for the behavior of models these actions will likely look extremely similar for all models. Without special support for exposing the models, writing similar adapter actions over and over again can be necessary to provide a web service. With web applications becoming more and more similar to web services in structure, this problem applies to applications as well.

In practice, this problem occurred in the *Ruby on Rails* web development community. With the advent of REST support in Rails 1.2 and Rails' strict guidelines for building RESTful controllers, developers began to notice extreme similarities between their controllers. As a response, a popular plugin for Rails was created that enabled controllers to be mapped to models automatically, dynamically generating the common RESTful actions while allowing aspects specific to individual models/controllers to be defined in a concise way [Gol09].

1.3.4 Dynamic Object Properties

Problems: Object-relational mapping, glue code required for serialization, and conversion of data

With client-side user interfaces becoming much more sophisticated than the static HTML forms the web was limited to in its early days, the data structures edited through such interfaces become more complicated as well.

Static forms map nicely to the static set of columns in a relational database table. When a form adjusts itself using JavaScript as the user interacts with it, or when a custom interface creates a dynamic data structure to send to the server, this mapping becomes a hindrance.

For example, an application that includes a contact database assigns multiple telephone numbers to a contact, each consisting of a *number* and a *type* field. Contacts can have any amount of telephone numbers. The intuitive and object-oriented way to store these numbers would be to save them in an array in the contact object. A conventional web application, using a relational database, could create such a contact object on the client in JavaScript. To send it to the server, it would have to be serialized to JSON or XML (in favorable cases) or into a HTTP querystring. Storing the contact to the database on the server would require the telephone number to be separated from the contact and stored into a separate table together with foreign keys to form the association.

The described case has been kept relatively simple and could easily be extended. To illustrate the problems with static schemas even better, the telephone numbers could be extended with additional information, requiring a schema adjustment in the database and an update of all stored numbers. Nested objects of arbitrary depth (and maybe even arbitrary type) become very hard to map to a relational database and to convert between representations.

1.3.5 Storage and Retrieval of Compound Objects

Problems: Object-relational mapping, glue code required for serialization and conversion of data, glue code required for object traversal, and dependency checking.

This example is closely related to the previous one. In the last paragraph of the *Dynamic Object Properties* example, the possibility of arbitrarily nested objects was introduced. But besides the issue of mapping it to a static relational schema, such a structure brings another problem.

It might not always be feasible to store a nested object as a single entity. The parts an object is composed of might not belong to that object exclusively; other objects could reference the parts as well. These parts should be saved in a collection of their own, so they can be addressed independently. If these independent object correspond to models in the application, they will likely have their own validations. Also, the associations between these independent objects have to be tracked explicitly through foreign keys.

For these reasons, saving compound object structures involving associations can be a complicated task. All objects have to be validated, the foreign keys forming the associations have to be properly set, and the objects have to be saved to the database. Complicated structures can introduce dependency problems into these steps: Before it is saved to the database, a new object usually does not have an ID that could be stored in another object as a foreign key. Circular references might prevent a group of objects from validating (if one object's validity depends on the validity of of its associations).

1.4 Goals

The problems presented in Section 1.2 and illustrated in 1.3 lead to four major goals aiming to solve them in the implementation of a new framework (see Table 1.1).

Problem	Goal
Heterogenous languages	A single language
Redundancy in code	Code reuse between server and silent
Glue code	Elimination of glue code
Object-relational mapping	Tight integration of a JSON database

Table 1.1: *The identified problems and the formulated goals. Each goal can be related directly to one motivating problem.*

1.4.1 A Single Language

All parts of the framework should be written in JavaScript. This includes the server-side part, the client-side part and especially the persistence layer which should be implemented using one of the available open-source JSON-based document databases (introduced in Section 2.4.3).

Using only JavaScript is, in part, a precondition for the other goals, just like the heterogeneity of languages is a cause for the other problems.

1.4.2 Code Reuse between Server and Client

It must be possible to specify the classes and models in such a way, that they can be used on both the server and the client. Where different implementations are required, the developer should be able to specify these differences as narrowly as possible.

This requires a highly modular structure in the models and efficient use of JavaScript's open prototypes that allow parts of their implementation to be changed after they have been created.

1.4.3 Elimination of Glue Code

The developer should be able to concentrate on implementing the core structure and functionality of the application's business models and the flow of user interaction without wasting much energy on maintaining the application infrastructure.

This can be achieved by a largely declarative style of programming, by exploiting JavaScript's ability to treat functions as first class types (enabling to flexibly add, remove or apply them to models) and by providing strict standards and conventions that meet the developer's actual requirements. These conventions are effective in two steps. Establishing them in the first place relieves the developer from having to decide on conventions of his own. Working with the provided conventions, he can develop the application without having to spend much time on configuring the framework's parts to interoperate.

1.4.4 Tight Integration of a JSON Database

By integrating the framework with one of the several open-source JSON-based document databases, issues related to object-relational mapping can be avoided. An ideal integration would involve that the business models in the application can be transparently stored and loaded from the database without using a special query language.

1.5 Existing Single-Language Frameworks

The wish to unify web development languages is not new. Two large software products that allow developers to write web applications using a single language are *Microsoft ASP.NET* and the *Google Web Toolkit (GWT).*

Both let the developer write business and interface logic in Java (for GWT) or one of the .NET languages (C#, F#, C++, Visual Basic for ASP). The written models, interfaces and controller logic are then compiled to HTML and JavaScript or executed and interpreted at runtime [Goo09a, Esp08]. This approach adds an additional layer of computation and complexity to a web development that can become too expensive for many applications.

Ruby on Rails, released in 2004, initiated a paradigm shift in web development away from complex frameworks [Dum05]. Three fundamental principles can be observed in Rails' design:

- The use of a flexible, dynamic, interpreted *language* to allow for powerful *development patterns* that provide a very succinct programming style.
- Being *opinionated* about how certain common tasks are performed best. Whereas frameworks before tried to be everything for everyone, provided they were configured correctly, Rails offered *one* way and discouraged (but not prohibited) developers from doing things differently.

- *Embracing web standards.* Rails made the use of REST principles extremely popular and found elegant ways to use the architecture of the world wide web and the related technologies to its advantage.

Web frameworks had previously accumulated complexity that made them unsuitable for many projects. The tremendous success of Rails are a strong hint that the principles behind it's leaner attitude were valid.

Inspired by Rails' success, many frameworks, based on similar principles, have been developed in other languages, but few languages are as flexible as Ruby and allow imitating essential features of Rails. JavaScript is one of the languages whose flexibility is on par with Ruby's [Cro08], but JavaScript was never used outside of web browsers, did not have a standard library, or possibilities to interact with an underlying operating system.

Today, the availability of fast, standalone JavaScript interpreters, upcoming standard libraries for JavaScript and JSON databases opens up the possibility to revisit and implement the single-language approach with respect to these modern paradigms.

1.6 Key Ideas

Using the advantages of JavaScript, its dynamic features and its availability in the browser, to realize the goals formulated in the previous sections leads to the following design ideas.

1.6.1 A Design Based on JavaScript

For the first time, it is possible to develop all aspects of a web application in the same language due to the availability of document oriented databases (see Section 2.4.3) that are using JavaScript as a query language and JSON (*JavaScript Object Notation* [Cro06]) for storing documents. SQL is not used anymore.

The consistent use of JavaScript and the JSON format makes it possible to remove most of the data reformatting steps between database and client. For the implementation of the framework, this effect is exploited as much as possible, leading to a new approach towards the role of the control layer.

In traditional MVC based web application frameworks (see Section 2.1.1), the control layer is central to the application (left side of Fig. 1.2). It reads and writes data from and to the persistence layer, creates model instances from this data, processes these instances, hands them over to the view layer for rendering and finally sends the rendered representation to the browser.

The browser then performs client-side processing, by simple forms or complex interfaces based on JavaScript (example: *Google Maps*[8]) and sends the data back to to the control layer where it is interpreted, transformed into model instances and processed again. In addition to data flow and processing, the control layer also deals with the application's control flow, by rendering the interface and redirecting clients, and aspects like authentication and session management.

Through the use of JavaScript, the number of the control layer's tasks on the server can be greatly reduced by shifting most of them to the client. This primarily affects the application's control flow and rendering which can be managed by the browser through JavaScript and dynamic HTML.

In conventional web applications, user interface aspects are distributed over server and client. The client is responsible primarily for accepting user interface events. These are processed and communicated to the server where they trigger rendering and redirection. By moving these control flow tasks to the client, the traditional control layer is decomposed into two parts.

The layer containing control flow management is labeled *logic layer* to set it apart from the old concept the control layer. The main difference here lies in the location the code is executed at: the client instead of the server. This, however, opens up new possibilities for building control flow management. Controllers can now directly access user interface events. Roundtrips to the server are not necessary anymore. A few web applications like the aforementioned *Google Maps* already work this way.

The responsibilities that remain on the server form the *adapter layer* (see Fig. 1.2). What remains in this layer are access control, session management and code that can not be executed on the client (for security or bandwidth reasons, for example). This way, the adapter layer becomes a very thin interface between the application in the browser and the persistence layer, only passing data through by default. To simplify its interface, communication with the adapter layer is performed through JSON-RPC requests only (see Section 2.5.3). For cases that require data flow manipulation, the adapter layer allows for callbacks containing application-specific code to be executed at certain points during request processing.

Implementing a client/server split as described is a theoretical possibility regardless of the language used on the server, but it becomes practical only with JavaScript. Code for the models can now run on both the client and the server whereas, with other languages, an entire model or certain aspects of a model would have to be implemented twice, in the server-side programming language and JavaScript.

Using the same code on both the server and the client further enables transparent server-side execution of model methods, by replacing the client-

[8]`http://maps.google.com/`

side version of methods with delegates that forward calls to the server and return the server's response.

The flexible nature of JavaScript's properties pattern [Fow97] allows customizing models in a declarative manner and let the framework process these declarations either by interpreting or by using them to generate methods on the models.

Both code sharing and a declarative interface for defining a model's structure and behavior are expected to greatly reduce the amount of code an application developer has to write. By setting standards for the design of model interfaces and encouraging adherence to these standards, uniform interface patterns are established in the framework. These established standards free the developer from having to come up with a design of his own, allowing him to keep focused on the core problems of the application.

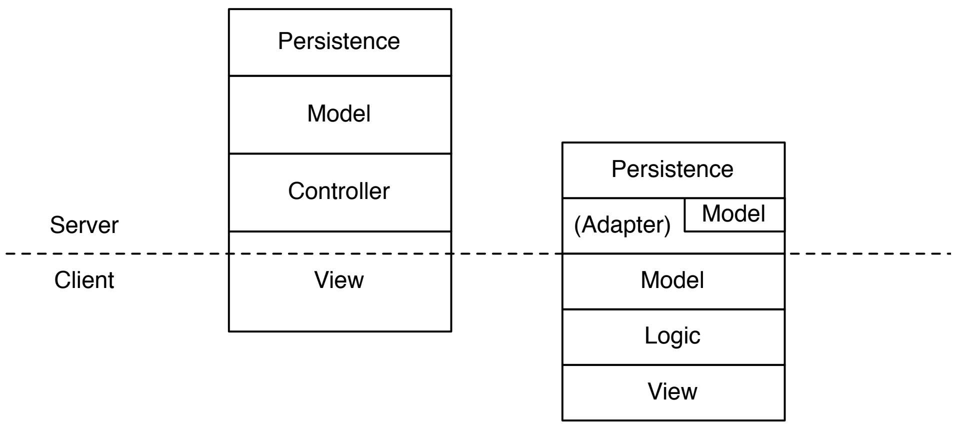

Figure 1.2: *Layering of web frameworks. A traditional MVC stack with an additional persistence aspect is shown on the left. The right hand side illustrates the structure the new framework is based on.*

1.6.2 Key Ideas Put to Work

Chapter 3 describes a design derived from these ideas. To improve understanding of the description, this section will provide a lowlevel overview of the steps and facilities involved in request processing. Although presenting the low-level implementation and the high-level ideas in this reverse order is unusual, it is necessary for putting the ideas presented later in context.

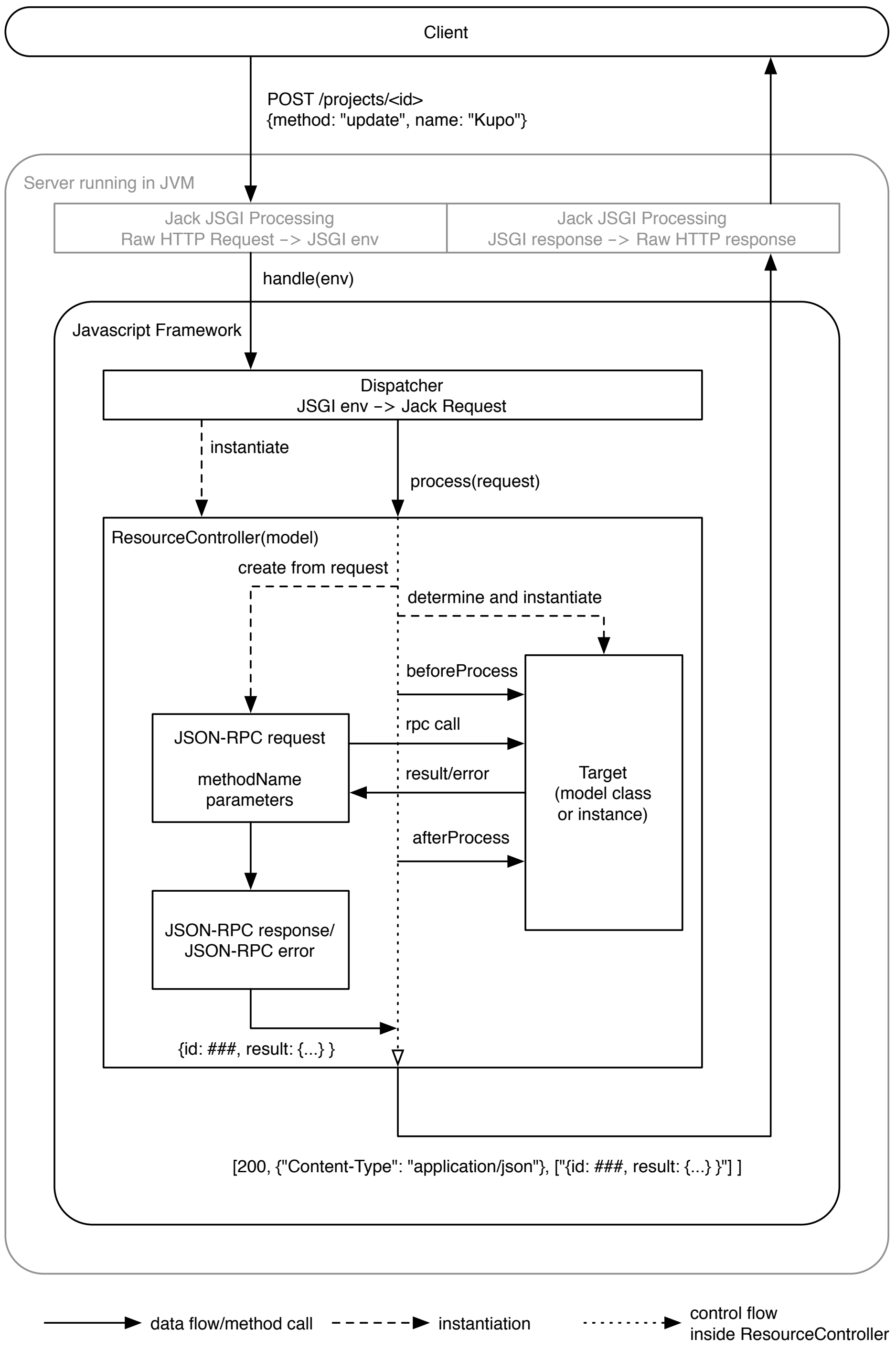

Figure 1.3: *A schema showing how the facilities of the framework process an incoming request from a client.*

Request Processing

Figure 1.3 shows how a request is processed in the framework. The request is received by the webserver hosting the framework. It passes the JSGI interface, is turned into a JSGI environment object (similar in structure and purpose to a CGI environment, see *Jack/JSGI* on page 62) and passed to the framework's dispatcher for handling.

The dispatcher analyzes the request URL by splitting it at the path separators and comparing the first segment to the existing models. If a matching model is found, it becomes the target for the resource controller which implements the adapter layer described on page 13. The dispatcher creates an instance of the resource controller by passing the model to its constructor.

The resource controller interprets the incoming request and creates JSON-RPC request object from it. The resource controller continues request analyzation by examining the request method and the the second segment of the URL. It determines whether the model *class* or a specific *instance* will become the *target* of the request.

If a model contains callbacks for data flow manipulation (*beforeProcess/ afterProcess*), these callbacks are executed before and/or after the actual method call. Next, the requested method is executed on the target, the result wrapped in a JSON-RPC response and sent back to the client.

Model Definition

Listing 1.1 shows an example model definition. The first three lines include external modules (see Section 4.3.1) that are needed for the definition. The `new Model(<name>, <spec>)` constructor creates a new model with a given name and a *spec* object (short for *specialization* or *specification*). The spec object contains certain predefined properties that are interpreted or executed at different stages of the model's life cycle to produce the desired behavior. After the Project model is created, the `completeAll` class method is defined on it.

Listing 1.1 An example model definition

```
var Model        = require('kupo/model');
var Associations = require('kupo/model/associations');
var Task         = require('./task.js');

var Project = new Model('project',{
  instance: {
    methods: {complete : function(){
      this.update('completed', true);
    }}
  },
  callables: ['completeAll'],
  associations: {
    "tasks" : Associations.hasMany(Task)
  }
});

Project.completeAll = function() {
  // ...
}
```

Chapter 2

2 Background

As outlined in Section 1.1, the World Wide Web as in its current form is the result of more than a decade of evolution. In this chapter, the web's history is explored in more detail to provide the context against which the new framework is designed.

2.1 Existing Approaches to Web Frameworks

Since the early days of the Web, server-side frameworks existed to support developers in creating dynamic websites. These frameworks can be separated into several categories by the paradigms that their architectures follow.

2.1.1 Request-/Action-Based MVC

The *Model-View-Controller* pattern is one of the most widespread and most important architectural patterns in application design. Applications adhering to MVC separate their logic into three independent components: The models encapsulate domain logic and data. The controller is responsible for the application's control flow. The views render data [Bur87].

This separation maps nicely to the components of a web application: The models are the business objects that often reside in a database, the views are HTML pages sent to the client, and the controller is the application server that processes requests by initiating actions on domain models and rendering of response pages.

Because of this nice match, many web application frameworks follow this pattern. An especially popular form of MVC web frameworks is the architecture shared by *Ruby on Rails*, *Django*, *Struts*[1] and *Spring*,[2] which can be described as *Full Stack MVC*, encompassing integrated parts for every level of the application that have been designed to work well together. These frameworks are *action-based*. Incoming HTTP requests are mapped to actions in the control layer which process the request, manipulate business models accordingly, hand data over to the view for rendering and send the view's output to the browser.

2.1.2 Component Based

Components are the oldest paradigm in web application frameworks. In 1996 NeXT Software Inc. released the *WebObjects* Framework, the very first web application framework, just after HTTP/1.1 and Java 1.0 had been released [Job95]. Other current examples are *Seaside*,[3] *Java Server Faces*[4] and *Tapestry*.[5]

The idea behind the component based approach is to make the web application development process similar to that of a desktop application. The main difference to the request-based MVC model is that the control flow during request processing is reversed. Pages are composed of components and as a component renders, it fetches business models as needed, processes data, renders subcomponents and returns a rendered HTML representation of itself (see the description of JSF in [Sun08]).

Microsoft's ASP.NET is especially notable amongst the component based frameworks for its event-based control flow, similar to an interactive desktop application.

Controls (the ASP.NET name for components) and script code embedded in a webpage can be executed on the server or on the client. Pages are compiled and exist as code objects in the server. When a request for a page hits the server, the page's Controls are initialized, interact with each other through event handlers and finally render themselves. Interaction and rendering create a *ViewState* that describes the state of the page and is always communicated back and forth between client and server through a hidden form field. User interactions, like submitting a form or clicking a link, fire events which are handled by re-requesting the page from the server. The page is then rendered again, taking the event into account; the ViewState is updated and sent back to the client [Esp08].

[1]`http://struts.apache.org`
[2]`http://www.springsource.org/`
[3]`http://www.seaside.st/`
[4]`http://java.sun.com/javaee/javaserverfaces/`
[5]`http://tapestry.apache.org/`

2.1.3 Free Form

The term *free form* is used in this thesis to refer to web applications that do not follow any of the established approach or do not use any kind of framework at all. Hence, *free form* does not denote an architectural paradigm. It is listed here to take into account that not all web applications are based on frameworks.

This is true especially outside the enterprise domain. Early frameworks were complex and targeted at the enterprise, using *Java Enterprise Edition* Servlet technology as a basis [Sun08]. Web developers who worked outside large companies often preferred leaner software stacks, such as those provided by PHP or Perl, that were easier to develop for and easier to deploy [Dum05].

For those platforms, however, frameworks supporting lightweight development approaches became available only recently. As a result, developers were forced to come up with their own custom solutions to recurring problems.

Sometimes implementation of an application or parts of an application without a framework is the result of a conscious decision. A framework offers a lot of convenience but can carry a significant CPU and/or memory overhead. To avoid this overhead, writing an application without a framework can be suitable when performance is critical or resources are scarce.

2.2 Web-Applications vs. Web-Services

In [BGP00] Baresi et. al. give a definition of the term “web application”:

> E-commerce, web-based booking systems, and online auction systems are only a few examples that demonstrate how WWW sites are evolving from hypermedia information repositories to hypermedia distributed applications, hereafter web applications. They blend navigation and browsing capabilities, common features of hypermedia, with “classical” operations (or transactions), common features of traditional information systems.

Informally, given this definition, any website offering more than simple information retrieval services can be considered a web application. With increasing capabilities of server- and client-side programming environments and growing ubiquity of fast internet access, web applications have become more sophisticated in recent years. This development has shifted the meaning of the term.

It would be unusual today to call an online-shop a web application. The term is used instead for applications that, a few years ago, would not have been considered for implementation on the web, such as business applications like calendars, collaboration tools, social networks and media sharing sites, word processors, spreadsheet-applications and even audio-, image- and video-editing software.

The W3C provides a definition of the term “web service” in [HB04]:

> A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

Removing implementation specific details from this definition (use of SOAP and WSDL as protocols and XML as a data format) results in the definition of a “web service” as *a software system designed to support interoperable machine-to-machine interaction over a network. Other systems interact with the Web service in a manner prescribed by its description using messages, typically conveyed using HTTP in conjunction with other Web-related standards.*

The shift of application logic from the server to the browser and the need for structured development in more and more complex applications led to web applications increasingly adopting characteristics of web services. The popularity of REST in the last two years led to a style of web application development that starts with an API serving potential clients in machine-readable data formats and then implements the actual application with this API. First users of the API are usually the client-side parts of the application that query the server for additional data or execute actions on the remote end as the user interacts with the application interface.

Opening up these APIs to the public increases interoperability between sites and services. In the enterprise software industry this style of application architecture is known as *Service Oriented Architecture* [Erl05]. In web development, a term often used to describe applications composed of services from multiple providers is *MashUp* [ZRN08].

2.3 JavaScript

JavaScript was originally developed by Brendan Eich at Netscape in 1995 and “aimed to provide a ‘glue language’ for the Web designers and part time programmers who were building Web content from components such as images, plugins, and Java applets" [Ham08]. It was released to the public as part of the the *Navigator 2.0* browser and later standardized as *ECMAScript.* The first edition (*ECMA-262*, 1997) was based on the Netscape's JavaScript, Microsoft's *JScript* and other implementations [Ham08]. The current standard is *ECMA-262 3rd Edition* (defined in 1999) and extends the original with "regular expressions, better string handling, new control statements, try/catch exception handling, tighter definition of errors, formatting for numeric output and minor changes in anticipation of forthcoming internationalization facilities and future language growth" [ecm99].

Currently the *ECMAScript 5th edition* (abbreviated as *ECMAScript 5* hereafter) exists as a final draft (an official 4th edition never existed). It "codifies de facto interpretations of the language specification that have become common among browser implementations and adds support for new features that have emerged since the publication of the third edition. Such features include accessor properties, reflective creation and inspection of objects, program control of property attributes, additional array manipulation functions, support for the JSON object encoding format, and a strict mode that provides enhanced error checking and program security." [ecm09]

The names *JavaScript* and *ECMAScript* are often used interchangeably today. Strictly speaking, *JavaScript* only denotes the *Mozilla* implementation of ECMAScript that was extended with several features, some of which are reintroduced into the standard with ECMAScript 5 (such as getters/setters).

Use of the Term in this Thesis

In conformity with the use of these terms in the development community the term *JavaScript* will be used throughout the document as a synonym for *ECMAScript.* The vendors of the engines mentioned in this thesis use the term *JavaScript* themselves to describe their engines [App09, Goo09c, Moz09b], although they do not implement all of the features defined in Mozilla's JavaScript standard (with the exception of Mozilla's own *SpiderMonkey/TraceMonkey* and *Rhino* engines [Moz09a, Moz09b]).

For the purposes of the thesis, the difference between the two is unimportant since the Mozilla JavaScript Extensions do not fundamentally alter the language but merely add convenience features. In places that explicitly discuss the ECMAScript standard, the term *ECMAScript* will be used. When specific features of Mozilla's JavaScript standard are discussed, they will be referred to as *Mozilla JavaScript Extensions*

Language Features

JavaScript features a C-like syntax and common control flow statements (if/else conditionals, for/while loops, switch statements). JavaScript is dynamically typed. It is a prototype-based language using functions as constructors. Functions are supported as first-class data types. They can be assigned to variables and passed around as closures. Functions can be called as methods on objects, as unbound functions or be bound to any object for the execution of a single call. JavaScript has literals for arrays and objects (which can be used as associative arrays). Strings can be evaluated as code at runtime [Cro08].

The language follows very few rules, but its open structure makes it extremely flexible and allows it to build higher level features such as classes on top of the language core.

2.4 Changes in Recent Years

Since about 2004, the World Wide Web underwent great changes in perception, use and implementation, summarized under the vague term “Web 2.0”, made popular by Tim O’Reilly [O’R05]. While validity of attaching a version number to *Web* can be debated, it is undeniable that around that time the use of the web changed and technologies changed with it. This section will concentrate on the technological changes, especially with regard to web development.

Two very important aspects were briefly mentioned before, the introduction of *XMLHttpRequest* and *AJAX* in Section 1.1 and the growing architectural similarities between web applications and web services in Section 2.2. In this Section, AJAX and a few other important developments will be examined more thoroughly.

2.4.1 Web 2.0 and AJAX

The availability of the *XMLHttpRequest* API enabled a new style of user interaction that turned the web page into a canvas to be filled with data fetched in the background, after the page itself had been loaded. Applications felt much more responsive, since the need to reload the entire page for each update was removed. This responsiveness, amplified through the rising availability of high-speed internet access, gave web applications an acceptance boost and finally resulted in the migration of applications to the web that would not have been considered before [Gar05].

AJAX led to a comeback of JavaScript, which was very hard to use successfully before. The problem was not the language in itself, but the implementations of the APIs used to communicate with the browser and the DOM. These were largely incompatible between browsers, making development difficult and frustrating [Koc09]. The language's newly gained attention quickly led to the development of client-side JavaScript frameworks like the ones listed in Section 1.1.

These frameworks not only abstracted away browser incompatibilities, but thanks to JavaScript's open structure, each could provide a different approach at making JavaScript development easier and more pleasant for the developer. This, in turn, further reinforced tendencies towards client-side development. A popular example of a client-side web application with a sophisticated user interface is *Google Maps*.[6]

2.4.2 JavaScript Engines

The increased use of JavaScript in the browser prompted vendors to pay more attention to JavaScript execution performance and led to the development of new JavaScript engines and, in case of Google's *Chrome*, even an entirely new browser dedicated to support complex client-side JavaScript-driven web applications. They all use modern interpreter technology in order to run JavaScript *much* faster and use memory more efficiently than previous engines. It is currently not possible to give an objective comparison of their performance characteristics, as all engines still undergo rapid changes and new speed records are established frequently.

SpiderMonkey is the name of Mozilla's C++ based JavaScript engine used in the popular Firefox web browser. Firefox 3.5 (released June 2009) features an improved version of SpiderMonkey called **TraceMonkey**. TraceMonkey is a bytecode interpreter with optional just-in-time (JIT) compilation capabilities. It recognizes most-executed loops in a JavaScript application using Trace-Trees technology [Eic08b, GF06]. Were the difference is irrelevant, *SpiderMonkey* will be used to label both the former and the current, optimized version of the engine hereafter.

Apple Inc. released version 4 of its *Safari* browser in June 2009, featuring a new JavaScript engine called **Nitro** (known under the code name *Squirrelfish/Squirrelfish Extreme* before). Like TraceMonkey, Nitro is a C++ based bytecode interpreter with several performance enhancing features like bytecode optimizations, a polymorphic inline cache, context threaded JIT compilation, and regular expression JIT compilation [Sta08].

Google's *Chrome* browser, released in 2008, is heavily focused on supporting complex JavaScript applications. Among stability-enhancing mechanisms,

[6]`http://maps.google.com/`

Chrome uses **V8**, a brand-new JavaScript Engine written in C++, optimized for running applications with large codebases. V8 features mechanisms for fast property access, dynamic machine code generation, and efficient garbage collection [Goo09b].

The last of the popular JavaScript engines is Mozilla's **Rhino**. Unlike the other engines, it is comparatively old and not as optimized for speed. Rhino was created in 1997 at Netscape. Initially intended as a JavaScript implementation written in Java for a Java-based browser, it was never used as such but became popular as an independent engine for embedding in other applications. Rhino can either interpret JavaScript files or compile them to Java classes [Moz09b]. It features an extremely easy integration with the Java runtime through its *LiveConnect* interface [Boy09].

Table 2.1 summarizes these informations into an overview. In Section 4.1.3 the engines are further compared with regard to their suitability for implementing a prototype for this thesis' design.

	Vendor, Product	License, Language	Features
TraceMonkey 2009	Mozilla, Firefox	MPL 1.1 / GPLv2, C++	Oldest engine. Bytecode interpreter with JIT compilation. TraceTree Optimizations
Nitro 2009	Apple, Safari 4	GPLv2, C++	Bytecode interpreter with context-threaded JIT compilation. Polymorphic inline cache. Regular Expression JIT compilation
V8 2008	Google, Chrome	BSD, C++	Optimized for large codebases. Fast property access, dynamic machine code generation, efficient garbage collection
Rhino 1997	Mozilla	MPL 1.1 / GPLv2, Java	Interpretive mode and compilation to Java classes. Seamless integration with Java runtime.

Table 2.1: *Overview of three new high-performance JavaScript engines and the older Rhino engine.*

2.4.3 Approaches to Database Systems

For a long time, web applications of any scale and kind have used relational databases like the open-source MySQL or PostgreSQL databases. In recent years, people in the web development community voiced their concerns about the suitability of relational databases for web applications, which often operate on textual or semi-structured data [Ark07].

Even outside the web, experts begin to question the use of RDBMS for every kind of problem. In [SMA$^+$07], Stonebraker argues that the design (and often the code) behind RDBMS stems from use cases and hardware preconditions of the 1970s that have become irrelevant since. He lists several examples where alternative approaches to data storage have greatly outperformed generic relational database systems.

Considering the web, in [Lin04] Ling et. al. write

> Traditional relational databases which assume that data is structured are no longer suitable for the new Web applications because the data on which the Web applications are based lacks structure and may be incomplete. Thus, many of the techniques that were previously used may not be applicable. This less structured data, also known as semi-structured data, is usually represented as a tree of elements, where the children are sub-elements of their parent element.

As described in Section 1.2.4, optimizing systems that are backed by relational databases for performance is usually achieved by sacrificing the conceptual benefits of relational schemas, as far as degenerating databases to key-values stores. Under these circumstances, carrying the overhead of a relational storage and query engine becomes questionable if its benefits are not used.

From these deliberations, the experiences big websites had with scaling databases (see [Atw08]) and from the research on XML databases, multiple vendors and open-source projects started to provide alternatives to relational databases in recent years [Jon09]. Three of these projects considered for use in the new framework are the CouchDB, MongoDB and Persevere databases, described in detail in Section 4.1.1. They offer storage of semi-structured data, accessible via standard protocols and formats (REST and JSON) and are optimized to provide high performance while keeping a lean feature set.

2.5 Protocols for Web Services

For the implementation of web services, several standards exist today. They all resemble traditional remote procedure calling protocols like *CORBA* (*Common Object Request Broker Architecture*, Object Management Group), *DCOM* (*Distributed Component Object Model*, Microsoft) or *Java RMI* (*Java Remote Method Invocation*, Sun) but use HTTP and open web standards and protocols for transport.

2.5.1 SOAP

SOAP (*Simple Object Access Protocol*) is a language-independent web service messaging standard that uses XML-based messages. It supports two types of communications: messaging and RPC.

SOAP messages consist of an `Envelope` element containing a `Header` and a `Body` element. The Envelope defines the message's namespaces. The Header is optional. The mandatory body contains the information exchanged in a message, or name, address and arguments of a method in the case of a RPC.

SOAP is thoroughly standardized and widely used in enterprise software development [YLBM08].

2.5.2 REST

The term REST (*Representational State Transfer*) was created by Roy Fielding in his dissertation *Architectural Styles and the Design of Network-based Software Architectures* to describe a set of design principles and constraints for distributed hypermedia systems [Fie00]. As such, REST is not the description of a specific architecture but a *meta*-architecture that actual architectures can follow. The HTTP protocol, for example, fulfills the criteria of REST, but a lot of architectures built on top of it do not, since they violate one or more of REST's requirements.

The key components in REST are *Resources*, defined as "any information that can be named," identified and referenced through *Resource Identifiers*, *Representations* of resources, consisting of data and metadata —actual bytes—, capturing the current or intended state of a resource, and *Connectors* of different types (client, server, cache, resolver, tunnel).

These components operate together under certain principles: The architecture must be client-server separated, communication between the Connectors should be stateless, and data labeled cacheable should be treated appropriately. Interfaces between connectors should be uniform and the entire system separated into layers to limit its complexity.

REST gained widespread attention through the publication of *RESTful Web Services* [RR07] and the following implementation of REST support in the popular *Ruby on Rails* framework [Han09]. Usage of the term in the web development community is vague and not always congruent with its original meaning. Instead, *RESTful* is often used to describe architectures that feature some of Ruby on Rails' implementation details, like human-readable URLs, usage of HTTP PUT and DELETE, in addition to GET and POST and respect for the `Accept` and `Content-Type` headers.

This very informal specification however has not stopped web services from advertising their APIs as RESTful. Even without a clear definition of what an individual service means by *REST*, a description of the addressable resources, supported methods and a data format for the representation of those resources is usually sufficient to successfully use such an API, because the only requirements for writing clients to REST APIs are support of the HTTP protocol (widely available in form of libraries) and compatibility with the media types used for the resource representations (which are usually standard formats like XML or JSON).

2.5.3 XML-RPC / JSON-RPC

XML-RPC (*Extensible Markup Language Remote Procedure Call*) is a precursor to SOAP, designed for calling methods remotely via HTTP, using XML as an exchange format. The XML-RPC specification is much less extensive than the SOAP standard. XML-RPC merely defines the basic setup of a *methodCall*, the call's properties and a few standardized data types (int, boolean, double, string, dateTime, base64, array, struct) [LDJ01].

JSON-RPC is very similar to XML-RPC. As the name suggests, JSON-RPC uses the JSON format instead of XML. JSON-RPC is not an official standard; since it is defined very simply, and easy to implement, many JSON-RPC drivers exist for all kinds of programming languages [jsone].

2.6 Existing JavaScript Based Frameworks

Several server-side JavaScript frameworks and environments already exist. Most of them use Mozilla's Rhino or SpiderMonkey engines. This section presents the candidates that are most relevant in how their design relates to this thesis, and provides a quick overview of other, less relevant approaches.

2.6.1 Helma

Helma, or *Helma Object Publisher*,[7] is a framework based on the Rhino engine. Its core feature is the close coupling between URLs and model objects.

Helma applications define a hierarchy of objects through a directory structure in the filesystem. Each directory inside the hierarchy maps to a JavaScript prototype inside Helma, with properties defined through JavaScript files placed inside the directory. These prototypes and subsequently all other

[7] `http://www.helma.org/`

objects within Helma are derived from the *HopObject*, which is a base object that can contain child HopObjects, serve URL requests and persist to Helma's internal XML database or to a SQL database [Hel09].

Helma shares the emphasis on business objects with the design developed in this thesis, keeping the controller layer very thin. It does this by mixing model and controller aspects in the HopObjects. Helma does not explicitly support code sharing between client and server. The default data providing mechanism, however, is server-side rendering of HTML templates. The use of JavaScript as a server-side programming language does not seem be motivated by the role of JavaScript on the client but by the prototypal nature of the language. Thusly Helma's JavaScript sources are not designed to be used on the client.

2.6.2 Persevere

Persevere[8] is a very powerful JSON database and JSON-RPC server, developed by the Dojo Foundation for use with the *Dojo Toolkit* client-side JavaScript framework. It extends plain storage mechanisms with the ability to define classes on the server. These classes provide type- and value-constraints to stored objects as well as methods that can be called remotely via JSON-RPC. Objects can be addressed via REST and accessed through AJAX and COMET interfaces supporting JSON-Query and JSON-Path [Doj09b].

Persevere's libraries and models operate entirely on the server-side. Client-side applications interoperating with Persevere are expected to implement their own model layer on top of the data provided by Persevere.

The concept of a document database enhanced with RPC and validation capabilities is very similar to the design in this thesis. Still, there are two important differences. While Persevere claims to use an object-oriented database, this object orientation is only a layer on top of a traditional SQL database. More importantly, Persevere does not support code sharing between server and client, making redundant implementations necessary. Persevere is designed primarily for use with the client-side *Dojo Toolkit*,[9] whereas the architecture in this thesis is intended to be used with any client-side library.

[8] `http://www.persvr.org/`
[9] `http://www.dojotoolkit.org/`

2.6.3 Aptana Jaxer

The idea behind Aptana's *Jaxer*[10] environment is to "Use your Ajax, HTML, JavaScript and DOM skills server-side" [Apt09]. Jaxer processes HTML pages on the server by parsing them through the Gecko engine (the HTML engine used in the Firefox web browser) and by executing inline server-side JavaScript fragments with SpiderMonkey. The server-side scripts can access and manipulate the DOM created by the HTML source. After scripts have processed and possibly modified the DOM, it is converted back to HTML and delivered to the client. JavaScript embedded in a page can be declared to be run at the server, the client or both locations, or to actually run on the server but generate delegate functions on the client that are executed either synchronously or asynchronously via XMLHttpRequest.

Beyond these main features (accessing the DOM server-side, delegating functions), Jaxer offers basic session, logging, network, database and utility capabilities. It does not provide architectural paradigms and examples shown on the project's homepage suggest that structuring an application is left entirely open to the developer. Jaxer does not explicitly support REST or JSON-RPC protocols.

Jaxer shares with this thesis' design the ability to share code between server and client. It requires the developer to work with SQL directly, violating the idea of using a single language and leaving the ORM problem entirely unsolved. The unstructured approach is likely to induce glue code due to missing low-level support libraries.

2.6.4 Others

Many other server-side JavaScript frameworks exist beyond the ones presented above. Of those, only a handful are in active development.[11] They are only described very briefly here, because they aren't as relevant in their design, in regard to this thesis, as the ones presented in detail above.

JavaScript Environments

The term *JavaScript Environments* encompasses a number of projects that aim to use JavaScript as a general purpose scripting language, similar to Perl, Python or Ruby. These projects do not primarily see JavaScript as a tool for server-side web development, but they support using it as such in several ways.

[10] `http://www.aptana.com/jaxer`

[11] Projects examined were assumed to be in active development if they had any announcements or commits during the last 6 months.

Jslibs[12] A standalone JavaScript interpreter and runtime environment that includes multiple modules such as *OpenGL*, *zlib*, a foreign function interface (*FFI*) and a *FastCGI* module. Jslibs uses the SpiderMonkey engine.

V8cgi[13] Similar to jslibs, v8cgi provides a JavaScript runtime environment (based on Google's V8 engine) and several modules for additional functionality. Unlike jslibs, v8cgi is focused on server-side web development. Therefore only modules are included that have relevance to server-side application development, like HTTP, MySQL, Sockets, or JSON. V8cgi can be compiled as a CGI-executable or as an *Apache* module.

GLUEscript[14] GLUEScript is the successor of the *wxJavaScript* project and aims to promote the use of JavaScript as a general purpose language. It is very similar to jslibs, uses the SpiderMonkey engine and can be compiled as an Apache module.

Servlet Adapters

Three projects, **Torino**[15], **Myna**[16] and **ESXX**[17], use Mozilla's Rhino engine to provide an adapter between *Java servlet containers* [Sun08] and JavaScript, making it possible to write servlets in JavaScript. In providing a plain interface between server and interpreter, these projects are similar in their intention to v8cgi.

Since all three use Rhino, they have access to the Java runtime via Rhino's *LiveConnect* feature, and can utilize any Java library. Because of that, the adapters provide only very basic APIs themselves, and encourage developers to use Java libraries.

Full-Stack Frameworks

The following projects provide more than simple adapters between engines and host environment. They are web development frameworks with extensive support for templating, database connectivity and request processing.

[12]`http://code.google.com/p/jslibs/`
[13]`http://code.google.com/p/v8cgi/`
[14]`http://gluescript.sourceforge.net/`
[15]`http://http://torino.sourceforge.net/`
[16]`http://http://www.mynajs.org/`
[17]`http://http://www.esxx.org/`

Phobos[18] Phobos is a multiple-language web development framework developed by Sun Microsystems, and running on the Java platform. It can be deployed to any Java servlet container. Although Phobos theoretically supports any scripting language implementation conforming to the JSR-223 standard [Sun09], it currently ships with JavaScript only. Phobos provides facilities for dispatching, templates, and ORM, which can be combined to an application through controllers and views written in JavaScript.

Axiom Stack[19] Axiom is a fork of Helma and similar in structure. It uses a customized version of the *Apache Lucene* database, or a relational database. The application objects are exposed through a JavaScript layer realized with Rhino. Requests are mapped to objects or templates in a RESTful fashion.

Trimpath Junction[20] Trimpath Junction is a clone of *Ruby on Rails* written in JavaScript, running on top of Helma. Junction uses the Rhino JavaScript engine and the SQLite database. It features support for offline applications through Google Gears[21] and for sharing code between server and client.

Ejscript[22] The *Ejscript Web Framework* provides a complete MVC stack on top of a custom *ECMAScript* engine. It features an integrated SQLite database and an object-relational mapper, data binding AJAX view controls, and uses jQuery for client-side programming. The Ejscript engine was created explicitly for server-side web applications and provides libraries that go beyond the default objects defined in ECMAScript, such as a in-memory web server module.

Frameworks with JavaScript Support

The frameworks listed here are not primarily JavaScript frameworks, but support JavaScript to glue together parts of applications.

Cocoon Flowscripts[23] *Cocoon* is a web development framework written in Java that promotes separation of concerns as its main feature. It claims to achieve this by building web applications from components that can

[18] `https://phobos.dev.java.net/`
[19] `http://www.axiomstack.com/`
[20] `http://code.google.com/p/trimpath/`
[21] `http://gears.google.com/`
[22] `http://www.ejscript.org/`
[23] `http://cocoon.apache.org/`

be combined using “component pipelines”. One feature of Cocoon are its *Flowscripts*, a “JavaScript API to manage control flow based on an extended version of the Mozilla Rhino JavaScript interpreter that supports continuations” [Apa09a].

Apache Sling[24] Apache Sling is a REST based web framework written in Java. It routes requests to Java servlets or scripts in multiple languages including JavaScript.

[24] `http://sling.apache.org/`

Chapter 3
Design

Section 1.2 already outlined the major problems in current web development and described how a server-side framework written in JavaScript opens up new possibilities to solve them.

In this chapter, an implementable design will be derived from these deliberations.

3.1 Key Ideas

This section presents the key ideas from Section 1.6 once more, together with suggestions on how they can be realized using the features that the JavaScript language provides.

3.1.1 Avoiding Data Transformation and Lightweight Creation of Models

In Section 1.3.2, it was demonstrated how data traverses through many stages between the database in the back-end and the DOM in the browser. The problems caused by this traversal were presented as well.

Conversions between most of the described steps become unnecessary if a JavaScript framework is combined with a JSON database. With data always present as JavaScript objects, transformation steps between different representations of data can be kept to a minimum. Only when data is transferred from server to server, a JSON encoding is necessary. Client-side, depending on the approach taken, data could even be inserted directly into the DOM, without an intermediate HTML representation. Additionally, a JSON database removes the need for an object-relational mapping layer between the database and the rest of the application.

Under these conditions, the creation of lightweight model instances can be achieved by creating an empty object, derived from a model's prototype, and storing the instance's actual data in a property of this object.

3.1.2 A Thin Controller and Uniform Models

Reducing necessary code for the application developer to write is one of the major motivations for the development of the new JavaScript framework. This is achieved through some overlapping concepts that are described in this section.

Resource Controller

Based on the *Exposure of Objects Through APIs* scenario in Section 1.3.3, model objects on the application server are expected to share very similar life cycles during request processing that rarely vary. This suggests a controller design based around these similarities. As a result, the following concept emerged:

Rather than manually defining controllers that provide access to models of a specific class, a single uniform *resource controller* is used to process incoming requests. To simplify its interface, the communication between clients and the resource controller is performed through JSON-RPC requests.

The resource controller analyzes the URL to determine the model on which to operate. The basic life cycle of each model class or instance during request handling is very similar, but to accommodate for special requirements of individual models, callbacks can be defined that manipulate control flow or data during request handling (see Figure 3.1).

The name *resource controller* refers to the Rails plugin of the same name [Gol09], and was chosen because it shares the plugin's ideas, taking them a step further. Also, the way classes and instances on the server are addressed through URLs (see Table 4.2 on page 63) fulfills the criteria for *resources* formulated by REST [Fie00], which made the name fitting.

Specifying Model Behavior

In addition to methods and properties defined on them, models contain various information defining their structure and behavior.

Associations declare the kinds of connections a model has to other models. Association declared in a model enable navigation through connected objects.

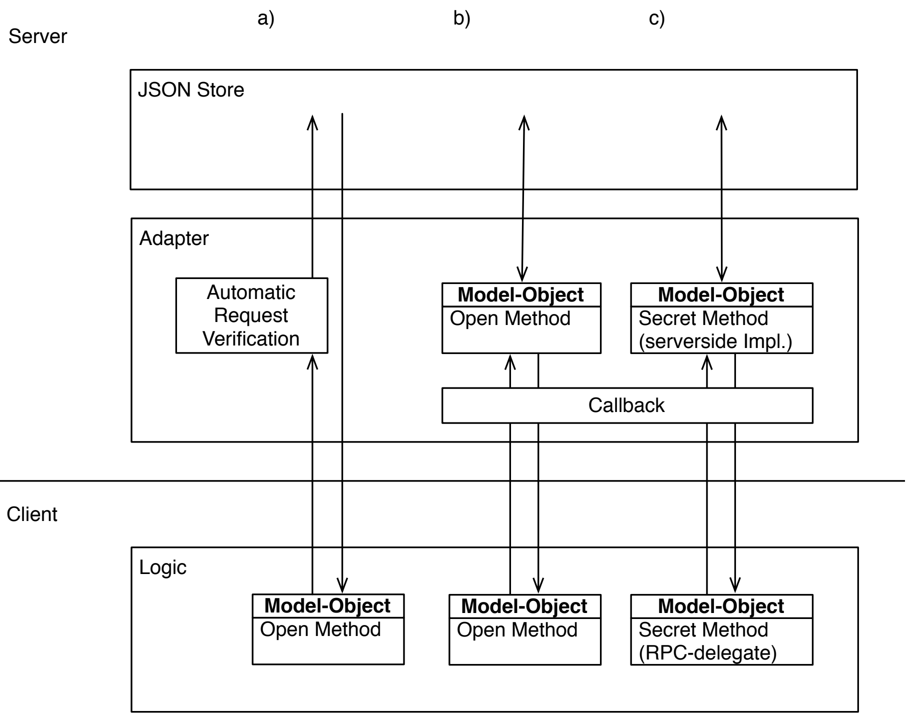

Figure 3.1: *Architecture of the framework. The three columns a, b and c illustrate different request processing variants. a) No application specific code in the adapter layer. Request validation is performed by means of the criteria defined in the model. b) A callback in the adapter layer intercepts the request. c) The model contains methods whose implementation is hidden from the client. Calls to those methods are delegated to the server.*

Validations define those states of a model instance, in which it is allowed to be persisted. These validations can be provided as simple functions, either defined manually or by validation generators provided with the framework. These validations are checked before an instance is saved to the database (variant *a* in Fig. 3.1).

Callbacks hook into a model instance's life cycle to provide a way to automatically alter model data. For example, a declared numeric property could be stripped from leading and trailing whitespace and explicitly converted to a numeric value before entering the validation phase.

Associations, validations and callbacks alter a model's structure and behavior but do not directly influence request handling in the resource controller. To manipulate the life cycle of a model during request processing, callbacks of a second type can be defined on the model. These are executed by the resource controller rather than by the model. Their execution in the resource controller's context grants them access to the request, session, and response objects which allows them to manipulate request processing and perform tasks like access control or HTTP redirection. This is illustrated in variant *b* of Fig. 3.1

Adapter Layer

With models and the resource controller working as described, the control layer can be drastically reduced. For simple cases controllers do not even have to be defined anymore; the resource controller automatically adjusts itself to the needs of a model by inspecting it.

In "traditional" MVC frameworks (see Section 2.1.1), the control layer's primary task is management of the application's control flow. With much JavaScript already running on the client and the controller layer slimmed down by using the resource controller, it makes sense to shift control flow management to the client as well, leaving only basic access control and session management on the server.

This changes the control layer's focus from controlling the application at its core to providing data to the client in the form of model instances, and executing methods on the models as requested by incoming JSON-RPC messages.

To set it apart from the old concept of the control layer and emphasize its new duties, this leaner variant will be called *Adapter Layer*. Its primary purpose is that of an adapter between the database and the application running on the client, and of a receiver of remote procedure calls. The control flow management, now shifted to the client, forms the *Logic Layer* (compare Fig. 1.2 on page 15).

For exceptional occasions, when the resource controller is not sufficient for a task, custom controllers can be defined. This makes it possible to provide static information or data from sources other than the defined models.

Exploiting Nested Structures

Nested structures are often found in web applications, in the form of compound, associated objects or objects of flexible structure (see Sections 1.3.4 and 1.3.5). Since web applications often operate on textual or semi-structured data, strong support for working with these structures can provide great productivity benefits [Ark07].

Support in the adapter layer for processing complex input saves the developer from explicitly traversing and processing the nested structure when saving or validating data. Without this support, automatic processing of write-requests to the resource controller would only be possible in very simple cases with non-nesting models.

3.1.3 Code Sharing, Code Splitting and Delegation

It is beneficial for several reasons to reuse and share code between server and client. If the application's front-end is designed to run in the browser, rich models are useful to easily access validations, associations and model functions. To avoid an impedance mismatch between model representations on server and client and to prevent redundant implementations, the code that defines and implements models on the server should be usable on the client as well.

Simply using *identical* code on client and server is inappropriate for an effective use of code sharing. The design of the sharing mechanism needs to take into account several cases:

- Aspects used *only* on the server
- Aspects used *only* on the client
- Aspects used on *both* server an client

Code for each of these cases should be writable in a concise way, avoiding duplication. This can be achieved by defining the cases in terms of their differences. The developer is able to provide a file containing shared aspects as well as files for location-specific aspects. The framework combines the contents of these files to load a version of the module that features all required aspects for the current location. This is illustrated in Fig. 3.2.

There are several features that benefit from code sharing. *Form validation* in the browser can be achieved easily if the form contents are used to create a model instance. This instance is then validated against the same validation definitions used on the server. *Security features* can be run on both the client and the server. Although the client-side execution should not be used to actually secure the application (this has to be done in the trusted environment of the server), they can provide convenience for the user by indicating a lack of access privileges to certain data or operations *before* he tries to perform a request that is about to fail [Lho06b]. *Persistence* aspects of models benefit greatly from code sharing and splitting. While many persistence features can be shared between server and client, the lower-level operations such as saving or retrieving records need to be implemented differently on server and client.

Remote *delegation* builds upon the code sharing and splitting mechanisms. This feature is used to make methods available to the client without exposing the methods' implementations. There are multiple reasons for hiding an implementation from the client (proprietary algorithms, security related code) or for executing it on the server (bandwidth or latency issues). Delegation can be implemented by writing a method as a server-only aspect of a model, exposing it as JSON-RPC-callable, and writing a client-side delegate that sends an RPC request to the method (variant *c* of Fig. 3.1). For simple cases it is even possible to generate the client-side method automatically.

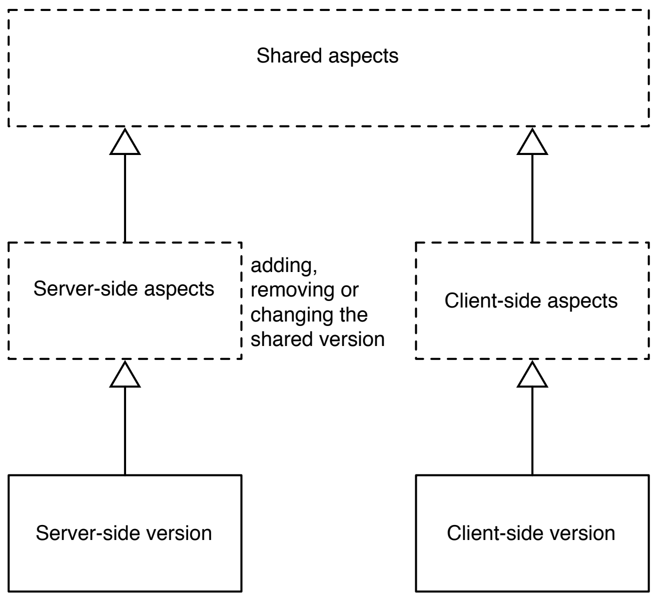

Figure 3.2: *Module inheritance with code sharing and splitting.*

3.2 Object Infrastructure Design

3.2.1 Controllers

Request processing on the server happens in two different types of controllers. Depending on the URL, requests are either routed to a custom controller or to the resource controller. The custom controller provides manually predefined actions to the client, the resource controller works with a model to provide access to that model's methods.

One of the core ideas behind the framework is the use of JSON-RPC (see Section 2.5) in combination with a strict schema for addressing recipients of remote method calls.

Resource Controller

Usually, requests will be handled by a generic *resource controller*. This controller is implemented only once and handles all requests to URLs that correspond to models. This is possible through a series of conventions and interfaces to which the models adhere. In most cases, requests directed at models are handled in very similar ways. To allow for exceptions, models can specify callbacks to be executed at specific stages during request processing.

The basic idea of the resource controller is that RPC receivers are always model instances or model classes. Clients can send JSON-RPC messages to URLs with the structure `/<model>` or `/<model>/<id>` to call methods on either the class or an instance of a model.

To make a remote procedure call, the client sends a POST request containing the name of the method to be called and its parameters. Which of a model's methods are RPC-callable is declared in the model.

Custom Controller

For special cases, custom controllers can be defined. They are addressed through URLs of the form `/<controllername>` and provide their actions as RPC-callable methods to the client. This makes it possible to provide static information or data from arbitrary sources to the client.

Controller Support Structures

Several facilities exist that abstract away some of the details of an incoming HTTP-Request or assist the developer in processing a request.

Any incoming request is first wrapped into a JavaScript object containing all information about the request:

- HTTP Method
- Content-Type
- URL
- Body (if present)
- Any additional headers

This is performed not by the framework itself but by underlying middleware connecting the framework to a HTTP server.

Incoming requests' bodies are parsed and converted to objects representing the remote procedure call that the request contained. Controllers processing the request should not need to differentiate between a JSON-RPC requests made via GET and a querystring or through a POST request containing a JSON object in the body.

A dispatcher selects the appropriate controller to handle the request. This is done by first looking up existing custom controllers and handing over control to the first controller that matches the URL. If no custom controller is found, the dispatcher looks for a model class matching the URL and invokes the generic resource controller, passing it the request object and the model class.

During the processing of the request, several helper functions assist the developer in formulating a correct HTTP response. Headers can be specified and cookies are set or deleted. To support sessions, an additional abstraction layer is provided. Session objects are stored into a data structure that is automatically saved when request processing is done and restored when the next request arrives. Several methods for storing sessions are possible, such as using the filesystem or a database.

After headers and cookies have been set, the return value of an action or procedure call is sent to the client in the response body.

Controllers or models may raise exceptions during request processing. These exceptions are caught, wrapped in a JSON-RPC error object and sent to the client.

3.2.2 Models

The resource controller is implemented only once and not extendable. Models thus provide a broad range of opportunities to specify structure and behavior in a way that allows the resource controller to automatically incorporate special aspects of individual models into the control flow during request processing.

Class Architecture

The object inheritance design for models follows a few principles:

- Minimization of repetition
- Hiding server-only aspects from the client
- Resource controller integration

In order to minimize repetition, the design integrates a concept that mimics classes as known in traditional object-oriented languages like Java or C++. Each model consists of instances, and a class that contains aspects which do not concern specific instances (see Fig. 3.3). At the top of the inheritance chain lies the *Class prototype*. The Class prototype provides functionality for all classes, especially methods that perform persistence tasks. These methods fetch and instantiate models from the database.

Corresponding methods, dealing with persistence for individual model instances, are located in the *Common Instance prototype*. These methods provide the means to save, delete or reload an instance. The Common Instance prototype also contains accessor methods that allow controlled access to the internal data of an instance.

The class for a concrete model inherits from the Class prototype and extends it with

- Information for the finders on how to locate model instances in the database
- A resource name for the model, so the resource controller can find the correct model when handling a request
- Class-level methods
- Validations for instance data
- Resource controller callbacks and RPC availability information about methods

Most importantly, the class contains a property called `instancePrototype` which inherits from the Common Instance prototype. This instance prototype is what the name implies: the prototype for instances of the model. It extends the Common Instance prototype with properties specific to an individual model. These properties include information about associations the model has to other models, callbacks for execution during the instance's life cycle, and other custom methods that should be available on the instance.

A model instance is created when one of the finder methods in the model's class is called or when an association in an instance of any model is accessed. The finder then fetches the model's data from the database, creates the model instance, initializes its state and stores the fetched data in the `.data` property. If associations are defined on the model, association proxies, making the associated objects available, are added to the new instance. These are the only properties in the instance, all other methods are provided by the instance prototype.

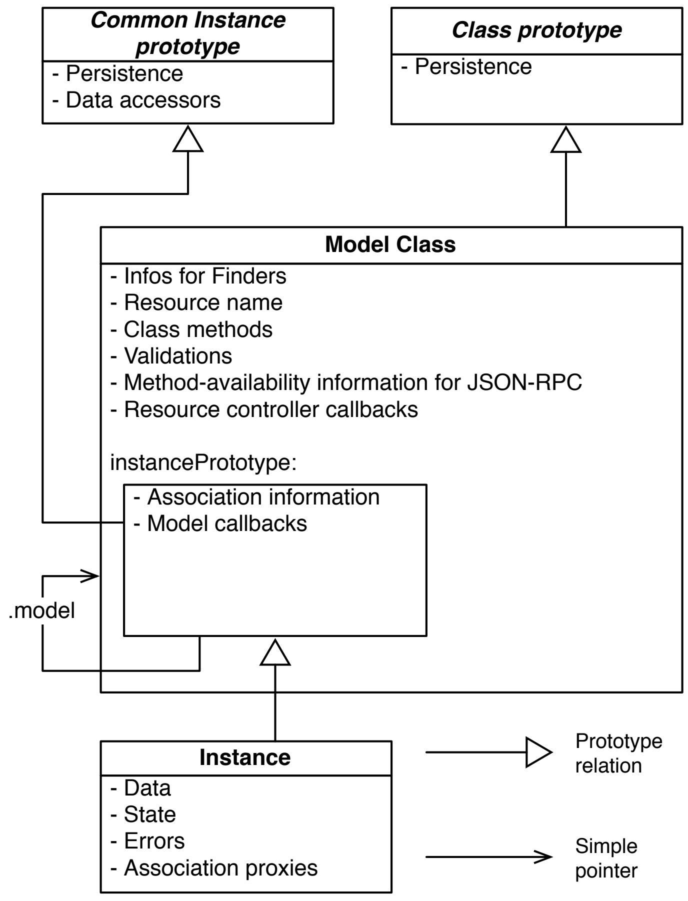

Figure 3.3: *Class layout for models, showing the relations between the Common Instance prototype, the Class prototype, a model's class and a model's instance.*

Resource Controller Integration

As described in Section 3.2.1, the resource controller acts as an interface, exposing the models to clients by making their methods available via JSON-RPC.

This is possible because use of the JSON-RPC protocol between client and server and the uniform persistence mechanisms of the models provide uniform access patterns.

Although this pattern will fit the majority of cases, even the best-designed (non-trivial) application will contain occasions in which exceptions to the standard procedures are needed. To enable the implementation of such exceptions, models define resource controller callbacks. These are functions the resource controller executes at certain points during request processing to perform input filtering, output filtering, redirects or the injection of data into the session.

Since models are not expected to be used outside the framework, and mixing in additional information from a separate file would add complexity without providing benefit, controller callbacks and information about method availability are stored in the models themselves.

Code Splitting

There may exist certain aspects in models that should not be exposed to the client. These aspects might contain security-related code or proprietary algorithms but also different persistence methods. To prevent these methods from being transmitted to the client when the JavaScript code for models is sent, the definition of a model can be split in two. The default definition of a model contains only code that can be shared between server and client. Additional files may optionally remove, replace or add certain methods and properties to a model's class and instance prototype to create a server- or client-specific version of the model (see Fig. 3.2).

Lightweight Model Instances

The model instance objects do not contain methods themselves. These methods are stored entirely in the instance prototype of a particular model. The instance only contains a `.data` property holding its stored data, and other properties keeping track of the instance's state. By storing only the contents of the `.data` property to the database, a quick instantiation and serialization of very lightweight instance objects is possible.

Associations and Nested Data

A model's definition contains information about its associations to other models. In the simplest case such an association consists of the association's name and its kind (indicating how the association is stored in the database, see Section 4.3.3). These two pieces of information can later be used to generate accessor proxies for the association in instances of the model. More complex associations can be implemented simply by defining the accessors manually.

Objects can be serialized and transferred between client and server together with their associations, so that sending an object and its associations can be done in a single request. This is required actually, if the resource controller should be able to handle find and save requests on non-trivial objects.

To enable automatic processing of instances sent to the resource controller together with associated objects, the framework has to support this kind of nested structure in the resource controller and the models. This requires a standard for representing associated objects in JSON-encoded RPC request as well as automated generation of these representations. In the model layer, these combined representations have to be resolved and the corresponding associated model structures have to be recreated. Automated tasks like validation and saving need support for working with these nested structures.

4 Chapter 4 Implementation

This chapter aims to explain the implementation process and rationalize some of the decisions made during development of the prototype for the ideas presented in this thesis. This prototype was called *Kupo* and is available as open-source. Instructions on how to download, install and run Kupo can be found in the appendix.

4.1 Choice of Back-End Software

Even a project as small and clearly defined as this prototype is unreasonable to build completely from scratch. Kupo relies on a number of lower-level tools and frameworks to implement the architecture devised in Chapter 3. Before implementation could start, it had to be decided a) which JSON database to use, b) which JavaScript engine to choose, c) which additional tools and libraries would support development best.

4.1.1 JSON Databases Compared

Although several prototypes and implementations for document-oriented databases existed when the work on Kupo began, only three were popular enough to be considered serious projects, and mature enough to be actually used.

Persevere

The Dojo Foundation's *Persevere* has already been introduced in Section 2.6.2 where it was examined as a framework similar to the one designed in this thesis. Ignoring its ability to provide JSON-RPC services, Persevere can be used (and is advertised) as a document-oriented database using JSON as its exchange format. Persevere is written in Java and closely tied to the client-side *Dojo Toolkit*. Access to the database is provided to clients via the HTTP protocol only. Persevere supports optional schemas, multiple indices and complex queries via the JSONQuery syntax [Doj09a, Doj09b].

CouchDB

CouchDB is a JSON database developed by the *Apache Foundation*. It is written in Erlang and focused on being fault-tolerant, distributed and fast [Apa09b]. CouchDB exposes data through a REST API, much like Persevere, and has no additional features beyond data access and maintenance interfaces. CouchDB is schema-free, the only way of adding structure and performance is the creation of inflexible *views* on a database. These views are created by JavaScript functions performing Map-Reduce operations on the full set of objects and are updated automatically whenever an object is added to or removed from the database [DG08, Apa09c]. The *Map* function of a view creates a collection of $[object\_id, key, values]$ triples, sorted by the key. Fetching this collection (this view) is the only way to retrieve a sorted or limited resultset from CouchDB. This requires any kind of request to the database that should perform reasonably fast to be planned and anticipated in advance.

MongoDB

MongoDB is an open-source, document-oriented database developed and commercially supported by 10gen, with a strong focus on performance and scalability, written in C++ [10g09b]. MongoDB uses an internal database format called *BSON*, an abbreviation for *Binary JSON*. The structure of this format is based on JSON, the encoding is binary. The database engine is aware of the BSON format and can inspect objects in the database for querying and index-building. BSON objects are stored in *collections* which roughly correspond to tables in relational databases. MongoDB does not enforce a schema on the data, the collections are only used for organizational purposes and for maintaining indices [10g09a]. Queries to a collection are performed using a Query-by-Example approach [RG02], supporting matches to nested properties and simple numerical comparisons. MongoDB is accessed via a binary protocol over TCP/IP connections. It does not use HTTP like Persevere and CouchDB. Drivers exist for multiple programming languages.

The use of CouchDB was ruled out early because its view mechanism was too inflexible and expensive to work with under the time constraints of this thesis, without providing any compensative benefit. From the remaining candidates, MongoDB was chosen over Persevere. For the purposes of the

Kupo prototype implementation both were extremely similar, but MongoDB had better driver support and a simpler structure. The critical factor for the decision towards MongoDB was the lack of a complete HTTP client interface in the Narwhal middleware (described in the next section). To use Persevere, writing a wrapper for Java's HTTP stack would have been required in addition to writing the database adapter itself.

Name, Language	Vendor	License	Features
Persevere Java	Dojo Foundation	BSD	Access via HTTP REST API only, optional schemas, multiple indices, complex queries via JSONQuery, JSON-RPC service provider
CouchDB Erlang	Apache Foundation	Apache License 2.0	Access via HTTP REST API only, requires prepared *views* for advanced queries
MongoDB C++	10gen	Affero GPL 3.0	Access through native drivers in many languages, very simple structure, BSON data format, collections, multiple indices, query-by-example

Table 4.1: *Overview of the three JSON databases considered for use in the implementation.*

4.1.2 The ServerJS Group

During the course of the work on this diploma thesis, Kevin Dangoor (Mozilla) initiated a workgroup under the name *ServerJS* that consists of developers from several companies, institutions and open-source projects that operate in web development, like Sitepoint, Aptana, Helma-NG, IBM, Mozilla, Microsoft and Google [Dan09].

The efforts of the workgroup are pursued mainly through a public mailing list,[1] a wiki hosted at Mozilla[2] and an IRC channel.[3]

The aim of the ServerJS group is the definition of a standard library and coding standards for JavaScript implementations outside the browser, much like the existing standards in other languages. The ultimate goal is to create an adapter layer that allows to use conforming programs on all available

[1] `http://groups.google.com/group/serverjs/`
[2] `http://wiki.mozilla.org/ServerJS`
[3] `irc://irc.freenote.net/serverjs`,
Logs are available at `http://log.serverjs.org/mochabot/`

engines and platforms, and a package management schema that enables the distribution and reuse of libraries written in JavaScript. Standardization efforts include:

1. Modules
2. Filesystem API
3. Binary Data Objects (byte arrays and/or strings)
4. Encodings and character sets
5. System Interface (stdin, stdout, stderr)
6. C unified API

Jack/Narwhal

The closely related open-source software packages *Jack*[4] and *Narwhal*[5] already implement parts of the standards of the ServerJS group. Jack originally was an adaptation of the *Rack*[6] interface, a protocol similar (in purpose) to CGI, written in Ruby [jac09]. The Narwhal project was later isolated from Jack to separate Jack's pure web-application related functionality from other parts of the library that provided more general functionality and became *Narwhal* [nar09].

At the time development of the framework began (April 2009), Jack/Narwhal was released with adapters for Rhino and V8cgi (see Sections 2.4.2 and 2.6.4).

Because of the large number of influential people and a conceivable interspersion of the ideas developed inside the *ServerJS* group, and the good applicability of Jack and Narwhal as a middleware, it was decided to incorporate the ServerJS ideas into the framework by using Jack and Narwhal as a basis for the implementation.

4.1.3 JavaScript Engines Compared

For the implementation of an application server, the available JavaScript engines presented in Section 2.4.2 had to be compared. Since the goal of implementing the prototype was a proof of concept rather than production ready software, performance was not the primary concern in a decision for one of the engines.

[4]`http://jackjs.org/`
[5]`http://narwhaljs.org/`
[6]`http://rack.rubyforge.org/`

Considered aspects were:

1. Ease of integration with third party libraries, especially the integration with MongoDB
2. Ease of building the engine / ease of deployment
3. Developer community / Maturity of the codebase
4. Supported JavaScript features
5. License

Integration of independent packages and complicated build setups can require a lot of time and effort without actually contributing anything relevant to a software project. Avoiding this was the primary goal in selecting an engine.

Even when a lot of build issues are avoided, problems with integration of other packages are to be expected. A strong, open developer community is usually the quickest way to overcome such hurdles.

Four engines were compared through these criteria.

Nitro

Nitro (formerly *Squirrelfish Extreme*) is the bytecode-based, JIT-compiling JavaScript Engine in *Webkit*, an open-source browser framework used by Apple's Safari Browser on Mac OS X and Windows, written in C++ [Sta08]. Webkit was originally forked from KHTML, the engine behind the *Konqueror* web browser of the KDE desktop environment.

Although Nitro is open-source and licensed under GPLv2, it is not used much beyond Webkit based browsers and the development community consists primarily of Apple engineers and KHTML developers. Its development is closely focused on the Mac and Windows versions of Safari. Documentation is relatively scarce.

TraceMonkey

Like Squirrelfish, Mozilla's *TraceMonkey* is a bytecode based interpreter/JIT compiler. It is the engine in the popular Firefox browser. TraceMonkey is a name used to differentiate recent versions of the *SpiderMonkey* engine from former ones that do not use Trace-Tree-Optimization [GF06], the most important optimization technique in TraceMonkey [Eic08b].

SpiderMonkey was the very first JavaScript engine, originally developed by Brendan Eich at Netscape [Ham08]. It was originally written in C and has been ported to C++, albeit keeping the procedural style of the original code, not using the object oriented features provided by C++. It is licensed under the Mozilla Public License (MPL) 1.1 / GPLv2.

Because of its age, SpiderMonkey has a large developer community and is used in numerous software packages for many different purposes. Its age, however, comes with severe drawbacks. The sourcecode itself is barely documented, makes extensive use of preprocessor macros and requires a complicated *autotools*[7] setup to build. SpiderMonkey's main advantage is its support for all of Mozilla's JavaScript Extensions to the ECMA standard.

Rhino

Like SpiderMonkey, *Rhino* is developed and maintained at Mozilla and supports all of Mozilla's JavaScript extensions. It is written in Java, able to interpret or compile JavaScript code and published under MPL 1.1 / GPLv2.

One of Rhino's most attractive features is the extremely easy integration of existing Java libraries. These can be addressed and used "on-the-fly" from JavaScript without the need to manually write adapters.

Being almost as old as SpiderMonkey, Rhino has a large developer community as well but, being written in Java means its codebase is much cleaner (due to the rigorous syntax and package layout rules of Java). It uses the *Ant* build system but thanks to its on-the-fly integration with Java libraries, it is possible to integrate external functionality into precompiled Rhino packages at runtime.

V8

V8 is the youngest of the listed engines and was developed by Google for use in the *Chrome* browser. It is written in object-oriented C++ that makes only light use of the preprocessor and heavy use of templates. V8 (and many related projects) use the more modern *SCons* build system[8] instead of autotools.

The code is well documented and released under the BSD License. Despite its young age, the code quality has attracted many external developers to V8, resulting in projects that made it especially interesting for the work on

[7] *Autotools* is a common name for the classical Unix suite of build tools, consisting of `make`, `configure`, `autoconf`, `automake` and `libtool`

[8] `http://www.scons.org/`

this thesis as many of them revolve around server-side web development, integrating V8 with web servers via CGI/FastCGI or as apache modules and providing low-level bindings for file access, I/O, MySQL, gd and other functionality (see Section 2.6.4).

Considerations

It was relatively clear from the beginning that SpiderMonkey and Squirrelfish were unlikely candidates for engines to be used as a basis of the framework. Squirrelfish was a little too exotic with its strong emphasis on supporting the Safari browser and the tight community around it. SpiderMonkey had a larger community and was already successful in several other open-source projects, but its codebase and built system posed too much of a risk of wasting time on integration issues during early development.

This left Rhino and V8 as the remaining options. Both were supported by Jack/Narwhal. V8 has a performance advantage over Rhino and was tried first. After some moderate success in building a dispatcher and a simple controller, Rhino was given a chance too and finally used for the remaining development because of two major advantages over V8. First, Narwhal's support of Rhino was much more mature than the V8 support and second, Rhino is much easier to deploy and integrate with other software. This was most important in regard to the MongoDB integration. Being able to write the integration of MongoDB's Java driver in JavaScript was expected to save a lot of time over writing bindings in C++ for V8.

Reliance on the standards developed in the ServerJS group however will make the code portable to other engines as these standards and their implementations stabilize.

4.2 Client-Server Scenarios

This section presents several scenarios and examples for communication between client and server to show how the framework actually handles commonly occurring, basic tasks, to serve as orientation the the following sections describing the framework's components, and to illustrate the workings of the implemented API.

4.2.1 Server-side

The processes in this section demonstrate common tasks and describe how requests are handled from the server perspective. A project management scenario with projects containing tasks, which are in turn assigned to users, serves as an example.

Simple Retrieval of a Collection

This scenario describes what happens if a collection of all instances of a model is requested. Fine details of how the request is processed are only presented here and left out in the following cases.

1. On the client, the `all` method is called on the Project model class. This results in a JSON-RPC request to `/projects`, calling `all` on the server-side.
2. On the server, the *Jack* middleware calls the *Dispatcher*'s `handle` method, passing the request environment as an argument.
3. The dispatcher analyzes the URL and looks for a controller called `ProjectController`.
4. The dispatcher does not find a `ProjectController` and looks for a model named `project`.
5. A new instance of the `ResourceController` is created by handing the `project` model over to its constructor.
6. The `handle` method of the controller is called, passing a Jack `Request` object as an argument.
7. The `handle` method is defined in `Controller`, the common prototype of `CustomController` and `ResourceController`. Inside, request processing is prepared (by initializing the request, cookies and sessions) and the `process` method of the current controller is called.
8. The `process` method of the resource controller recognizes the request as a POST request. The absence of a second part in the URL indicates that the client is sending a remote procedure call to the `Project` class which is set as the resource controllers `target`.
9. The `process` method constructs s JSON-RPC object from the contents of the request with `all` as the method name and no parameters.
10. BeforeProcess callbacks defined on the `Project` model are executed in the context of the resource controller before the `all` method is called.

11. The `all` method retrieves a collection of data objects for all available projects from the database. For each data object in this collection, a new Project instance is created, its state is initialized to "clean" and the data object is installed in the instance's `data` property. The resulting collection of initialized Project instances is returned to the controller.

12. This collection is wrapped in a JSON-RPC response object, returned back through the call stack and sent to the client.

Simple Retrieval of a Single Model Instance

This scenario describes what happens if an instance of a model is requested for the first time.

Steps 1-9 are the same as in the previous example, with one exception: The JSON-RPC request sent to the server contains `find` as the method name and `"4a3a87e0b98a5e4d343ae72c"` as the method's parameter.

10. The `process` method constructs s JSON-RPC object from the contents of the request with `find` as the method name and the parameter `"4a3a87e0b98a5e4d343ae72c"`.

11. BeforeProcess callbacks defined on the `Project` model are executed in the context of the resource controller before the `find` method is called.

12. The `find` method retrieves a data object for the project with the given ID from the database. A new Project instance is created, its state is initialized to "clean" and the data object is installed in the instance's `data` property.

13. The instance is wrapped in a response object, returned back through the call stack and sent to the client.

Simple Authorization

A project should only be accessible if the current user is a member of the project.

1. The method `find("4a3a87e0b98a5e4d343ae72c")` is called remotely on the project class at `/project`

2. Handling proceeds as in the previous use case until the beforeProcess callbacks are executed in the resource controller

3. One of the callbacks will inspect the request's headers and abort further processing of the request if the authorization requirements (project membership of the requesting user, identified through HTTP-Basic authentication headers or session variables) are not met.

4. Aborting request handling is achieved either by throwing an Exception, which is caught by the Dispatcher, wrapped into a JSON-RPC error response and sent to the client, or by setting the `result` property of the resource controller to a non-null value. In that case, the called function will not execute, but the result produced in the beforeProcess callback will be returned to the client.

Indirect Authorization

A User wants to access a task. Authorization is implemented via the project-task association. If the user is a member of the project he should be granted access, otherwise the request should be rejected.

This is very similar to the previous use case, only the means of verifying the authorization requirements have to be formulated differently.

Instead of checking credentials against the task model, the association to the project has to be followed first, and the credentials checked against the project.

Custom Controller Actions

This scenario illustrates the process of handling a request with a custom controller.

It is assumed the client wants to authenticate to the application by sending his credentials to a login controller that generates a session for him.

1. The client requests the `/sessions/login` URL.

2. The *Jack* middleware calls the *Dispatcher*'s `handle` method, passing the request environment as an argument.

3. The dispatcher analyzes the URL and looks for a controller called `SessionsController`.

4. The dispatcher finds the `SessionsController`, creates an instance and calls the `handle` method on the instance, passing the request as an argument.

5. The `handle` method is defined in `Controller`, the common prototype of `CustomController` and `ResourceController`. Inside, request processing is prepared (by initializing the request, cookies and sessions) and the `process` method of the current controller is called.

6. The `process` method of the `SessionsController` analyzes the second part of the URL, looks for a `login` method in itself and calls it, passing the request parameters as arguments.

7. The result of the method call is wrapped in a response object, returned back through the call stack and sent to the client.

4.2.2 Client-Side

The framework does not specify or implement a client-side part with the exception of a small bootstrapping procedure and the use of the models on the client. Implementation of the logic layer is left to the developer who can chose whichever client-side frameworks he deems suitable.

The scenarios in this section provide some guidance to illustrate how the few client-side parts of the framework communicate with the server-side API.

Client-side Startup

1. The browser requests `/index.html`. Since this URL does not correspond to a controller or model on the server, the dispatcher appends it to the application's `public` directory, locates the `index.html` file and returns it to the client.

2. The `index.html` page includes `/clientjs/base.js`, a bootstrapping script that creates a `system` object for use in the the Narwhal SecurableModules mechanism (see *SecurableModules* on page 61). It contains functions to load JavaScript modules via XMLHttpRequest.

3. `base.js` bootstraps a minimal SecurableModules environment to load `sandbox` module from the server containing the full SecurableModules mechanism.

4. When the SecurableModules mechanism is in place, it can be used to load models and library modules from the server.

Client-side Collection Retrieval

After bootstrapping has finished, models can be loaded. Assuming the client wants to display all instances of the Project model, the following steps are executed:

1. The client brings the Project model into scope by requiring the model: `var Project = require('model/project').Project`.
2. The client calls the `all` method on the Project.
3. The client-side implementation of `all` creates a JSON-RPC request with the `all` method and sends it to the `/project` URL on the server.
4. The response from the server consists of plain JavaScript objects. From these, actual Project instances are reconstructed and returned back to the client.

Simple Retrieval of an Associated Collection

The client wants to retrieve a collection belonging to another model instance, in this case a list of tasks.

1. The project is assumed to already exist on the client.
2. The client calls `tasks.all()` on the project to retrieve a list of the projects tasks.
3. The `tasks` association proxy converts the `all()` call to a simple `find` call on the `Task` class, forming the parameters to `find` so that `find` only returns tasks associated to the project.
4. Since the association has not been fetched yet, the project sends an RPC request to the server at `/tasks` calling `find` with the appropriate parameters and returns the response.

Simple Retrieval of an Element from an Associated Collection

The client wants to retrieve an element from a collection belonging to another model instance, in this case a task belonging to a project.

1. The project is assumed to already exist on the client.
2. The client calls `tasks.find("df023573707f404a6cf45500")` on the project to retrieve a list of the projects tasks.

3. The `tasks` association proxy converts the `find` call to a simple `find` call on the `Task` class, extending the parameters to `find` so that it only returns a task that is both associated to the project *and* has an ID of `df023573707f404a6cf45500`.

4. Since the association has not been fetched yet, the project sends an RPC request to the server at `/tasks` calling `find` with the appropriate parameters and returns the response.

RPC with a Hidden Model Function

A bank account is requested to transfer money to another account. The details of the transaction processing should not be visible to the client. The transfer is therefore realized through RPC.

1. The client-side bank account instance receives the `transmit` message with a target account and an amount.

2. The client-side `transmit` method is either a hard-coded RPC delegate or generated automatically from a declaration.

3. The `transmit` method builds an RPC request and sends it to the server.

4. The addressed resource is `/account/123`, which is resolved by the resource controller and secured through a before-filter as described in the other use cases.

5. The RPC object describes the message and its arguments. It is executed as a method call on the account object corresponding to the addressed resource.

6. The response is sent back to the client.

7. The account instance on the client returns the result in the RPC response to its caller.

4.2.3 Unimplemented Scenarios

The scenarios presented in this section have been targeted during the design but not been implemented. Their description can serve as guidance for how to implement them.

Form Generation

Rules are required that allow for unambiguous mapping of form fields to elements of an object tree, so forms can be designed manually. At the same time automated form generators should provide a quick way to generate simple forms for object trees by following those rules.

Creation of an compound Object with Associations in a Single Form

The form elements have to be named in a way that allows the object and its associations to unambiguously be created (see previous section).

1. The client serializes the created object into a JSON-RPC message and sends it to the server.
2. The controller receives the input object.
3. The Project class' constructor receives the project-part of the attribute tree.
4. Subtrees corresponding to fixed attributes are stored as such.
5. Subtrees corresponding to associations are handed to a constructor for the association's class recursively and then attached to the top level model.
6. The tree of model instances is recursively saved.

Model Validation for Server and Client/Forms

There are two possibilities to implement client-side form validation. A model's validation could be mapped to form fields an executed on the form or the logic layer could create a model instance from the form data, validate it and map the errors back to the form.

4.3 Implementation of the Server-side

Processing a request in the framework follows the schema developed in Chapter 3:

Requests are handled by the Dispatcher's `handle` method which uses the `Fetcher` to load a custom controller or model identified by the URL. If a custom controller is found, an action, defined by the second part of the

URL, is executed. If no custom controller is found, a model, matching the URL, is used to create an instance of the resource controller. Callbacks defined in the models can be used to influence request processing in the scope of the resource controller or the model's life cycle.

All these steps will be described in detail in the following sections. For completeness, the steps an incoming request takes through the underlying platform and middleware (Rhino, the *Simple* HTTP server, Narwhal and Jack) before being passed to the framework are also explained.

4.3.1 The Underlying Platform: Rhino, Narwhal and Jack

The application server is started by executing Narwhal (see Section 4.1.2) at the prompt within the Kupo application directory (see Listing A.1 on page 92). The Narwhal startup binary launches the Rhino interpreter in a Java virtual machine and instructs Rhino to start the Narwhal bootstrapping process, passing it the remaining commandline arguments. Narwhal initializes a few basic modules like `filesystem` (containing filesystem functions) and `sandbox` (containing the *SecurableModules* system described in the next section) before starting *Jack*, the actual HTTP interface handling incoming requests.

SecurableModules

The Narwhal environment implements the *SecurableModules* mechanism specified by the ServerJS working group. It was originally designed in [Awa, AK09a, AK09b] and later modified by the working group [Ser09].

To load a module, the developer calls the `require` function, passing it the module identifier. The result of this call can be any data structure that is generated by executing the module, usually a single object (like a constructor function) or a collection of objects (a library of functions).

Modules are JavaScript files that can contain any code, but only the value assigned to the (predefined) `exports` variable is returned by the `require` call. Require operates in two steps. First the passed identifier is resolved to an actual JavaScript source, usually by prefixing the identifier with all known library paths and extending it by "`.js`". The first JavaScript file found is loaded as a string and wrapped in a `function(require, exports, module, system, system.print){<module text>}` (called the *factory* for that module) which is then executed. Passed into the function are the `require` function to allow subsequent module loading, `exports`, storing exported objects, `module`, containing module identification information, and `system`, a collection of platform specific objects and methods.

This mechanism has several benefits:

- It abstracts away the physical location of files. As long as the module identifiers are formed consistently, they can have any format. Identified modules can be loaded from the filesystem, a network connection or a database
- The created factory functions can easily be cached. The module objects created by the factory functions can be cached as well.
- Executing the code inside the factory function can be implemented in a way that prevents accidental pollution of the global namespace.
- The *SecurableModules* implementation in Narwhal additionally supports sandboxes. These are isolated object spaces which contain instances of module objects that are not shared (see [AK09b]).
- Through upcoming features in either single JavaScript engines or the ECMAScript standard, running modules in a sandbox can prevent not only accidental but even deliberate modification of the global object through restricting or manipulating the scope chain and capabilities of the factory function. Hints on how this can be achieved today and in future versions of ECMAScript are given in [AK09a].

After the boot-up and initialization of basic modules (`filesystem`, `sandbox`), Narwhal examines the directory passed through the commandline (the Kupo application directory), and recognizes the directory as a package through the information in `package.json`. It then executes `main.js` which continues the boot process by launching Jack.

Jack/JSGI

Jack provides an interface between a HTTP Server and an application written in JavaScript, similar to *CGI*, *Rack* (Ruby) or *WSGI* (Python). Jack consists of the *JSGI* protocol (*JavaScript Standard Gateway Interface*) defining the interface and libraries implementing this protocol [jac09]. Jack loads the `jackconfig.js` module which exports the application to be run. The application takes the form of a function adhering to the JSGI protocol. JSGI specifies how application functions should interpret their parameters and what kind of object they should return.

Upon launch Jack instantiates a HTTP server. In the default case, *Simple*,[9] a lightweight webserver written in Java is used. A JavaScript handler function is installed as the request handler for Simple. When a request comes in, Jack converts Simple's request description object into a generic format

[9] `http://www.simpleframework.org/`

and passes it to the application (compare *Controller Support Structures* on page 42). In Kupo's case, the `Dispatcher.handle` method acts as the application.

4.3.2 Dispatcher

The dispatcher, which analyzes incoming requests, is kept simple. The framework does not allow for custom URLs, route recognition is hardcoded and follows very strict rules. Inside the dispatcher, only the first part of the URL (up to the first forward-slash `/`), is examined and a controller or model with a matching name is loaded (compare *Controller Support Structures* on page 42).

The next part of the URL, is examined in the controller handling the request, but is treated just as strictly. In a custom controller, this part identifies the action to execute when the HTTP GET method is used. The schema used by the resource controller to analyze the request is shown in Table 4.2. Strict rules reduce confusion and prevents the developer from having to decide on URL schemas for his application.

Method	URL	Target	Parse JSON-RPC from
`GET`	`/model/<method>`	class	path and querystring
`GET`	`/model/<id>/<method>`	instance	path and querystring
`POST`	`/model`	class	request body
`POST`	`/model/<id>`	instance	request body

Table 4.2: *Request schemas for the resource controller. JSON-RPC requests can be made either by POST or by GET. The resource controller internally creates a JSON-RPC object by parsing either the last segment of the requested path and the querystring for method and parameters, or by deserializing and examining the POST body.*

JavaScript and Static File Delivery

Two exceptions from the standard route resolution are implemented. GET requests for paths prefixed by `/js/` are resolved through the module loader. The file content of a found module is returned to the client. This feature is used by the `require` function on the client to fetch modules via XMLHttpRequest.

If the request URL matches neither a custom controller, nor a model, it is appended to the `public` path of the app (see Listing A.1 on page 92). If the resulting path resolves to a file, that file is delivered to the client.

4.3.3 Models

Defining models is extremely concise. In the simplest case, a model can be defined in just one line of code through invocation of the `Model(name, spec)` constructor with an empty `spec` object. This makes persistence available to its class and instances. Models can be extended with validations or other information, by adding properties to the `spec` object. Simple validations are declared in just a single line. For more complex cases, the developer writes functions to perform the validation. This implements the ideas described in *Specifying Model Behavior* on page 36 An example model declaration was given in Chapter 1, Listing 1.1 on page 18.

During development it was decided to keep internal state of the models accessible. Prohibiting access to this state would have required complicated closure constructs that had obscured the code and made custom extensions to the models very complicated. Such measures are inadequate for a lightweight development approach. Instead the developer is trusted only to use the provided methods to actually read and change internal state.

By providing respective helpers, the framework enables a declarative definition of the models' behavior. Although all validations and callbacks are functions, the developer does not have to write them manually, instead they are generated. To validate the presence of a property in an instance, the return value of `Validations.validatesPresenceOf(<propertyname>)` is added to the `validations` array in the model's `spec` object, for example. This return value is a function that performs the validation.

Associations

Four types of associations were implemented in Kupo, based on association types known from other frameworks and entity relationship modeling [Che76, THB$^+$06]. They are shown in Fig. 4.1. Their definitions are:

1. **belongsTo**
   A declaration that *A belongs to B* means that there exists a $1:1$ or $n:1$ association with the foreign key (key of an instance of B) stored in one or many instances of A.

   The B instance belonging to an A instance is found by using the foreign key in A to retrieve it directly.

2. **hasOne**
   A declaration that *A has one B* means that there exists a $1:1$ association with the foreign key (key of an instance of A) stored in one instance of B. This corresponds to the $1:1$ case of a *belongsTo* association.

The B instance associated with an A instance is found by searching the database for the instance of B with a foreign key equal to A's key.

3. **hasMany**
   A declaration that *A has many B* means that there exists a $1 : n$ association with the foreign key (key of an instance of A) stored in many instances of B. This corresponds to the $n : 1$ case of a *belongsTo* association.

   The B instances associated with an A instance are found by searching the database for any instances of B with a foreign key equal to A's key.

4. **belongsToMany**
   A declaration that *A belongs to many B* means that there exists a $1 : n$ association with the foreign keys (key of instances of B) stored in a list in one or many instances of A.

   The B instances belonging to an A instance are found by using the foreign keys in A to retrieve them directly.

Associations are declared in the model's `spec` object as shown with a *hasMany* association in Listing 4.1. Declarations of the other association types follow the example.

`Associations.hasMany` (and the other association declarators) actually generate a pair of functions that is stored under the given name in the `associations` property of the `spec` object. The first one is the `registerCallbacks(model, assocName)` function. It is executed in the `Model` constructor and extends the model's callback chain with `beforeSave`/`afterSave` callbacks related to the association. The second function, `installProxy(instance, assocName)`, is executed whenever an instance of a model is created, to instantiate and install a proxy object for this association on the model instance. In the example in Listing 4.1 `installProxy` would be called in the `Project.makeInstance` method with the new instance and `"tasks"` as parameters. This late installation ensures that all state inside the proxy stays local to the model instance.

Listing 4.1 Declaration of an association

```
var Model        = require('kupo/model').Model;
var Task         = require('./task').Task;
var Associations = require('kupo/associations').Associations;

exports.Project = new Model('project',{
  associations : {
    'tasks' : Associations.hasMany(Task)
  }
})
```

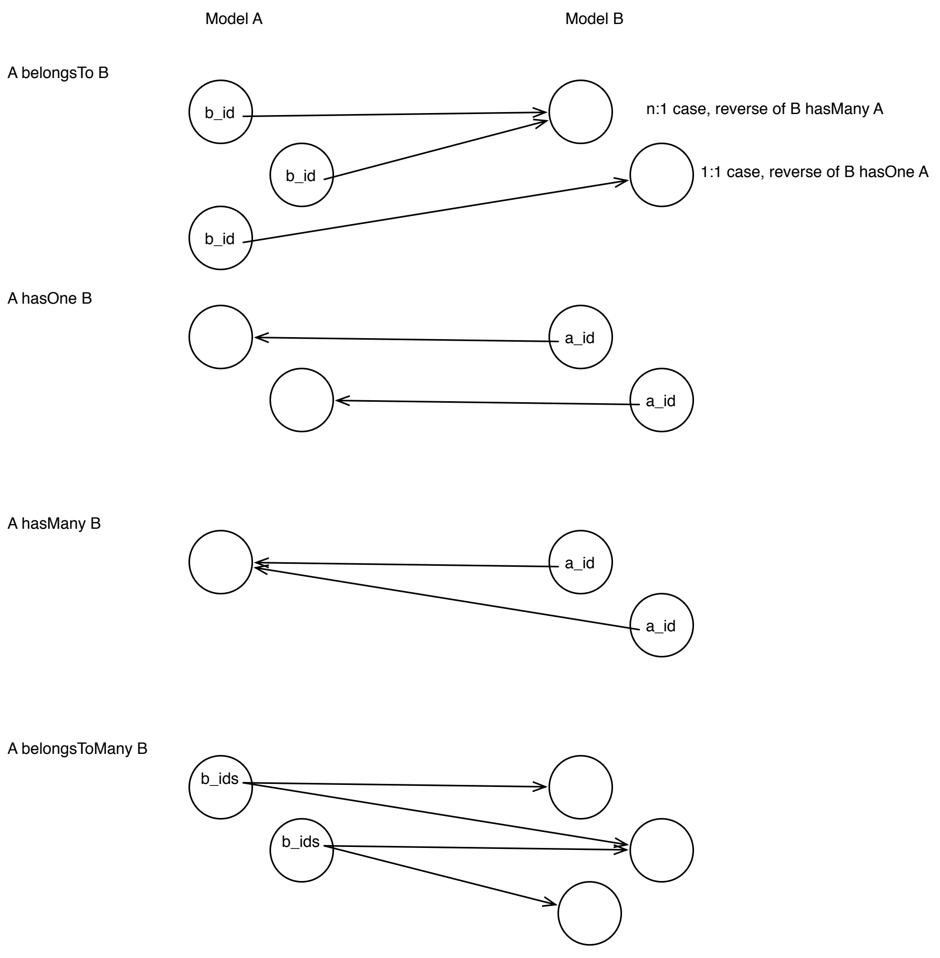

Figure 4.1: *The four association types implemented in the Kupo prototype. The arrows indicate the direction the references are stored in.* Finding *associated instances works regardless where the reference is stored by either following the foreign keys (belongsTo/belongsToMany), or by searching the database for them (hasOne/HasMany).*

The instantiated proxy can now be accessed on the model instance via `aProject.task`. It provides several methods for managing the association, for adding, creating or removing members. To perform these tasks, the proxy manipulates a property in the model instance's data with a name generated from the association name (like `task_id`). It manages the instances state and has an additional, internal state that keeps track of transient changes to the association. When the instance is saved, the callbacks installed by `registerCallbacks` take care of persisting that transient state and resetting the proxy. The transient state consists of a cache (reducing the cost of repeatedly fetching the association members), a collection of unsaved association members and a list of callback-functions to be executed when the host object (the object containing the association proxy) is saved.

This is best illustrated by an example:

1. The `Project` model is defined with `hasMany(Task)`. The association's `registerCallbacks` function installs an `afterSave` callback in `Project`.

2. A new project is instantiated (called `pi` from now on). The proxy is initialized with the project instance, `Task` as the target model, `"tasks"` as the association name and installed in `pi.tasks`.

3. A task `t` is added to the tasks collection via `pi.tasks.add(t)`.

4. The *hasMany* definition given above states that the task should carry the project's ID in a field called `project_id`. The project was not saved to the database yet and thus does not have an ID. Storing the ID in the task has to be deferred.

   The proxy stores the task in its `newInstances` array and pushes a function to its internal list of callbacks that updates the task with the project's ID and saves it.

5. The project instance is saved via `pi.save()`. The `afterSave` callback then calls `this.tasks.afterSave()`. Inside the proxy's `afterSave` method, the list of internal callbacks is examined and the callback we installed in the last step is executed.

   It updates the task with the project's ID (which is available now that the project has been saved to the database) and saves the task itself.

6. Afterwards, the proxy's list of callbacks and the `newInstances` list is cleared. Requests for members of the association will now be served from the database.

This would read slightly different for the other association types of course, but should be sufficient to illustrate the kind of state an association proxy has to manage. The proxies are kept rather simple in the Kupo prototype. The associations are prone to corruption in some cases where any of the instances involved in an association is invalid and does not save properly. Basic cases of association assignment and storage have been successfully unit-tested (see `packages/kupo/tests/association-tests.js`), but extensive support for recovery from arbitrary cases of failed validations in associations would have been beyond the scope of this thesis.

4.3.4 Controllers

Several components implement the *Thin Controller* concept presented in Section 3.1.2.

The controllers forming the adapter layer (see page 38) are kept very slim. A common `Controller` class, ancestor to both the `CustomController` and the `ResourceController`, performs the preparations described in *Controller Support Structures* on page 42, setting up the request object, cookies and session, before the actual request processing begins.

The *resource controller* analyzes the request as described in the previous section and executes the remote call on the addressed target (either an instance or a collection of objects). Before doing so, the model's callbacks have the opportunity to manipulate the request and response object to influence the request processing sequence.

Because of this simple structure, the amount of code required to expose a model's methods as remote procedure calls is kept to a minimum, allowing the developer to concentrate on the business aspects of his models. This is achieved by making the models aware of their context and their integration into the resource controller. This awareness was constructed deliberately, deferring the software engineering principle of *separation of concerns* [Dij82] and favoring ease of implementation for the developer (see *Resource Controller Integration* on page 45).

In cases that can not be handled through the resource controller's architecture, *custom controllers* can be used. The first segment of a URL addresses a custom controller. Should a model of the same name exist, the custom controller takes priority during routing. Action and parameters are retrieved by extracting a JSON-RPC object from the request (either GET or POST). Remote procedure calls sent to a custom controller invoke its manually defined actions.

4.4 Implementation of the Client-side

The framework design is focused on the server-side of web applications and leaves implementation of client-side operations open (the *logic layer*, compare Section 1.6.1). Still, implementing some aspects of client-side development was necessary to enable server-client communication.

Especially the models have to be made available to the client, so they can be reused. Since server-side aspects of the models should stay on the server, protected from public access, the definition of models has been split as described in Section 3.1.3.

4.4.1 Client-side Module Loading

Access to the model definitions is provided through the SecurableModules mechanism used in the Narwhal framework and remoted to the server through a special URL handled by the dispatcher.

Narwhal is written platform-agnostic. The only interface to the underlying JavaScript engine and environment is provided via a special `system` object, created during the bootstrapping process. This object contains several identifier variables and basic filesystem functions used by the Module loader, `read` and `isFile`.

On the client, usually a browser, part of the bootstrapping was emulated in `/public/clientjs/base.js`. This script constructs a system object that uses the XMLHttpRequest API to implement `read` and `isFile`. Both functions send requests to the `/js` path on the server which are then handled and answered by the server-side module loader (see 4.3.2).

4.4.2 Splitting

Giving clients access to the code defining a model's behavior, without exposing secret methods, required splitting the code into a generic part that can safely be exposed and a location-specific part (server-only or client-only). At the same time, model code is split into an abstract part, equal for all models and a concrete part that is defined by individual models designed by the developer. This results in a two-dimensional inheritance structure shown in Table 4.3.

Additional goals for implementing this structure were:

1. **Ease of use** – The developer should write models in a generic fashion and be able to easily extend them with server-specific aspects by manipulating the generic model in a way natural to JavaScript, by removing or adding properties to the objects the model consists of.
2. **Avoiding unnecessary code** – The developer should not be *required* to write his model in a split fashion. If he wants to use a single version of his model for both server and client, he should only have to maintain a single file. If a split approach is taken, the code necessary to define the server-side aspects should be kept to a minimum.

Implementing a two-dimensional inheritance structure that accomplished both these goals was possible through the exploitation of JavaScript's inheritance and function application mechanisms. The complex relationships between classes are shown in Fig. 4.2. Two steps are required for the implementation, described in the following sections.

		Location	
		Generic	Specific
Implementation	Abstract	Basic Aspects shared by all versions of all models	Models extended with specific persistence aspects and/or secret methods
	Concrete	Basic version extended by a concrete implementation	Location-specific version extended by a concrete implementation

Table 4.3: *Two different directions of model inheritance: Location based (generic or client-/server-specific) and implementation based (abstract or concrete). The arrow shows the way through the inheritance chain taken by a concrete, server-specific model: first the generic model is extended location-specific, then the model is extended with a concrete implementation.*

Inheritance Structure

First, all objects related to models had to be separated into a generic and a specific part. The generic `ClassPrototype` contains methods unrelated to persistence that operate on a class-level. To obtain a location-specific `ClassPrototype`, the generic one is inherited and extended with the `find/all` persistence methods. The `CommonInstancePrototype` provides methods shared by all models. To create its location-specific version, it is extended the same way as the `ClassPrototype`.

A concrete model is created by calling the `Model()` constructor which creates an object inheriting from the `ClassPrototype` and extends it with a name and the `spec` object, describing its behavior (see Section 1.6.2). The last step in creating the Model is the creation of the Model's instance prototype by calling the `InstancePrototype(model)` constructor and storing the resulting instance prototype in the same-named property on the model.

To obtain location-specific versions of the concrete model and instance prototype, these constructors described in the last paragraph are replaced by ones that use the location-specific versions of `Class-` and `CommonInstance-Prototype`. The constructors themselves apply their generic counterparts to the new object in construction, effectively "subclassing" them (this is represented through the term *pseudo-super* in Fig. 4.2).

Finally, they execute location-specific initialization statements. In the server-specific Module constructor for example, this means overwriting the generic InstancePrototype previously generated by the generic Module constructor with the server-specific InstancePrototype and installing a connection to the database. In the InstanceProtoype constructor, no location-specific statements are required in the current version.

Loading the Structure

To work with these two versions of models, the *SecurableModules* loader provided by the Narwhal framework had to be extended. This was done by specifying a new loading preference.

Modules are usually identified by their filename in relation to the search paths. Kupo's `model` module, for example, is identified by `kupo/model`. These identifiers are provided without a filename extension. The loader checks wether a file is available by trying to attach different extensions to the identifier. First, the blank string is used, then the `".js"` extension. By modifying the loader to try a `".server.js"` or `".client.js"` extension before `".js"`, specifying locations-specific versions of any module can be achieved by naming the module's file accordingly.

To reference generic module versions from within a location-specific module, the module's identifier can be suffixed with the `.js` extension, causing the loader to fetch the generic module (by extending the identifier with the blank string which produces a match before the other extensions are tried).

Splitting and Loading Combined

The way a developer now implements a split model (`Project` in this example) is to define a generic version in `app/model/project.js` by fetching the Model constructor via `require('kupo/model')`. The developer invokes the Model constructor and exports the resulting model.

He now creates a second file `project.server.js` which loads the generic Project through `require('./project.js')` and modifies it by adding server-specific methods.

To work with the Project model, the framework loads `model/project` (Step 1 in case *a* of Fig. 4.3), which the loader resolves to the server-specific version. Inside `project.server.js`, the generic project is loaded (Step 2 of case *a*). In `project.js`, requiring `kupo/model` loads the abstract, server-specific model (Step 3) which is then extended to a version that has the server-specific aspects of the abstract model (its database persistence) but is missing the server-specific Project extensions.

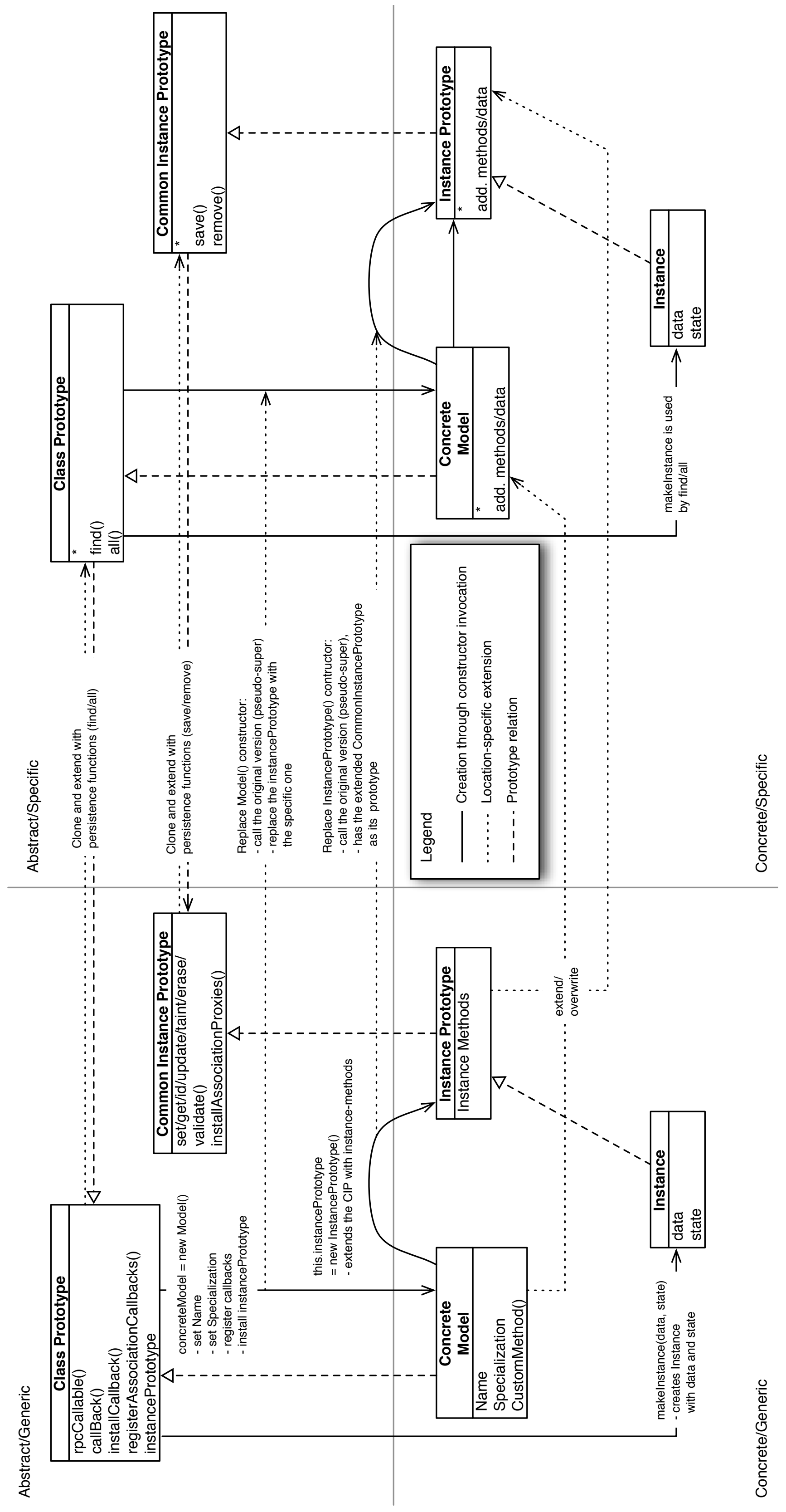
Abstract/Generic
Abstract/Specific
Concrete/Generic
Concrete/Specific
Class Prototype
rpcCallable()
callBack()
installCallback()
registerAssociationCallbacks()
instancePrototype
Common Instance Prototype
set/get/id/update/taint/erase/
validate()
installAssociationProxies()
Concrete Model
Name
Specialization
CustomMethod()
Instance Prototype
Instance Methods
Instance
data
state
concreteModel = new Model()
- set Name
- set Specialization
- register callbacks
- install instancePrototype
this.instancePrototype
= new InstancePrototype()
- extends the CIP with instance-methods
makeInstance(data, state)
- creates Instance
with data and state
extend/
overwrite
Clone and extend with
persistence functions (find/all)
Clone and extend with
persistence functions (save/remove)
Replace Model() constructor:
- call the original version (pseudo-super)
- replace the instancePrototype with
the specific one
Replace InstancePrototype() contructor:
- call the original version (pseudo-super),
- has the extended CommonInstancePrototype
as its prototype
Class Prototype
*
find()
all()
Common Instance Prototype
*
save()
remove()
Concrete Model
*
add. methods/data
Instance Prototype
*
add. methods/data
Instance
data
state
makeInstance is used
by find/all
Legend
Creation through constructor invocation
Location-specific extension
Prototype relation

Figure 4.2: *Two-dimensional inheritance in the framework. The schema combines location- and model-specific refinement of the base classes.*

At this point, a hybrid project model exists that is neither purely generic nor fully specific. The model functionality (persistence etc.) is server-specific, but the project traits are still generic. This hybrid however is never used directly. Only the version obtained through `project.server.js` (Step 1), extending this hybrid with the project's server-specific traits, is used.

If a model does not have different location-specific implementations, requiring the Project model will initialize it differently (case *b* of Fig. 4.3). Requiring `model/project` (Step 4) will directly provide the generic project module. Inside, requiring `kupo/model` will still provide the generic model module (Step 3).

Because of the appearance of the hybrid model in Step 3, this loading mechanism seems complicated at first, but since this peculiarity can be ignored in practice, code splitting is easy and enables the developer to describe his models with minimal effort.

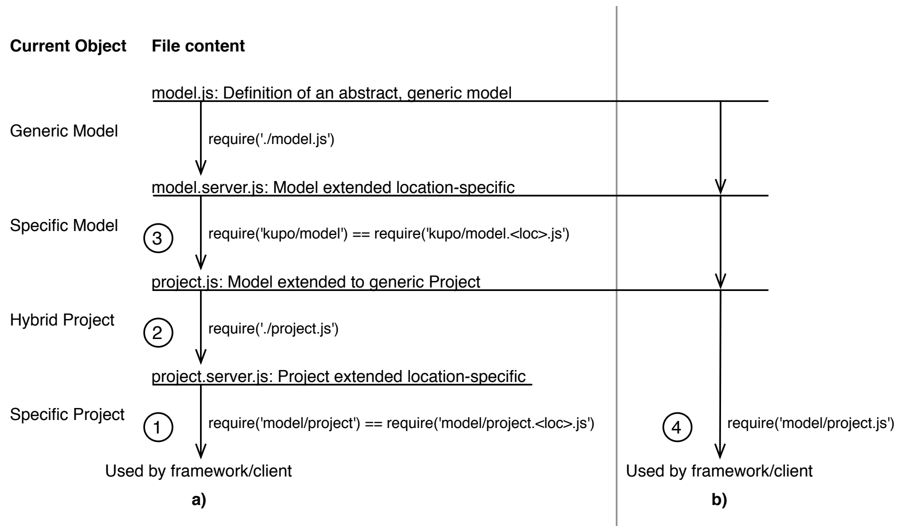
Current Object
File content
model.js: Definition of an abstract, generic model
Generic Model
require('./model.js')
model.server.js: Model extended location-specific
Specific Model
3
require('kupo/model') == require('kupo/model.<loc>.js')
project.js: Model extended to generic Project
Hybrid Project
2
require('./project.js')
project.server.js: Project extended location-specific
Specific Project
1
require('model/project') == require('model/project.<loc>.js')
4
require('model/project.js')
Used by framework/client
a)
Used by framework/client
b)

Figure 4.3: *The split model definitions and the modified module loader work together to obtain a concrete, location-specific Project model by inserting additional steps into the loading sequence on the server. The term* <loc> *in the require statements represents either "server" or "client", depending on the actual location. a) Sequence for a model that has location specific traits. b) Sequence for a model that does not have location specific traits.*

4.4.3 Support for Nested Structures

Support for nested structures and the combined transfer of multiple associated models in a single request, as described in *Associations and Nested Data* on page 46, was not implemented in Kupo. Doing so would have required a great amount of work without providing much insight.

Serialization of object graphs is a solved problem with solutions for JavaScript/JSON already existing [Zyp08b]. Restoring the associated objects from a serialized JSON representation would be trivial. Supporting deep associations for validating and persisting objects to the database would require a more robust implementation of the associations (see Section 4.3.3).

4.4.4 Automated Delegation

Automated delegation, that is, an automatic generation of client-side methods that delegate their execution to a method on the server, was not implemented in Kupo due to time constraints. Such a feature would not be very complicated to realize, however.

Methods to be delegated could be declared in a list in the `spec` object, similar to the `callables` array. Upon initialization of the model, for each name in this list a method would be generated by following the schema in Listing 4.2.

Listing 4.2 Template for generating an automated delegation

```
model[<methodname>] = function() {
 var args = Array.prototype.slice.call(arguments);
 return this.model.connection.call(<methodname>, args)
}
```

4.4.5 Example Application

To illustrate how a client and the server can work together, a small example application was created, using the project and task models that served to illustrate code examples throughout the entire thesis.

The application initializes modules client-side, loads projects from the server, loads tasks from the server, and allows to create and validate tasks and assign them to projects. After starting Kupo and MongoDB, it it accessible by opening `http://localhost:8080/index.html` in the browser .

The example does not use a proper MVC separation on the client-side since this is out the scope of this thesis. Instead it uses a small HTML skeleton and the jQuery framework to load and display models. Data is injected directly into the DOM.

Chapter 5
Evaluation

In Chapters 1 and 3, motivations for creating a new framework based on server-side JavaScript were explained and developed into a design. To evaluate this design, its aspects need to be examined in combination as well as individually and compared to existing approaches.

5.1 Goals and Design

In Chapter 1, a number of goals was formulated that aimed to solve the problems that still plague web development despite the great progress programming tools in that area have made in recent years.

The problems identified were

- A heterogenous mix of languages
- Redundancy in code
- Necessity of "Glue Code"
- Complexities of Object-Relational Mapping

These problems are all interrelated and have their origin in the mix of different languages used in web development. Using JavaScript on the server opened up the possibility to create a design free of these problems through the integration of four corresponding goals formulated in section 1.4.

5.1.1 A Single Language

Using only JavaScript for all server-side aspects of the framework and the client-side of a small example app (see Section 4.4.5) was successful.

The Rhino engine enabled using the MongoDB Java driver directly in JavaScript code in a seamless way. Given more time, it would have easily been possible to implement a native JavaScript driver for MongoDB or to use the HTTP interfaces of CouchDB or Persevere. Querying the database using JavaScript and working with the returned objects isn't completely seamless due to limited flexibility in MongoDB's Query-by-Example approach and a minor mismatch between JSON and the BSON format. These problems are negligible implementation details though, and not fundamental issues.

On the client, jQuery was used to demonstrate how to write a control layer without the use of a separate template language, by directly injecting data from the models into the DOM.

Some other JavaScript based frameworks naturally achieve using a single language too, as long as a JSON database is used. This is the case in Persevere (when used as an application server) and Helma but not in Aptana's Jaxer, which makes extensive use of SQL and HTML.

Several other, non-JavaScript frameworks try to prevent the developer from using multiple languages too. *Ruby on Rails*, for example, has facilities to generate client-side JavaScript using Ruby Code (RJS) and hides much of its relational database interaction behind a powerful ORM layer (ActiveRecord) [THB+06]. The *Google Web Toolkit* and Microsoft's *ASP.NET* allow developers to write web applications using only a single language. Both achieve this goal by compiling the host language to HTML and JavaScript, adding additional layers of computation and complexity to the software stack (see 1.5).

5.1.2 Code-Reuse Between Server and Client

Using the models on the client has yielded the desired effects. Validations defined on a model can be applied to an instance derived from form fields and give the user feedback about missing data before sending the instance to the server. Working with full-featured models on the client further enables access to associations and custom methods and improves data integrity in the instances by providing a clearly defined interface instead of pure data objects.

Creating location-specific versions of a module, by defining them in terms of their differences, works well in JavaScript because of the language's ability to freely extend or modify defined objects.

Of the other examined JavaScript frameworks, only Jaxer implements a form of code sharing. Helma and Persevere provide definition of models on the server-side, but Helma's general approach is more traditional in keeping more application code on the server, and Persevere limits its models to offering JSON-RPC services and schema definitions. Trimpath Junction offers the possibility to emulate an application server by actually executing server-side code in the browser [jun09]. Google Web Toolkit allows to use the same models (written in Java) for client- and server-side aspects of an application, compiling the client-side code to JavaScript for deployment [Goo09a].

5.1.3 Elimination of Glue Code

This aspect is harder to evaluate than the other goals since no complex enough application was developed to reveal how much glue code would have to be written and which of that glue code would have been avoided by providing additional support in the framework.

To be able to make a statement on this issue, several independent applications would need to be evaluated. Generally, however, it can be seen as a rule of thumb that increasing complexity always increases the need for configuration [vGB01]. In that regard, the simple design devised in this thesis has an advantage, a large part of reducing glue code however would have to be achieved through libraries for dealing with common tasks.

An example for JavaScript's special suitability towards this goal is the automated delegation drafted in Section 4.4.4.

It should also be regarded that not all of the server-side complexity has been avoided, but that some of it has been shifted to the client, into the logic layer (see Section 3.1.2) since control flow management for the application still has to be implemented somewhere.

Reduction of glue code and extremely easy ways to implement common tasks are what made the Ruby on Rails framework popular [Dum05]. Its strategy towards using conventions and metaprogramming spawned inspirations and imitations all over the web development community. Trimpath Junction, for example, is a clone of Rails implemented in JavaScript on top of Helma (see page 33).

How well exactly other frameworks reduce the need for glue code is hard to judge without actually using them but some general observations and assumptions can be made: Java as a static, compiled language is largely unsuitable for *elimination* of glue code. There exist many approaches towards automatically *generating* glue code [vGB01, DGHK05], but since the entire application's configuration has to be known at compile-time, flexibility of

Java-based stacks is still limited. Most dynamic, interpreted languages used for web development (PHP, Python, Perl) do not have this problem. Since the features and the MVC approach taken by Rails have become popular, many frameworks in these languages have adopted at least some of them (in [rai06], several PHP frameworks are described).

5.1.4 Tight Integration of a JSON Database

Integrating MongoDB into the framework as a persistence layer was successful. MongoDB's simple API structure and its Query-By-Example approach to locating objects were a good match for the frameworks simplicity. Persevere's additional features are mostly optional and it could have been used just as well or even better as MongoDB, assuming a HTTP client interface would have been available in Narwhal.

The general direction away from relational databases has not turned out to be a disadvantage. Similar to the avoidance of glue code, a more convincing statement the use of a JSON database would require more real-world programming experience than possible in the scope of this thesis.

Experiences other frameworks had with JSON databases are not easy to find, since non-relational databases are not used widely. Because the JSON format specifies just a few simple data types and structures that are available in any language, integration of a JSON database can be expected to be simple for other languages too. Using JavaScript for the application server has the added benefit of absolutely no impedance mismatch between host- and database-language, avoiding the need for an additional conversion layer.

Although using a JSON database simplifies database operations compared to using a traditional relational database, performing queries and save operations manually is still necessary. In this regards, the combination of JavaScript and a JSON database leaves room for improvement toward even closer integration.

5.2 Use Cases Revisited

In Section 1.3, several use cases were presented that pose challenges to existing approaches in web application development. It was shown how the abstract problems in Section 1.2 caused the challenges. In this section, it will be examined how achieving the formulated goals, described in the previous section and in 1.4, helps to overcome these challenges and how other frameworks and their solutions perform in this regard.

5.2.1 Model Validation

Validating models in multiple languages at different stages of the application involves the danger of having implementations of the validation rules diverge. The abstract design deficiency behind this case is the use of multiple redundant implementations in different languages.

To solve this problem, code sharing can be used. By implementing models and their validations in JavaScript, and using the code for models on both the server and the client, only a single implementation of the validation rules is used. A practical implementation for validating forms on the client-side could create a model instance from form data, let the instance validate itself, and use generated validation errors to mark the respective form fields as invalid to the user.

When code sharing is not possible because the server-side of the application is not implemented in JavaScript, the remaining possibilities for form validation are to either submit the form regularly or to use AJAX-based validations. Both variations send the form data to the server. In AJAX-based validation, the server validates the data and sends information about errors back to the client in an the response. When AJAX is not used, the server can render the entire page again, marking invalid fields to the user. In each case the request to the server carries the overhead of a HTTP roundtrip and the server-side data processing.

Another possibility is to generate client-side form-validating JavaScript from the validations defined in the server-side models. This is, however, impossible with custom, imperative validation functions and only works with declarative, interpreted validations.

5.2.2 Data Transformation

The multiple stages of data transformation in existing web application stacks, between the database in the back-end and the DOM in the frontend, involve a lot of expensive string processing, contain impedance mismatches and open up potential weak spots for injection attacks. Multiple design deficiencies of traditional web application frameworks are involved in this use case: Object-relational mapping is required to convert data from the relational to an object-oriented structure. Different languages require data conversion between them. The conversion code is glue code because it does not immediately contribute to solving the problem the application was designed for.

Using JavaScript as a single language and a JSON database as the database remove many conversion steps. The application can work with JavaScript objects all the time and only has to serialize data to JSON strings for communication between server and client. This also eliminates opportunities for injection attacks. Keeping the server lean and shifting most of its tasks to the client better distributes the remaining data processing (HTML templates, for example, that do not have to be rendered on the server anymore).

As long as multiple languages and relational databases are involved, most data transformation steps simply cannot be avoided. Some of the JavaScript-based frameworks examined in Section 2.6 use JSON databases instead of (or in addition to) relational databases, yet most of them still perform front-end rendering tasks on the server and send complete HTML pages to the browser. Apatana's Jaxer adds even more transformation steps, since it parses HTML templates into a DOM before processing server-side JavaScript. The modified DOM is converted back to HTML afterwards, before being sent to the client.

The solution used in the framework designed in this thesis leaves room for improvement. Although removing transformation steps removes potential points for injection attacks, the use of JavaScript increases the possible damage a successful injection could deal. A malicious function injected into the framework could potentially modify essential functionality. Extra caution has to be taken to prevent these injections.

Even though a lot of the transformation steps can be successfully removed by using JavaScript and JSON, it is desirable to additionally remove the transformation between the database and the JavaScript interpreter. To move data between the running application and the database, JSON strings are used. A tighter integration between the database and the interpreter could remove this step. This would require the implementation of a database driver on the JavaScript interpreter level.

5.2.3 Exposure of Objects Through APIs

Creating web services by providing an interface to an application's models through the control layer can result in repetitive, redundant code if very similar actions have to be written for many exposable methods of every model in the application.

To solve this problem, the designed framework uses the resource controller, a single controller, responsible for exposing the methods of all available models to clients in identical ways. This is made possible by providing strict standards for the models' behavior. The separation of responsibilities between models and controllers is relieved in a clearly described way, by storing controller callbacks and information about the availability of methods in the model's definition. This avoids the need for additional files that merely contain information for handling remote procedure calls and emphasizes the framework's focus on the object exposure via JSON-RPC.

Other frameworks are providing similar features. Ruby on Rails' support for REST popularized the attitude of treating web applications and web services as equal, and inspired the behavior described in the previous paragraph. Rails has always provided strong conventions for the behavior and structure of models and controllers, even more after adopting REST. The resource controller in this thesis' design takes the ideas in Rails' popular *resource controller* plugin further.

Of the other JavaScript based frameworks, Persevere is especially notable. Its exposure of model methods as remotely callable procedures is similar to the resource controller, the models in Persevere, however, are intended for server-side use only and not for use on the client.

The main problem with the resource controller is the possible inflexibility of its mechanics. This can unfortunately only be properly evaluated in extensive practical tests.

5.2.4 Dynamic Object Properties

Objects with dynamic properties of unknown, arbitrarily nested structure can not easily be mapped to relational databases. In extreme cases, doing so can degenerate a relational database to a key-value store and require sophisticated support from object-relational mapping facilities. Additionally, complex data representations on the client-side (as JavaScript objects) need to be serialized and converted to objects in the server-side implementation language. As in the last example, such conversion requires glue code and additional processing resources on the server.

Again, using a JSON database avoids the need for object-relational mapping. JavaScript on the server-side and identical representations of objects by using the same models on client and server makes conversion between languages unnecessary.

Frameworks written in other languages can free themselves from some of the mentioned problems by using non-relational databases as well. However, they still have to find a way to convert complex objects from JavaScript to the data structures offered by their host language.

5.2.5 Storage and Retrieval of Compound Objects

Storing multiple, associated objects was identified as a special case of the *Dynamic Object Properties* problem. To store instances together with independent, associated objects, the compound structure has to be disassembled, and their parts stored independently in the database. Explicit association information is needed, to reassemble the compound objects later.

Storing multiple associated objects at once means validating them, generating keys and storing those keys as association keys. These tasks have to be performed in a sequence that does not create dependency problems during validation or foreign key assignment. Common lightweight web application frameworks do not offer support for this complex procedure and prefer simpler data structures, not least because the other problems described earlier grow with the complexity of handled objects as well.

In simple cases, storing can be done manually, by specifying the required steps explicitly in the application. Since multiple models are involved, this poses the risk of creating ad-hoc code in unsuitable locations. Because it can be unclear *where* exactly to specify these steps, an incautious (or frustrated) developer might put related code into the application's control layer or into the wrong model, hurting the application's maintainability.

The code required to perform these steps is glue code the developer has to write manually, which to avoid is one of the goals of the design developed in Chapter 3. Additionally, without support for storing compound objects, the usefulness of the resource controller is limited to very simple operations involving single objects. The use of JavaScript on the server and a JSON database can only supports this measure and it not essential to it.

Therefore, such a mechanism was not implemented in the example framework. For the time constraints, the amount of work required would have been too great, without yielding useful insight.

Since version 2.3, the Ruby on Rails framework supports a feature called *Nested Forms* that allows its *ActiveRecord* ORM layer to perform database operations on compound objects [Koz09].

5.3 General Architecture Evaluation

Overall the architecture designed in this thesis can be considered successful and promising in regard to an exploration where web development might be headed. As additional benefits of the ones lined out in the previous sections can further be listed:

- Dynamic applications become easier to implement, because the developer can concentrate entirely on the client-side of the application and the definition of models.

- The distribution of responsibilities between client and server becomes clearer. The server provides the API and the client implements the user interface on top of it.

- By shifting control flow management and rendering from the server to the client, computing power is better distributed, freeing up resources on the server.

An architecture this opinionated of course comes with some expenses:

- Control over the program is given up. The API on the server has to be consistent and secure to prevent invalid data to be stored since the controllers and models running on the client can not be trusted.
- Incompatibilities between browsers' JavaScript implementations are less severe than just a few years ago but they still exist. In general, running the application in the browser means running it in an unknown, always changing environment. Adjusting for this requires additional work from the developer.
- The openness of JavaScript makes the server-side architecture more dynamic at the expense of stability. Since an exploit injected into the server process can potentially alter the entire program, special countermeasures have to be taken to prevent injections. Developers need to be aware of that.

Balancing security and flexibility always involves tradeoffs. It depends on the priorities of the developer what kind of approach is best suited for his application. It can also be expected that, as JavaScript matures (such as with the first steps taken in ECMAScript 5), the language becomes more robust and better suited for large scale application development than today.

5.4 Perspectives and Problems

The combination of the new concepts lined out in the previous sections serves as a strong motivation to shift more and more parts of web applications to the client, turning the server-side part of the application into a pure web service.

From this, applications benefit in two ways. By reducing internal coupling between the front-end and the back-end of the application, maintainability is increased. A (possibly public) API is created adding value to the application itself and the web service ecosystem it exists in.

5.4.1 Remaining Tasks

The concepts devised in this thesis and their implementation are very prototypical. A lot of work remains to be done to better evaluate the design and to solve the remaining problems. In this section, these tasks will be presented by topic and subsequently prioritized in an overview.

Testing and Evaluation

The concepts and the implementation have to be tested much more thoroughly. Ideally, the framework would be used to create several small applications to reveal weak points and missing functionality. At the same time, functionality would have to be developed further to gain insight in how the concepts perform in practice.

Concrete questions that could not be answered satisfactorily yet include:

- How much glue code is actually avoided by using strict conventions?
- What missing support functionality would have to be provided to ease the implementation of common tasks?
- Is the resource controller really practicable? Is it too restrictive?

Integrating Platform and Database

Just like Narwhal, the Kupo implementation was intended to be run on different JavaScript engines. The current limitation to Rhino simply had practical reasons. Making Kupo platform independent would fulfill its original intention.

Integration of a JSON database needs to be improved. MongoDB was useful for a prototype, but peculiarities of its BSON format and limitations of the Query-by-Example method would have to be overcome for a practical use of the framework. Writing a native driver in JavaScript or modifying the existing adapter to the Java driver would achieve this.

Persevere's JSONQuery features make it an attractive candidate for a database back-end too. Integrating Persevere into the framework could be done by writing a HTTP driver for its REST interface or by integrating its engine directly into the framework via Rhino's LiveConnect feature. A port of Kupo to *v8cgi* could make use of v8cgi's HTTP implementation.

In a next step, the need to generate and parse JSON strings for communicating with the database could be removed. With a database driver tightly integrated (that rules out using MongoDB's Java driver or Persevere's REST API), the database's internal representation can be used to directly generate JavaScript objects. In the other direction, JavaScript objects could be converted directly into the database format without an intermediary string representation.

Should this kind of integration be successful, moving to a completely seamless database integration, persisting objects transparently, becomes a possibility. Accessing the database at the interpreter level would open up the chance to enhance objects with persistence capabilities.

Security

JavaScript does not have security features. In the browser, this is not a problem, since scripts are executed in a sandbox with very limited possibilities for doing damage. On the server, a malicious function injected into the framework can potentially alter any aspect of the system, access filesystem functions and lead to serious corruption of the entire application server.

To prevent this, protection mechanisms against injections have to be designed and implemented. These could be based on features of the used engine, on the SecurableModules system [AK09a, AK09b], or on new security features in upcoming JavaScript standards like *ECMAScript Harmony* (informal name) [Eic08a].

Missing Features

Some features were planned for inclusion into the framework prototype but could not be realized (see Section 4.2.3).

Support for nested structures and compound objects is a solved problem, in theory, but requires much effort to efficiently implement. This functionality would require extensive unit tests and needs to be examined in regard to its actual usefulness in application development.

Automated delegation of model methods only available on the server was not implemented due to time constraints. This is not a complicated task, however, neither in theory nor in practice.

A lot of helpers for solving common tasks and avoiding glue code are still missing. This is mainly because judging which features are actually needed and which are not, is hard to do in an abstract scenario. The needed helpers would be most efficiently identified during implementation of a more complex application. Some helpers that are definitely missing, are authentication and authorization assistants.

Examination of the Client Side

This thesis was focused on the server-side of web application development. Yet a lot of the integrated concepts are tied closely to the idea of shifting responsibilities that are actually user interface related, but traditionally performed on the server, to the client. How the client would actually perform its new tasks was left largely unanswered.

It is generally not a disadvantage to leave the choice of a client-side framework to the developer, but it needs to be examined where this thesis' design requires additional client-side support and what this support will be.

Overview

Absolutely necessary are support for automated delegation, authorization and authentication helpers, and improving the MongoDB integration.

Reasonably performing further tasks would require the development and examination of a more complicated application with the framework. Especially the resource controller needs to be evaluated and missing helpers for common tasks need to be identified.

Over the course of the implementation of an application, the potential need for stronger client-side support will likely be revealed.

Support for nested structures and compound objects requires cautio to implement properly. The implementation will likely be easier with improved database support, that is, the integration of Persevere and elimination of JSON strings as an exchange format for the database.

Security is not essential for gaining insights into the advantages and disadvantages of the architecture but a necessary feature, should the framework ever be used in a public environment.

An integration of the database at the interpreter level is pure speculation at this point.

5.4.2 Future Work

Several issues remain that were not solved in the work on this thesis. To improve client-server communication beyond simple remote procedure calls, new ways for executing methods remotely or distributed in context of a pure JavaScript environment need to be found. The present framework can provide a basis for research in this direction.

As a first step towards more security and data consistency between client and server, the communication between both parties could implement an object capability model as described in [Clo08]. During an early phase of this thesis' design a different approach to addressing model classes and instances on the server raised questions on how to authenticate access to members of a secured collection in a context-free way. Among the devised solutions one was modeled after the object capability model. The entire addressing approach was replaced by something simpler, however, because an implementation would have been beyond the scope of this thesis.

The design in this thesis could provide the basis to support a consumer-producer style processing model for web applications. In such an environment, server and client would work together on an object, each in a way that is not possible on the remote end. The server might perform database operations while the client lets the user interact with the object. Blocking calls or promises could be used to suspend execution on one side of the communication until the other end can provide input that allows computation to proceed. For example, a function could be called on the server to fetch information from the database. The server might recognize that not enough information was provided to fulfill the request and return a promise to the client that upon access will spawn a dialog asking for more information, send this information back to the server, complete the request there, and return with a new Promise that can be fulfilled. An architecture directed towards asynchronous messaging between event loops that could serve as orientation and provide underlying mechanisms has been implemented in *Waterken* with the server-side of the stack written in Java [Clo09].

Slightly related, another measure to improve communication between server and client may lie in remoting closures while keeping the traditional style of the client making single request after request to the server:
Instead of manually maintaining a session on both ends of the communication, and accessing it explicitly on the server, the session could be moved into a closure in which functions on the server are executed. A function, remotely run on the server, would produce a return value and operate on the closure for side effects. The modified closure and the return value would then be returned to the client. There are some caveats to this model that need to be solved: Within the limits of the interpreter, explicit access to a function's closure is not possible and sending a closure of potentially unknown size is infeasible for remote connections. The closure might also contain information that explicitly may not be exposed. Therefore, a "closure-like" object would have to be constructed instead, providing a context in which remote execution of the function happens. Ideally this construction could be automated.

Chapter 6
Conclusion

Developing and implementing a framework design on the basis of software and technologies that are still in very active development, in a field that is currently undergoing rapid changes has proven challenging.

In this thesis, some of the ideas that are currently discussed in the web development community have been combined to a new approach. The distinctive property of that approach is that it does not hide its concepts behind overly thick and complicated layers of abstractions, adapters and libraries, but that it tries to implement them on the mechanisms and technologies available on the web.

The way of developing software with leaner stacks that embrace web standards has proven useful in practice and is used successfully by thousands of developers everyday.

The combination of web standards with a single-language environment for creating software is promising and can open up new perspectives for web application development.

Appendix A

Kupo Information

This chapter contains information on where to download, how to install, and how to run Kupo and its tests.

A.1 Requirements

Kupo should run on all Unix systems. It was tested with Mac OS X 10.5.7 and Debian 5.0. To use Kupo, you need to have a running MongoDB Server on your machine. Binaries for many operating systems are available at `http://www.mongodb.org/display/DOCS/Downloads` You also should have git[1] installed to fetch Kupo from its repository.

A.2 Installation

To install Kupo, just place the source directory somewhere and make sure the correct versions of the Jack[2] and Narwhal[3] frameworks are in place under the `packages` directory.

The best way to install Kupo is to clone the repository using git.

1. `git clone git://github.com/janv/kupo.git`
2. `cd kupo`
3. `git submodule update --init`

[1] `http://www.git-scm.com/`
[2] `http://github.com/janv/jack/master`
[3] `http://github.com/janv/narwhal/master`

The URLs for the Jack and Narwhal repositories given here point to clones of the respective repositories, created to be in sync with the Kupo repository. The original repositories can be found at [jac09, nar09].

A.3 Startup

To start Kupo, MongoDB needs to run first. Start the MongoDB server with `"mongod --dbpath <path_to_kupo>/db run"`. This causes MongoDB to use the `db` subdirectory of Kupo to store its data files.

Afterwards, execute `"packages/narwhal/bin/narwhal ."` in the kupo directory. Alternatively the provided startup script `./start` can be used.

Listing A.1 The Kupo directory layout

```
.                   - The application directory
jackconfig.js       - Initialization file for Jack
main.js             - Initialization file for Narwhal
package.json        - Package descriptor for Narwhal
start               - Startup shellscript
app/                - Contains application specific
    controller/       models and controllers
    model/
db/                 - Can be used to store the database
packages/           - Packages required by the app
    jack/           - Jack framework
    kupo/           - The Kupo framework as a Narwhal package
        jars/       - Contains the mongo-java-driver jar
        lib/kupo/   - The actual modules
        tests/
    narwhal/        - Narwhal framework
public/             - Publicly accessible static content
    clientjs/       - Client-side JavaScript code
```

The Kupo repository is laid out as a Narwhal package. The root directory contains the application that is developed (a sample application at the moment). All required frameworks are located as packages in the `packages` directory, so they are automatically discovered by `narwhal`. These packages are *Jack*, *Narwhal* itself and the Kupo framework files.

The file `main.js` is executed by `narwhal` to run the package. `Main.js` runs the `jackup` executable of the Jack framework, passing it the current directory as a Jack application. Jack applications are initialized using the file `jackconfig.js` in which the *app* (a simple function adhering to the Jack protocol) is exported through the `exports.app` variable.

A.4 Tests

The Kupo release includes unit tests for some of its components. To run them you need to:

1. Start MongoDB
2. Change to `<path_to_kupo>/packages/kupo/tests`
3. Run `"../../narwhal/bin/narwhal ."`

A.5 License

Kupo is licensed under the MIT License

Bibliography

[10g09a] *Mongo Concepts and Terminology*, 10gen, June 2009. [Online]. Available: http://www.mongodb.org/pages/viewpage.action?pageId=131424

[10g09b] (2009, June) Mongodb. Open Source Project. 10gen. [Online]. Available: http://www.mongodb.org/

[ACKM04] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web services: concepts, architectures and applications.* Springer, 2004.

[AK09a] I. Awad and K. Kowal, "Module system for es-harmony," 2009. [Online]. Available: https://docs.google.com/Edit?tab=view&docid=dfgxb7gk_34gpk37z9v

[AK09b] I. Awad and K. Kowal, "Modules," Presentation Slides, 2009. [Online]. Available: http://docs.google.com/Presentation?id=dcd8d5dk_0cs639jg8

[Apa09a] (2009, July) Apache cocoon. Open Source Project. Apache Foundation. Version 2.1. [Online]. Available: http://cocoon.apache.org/2.1/index.html

[Apa09b] (2009, June) Couchdb. Open Source Project. Apache Foundation. [Online]. Available: http://couchdb.apache.org/

[Apa09c] *Introduction to CouchDB views*, Apache Foundation, June 2009. [Online]. Available: http://wiki.apache.org/couchdb/Introduction_to_CouchDB_views?action=recall&rev=19

[App05] *Dynamic HTML and XML: The XMLHttpRequest Object*, Apple Inc., June 2005. [Online]. Available: http://developer.apple.com/internet/webcontent/xmlhttpreq.html

[App09] *Apple Safari 150 Features*, Apple Inc., June 2009. [Online]. Available: http://www.apple.com/safari/features.html

[Apt09] (2009, June) Jaxer. Open Source Project. Aptana Inc. [Online]. Available: http://www.aptana.com/jaxer

[Ark07] A. Arkin, "Read consistency: Dumb databases, smart services," September 2007. [Online]. Available: http://blog.labnotes.org/2007/09/20/read-consistency-dumb-databases-smart-services/

[Atw08] J. Atwood, "Maybe normalizing isn't normal," July 2008. [Online]. Available: http://www.codinghorror.com/blog/archives/001152.html

[Awa] I. Awad. Message on serverjs mailing list from mon, 2 feb 2009. eMail. [Online]. Available: http://groups.google.com/group/serverjs/msg/752ab381d9eb5194

[Bar01] G. Barish, *Building scalable and high-performance Java Web applications using J2EE technology.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[BGP00] L. Baresi, F. Garzotto, and P. Paolini, "From web sites to web applications: New issues for conceptual modeling," in *ER '00: Proceedings of the Workshops on Conceptual Modeling Approaches for E-Business and The World Wide Web and Conceptual Modeling.* London, UK: Springer, 2000, pp. 89–100.

[BLFN96] T. Berners-Lee, R. T. Fielding, and H. F. Nielsen, *Hypertext Transfer Protocol – HTTP/1.0*, Internet Draft, February 1996, in W3C archives. [Online]. Available: http://www.w3.org/Protocols/HTTP/1.0/spec.html

[Boy09] N. Boyd, *Scripting Java*, Mozilla Foundation, July 2009. [Online]. Available: http://www.mozilla.org/rhino/scriptjava.html

[Bur87] S. Burbeck, *Applications Programming in Smalltalk-80: How to use Model-View-Controller (MVC)*, 1987. [Online]. Available: http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html

[Cap08] C. Cap, "Suche nach objekten," Presentation Slides, May 2008.

[Cha01] S. Champeon, "Javascript: How did we get here?" April 2001. [Online]. Available: http://www.oreillynet.com/pub/a/javascript/2001/04/06/js_history.html

[Che76] P. P.-S. Chen, "The entity-relationship model—toward a unified view of data," *ACM Trans. Database Syst.*, vol. 1, no. 1, pp. 9–36, 1976.

[Clo08] T. Close, "Web-key: Mashing with permission," in *WEB 2.0 SECURITY AND PRIVACY 2008*, 2008. [Online]. Available: http://w2spconf.com/2008/papers/s4p2.pdf

[Clo09] T. Close. (2009, July) Waterken. Open Source Project. [Online]. Available: http://waterken.sourceforge.net/

[CNW01] F. Curbera, W. A. Nagy, and S. Weerawarana, "Web services: Why and how," in *In OOPSLA 2001 Workshop on Object-Oriented Web Services. ACM*, 2001.

[Cro06] D. Crockford, “The application/json media type for javascript object notation (json),” IETF RFC 4627, July 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4627.txt?number=4627

[Cro08] D. Crockford, *JavaScript: The Good Parts.* O’Reilly Media, Inc., 2008.

[Dan09] K. Dangoor, “What server side javascript needs,” January 2009. [Online]. Available: http://www.blueskyonmars.com/2009/01/29/what-server-side-javascript-needs/

[DG08] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[DGHK05] L. Davis, R. Gamble, M. Hepner, and M. Kelkar, “Toward formalizing service integration glue code,” in *Services Computing, 2005 IEEE International Conference on*, vol. 1, July 2005, pp. 165–172 vol.1.

[Dij82] E. W. Dijkstra, *Selected writings on Computing: A Personal Perspective.* New York, NY, USA: Springer-Verlag, 1982, pp. 60–66.

[Do07] C. Do. (2007, June) Seattle conference on scalability: Youtube scalability. Google Tech Talk. [Online]. Available: http://video.google.com/videoplay?docid=-6304964351441328559

[Doj09a] *JSONQuery - Persevere Online Documentation*, Dojo Foundation, June 2009. [Online]. Available: http://docs.persvr.org/documentation/jsonquery

[Doj09b] *REST - Persevere Online Documentation*, Dojo Foundation, July 2009. [Online]. Available: http://docs.persvr.org/documentation/http-rest

[Dum05] E. Dumbill, “Ruby on rails: An interview with david heinemeier hansson,” August 2005. [Online]. Available: http://www.oreillynet.com/pub/a/network/2005/08/30/ruby-rails-david-heinemeier-hansson.html

[ecm99] *ECMA-262: ECMAScript Language Specification*, 3rd ed. Geneva, Switzerland: ECMA (European Association for Standardizing Information and Communication Systems), December 1999. [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-262.htm

[ecm09] *ECMA-262, 5th Edition Candidate Draft April 2009.* ECMA (European Association for Standardizing Information and Communication Systems), April 2009. [Online]. Available: http://www.ecma-international.org/publications/files/drafts/tc39-2009-025.pdf

[Eic08a] B. Eich, "Ecmascript harmony," eMail, Mozilla Foundation, August 2008. [Online]. Available: https://mail.mozilla.org/pipermail/es-discuss/2008-August/006837.html

[Eic08b] B. Eich, "Tracemonkey: Javascript lightspeed," August 2008. [Online]. Available: http://weblogs.mozillazine.org/roadmap/archives/2008/08/tracemonkey_javascript_lightsp.html

[Erl05] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005.

[Esp08] D. Esposito, *Programming Microsoft ASP.NET 3.5.* Redmond, WA, USA: Microsoft Press, 2008.

[Fie00] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, IRVINE, 2000. [Online]. Available: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[Fow97] M. Fowler, "Dealing with properties," 1997. [Online]. Available: http://martinfowler.com/apsupp/properties.pdf

[Fow02] M. Fowler, *Patterns of Enterprise Application Architecture.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[Gar05] J. J. Garrett, "Ajax: A new approach to web applications," February 2005. [Online]. Available: http://www.adaptivepath.com/ideas/essays/archives/000385.php

[GF06] A. Gal and M. Franz, "Incremental dynamic code generation with trace trees," Donald Bren School of Information and Computer Science, University of California, Irvine, Tech. Rep. 06-16, 2006. [Online]. Available: http://www.ics.uci.edu/~franz/Site/pubs-pdf/ICS-TR-06-16.pdf

[Gol09] J. Golick. (2009, July) Resource controller. Open Source Project. [Online]. Available: http://github.com/giraffesoft/resource_controller/

[Goo09a] *Product Overview - Google Web Toolkit*, Google, July 2009. [Online]. Available: http://code.google.com/webtoolkit/overview.html

[Goo09b] *V8 Design Elements*, Google, July 2009. [Online]. Available: http://code.google.com/apis/v8/design.html

[Goo09c] (2009, July) V8 javascript engine. Open Source Project. Google. [Online]. Available: http://code.google.com/p/v8/

[Ham08] N. Hamilton, "The a-z of programming languages: Javascript," July 2008. [Online]. Available: http://www.computerworld.com.au/article/255293/-z_programming_languages_javascript?fp=4194304&fpid=1

[Han09] D. H. Hansson, "Discovering a world of resources on rails," Presentation Slides, June 2009. [Online]. Available: http://media.rubyonrails.org/presentations/worldofresources.pdf

[HB04] H. Haas and A. Brown. (2004, February) Web services glossary. W3C Web Services Architecture Working Group. W3C Working Group Note 11. [Online]. Available: http://www.w3.org/TR/ws-gloss/

[Hel09] *HopObject - Helma Javascript Reference*, Helma, July 2009. [Online]. Available: http://helma.server-side-javascript.org/reference/HopObject.html

[IBNW09] C. Ireland, D. Bowers, M. Newton, and K. Waugh, "A classification of object-relational impedance mismatch," in *DBKDA '09: Proceedings of the 2009 First International Conference on Advances in Databases, Knowledge, and Data Applications.* Washington, DC, USA: IEEE Computer Society, 2009, pp. 36–43.

[jac09] (2009, July) Jsgi & jack. Open Source Project. [Online]. Available: http://jackjs.org

[Job95] S. P. Jobs, "The future of objects - openstep day at object world 1995," Keynote presentation, NeXT Computer Inc., August 1995. [Online]. Available: http://video.google.com/videoplay?docid=5888348343612063265

[Jon09] R. Jones, "Anti-rdbms: A list of distributed key-value stores," January 2009. [Online]. Available: http://www.metabrew.com/article/anti-rdbms-a-list-of-distributed-key-value-stores/

[jsone] (June, 2009) Json-rpc project homepage. [Online]. Available: http://json-rpc.org/

[jun09] *Trimpath Junction: Runtime Environments*, July 2009. [Online]. Available: http://trimpath.googlecode.com/svn/trunk/junction_docs/files/junction_doc_run-txt.html

[KBA+09] G. King, C. Bauer, M. R. Andersen, E. Bernard, and S. Ebersole, *Hibernate Reference Documentation*, Red Hat Middleware, LLC., June 2009, version 3.3.2.GA. [Online]. Available: http://docs.jboss.org/hibernate/stable/core/reference/en/html/session-configuration.html

[Koc09] P.-P. Koch, "Compatibility master table," July 2009. [Online]. Available: http://www.quirksmode.org/compatibility.html

[Koz09] M. Koziarski, “Riding rails: Nested model forms,” January 2009. [Online]. Available: http://weblog.rubyonrails.org/2009/1/26/nested-model-forms

[LDJ01] S. S. Laurent, E. Dumbill, and J. Johnston, *Programming Web Services with XML-RPC.* Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2001.

[Lho06a] R. Lhotka, *Expert C# 2005 Business Objects*, 2nd ed. Berkely, CA, USA: Apress, 2006.

[Lho06b] R. Lhotka, “Should validation be in the ui or in business objects,” March 2006. [Online]. Available: http://www.lhotka.net/WeBlog/ShouldValidationBeInTheUIOrInBusinessObjects.aspx

[Lin04] T. W. Ling, *Semistructured Database Design (Web Information Systems Engineering and Internet Technologie).* Santa Clara, CA, USA: Springer-Verlag TELOS, 2004.

[ME92] P. Mishra and M. H. Eich, “Join processing in relational databases,” *ACM Comput. Surv.*, vol. 24, no. 1, pp. 63–113, 1992.

[Moz05] *XML Extras*, Mozilla Foundation, November 2005. [Online]. Available: https://developer.mozilla.org/index.php?title=en/XML_Extras&revision=1

[Moz09a] *JavaScript:Tracemonkey*, Mozilla Foundation, July 2009. [Online]. Available: https://wiki.mozilla.org/JavaScript:TraceMonkey

[Moz09b] *Rhino history*, Mozilla Foundation, July 2009. [Online]. Available: http://www.mozilla.org/rhino/history.html

[nar09] (2009, July) Narwhal. Open Source Project. [Online]. Available: http://narwhaljs.org

[New06] T. Neward, “The vietnam of computer science,” June 26 2006. [Online]. Available: http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx

[Ope09] *Opera version history*, Opera Software ASA, July 2009. [Online]. Available: http://www.opera.com/docs/history/#facts

[O'R05] T. O'Reilly, “What is web 2.0,” September 2005. [Online]. Available: http://oreilly.com/web2/archive/what-is-web-20.html

[O'R06] T. O'Reilly, “Database war stories 3: Flickr,” April 2006. [Online]. Available: http://radar.oreilly.com/2006/04/database-war-stories-3-flickr.html

[Rag93] D. Raggett, *A Review of the HTML+ Document Format*, Internet Draft, 1993, in the W3C archives. [Online]. Available: http://www.w3.org/MarkUp/htmlplus_paper/htmlplus.html

[rai06] (2006, May) Rails-inspired php frameworks. [Online]. Available: http://www.h3rald.com/articles/view/rails-inspired-php-frameworks

[RG02] R. Ramakrishnan and J. Gehrke, *Database Management Systems*, 3rd ed. McGraw-Hill Science/Engineering/Math, August 2002, ch. 6.

[RR07] L. Richardson and S. Ruby, *Restful web services.* O'Reilly, 2007.

[Ser09] *ServerJS/Modules/SecurableModules*, ServerJS working group, July 2009. [Online]. Available: https://wiki.mozilla.org/index.php?title=ServerJS/Modules/SecurableModules&oldid=150641

[SMA$^+$07] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, "The end of an architectural era: (it's time for a complete rewrite)," in *VLDB '07: Proceedings of the 33rd international conference on Very large data bases.* VLDB Endowment, 2007, pp. 1150–1160.

[Sta08] M. Stachowiak, "Introducing squirrelfish extreme," September 2008. [Online]. Available: http://webkit.org/blog/214/introducing-squirrelfish-extreme/

[Sun08] *The Java EE 5 Tutorial*, Sun Microsystems, October 2008. [Online]. Available: http://java.sun.com/javaee/5/docs/tutorial/doc/bnadp.html

[Sun09] *An Overview of Project Phobos*, Sun Microsystems, July 2009. [Online]. Available: https://phobos.dev.java.net/overview.html

[THB$^+$06] D. Thomas, D. Hansson, L. Breedt, M. Clark, J. D. Davidson, J. Gehtland, and A. Schwarz, *Agile Web Development with Rails.* Pragmatic Bookshelf, 2006.

[vGB01] J. van Gurp and J. Bosch, "Design, implementation and evolution of object oriented frameworks: concepts and guidelines." *Softw., Pract. Exper.*, vol. 31, no. 3, pp. 277–300, 2001. [Online]. Available: http://www.rug.nl/informatica/onderzoek/programmas/softwareengineering/publication_files/DesignImplementationandEvolutionofObjectOrientedFrameworks.pdf

[W3C92] *HTML Tags*, W3C, November 1992. [Online]. Available: http://www.w3.org/History/19921103-hypertext/hypertext/WWW/MarkUp/Tags.html

[W3C99] *CGI: Common Gateway Interface*, W3C, October 1999. [Online]. Available: http://www.w3.org/CGI/

[Yen93] S. P. Yen, “University of minnesota gopher software licensing policy,” eMail, March 1993. [Online]. Available: http://www.nic.funet.fi/pub/vms/networking/gopher/gopher-software-licensing-policy.ancient

[YLBM08] Q. Yu, X. Liu, A. Bouguettaya, and B. Medjahed, “Deploying and managing web services: issues, solutions, and directions,” *The VLDB Journal*, vol. 17, no. 3, pp. 537–572, 2008.

[ZRN08] N. Zang, M. B. Rosson, and V. Nasser, “Mashups: who? what? why?” in *CHI ’08: CHI ’08 extended abstracts on Human factors in computing systems.* New York, NY, USA: ACM, 2008, pp. 3171–3176.

[Zyp08a] K. Zyp, “Ajax performance analysis,” April 2008. [Online]. Available: http://www.ibm.com/developerworks/web/library/wa-aj-perform/

[Zyp08b] K. Zyp, “Json referencing in dojo,” June 2008. [Online]. Available: http://www.sitepen.com/blog/2008/06/17/json-referencing-in-dojo/

Note

As explained in the preface, due to the subject of this thesis, a lot of the referenced sources are not officially published papers or books but articles from weblogs, discussion forums, knowledge bases or online documentations.

Sources that are in danger of disappearing from the web are included on the DVD-ROM attached to this thesis in HTML or PDF format.

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Dortmund, 20. Juli 2009