NTNU

Norwegian University of
Science and Technology

An approach to rapid development of modern ubiquitous Internet applications

Exploring the benefits of reusable server side components

Martin Andreas Juell
Gaute Larsen Nordhaug

Master of Science in Computer Science
Submission date: June 2011
Supervisor: John Krogstie, IDI
Co-supervisor: Bjørn Rustberggard, Inspera AS

Norwegian University of Science and Technology
Department of Computer and Information Science

PROBLEM DESCRIPTION

Popular Internet applications can grow rapidly into having millions of users. This is an important challenge for application developers, as failing to handle increasing load can disrupt an application's popularity surge and cause massive monetary losses.

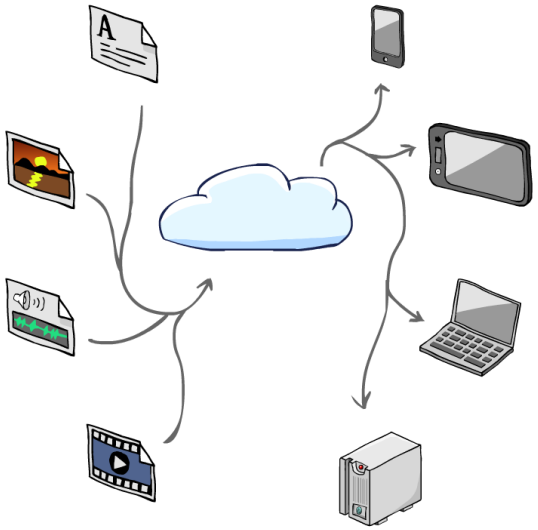
Many popular applications are *ubiquitous*, meaning they are used not only from web browsers on desktop computers, but also handheld devices, as well as other services operating on servers, connecting to the application via an Application Programming Interface (API). For traditionally designed web applications, this ubiquity is hard to achieve, as the difference in architecture creates a barrier for reusability of server side code.

The objective of the project is to design an API for rapid development of modern ubiquitous applications, and a hyperscalable backend for that API. The design should emphasize that the API is for use by a variety of clients, based on differing technologies, for different purposes.

Assignment given: Jan 2011
Supervisor: John Krogstie

AN APPROACH TO RAPID DEVELOPMENT OF MODERN UBIQUITOUS INTERNET APPLICATIONS

Exploring the benefits of reusable server side components



MARTIN ANDREAS JUELL, GAUTE LARSEN NORDHAUG

IDI
Norges Teknisk-Naturvitenskapelige Universitet
June 2011

Martin Andreas Juell, Gaute Larsen Nordhaug: *An approach to rapid development of modern ubiquitous Internet applications,* Exploring the benefits of reusable server side components, © June 2011

ABSTRACT

Popular Internet applications can grow rapidly into having millions of users. This is an important challenge for application developers, as failing to handle increasing load can disrupt an application's popularity surge and cause massive monetary losses.

Many popular applications are *ubiquitous*, meaning they are used not only from web browsers on desktop computers, but also handheld devices, as well as other services operating on servers, connecting to the application via an Application Programming Interface (API). For traditionally designed web applications, this ubiquity is hard to achieve, as the difference in architecture creates a barrier for reusability of server side code.

Using a design science research methology, this report details an approach to solving scalability issues and greatly improving reusability and development speed for modern ubiquitous internet applications. The crux of the approach is a bare-essentials data access and user management API, whose implementation is intended to serve as the entire server side of the application.

For applications that can cope with its reduced feature set, it has several major advantages. API implementations are interchangeable, eliminating vendor lock, and also completely reusable across applications, saving development effort. Presentation and application logic is shifted to the client side, reducing server strain, and the API is easily implemented with a modern, hyperscalable data store in a cloud environment, providing great elasticity and scalability.

The functionality of the API is derived from an analysis of target applications, and the approach is evaluated through the development of a prototype, a blog application with clients for several platforms. The prototype development process reveals some architectural and practical limitations to the design, but also showcases the power of reusable components when those components are readily available.

The approach presented here is not ideal for all types of applications. However, when applicable, it helps developers save time and overcome these important challenges in application development.

PREFACE

This report is a documentation of the project work performed as the Master's thesis in Computer Science by Gaute Larsen Nordhaug and Martin Andreas Juell in the spring of 2011. The project work is carried out in the last semester of the five-year integrated M.Sc. program in Computer Science at the Norwegian University of Technology and Science (NTNU). The scope of the project is 30 credits for each author, equivalent to one semester of full course load.

The assignment was defined in cooperation with Inspera AS and supervisor John Krogstie at the department of Computer and Information Science (IDI). While the assignment was carried out to solve some pressing problems for Inspera, it was designed in such a way that others too may benefit from our efforts and experiences.

We would like to thank our supervisors at Inspera, Sondre Bjørnebekk, Bjørn Rustberggard, Naimdjon Takhirov and John Arne Skjervold Pedersen, as well as IDI supervisor John Krogstie for their invaluable feedback and support throughout the design and writing process.

Trondheim, Norway, June 2011
Martin Andreas Juell and Gaute Larsen Nordhaug

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ABBREVIATIONS

2PC	Two-Phase Commit
ACID	Atomicity, Consistency, Isolation, Durability
ACL	Access Control List
AMF	Adobe Message Format
API	Application Programming Interface
AWS	Amazon Web Services
AZ	Availability Zone
BASE	Basically Available, Soft state, Eventually consistent
CAP	Consistency, Availability, Partition Tolerance
CMS	Content Management System
CPU	Central Processing Unit
CRUD	Create, Read, Update, Delete
CTO	Chief Technical Officer
DaaS	Database as a Service
DBMS	Database Management System
DNS	Domain Name System
EC2	Elastic Compute Cloud
HATEOAS	Hypermedia As The Engine Of Application State
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IaaS	Infrastructure as a Service
ICS	Inspera Content Server
IDI	Institutt for Datateknikk og Informatikk/ Department of Computer Science

JSON	JavaScript Object Notation
NTNU	Norsk Teknisk-Naturvitenskapelig Universitet/ Norwegian University of Technology and Science
NoSQL	Not Only SQL
PaaS	Platform as a Service
PDA	Personal Digital Assistant
RBAC	Role Based Access Control Model
REST	Representational State Transfer
RFC	Request For Comments
RPC	Remote Procedure Call
SaaS	Software as a Service
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
USD	United States Dollars
W3C	World Wide Web Consortium
WebDAV	Web-based Distributed Authoring and Versioning
XML	Extensible Markup Language

Part I

INTRODUCTION

1 BACKGROUND AND OBJECTIVE

1.1 MASSIVELY SCALABLE UBIQUITOUS APPLICATIONS

Many Internet applications have experienced rampant growth over the last few years. Social media giant Facebook boasts over 500 million active users[4], and has had 2011 valuations ranging from 50 billion[5] to 124 billion USD[6]. Microblogging service Twitter is expected to pass 200 million registered users in 2011, and has been valued at 8-10 billion USD[7].

These two applications, and many like them, represent a new generation of Internet applications in that they are not only available in web browsers on desktop computers. Their extensive Application Programming Interfaces (APIs) allow access to native applications running on mobile devices, and serve data to other services running on servers.

These applications and services can be developed by the original application's developers, or by third parties. Twitter, for instance, is easily integrated in news websites, games, and even household appliances[8], adding value both to the third party and to Twitter itself. Over 2.5 million websites have integrated with Facebook[4].

This report will use the term *ubiquitous application* to mean an application that is accessible from a variety of devices and platforms, possibly for a variety of purposes.

Web vs other clients

Traditional web applications operate by generating HTML on the server side that is displayed in a browser. The application logic as well as the presentation (in the sense of generating HTML for the browser to display) are handled on the server side, see Figure 1.1 . Mobile applications and third-party services, however, typically operate on raw data in a neutral format, and handle presentation themselves, as illustrated in Figure 1.2. For a ubiquitous application, the website requires a disproportionate amount of special attention, and the code used to generate HTML has no value to the other clients, as it is a very impractical format for them to consume.

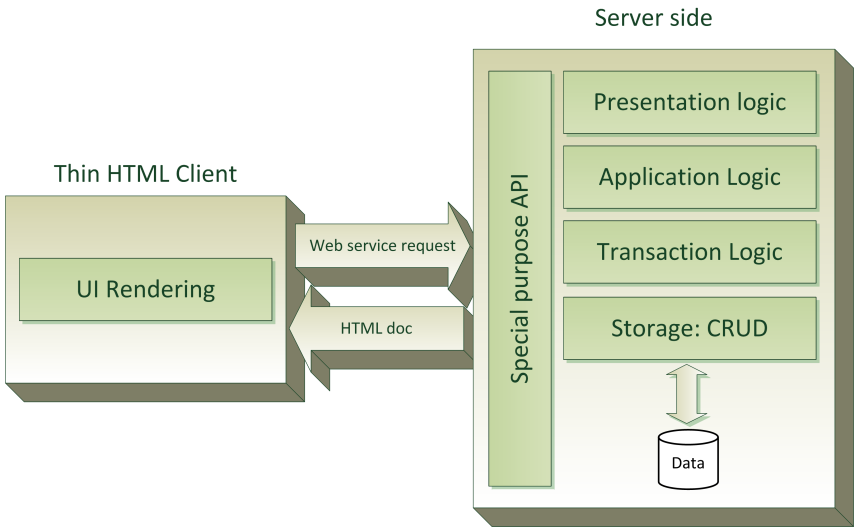


Figure 1.1: Traditional web application architecture

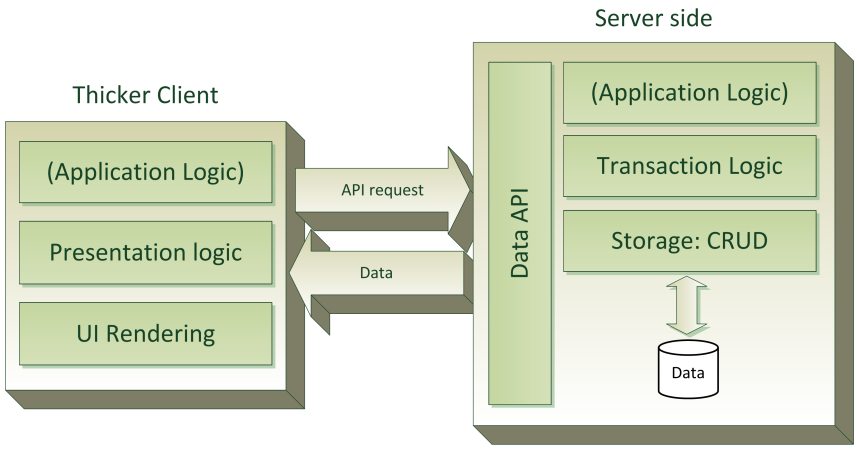


Figure 1.2: Typcial architecture for mobile client or consuming service

Scaling and performance

As an Internet application becomes popular, its ability to handle increasing amounts of load will be absolutely critical to its future success. Early social networking pioneer Friendster has become infamous for the website's inability to handle traffic as load increased, which ultimately led to it being far less successful than stakeholders had hoped[9][10]. Google measured that an added latency of 100 milliseconds on web searches is enough to cause a statistically significant drop in search traffic [11] .

1.2 CASE: INSPERA

Inspera AS[1] is a Norwegian technology company that makes cross-media publishing tools geared towards education. All Inspera software is based on Inspera Content Server (ICS), a flexible XML-based content management system. However, an interview with Inspera's Chief Technical Officer (CTO) reveals that there are some caveats to using ICS to build next-generation Internet applications. ICS is designed to run on traditional, manually configured web servers, which are currently hosted by an expensive hosting provider. It also depends on expensive per-CPU licenses such as Oracle Database and Oracle Coherence. In addition, ICS operates in the traditional fashion shown in Figure 1.1, which makes prototyping and developing ubiquitous applications slower and more costly than it needs to be.

If an application made by Inspera experiences massive popularity growth, ICS will not be able to scale to meet demand, at least not without Inspera suffering heavily from license and operational costs.

Creaza Video

Creaza Video [2] is one of Inspera's flagship applications. It features a sophisticated timeline-based video editor that runs within the user's web browser. The user can upload their footage and audio from a variety of devices and create high quality productions without having to install any software on their com-

1 http://www.inspera.no
2 http://www.creaza.com

puter. Completed productions can be downloaded or exported to services like YouTube[3] or Facebook[4] for sharing.

Creaza was originally a product designed for licensing in the educational market, but a new version is scheduled for launch to the general consumer public in August 2011. Creaza is based on ICS, which means that scalability issues are likely to occur if Creaza is well received by consumers.

1.3 RESEARCH APPROACH AND OBJECTIVES

1.3.1 *Research questions*

This project aims to examine the problems described in Section 1.1, using ICS and Creaza as a study case. A design science methodology (Section 1.3.4) is used, with the intention of producing:

1. A contribution to solving the problems in the general case
2. A solution to help Inspera specifically in overcoming these problems
    a) In Creaza
    b) In future applications.

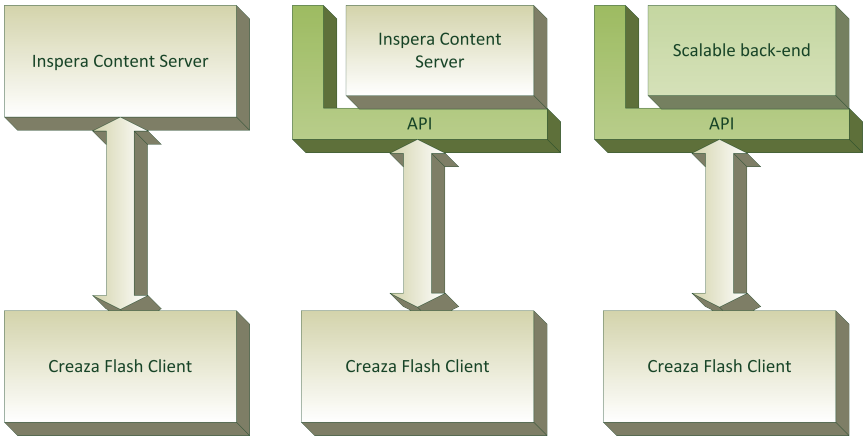
The objective of the project is to outline an approach to Internet application development which:

1. Allows reuse of components
    a) across application clients
    b) across applications.
2. Ensures scalability at low monetary cost.
3. Can be used for new applications as well as existing ones, such as Creaza.

1.3.2 *Solution hypothesis*

The backing hypothesis in this project is that it is possible to solve these problems by designing a new general API and a scalable implementation. The API should contain a core set of functionality common to a class of applications, and its implementation should be reusable not only across different application

3 http://www.youtube.com
4 http://www.facebook.com



(a) Original situation

(b) All communication with ICS through general-purpose API with ICS implementation

(c) Implementation substituted for simpler, more scalable back-end.

Figure 1.3: Exchange of back-end for Creaza.

clients, but across applications as well. Reusable cross-platform core functionality allows rapid prototyping of applications, and reduces development and maintenance costs. The API should allow for multiple implementations, so that new advances to server side development can be adopted without having to modify applications, and so that Inspera can create a transitional implementation by using bindings to ICS. This will allow Creaza and other legacy applications to be ported onto the new architecture in a three-step process shown in Figure 1.3.

1.3.3 *Solution approach*

First, the functionality to be supported by the API must be determined. The design should emphasize that the API is for use by a variety of clients, based on differing technologies, for different purposes. In addition, it is a goal for the API to differentiate from competing offerings in a way that makes it attractive to a fitting niche of developers. As research for both these points, a few popular APIs that are similar to what Inspera are planning should be examined, especially with regard to what they do to attract developers in terms of support, documentation and community management.

Given API functionality and a few demo applications, the second part of the project should look into how to design a new,

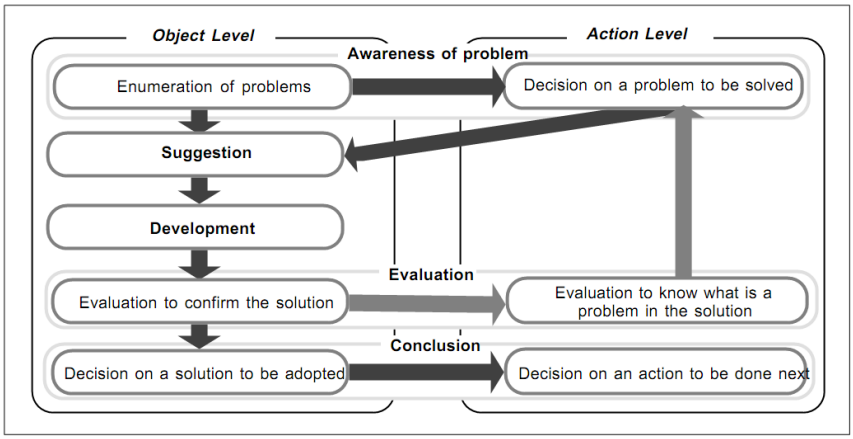


Figure 1.4: Design science workflow as described in[1]

lighter, scalable back-end. It should not depend on any licenses payable per CPU, and the source code should be significantly smaller in size and thus more maintainable than ICS.

Scope

It is not the goal of this project to implement the new platform in code, but to perform a thorough analysis upon which Inspera can base the development, which is planned to start in the summer of 2011. The main target is to build a proof-of-concept prototype of a platform based on the API, and evaluate its viability.

1.3.4 *Research method: Design science*

Design science is a research approach in which problems are solved by designing, creating, and evaluating artifacts. Its process, outlined in Figure 1.4 starts with the awareness of a problem, possibly selected from a set of possible problems to solve. Through application of previous knowledge and existing theory, potential solutions are suggested, developed, and evaluated according to a set of guidelines, presented in Table 1.1. Research based on design in this manner produces four main types of output, as first defined by March and Smith in [12]:

CONSTRUCTS are the conceptual vocabulary of the domain in which the problem and solution exist. These may not be clearly defined at first, but are refined as work progresses.

Guideline	Description
Guideline 1: Design as an Artifact	Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.
Guideline 2: Problem Relevance	The objective of design-science research is to develop technology-based solutions to important and relevant business problems.
Guideline 3: Design Evaluation	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.
Guideline 4: Research Contributions	Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.
Guideline 5: Research Rigor	Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.
Guideline 6: Design as a Search Process	The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.
Guideline 7: Communication of Research	Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.

Table 1.1: Design science guidelines [2]

MODELS are sets of statements expressing relationships among constructs. A model proposes a description of how the domain, or aspects of it, work.

METHODS are defined ways to perform a task. Since design science is conducted primarily for utility, methods can not only be problem solving tools, but also the object of research. In these cases, an improved method is often the result.

INSTANTIATIONS are the final product of a design science effort. An instantiation operationalizes constructs, models, and methods. It is the realization of the artifact in an environment. Sometimes, an instantiation may precede a complete articulation of the constructs, models, and methods that it embodies, in the same way that flying aircraft were constructed before flight was fully understood [13]. This understanding arose with the help of the instantiations, and is unlikely to have occurred without them.

Table 1.1 lists the design science guidelines followed throughout this project. This project was conducted to solve a pressing business problem: Scaling issues in applications, critical to profitability (G2). The main artifact produced in this project(G1) is the Moraxus API and sample implementation. The methods and models this design entails, as well as details of the instantiation prototype are described in chapters 5-6. The development of the artifact was carried out in an iterative fashion, as encouraged by G6. The work is analytically evaluated (G3, G5) in chapter 7.

In addition to solving Inspera's scalability problems, the method for solving them and the API can contribute to solving similar problems for other organizations (G4), or at the very least, add to the base of knowledge in this domain. This report has been composed to facilitate being read by audiences of varying previous knowledge of the concepts discussed (G7).

1.4 THESIS OUTLINE

Chapters 2 and 3 discuss underlying theoretical concepts in scalability and API design, respectively. Chapter 4 describes the requirements determined for the solution, while the solution itself is presented in chapter 5. Chapter 6 discusses the prototype that was implemented to evaluate the design, the evaluation

itself follows in chapter 7. Chapter 8 concludes the report and outlines future work.

The report has three appendices. Appendix A contains a more in-depth discussion of Platform as a Service (PaaS) providers discussed as alternative approaches in Section 7.8 . Appendix B documents the workings of the authentication mechanisms used in the API. Finally, Appendix C contains the API documentation.

Part II

THEORETICAL BACKGROUND

2 ENSURING SCALABILITY

Structural scalability is the ability of a system to expand in a chosen dimension without major modifications to its architecture. *Load scalability* is the ability of a system to perform gracefully as the offered traffic increases [14]. If an Internet application experiences vast growth over a relatively short time, it must be scalable by both of these metrics. This chapter discusses how one can ensure scalability in an Internet application by carefully designing the hardware and software architecture on which it is based.

Vertical vs horizontal scaling

Twitter had a 1382% growth in members from Feb 08 to Feb 09[15]. Accommodating extreme growth requires a carefully crafted software architecture and hardware setup on the server side. A *vertical scaling* approach, in which existing nodes are made more powerful in order to accommodate growth, is expensive due to the cost of high performance servers, and 1382% more powerful hardware can be hard to come by. The alternative is *horizontal scaling*; adding more nodes and sharing load among them.

2.1 CLOUD COMPUTING

Since the introduction of Amazon *Elastic Compute Cloud (EC2)*[1] in 2006 [16], cloud environments have become widespread due to their horizontal scaling capabilities. A cloud environment, as defined in the US National Institute of Standards and Technology working definition [17], has the following characteristics:

- *On-demand self-service*. A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service's provider. Providers have broad network access. Capabilities are available over the network and accessed through standard mechanisms

1 http://aws.amazon.com/ec2/

that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, laptops, and Personal Digital Assistants (PDAs)).

- *Resource pooling.* The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources, but may be able to specify location at a higher level of abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, network bandwidth, and virtual machines.

- *Rapid elasticity.* Capabilities can be rapidly and elastically provisioned, in some cases automatically, to quickly scale out, and rapidly released to quickly scale in. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be purchased in any quantity at any time.

- *Measured Service.* Cloud systems automatically control and optimize resource use by leveraging a metering capability[2] at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.

The definition also divides cloud services into three service models:

- *Software as a Service (SaaS).* The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through a thin client interface such as a web browser (e.g., web-based email). The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings.

2 Typically through a pay-per-use business model

- *Platform as a Service (PaaS).* The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly application hosting environment configurations.

- *Infrastructure as a Service (IaaS).* The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, deployed applications, and possibly limited control of select networking components (e.g., host firewalls).

For development of new software, *PaaS* and *IaaS* are the two relevant categories to consider, and the developer must carefully consider the needs of the application being developed in order to choose between them.

IaaS:	*PaaS*
Allows complete control over what software technologies to use.	Restricts languages and tools to those supplied by the provider.
Requires manual administration of server instances, typically charged by the instance hour.	No such administration necessary, payment typically by the CPU hour.
Special care must be taken in software design to ensure resilience against node or network failures (Discussed in Section 2.2).	Failure resilience is built into platform and is the responsibility of the provider.

2.1.1 *Benefits*

The most important benefits to using cloud services are:

- *Flexibility.* A system's capacity can be changed on a timescale of minutes or seconds to accommodate increasing or decreasing load. This process is automated by the provider in *PaaS* environments, and can be automated by the customer in *IaaS*.
- *Scalability.* Providers have vast amount of computing, network and storage resources available.
- *Simplicity.* No management of physical servers is necessary. Failover procedures in the cloud setup protect against system outages from hardware failure.
- *Cost.* In addition to the savings incurred through the previous points, the pay-per-use business model combined with the savings of a multi-tenant setup provide for favorable pricing conditions. Deelman et al. [18] conclude that cloud computing offers cost-effective solutions for data-intensive applications with low *CPU* costs, such as the platform presented in chapter 5.

For a small company like Inspera, these advantages are currently only attainable through purchasing cloud services from a provider. Since the goals for the platform developed in this project include reducing time to market for new application ideas, as well as supporting near-infinite scalability for an application gaining traction, it is essential that the platform is able to run on a cloud environment. However, since different applications have different requirements, but ideally should be able to use the same API, it is beneficial if the API is not tied to one provider, or even to one of the *PaaS* or *IaaS* categories.

2.1.2 *Disadvantages*

Even though the use of cloud services can provide huge advantages, there are some disadvantages as well. One of the most criticized aspects of cloud computing is the possible lack of security and privacy. When storing information on remote servers, other people will have access to it. And even though the cloud computing service can be just as secure as other solutions, the volume of data stored on the most prominent cloud service providers make them a more interesting target for hackers. An example of this would be the hacking of the PlayStation Network in April 2011, where millions of users got their personal information, and possibly credit card information, stolen [19].

Another source of concern is the lack of control over the stability of the services. If a developer's cloud provider has trouble, his application will be down until they fix it, there is nothing he can do.

2.2 DESIGNING A CLOUD BACK-END FOR SCALABILITY AND RESILIENCE

This section describes some of the challenges that come with designing systems to run on a large number of commodity servers.

2.2.1 *General resilience*

In a cloud environment, or any environment based on lots of commodity computers, nodes may fail, as may the network connecting them. If all traffic to the application passes through a single node, such as a load balancer or database master, this node is a *single point of failure:* If this node goes down for some reason, as can happen, the entire application will suffer. The ideal solution is to develop an architecture with no such single points of failure. If this is not possible, an alternative is to use one or more extra nodes that are kept exactly the same as the first node through active replication, where only the master node is used, and the job of the secondary node(s) is simply to monitor the main node and take its place in the event of a failure. An example of such a setup is the database master-slave setup in Figure 2.1.

Rambo architectures and the Chaos Monkey

In [20], John Ciancutti of the on demand video streaming company Netflix [3] describes some important aspects of their system's fault-tolerance abilities. He refers to their Amazon Web Services (AWS) architecture as their "Rambo architecture", meaning that each subsystem should be able to work on its own, even if all the systems it relies on goes down.

> If our recommendations system is down, we degrade the quality of our responses to our customers, but we still respond. We'll show popular titles instead

3 http://www.netflix.com

of personalized picks. If our search system is intolerably slow, streaming should still work perfectly fine.

One of the first systems our engineers built in AWS is called the Chaos Monkey. The Chaos Monkey's job is to randomly kill instances and services within our architecture. If we aren't constantly testing our ability to succeed despite failure, then it isn't likely to work when it matters most – in the event of an unexpected outage.

Regions and availability zones (AZs)

Cloud providers like Amazon often offer the opportunity to distribute a customer's server instances across fault-isolated boundaries. Amazon, for instance, operates a number of *regions* spread across the world, with a number of Availability Zones (AZs) within each region. Data can be transmitted between availability zones at rates substantially lower than those charged for ordinary Internet traffic. One can think of each zone as a separate data center[4], and if one AZ experiences some malfunction, server instances in other AZs should still be fine, see Figure 2.1.

However, just having different zones may not be enough in the event of a major outage, like the one Amazon experienced in April 2011, when the servers in an Availability Zone went down for multiple days with the result that hundreds of application and websites went down [21].

2.2.2 *Databases*

Most conventional Database Management Systems (DBMSs) give a set of guarantees known as ACID; Atomicity, Consistency, Isolation and Durability.

ATOMICITY means that database transactions are either completed in full or, if they fail or are aborted, fail completely and leave the database unchanged.

CONSISTENCY ensures that any transaction the database performs takes the it from one consistent state to another.

4 Amazon stresses that it does not guarantee any physical proximity or separation between AZs

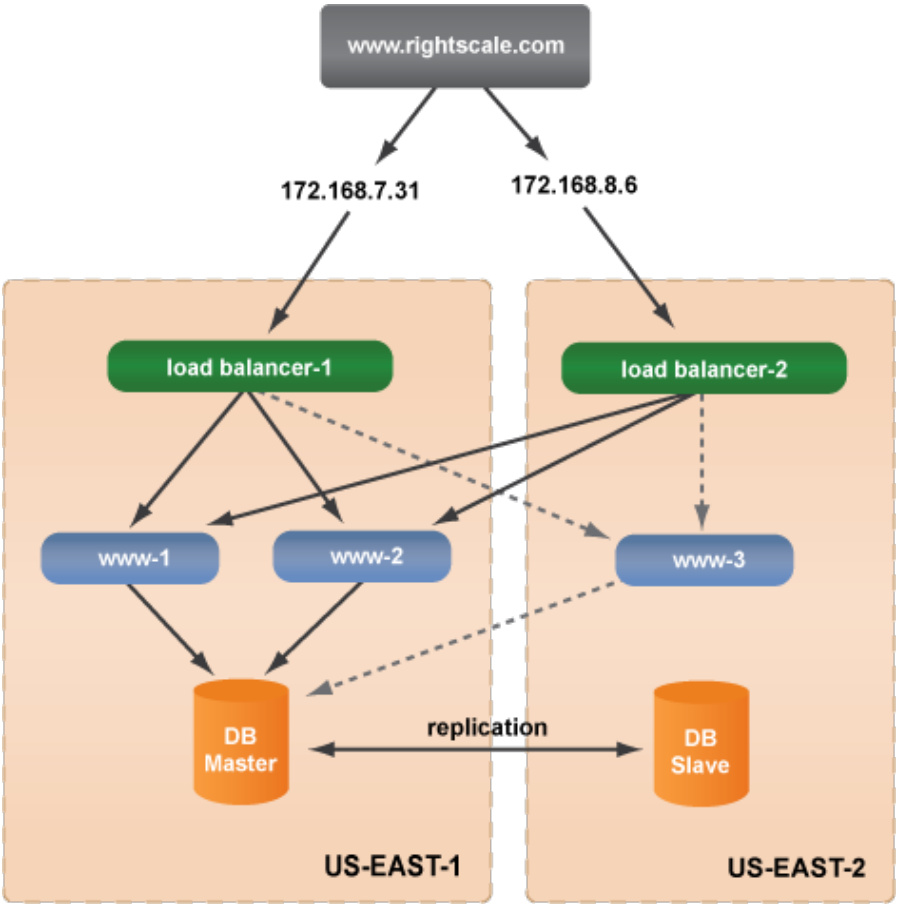


Figure 2.1: Redundant server setup over 2 AZs.

If US-EAST-1 fails, DNS will redirect all traffic to load-balancer-2, the DB slave can be promoted to master, more instances can be spawned, and service can resume as normal. This process can be automated.

ISOLATION ensures that transactions do not interfere with each other (e.g. data in an unfinished transaction is not visible to other transactions)

DURABILITY ensures that committed transactions endure any kind of system failure.

Database vendors long ago recognized the need for partitioning databases and introduced a technique known as Two-Phase Commit (2PC) for providing ACID guarantees across multiple database instances. The protocol is broken into two phases: First, the transaction coordinator asks each database involved to pre-commit the operation and indicate whether commit is possible. If all databases agree the commit can proceed, then phase 2 begins. The transaction coordinator asks each database to commit the data. If any database vetoes the commit, then all databases are asked to roll back their portions of the transaction. [22]

2PC ensures that all copies of data are consistent, even if they are partitioned. The problem with this, however, is that the system is only able to accept transactions if all copies of the transaction's data are available, decreasing the overall availability of the system and partly defeating the purpose of the replication.

The CAP theorem

The CAP theorem[23] states that it is impossible for a distributed system to achieve both consistency, availability and partition tolerance at the same time.

CONSISTENCY All records should be the same in all replicas.

AVAILABILITY All replicas can accept updates or inserts.

PARTITION TOLERANCE The system still functions when distributed replicas cannot communicate due to network or node failures.

BASE

ACID database systems using 2PC achieve consistency across partitioned, but sacrifice availability. Another option are so called BASE databases : *Basically Available, Soft state, Eventually consistent*. These data stores remain available despite partial failures, but do not guarantee consistency across all replicas at all times.

These data stores typically control concurrency through locks on small units of data, or multi-versioning of items, with older versions being discarded in favor of new ones as changes propagate across replicas[24].

Scalable ACID

Thomson and Abadi [25][26] argue that NoSQL data stores[5] are a lazy solution, and that by tightening ACID's Isolation argument so that transactions are carried out in a deterministic order, one can escape the need for Two-Phase Commit. Cattell [24] analyzes and compares a total of 23 scalable data stores, 6 of which provide ACID or ACID-like guarantees. Relational SQL databases have been dominant on the database scene for decades, and proponents argue that new SQL systems can perform as well as NoSQL alternatives, and that there is no reason to switch. NoSQL enthusiasts, on the other hand, prefer the flexible schemas and proven scaling of those systems.

Cassandra[6]

In the interest of brevity, this report will not discuss all the tens of different options for a suitable data store in an *IaaS* environment. Bearing in mind both the specifics of this project and the general Internet application case, Apache Cassandra was selected as the data store to use in further experimentation. The reasons for this are:

- Cassandra is truly decentralized. Nodes are organized in a ring, with no node being the master or more significant than others. This eliminates bottlenecks and ensures that there is no single point of failure.
  - New nodes can be added to the ring simply by starting Cassandra on a machine/instance and pointing it to a node currently in the ring. Cassandra handles redistribution and replication itself.
- Cassandra allows flexibility along the CAP spectrum. For any read and write operation, the developer can choose how many replicas to read or write before the call should return.

5 Not Only SQL data stores: common denominator for non-ACID data stores
6 http://cassandra.apache.org/

  - Writing one or two replicas allows operations that need to be fast, but not consistent, to be so.
  - Quorum reads and writes (reading or writing N/2 + 1 replicas, where N is the total number of replicas) are slower, but a quorum write will always be read by a subsequent quorum read.
  - The number of replicas to keep of any data is adjustable.

- Cassandra has a powerful, yet flexible, data model, based on *columns*, *supercolumns*, and *column families*. Columns consist of a name and a value, both binary types, as well as a 64-bit timestamp used for version control. The larger structures are arrays and containers of columns. Details on the data model are available in [27].

3

BACKGROUND ON API DESIGN

This chapter describes some of the greatest challenges of API design, and outlines the guidelines followed by the authors in order to overcome them. It also examines some successful APIs and published guidelines to determine industry best practices. Section 3.1 draws heavily from [28].

The value of a quality API

For the developer, APIs are an investment in the sense that they take a lot of time to develop. The rewards can be proportionally great when customers in turn invest in the API by buying licenses, learning the API, and writing software for it. An API that provides great value will draw customers, and if it is well written and able to evolve, the customers will stay. On the other hand, a poorly written API will lead to frustrated customers, high support costs, and lost business. To add to the risk of investing in developing an API, once an API has been released to the public and software has been written for it, it cannot be changed in any way that can break existing software written for the API. In a public API, functionality can be amended, but never changed or removed.

3.1 GENERAL DESIGN PRINCIPLES

In[28], Joshua Bloch outlines a few key goals to strive towards in order for an API to succeed. It should be:

- Easy to learn
- Easy to use, even without documentation
- Hard to misuse
- Easy to read and maintain code that uses it
- Sufficiently powerful to satisfy requirements
- Easy to extend
- Appropriate to audience

Some of the goals, such as ease of adaptation and use, are fairly obvious goals, while others are more subtle. Bloch stresses that sufficiently powerful does not mean that the API has to be powerful in an absolute sense - it should only be *powerful enough*, and *as small as possible, but no smaller.* More functionality can always be added later, and the API should be built in such a way that it does a few select things very well, while leaving room for expansion. Being appropriate to audience is also somewhat of a subtle point. Bloch's example is that an API for stock analysts should differ from an API for physicists, because their terminology and mindset are different, as are the problems they are looking to solve.

One of the most important ways to reach many of the above goals is through proper naming. A good name will tell the user what a component does, and makes it easy for him to write his code without reading more documentation than necessary. It also makes code written for the API easier to read. Ideally, every component, method and parameter should have a good name, and the code should read like prose, see Listing 3.1 . If it is hard to come up with a good name for a component, then chances are the component violates the principle of doing one thing well.

Listing 3.1: Code that reads like prose

```
if (car.speed() > 2 * SPEED_LIMIT){
     generateAlert("Watch out for cops!")
}
```

Listing 3.2: A boilerplate method for the W3C XML API for Java.

```
private static final void writeDoc(Document doc,
  OutputStream out)throws IOException{
  try {
    TransformerFactory tf = TransformerFactory.newInstance()
        ;
    Transformer t = tf.newTransformer();
    t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM,
      doc.getDoctype().getSystemId());
    t.transform(new DOMSource(doc), new StreamResult(out));
  } catch(TransformerException e) {
    throw new AssertionError(e);  // Can't happen!
  }
}
```

In order not to restrict one's freedom to change implementations in future releases, it is important that the API does not leak

any more of its internals than it has to. Occasionally, this will go hand in hand with another important aspect: Not making the client do anything a module could have done for it. Listing 3.2 is a method from a client using the W3C XML API for Java. Had this method been a part of the API, the client programmer would not have had to learn how to create and use a `Transformer` object, and the implementors would have been free to change the implementation to something different, possibly moving away from the `Transformer` design for this task.

Finally, Bloch emphasizes the importance of following platform customs. An API written for one platform should not necessarily be ported to another by translating the code line by line. Instead, the functionality should be reimplemented in a way that is natural to use for a developer on the target platform. The API developed in this project needs to be accessible from a variety of clients including web browsers, mobile applications, and other web servers using HTTP. One of the platform conventions one might follow when developing an HTTP-based API is Representational State Transfer (REST), discussed further in Section 3.2.2.

3.2 INDUSTRY STANDARDS FOR INTERNET APIS

This section provides a brief discussion on the two most popular approaches for designing web services: SOAP and REST.

3.2.1 *SOAP and RPC*

Simple Object Access Protocol (SOAP)[1] is a protocol for passing XML formatted messages between applications or web services. It is not tied to any particular transport protocol, though HTTP is a popular choice, nor is it tied to any specific operating system or programming language. The SOAP protocol specifies the message format, message exchange patterns as well as several other aspects of the message passing. A SOAP message itself consists of three parts: an envelope, a header and a body. See Listing 3.3 for an example of a simple SOAP message sent over HTTP. An advantage of using HTTP as the transport protocol is that it is not normally filtered by firewalls.

1 http://www.w3.org/TR/soap12-part1/

Listing 3.3: A SOAP message sent over HTTP

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 299

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-
    envelope">
  <soap:Header>
  </soap:Header>
  <soap:Body>
    <m:GetStockPrice xmlns:m="http://www.example.org/stock">
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```

Remote Procedure Call (RPC) can also be used as the transport protocol for SOAP. An RPC allows a program running on one computer to call a function or program running on another computer. The advantage of using RPC in combination with SOAP is that is allows both parts of the interaction to initiate a connection, while when relying on HTTP only the client can do so.

3.2.2 *RESTful Web Services*

RESTful web services are web services that conform to the constraints of the Representational State Transfer (REST) paradigm. These constraints were first set forth in chapter 5 of Roy Fielding's dissertation [29], and include:

- Client-server architecture
- Stateless communication between client and server
- Uniform interface even for different clients
- Allowing for cache and/or a layered system

Fielding also emphasizes the notion of *resources*, *identifiers* and *representations*.

> The key abstraction of information in REST is a resource. Any information that can be named can be a resource: a document or image, a temporal service (e.g. "today's weather in Los Angeles"), a collection of other resources, a non-virtual object (e.g. a person), and so on. In other words, any concept that might be the target of an author's hypertext reference must fit within the definition of a resource. A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time. [...]
>
> REST uses a resource identifier to identify the particular resource involved in an interaction between components. REST connectors provide a generic interface for accessing and manipulating the value set of a resource, regardless of how the membership function is defined or the type of software that is handling the request. The naming authority that assigned the resource identifier, making it possible to reference the resource, is responsible for maintaining the semantic validity of the mapping over time. [...]
>
> A representation consists of data, metadata describing the data, and, on occasion, metadata to describe the metadata (usually for the purpose of verifying message integrity). Metadata is in the form of name-value pairs, where the name corresponds to a standard that defines the value's structure and semantics. Response messages may include both representation metadata and resource metadata: information about the resource that is not specific to the supplied representation.

In a RESTful web service, these ideas are implemented using the HTTP protocol. Resources are identifiable through their URI, and manipulated through HTTP requests. Typically, no special methods are exposed, instead clients express their intent using the methods described in RFC 2616[30], the most important of which are `GET`, `PUT`, `POST`, and `DELETE`. The use of these methods is illustrated in Table 3.1.

To conform to the RFC, the `GET`, `PUT`, and `DELETE` methods must be *idempotent*, i.e. a sequence of identical calls to the method should be equivalent to one call. The `GET` method also has the constraint that it must be *safe*, i.e. that its calls should not have user-noticeable side effects.

Resource	Collection URI, such as `http://site.com/res/`	Element URI, such as `http://site.com/res/14`
`GET`	List the URIs and perhaps other details of the collection's members.	Retrieve a representation of the addressed member of the collection, expressed in an appropriate Internet media type.
`PUT`	Replace the entire collection with another collection.	Replace the addressed member of the collection, or if it doesn't exist, create it.
`POST`	Create a new entry in the collection. The new entry's URL is assigned automatically and is usually returned by the operation.	Treat the addressed member as a collection in its own right and create a new entry in it.
`DELETE`	Delete the entire collection.	Delete the addressed member of the collection.

Table 3.1: Typical semantic interpretation of HTTP methods in a RESTful web service[3]

Another important aspect of REST is called *Hypermedia As The Engine Of Application State (HATEOAS)*. In [31], Fielding describes this as follows:

> A REST API must not define fixed resource names or hierarchies (an obvious coupling of client and server). Servers must have the freedom to control their own namespace. Instead, allow servers to instruct clients on how to construct appropriate URIs, such as is done in HTML forms and URI templates, by defining those instructions within media types and link relations. [...]
>
> A REST API should be entered with no prior knowledge beyond the initial URI (bookmark) and set of standardized media types that are appropriate for the intended audience (i.e., expected to be understood by any client that might use the API).

Martin Fowler further elaborates on this in [32].

3.2.3 *Comparison/discussion*

While both SOAP and REST are viable alternatives, there are definitely some advantages and disadvantages tied to each technology. For one, the HTTP request operations `GET`, `POST`, `PUT` and `DELETE` map to the basic Create, Read, Update, Delete (CRUD) operations used in data storage, which makes REST a nice choice for applications that mostly deal with those kinds of operations.

The REST principle is also regarded as being easier to understand than the SOAP protocol, and it also allows for arbitrary data formats, while SOAP is tied to XML. On the other hand, SOAP has better support for security, with the use of `WS-Security` (a SOAP extension that focuses on assuring integrity and confidentiality), atomic transaction and reliable messaging.

These enterprise level security features were not deemed necessary for this project. However, having the opportunity to use alternative data formats like JavaScript Object Notation (JSON) and Adobe Message Format (AMF) for JavaScript and Flash applications was considered a decisive advantage, as was the ability to more easily access the API through a web browser. REST is also a proven paradigm with widespread use in public APIs (Google for instance, abandoned their SOAP API in 2006 [2]) . For

2 http://code.google.com/apis/soapsearch/

these reasons, the API developed in this project is based on the REST principles.

3.3 EXAMPLE APIS

This section examines a selection of interesting aspects of various existing APIs, some of which are present in the API designed in this project.

3.3.1 *HiFi API*

HiFi [3] is a Content Management System (CMS)/web publishing engine built upon and around a flexible API. This API is relevant for this project for multiple reasons. First, HiFi designed its API first, before building the application itself on top of the API [33]. Second, they use a templating language to be able to easily create powerful and complex queries with support for recursion and object relationships. Listings 3.4 and 3.5 show a couple of example queries.

Listing 3.4: A simple query, getting all post objects.

```
{"type":"post", "orderBy":"-publishedAt"}
```

Listing 3.5: A slightly more complex query.

```
{"type":"page","orderBy":"-publishedAt","parent":
{"type":"feed","url":"/blog"}}
```

The combination of powerful queries and a system that is built on top of its own API makes it easy to create complex applications running on the client side, which is something this project would like tho achieve.

3.3.2 *Twitter API*

The Twitter API [4] is an example of a hugely successful REST API, with more than 75% of Twitter's traffic coming from outside twitter.com[34], and a majority of the status updates originating from third party applications. It is a relatively traditional REST API with the exception that `POST` requests both

3 http://www.gethifi.com/
4 http://dev.twitter.com/doc

can create, edit and delete data. A `GET` request to `http://api.twitter.com/version/statuses/show/:id.format` will for instance return the status update with the given ID in the selected format, while a `POST` request to `http://api.twitter.com/version/statuses/update.format` will create a new status based on the request parameters.

Twitter shows how important an API can be for the process of creating an active ecosystem surrounding the platform, with third party developers creating additional services on top of it.

3.3.3 *SnapBill API*

The SnapBill API [35] has chosen to deviate from the REST standard on a few key points, while maintaining the spirit of its hypermedia argument in a possibly more practical way. In the SnapBill API, only the `GET` and `POST` verbs are used, and they do not have the meanings specified in the Request For Comments (RFC). The reason `PUT` and `DELETE` are omitted is that there is no easy way to make these calls from a web browser apart from executing a script that does so. The `GET` function is not used to retrieve resources, but rather to discover the API itself. Executing a `GET` to the application entry point (e.g. `https://api.snapbill.com/v1`) returns a HTML page containing hyperlinks to similar pages representing the main sections of the API. The pages returned from clicking these links are similar, with links to API methods. A `GET` request (i.e. clicking on a link) to an API method returns a HTML page containing a form with a field for every argument the method can take, and a button to `POST` to the same Uniform Resource Identifier (URI). All method calls are made using `POST`, regardless of whether or not they contain data or have side effects.

Benefits and drawbacks

Deviating from RESTful use of HTTP verbs also deviates from the vision of one standardized way to manipulate web resources. Using HTTP methods as stated in the RFC is an emerging norm for web services, and deviating from that norm might be a disadvantage, as developers using the API will have to familiarize themselves with a new way of working. The HTML form discovery allows humans to discover the API in a HATEOAS-like manner, and reduces the need for extra documentation.

3.3.4 *WebDAV*

Web-based Distributed Authoring and Versioning (WebDAV) is not an API, but rather an extension of the HTTP protocol with additional methods and headers. [36] states that the basic goal of WebDAV is to "support remote collaborative authoring of Web sites and individual documents", basically to let users edit and collaborate on documents on remote servers. Features such as creation/modifying/deletion of files, metadata like author and creation time, locking of files to prevent overwrites and moving/copying of files makes the protocol highly relevant for this project.

Part III

DESIGN & IMPLEMENTATION

4 REQUIREMENTS

The focus of this chapter is to determine the requirements needed for a solution to the objective presented in Section 1.3.

4.1 BACKGROUND

Aside from the opportunity of opening the API developed to third-party developers, the main purpose of developing the API is for current and future rich media applications made by Inspera to use it to scale without scaling costs. The most prominent of these applications is Creaza, which will be discussed in detail in Section 4.2.1. These applications are currently powered by ICS (see Section 1.2) on the server side. ICS has a wide array of features, only a few of which are actually used by the applications. Looking at the list of design guidelines in Section 3.1, it's clear that ICS is *too powerful* for its use, which also makes it *harder to learn* than it has to be.

4.2 FUNCTIONALITY REQUIREMENTS FOR SAMPLE APPLICATIONS

This section examines a few real and imagined scenarios for use of the API in order to determine its requirements.

4.2.1 *Creaza*

Creaza's sophisticated video editor uses Adobe Flash to do all video operations (such as clipping, applying effects and transitions) for preview on the client side. Creaza productions are stored in an Extensible Markup Language (XML) format containing references to media objects along with information on how they are used. It is essential that there is some way to refer to content that is not too different from how this is currently done, i.e. by each content element and each revision of an element having unique numeric identifiers.

One of Creaza's core features is uploading media for use in video productions, and the next version also allows users to or-

ganize their files in a directory tree. Upload functionality, as well as the capability of organizing data in hierarchical collections and being able to retrieve them by their paths is therefore necessary. Tasks connected to organizing include moving, copying and deleting content.

The consumer version of Creaza introduces a new concept, the *collaboration space.* The idea is that a group of users, e.g. people who attended the same event, shall be able to share some or all their media from an event into a common pool, organized as a directory tree to which all members of the group have access. This allows them to use everything in the pool for their productions, regardless of who originally added it. This poses new requirements.

Permissions for content must be managed not only on a user level, but it must also be possible to create groups and manage permissions on the group level. In addition, to avoid replication of content, it must be possible to have several paths point to the same content, so that an object can be referred to both by its path in the user's folder, and also by its path in a group's collaboration space. In addition, it should be possible to tag content with text tags, and search for content based on name, tags or other metadata.

In order for users to easily invite contacts that are not yet Creaza users into a collaboration space, Creaza wants to access the user's list of contacts from any applications such as Facebook or Google that the user allows it to. This should happen without the user disclosing his password for the third-party service to Creaza. Users should be able to use the credentials of the third-party service account through which they were invited to sign in to Creaza.

Requirements

- Storage and retrieval of content
  - Both text content like XML, and binary content like video
  - Must be able to associate metadata with content
  - Content is uploaded via HTTP
  - Content should be organized in a hierarchy
    - Must be able to move and copy items across hierarchy, as well as delete

  - Content must be able to refer to other content by hierarchy path or numeric ID

- User and permission management
  - All content has an owner and/or a creator with full access
  - Any content that is part of a project is readable by anyone in that project
  - Must be able to set permissions by collection and by group.

4.2.2 *A blog application*

The core feature of a blog is displaying the blog's postings, either in their entirety or short versions, in reverse chronological order. In addition, any posting must be individually accessible through a dedicated URL. The owner of a blog must be able to make new postings, as should anyone they wish to allow to do so. It is often possible for readers to make comments on postings, which may have to be moderated before they are visible to other readers. While the postings themselves are text, they may contain attachments such as images or videos to be displayed inline. The postings should be searchable.

Requirements

- Storage and retrieval of content
  - Text content (blog data, posts etc) and binary data (attached images/videos)
  - Upload of binary data via HTTP
  - Must be able to retrieve content by ID, retrieve all objects of a certain type or with a certain parent ID.
  - Should support text search
- User/permission management
  - Logging in and out
  - All content has an owner
  - Different permissions for interaction with different object types. Only the owner of a blog should be able to create new posts, but everyone should be able to comment on a blog post.

4.2.3 *A simple CMS*

Consider a mobile application with a need to regularly show new information, in the form of announcements, support or just new content. This could either be done by releasing new versions of the application, requiring users to constantly update them, or by having a central location the application could query for new information each time it launched.

- Storage and retrieval of content
  - Text content in addition to binary data such as images.
  - Must be able to retrieve content by ID or type.

4.3 NATURE OF TARGET APPLICATIONS

The core functionality of many Internet applications is essentially an Internet-based version of a traditional desktop application. Examples of this include Google Documents[1] (Word processing, spreadsheet, etc.), Splashup[2] (Image editor), and Creaza[3] (Video editing and rendering). In their desktop form, these applications operate on files in some format, that are loaded into the application, manipulated within it, and stored to the filesystem. Changes to one such file does not propagate to other parts of the filesystem, except for the case when one file references or includes another. This differs from applications like warehouse management, where adding an order will change the number of items remaining in the warehouse and affect future orders, and where sums and other aggregations and queries over all the data in the application are frequently used. This nature is not unique to Internet applications that have desktop equivalents, however one can argue that Creaza is part of a group of applications that share a few key properties:

- Work is done on individual units of data, that do not influence each other, apart from references.
- Changes to one such unit does not influence the application as a whole.
  - Aggregations of data across the whole application are not used, or do not have great influence on application behavior.

1 http://docs.google.com
2 http://www.splashup.com
3 http://www.creaza.com

Regardless of whether these applications have a desktop equivalent or not, they tend to do most of their work on the client side, and the most important abstraction the API must provide to these applications is the notion of a "filesystem" in which files can be stored, organized and retrieved. By using *the cloud* as this filesystem the rest of the application can be implemented on the client side, removing the need for application specific programming on the server.

4.4 SUMMARY OF FUNCTIONAL REQUIREMENTS

The previous sections shows that the needs of the sample applications are centered around content. The main goal of the API is thus to be a way to store and retrieve content, but it also has to be able to handle user management and permissions. The basic functional requirements from the previous discussion are summarized in Table 4.1. These requirements also takes advantage of the cloud computing model, with its cheap data storage and its ability to scale. It should also be possible to download a private version of the platform, to be able to test and develop applications code on a local machine.

Requirement	Description
R1	Interaction with the system should be done through a REST API.
R2	Objects should be stored in a tree structure.
R3	It should be possible to refer to objects both by their hierarchy path and their unique ID.
R4	Objects should contain metadata such as creation time and owner.
R5	Each object can represent a collection/folder or a content element/file.
R6	Content element objects can both contain text information (such as XML) or binary data, such as images or video.
R7	It should be possible to move/copy objects to different locations in the tree.
R8	Objects should not be updated or deleted, instead new revisions should be created.
R9	Objects should be queryable on their properties and their relationship with other objects.
R10	It should be possible to set permissions for each object (or sub tree), both for individuals and for user groups.
R11	The system should be able to serve object data in multiple formats, such as XML, JSON and AMF.
R12	Users should be able to authenticate themselves using credentials they already have, e.g. Facebook, Twitter, Google, Yahoo
R13	It should be possible to create and manage groups of users.
R14	Users should be able to allow the API to use information from their accounts on other services by authenticating to that service, without disclosing their password to the API.

Table 4.1: Requirements

5 THE MORAXUS PLATFORM

In order to overcome reusability and scalability challenges in a way satisfying the requirements from Section 4.4, the Moraxus platform was developed. Its core functionality is storage, management and retrieval of any kind of data, as well as management of users and their access to the data. What sets Moraxus apart from competing infrastructure and data store offerings, such as Google App Engine, is that no server-side development is necessary, or even possible. Once Moraxus is set up, the developer only interacts with it through its REST interface [1].

Benefits

There are several reasons for this design:

- Scope. Allowing developers to develop their own server side modules would drastically increase the amount of work required to make Moraxus a reality.
- Differentiation. Providing infrastructure and a data store while leaving server side development to the customer is something several mature services such as Google App Engine already provide, creating too similar a service with no real competitive edge would be a waste.
- Encapsulation. All implementation details of Moraxus are hidden behind the HTTP interface. This allows for changes to anything about the implementation, no matter how drastic.
- API centered. API centered platforms allows third-party services to access its data, which is a desirable feature as seen in Section 1.1.
- Freedom for developers. While developing an application on Google App Engine requires knowledge of Java or Python specifically, all a developer needs to know to program for Moraxus is how to create HTTP requests and parse JSON/XML using their favorite language or tool.

1 A web-based control panel is planned. It too, however, will interact with Moraxus via the HTTP interface.

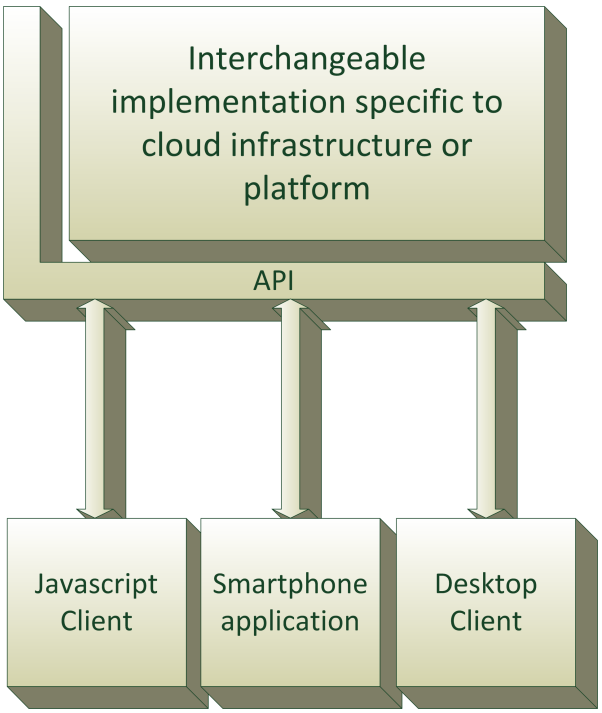


Figure 5.1: The Moraxus platform

- Reusability. Implementations are completely reusable across applications and clients, and generic client side library functions for making API calls are reusable by other Moraxus-based clients written in the same language. See Figure 5.2 for an architectural overview of the components of a sample Moraxus applications, with samples of the type of code present in each component.

As discussed in Section 1.1, the ability to scale is crucial for any growing web application. To be able to do this in a resilient and efficient manner, the platform is designed to run in a cloud environment, as discussed in Section 2.2. Since its API is HTTP only, it can be implemented in any language by any suitable cloud platform or infrastructure, and implementations are interchangeable, see Figure 5.1. It should be noted that Moraxus not is a competitor to full fletched PaaS solutions, but a platform where developers quickly and easily can connect to, and use, a scalable back-end when developing new applications.

Previous work

Architectures similar to this have been described before. Chang et al. [37] describe what they call a "2-tier cloud architecture", for rich applications . They argue that rich applications are best

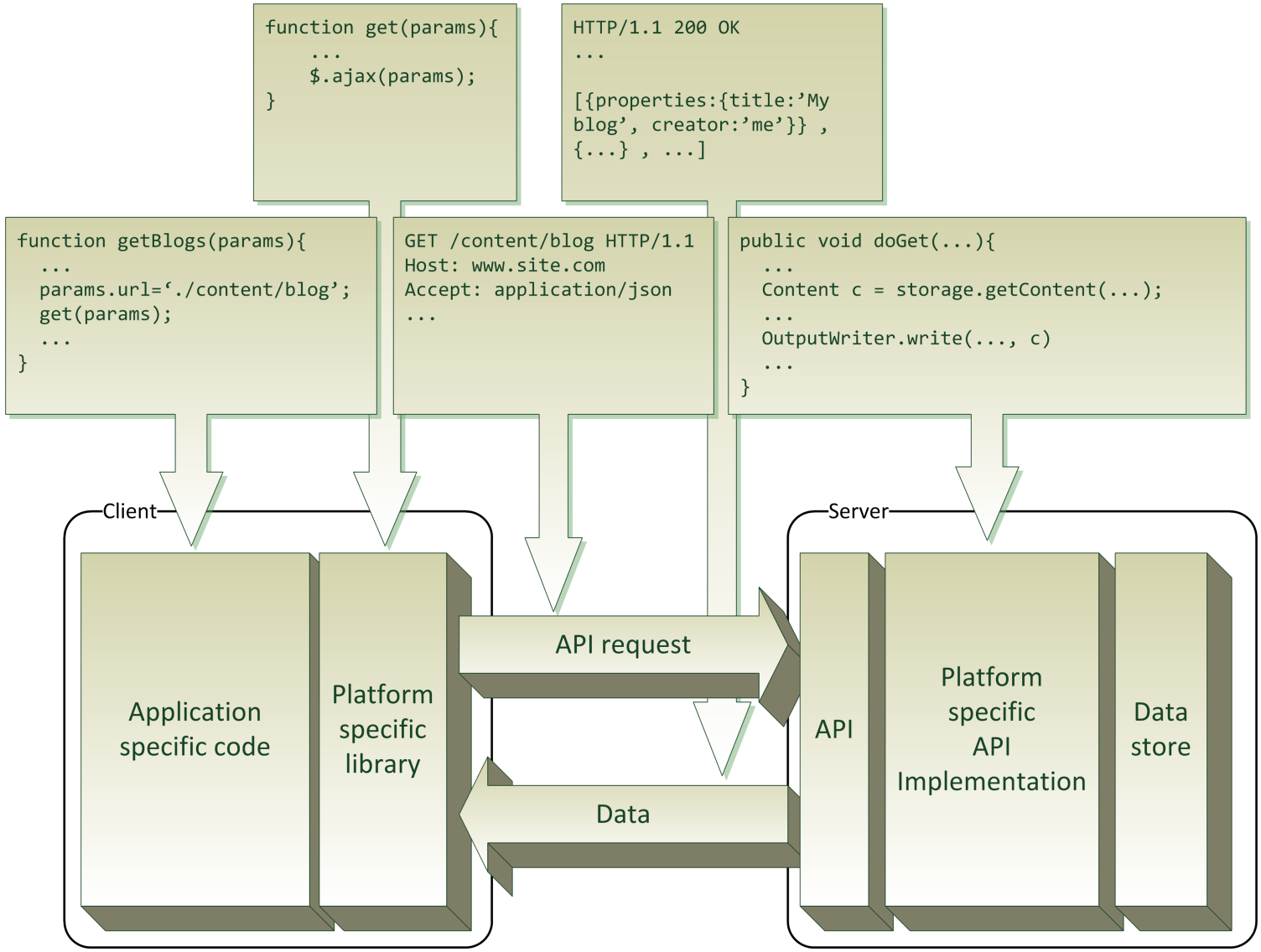


Figure 5.2: Moraxus architecture

suited handling presentation and application logic on the client side, as this allows a simpler API and reduces strain on the server. The *cloud tier*, or server side, handles only storage and exposes a REST API for interaction with the application.

Haselmann et al. [38] have attempted to design a generic REST API for Database as a Service (DaaS) providers. While they found Structured Query Language (SQL) to be a troubling element to success they are optimistic about the feasibility of such an API for NoSQL data stores.

Drawbacks

There are also some drawbacks to the platform design, most notably that applications that require a lot of custom server side processing and aggregation of data will not have a way to do so. These applications should be developed by other means. The applications that benefit the most from using Moraxus are those that mostly modify their data on the client side, such as:

- Editors, such as spreadsheet applications, word processors, image manipulation programs, video and audio editors. Using Moraxus, one can enrich these applications with cloud storage, allowing easy sharing, access control, version control, and universal availability of content at very little development cost.
- Games. A user's game state can be kept with Moraxus, and when he logs back on, regardless of where, he can pick up playing where he left off. System-wide high score lists can be kept.
- Applications that needs to synchronize data between all application instances, for example announcements to mobile apps.
- Any application that uses its server back-end in a similar fashion.

5.1 FEATURES

While the key objective of the Moraxus platform is to store and handle content, several complementary features, such as a powerful query system, is available to make this more efficient and flexible.

5.1.1 *Data store*

The data store is organized as a tree. A node can be a content element, a collection of content, or both. Nodes can have any number of fields containing any type of data, although they must contain the field `type`. Nodes can be required to have certain fields in order to be part of a collection or have a certain type. Figure 5.3 on the following page shows an example of how this might work in practice.

5.1.1.1 *Versioning*

A content element can have a set of revisions, and if desired, some of the fields can be specific from revision to revision. To be able to handle versioning of objects, the nodes itself does not contain the object data. Instead the node contains a list of object IDs, the revisions of the object, and the current active revision ID.

5.1.1.2 *File uploads*

Moraxus allows for uploading of all kinds of files to its data store. Uploaded files are stored as objects with a field of binary data for the file content. Certain known file types, such as .png or .bmp image files will be stored with a collection of metadata in the form of name-value pairs. For an image file data such as width and height will be stored, while data such as file size will be stored for all uploads.

5.1.2 *Queries*

With no server side development possible, a flexible and powerful query system is needed to be able to create complex applications. Moraxus will thus have a query system inspired by the HiFi API (see Section 3.3.1). Objects can be queried on their properties, their parent/child objects and, in the case of file uploads, their file type. The results can be ordered on any field. With the addition of recursive and nested queries, this system should be easy to learn and use, yet flexible and useful.

5.1.3 *Authentication*

Authentication is the process of verifying a user's identity, for instance when logging into a website. Moraxus will not have

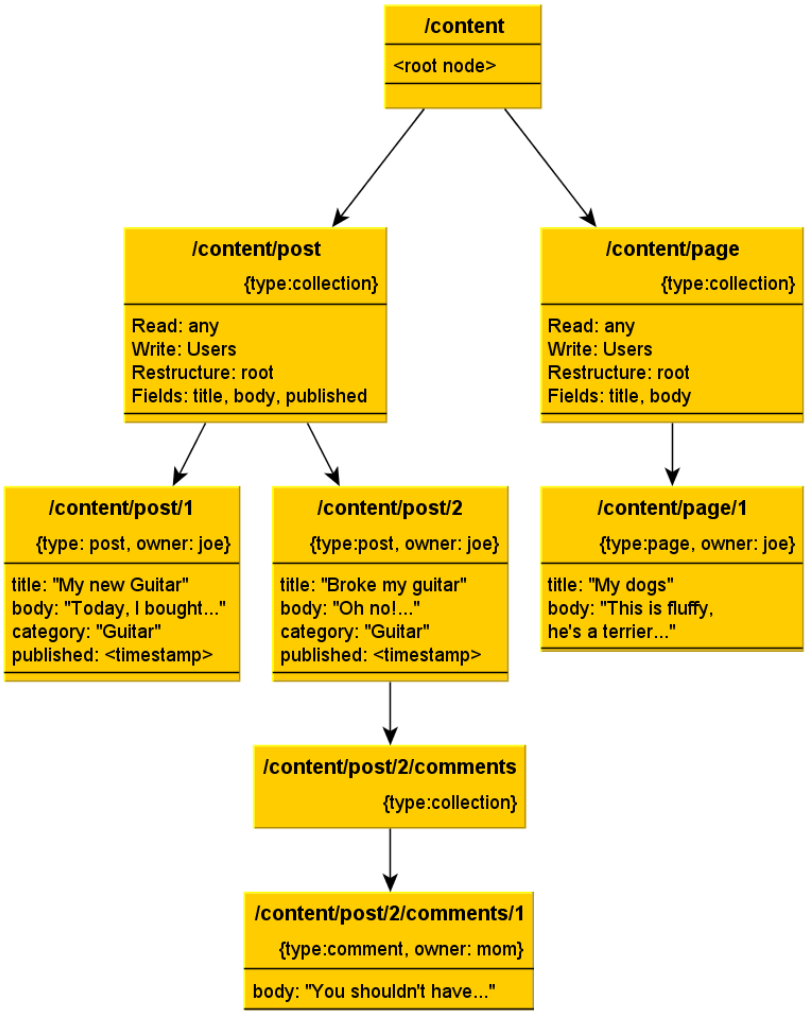


Figure 5.3: Example data store organization for a blog application

its own verification process, but will instead rely on third party services and protocols to provide this functionality. Two such services are OpenID and OAuth, see Appendix B.

5.1.4 *Access control*

Moraxus uses a Role Based Access Control Model (RBAC), as discussed by Kumari[39], where users have given roles (owner, editor etc) in relation to objects. Each role has a set of permissions (read, write, delete etc) connected to it.

Access to objects and collections are controlled by Access Control Lists (ACLs). An ACL specifies which users have which roles in relation to an object. So an ACL list for a given object could for instance contain the entries `(Alice, Owner), (Bob, Editor)` which would give Alice the right to read, write, delete or restructure the object, and Bob the right to read, edit or overwrite it. If a user tries to something to an object which he does not have permission to do a `403 Access Denied` error is returned.

5.1.5 *Groups and sharing*

Moraxus supports the creation of groups, which simply is a list of users. The purpose of groups is to simplify the process of sharing objects with multiple people, such as in Creaza's collaboration space (see Section 4.2.1). When sharing an object with a group all group members can either be given `READ` access or `WRITE` access to the object.

5.1.6 *Application descriptors*

With the goal of having the users do as little server side administration and configuration as possible, finding an elegant practical way to handle permissions and security is difficult. Permission and access checks will have to made on the server side as requests originating from the client side of course can be manipulated. This will be done with the help of application descriptors, application specific files with configuration details about each application. See Listing 5.1 for an example application descriptor.

Listing 5.1: An example XML configuration file

```
<app-descriptor>
        <app-instance>
                <name>PRO blog</name>
                <version>1.0</version>
                <url>http://moraxus-api.appspot.com/
                    blogClient/index.html</url>
                <owner>Gaute</owner>
        </app-instance>
        <permissions>
                <default>read</default>
                <permission>
                        <url>content/comment</url>
                        <value>write</value>
                </permission>
                ...
                ...
        </permissions>
</app-descriptor>
```

5.1.7 *Data migration*

Moraxus supports the transfer of its complete database from one Moraxus implementation to another, for instance from one running on Google App Engine to one running on Amazon EC2. This will allow developers to move their application to other infrastructures, if it should prove beneficial, and avoid vendor lock-in.

6 PROTOTYPE

In order to investigate the viability of the Moraxus design, a prototype was developed. The prototype consists of an implementation of the core functionality in the Moraxus API, as well as a simple multi-platform application using that functionality. Rather than implementing every last part of the specification, emphasis was placed on completing functionality that would help evaluate how effective development with Moraxus would be in practice.

6.1 PROTOTYPE MORAXUS IMPLEMENTATION

While the Moraxus API can be implemented on a variety of platforms and infrastructures, the prototype was implemented on Google App Engine only. Arrangements were made for the prototype to run on Amazon EC2 using Cassandra (Section 2.2.2) as the data store, but it was soon decided that basing the prototype on a PaaS was preferable to an IaaS, as the time spent setting up the environment before actual development is significantly shorter. App Engine was chosen as it is known to perform well, allows development in the favorite language of the authors, and is free to get started with.

The prototype implements what are thought to be the most important requirements, listed in Table 6.1. It is implemented in Java and it is currently running at `www.moraxus-api.appspot.com`. It uses Google's BigTable[1] NoSQL database system for data storage and Google Accounts for user authentication.

6.2 SAMPLE APPLICATION: A SIMPLE BLOG

The sample application implemented in the prototype is a simple blog service. It contains a selection of blogs, each of which has an owner, and a set of postings. Each posting is written by an author, and has a title, a body, and optionally a category. It is possible for users to leave comments on postings, which appear underneath it. Each comment has an author and a body.

1 http://labs.google.com/papers/bigtable.html

Requirement	Description
R1	Interaction with the system should be done through a REST API.
R2	Objects should be stored in a tree structure.
R3	It should be possible to refer to objects both by their hierarchy path and their unique ID.
R5	Each object can represent a collection/folder or a content element/file.
R9	Objects should be queryable on their properties and their relationship with other objects. *Partly implemented, objects are not queryable on their relationship with other objects.*
R10	It should be possible to set permissions for each object (or sub tree), both for individuals and for user groups. *Only possible for individuals*
R11	The system should be able to serve object data in multiple formats, such as XML, JSON and AMF.

Table 6.1: Requirements implemented in the prototype

In order to be allowed to make a posting, a user must have write access to the collection object representing the blog to which he wants to post. The default behavior in Moraxus is to make objects private to the owner, which means that the creator of a blog is the only person who can make postings on it. This behavior was left unchanged, so only the owner of a blog can make postings on that blog. This highlights an interesting aspect what developing with Moraxus is like. Since the ability to grant or revoke permission to objects is in the API, it is possible for an end user to make a blog to which several people can post by sending a correctly formatted API request modifying a blog's permission, even though there is no such functionality in the application's user interface.

This shows:

- That this kind of functionality is very easy to amend to applications.
- That developers need to be careful about what they assume about the objects in the data store.

Method	URL	Parameters	Description
`POST`	`content/blog/`	`title`, `creator`	Creating a new entity of type blog with the given title and creator.
`GET`	`content/blog/2/post`		Get all blog posts from blog 2
`DELETE`	`content/blog/2/post/3`		Delete blog post 3 from blog 2
`GET`	`user?method=getUser`	`destinationUrl`	Handle the user login process and redirect the user back to the destinationUrl when he has logged in.

Table 6.2: Some of the blog application's API calls

- In this case, the author of every posting is stored in the posting, it is not inferred by checking who owns the blog.
- This is more similar to working with a filesystem than working with a database through an application-specific API.

The API call used to create a posting makes the comments collection in that posting writable to any user, allowing comments to be submitted. Table 6.2 shows examples of other calls used in the blog application.

6.2.1 *Functionality*

This blog application has the following functionality:

- Login (with the help of Google IDs)
- Logout
- Create a new post
- Comment on a post
- Delete/update a post
- Delete/update a comment

- Get latest posts
- Get all posts of a certain category

6.2.2 *Web client*

The web version of the blog is a JavaScript application that uses the jQuery library [2] to simplify the code needed to make HTTP requests. There are five pages, the front page, blog view (shows the postings of a blog), posting view (shows a single posting along with comments and a form to post a new comment), the new post form, and the login page. The pages themselves do are static, and contain JavaScript calls that fetch the actual data. While this causes more than one HTTP request to the server per page, each request can be cached if the exact URL has been previously retrieved. This means that frequent visitors of the blog will only load the HTML and JavaScript once, and only load data on request.

Listing 6.1 shows the function for posting a new comment to a post. All interaction with the Moraxus platform is done through calls like this, while the rest of the code simply handles the data returned from Moraxus. Figure 6.1 shows a screenshot of the blog client.

Listing 6.1: Code for posting a new comment

```
function postComment(comment, name, postId){
        $.ajax({
                type: 'POST',
                url: BASE_URL + "id/postId",
                data: {"type":"comment", "content": comment,
                     "name": name},
                success: function(resp) {
                        //do something
                },
                dataType: "json"
        });
}
```

2 http://jquery.com/

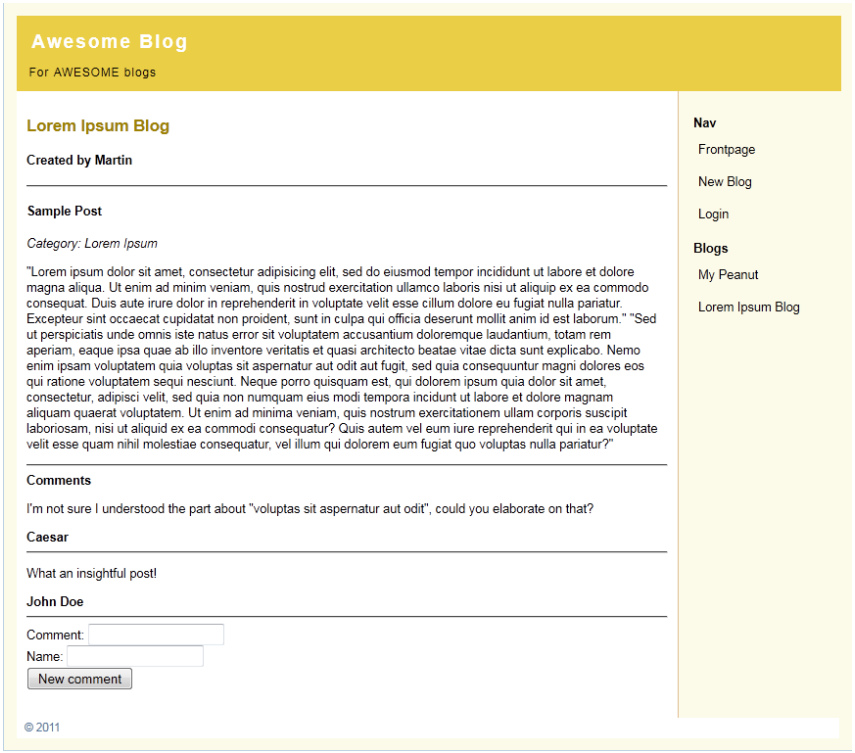


Figure 6.1: The JavaScript blog client

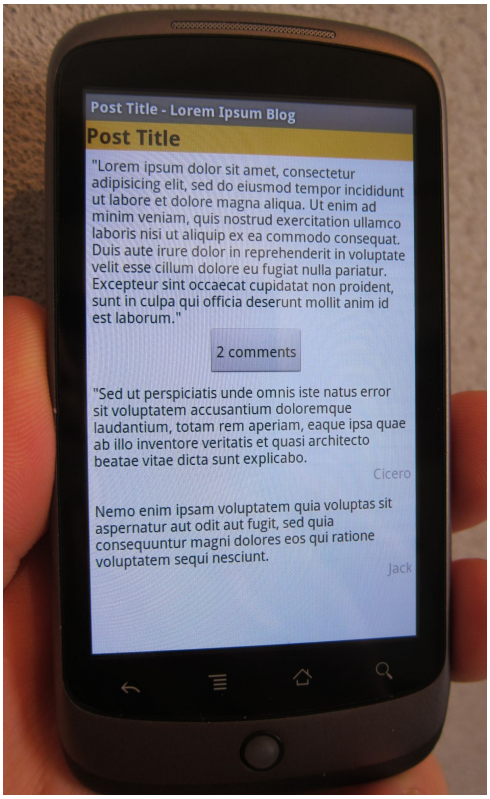


Figure 6.2: The Android blog client

6.2.3 *Mobile client*

A mobile client was also created, using the multi-platform mobile toolkit Appcelerator Titanium. It's communication layer works in exactly the same way as the JavaScript client, however it uses native UI controls for presentation and user interaction. Its UI is designed to display well on smaller screens. Figure 6.2 shows the blog application running on an Android device.

Part IV

EVALUATION AND CONCLUSION

7 EVALUATING MORAXUS

This chapter evaluates the output of this project:

- A model of content organization: the hierarchical Moraxus data model.
- Methods for accessing and manipulating data stored in such a model: The Moraxus API.
- A method for developing ubiquitous applications, where application-specific code resides on the client side, and a configured instance of a completely general Moraxus implementation is used as the entire server side.

In this chapter, the above output is referred to collectively as *Moraxus*. The prototype discussed in chapter 6 is an instantiation of Moraxus, but is considered more of an evaluation tool than a finished product to subject to scrutiny.

7.1 EVALUATION METHOD

The basis for this evaluation is the prototype, the design itself and our experiences from working with the prototype and developing the sample application. We will try to evaluate Moraxus based on how well it solves the objectives presented in Section 1.3, which was to create a solution that:

1. Allows reuse of components
2. Ensures scalability at low monetary cost
3. Can be used by new applications as well as existing ones

Due to our approach of allowing Moraxus to run on almost any platform, evaluating factors like the performance or scalability of our system is a difficult or infeasible, as the performance is entirely dependent on the underlying infrastructure.

With the evaluation being performed by the authors of this report, and the designers of Moraxus, it is hard to quantify the time needed to develop an application on Moraxus compared to other frameworks, as we already know the details of the API and

how to get started using it. We argue, however, that due to its small feature set, adherance to REST principles, and straightforward organization, Moraxus is easy to learn for anyone familiar with Internet application development.

7.2 FULFILLMENT OF OBJECTIVES

Reusability and development speed

The prototype development process has revealed the biggest advantage to using Moraxus: code reusability. Given a server-side implementation, client applications can be developed quickly, even more so in the presence of a client side library. The reusable components can be improved and the improvement will spread to all applications. With a few general client side library functions, interacting with the data store is about as easy as using a filesystem or local database, with the added power of centralized storage. This makes it very easy to get started developing on the platform for new users, as there is no need to read through complex documentation or to do any configuration before using Moraxus. With regard to objective 1, allowing reuse across applications and platforms, Moraxus does very well for the applications it supports.

Scalability and performance

The Moraxus API can be implemented using almost any platform or infrastructure, and its scalability and performance depends greatly on the implementation details of the underlying technology. We argue, however, that this is an advantageous design. Cloud infrastructure and scalable data stores are areas of heavy development, and a design that eases interchange of underlying components allows using the latest advances in these fields, without reinventing the wheel or changing the interface. If Moraxus gets widely adopted, open source implementations for various platforms could be made available, facilitating switching to cheaper, faster, or more reliable components and services as they emerge.

The Google App Engine team[40] demonstrates that the Google App Engine easily can handle traffic of more than 40 000 requests per second (during the royal wedding in England, occurring at the moment the Duke kissed the Duchess of Cambridge), so while the prototype itself was not evaluated for performance, we

argue that this performance should apply to Moraxus as well, and that the design developed achieves the scalability required for most applications.

7.3 ARCHITECTURAL CHALLENGES

The architecture of Moraxus makes it excel for applications that can manage without specific server side application logic. While this pattern fits several classes of applications, there are other applications which require more sophisticated functionality on the server side that Moraxus is not a good fit for. There are also other challenges with this architecture, especially related to the fact that Moraxus should be able to run on most platforms and infrastructures:

QUERIES A complex query system allowing for recursive queries on arbitrary attributes will have to be implemented in an efficient manner, to secure good performance. This will have to be done for every implementation of Moraxus, as different platforms use different data storage solutions and technologies.

CONSISTENCY The behavior of Moraxus have to be consistent across different platforms with different encoding systems etc. Developers should not have to know anything about the underlying infrastructure their Moraxus implementation is running on.

7.4 PRACTICAL CHALLENGES

During the development of the prototype and sample applications, several practical problems with the solution became apparent.

Cross-site JavaScript

As a security measure implemented in all modern web browsers, JavaScript running on a page is not permitted to interact with other web sites. This poses a problem for Moraxus. If Moraxus is running at `www.moraxus-api.appspot.com` and a developer's JavaScript application is hosted at `www.example.com`, then the application will not be allowed to write to Moraxus. Note that this only is a problem for JavaScript applications.

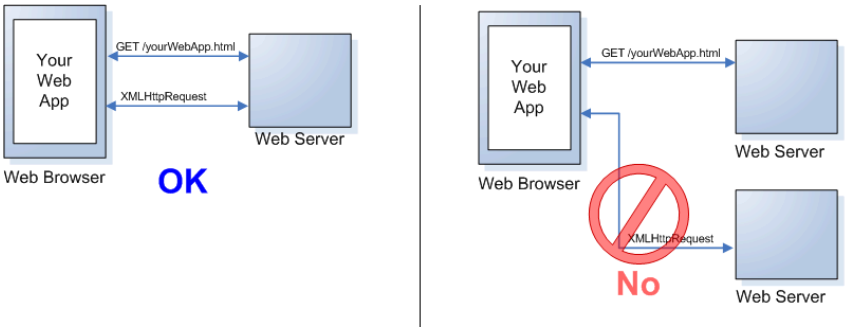


Figure 7.1: The problem of cross-site JavaScript

There exists multiple workarounds for this problem, with varying degrees of browser support and elegance. The most promising solution is probably HTML5s *postMessage* [1], but this feature is only implemented in the most recent browsers. Another possible solution would be to offer the Moraxus platform as part of a hosting service.

Complex application descriptors

When developing anything else than very simple applications configuring the application descriptor will be tedious work, with different types of object having different default permissions for different user groups (owner/logged in/anonymous). The blog application would for instance only want to allow the owner of a blog to `POST` to `/content/post/<blogID>`, but everyone should be able to `POST` to `/content/post/<blogID>/comment`.

Finding a more elegant way to do this is an important task, as Moraxus strives to be an easy to use and intuitive platform.

7.5 FEATURES

While limited, we argue that the feature set offered by Moraxus makes it a good choice for applications such as those discussed in Section 4.3. Recently released services such as Google Storage (Appendix A.3) and Storage Room (Appendix A.4) shows that services offering data storage and little else are viable. Still, with an extremely minimalistic feature set the range of applications that would benefit from running on the Moraxus platform is limited. To increase the viability of the platform, we suggest some additional features as future modules:

1 http://www.w3.org/TR/html5/comms.html

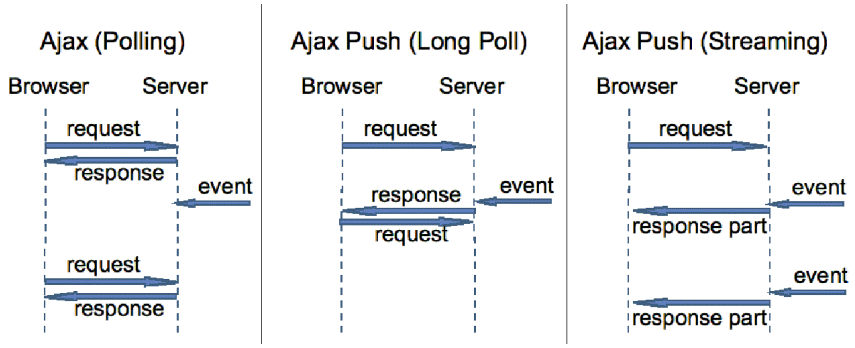


Figure 7.2: Server push techniques

Server push

The traditional event model of the web, that a client connects to a server requesting a resource, for example a web page, is called **polling**. However, the server has no way to initiate contact with a client, or to **push** data to the client. 'Comet' is an umbrella term for techniques that tries to achieve this, either by the use of streaming or long polling. Streaming is the process of creating a single persistent connection from the client to the server which then can be used by the server to send messages to the client. However, no standard for this is defined and it does not work across all browsers. The second option is the use of long polling, which essentially is the same as traditional polling, except that the server does not respond before something happens. However, this require the client to make a lot of requests. See figure 7.2 for an illustration of these techniques.

Giving Moraxus the ability to push information to connected clients makes it viable to create a wider range of application on the platform. An example of this would be a multiplayer game (chess, tic-tac-toe, etc) implementation where both players listen for changes to a game object.

Advanced Configuration Options

To address the needs of a wide variety of applications it would be useful to allow developers more flexibility when configuring their applications. Being able to select where their data should be stored is one important aspect, a Norwegian company would probably prefer their data to be stored somewhere in Europe and not in the US, due to latency. Another interesting possibility

is to choose where along the CAP spectrum (see Section 2.2.2) an application should be.

7.6 USABILITY

At the moment all communication with Moraxus goes through HTTP requests or the application descriptor XML file, making it unnecessary hard to define types, set URL permissions etc. However, a web-based control panel, as mentioned in chapter 5 would help a lot with these issues.

Client libraries

Working with Moraxus involves making a lot of HTTP requests and JSON/XML parsing. As most developers would agree with these are not the most enjoyable tasks when programming, making the use of Moraxus a tedious and monotone task. As the goal of the Moraxus platform is to make it easy and convenient to develop new applications, client libraries containing API wrappers would greatly reduce these problems by making it easier and faster to write code interacting with our platform. An API wrapper is simply a layer between the application code and the API in form of a language specific client library. Adding a wrapper around the API interaction would make the application code both cleaner and more elegant. Listing 7.1 shows how the post comment call from Section 6.2.2 could be made simpler with the help of the reusable helper function *post*.

Providing API wrappers for popular languages and platforms would both lower the barrier of entry to use Moraxus and simplify application development significantly.

Listing 7.1: Posting a comment with the help of an API wrapper.

```
function postComment(comment, name, postId){
        post(postId, {"type":"comment", "content": comment,
            "name": name});
}

function post(parentID, data, onSuccess){
        $.ajax({
                type: 'POST',
                url: BASE_URL + "id/" + parentID,
                data: data,
                success: function(resp) {
                        onSuccess();
                },
                dataType: "json"
        });
}
```

7.7 DISCUSSION

While Moraxus can support large, complex applications, its advantages are most apparent when using an existing Moraxus implementation to quickly create new, smaller applications. Given a client library and a working Moraxus instance, application developers can create great cloud-enabled functionality with very little effort or know-how. The ability to exchange the underlying infrastructure or platform also is the most effective when a variety of platforms is already available. For Moraxus to really shine as a platform, implementations should be open source, or at least freely available. This will allow application developers to choose freely between cloud providers, and encourage the providers to compete on price.

7.8 FURTHER WORK

During the course of this project several services with similar feature sets as Moraxus has been launched, which are discussed in Appendix A. Storage Room (Appendix A.4), launched in the same month as this report, is the most similar. It allows storage of arbitrary data in a hierarchical structure with validatable collections, and access to this data through a RESTful JSON API.

It also includes a web-based control panel in which content can be edited and organized. While this service is not an implementation of the Moraxus API exactly as described here, it is an instantiation of the ideas which it embodies, and has been well received[2].

While Storage Room and services like it may offer developers of this kind of application what they need, there is also room for further development. Open source implementations of the ideas presented here would give the all the same benefits, while avoiding vendor lock to commercial services.

2 http://news.ycombinator.com/item?id=2616041

8 CONCLUSION

At the start of this report, we highlighted scalability and lack of code reusability as problems in traditionally designed Internet applications. We proposed solving these problems using an API containing essential server side functionality, whose implementations could be reused across different applications and clients.

After a discussion of theory and the state of the art in API design, data stores and infrastructure, we gathered requirements for the API based on a generalized analysis of target applications. Due to the nature of the target applications, and analyses performed by previous work on closely related topics, we reasoned that data management and user access control are the only truly essential server side components of many applications. Remaining application and presentation logic can be shifted to the client side, which reduces server strain, improves cache efficiency and saves development effort.

Based on this insight, we designed the Moraxus API. It contains a generalized set of functionality centered around storage, management and retrieval of arbitrary data, as well as managing users, groups, and their access to the content. As it is designed to serve as the entire server side of applications, no additional server side development is necessary, or even possible. The API can be implemented using a variety of cloud services and data storage solutions, and has a mechanism for easy migration of data between implementations.

We developed a prototype implementing essential API features, and a multi-platform blog application using the prototype for data storage and user authentication. Based on the prototype, we evaluated the ability of the development approach to solve the identified problems related to reusability and scalability. For applications that do not require a complex back-end, using an existing Moraxus implementation reduces development time by a significant amount. Using client-side libraries for communicating with Moraxus allows for further time savings. Moraxus' ability to leverage advances in scalable data storage and high performance cloud infrastructures makes it easy to develop and reuse highly scalable application components.

While Moraxus has some flaws and is not applicable in every setting, we argue that it is a valuable contribution to solving problems related to scalability and code reusability.

BIBLIOGRAPHY

[1] Takeda H, Veerkamp P, Tomiyama T, Yoshikawa H. Modeling design processes. AI Mag. 1990 October;11:37–48. Available from: http://portal.acm.org/citation.cfm?id=95788.95795.

[2] Hevner AR, March ST, Park J, Ram S. Design Science in Information Systems Research. MIS Quarterly. 2004;28(1):pp. 75–105. Available from: http://www.jstor.org/stable/25148625.

[3] Richardson L, Ruby S. RESTful Web Services. O'Reilly; 2007.

[4] Facebook. Facebook Statistics; 2011. [Cited May 30, 2011]. Available from: https://www.facebook.com/press/info.php?statistics.

[5] Craig S, Sorkin AR. Goldman Offering Clients a Chance to Invest in Facebook. New York Times; 2011. [Posted Jan 2, 2011 , cited May 30, 2011]. Available from: http://dealbook.nytimes.com/2011/01/02/goldman-invests-in-facebook-at-50-billion-valuation/.

[6] Carlson N. Facebook Valued At $124 Billion In (Wacko) Private Market Transaction. Business Insider; 2011. [Posted Jan 6, 2011 , cited May 30, 2011]. Available from: http://read.bi/gB4D1j.

[7] Ante SE, Efrati A, Das A. Twitter as Tech Bubble Barometer. Wall Street Journal; 2011. [Posted Feb 10, 2011 , cited May 30, 2011]. Available from: http://dealbook.nytimes.com/2011/01/02/goldman-invests-in-facebook-at-50-billion-valuation/.

[8] Ganapati P. Toaster, Toilet Lead Appliance Invasion of Twitter. Wired; 2009. [Posted Aug 5, 2009 , cited May 30, 2011]. Available from: http://www.wired.com/gadgetlab/2009/08/twittering-toaster/.

[9] Hoff T. Friendster Lost Lead Because Of A Failure To Scale; 2007. [Posted Nov 13, 2007 , cited Apr 07, 2011]. Available

from: http://highscalability.com/blog/2007/11/13/friendster-lost-lead-because-of-a-failure-to-scale.html.

[10] Rivlin G. Wallflower at the Web Party. New York Times; 2006. [Posted Oct 15 2001 , cited May 26 2011]. Available from: http://www.nytimes.com/2006/10/15/business/yourmoney/15friend.html?pagewanted=all.

[11] Levy S. In: In the Plex: How Google Thinks, Works, and Shapes Our Lives. Simon & Schuster; 2011. p. 185–188. Available from: http://books.google.com/books?id=V1u1f8sv3k8C.

[12] March ST, Smith GF. Design and natural science research on information technology. Decis Support Syst. 1995 December;15:251–266. Available from: http://portal.acm.org/citation.cfm?id=222827.222832.

[13] Vaishnavi V, Kuechler W. Design Research in Information Systems. Order. 2008;48(2):1–393. Available from: http://desrist.org/design-research-in-information-systems.

[14] Bondi AB. Characteristics of scalability and their impact on performance. In: Proceedings of the 2nd international workshop on Software and performance. WOSP '00. New York, NY, USA: ACM; 2000. p. 195–203. Available from: http://doi.acm.org/10.1145/350391.350432.

[15] McGiboney M. Twitter's Tweet Smell Of Success. The Nielsen Company; 2009. [Posted March 18, 2009 , cited Apr 07, 2011]. Available from: http://blog.nielsen.com/nielsenwire/online%20mobile/twitters-tweet-smell-of-success/.

[16] Amazon Web Services Blog. Amazon EC2 Beta; 2006. [Posted August 25, 2006 , cited May 05, 2011]. Available from: http://aws.typepad.com/aws/2006/08/amazon_ec2_beta.html.

[17] Mell P, Grance T. The NIST Definition of Cloud Computing (Draft). National Institue of Standards and Technology; 2011. [Posted Jan 2011 , cited May 07 2011]. Available from: http://csrc.nist.gov/publications/drafts/800-145/Draft-SP-800-145_cloud-definition.pdf.

[18] Deelman E, Singh G, Livny M, Berriman B, Good J. The cost of doing science on the cloud: The Montage example. In: High Performance Computing, Networking, Storage and Analysis, 2008; 2008. p. 1.

[19] Seybold P. Update on PlayStation Network and Qriocity; 2011. [Posted Apr 26 2011 , cited May 25 2011]. Available from: http://blog.us.playstation.com/2011/04/26/update-on-playstation-network-and-qriocity/.

[20] Ciancutti J. 5 Lessons We've Learned Using AWS; 2010. [Posted Dec 2010 , cited Apr 27 2011]. Available from: http://techblog.netflix.com/2010/12/5-lessons-weve-learned-using-aws.html.

[21] Amazon Web Services Team. Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region; 2011. Available from: http://aws.amazon.com/message/65648/.

[22] Pritchett D. BASE: An Acid Alternative. Queue. 2008 May;6:48–55. Available from: http://doi.acm.org/10.1145/1394127.1394128.

[23] Lynch N, Gilbert S. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services; 2002. Available from: http://lpd.epfl.ch/sgilbert/pubs/BrewersConjecture-SigAct.pdf.

[24] Cattell R. Scalable SQL and NoSQL Data Stores; 2011. [Revised April 24, 2011 , cited May 10, 2011]. Available from: http://www.cattell.net/datastores/Datastores.pdf.

[25] Thomson A, Abadi D. The problems with ACID, and how to fix them without going NoSQL; 2010. [Posted August 31, 2010, cited May 09, 2011]. Available from: http://dbmsmusings.blogspot.com/2010/08/problems-with-acid-and-how-to-fix-them.html.

[26] Thomson A, Abadi DJ. The case for determinism in database systems. Proc VLDB Endow. 2010 September;3:70–80. Available from: http://db.cs.yale.edu/determinism-vldb10.pdf.

[27] Cassandra Wiki. Data Model;. [Updated Mar 7, 2011 , cited June 10, 2011]. Available from: http://wiki.apache.org/cassandra/DataModel.

[28] Bloch J. How to design a good API and why it matters. Google Tech Talks; 2007. [Talk given on Jan 24 2007, uploaded Oct 08 2007, cited Jan 25 2010 - slides available at http://lcsd05.cs.tamu.edu/slides/keynote.pdf, cited Jan 25 2010]. Available from: http://www.youtube.com/watch?v=aAb7hSCtvGw.

[29] Fielding RT. Architectural Styles and the Design of Network-based Software Architectures; 2000. [cited Mar 15, 2011]. Available from: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[30] Internet Engineering Task Force. RFC 2616: Hypertext Transfer Protocol – HTTP/1.1; 1999. [Posted Jun 1999 , cited Mar 15 2011]. Available from: http://www.ietf.org/rfc/rfc2616.txt.

[31] Fielding RT. REST APIs must be hypertext-driven; 2008. [Posted Mar 20 2008 , cited Mar 16 2011]. Available from: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven.

[32] Fowler M. Richardson Maturity Model: Steps toward the glory of REST; 2010. [Posted Mar 18 2010 , cited Mar 16 2011]. Available from: http://martinfowler.com/articles/richardsonMaturityModel.html#level3.

[33] Jordan K. First we built an API, then we built a CMS; 2010. [Posted 22 Dec 2010, cited Mar 3, 2011]. Available from: http://www.gethifi.com/blog/first-we-built-an-api-then-we-built-a-cms.

[34] Adam DuVander. Twitter Reveals: 75calls per day); 2010. [Cited June 11, 2011]. Available from: http://blog.programmableweb.com/2010/04/15/twitter-reveals-75-of-our-traffic-is-via-api-3-billion-calls-per-

[35] Yudaken J. A Browsable RESTish API; 2011. Posted 20 Feb 2011, cited Mar 10, 2011]. Available from: http://developers.snapbill.com/2011/02/a-browseable-restish-api/.

[36] Whitehead J. In: WebDAV: versatile collaboration multiprotocol. vol. 9; 2005. p. 75–81. Available from: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1407781.

[37] Zhang W. 2-Tier Cloud Architecture with maximized RIA and SimpleDB via minimized REST. In: Computer Engineering and Technology (ICCET), 2010 2nd International Conference on. vol. 6; 2010. p. V6–52 –V6–56. Available from: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5486290.

[38] Haselmann T, Thies G, Vossen G. Looking into a REST-Based Universal API for Database-as-a-Service Systems. In: Commerce and Enterprise Computing (CEC), 2010 IEEE 12th Conference on; 2010. p. 17 –24. Available from: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5708388.

[39] Sirisha A, Kumari GG. API access control in cloud using the Role Based Access Control Model. In: Trendz in Information Sciences Computing (TISC), 2010; 2010. p. 135 –137.

[40] Peter S Magnusson. Royal Wedding Bells In The Cloud; 2011. [Posted May 6 2011 , cited June 6, 2011]. Available from: http://googleappengine.blogspot.com/2011/05/royal-wedding-bells-in-cloud.html.

Part V

APPENDIX

A

PAAS SERVICES

In order to evaluate the viability of our platform and to be able to compare it to other services, this appendix introduces a few PaaS services and their feature sets.

A.1 GOOGLE APP ENGINE

Google App Engine[1] is a PaaS developed by Google which supports server side programming in Java, Python and Go. The platform is well integrated with other Google products by allowing authentication through Google Accounts and storage through their BigTable technology. The platforms makes it very easy to write scalable applications, but only a limited range of applications are supported due to restrictions in the platform.

A.2 CLOUD FOUNDRY

Cloud Foundry[2] is an open source PaaS product developed by VMware[3]. Figure A.1 shows the three components of Cloud Foundry.

The Cloud Provider Interface allows Cloud Foundry to run on a multitude of different cloud solutions such as *public clouds* (EC2), *private cloud solutions* and *micro clouds*, which essentially is a sandbox solution that can be downloaded to run on a local computer.

The application service interface provides access to different applications and services, for instance MongoDB og MySQL datastorage. And since Cloud Foundry is open sourced, missing services can be implemented by anyone.

A.3 GOOGLE STORAGE

The beta version of Google Storage [4] was released for developers in May 2011. Google Storage is a service that allows for storing

1 http://code.google.com/appengine/
2 http://www.cloudfoundry.com/
3 www.vmware.com/
4 http://code.google.com/apis/storage/

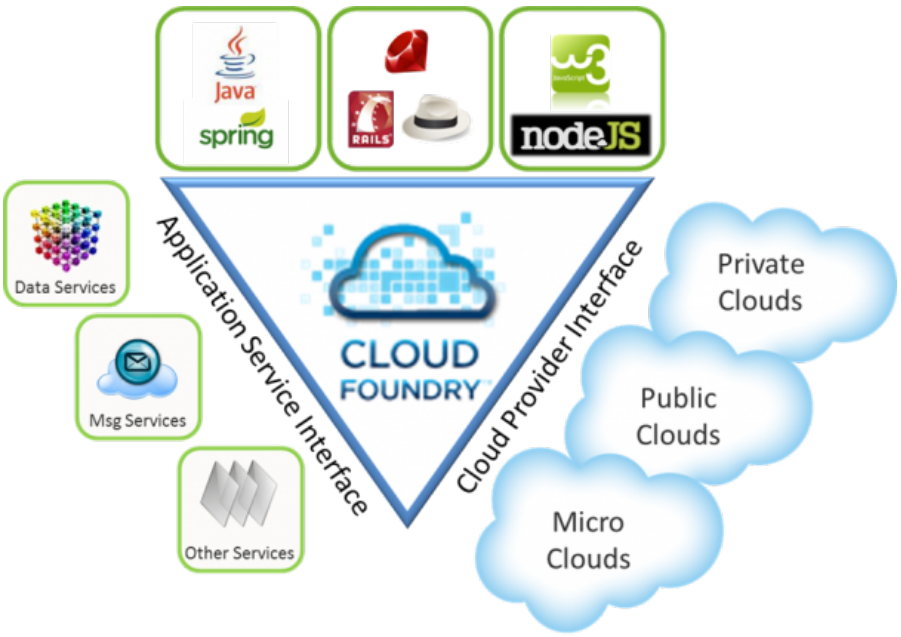


Figure A.1: Cloud Foundry

data in Google's cloud. It is accessible through a REST API, allows for sharing of objects through Google Accounts and use OAuth for authentication. However, it does not support any kind of queries at all, making it impractical for anything else than data storage and retrieval by object ID.

A.4 STORAGE ROOM

Storage Room[5], launched in June 2011, focuses on content management for mobile applications. They allow users to create and store their own data structures through a RESTful JSON API, which also supports file uploads and authentication.

5 http://storageroomapp.com/

B

AUTHENTICATION TOOLS

This appendix gives a short introduction to two of the most widespread standards for authentication and authorization over the internet, OpenID and OAuth.

B.1 OPENID

OpenID [1] is a standard that describes how users can be authenticated in a decentralized manner. This allows a user to log into a service with his credentials from an OpenID provider, for instance Facebook or Google, without disclosing these to any third parties. For the user this process works like this:

1. He will sign in with his OpenID on the website that supports OpenID (see figure B.1).
2. The browser will then take him to the OpenID provider's web site.
3. If he is not logged in to the providers web site he will have to do so.
4. The user have now verified his identity and just have to allow the provider to share information with the requesting web site.
5. He will then be sent back to the original site.

This process benefits both the users and the developers. Users are allowed to use an existing account to sign into multiple websites, avoiding the hassle of creating and remembering new sets of usernames and passwords. And developers does not have to write their own, hopefully secure, login systems, which often can be both time consuming and difficult.

B.2 OAUTH

OAuth [2] is another open standard for authentication. It differs from OpenID in the sense that OpenID mainly is for authenti-

1 http://openid.net/
2 http://oauth.net/

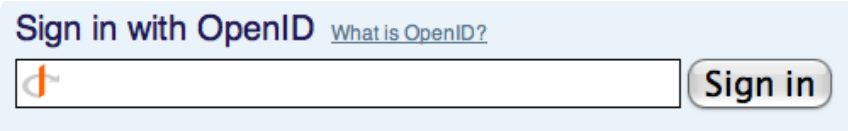


Figure B.1: An OpenID login form

cation (let www.moraxus.com know that I am me) where OAuth also is designed to allow authorisation without giving away a password. An example of this would be to allow a website to Tweet on a user's behalf or to allow a service to upload photos to a user's Flickr account. For the end user the authentication/authorisation process is very similar to the OpenID authentication process, even though the technical details are different.

C

MORAXUS DOCUMENTATION

This chapter describes how Moraxus is intended to be used, by documenting the functionality of the REST API. It is assumed that Moraxus is hosted on `www.moraxus-api.appspot.com`. A blog application will be used as an example throughout this chapter, where each blog has a collection of posts and each post has a collection of comments.

C.1 REQUEST URLS

Moraxus' functionality can be accessed through four different request endpoints:

`/content/<path/to/object>` This is the standard way to access resources, in a very REST-like manner

`/id/<objectid>` An alternative way to access resources by their ID, mostly useful in cases where the resource's full path is not known.

`/user` User operations.

`/group` Group operations

C.2 REQUEST METHODS

The request URLs supports the use of several HTTP request methods, with this section describing the functionality of each method. HTTP request headers are also used to offer additional functionality.

C.2.1 *Content Requests*

GET

`GET` requests are used to get the data fields of an object or, if the request is made to a collection, a list of the objects in the collection. A request made to `/content/blog/32/post/56` could for instance return the result shown in Listing C.1.

Listing C.1: Sample GET object response

```
HTTP/1.1 200 OK
Content-Length: length
Content-Type: application/xml
Date: Mon, 23 May 2011 11:53:09 GMT

<?xml version="1.0" encoding="UTF-8"?>
<post>
        <id>56</id>
        <title>Hello World</title>
        <creator>Kurt</creator>
        <content>Dear diary, today I ate a dozen bullfrogs.
            They were   delicious</content>
</post>
```

HEAD

HEAD requests are used to get metadata about an object or collection, such as its creation time, when it last was modified and who the owner is.

POST

New objects and collections are created through POST requests, with object data sent as the request content. When posting to an URL, the resource located at that URL will become the parent of the new object. A sample POST request for the creation of a blog post can be seen in Listing C.2. In this case, the resource located at /content/blog/23/post will be the new post's parent.

Listing C.2: Sample POST request

```
POST /content/blog/23/post HTTP/1.1
Host: moraxus-api.appspot.com
Content-Type: text/xml; charset=utf-8
Content-Length: length

{"type":"post",
 "title": "Hello World",
 "creator": "Kurt",
 "content": "Dear diary, today I ate a dozen bullfrogs.
  They were     delicious"}
```

PUT

`PUT` requests are used to modify, move or create a copy of an existing resource.

MOVING/COPYING When making a copy or moving an object, the request should be made to the new location of the object/copy. To indicate the wish to copy/move an object an additional request header, inspired by WebDAV (Section 3.3.4), is added to the request. "`Copy-Source`" to copy an object, "`Move-Source`" to move an object, both with the ID of the original object as their value. So to move the blog post with ID 56 (which is currently located at `/content/blog/23/post`) to the blog with ID 20, the request shown in Listing C.3 would be sent. To do this, `WRITE` access to the orginal object as well as to the new location would be required.

Listing C.3: Sample `PUT` request

```
PUT /content/blog/20/post HTTP/1.1
Host: moraxus-api.appspot.com
Content-Type: text/xml; charset=utf-8
Content-Length: length
Move-Source: 56
```

UPDATING When modifying an object a new revision is actually created and added to the list of different revision of the object. This new revision is then set as the active version of the object.

DELETE

Resources are not actually deleted, they are just flagged as so (and their entire sub tree). Users with the *restructure* permission can then restore the resource. To permanently delete an object the request header "`Permanently-Delete`" should be set to true.

C.2.2 *ID Requests*

Requests to `/id/<objectID>` works the same way as requests to `/content/<path/to/object >`.

C.2.3 *User Requests*

Requests to `/user` are used to create, edit, delete and authenticate users. A user has a list of OpenID/OAuth (see Appendix B) IDs that can be used to login with that user.

GET

`GET` requests to `/user/<userID>` are used to get information about a user, such as username and other relevant fields.

POST

New users are created with `POST` requests to `/user`.

PUT

`PUT` requests are used to modify user information or to adding/removing OpenID/OAuth IDs.

DELETE

`DELETE` requests delete users.

C.2.4 *Group requests*

Requests to `/group` are used to handle group operations. A group is an entity with a name, and ID and a list of members. Permissions for groups works the same way as permissions for objects (a user would for instance need WRITE access for a group to add new members to it).

GET

`GET` requests are used to get information about a specific group. A request to `/group/<groupID>` would return information about the group, such as name, members etc.

POST

New groups are created by `POST` requests to `/group` containing information about the group and possibly a list of user IDs.

PUT

`PUT` requests are used to modify group info, add users or remove users.

DELETE

`DELETE` requests delete groups.

C.3 RESPONSE CODES

Moraxus uses the standard HTTP status codes when responding. `404` means that resource not could be found, `401` means that access to the object is denied etc. This makes it easier to implement correct error handling on the client side.