A Framework for Service-Oriented Extensions to Ruby on Rails

S.F. Henzen

December 8, 2008

MASTER THESIS
TRESE Group
Department of Computer Science
University of Twente
The Netherlands

In cooperation with **Nedforce B.V.**

Abstract

Service-Oriented Architecture (SOA) is a relatively new architectural style in software development. A SOA separates a system's functionality into distinct units called services, which can be distributed over a network.

Ruby on Rails is a free full-stack web development framework implemented in the Ruby programming language. It allows programmers to create applications that process HTTP requests, query or update a database, and generate HTTP responses. Rails can also be used in a SOA to provide or invoke services, but practical experiences have shown that Rails support for these tasks is limited.

In this thesis we identify several limitations of Ruby on Rails with regard to Service-Oriented Architecture. Limitations are discovered through case studies. We study four medium-sized projects at a Dutch software company. Limitation were discovered mainly in functionality for invoking services, but also in functionality for providing certain types of services.

To allow for the extension of Rails' functionality for invoking and providing services, an extensible framework was developed. Protocol handlers for XML-RPC and a broad range of REST-ful services were also developed as extensions for this framework. With this framework several test cases that require extended functionality were successfully implemented. A survey has shown that application programmers rate the framework high on many quality attributes, including learnability and maintainability.

Foreword

This research was triggered by the experiences of Nedforce in the development of web-based applications. Nedforce is a Netherlands-based company that develops web applications, typically for medium-sized businesses, governments and education institutes. I would like to thank Nedforce for making this research possible. I would especially like to thank the developers at Nedforce. They have provided a lot of input for my case studies, and they have helped evaluate the framework that is proposed in this thesis.

I would also like to thank my supervisors: Luís Ferreira Pires (University of Twente), Klaas van den Berg (University of Twente), Roderick van Domburg (Nedforce) and Igor Borovykh (Nedforce) for their support.

List of Abbreviations

AWS ActionWebService, A framework for XML-RPC and SOAP in Rails

CoC Convention over Configuration, A design principle

CRUD Create, Read, Update, Delete, four common actions used to manipulate a set of data

DRY Don't Repeat Yourself, A design principle

HTTP Hypertext Transfer Protocol, A network protocol

IoC Inversion of Control, A design pattern

JSON Javascript Object Notation, A markup language

MEP Message Exchange Pattern, Pattern describing the sequence of messages exchanged in a network interaction

MVC Model-View-Controller, A design pattern

Rails Ruby on Rails, A web development framework

REST Representational State Transfer, A service design style

RPC Remote Procedure Call, A service design style

SOA Service-Oriented Architecture, An architectural style

UDDI Universal Description, Discovery and Integration, a standard for service repositories

URI Uniform Resource Identifier, A uniform syntax for uniquely identifying resources.

WSDL Web Service Definition Language, A language for defining web services

XML eXtensible Markup Language

XML-RPC eXtensible Markup Language - Remote Procedure Call, A network protocol for performing remote procedure calls using XML

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this research we examine the potential of the Ruby on Rails web development framework for use in Service-Oriented Architectures. We also propose a framework for Rails that makes it more suitable for use in Service-Oriented Architectures. This chapter briefly describes the core concepts 'Service-Oriented Architecture' and 'Ruby on Rails', and outlines the problem motivating this research. It then describes the approach and the structure of the rest of this thesis.

1.1 Motivation

Service-Oriented Architecture (SOA) is a relatively new architectural style in software development [1]. A SOA separates tasks into distinct units called services, which can be distributed over a network. A service, is a ubiquitous interface that provides access to pieces of data or functionality. Service clients can access this service to perform a task, such as querying or updating a central database [2]. Services can be combined to create Service-oriented applications. There can be services that invoke multiple other services to enact a long-running process [3]. There can also be services that are invoked by other services to coordinate an interaction [4][5][6]. These possibilities allow complex and distributed systems to be designed as interactions of services. Services can be reused in many different applications. Many standards for transporting and handling information in networked applications are based on the SOA architectural style.

Ruby on Rails, or simply Rails, is a free full-stack web development framework implemented in the Ruby programming language. It runs on top of an HTTP server. It provides functionality to process HTTP requests, generate HTTP responses (typically in HTML format), and interact with a database. In its architecture it is comparable to a subset of the J2EE framework [7], but it has several advantages. In Rails, there are no heavy toolsets, no complex congurations, and no elaborate processes. The focus is on rapid prototyping and the ability to respond quickly to changes. Therefore, Rails has been called an ‘Agile’ framework (see Section 2.2.7). Rails allows users to prototype applications by doing little more than specifying the data model. Trivial parts of the application, like CRUD interfaces for objects in the database, do not require much coding or configuration at all (see Section 2.2.6).

Support for SOA is very important for current and future projects using the Rails framework. However, limitations are encountered when using a Rails application as part of a service-oriented system. These limitations are the subject of this thesis.

1.2 Problem Description

At present, Rails has some support for providing and invoking RPC-style web services (see Section 2.3.1). It also includes functionality for providing and invoking REST-ful services. However, practical experiences have shown that Rails support for SOA is not yet sufficiently developed to allow for the development of high-quality service-oriented applications [8].

Rails was originally designed to be a platform that simply processes HTTP requests, interacts with a database and generates HTTP responses in HTML format. Since then it has evolved beyond that scope and is used for all kinds of applications. However, since it was not designed for use in service-oriented systems, it has been argued that Rails should not be used as such [9].

In this research we explore the limitations and possibilities of Rails with regard to SOA. The central problem of this research is *how to use Rails in a SOA environment.*

1.3 Objectives

The first objective of this research is to identify the limitations of Rails with regard to SOA. These limitations are discovered through case studies supported by a litarature study. The limitations we have discovered include missing functionality for invoking or providing services with Rails, and poor integration for some of the service- related functionality that is available to the Rails framework.

The second objective of this research is to develop a framework for integrating service-oriented functionality into Rails. This framework provides a common structure for extending Rails’ functionality with regard to invoking and providing services. We evaluate the framework with a set of test cases.

1.4 Approach

The research presented in this thesis consists of two parts, corresponding to the two objectives. The first part is investigating the limitations Rails has when used for developing service-oriented applications. The second part is developing a framework for integrating service-oriented functionality into Rails. This solves some of the limitations found in the first part.

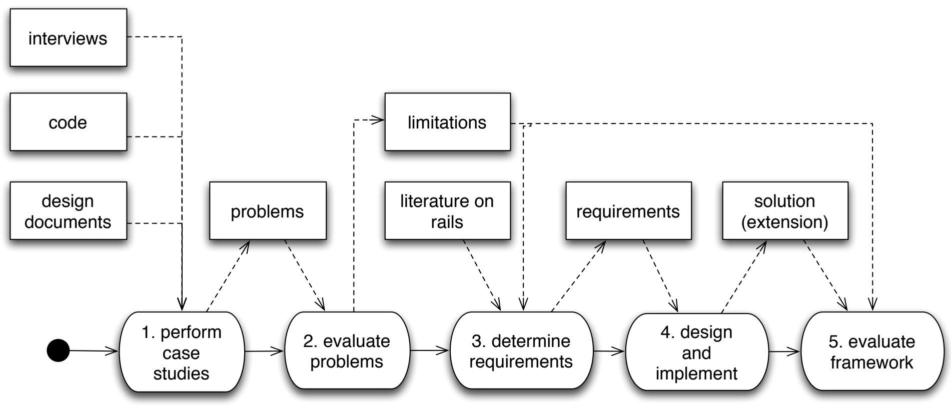

Figure 1.1: Approach (UML Activity Diagram).

1.4.1 Identifying the Limitations of Rails

First we investigate limitations of Rails when used for service-oriented applications. The approach for this is as follows (the numbers correspond to the activities in shown in Figure 1.1):

1. Conduct exploratory case studies at a Dutch software company. Here we evaluated four projects in order to get an overview of what problems may be encountered when developing service-oriented applications with Rails. We evaluated several aspects of each project: environment, functionality and development process. We sought evidence of problems encountered in each of these areas by conducting interviews and reviewing design documents and source code.

2. Evaluate the problems encountered in the case studies to find limitations of Rails with regard to SOA. An important consideration here is whether a problem is caused by a limitation or by a misuse of Rails. Problems caused by misuse may also indicate that a certain part of the Rails platform is unnecessarily complicated or poorly structured. Another important consideration is whether the problem is still relevant. Since Rails is a young platform, it is always evolving and changing.

Case studies on products delivered a year ago may yield problems that where already solved or mitigated in more recent versions of Rails.

1.4.2 Developing a Framework for Service-Oriented Extensions to Rails

The second step is to develop a framework for integrating service-oriented functionality into Rails. The approach for this is as follows. (the numbers correspond to the activities shown in Figure 1.1):

3. Determine initial requirements for the framework. In this step we select the limitations we want to solve with our framework. From these limitations the initial functional requirements are derived. Additional requirements come from how we decide to integrate the framework into Rails.

4. Design and implement the framework.

5. Evaluate the framework. In this phase we develop several test cases. We look at the limitations we found and evaluate how these can be mitigated using the framework. We also involve programmers that have experience developing service-oriented applications using Rails and let them rate the framework with regard to several quality attributes, including usability, extensibility, maintainability and simplicity.

1.5 Structure

The thesis is structured as follows:

Chapter 2 describes the background for this research. It gives an overview of concepts related to Service-Oriented Architecture. It explains the architecture and workings of Ruby on Rails. Current support for SOA in Rails is also described.

Chapter 3 describes the first phase of the research In which we conduct case studies to identify limitations of Rails with regard to SOA. It contains the results of the case studies, the evaluation of the case studies and the limitations of Rails that were discovered.

Chapter 4 describes the second phase of the research, in which we propose a framework for integrating service-oriented functionality into Rails. It starts with the requirements and architecture for the framework we propose. It then describes in detail how the framework was designed and implemented.

Chapter 5 describes a set of test cases that were implemented using the framework. These test cases demonstrate the features and workings of the framework.

Chapter 6 contains the evaluation of the framework. It describes how the framework compares to several other technologies currently available for providing and invoking services with Rails. It also presents a survey that was conducted to gather feedback on the framework from experienced Rails developers.

Chapter 7 summarizes the conclusions.

Chapter 2

Background

This chapter introduces the basic concepts that are important in this thesis. In Section 2.1 the Ruby programming language is introduced. Section 2.2 gives an overview of the Ruby on Rails web development framework. Section 2.3 defines Service-Oriented Architecture, and gives an overview of the main technologies that are currently available for it. How the technologies described in Section 2.3 are currently supported in the Rails platform is discussed in Section 2.4.

2.1 Ruby

Ruby is a general purpose object-oriented language [10]. It was inspired by Perl and Smalltalk. Ruby originated in Japan during the mid-1990s and was initially developed by Yukihiro Matsumoto. There is currently no full specification of the Ruby language, so the original implementation is considered to be the reference. As of 2008, there are a number of upcoming alternative implementations of the Ruby language, including YARV, JRuby, Rubinius, IronRuby, and MacRuby.

Ruby was designed with an emphasis on human needs rather than computer needs. It focusses on maximizing programmer productivity, rather than computer efficiency.

Ruby is a fully object oriented language. There are no primitive types, like Java's integer, boolean, etc. There are only objects. An integer would be an object of the Integer class.

Ruby is an interpreted language. There is no distinction between compile time and runtime. The JRuby and IronRuby implementations provide just-in-time compilation functionality. The other implementations use single-pass interpretation.

Ruby is dynamically typed. This means that classes can be dynamically defined and redefined at runtime. Through reflection programmers can alter the behaviour of a class at any point during the execution of a program.

Changing the behaviour of a class changes the behaviour of all objects of that class. Behaviour of built-in classes, like Integer or String, can also be modified dynamically.

Methods in ruby can be treated as objects. A method can be passed to another method as a parameter, and methods can return new methods as a result. Ruby has been called 'multi-paradigmatic' because it allows programmers to use constructs from functional programming, specifically higher-order functions.

2.2 Ruby on Rails

Ruby on Rails (Rails) is a full-stack open source web development framework implemented in the Ruby programming language. It allows programmers to create applications that process HTTP requests, query or update a database, and generate HTTP responses. Figure 2.1 shows the various components of a typical web application that uses Rails. Rails relies on a webserver for the handling of incomming HTTP connections. It consists of four main modules: Dispatcher, ActionController, ActiveRecord and ActionView. Rails relies on a database for persistent storage of data. Rails can work with various databases and webservers.

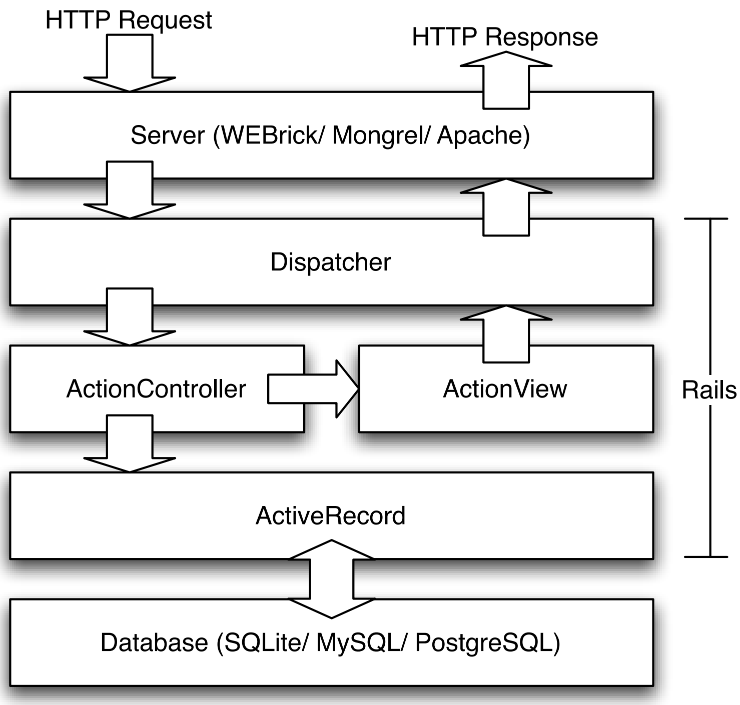

Figure 2.1: Rails stack, arrows indicate typical HTTP interaction

2.2.1 HTTP Processing, Response generation

The Rails framework receives HTTP requests from a webserver. When a request comes in, it is dispatched to an ActionController instance. ActionController can give instructions to query or manipulate data in a database through ActiveRecord. It can also give instructions to generate a response through ActionView.

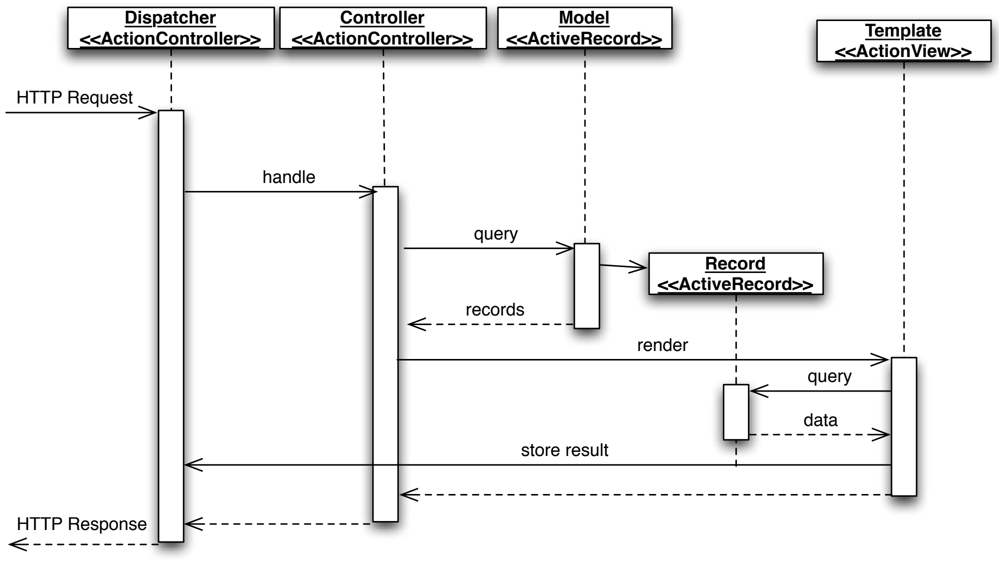

Figure 2.2: Typical HTTP request

Figure 2.2 shows a typical processing of an HTTP request by the ActionController. The ActionController class controls the main event loop. Using a set of routing rules, a request is mapped to a method in an ActionController instance based on its URL and its HTTP action (GET, PUT, POST or DELETE). ActionController automatically calls this method. This method is defined by a programmer. It can use information from the request to access the database and prepare data for response generation. It can then either explicitly call a method that renders the response (as shown in Figure 2.2) or just terminate, in which case the ActionController takes control again and decides how to generate a response using certain conventions. ActionView is the module that generates the response. This is done through a templating system. The programmer can define a template for the response in a Domain-Specific language (e.g. ERB, RJS) [11]. The resulting response can be in many formats (e.g. HTML, XML, etc.). When the response is generated, the ActionController takes control again and passes it to the webserver.

2.2.2 The Inversion of Control Pattern

Rails allows programmers to define how requests are processed through Inversion of Control (IoC) [12]. IoC means that instead of a programmer specifying the series of operations to be performed in response to a request, a programmer rather registers desired responses to particular events, and then lets Rails take control over the precise order and set of events to trigger. This concept is used by many popular web application frameworks (Spring, PicoContainer, HiveMind [13], EJB [14]). In Rails, users specify responses to HTTP requests in ActionController and ActionView. There are also hooks to register responses to data manipulation events in ActiveRecord.

2.2.3 The Push MVC Pattern

Rails follows the Model-View-Controller (MVC) architectural pattern [15], which is quite common in web application development frameworks [16]. MVC prescribes that the application state should be contained in a model. In Rails applications the application state is usually stored in a database. It can be accessed through the ActiveRecord module, which performs Object-Relational Mapping [1]. This ORM and the database together provide the model component. ActiveRecord is part of Rails, but can also be used independently.

MVC prescribes that the view should be the graphical representation of the model. There are two types of MVC: Push and Pull. In Pull MVC the view component accesses the model component directly and 'pulls' information from it to display to the user. In Push MVC the controller is responsible for passing data to the view component. Data is 'pushed' to the view by the controller. In both types the controller is responsible for performing operations that alter the model. Rails uses push MVC.

2.2.4 The DRY Principle

Rails adheres to the DRY (Dont Repeat Yourself) principle. This means that every piece of functionality, if possible, should only be present in one place. This is closely related to the principle of Separation of Concerns (SoC) and concepts like Aspect Oriented Programming. An example of how Rails applies the DRY principle is by allowing programmers to define filters. A request can pass through a filter before it is dispatched to a more specific action method, thus providing a mechanism to specify behavior that is common to many actions in a single location.

2.2.5 Creating ActiveRecords

Rails is designed adhering to the Convention over Configuration (CoC) principle. This means that useful default behavior is provided, and this only needs to be changed when necessary. The following ActiveRecord definition gives an example of this.

```
class Project < ActiveRecord::Base
   has_many :milestones
end
```

Through CoC, Rails knows that this class represents the 'projects' table in the database, and that an instance of this class represents a row in that table. It gives instances of this class accessors for all the attributes of this table. 'has_many :milestones' means that projects have a one-to-many relation with milestones in the database. Rails gives instances of this class an accessor for an attribute 'milestones' that can be used to retrieve all milestones from the database that belong to this project, and add new milestones to the project.

2.2.6 Using Generators

Generators are scripts that allow users to quickly develop common functionality. For example, to create an application to manage person data, users just need to run the following Ruby script:

```
generate scaffold Project name:string leader:string deadline:date
```

Running of this script generates:

1. A new script that can be used to create a table named 'projects' with the specified properties in the database.
2. An ActiveRecord class named 'Project' for this table.
3. An ActionController and ActionViews that allow the user to create, read, update and delete data in this table (A CRUD interface, [17]).

The classes generated by this script form a working piece of functionality that can be used right away to interact with the data in the database from a web interface. The standard classes can easily be modified to describe new interactions beyond simple CRUD actions, for example assigning a person to a project.

Test support is built-in. To test applications, Rails users define methods that invoke functionality of the application and assert whether certain post-conditions are met. A skeleton for these methods is automatically generated when generators are used to create classes.

2.2.7 Agile Software Development and Rails

Rails is especially suitable for Agile Software Development methods. Agile is a common name for a set of software development methods that aim to provide a lightweight and flexible development process. Agile methods offer an alternative to traditional plan-driven methods, which are not used as intended because they are too mechanistic and rigid [18][19]. There are many agile methods, including: (from [20])

1. Scrum development process [21]
2. Extreme Programming (XP) [22]
3. Pragmatic Programming (PP) [23]

The authors of these methods have also co-authored and undersigned the ‘Agile Manifesto’ [24], which contains some of the common characteristics of agile methods. These common characteristics include:

1. An aim to deliver working software fast and frequently, preferably every couple of weeks.
2. A focus on working software as the main measure of progress and the authoritative source of documentation.
3. Allowing for changing requirements at any stage of the development process.
4. Employing close customer-developer interaction.
5. Employing face-to-face conversation as the main method of conveying information to and within a development team.

In [25] the characteristics that agile methods aim to be simple, easy to learn and modify and sufficiently documented were added.

Rails does not prohibit the use of any development methods, including non-agile methods. It does not prescribe a process. It does, however, support the practices and processes found in many agile methods.

CoC and DRY help in keeping the size of the codebase small. Changes in the later stages of a development process thus impact less code in Rails projects than they would in other projects. This makes it easier to respond to last-moment changes, an ability that is required in agile methods.

Generators allow developers to deliver working software early in the development life-cycle. This can be leveraged to improve customerdeveloper interaction. Note that developers can also choose to forego generators and build their applications from the ground up following a detailed design.

Test-Driven Development (TDD) is a practice that is incorporated in several agile methods, including XP. TDD prescribes that tests are written before the code. Rails has built-in support for testing, which makes TDD easy. However, Rails does not require that its testing capabilities are actually used.

2.3 Service-Oriented Architecture

In this thesis we define a Service-Oriented Architecture (SOA) as an architectural style for a system in which multiple applications interact by providing and using services . An application provides a service when it exposes parts of its functionality as an interface that can be used by all other applications in the system [2]. An application invokes a service by sending messages to it and/or receiving messages from it. Applications that provide or invoke services can be distributed over a network.

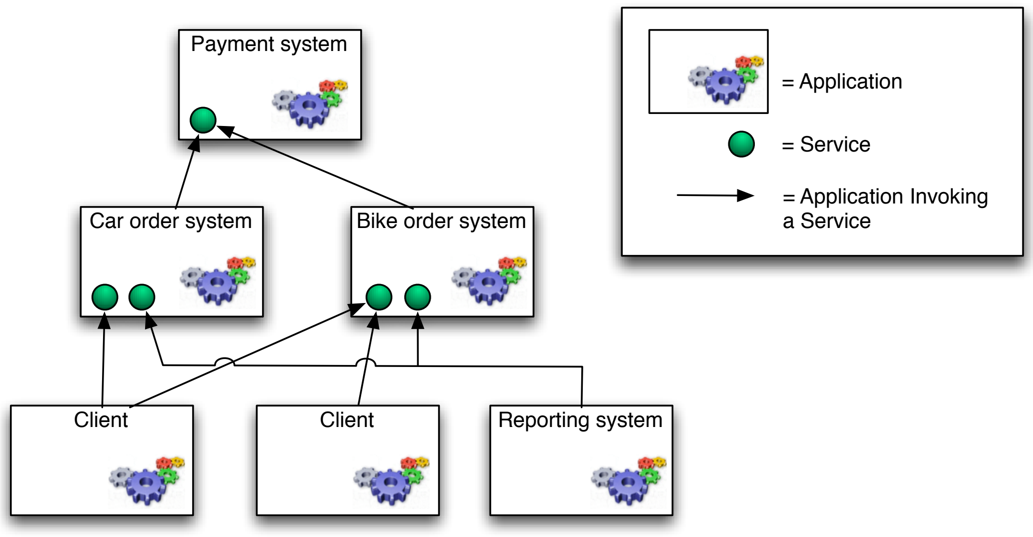

Figure 2.3: Example of a Service-Oriented Architecture

Figure 2.3 shows an example of a SOA. The system in this figure belongs to a hypothetical company that sells both cars and bikes. It has two different order processing systems for this. Clients can order cars or bikes via the respective order processing systems through a service. The order processing systems both use the same payment processing service of a payment system. The company also owns a reporting system that collects sales data and extracts statistical information from it. The reporting system collects sales data through another service that the order systems provide.

Applications can provide multiple services, use multiple services, and an application is not limited to either invoking or providing services. A distributed Service-Oriented Architecture has several advantages over a monolithic architecture:

- Greater reliability: when the application providing the car order service crashes nothing else breaks, and clients can still buy bikes.
- Better modularity: When the reporting system is replaced nothing else needs to be changed
- Better extensibility: Other order processing systems using the same payment processing system can be added without changing anything else.

2.3.1 Service Interaction Styles

There are three important interaction styles a service can follow. These are message-oriented, RPC and REST. Each of these styles imply different message semantics and a different message exchange pattern.

A Message-oriented Service offers a set of endpoints. Each endpoint can have a certain input message format and a certain output message format. It is not necessary to output a message for every input message, nor is it necessary to receive an input message before every output message. Any message format is allowed. Application- or domain-specific protocols can be layered on top of message-oriented services. This has been called the framework approach. Message-oriented interaction leaves a lot of decisions with regard to message semantics and message exchange pattern to the application developer [26].

RPC stands for Remote Procedure Call. An RPC-style service exposes a set of procedures. The client ‘calls’ a procedure and the service executes the procedure and returns the result. A request to an RPC-style service consists of a procedure name and a set of parameters. A response can be either the result of the procedure referenced by the request or an error. Note that a request should always be followed by a response, and that there are no unsolicited responses.

RPC-style has more constraints on message format and interaction pattern than message-style. This allows for more meaningful processing by a common software framework. Having more constraints makes a common framework more complicated, but it can also do a lot more for the application programmer. For example, the procedure name could be used to automatically call a corresponding method.

REST stands for Representational State Transfer. A REST-style service exposes pieces of data or functionality as resources. Resources are queried and manipulated through a constrained set of well-defined operations. This set of operations often includes the CRUD operations (Create, Read, Update, Delete). A service client can interact with a REST-style service by requesting an operation on a resource. An operation may require a representation of a resource as input. It may also yield a representation of a resource as output.

A protocol supporting REST has a constrained set of operations, whereas a protocol supporting RPC allows programmers to define their own set of operations in the form of function calls. Providing a constrained set of operations allows for more meaningful processing by a common software framework. For example, a request that merely retrieves a representation of a resource can be cached by the client or an intermediary. Also, any request that changes a resource is potentially harmful, so it can be filtered by a firewall.

2.3.2 Service Technology

There are several technologies supporting the various service interaction styles. In this section we discuss the most common technologies, as shown in Table 2.1

Interaction Style	Supporting Technology
REST	HTTP and URI
RPC	XML-RPC or SOAP
Message-oriented	SOAP

Table 2.1: Service technologies

HTTP stands for Hyper Text Transfer Protocol. It supports the REST-style by providing a constrained set of operations to query and manipulate resources through the exchange of representations. It provides several operations, including the four CRUD operations. HTTP uses Uniform Resource Identifiers (URI, [27]) to uniquely address resources. Table 2.2 shows the contents of an HTTP request, and how it can be mapped to the REST style. Table 2.3 shows the CRUD request methods and their semantics. This table shows the *intended* use of these methods. The method semantics shown here can be ignored by applications.

Message Contents	Mapping to REST-style
request method	Token indicating the desired operation on the requested resource
URI	Unique identifier of a certain resource
header fields	Additional information required to perform the desired operation, including the format of the enclosed representation
body	A representation of a resource

Table 2.2: HTTP request contents

HTTP provides a uniform interface to query and manipulate resources, but whether this interface is used as intended depends on application developers. Another thing that HTTP leaves to application developers is how the

Request method	Semantics
GET	retrieve whatever information is identified by the Request-URI
PUT	request that the enclosed entity be stored under the supplied Request-URI
POST	request that the server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI
DELETE	requests that the server delete the resource identified by the Request-URI

Table 2.3: HTTP methods and their semantics

body of the request is used. According to REST the body should contain a representation of a resource, but in reality anything can be transported in an HTTP body. HTTP does not prescribe how the body should be encoded or what information about resources it should contain. HTTP can be used to provide REST-style services when the HTTP actions are used as intended and an encoding format (e.g. XML) and representation semantics are defined for the HTTP body. We call services that use HTTP in this way REST-ful services.

There are many examples of public REST-ful services available, such as Amazons' Simple Storage Service [28] or Google's API's [29].

XML-RPC is a protocol that supports the RPC interaction style. An XML-RPC request contains a procedure name and zero or more parameters. A response is either a result of a procedure or an error. XML-RPC prescribes an XML-based format for encoding requests and responses. The format includes encoding of primitive data types, such as strings and arrays. An XML-RPC request consists of a procedure name and a list of parameters. An XML-RPC response contains either a list of results or an error.

XML-RPC uses HTTP as a transport mechanism. The XML-RPC message is contained in the body of the HTTP message. Figure 2.4 shows a request to a service using XML-RPC. The XML-RPC layer encodes a procedure name and parameters and sends them to the HTTP layer as the body of a request. It usually employs a single URI to identify the entire service. The HTTP action is always POST. At the service side, the HTTP URI and action are not used. All relevant data is in the body of the request. This body is passed to the XML-RPC layer which decodes it and extracts the procedure name and parameters. [30]

SOAP is an XML-based protocol for message-oriented interaction, with features for RPC-style interaction. SOAP once stood for Simple Object Access Protocol, but this acronym was dropped with Version 1.2 of the standard, as it was considered to be misleading. SOAP was originally based on XML-RPC, and version 1.0 and 1.1 supported RPC-style interaction only.

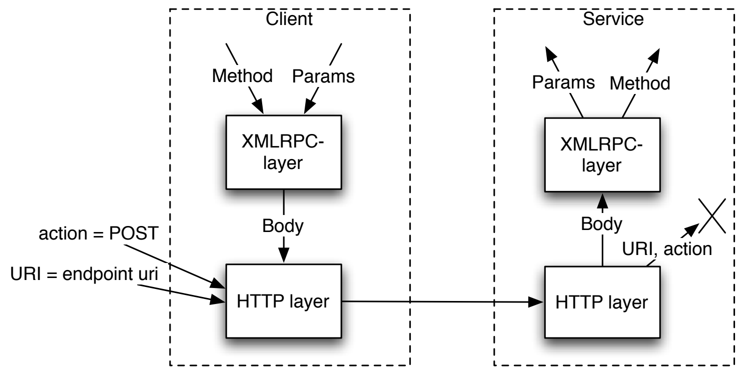

Figure 2.4: XML-RPC request layered on top of HTTP

Version 1.2, the most recent version at the time of this writing, supports message-oriented interaction. A SOAP message contains a header and a body. The SOAP header contains control data and metadata about the contents. The SOAP body contains the message. This message may be in any XML-based format. SOAP 1.2 defines a message format for RPC-style interaction, but usage of this message format is optional. [31], [32]

Services that use SOAP can be described in Web Service Description Language (WSDL). WSDL is an XML-based language that provides a syntax for describing services. It describes services in terms of endpoints with input and output messages. An endpoint description includes a 'style' attribute, which describes the message format. The styles used to describe SOAP services are RPC/literal and document/literal. RPC/literal messages conform to SOAP's RPC message format. Document/literal messages can contain any message in XML format. RPC/literal messages are thus a subset of document/literal messages [33]. An endpoint can also have a certain Message Exchange Pattern (MEP). A MEP describes the pattern of messages required to interact with a service (e.g. in-out for request-response). With the document/literal style and MEPs WSDL can describe the full range of message-oriented SOAP interactions. WSDL documents can be used to automatically generate stubs and skeletons for client programs. They can also be used to advertise a service.

WSDL documents can be retrieved through servers conforming to the Universal Description, Discovery and Integration (UDDI) standard. A UDDI-compliant server acts as a registry for services and can be accessed through SOAP [34]. SOAP, WSDL and UDDI are referred to as the WS basic technologies [35].

There are many other standards that can be used with SOAP to provide features like authentication, transactions, sessions, etc. These standards are usually prefixed by 'WS-' (WS-Coordination [4], WS-Atomictransaction [5], WS-BusinessActivity [6]).

SOAP uses an internet application-layer protocol as a transport protocol. HTTP is the most common protocol for transporting SOAP messages. This is also prescribed by an interoperability standard called 'basic profile'[36]. HTTP limits interaction patterns to request-response only, thus not supporting the full range of possible message-oriented interaction patterns.

Figure 2.5 shows a request to a service using SOAP over HTTP. This is similar to the XML-RPC request in Figure 2.4. One difference is that the SOAP request contains a header that can contain additional processing and routing information. Another difference is that the SOAP body can contain an XML document in any format, not just an RPC-style procedure name and parameters.

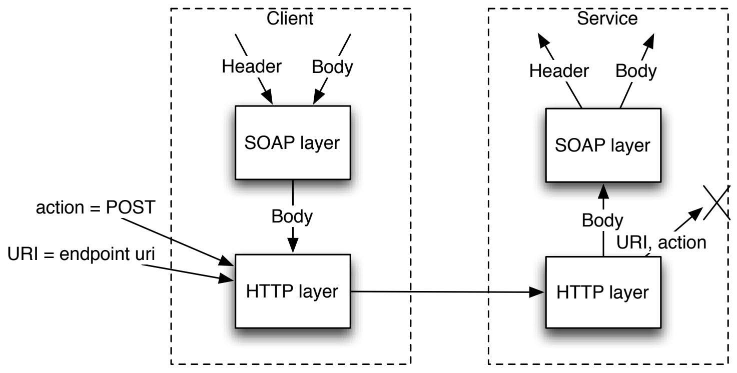

Figure 2.5: SOAP request layered on top of HTTP

Using HTTP as a transport protocol, as done in case of SOAP and XML-RPC, has several disadvantages:

1. it makes implementing security policies through firewalls more difficult, because deeper packet inspection is needed to determine what a request is trying to do.

2. It decreases performance, because both the HTTP and the protocol message need to be encoded and decoded.

3. In case of SOAP, it increases the complexity of the code unnecessarily, because some features in the HTTP layer are duplicated in the SOAP layer (Such as addressing [37]).

4. In case of SOAP, request-response becomes the only possible interaction pattern, although other patterns are allowed by the SOAP protocol.

2.4 SOA in Rails

In this section we focus on how to provide REST-ful services over HTTP, and how to provide services using the XML-RPC and SOAP protocols in Rails. Figure 2.4 shows the six possible tasks Rails could perform with regard to these protocols. It also shows the names of the components that are commonly used for that task. This section goes through each of these tasks, and explains how they can be performed in Rails.

		Protocols		
		HTTP	XML-RPC	SOAP
Roles	Invoke	ActiveResource, Restclient, HTTParty, NET::HTTP, Open-uri	AWS, xmlrpc4r	AWS, soap4r
	Provide	Rails basic functionality	AWS, xmlrpc4r	AWS, soap4r

Table 2.4: Components available for SOA classified according to tasks

2.4.1 Providing REST-ful services

Rails is very suitable for providing REST-ful services over HTTP. It was originally designed to provide HTML websites only. It supported HTTP GET and POST operations on URI's. Now Rails has built-in functionality to provide almost any kind of REST-ful service over HTTP. It supports the full set of HTTP operations, including PUT and DELETE. In accordance with the REST-style, a Rails ActionController can be seen as managing a certain resource or set of resources. Requests can be routed to the controller based on their URI, and further routed within the controller based on their HTTP operation. The controller can query or update a resource and generate a response. The response can be human-readable (image, HTML, plain text) or machine-processable (XML, YAML, JSON).

2.4.2 Invoking REST-ful services

Rails also includes functionality for invoking certain REST-ful services. ActiveResource is a Rails module that allows users to work with remote resources over HTTP as if they were local objects. It translates method calls to HTTP requests. It assumes that representations come in the form of a certain XML format. The use of ActiveResource is limited to a subset of REST-ful services. The method-to-request translation does not allow programmers to perform arbitrary operations on arbitrary resources. It expects a certain result in a certain encoding format from the service for each operation. However, a REST-ful service can respond with a broad range of representations of resources encoded in any encoding format.

When creating a Rails application it is quite easy to provide a service that the ActiveResource module of another applicaiton can use. When you use a generator to scaffold an entity you automatically provide a service that manages an ActiveRecord and has both an HTML representation and an XML representation that ActiveResource can use.

Alternative options for invoking REST-ful services are restclient [38] , HTTParty [39], open-uri and the rails core library NET::HTTP. These libraries are not dependent on Rails.

Restclient is a stateless library that can be used to perform any operation on any remote resource, using any representation as input. The input can be provided as raw data or as a set of key-value pairs. The key-value pairs are then URL-encoded as if they were data from an HTML form. The HTTP response is returned as a set of response headers and the unprocessed data from the response body. It also handles HTTP response that indicate an error or a redirect.

HTTParty provides a framework for creating objects that represent a service in the program, analogous to ActiveResource instances. These objects can be configured to generate certain HTTP requests, as well as to process the HTTP response in a certain way. HTTParty supports XML and JSON formats.

Open-uri is a simple stateless library that can perform an HTTP GET operation when provided with a URL. It also includes functionality for HTTP basic authentication.

NET::HTTP is a stateless library for performing HTTP requests. It provides no means of processing response data, generating request data or handling errors. It just provides a set of functions that can be used to generate and send any HTTP messages based on a set of parameters. Both RESTclient and HTTParty use this library.

2.4.3 Providing and Invoking Services using XML-RPC and SOAP

To provide or invoke a service using XML-RPC or SOAP the Rails application programmer has several options. The most common option is to use ActionWebService [40]. This was a part of the Rails framework until Rails 2.0. Now it is no longer supported. You can however still use it in Rails 2.0 projects with some tweaking. ActionWebService is a module for providing and invoking both XML-RPC and SOAP services. It is limited to RPC-style interaction.

ActionWebService allows programmers to define procedures and their parameters in an API file. An API file is similar to a java interface, it only specifies signatures of procedures. The actual handling of a procedure call is done by an ActionController method. ActionWebService provides the procedures in an API file through an XML-RPC and a SOAP interface. When a procedure call comes in it is first checked against the API file to see wether its valid. Then it is sent to an ActionController method. Which ActionController method depends on certain predefined rules. The return value of the ActionController method is used to generate a response. ActionWebService can also be used to invoke XML-RPC and SOAP services on other hosts by using an API file. In this case the API file forms a stub for the service. An API file can also be used to generate a WSDL file that describes the service it represents.

Another option for using XML-RPC or SOAP is to use the Ruby libraries that underlie ActionWebService. These are called xmlrpc4r [41] and soap4r [42]. Both can be used to provide or use services independently of Rails. They can function as full-stack standalone XML-RPC or SOAP servers or clients. They are not integrated into the Rails platform, but they can be used from it.

There is also a port of the popular WSO2 web services framework to the Ruby language, but it is still highly experimental and there is very little documentation available [43]

2.5 Summary

Ruby is a general purpose programming language that features dynamic typing and higher-order functions. Ruby on Rails is a framework for developing web applications implemented in Ruby. It includes many tools to process HTTP requests and generate HTTP responses. It also includes a database abstraction layer.

Service Oriented Architecture (SOA) is an architectural style in which services are the main means of interaction between applications. A service is an interface to an applications' data or functionality that all other applications in a distributed system can access. In this chapter we have described three styles of services:

- Message-oriented: A service exposes a set of endpoints to and from which messages can be sent.
- Remote Procedure Call (RPC): A service exposes a set of procedures, which can be called by a client.
- Representational State Transfer (REST): A service exposes a set of resources on which a fixed set of actions can be performed.

We have also described three important protocols for providing and invoking services:

- HTTP (for REST)
- XML-RPC (for RPC)
- SOAP (for RPC and Message-oriented)

Rails' basic functionality includes providing REST-ful services over HTTP. It can also invoke REST-ful services that follow certain Rails-imposed conventions. To invoke or provide other service types Rails needs external tools.

Chapter 3

Case Studies

This chapter presents the case studies we performed to identify limitations of Rails with regard to SOA. The case study approach will be discussed in Section 3.1. In this section the aspects of each case we explore and the sources of evidence for problems we consider are described. The constraints for selecting a case are also described.

Section 3.2 to Section 3.5 contain the reports of the four case studies we performed. Each report has several sections, corresponding to the aspects described in Section 3.1.

Section 3.6 gives an overview of the discovered problems. Problems are evaluated in the context of recent developments. They are classified according to the various SOA tasks described in Section 2.4.

3.1 Case Study Approach

The case studies are designed and implemented based on the principles outlined in [44]. This book provides a framework for defining a case study. Following this approach, we define the following aspects of the case studies here:

- The goal of the case studies.
- What is considered a case.
- The propositions.

The goal of the case studies is to find limitations of Rails when used in SOA's.

A case in this research is defined as the project of developing a single product for a specific customer. A product is a working piece of software that has a set of features that together achieve a set of consistent goals for the customer. The product should be 'finished', meaning it was at some point accepted by the customer, and after that has been operational for some time. We consider only cases that are part of a SOA as defined by Section 2.3.

There is only one proposition preceding this case study, which is: When Rails is used to perform a SOA task, limitations are encountered. This gives the case study a broad scope and an exploratory character.

For each case, we look at the following aspects:

- Background. How the need for the system arose, and what the stakeholders of the system are.
- Purpose. Here we examine the functions of the final product. This serves as an orienting step. It provides a context in which other aspects can be explored. It serves as a guideline for the rest of the case study, ensuring that we dont overlook important functions and related integration problems.
- Environment. Each project is part of a SOA. This implies that there are other systems that interact with it. In this part, we list per interaction how it works and what, if any, problems still exist in it. In every case there is an interaction with the web browser on the client machine. This interaction is not described here, since it merely provides a user interface to the product.
- Data structure. Here we describe how data is structured inside the product.
- Workflows and interactions. This ties environment and data structure together. It describes workflows in which operations are performed on internal data entities and service requests are performed to external systems.
- Development process. Here we examine the process by which the system and its links to other systems were developed. We focus on the problems encountered during the development process, and the solutions used. Most of the problems will be described in this section.

These aspects also correspond to the different sections of each case study report, except for background, which is used as an introduction.

Evidence for limitations encountered in each of these sections is collected from multiple sources. Data from the following sources is collected:

- Design documents
- Source code
- Interviews

From the documents and the source code a diagram of the environment and the data structure of the product is constructed. The programmers that worked on the product are involved with constructing these diagrams to ensure their correctness. These diagrams are then used as the basis for structured interviews with programmers. For each connection to another system, we ask programmers about the following issues:

- How difficult it was to implement.
- Whether problems were encountered in implementing the connection.
- How these problems were solved.

3.2 Portal

Portal was developed for a firm that sells digital publications. A digital publication is a special type of digital document that can be read like a book with a flash tool. It can contain dynamic content such as audio and video. It can also contain simple interactive elements. The publishing firm had a website in place to make these publications available online. It allowed readers to subscribe to these publications, pay for them, and access them with the flash tool. At some point, the need arose to allow whole companies access to publications. At the business level, this was implemented with licenses. A company could buy a license to a digital publication, which allowed that company to subscribe a limited amount of its employees to that publication. Portal is the system that was implemented to allow the publishing firm and the companies to manage these licenses and subscriptions. It was chosen as a case because of its complicated interaction with existing systems.

3.2.1 Purpose

Portal has several purposes:

- It allows the publishing firm to manage companies and their licenses.
- It allows companies to manage subscriptions for their employees
- It gives employees of companies an alternative way to access digital publications. When employees are subscribed to a publication they can access that publication either through the original website of the publishing firm or through Portal.

3.2.2 Environment

Although users and subscriptions are managed in Portal, Portal is not the authoritative source for users and subscriptions. The authoritative source for this data is the publisher system on the publisher server (see Figure 3.1) The portal synchronizes users and subscriptions with this system by invoking an RPC-style service using XML-RPC. The publisher system manages all users that use digital publications, whereas the portal only manages a subset of these users. This means that the client company users in Portal are also users on the publisher system.

The digital publication data resides in the publisher library. It is exposed to readers from this system through a REST-ful service. No specific data about digital publications is present in Portal.

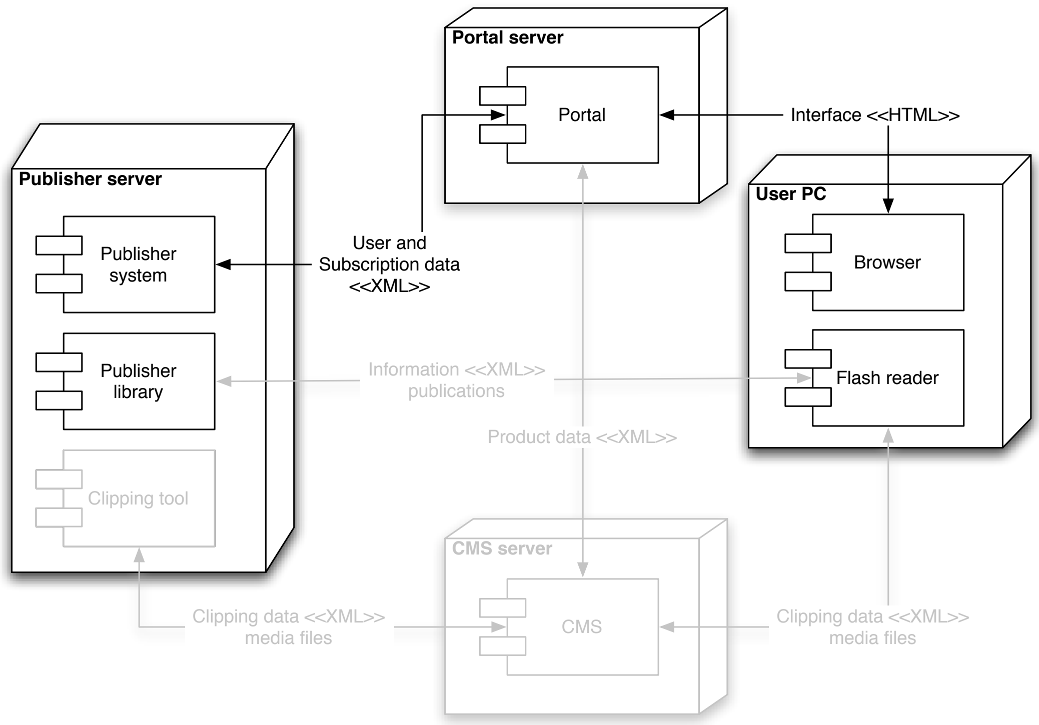

Figure 3.1: Publisher environment from portal perspective

3.2.3 Data Structure

Figure 3.1 describes the main internal data structure of the portal system. This schema was deduced from the Portal database. The portal manages companies' licenses to use products. A product is a digital publication or a set of digital publications. When a company has a license to use a product, it can create several subscriptions for its employees. A subscription is a permission to access a digital publication. An employee is a type of user. When an employee or subscription is created it is synchronized with the publisher system, so that employees that have a subscription in the portal system can access the associated digital publication in the publisher system. A role represents the type of the user (e.g. client company employee). Next to Client company Employees there are also two other types of users. These are also shown in Figure 3.3:

- Portal Administrators. These users work at the publishing firm. They can create new users, licenses and companies.
- Company administrators. These users work at companies, just like the employees. They can create new employees and subscriptions for these employees, within the limits of what the license of their company allows.

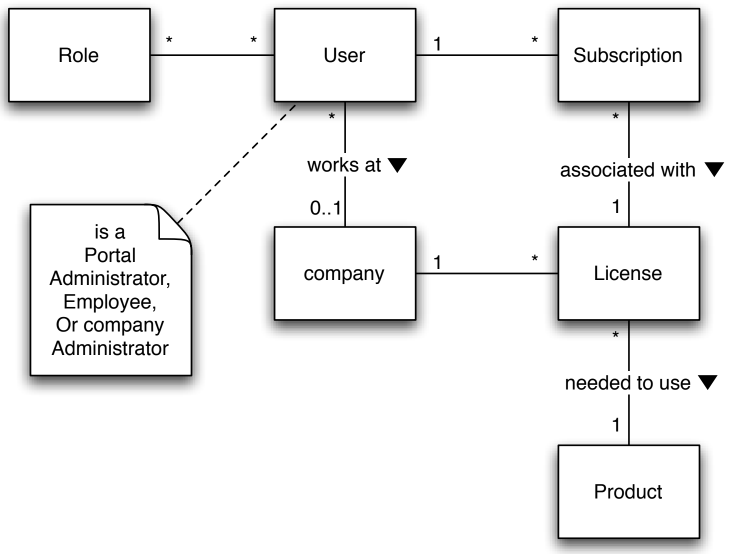

Figure 3.2: Portal data structure (UML Class)

3.2.4 Workflows and Interactions

The overall workflow in the system is as follows:

- The portal administrator adds the company and creates a license for that company on a certain product.
- He then creates an account for a company administrator.
- The company administrator logs in and creates accounts and subscriptions for the other employees at its company. The number of subscriptions he can create is limited by the attributes of the license.

- Company employees can log in and use the content to which they were subscribed by the company administrator.

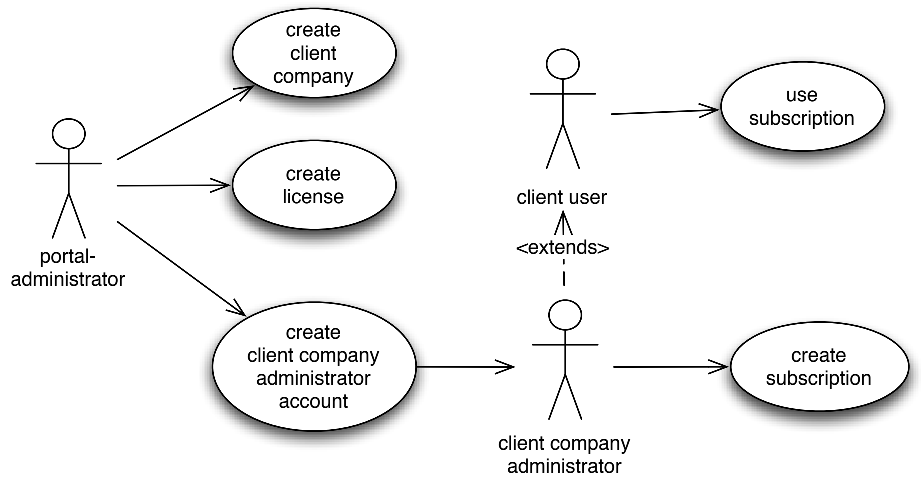

Figure 3.3: Portal roles (UML Use Case)

When an employee account is created it needs to be synchronized with the publisher system. This is required to allow the employee access to digital publications stored in the publisher library. Synchronizing employees works as follows:

- When an employee account is created in the portal it is not given a password, so it cannot log in. The portal sends an XML-RPC request to the publisher system with the employees information. This request is handled by a background process.

- From this request, the publisher system creates a new user account in its database. It generates a password for this user account and returns it to the portal. The portal receives the password and stores it. From that point onward the employee can log in to the publisher system and the portal with the same account.

- The portal administrator can later change this password in the portal. This triggers another XML-RPC request to the publisher to update the password.

- When the password is changed in the publisher system, it is not synchronized with the portal. Thus, in that (illegal) state, the client company user can login to the portal, but it cannot access the publications of the publisher system. The administrator can fix this by also changing the password in the portal, which causes it to sync with the publisher system.

- If a user (identified by e-mail address) already exists on the publisher system when it is created in the portal then that user's password is returned by the initial XML-RPC request. The employee can then login to the portal like any other user using the password it already has.
- A user is never deleted in the publisher system. At most, it is put on non-active. When a user is deleted in the portal, it can no longer log in to the portal, but it may still be able to log in to the publisher system.

When an employee account is created, the company administrator may add subscriptions to the account in the Portal system. These are also synchronized with the publisher system. The subscription data is needed in the Portal system to limit the amount of accounts per company. The subscription data is needed in the publisher system to determine if a user is authorized to access certain content.

The subscription and user data duplication in both the portal and the publishers information system is widely considered an anti-pattern in SOA [45]. The problems that it causes are clearly visible in this case. It requires complex synchronization actions. Due to limitations in the publishers system complete synchronization is not possible and the system may reach an illegal state.

An employee has access to digital publications through an iFrame [46]. The portal embeds a URI with a hashed e-mail address and password in an iFrame on an HTML page, which the employee's browser uses automatically to render the publisher website in a certain location on the site. The employee can interact with the library part of the publisher application directly without any need for interaction with the portal.

Using a hashed e-mail address and password in an URL is not very safe, however. Since the hashed information is sent to the user, and the user also knows the original information, it may be possible for him to determine the hashing algorithm.

There are several other connections and entities in Figure 3.1, outlined in gray. The CMS component is one of these. CMS stands for Content Management System. This component provides multimedia content for use in digital publications. It is the subject of the next case study.

3.2.5 Development Process

In theory it would have been easier to develop the functionality of this portal system in the publisher system, but in practice this was not the case. The publisher system was not flexible enough to add these features (licenses, client companies and client company administrators) in the allowed time of 3 weeks. Therefore, an external company was contracted to implement this portal approach. The drawbacks of this approach for the publishing company included:

- Need for an extra server process and database.
- Additional point of failure.
- Cost overhead of outsourcing.
- Need to develop complex synchronization functionality.

However, despite these drawbacks, contracting an external company to develop a portal solution was still a viable option.

Interacting with the publisher system from the portal caused several problems. The user synchronization service was invoked through ActionWebService (see Section 2.4).

A limitation of ActionWebService that was encountered in this project is that it does not support basic HTTP authentication. This was resolved by using Ruby's reflection features (see Section 2.1) to change the ActionWebService module's implementation at runtime.

Another problem encountered during the development was that the XML structure used to communicate with the service of the publisher system was not properly documented. Sometimes the types or names of arguments were wrong, and the programmers had to try several things before communication was achieved. Also tied into this is that AWS and the framework of the publisher system use a slightly different representation of the XML-RPC type 'array'. Therefore, the two systems could not communicate naturally.

Another problem was the set of procedures the publisher system exposed. The behavior of these procedures was sometimes complex and counterintuitive. They were not always given descriptive names.

After deployment, the background process that handled the synchronization operations could not handle the amount of actions required to synchronize the user and subscription data. The interactions were performed in a synchronous way, so they queued up. Processing a large batch of user data caused over a 100 requests per second. The amount of requests queued up for the background process eventually caused it to exceed its memory limit and be terminated. Measures were taken to prevent this from happening in the future. This illustrates the performance problems synchronization can cause.

3.3 Content Management System

The Content Management System was developed for the same publisher of digital documents as the portal from Section 3.2, and it lives in the same environment as the portal. As digital publications became more complex, the need for integration with multimedia resources such as video and music arose. A system was needed to store and organize these multimedia resources, and make them accessible to users of digital publications over the internet. The Content Management System (CMS) of this case study was developed for this. It was chosen as a case for this research because it interacts with three other systems in the publisher environment.

3.3.1 Purpose

The most important purpose of the CMS is to store and serve multimedia content to the users of digital publications. Digital publications are linked to content in the CMS. The CMS can also store additional information for an occurrence of a piece of multimedia content in a digital publication. A secondary purpose of the CMS is to provide options for organizing multimedia content, so that it can easily be found by the editors of digital publications.

3.3.2 Environment

The environment of the CMS system is the same as the environment of the portal, which is a set of interconnected applications at a publisher (see Section 3.2.2).

The CMS provides a service using XML-RPC that is used by another application called clippingtool. This application is used by the editors of digital publications to define where content from the CMS needs to be inserted. This XML-RPC API is exposed with the ActionWebservice module (see Section 2.4).

The readers of digital publications download a flash application to display their publications in their browser. the CMS provides a REST-ful service through which this flash application can retrieve pieces of multimedia content linked to the digital publication it is displaying. Metadata about this content is transfered in a custom XML format.

The CMS also allows users to link content to products from the Portal system in the publisher environment. A product, as defined in Portal, is a digital publication or set of digital publications (see Section 3.2.3). The flash reader can retrieve the content that is associated through a product with the publication it is displaying. This content is ‘global’ for that publication, which usually means it is displayed on every page.

To allow the CMS to retrieve information about products the portal provides a REST-ful service. The CMS can retrieve XML-representations of each product or set of products through this service.

3.3.3 Data Structure

An item of content is called a component in the CMS. A link between a component and a publication is called a clipping. A clipping has a publication

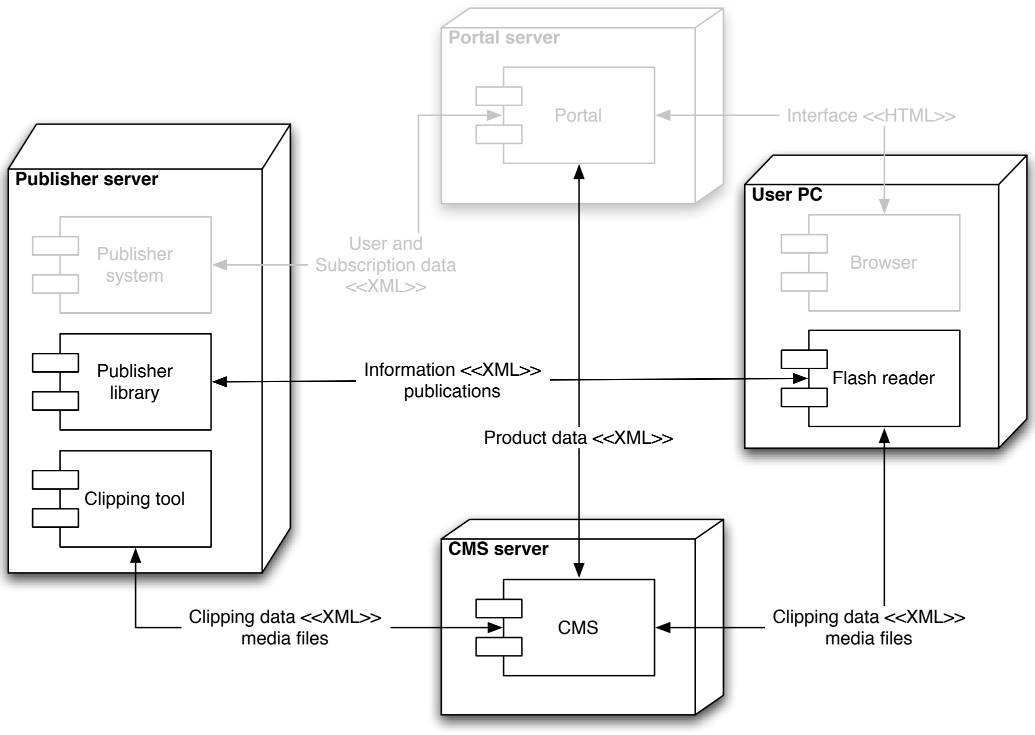

Figure 3.4: Publisher environment from CMS perspective.

id that references the publication in the publisher system.

Figure 3.5 shows the data structure of the system. A component is created from a component_template that has several template_parameters. A template_parameter is a name and type of an attribute, for example title:String or picture:File. When a new component is created, several parameter objects are created as well. The names and types of these parameters are copied from the template_parameters associated with the template from which the component was created. Values for these parameters can then be provided. Extra parameters, which are specific for this component, can also be added.

A clipping can overwrite parameters of its component. This is done by creating a setting entity that represents a clipping-specific value for a certain parameter.

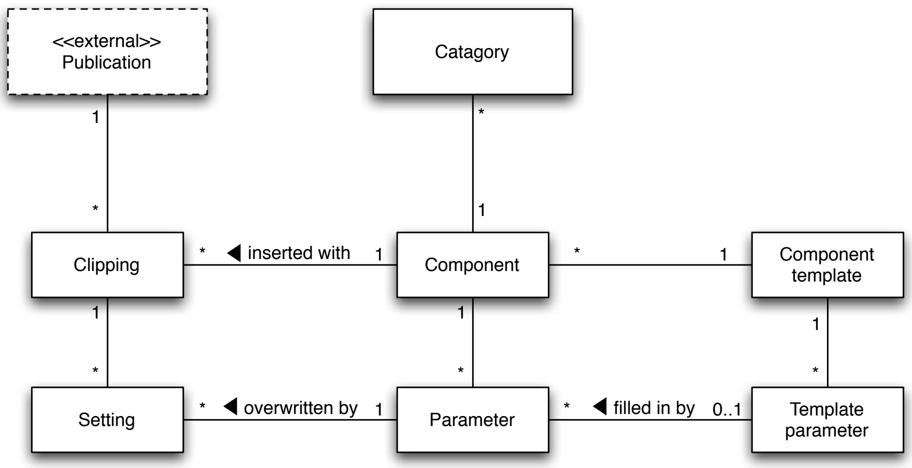

Figure 3.5: CMS data structure

3.3.4 Workflows and Interactions

The workflow for linking a piece of content to a digital publication is as follows:

1. The clippingtool retrieves a list of components from the CMS through an RPC-style service. The publication editor can then insert these items.

2. When the editor inserts a component into a publication the clippingtool tells the CMS about it through the RPC-style service. The CMS then creates a clipping for that component in that publication.

When a reader accesses a digital publication the following happens:

1. The reader downloads a flash application that can present the digital publication to him. His browser automatically takes care of this.
2. The flash application accesses the Publisher library to retrieve the requested digital publication.
3. The flash application then accesses the CMS to retrieve a list of clippings for the current page of the digital publication. This is done through the REST-ful service the CMS provides.
4. The flash application processes this list. The list contains metadata about the clipping and URIs of pieces of binary content. This content can then be retrieved and displayed on the page.

3.3.5 Development Process

The requirements on how content should be stored were not immediately clear. First, it was understood that the CMS would simply contain components with parameters. These components could have a wide range of types (for example 'picture' or 'movie'). Later, it became clear that not all these types could be defined in advance. Therefore, the template entity was introduced and used as described in Section 3.3.3. Even later, it became clear that links between content and digital publications also needed to be stored in the CMS system (the clipping entities). Extra properties of these links also needed to be maintained.

The connection with the clippingtool was easier to develop than the connection between the publisher system and Portal. the CMS was the service provider in this connection, so the programmers could decide which procedures to expose and how to name them. They also had some control over the clippingtool source code. These factors eliminated the problems encountered in the Portal.

Initially, there were no requirements on how the CMS should communicate with the reader. The reader would be developed after the CMS. The only known was that it would be able to communicate with REST-ful services, and that it could parse XML. There were no requirements on the XML format. Rails proved an effective platform for providing REST-ful services, so this was initially no problem. The built-in Rails mechanism to serialize ActiveRecords to XML was used to expose resources through a read-only service. When the reader was being developed, requirements on the XML format and content started to emerge. This led to the programmers choosing to overwrite the method that serializes a certain ActiveRecord to XML (called `to_xml`). This was no clean solution, since every XML representation of that ActiveRecord was now changed. This violates the MVC pattern, of which one of the goals is loose coupling between models and their representations. In retrospect, they could also have used Rails response generation mechanism to generate a custom representation of the ActiveRecords.

The communication with the portal application as described in Section 3.3.2 was added much later in the project's cycle. It was relatively easy to provide an REST-ful service from the Portal application. It was harder to invoke that service from the CMS side. Since CMS was developed in Rails 1.2, there was no ActiveResource (see Section 2.4) available for easily invoking REST-ful services provided by other Rails applications. Standard libraries for communication over HTTP and XML parsing were used to invoke the service.

3.4 Froodi

Froodi is a website that allows people to search metadata about podcast feeds and episodes. This is a publicly available service that this website offers, supported by advertisement. Anyone can search the Froodi podcast database. It was chosen because it forms a massive service-oriented system with all the podcasts it indexes.

3.4.1 Purpose

The system has two purposes. The first is to retrieve podcast feeds, parse them, and store the information they contain in a database. The second purpose is to provide a user-friendly publicly available interface for people to search this database. In addition to this people can register on the site, which allows them to make lists of their favorite feeds and share them with friends.

3.4.2 Environment

A podcast is basically an extended RSS feed [47][48]. RSS stands for 'Really Simple Syndication'. An RSS feed is an XML document with a certain format. This document can be retrieved through HTTP and processed by an application, e.g. an RSS reader. The RSS feed contains some metadata and a list of items that is updated regularly. Each item has a release date. By looking at the release date the application processing the feed knows which items are new. A podcast contains a link to an MP3 or video file in every item. In a podcast, an item is usually referred to as an episode.

Froodi scans a predefined list of feeds for updates in metadata and new items periodically. It could use release dates to determine which content is new and which was already retrieved, but this is not always reliable. Changes in old data could remain unnoticed. Therefore, Froodi scans all data from the feed and compares it with the database. Data that is not yet in the database is added.

There may be several URIs in the feed list that refer to the same podcast. Froodi compares metadata to find these duplicates, but this may not be accurate. Another solution that was considered but not implemented was to take a hash of the media file of the last episode of all podcasts and compare these to find duplicates. This would however require a lot of time and bandwidth and might still not be accurate.

A big problem with scanning the feeds was performance. To index all feeds, Froodi takes more than 8 days. An performance analysis concluded that the standard Ruby RSS processing library that was used to parse the feeds was slow. The indexing and storing operations that Froodi's search engine performed were also slow. No improvements have been made on this so far.

Figure 3.6 shows the environment of Froodi. As you can see, in addition to scanning feeds, Froodi also interacts with the client in several ways. In addition to the html interface, it also provides its own RSS podcast feeds. These feeds are basically the same as the original feeds it scanned. The differences are that

1. They are generated using the data cached in Froodi, not the data from the real-time feed.

2. They have an extra field, a Froodi id, that references their id in the Froodi database.

3. They are retrieved from a uniform and consistent URI space relative to the Froodi site.

4. The data they contain is more standardized.

The advantage of this approach for the owner of Froodi is that it receives information about who accesses which feeds. This allows for targeted marketing later on. The user can play these feeds (or actually the content enclosed in these feeds) using either Froodis flash player or their own desktop player.

Another way Froodi can interact with a client is through providing channels. A channel is basically a bundle of podcast feeds that users can create. A channel can be represented in OPML [49]. Each channel has its own URI. Several popular desktop players currently in existence can import OPML podcast bundles [50].

3.4.3 Data Structure

The search functionality is quite extensive. Podcast metadata retrieved from feeds may include a title, a description, author name, etc. metadata from the individual episodes contained in the feeds may also include a summary

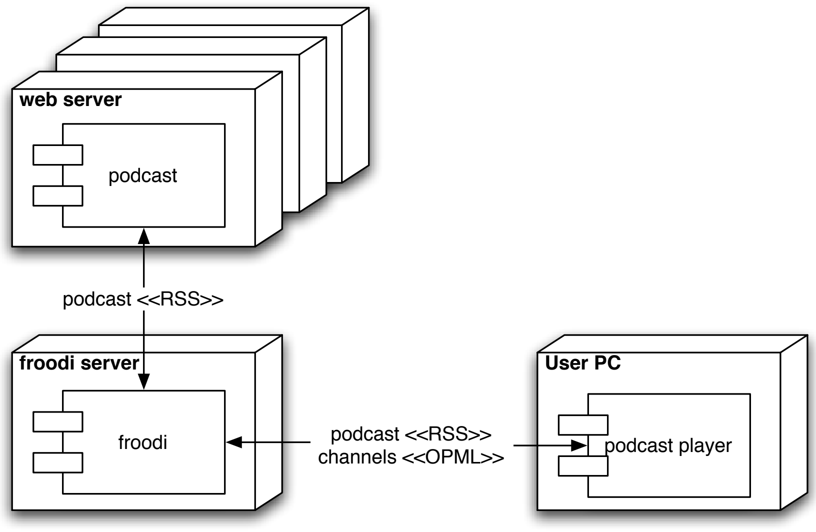

Figure 3.6: Froodi Environment

and a title, as well as duration, release date, etc All this data is indexed by Froodi and can be searched. For this, Froodi uses a Rails plugin called Ferret [51]. Ferret allows for full-text search and relevance sorting across multiple fields. It indexes the relevant database entities on the file system and presents an API to search this index and sort results by relevance. It also makes it possible to give fields a weight, which influences relevance sorting.

When a user registers or logs into the site, the following features become available to him:

1. Subscribing to podcasts or channels.
2. Tagging podcasts or channels.
3. Commenting on podcasts or channels.

Figure 3.7 shows how the database is structured. A taggeable, rateable, commentable or subscribeable can be an entity of any type. Currently, only podcasts and channels are taggeable, rateable, etc., but this can easily be extended [52].

Podcasts can belong to channels. As mentioned in Section 3.4.2, a channel is a bundle of podcasts created by users. Podcasts can also have tags associated with them. A tag is like a keyword. These tags are applied by users. Podcasts can be browsed by tag or channel.

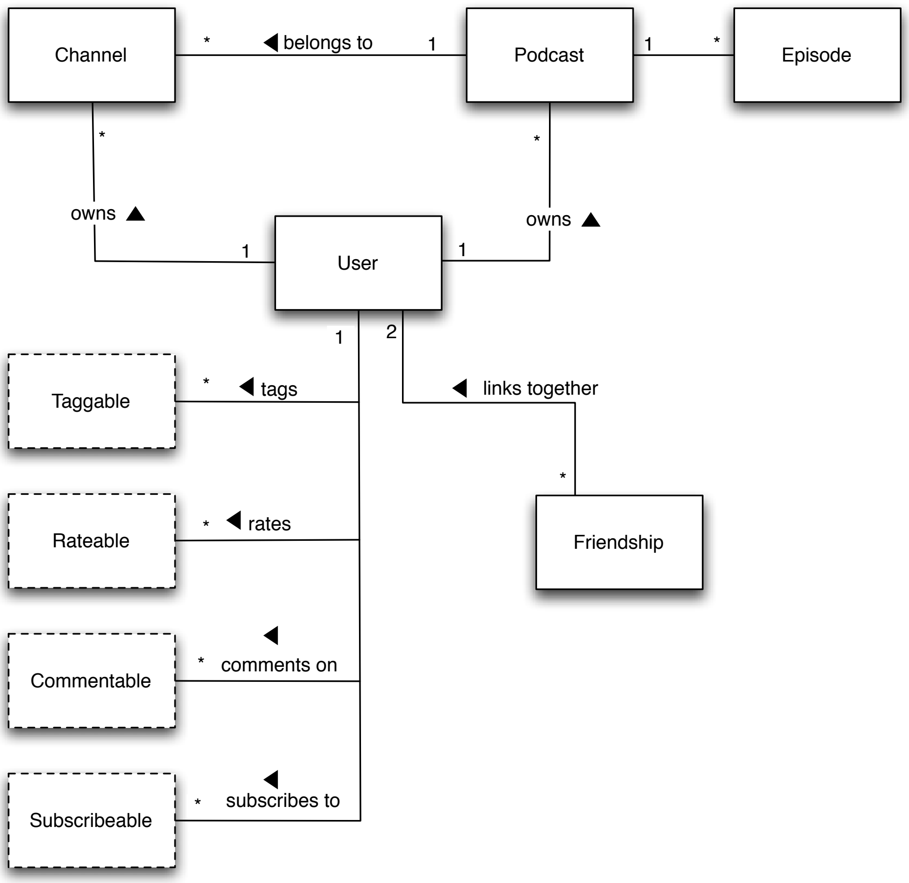

Figure 3.7: Froodi Data Structure

3.4.4 Development Process

Implementing the feed reader was not trivial. The RSS format allows for many dialects, which makes it difficult to extract uniform information. Functionality to deal with this had to be implemented. Also, REST-ful services that provide RSS feeds could not be accessed easily. There is no functionality in the Rails platform to invoke most REST-ful services. Standard libraries for HTTP communication and XML parsing were used to invoke the service.

Providing a REST-ful service that provides RSS and OPML documents was easy. Rails proved to be an effective platform for providing REST-ful services. The response generation mechanism proved to be flexible enough to generate RSS and OPML documents.

There were plans for developing a desktop podcast player to rival iTunes and other popular players. This player could use the profile information from the Froodi site to provide a highly customized user experience. Integrating this player with Froodi would not be difficult. The OPML exporting of channels and the forwarding of feeds through Froodi provide the tools to develop this integration. This desktop player has not been realized.

3.5 World Usability Day

The World Usability Day website is a product that allows users to manage information about events organized for the World Usability Day [53]. It was chosen because of its integration with the Eventbrite site.

3.5.1 Purpose

The World Usability Day is a worldwide event that happens once a year. It consists of many local events like symposia or workshops, focused around the theme of usability. Typical participants are universities and companies that are involved with usability in some way. The World Usability Day website allows interested people to find information about events, and event organizers to register events.

3.5.2 Environment

Figure 3.8 shows the environment of the product. When an event organizer registers an event, he is redirected to the Eventbrite site where the actual registration takes place. Eventbrite is a site where people can register and manage events [54]. The WUD website is not the authoritative source on WUD events. WUD events are registered and managed at Eventbrite first. The WUD application then retrieves information about events in an XML format through an REST-ful service provided by the Eventbrite site. These events are then displayed on the WUD site.

3.5.3 Data Structure

As you can see in Figure 3.8, the site contains information on users and events. Events are mainly managed in Eventbrite and downloaded from there, as mentioned in Section 3.5.2. The WUD site also contains functionality to have event organizers register and manage an event. This functionality is deprecated in favor of putting all event information on Eventbrite, but was retained as a fallback option in case the integration with Eventbrite failed.

The administrator of the site can change events and users. When he changes an event, the event gets a flag that means its properties should no longer be downloaded from Eventbrite. Eventbrite is then no longer the authoritative source of information about the event, the WUD site is. Events are grouped by years (or editions of the World Usability Day). Besides events and users, the administrator can also change news items, documents, etc.

3.5.4 Workflows and Interactions

The workflow of creating an event and having it displayed on the website is as follows:

1. An event organizer visits the World Usability Day website. Here he discovers a link to the World Usability Day context of the Eventbrite website.
2. On the Eventbrite website, he creates a (payed) account and uses Eventbrite's functionality to publish, promote and sell his event.

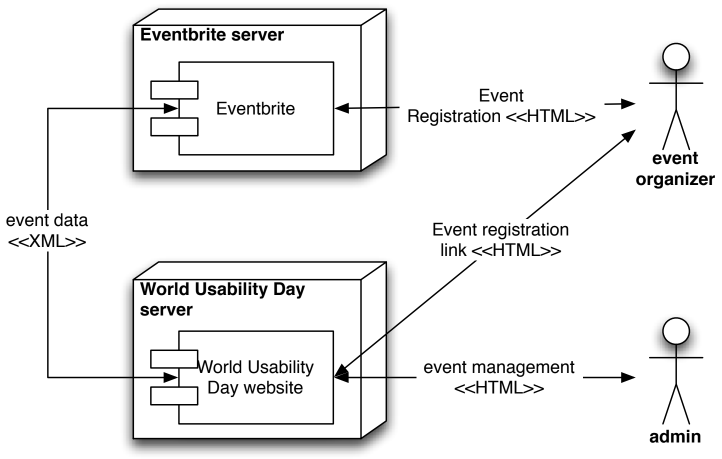
Eventbrite server
Eventbrite
Event Registration <<HTML>>
event organizer
event data <<XML>>
Event registration link <<HTML>>
World Usability Day server
World Usability Day website
event management <<HTML>>
admin

Figure 3.8: World Usability Day Environment

3. Periodically, representations of the event are downloaded from the Eventbrite website and stored in the database backing the WUD website. This information is then visible on the website.

4. When an event that is contained in the WUD website database can not be found on the Eventbrite site, its status is set to deleted.

5. When an event is modified by the administrator in the WUD website, its status is set to manual. This event is no longer updated with information from the Eventbrite website.

3.5.5 Development Process

Most of the functionality of the WUD as it exists now was developed by a company other than the company where we did this case study. This other company developed the previous version of the website, which was then adapted by the company where we did the case study to accommodate the 2007 edition of the World Usability Day. The previous version did not integrate with Eventbrite. The other company developed all the stand-alone functionality for managing events and managing content. Our company developed the integration with Eventbrite.

Redirecting the user to the World Usability Day context in Eventbrite was a trivial matter. Eventbrite uses a specific URL for each event context, so a redirect to that url was all that was necessary.

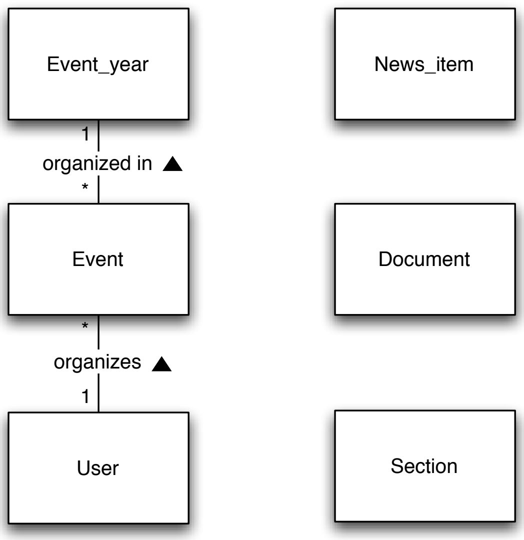

Figure 3.9: World Usability Day Entities

The retrieval of events and updating local data with them was more difficult. The Eventbrite application provides an REST-ful service through which XML formatted data of events can be retrieved. There is no functionality in the Rails platform to invoke most REST-ful services, including this one. Standard libraries for HTTP communication and XML parsing were used to invoke the service.

3.6 Evaluation of Discovered Problems

In the case studies we encountered the following problems:

Portal

1. Data duplication pattern causing difficult synchronization process.
2. Insufficient functionality in publisher system XML-RPC interface causing possible illegal states.
3. Hashed password in URL causing security risk.
4. Other platform doesnt understand ActionWebService's array representation for XML-RPC.
5. ActionWebservice has no support for basic HTTP authentication.
6. Incorrectly described XML-RPC interface of publisher system causing confusion.
7. Unclear and complex semantics of publisher system XML-RPC interface causing confusion.
8. Background process handling XML-RPC requests could not handle load.

CMS

9. Relations between external entities stored locally requires complex and hard-to-maintain interactions.
10. The requirements on the data model and relation to external entities changed late in the development process. This required many changes in the service interface.
11. A model was changed to change its representation. This violates the MVC pattern.
12. Invoking REST-ful services from other Rails applications was not supported by Rails 1.2.

Froodi

13. RSS feeds: Inconsistency of data unit and type among different feeds.
14. RSS feeds: Possibility of duplicates, which is hard to detect.
15. RSS feeds: Hard to decide which data is new, changes could be made in old data and could remain undetected.

16. Invoking REST-ful services that expose RSS feeds not supported in Rails.

WUD 17. Invoking REST-ful services that expose information in a custom XML format not supported in Rails.

Problems 2, 8, 9 and 11 can only be attributed to design and implementation errors. Solving these is beyond the scope of this research. Problem 1 originates from the use of the SOA anti-pattern of data duplication. Problem 3 is a common problem. A common solution to this is a third application that handles authentication for both other applications. The simplest form of this is a single sign-on system [ref]. Problem 6 is a communication problem. A service should be properly documented for other applications to use. Using a domain-specific language for this, such as WSDL, could have solved this problem to some degree. A related problem, problem 7, could not have been solved with WSDL, since WSDL only documents the syntax of a service, not its semantics. Solving problem 7 is a matter of good communication outside the actual programming activities. It also shows that a service should provide a simple and understandable interface, especially when the applications that use it are developed by different programmers.

With regard to problem 12: As of Rails version 2.0 programmers can use ActiveResource (Section 2.4) for easy communication between Rails applications, so this problem is no longer relevant. Problem 13-15 are mainly caused by lack of standardization of the RSS format. This includes feed providers not adhering to the format, and the format not being strict enough to allow for easy interoperability.

That leaves problems 4, 5, 10, 12, 16 and 17. These are all problems originating from missing or poorly integrated functionality in Rails. Problems 4 and 5 illustrate limitations of ActionWebService when invoking RPC-style services. Problem 10 illustrates Rails' lack of agility with regard to RPC-style services when using AWS. A change in a service interface should be easy to accomplish. Since AWS is no longer part of the Rails framework, we argue that an alternative to AWS should be sought. The most serious problems however are 16 and 17. These show that Rails has no functionality for invoking services over HTTP.

Figure 3.1 shows the various SOA tasks Rails could perform, as identified in Section 2.4. The numbers refer to the problems encountered when performing these tasks. SOAP was not used in any of the cases. From the problems encountered in the cases we argue that the main limitations of Rails with regard to SOA are:

- Limited functionality to invoke services over HTTP
- Missing or poorly integrated functionality to provide or invoke RPC-style services

		Protocols		
		HTTP	XML-RPC	SOAP
Roles	Invoke	16: RSS, 17: XML	4: Array representation, 5: Basic authorization	Not Used
	Provide	🟥	10: Late changes	Not Used

Table 3.1: SOA tasks with problems

3.7 Summary

In this chapter we evaluated four Rails projects. All these Rails projects are part of a SOA. They provide and invoke REST-ful and RPC-style services. Programmers encountered several problems when working on these projects. Most of these problems could not be attributed to limitations of Rails. Some of them could, however. Based on the encountered problems we have concluded that the main limitations of Rails with regard to SOA are:

- Limited functionality to invoke services over HTTP
- Missing or poorly integrated functionality to provide or invoke RPC-style services

Chapter 4

Framework

In this chapter we discuss a framework for service-oriented extensions to Ruby on Rails. In Section 4.1 we discuss the requirements for this framework and how they relate to the limitations identified in Chapter 3. Here we also discuss the overall architecture of the framework. The framework is split into two independent modules, one for invoking and one for providing services with Rails (service_invoker and service_provider). Design considerations for the two modules are discussed in Section 4.2 and Section 4.4. How the modules are implemented and how they can be used is discussed in Section 4.3 and 4.5.

4.1 Requirements and Architecture

The case studies have shown that Rails ability to invoke RPC-style and REST-ful services needs improvement (see Section 3.6). Invoking REST-ful and RPC-style services consists of generating requests and processing responses. The case studies have also shown that Rails' abilities to provide RPC-style services needs improvement. Providing RPC-style services consists of processing requests and generating responses.

The goal of the framework is to allow programmers to extend Rails' ability to invoke and provide RPC-style and REST-ful services. To do this the framework does two things:

- Provide an extensible mechanism for invoking services over HTTP.
- Provide an extensible mechanism for providing services over HTTP.

To perform these two tasks the framework has been split into two parts: the service_provider and the service_invoker. The service_provider leverages Rails' existing abilities to provide services over HTTP by providing an extension mechanism that can be used to implement any protocol that provides services over HTTP. The service_invoker allows Rails to perform HTTP requests and process HTTP responses. It provides an extension mechanism that can be used to implement any protocol that performs requests to services over HTTP.

An important consideration with regard to using and providing services is whether to limit the framework to the HTTP protocol, or to allow other internet application-layer protocols. However, because Rails already supports providing REST-ful services over HTTP, and because two important service protocols (XML-RPC and SOAP) are commonly used over HTTP, the choice was made to limit the framework to the HTTP protocol.

A general non-functional requirement for the framework is that it should be easy to use. This means that it should integrate well with the Rails architecture. If our solution would depart to much from this architecture it would be hard to learn and use for Rails programmers. It also means that it should provide programmers with powerful but simple and intuitive tools for providing and invoking services.

4.1.1 Service Invoker

The service_invoker extension overlaps in functionality with ActiveResource. ActiveResource can only invoke services that follow the ActiveResource message format and interaction semantics. The case studies have shown that that is not enough. We propose a framework for invoking services similar to ActiveResource, but extensible and less prescriptive.

The service_invoker is designed to be used by specifying models, similar to ActiveResource and ActiveRecord. Models are globally available to all parts of the application, and can be explicitly invoked to query or manipulate resources. In service_invoker, as in ActiveResource, a model represents a service. Conceptually the model is the right place to put this framework because a model element is supposed to represent application state, and the services that an application invokes can be seen as part of the application state of that application. Figure 4.1 shows the modules of Rails and the service_invoker framework.

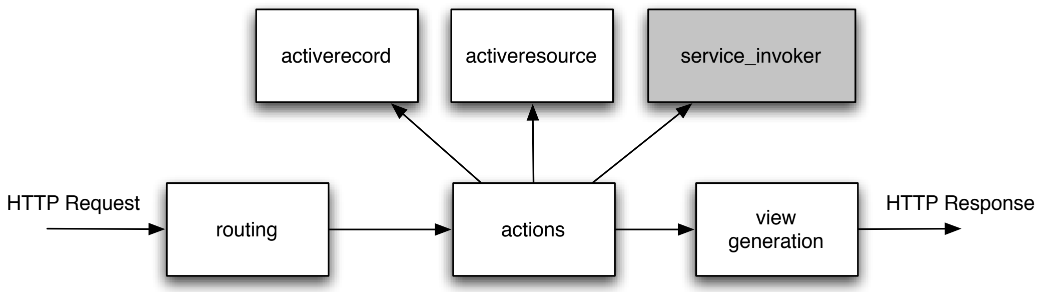

Figure 4.1: Rails and the service invoker framework

The service_invoker is designed as a framework on which extensions can be built that handle certain protocols (like XML-RPC or SOAP). Figure 4.2 shows the layered architecture of the service_invoker framework. The first layer is the actual service invoker that was implemented. It offers the basic functionality of doing HTTP requests and processing responses. It also provides an extension mechanism. The second layer builds on that extension mechanism by providing handlers for several protocols, and through them functionality to invoke services. The third layer is an actual Rails application that uses the service_invoker framework. It uses one or more of the provided extensions to invoke services and process their responses. This framework has no dependency on Rails. It can easily be used without Rails.

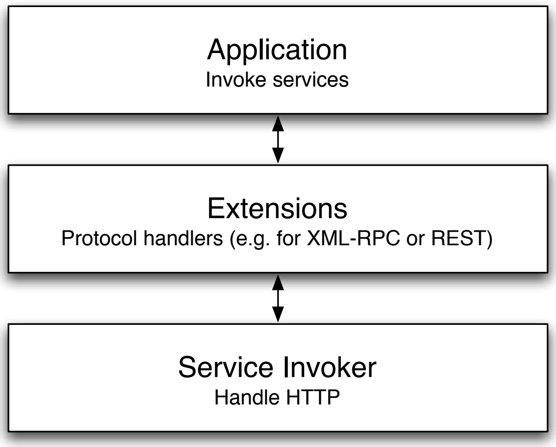

Figure 4.2: The layered architecture of the invoker extension

4.1.2 Service Provider

As described in Section 2.4, Rails can provide RPC-style services using XML-RPC and SOAP with ActionWebService. The service_provider framework will provide a more extensible and customizable alternative to ActionWebService. It does so by adding an extensible request decoding and routing mechanisms to Rails.

The routing mechanism of Rails is focused on the URI and request method in the HTTP request. The service_provider allows messages to be routed based on what they contain in their HTTP body. It assumes that requests can be routed to a method using a fixed set of rules.

Figure 4.3 shows the modules of Rails and the service_provider framework.

Service_provider is designed as a framework on which extensions can be built that handle certain protocols. A protocol implies a message format and a routing mechanism. XML-RPC for example prescribes a certain format for encoding of messages. It also implies that messages be routed based on the procedure name they contain.

Figure 4.4 shows the layered architecture of the service_provider framework. It is built on Rails, which is the first layer. Rails allows for routing based on URI and request method. The second layer, the actual service_provider, provides an extension mechanism that allows extensions to implement decoding and routing based on the HTTP body. The third layer builds on that extension mechanism to provide handlers for certain protocols (like XML-RPC and SOAP). It performs decoding of the protocol message in the HTTP body, and it allows application programmers to define rules for routing the request based on the results of that decoding. The fourth layer is an actual Rails application. The application programmer defines routing rules for his application through the extension. The application can also use the result of the decoding implemented by the extension. In case of XML-RPC, that would be the parameter list.

4.2 Design of service_invoker

The simplest design for a service invoker would be to use a singleton object with a single public method that developers can call to invoke services. Extensions could use inheritance to extend the singleton and overwrite the method as shown in Figure 4.5. The invoking method would have a lot of parameters, from which it would be able to construct an HTTP request and process the HTTP response in many ways. The main limitation of this approach is that messages cannot be reused. The invoke_service method would need all parameters to construct a request or process a response every time it was called.

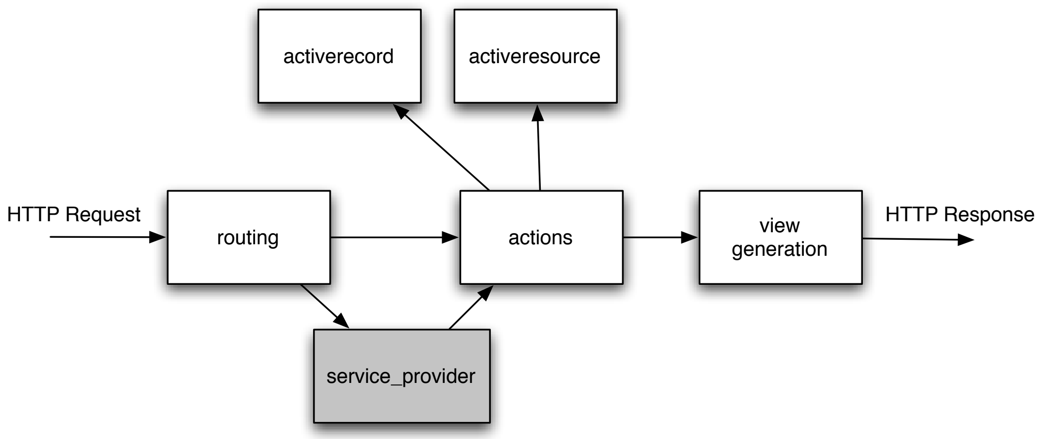

Figure 4.3: Rails and the service provider extension

To allow for message reuse, we define several extra types to represent messages in our system, as shown in Figure 4.6.

Extensions to the service_invoker come in the form of configurable objects with a method for generating request objects (RequestBuilders), and modules that add functions to process response objects (ResponseProcessors). RequestBuilders follow a factory pattern. ResponseProcessors are included in the response objects returned by the ServiceInvoker. The ServiceInvoker can be configured to use specific RequestBuilder and ResponseProcessor modules.

Programmers can specify their own invoker singletons by extending the base invoker. For each invoker they can define which factory and which processors it should use. In these singletons they can also define functions that perform a specific request or do a specific response processing operation by wrapping the base method, like post_orders or get_sales_numbers in Figure 4.8.

4.3 Implementation of service_invoker

the service_invoker is implemented as a plugin. It is automatically loaded when Rails starts. It can be used anywhere in the application by calling `ServiceInvoker::Base.invoke()`. This method takes a url, an HTTP method, a body and a list of headers. Figure 4.9 shows the main objects in

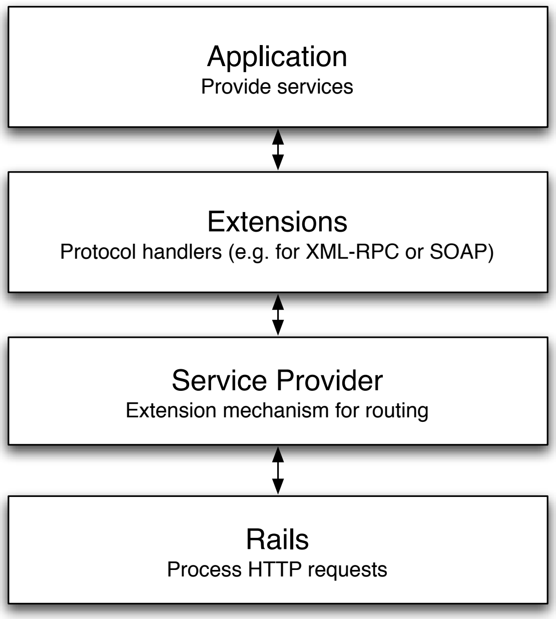

Figure 4.4: The layered architecture of the provider extension

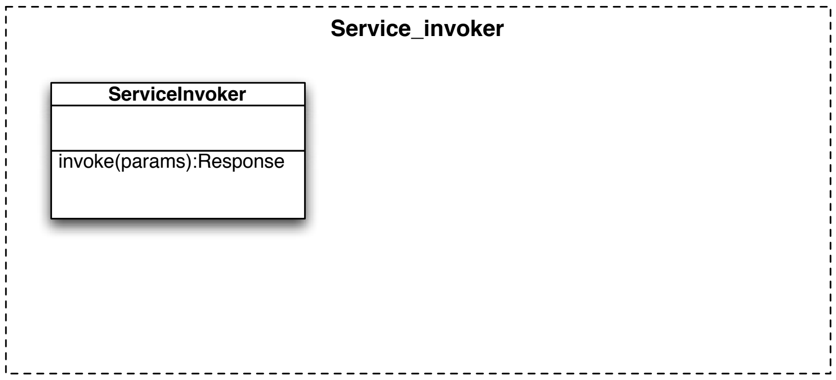

Figure 4.5: Service Invoker as a singleton

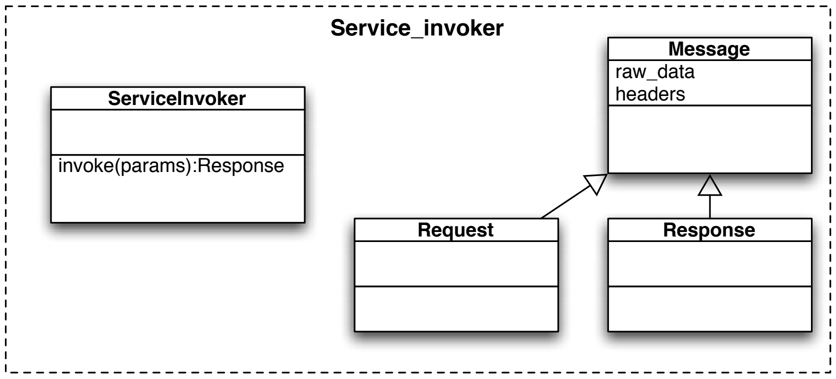

Figure 4.6: Service Invoker with separate message object

this extension processing a call to `ServiceInvoker::Base.invoke()`. The final result is a `ServiceInvoker::Response` object. This object encapsulates the HTTP response that follows from invoking the specified service.

The plugin provides an easy extension mechanism in the form of custom request builders and response processors. To use these custom builders and processors you need to make an invoker class that extends the ServiceInvoker::Base class, and specify what builder and processors it should use. This invoker class contains the location of a service and the knowledge of how you can use this service. service_invoker includes custom builders and processors for several common use cases. A simple invoker class could look like this:

```
class Example < ServiceInvoker::Base
  request_builder BasicBuilder,
    :endpoint_url => "www.example.com/rest"
```

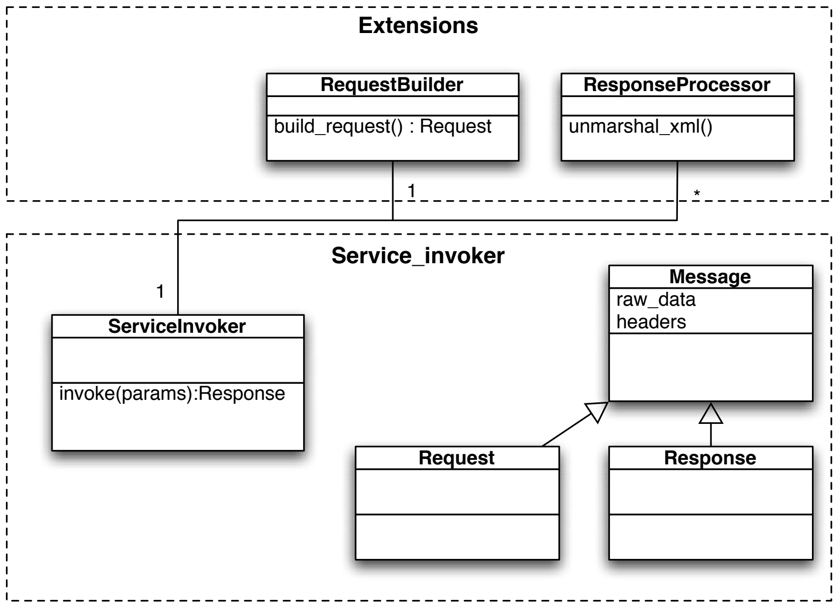

Figure 4.7: Service invoker with request builder and response processors

```
  response_processor XMLProcessor
end
```

This invoker uses the BasicBuilder. The default builder is ServiceInvoker::AbstractBuilder. BasicBuilder extends the AbstractBuilder to add, among other options, an option for a default url. this means that when ExampleInvoker.invoke() is called it is not necessary to give an url parameter. This invoker also uses the XMLProcessor response processor. This means that the response object that is returned contains functionality to process XML. A call to Example.invoke() would look like Figure 4.10. All classes in this diagram directly inherit from their standard version in Figure 4.9.

The extension also includes a custom builder and processor for the XML-RPC protocol. This builder simply generates an HTTP POST request with an XML-RPC body. The body is defined by the arguments passed to the builder (i.e. the procedure name and parameters of the targeted XML-RPC procedure). You can define an XML-RPC service invoker like this:

```
class Example < ServiceInvoker::Base
  request_builder XMLRPCBuilder,
    :endpoint_url => "www.example.com/xmlrpc"
  response_processor XMLRPCProcessor
end
```

The `XMLRPCBuilder` allows options for specifying XML-RPC messages

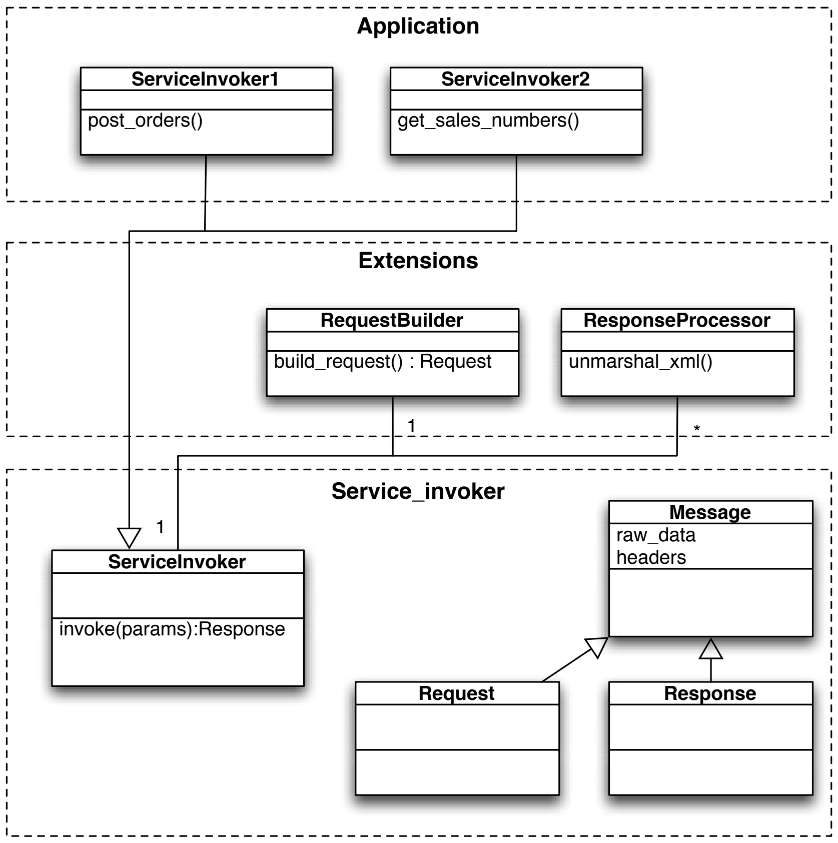

Figure 4.8: Service invoker with subclasses for use in specific applications

to be passed to the invoke method. For example, this line:
`Example.invoke(:method => 'say_hello', :args => ['foo'])`
sends the following XML-RPC message to www.example.com/xmlrpc:

```
<?xml version="1.0" ?>
  <methodCall>
  <methodName>say_hello</methodName>
  <params>
    <param><value><string>foo</string></value></param>
  </params>
</methodCall>
```

The response object that is returned from this method call contains methods to process XML and XML-RPC through the `XMLRPCProcessor`. Both the builder and the processor use the xmlrpc4r library mentioned in Section 2.4.

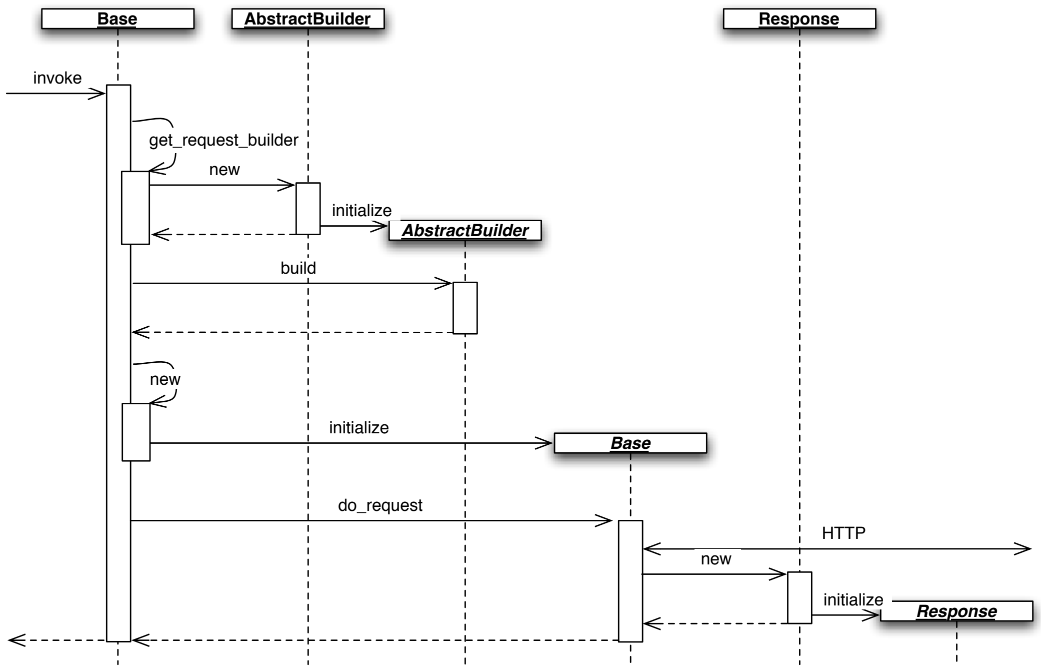

Figure 4.9: Service invoker performing an HTTP request

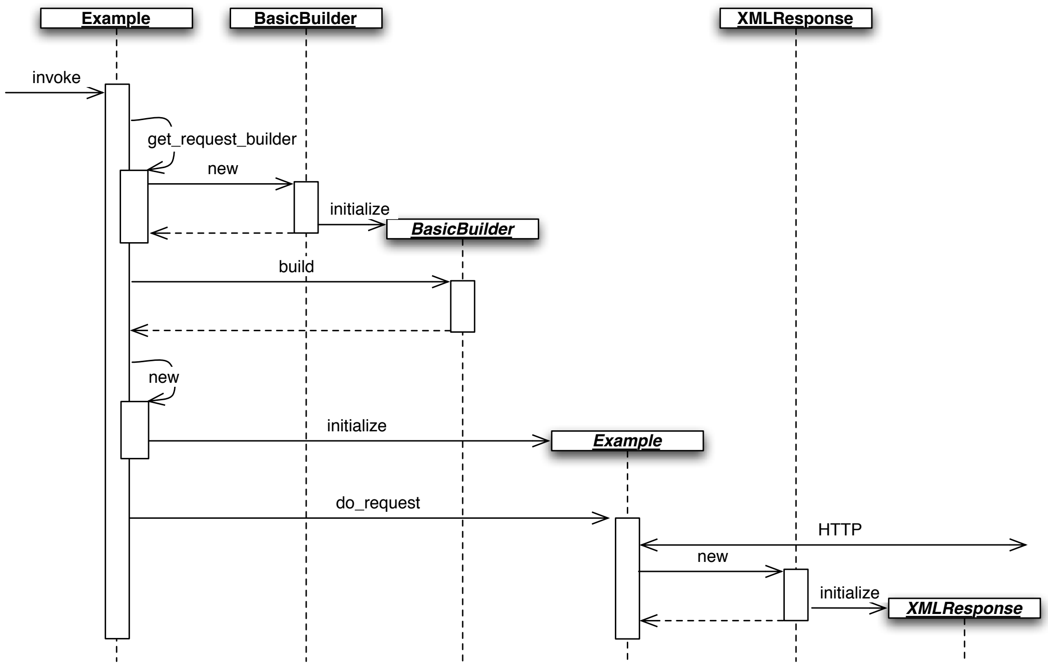

Figure 4.10: Extended service invoker performing an HTTP request

4.4 Design of service_provider

The provider framework is simply called service_provider, and adds an extensible decoding and routing mechanism to Rails. As described in Section 2.2, Rails' own routing mechanism allows programmers to specify routing rules, which are used by the ActionController module to map a request to a controller class and a method name. This controller is then automatically instantiated and the method is called. Rails' own routing mechanism only uses the HTTP URI and request method. Rails' decoding mechanism decodes the HTTP body, but only recognizes url-encoded key-value pairs and plain XML documents. When another format is received it is passed to the handling ActionController method as raw data.

The extensible decoding mechanism allows extension developers to decode the HTTP body in any way. The extensible routing mechanism allows extension developers to define a syntax for routing requests based on the decoded HTTP body.

There are several options for extending Rails' routing mechanism to allow for the routing of requests based on their HTTP body:

1. Recognize a request as needing extended decoding and routing and send it to a separate module instead of the normal routing mechanism.

2. Use normal decoding and routing to send a request needing extended routing to a separate module for further decoding and routing before sending it to the ActionController. This requires that a request needing special routing can be distinguished from a normal request by normal routing.

3. Do not extend the routing mechanism directly, use only normal routing and have programmers build a large ActionController method to perform the extended routing manually.

The first option does not take advantage of Rails ability to do routing based on HTTP method and URL. A mechanism that efficiently detects wether a request is a layered request would look at the HTTP method (both XML-RPC and SOAP use only the POST method). This would duplicate functionality. It would also decrease performance of normal operations, since every non-service request would need to be scanned by the detection mechanism.

The third option is not really an extension. In this case service_provider could provide some functions for easily processing a service request, but this does not take advantage of Rails' abilities like per-method filters. Rails' mechanism to automatically search for a template to generate a response from wouldn't be useful either.

The solution used for this extension is the second. It uses Rails' normal decoding and routing mechanism to route a request that needs extended decoding and routing (e.g. an XML-RPC or SOAP request) to an additional routing mechanism that decodes the HTTP body and routes the request further based on the data in the HTTP body. This mechanism requires that requests that need extended decoding and routing can be distinguished from normal requests based on their HTTP URL and method.

You use service_provider by including a module in an ActionController instance. As described in Section 2.1, classes can be dynamically modified in any way. A module is one of the mechanisms that allows this. A module is not quite a class, but similar. Every class has a method include(module) which can be used to include a module. When a module is included in a class the class inherits al functions from the module.

service_provider extends an ActionController class by giving it a method to which requests can be routed for further decoding and routing. It also gives it a singleton object called Dispatcher which can be used to configure routing and processing of requests. Figure 4.11 shows the basic elements of the service_provider.

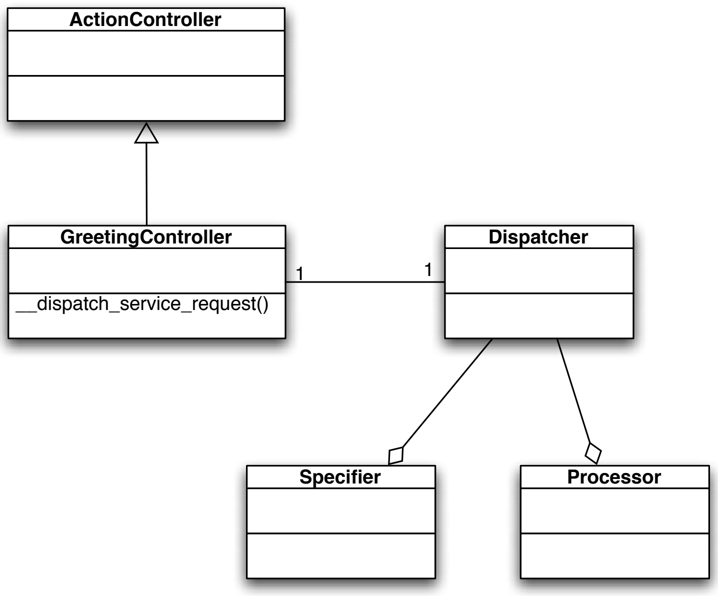

Figure 4.11: Basic elements of service_provider

Without extensions service_provider does nothing. Extensions add specifiers and processors to the dispatcher singleton. A specifier adds functionality to the dispatcher which allows programmers to specify routing rules. A processor is called when a request comes in. It decodes the requests' HTTP body and tries to match the data within to the routing rules specified by the programmer.

4.5 Implementation of service_provider

service_provider is implemented as a plugin. An XML-RPC extension was developed to demonstrate its functionality. The following controller code uses this extension:

```
class XmlrpcgreetingController < ApplicationController
  include XMLRPCProviding
  service_dispatcher.draw_routes do |map|
    map.xmlrpc 'greetings', :action => 'do_greeting'
    map.xmlrpc ':action'
  end

  def do_greeting
    result = "Hello " + self.service_request.params.first
    render_xmlrpc(result)
  end
end
```

What this code does is it provides an XML-RPC service with one function that returns a string containing 'hello' and the first parameter of the request. by including the XMLRPCProviding module (second line), the service_provider is also automatically included. It gives the class a singleton object which can be used to configure routing and processing of requests, and which is accessed through the service_dispatcher method (third line). The XMLRPCProviding module registers a specifier and a processor to the dispatcher. Routes can be specified through the draw_routes function, which takes an anonymous function as a parameter (this is called a block in Ruby) that allows programmers access to the functions from the specifier ('xmlrpc' in this case). The first routing rule shown here specifies that if a request with an XML-RPC procedure-name 'greetings' comes in it should be routed to the do_greeting action. The second rule specifies that any request should be mapped to an action with the same name as the XML-RPC procedure-name, so if a request comes in with procedure-name 'do_greeting' it is also routed to the do_greeting action. This syntax was borrowed from Rails' standard routing syntax.

When a request comes in the XMLRPCProviding processor uses the rules specified by draw_routes to process and route it. The object returned by the service_request method represents the request after processing by the XMLRPCProviding processor. The render_xmlrpc method is a convenience method also included by XMLRPCProviding to easily render XML-RPC encoded responses from Ruby objects.

4.6 Summary

In this chapter we proposed two extendible frameworks to help integrate SOA functionality in Rails. These frameworks were implemented as Rails plugins.

The service_invoker plugin adds functionality for invoking services over HTTP. It's extension mechanism allows for the invoking of any service over HTTP.

The service_provider plugin allows for the providing of services over HTTP with Rails. This plugin provides an extension mechanism that allows programmers to customize Rails' own decoding and routing mechanisms. It favors the RPC-style by assuming a request can be translated to a method call.

Chapter 5

Test Cases

In this chapter we discuss the implementation of several test cases using the framework. This chapter outlines what the cases entail and how they were implemented with the framework. Section 5.1 gives an overview of the cases described in this chapter. The individual cases are described in Section 5.3 through Section 5.9. Section 5.2 outlines the general design of the implementations.

5.1 Overview

This chapter contains the following cases:

Case 1: Providing a 'hello' service using XML-RPC (Section 5.3).

Case 2: Invoking a 'hello' service using XML-RPC (Section 5.4).

Case 3: Invoking Google Maps using REST and HTTP (Section 5.5).

Case 4: Invoking Flickr using XML-RPC (Section 5.6).

Case 5: Invoking Eventbrite using REST and HTTP (Section 5.7).

Case 6: Invoking Thumbalizr using REST and HTTP (Section 5.8).

Case 7: Invoking a document converter using REST and HTTP (Section 5.9).

These cases cover the invocation of REST-ful services over HTTP, as well as the provisioning and invocation of services using XML-RPC. Table 5.1 shows the cases classified according to the SOA tasks described in Section 2.4. The tasks that are not covered by the cases are the provisioning and invocation of services using SOAP and the provisioning of REST-ful services. Providing and invoking services using SOAP is not covered because SOAP interaction is beyond the scope of this research. Providing REST-ful services is the core function of the Rails framework itself. The provisioning of REST-ful services is not part of the framework.

		Protocols		
		HTTP	XML-RPC	SOAP
Roles	Invoke	Case 3: google maps Case 5: eventbrite Case 6: thumbalizr Case 7: document converter	Case 2: hello Case 4: flickr	
	Provide		Case 1: hello	

Table 5.1: Test cases classified according to SOA tasks

5.2 General Design of Implementations

In case 1 we provide a simple service. For this case we build a Rails application that includes the service_provider plugin described in Section 4.4 and Section 4.5. This Rails application also includes the functionality for providing the desired service using the service_provider, as described in Section 5.3. To test if the service is provided as expected, we use case 2.

In case 2 through 7 we invoke a service. For these cases we build a Rails application that includes the service_invoker plugin described in Section 4.2 and Section 4.3. The functionality for invoking all the services in cases 2 through 7 is included in this application. This Rails application has a web interface that can be used to invoke these services and display the results.

5.3 Case 1: Providing a 'hello' service using XML-RPC

In the first case we provide a simple service with XML-RPC using service_provider and its XML-RPC extension. The service exposes a single procedure that returns a string containing 'hello' and the first parameter sent with the procedure call.

Section 4.5 shows the implementation of this case. It shows how the XMLRPCProviding extension was used to expose a method through a service using XML-RPC. This implementation successfully provided the required service, which is invoked in Case 2.

5.4 Case 2: Invoking a ‘hello’ service using XML-RPC

In this case we invoke the service that was provided in Case 1. For this we use service_invoker. We send the service a single string. The returned result is the string ‘hello’ with the string we sent appended to it.

To perform this task we implement a subclass of ServiceInvoker::Base called ProviderInvoker. Figure 5.1 shows the ProviderInvoker and the extensions it uses. This figure is similar to Figure 4.8, except that it only shows the application and the extensions level. The ProviderInvoker uses the XMLRPCBuilder and XMLRPCProcessor, as well as the XMLProcessor. It defines an application-specific function called do_greeting_request that takes a name as a parameter, and uses the functionality provided by ServiceInvoker::Base and the XMLRPCbuilder to invoke the service. The code of the ProviderInvoker can be found in Section A.1. This implementation was successful in invoking the service provided in Case 1.

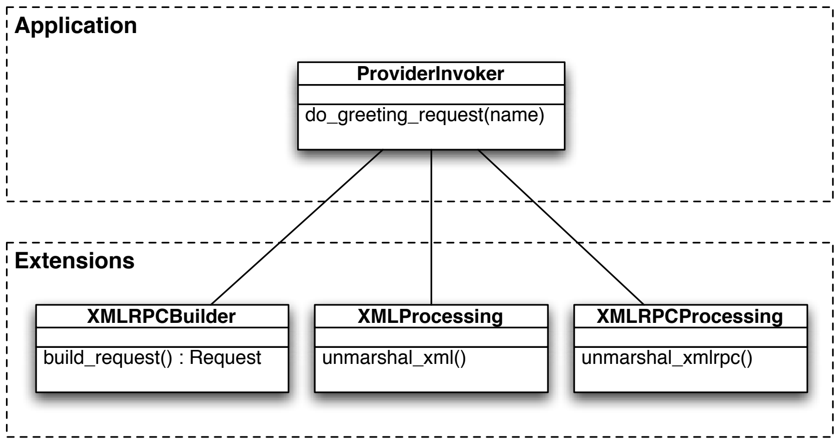

Figure 5.1: Implementation of Case 2. ProviderInvoker and extensions.

5.5 Case 3: Invoking Google Maps using REST and HTTP

In this case we use the service_invoker with the Google Maps API in order to retrieve directions and driving distance information. The Google Maps API is a REST-ful service. It exposes several resources that can be queried in order to receive address information, coordinates, driving directions and driving distance between two coordinates or addresses. It uses JSON encoding. We are mainly interested in the driving distance between two coordinates.

To retrieve driving distance between two coordinates we implement a subclass of ServiceInvoker::Base called GoogleMapsInvoker. Figure 5.2 shows the GoogleMapsInvoker and the extensions it uses. GoogleMapsInvoker uses the BasicBuilder to generate requests. The instance of Basicbuilder it uses is configured to send requests to a static URL using several default URL parameters. How this configuration is performed can be seen in the code in Section A.4. The GoogleMapsInvoker includes JSONProcessing. It also uses a response processor called MapsProcessing that was implemented specifically for this application. MapsProcessing provides functionality to extract the driving distance in meters from Google Maps' responses. Figure 5.2 shows the ServiceInvoker that was defined and the extensions it uses. The MapsProcessing extension is shown in Figure 5.2 on the application level. The code for both the GoogleMapsInvoker and MapsProcessing module can be found in Section A.4. This implementation was successful in retrieving driving directions and distance information for two test coordinates in the Netherlands.

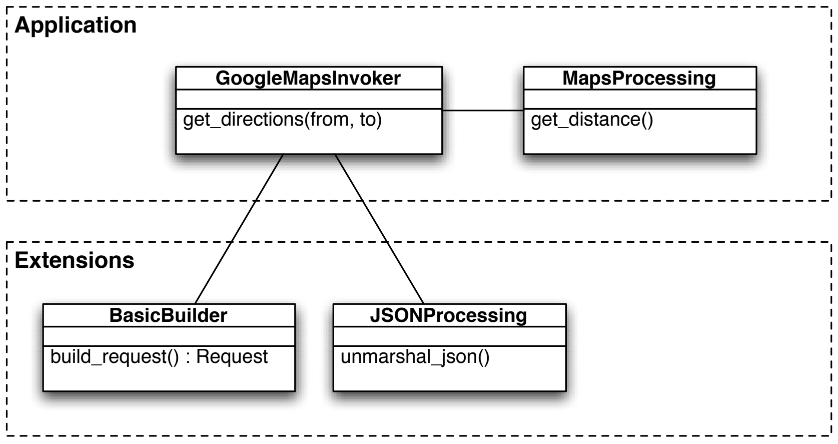

Figure 5.2: Implementation of Case 3: GoogleMapsInvoker and extensions.

5.6 Case 4: Invoking Flickr using XML-RPC

Flickr is a service that allows users to share photos. The photos on Flickr can be accessed with a web-browser, through a REST-ful service and through an RPC-style service using XML-RPC. In this case we invoke the Flickr service using XML-RPC. We call two procedures: one just returns the parameters we sent and the other returns links to the ten most recently posted photos on the website.

To perform this task we implement a subclass of ServiceInvoker::Base called FlickrInvoker. Figure 5.3 shows the FlickrInvoker and the extensions it uses. The FlickrInvoker uses the XMLRPCBuilder to generate XML-RPC requests, and the XMLRPCProcessing module to enable XML-RPC decoding. The FlickrInvoker also uses a custom processor called FlickrProcessing that was implemented specifically for decoding Flickr responses. Flickr responses use an elaborate encoding format that circumvents the constraints of XML-RPC. A flickr response contains a single string. This string contains an entire XML document, which is XML-escaped (meaning XML characters like '<' and '>' are replaced). FlickrProcessing extracts this string, decodes the contained XML document and converts the document to a set of Ruby objects. The code for both the FlickrInvoker and the FlickrProcessing module can be found in Section A.5.

The FlickrInvoker also employs a feature called callbacks that is included in the service_invoker framework. Using the before_request and after_request methods, application programmers can define functionality that is to be performed in specific stages of a request's life-cycle. In FlickrInvoker, this functionality is used to write the raw data of the request and the response to a log file.

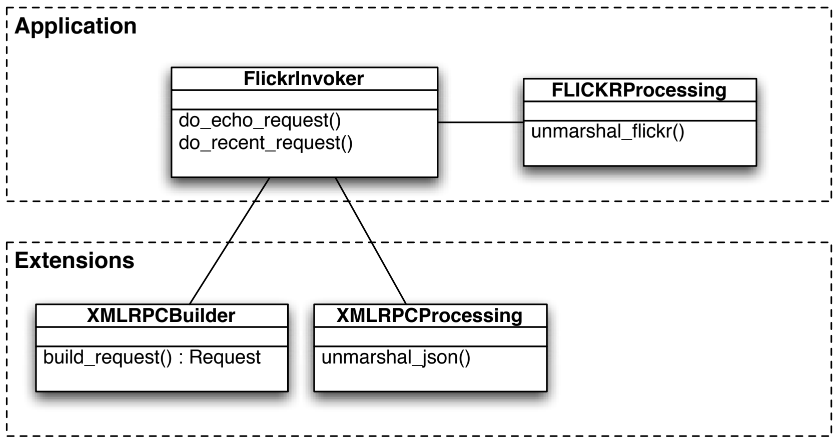

Figure 5.3: Implementation of Case 4: FlickrInvoker and extensions.

5.7 Case 5: Invoking Eventbrite using REST and HTTP

In this case we replace part of the functionality that was developed for the World Usability Day case study discussed in Section 3.5. As described in Section 3.5, the World Usability Day website interacts with a service exposed by Eventbrite. This service provides a REST-ful interface that is used to retrieve information about a specific set of events. In this case we implement the functionality for invoking Eventbrite from the World Usability Day site using service_invoker.

To perform this task we implement a subclass of ServiceInvoker::Base called WUDInvoker. Figure 5.3 shows the WUDInvoker and the extensions it uses. The WUDInvoker uses the BasicBuilder to generate requests. The instance of Basicbuilder it uses is configured to send requests to the URL that represents the set of events we want to retrieve. No application-specific functionality is needed to generate the request. A simple call to WudInvoker.invoke sends a GET request to the pre-configured URL.

The response from Eventbrite is an XML-encoded list of events. To decode this list the WUDInvoker uses the XMLProcessing module. It also uses the WUDProcessing module that was implemented specifically for this application. The WUDProcessing module performs the task of extracting the information we are interested in from the response, after the XMLProcessing module is used to convert the response into a set of Ruby objects. The code for both the WUDInvoker and WUDProcessing module can be found in Section A.6. This implementation was successful in retrieving relevant information about events from eventbrite.

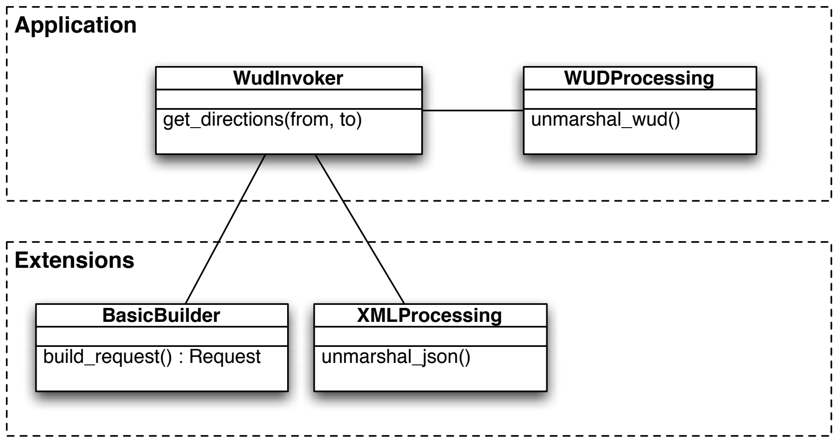

Figure 5.4: Implementation of Case 5: WUDInvoker and extensions.

5.8 Case 6: Invoking Thumbalizr using REST and HTTP

After the case study of Section 3.2 the portal product was extended to allow users to upload their own content. Possible content types included URL and document. A requirement was that previews of these content types could be shown. For this, two external services were used, Thumbalizr [55] and a service we will call Document Converter. The service_invoker was used to invoke these services. The implementation of this test case and the test case in the next section were used as part of the portal product.

Thumbalizr is an application that generates a snapshot of a website. It provides a REST-ful service. a GET request to the REST-ful Thumbalizr service returns a snapshot of a website. It may be necessary to do several requests before the result is returned, since generating a snapshot might take a while.

To perform this task we implement a subclass of ServiceInvoker::Base called ThumbalizrInvoker. Figure 5.5 shows the ThumbalizrInvoker and the extensions it uses. The WUDInvoker uses the BasicBuilder to generate requests. The FileProcessing response processor is used to store the response as a file. An application-specific method called get_thumbnail was implemented to easily request a thumbnail using only the URL of the desired website as a parameter. The code for the ThumbalizrInvoker can be found in Section A.7. This implementation was successful in retrieving thumbnails for URLs from Thumbalizr.

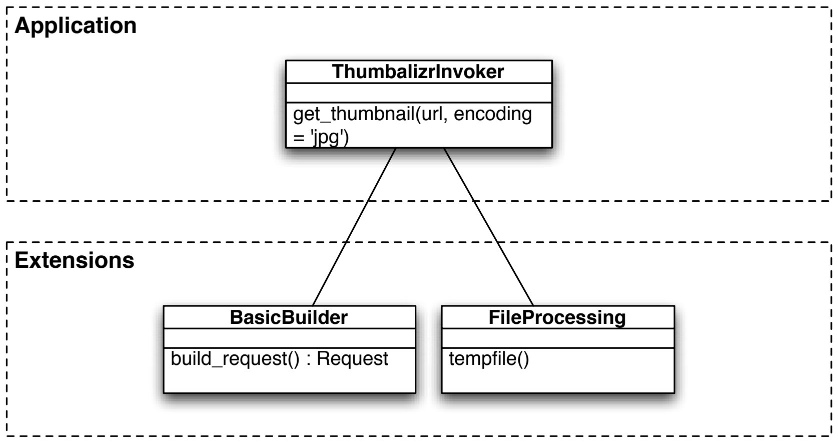

Figure 5.5: Implementation of Case 6: ThumbalizrInvoker and extensions.

5.9 Case 7: Invoking a document converter using REST and HTTP

The implementation of this test case was also used as part of the portal product, as explained in the previous section. The document converter service creates preview images from a document. One of these images is then used by the portal product to preview a document to the user. The document converter provides a REST-ful service. To process a document a set of URLs must be sent to the document converter. The document converter then downloads the document from one of the URLs, converts the document into a set of image files, and sends a job id to one of the other URLs. The application behind the second URL can then generate a URL from the job id and download the document from that URL. Figure 5.6 shows this interaction schematically. The interaction with the document converter service effectively shows that asynchronous communication is possible when using HTTP, provided that both applications can invoke *and* provide RESTful services.

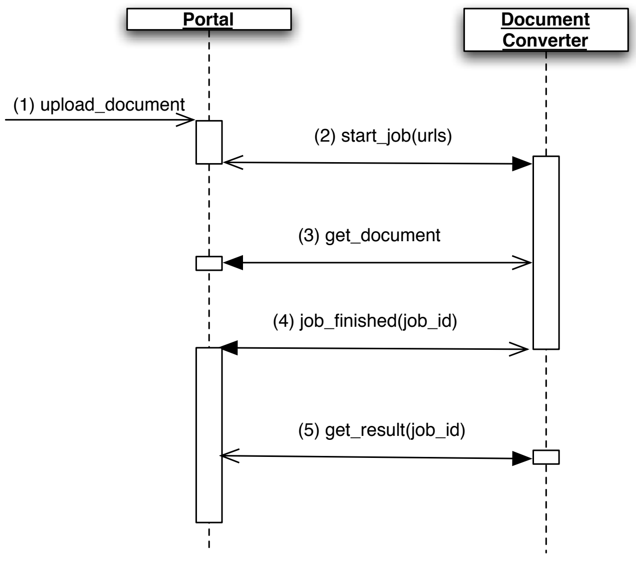

Figure 5.6: Document converter interaction pattern

Step 2 and 5 of Figure 5.6 were implemented in a subclass of ServiceInvoker::Base called DocumentInvoker. Figure 5.7 shows the DocumentInvoker and the extensions it uses. The DocumentInvoker only uses the BasicBuilder to generate requests. The response is processed by other functionality in the portal product. The DocumentInvoker contains two application-specific methods. The first one, start_job, implements step 2 of Figure 5.6. It tells the document converter to start a conversion job. It sends the URLs the document converter needs to perform step 3 and 4. The second application-specific method, get_result, implements step 5 of Figure 5.6. It retrieves the converted document from the document converter based on a job id.

The code for the DocumentInvoker can be found in Section A.8. This implementation was successful in starting jobs at- and retrieving results from the document converter.

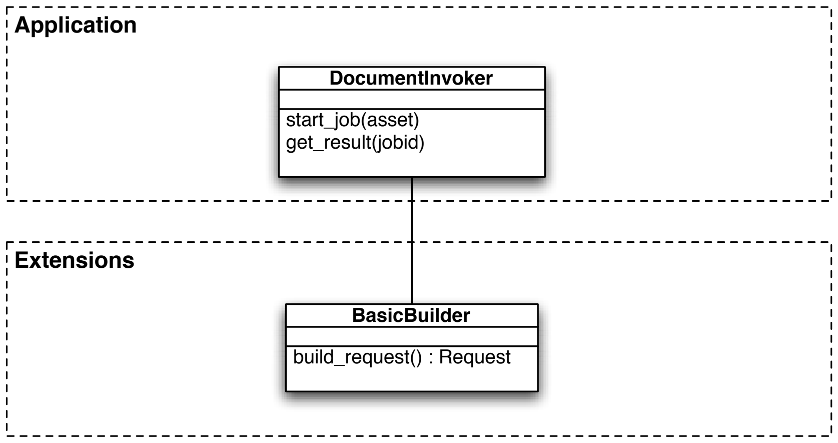

Figure 5.7: Implementation of Case 7: DocumentInvoker and extensions.

5.10 Summary

In this chapter we have outlined several test cases, and we have presented implementations of these test cases. These cases covered the invocation of REST-ful services over HTTP, as well as the provisioning and invocation of services using XML-RPC. All cases were implemented successfully using the framework.

Chapter 6

Evaluation

This chapter contains the evaluation of the framework proposed in this thesis. In this chapter we evaluate the various test implementations discussed in Chapter 5.

We analyze several technologies that can be used as alternatives to the framework in Section 6.1. We look at features of these alternative technologies and how they relate to the framework. We also indicate how these alternatives can be applied to the test cases. We then discuss advantages and disadvantages of the various technologies.

In Section 6.2 we present a survey that was conducted among developers at the Dutch software company at which we performed the case studies. In this survey, developers are asked to rate the test cases presented earlier on several quality attributes, as compared to several alternatives.

The summary presents an overview of the evaluation.

6.1 Alternatives

In this section we describe several alternatives to the framework. We look at several features of these alternatives and how they compare to the framework. We also look at how the test cases of Chapter 5 can be implemented using these alternatives, and how these implementations compare to the framework implementation.

6.1.1 Approach

For this evaluation, we look at several alternatives. In Section 2.4 we have given an overview of the components available for performing SOA tasks with Rails. Here we look at components that can be used for REST-ful or XML-RPC interaction. We specifically look at components that share features with our framework, and can thus be used as alternatives. The following components are considered:

ActionWebService (AWS): A Rails framework for providing and invoking services using XML-RPC and SOAP.

ActiveResource: A Rails framework for invoking certain REST-ful services.

HTTParty: A framework for invoking certain REST-ful services over HTTP.

RESTclient: A library for invoking certain REST-ful services over HTTP.

NET::HTTP: A Ruby standard library for HTTP interaction.

open-uri: A Ruby standard library for retrieving documents over HTTP.

All of these alternatives depend on the NET::HTTP library for HTTP interaction. They are all implemented in Ruby and can be used by the Rails platform, as described in Section 2.4.

We compare these alternatives to the framework by determining which of the following features they include:

- Support for REST-ful services using various data markup languages: XML, JSON, URL-encoded.
- Support for the XML-RPC or SOAP protocols.
- Support for configuration, meaning that the same or similar requests or responses can be sent multiple times after defining an initial set of parameters.
- Extensibility, meaning that it provides an extension mechanism for handling new protocols or data markup languages.

We also determine whether they are limited to the invoking or providing role.

We also compare three implementations from Chapter 5 to an alternative implementation that uses one of the components listed above. These implementations are compared mainly based on lines of code.

6.1.2 Results

Table 6.1 shows how the alternatives compare to the framework based on the features described in Section 6.1.1. It lists the service_provider and service_invoker along with the alternatives in the first column. The other columns represent the features as defined in Section 6.1.1. An X in a cell denotes that the corresponding component supports the corresponding feature.

For three cases there is an alternative implementation available, using one of the mentioned alternative components. Table 6.2 shows the alternatives that were used, and the lines of code required to build the alternative implementation. The code for the alternative implementation of the 'invoke XML-RPC' and 'provide XML-RPC' cases can be found in Section A.3 and Section A.2

6.1.3 Discussion

Based on the comparison with alternatives we argue that our framework has several advantages. In this section we discuss those advantages.

Uniform interface

A general advantage the framework has over its alternatives is that it combines several otherwise independent tasks and provides a uniform interface to the programmer. Table 6.1 shows how the service_invoker and service_provider combined support a wide range of protocols, unlike any of the alternatives. The framework also provides a fixed place for certain functionality regardless of the protocol used. This approach is comparable to how Rails provides a fixed place for most functionality. In Rails, this has been shown to improve productivity and facilitate learning. The service_invoker provides a general HTTP handler with features like raw data and header manipulation which are available to programmers through all protocol extensions.

No API file

The service_provider provides an alternative to ActionWebService, but it has several advantages over ActionWebservice. One advantage is that it does not require a separate API file. Exposed methods are described in the ActionController, where they are also defined. Table 6.2 shows that this does not necessarily lead to reduction in the size of the codebase.

		Features								
		Roles		REST-ful services						
		Invoking	Providing	URL-enc	XML	JSON	XML-RPC	SOAP	Configurable	Extensible
Components	service_provider		x				x		x	x
	service_invoker	x		x	x	x	x		x	x
	ActionWebService	x	x				x	x	x	
	ActiveResource	x			x				x	
	HTTParty	x			x	x			x	
	RESTclient	x		x						
	NET::HTTP	x	x							
	open-uri	x								

Table 6.1: Comparison of framework with alternatives (x = supported)

		Framework		Alternative	
		component	loc	component	loc
Cases	Case 1: Providing a 'hello' service	service_provider	11	ActionWebService	14
	Case 2: Invoking a 'hello' service	service_invoker	11	ActionWebService	12
	Case 5: Invoking Eventbrite	service_invoker	27	open-uri	~32

Table 6.2: Implementations of cases using framework and alternatives

The rules approach

The XML-RPC handler that was implemented for the service_provider takes a different approach to specifying methods than ActionWebService. The service_invoker does not require the programmer to explicitly declare exposed procedures. A set of procedures can be exposed by merely providing a routing rule that matches their procedure name. Based on this simplification we expected a reduction in the size of the codebase. However, reduction in codebase is not indicated by Table 6.2 because it is only achieved when exposing a large amount of procedures that can be easily described by a single rule. The 'provide XML-RPC' case only exposes a single procedure.

The rules approach taken by the service_provider achieves loose coupling between the exposed procedures and the handler methods. It allows for a method to be defined that catches all procedure calls not matched to other methods. A method can also be defined that handles all procedures that match a certain pattern. This gives application programmers a great deal of flexibility. A drawback of this approach is that it is impossible to explicitly state which procedures are exposed. This makes automatic generation of machine-processable interface definitions (like WSDL documents) hard to achieve.

Extensibility

The service_invoker can possibly handle any protocol for invoking services over HTTP, including XML-RPC and SOAP. It provides a mechanism that allows application programmers to decode service responses in any format. Alternatives like RESTclient, HTTParty, open-uri and NET::HTTP either include no decoding mechanism or can only be used to decode a limited number of formats, as shown in Table 6.1. None of these alternatives is extendible, so supporting new formats with these alternatives is difficult. For service_invoker we implemented decoding of the XML, JSON and URL-formatted responses, as shown in Table 6.1. It can also decode YAML responses and it provides a mechanism to handle the raw data in the response body as a file. It can also decode XML-RPC responses. These decoders can all be used independently, without prohibiting the use of other decoders.

The decoding mechanism of the service_invoker never loses original data after a decoding step. Application programmers can always use raw data or the result from any decoding step. This feature is not present in any of the alternatives.

The ThumbalizrInvoker that was implemented in Case 6 includes the FileProcessing module. This gives the response object a method `tempfile` which stores the body of the response as a file and returns a pointer to that file. This is useful when retrieving images that are later exposed to the user.

The service_invoker was used in the Document Converter case because it supports many different request and response formats. Sending requests required URL-encoded key-value pairs, while responses were either Files or simple HTTP responses with an empty body. Service_invoker can be used to handle each of these request and response formats.

Application programmers can specify their own decoding step, which may rely on another decoding step. This can provide a nice separation of concerns in many cases, since it allows programmers to put all functionality for providing the right response data in the right format into the service_invoker. This feature was used in the Google Maps, Flickr and Eventbrite cases.

Separation of Concerns

The Eventbrite case was an actual case from the World Usability Day project. The original code that was used for the case used open-uri to perform an HTTP request to the service directly, and then used several other libraries to process the results. The original implementation in the World Usability Day project required about 32 LOC. This is the alternative implementation mentioned in Table 6.2. The total size of the function that performed this operation however was about a 100 LOC. It also performed many other tasks, such as interpreting the event date and time information and synchronizing events with the database. It had several interwoven concerns. The code shown in Section A.6 separates the concern of retrieving and decoding relevant organizer and event data into the WudInvoker class. This should facilitate the implementation of other concerns.

6.2 Survey

6.2.1 Approach

For this survey we approached developers that work at the Dutch software company at which we conducted the case studies. These are experienced Ruby and Ruby on Rails developers with a college or university level education in Computer Science.

The translated text of the survey can be found in Appendix B. We first asked the subjects how experienced they where with several technologies that can be used as alternatives to the proposed frameworks. The purpose of this question was mainly to prime the right memories in the subjects. After this they are presented with several implementations of test cases, and asked to rate these implementations on several quality attributes. We present them with the following cases:

Case 1: Providing a 'hello' service using XML-RPC (Section 5.3).

Case 2: Invoking a 'hello' service using XML-RPC (Section 5.4).

Case 3: Invoking Google Maps using REST and HTTP (Section 5.5).

Case 4: Invoking Flickr using XML-RPC (Section 5.6).

Case 5: Invoking Eventbrite using REST and HTTP (Section 5.7).

The subjects where asked to consider how they would implement a case using another technology, and to give the provided implementations a rating between 1 and 9. They where asked to give a rating of 5 if they thought using this implementation over using an alternative would yield the same result with regard to a certain attribute. They were asked to give it a higher rating if they thought it was an improvement, and give a lower rating otherwise. This is an ordinal scale. The quality attributes are:

Understandability: Effort required to understand the function of the implementation.

Learnability: Effort required to learn how to use and replicate the implementation.

Reusability: Effort required to use the implementation as part of multiple project.

Maintainability: Effort required to change the implementation.

Extensibility: Effort required to add functionality to the implementation.

Tailorability: Effort required to configure the implementation for a more specific purpose.

Simplicity: General conciseness and lack of complexity of the implementation.

Testability: Effort required to test the implementation.

We assumed the meaning of these attributes was well understood by the subjects. These attributes are not orthogonal. If classified according to the ISO 9126 standard [56], understandability and learnability both fall under the usability category. In this standard, testability is an attribute of maintainability. Reusability, extensibility, tailorability and simplicity are not included in that standard. Simplicty is of influence to many other quality attributes. Tailorability, extensibility and reusability overlap, and they can be classified under the portability attribute in the ISO 9126 standard. From a rating for the portability attribute we can draw conclusions on how useful the framework is in different environments, which is an important quality of a framework.

6.2.2 Results

The survey had 5 participants. Table 6.3 shows the median rating of every case per quality attribute. A rating of 5 is neutral. Everything above that is positive, everything below that is negative. The median of a set of ratings is the highest of the lowest 50% of the ratings. We use the median because the cases were rated on an ordinal scale.

	Google Maps	Flickr	Eventbrite	Providing XML-RPC	Invoking XML-RPC	**Median**
Understandability	7	4	7	8	8	**7**
Learnability	7	6	7	8	8	**7**
Reusability	8	7	7	7	7	**7**
Maintainability	7	7	7	8	8	**7**
Extensibility	8	7	6	8	8	**7**
Tailorability	6	7	6	7	7	**7**
Simplicity	7	5	7	7	8	**7**
Testability	7	7	7	8	8	**7**
Median	**7**	**7**	**7**	**8**	**8**	**7**

Table 6.3: Median rating by developers per case, per quality attribute. (on a scale of 1 to 9, where 1 is bad, 5 is neutral, and 9 is excellent)

Figure 6.1 shows a boxplot for every quality attribute over all cases. This can be seen as a general rating for the framework on these quality attributes. A boxplot is constructed as follows: the plot starts at the lowest rating that was given. The start of the box is at the highest of the lower 25% of the ratings. The line in the box is the median. The end of the box is the lowest of the upper 25% of the ratings. The end of the plot is the highest rating that was given.

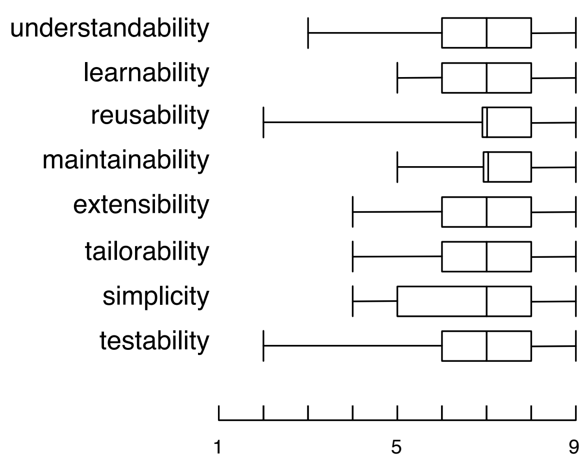

Figure 6.1: boxplot of ratings for quality attributes over all cases. (on a scale of 1 to 9, where 1 is bad, 5 is neutral, and 9 is excellent)

6.2.3 Discussion

When interpreting the results we have to take into account the following threats to their validity:

Low number of participants: The survey had only 5 participants, making the role of statistical outliers more prominent.

Interpretation of quality attributes: Participants may have had divergent interpretations of the various quality attributes.

Learning effect: The order of the cases may have caused a learning effect which may have caused participants to rate cases later in the survey different from earlier cases.

Almost all implementations scored above neutral on all attributes. Case 1 and 2, the 'hello service' cases, scored especially high on almost all attributes. These were the shortest implementations in terms of LOC, and the simplest cases. We expected them to score high on understandability and learnability. We also argue that the way our framework handles XML-RPC procedures is very intuitive specifically for Ruby programmers. This is due to the dynamic nature of Ruby, in which multiple method calls, like XML-RPC procedure calls in the framework, can be handled by a single method definition.

The two cases involving the invocation of REST-ful services, the Google Maps and Eventbrite cases, score significantly lower than the cases involving XML-RPC interaction. This can be explained by the wide range of REST-ful services that are supported. This in combination with the uniform platform approach of the service_invoker makes the application code more complex.

Developers rated the implementation of the Google Maps case high in terms of extensibility and reusability. This was expected, since the implementation provides many possibilities for extracting additional information from the response. The ratings for understandability were lower. This may be because the implementation uses generic functionality to process the result, and because there are no functions in the framework that are specifically tailored to the Google Maps API.

Developers rated the implementation of the Flickr case low in terms of understandability and simplicity. This can be explained by the complicated encoding of Flickr responses, which leads to complicated response processing. The survey also shows that developers rate this implementation high in terms of extensibility. This is probably because it provides a basis which can be used to call other procedures of the Flickr XML-RPC service.

Developers rated the implementation of the Eventbrite case relatively low with regard to reusability and extensibility. This may be because it can only invoke a single resource and processes responses in an application-specific way.

6.3 Summary

In this chapter we evaluated the framework proposed in Chapter 4. We have compared it to several alternative technologies. Based on this evaluation we have described several advantages of the framework over alternatives.

Where possible, we have also discussed implementations of the test cases described in Chapter 5 using alternative technologies. In these cases, the same amount or less lines of code were needed for the implementation using the framework.

For some of the test implementations described in Chapter 5 we have gathered feedback from application programmers with regard to the following quality attributes: understandability, learnability, reusability, maintainability, extensibility, tailorability, simplicity and testability.

Almost all implementations scored above average on all attributes. The service_invoker and service_provider scored especially high on interactions using XML-RPC. Scores were lower in the cases that involved REST-ful services.

Chapter 7

Conclusions

In this chapter we present the conclusions of this research. The central problem that this research deals with is that Rails is unsuitable for a role in a service-oriented system. The main objectives are to better define the limitations of Rails with regard to SOA, and to propose a framework that deals with the most important of these limitations. In Section 7.1 we describe how limitations to Rails with regard to SOA were discovered and what these limitations are. In Section 7.2 we describe the framework that was developed to be able to deal with these limitations, and how it deals with these limitations. In Section 7.3 we discuss the effectiveness of our framework in dealing with these limitations based on an evaluation that was performed.

7.1 Limitations of Rails

A Rails application is part of a SOA when it provides or invokes a service. In this thesis we distinguished three service interaction styles:

- Message-Oriented
- Remote Procedure Call (RPC)
- Representational State Transfer (REST)

We identified several protocols that can be used to support these interaction styles. Through case studies, we have identified several limitations of Ruby on Rails with regard to these interaction styles and protocols.

We have studied four medium-sized projects at a Dutch software company. The goal of these studies is to identify problems that were encountered when developing Rails applications that interact with other systems in a SOA.

These case studies have shown that Rails' ability to invoke REST-ful services is limited and insufficient in many cases. We also conclude from the case studies that Rails' ability to invoke and provide RPC-style services is limited and poorly integrated. Message-oriented services were not used in the cases, so we drew no conclusions with regard to support for these services in Rails.

7.2 A Framework for Service-Oriented Extensions to Ruby on Rails

In this thesis we proposed a framework for Service-Oriented Extensions to Ruby on Rails. One of the limitation of Rails we encountered was lack of functionality for invoking REST-ful services. A goal of our framework has been to enable Rails to invoke any type of service over HTTP, including any REST-ful service.

Another limitations of Rails we encountered was poorly integrated functionality for invoking and providing RPC-style services. One of the goals of our framework has been to provide a generic extension mechanism to Rails for integrating this functionality.

Our framework consists of two parts:

service_invoker allows Rails to invoke any type of service over HTTP. Protocol handlers for XML-RPC and a broad range of REST-ful services were implemented for this framework.

service_provider allows Rails to provide services of any type over HTTP. It builds on Rails' built-in functionality to process HTTP requests. Service_provider mainly focusses on RPC-style services. A protocol handler for XML-RPC was implemented.

7.3 Framework Evaluation

We have implemented seven test cases to determine how effective our framework is with regard to its goals. These goals were providing and invoking services using XML-RPC and invoking REST-ful services. The test cases covered tasks related to these goals. All seven cases were implemented successfully using the framework.

We have also looked at existing technologies that can serve as an alternative to the framework. We showed that the following components overlap in functionality with our framework:

- Restclient
- HTTParty
- NET::HTTP
- NET::XML-RPC

- ActionWebService

By evaluating these alternatives we identified several advantages of our framework:

Uniform interface: The framework provides a uniform interface for providing and invoking services using a wide range of protocols.

Extensibility: The service_invoker and service_provider allow application developers to use a wide range of protocols. They do this by allowing application programmers to implement extensions that contain encoding and decoding mechanisms for protocol messages. Extensions for XML-RPC and a wide range of REST-ful protocols were developed.

The rules approach: The service_provider and the XML-RPC extension achieve loose coupling between procedure calls and handler methods by allowing application programmers to specify rules that determine which methods are exposed.

Separation of Concerns: The framework allows application programmers to put most functionality related to invoking and providing services into extensions, providing a good separation of concerns.

Reduction in size of codebase: In three of the test cases, we performed LOC measurements on equivalent implementations using either our framework or an alternative. In all three cases, the implementation based on our framework used slightly less lines of code.

For five of the seven test implementations we have conducted a survey to gather feedback from application programmers. Application programmers were asked to rate the test implementations with regard to the following quality attributes: understandability, learnability, reusability, maintainability, extensibility, tailorability, simplicity and testability. The survey has shown that application programmers rate all the test implementations relatively high on each of these attributes.

Ratings were especially high on cases where simple services were invoked or provided over XML-RPC. This was explained using the size of their codebase. The size of the codebase was bigger for test cases involving REST-ful services, and the scores for these test cases were lower. A broad range of REST-ful services was tested, each with their own encoding format and representation semantics. This can make application code built on a generic platform such as the service_invoker more complex, and may explain the lower scores and bigger codebase.

7.4 Future Work

The plugins we built in this thesis are meant to be extended. Support for message-oriented interactions is a possible target for future work. Adding support for the SOAP 1.2 protocol will make that possible.

The service_provider has not been evaluated in details. It was only used for providing a service to another Rails application using XML-RPC. Its effectiveness in providing services with other protocols to applications on other platforms needs to be evaluated in future work.

Another direction for future work is to examine the possibilities of using other protocols for transport. With our extensions, Rails can still only provide and invoke services over HTTP. It is not possible yet to use it to invoke or provide services using another application-layer Internet protocol.

Other topics for future work are how programmers use the framework developed in this thesis and how it fits into an agile development process. In a broader sense, one could look at how an agile development process using Rails is influenced by the requirements of service-orientation.

Appendix A

Code Samples

This appendix contains code samples from the test cases used in Section 6. Note:

- Pieces of code that are not suitable for publication in this thesis are replaced by the text `<anonymized>`
- Pieces of code that are abbreviated are suffixed with ...

A.1 Invoking a 'hello' service using XML-RPC

This was used to test communication between the service_provider and service_invoker.

```
class ProviderInvoker < ServiceInvoker::Base
  request_builder XMLRPCBuilder,
    :endpoint_url => "<anonymized>"

  response_module XMLRPCProcessing
  response_module XMLProcessing

  def self.do_greeting_request name
    self.invoke(:name => 'do_greeting', :args => [name])
  end
end
```

A.2 Providing a 'hello' service using XML-RPC with AWS

This was used to compare service_provider with ActionWebService. It provides a service similar to the code in Section 4.5

```
# greeting_api.rb
class GreetingApi < ActionWebService::API::Base
  inflect_names false

  api_method :do_greeting,
    :expects => [:string],
    :returns => [:string]
end

# greeting_controller.rb
class GreetingController < ApplicationController
  def do_greeting(username="nobody")
      "Hello #{username}"
  end
end
```

A.3 Invoking a ‘hello’ service using XML-RPC with AWS

This was used to compare service_invoker with ActionWebService. It invokes the same service as the code in Section A.1.

```
# greeting_api.rb
class GreetingApi < ActionWebService::API::Base
  inflect_names false

  api_method :do_greeting,
    :expects => [:string],
    :returns => [:string]
end

# aws_controller.rb
class AwsController < ApplicationController
  web_client_api :greeting, :xmlrpc, "http://localhost:4000/greeting/api"#, :ha

  def index
    @result = greeting.do_greeting 'stefan'
  end
end
```

A.4 Invoking Google Maps using REST and HTTP

This was used to retrieve directions and driving distance from google maps.

```
module MapsProcessing
  def get_distance
    unmarshal_json['Directions']['Distance']['meters']
  end
end

class GoogleMapsInvoker < ServiceInvoker::Base
  request_builder BasicBuilder,
    :endpoint_url => "http://maps.google.com/maps/nav",
    :default_get_params => {
"key" => "<anonymized>"}

  response_module JSONProcessing
  response_module MapsProcessing

  def self.get_directions(from, to)
    self.invoke(:url_params => {"q" => "from: @#{from} to: @#{to}"})
  end
end
```

A.5 Invoking Flickr using XML-RPC

This code was used to retrieve the last 10 uploaded fotos from Flickr

```
module FLICKRProcessing
  def unmarshal_flickr
    Hash.from_xml(CGI::unescapeHTML(unmarshal_xmlrpc))
  end
end

class FlickrInvoker < ServiceInvoker::Base
  request_builder XMLRPCBuilder,
    :endpoint_url => "api.flickr.com/services/xmlrpc/"

  response_module XMLRPCProcessing
  response_module FLICKRProcessing

  before_request {|c| logger.info(c.request.raw_data)}
  after_request {|c| logger.info(c.response.raw_data)}

  def self.do_echo_request
    Struct.new('EchoStruct', :api_key, :name, :name2)
    args = Struct::EchoStruct.new('<anonymized>', 'value', 'value2')
    self.invoke(:name => 'flickr.test.echo', :args => [args])
  end

  def self.do_recent_request
    Struct.new('Recent', :api_key, :per_page, :page)
    args = Struct::Recent.new('<anonymized>', '10', '1')
    self.invoke(:name => 'flickr.photos.getRecent', :args => [args])
  end
end
```

A.6 Invoking Eventbrite using REST and HTTP

This code was used to retrieve information about events at the World Usability Day from the Eventbrite REST-style service.

```
module WUDProcessing
  def unmarshal_wud
    xml = unmarshal_xml
    result = {}
    xml["rsp"]["user"]["organizers"]["organizer"].collect do |o|
      result[o["name"]] = get_organizer_events(o)
    end
    result
  end

  private
  def get_organizer_events(organizer)
    return nil if organizer["events"].nil?
    return [organizer["events"]["event"]]
        if organizer["events"]["event"].is_a? Hash
    return organizer["events"]["event"]
        if organizer["events"]["event"].is_a? Array
  end
end

class WudInvoker < ServiceInvoker::Base
  request_builder BasicBuilder,
    :endpoint_url => "www.eventbrite.com/rest/user_list_events/<anonymized>"

  response_module XMLProcessing
  response_module WUDProcessing
end
```

A.7 Invoking Thumbalizr using REST and HTTP

This code was used to interact with the Thumbalizr service in the portal case

```
class ThumbalizrInvoker < ServiceInvoker::Base
  request_builder BasicBuilder,
    :endpoint_url => 'http://api.thumbalizr.com/',
    :default_get_params => {"api_key" => "<anonymized>"}

  response_module FileProcessing

  def self.get_thumbnail(url, encoding = 'jpg')
    result = self.invoke(
      :url_params => {:url => url, :encoding => encoding}
      )
    result.response
  end
end
```

A.8 Invoking a document converter using REST and HTTP

This code was used to interact with the document converter service in the portal case.

```
class DocumentInvoker < ServiceInvoker::Base
  request_builder BasicBuilder,
    :endpoint_url => <anonymized>

  # send a request to start the job
  def self.start_job(asset)
    self.invoke(
        :method => :post,
        :url_path => <anonymized>,
        :user => <anonymized>,
        :password => <anonymized>,
        :headers => {'Content-Type' => 'text/plain'}
        :post_params => {
          :docUrl => "<anonymized>",
          :pingStart => "<anonymized>",
          :pingDone => "<anonymized>",
          :pingError => "<anonymized>",
   ...
```

```
        }
      )
  end

  def self.get_result(jobid)
    result = self.invoke(
      :method => :get,
      :url_path => <anonymized>,
      :user => <anonymized>,
      :password => <anonymized>
    )
    result.response
  end
end
```

Appendix B

Survey

This is the survey that was conducted to evaluate the framework developed in Chapter 4. It was presented to programmers as a plain text file with some markers where they could fill in their answers. It was originally in dutch, this is the translated version. It contains references to the code samples in Appendix A. These could be accessed through hyperlinks in the original survey.

On a scale of 1 to 5, rate your familiarity with the following tools, where 1 is never heard of and 5 is used in production

ActionWebService	[ ]
ActiveResource	[ ]
ruby Net::HTTP	[ ]
xmlrpc4r	[ ]

On a schale of 1 to 9, rate the following code snippets on the mentioned 'ilities' as oposed to an implementation where one of the technologies mentioned above was used. 1 means the alternative is better and 9 means the presented snippet is better. The code in the snippets has been written to test the service_invoker and service_provider plugins from my thesis. If you don't know how to implement a snippet using one of the technologies above you can fill in 'don't know'. Use this as a last resort however, if you have any idea how to implement an alternative try and answer it anyway. Contact me if one of the ilities is not clear.

To get a good picture of what this code does I recommend you download the two demo applications. These can be found in <anonymized>. These are full-blown Rails 2.1 applications that use the latest versions of my plugins, including documentation. All the code below is used in these applications. See app/models, app/controllers and routes.rb. For the XML-RPC examples: The invoker application looks for a service on port 4000 by default, so start the provider application there. Contact me if something doesn't work.

If there are any other reasons why you like the snippets or an alternative please mentioned those under the survey. Suggestions for improvements and additions are also welcome.

Google Maps Invoker
Retrieve directions and driving distance from google maps
See Section A.4

Understandability	[ ]
Learnability	[ ]
Reusability	[ ]
Maintainability	[ ]
Extensibility	[ ]
Tailorability	[ ]
Simplicity	[ ]
Testability	[ ]

Flickr invoker
Retrieve 10 most recent photos from flickr
See Section A.5

Understandability	[ ]
Learnability	[ ]
Reusability	[ ]
Maintainability	[ ]
Extensibility	[ ]
Tailorability	[ ]
Simplicity	[ ]
Testability	[ ]

WUD invoker
Retrieve event information for the world usability day
See Section A.6

Understandability	[ ]
Learnability	[ ]
Reusability	[ ]
Maintainability	[ ]
Extensibility	[ ]
Tailorability	[ ]
Simplicity	[ ]
Testability	[ ]

XML-RPC providing
Provide a simple ‘hello(name)’ service
See Section 4.5

Understandability	[ ]
Learnability	[ ]
Reusability	[ ]
Maintainability	[ ]
Extensibility	[ ]
Tailorability	[ ]
Simplicity	[ ]
Testability	[ ]

XML-RPC invoking
Invoke aforementioned ‘hello(name)’ service See Section A.1

Understandability	[ ]
Learnability	[ ]
Reusability	[ ]
Maintainability	[ ]
Extensibility	[ ]
Tailorability	[ ]
Simplicity	[ ]
Testability	[ ]

Bibliography

[1] S. Ambler, "Mapping objects to relational databases: O/r mapping in detail," 2006. [Online]. Available: http://www.agiledata.org/essays/mappingObjects.html

[2] H. He, "What is service-oriented architecture," 2003. [Online]. Available: http://webservices.xml.com/pub/a/ws/2003/09/30/soa.html

[3] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana, "Business process execution language for web services, v1.1," BEA, IBM, SAP, Siebel, Specification, 2003. [Online]. Available: http://www.ibm.com/developerworks/library/specification/ws-bpel/

[4] L. F. Cabrera, G. Copeland, M. Feingold, R. W. Freund, T. Freund, S. Joyce, J. Klein, D. Langworthy, M. Little, F. Leymann, E. Newcomer, D. Orchard, I. Robinson, T. Storey, and S. Thatte, "Web services coordination (ws-coordination), v1.0," IBM, BEA Systems, Microsoft, Arjuna, Hitachi, IONA, Specification, 2005. [Online]. Available: http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-tx/WS-Coordination.pdf

[5] ——, "Web services atomic transaction (ws-atomic transaction), v1.0," IBM, BEA Systems, Microsoft, Arjuna, Hitachi, IONA, Specification, 2005. [Online]. Available: http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-tx/WS-AtomicTransaction.pdf

[6] ——, "Web services business activity framework (ws-business activity) 1.0," IBM, BEA Systems, Microsoft, Arjuna, Hitachi, IONA, Specification, 2005. [Online]. Available: http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-tx/WS-BusinessActivity.pdf

[7] A. Rustad, "Ruby on rails and j2ee: Is there room for both? two web application frameworks compared," 2005. [Online]. Available: http://www.ibm.com/developerworks/linux/library/wa-rubyonrails/

[8] "Nedforce," Website, 2008. [Online]. Available: http://www.nedforce.com/

[9] B. Tate, “Crossing borders: What’s the secret sauce in ruby on rails,” 2006. [Online]. Available: http://www.ibm.com/developerworks/java/library/j-cb05096.html

[10] A. Hunt and D. Thomas, *Programming Ruby, 2nd edition.* The Pragmatic Bookshelf, 2007.

[11] ——, “A conversation with andy hunt and dave thomas, part vi - programming close to the domain,” 2003. [Online]. Available: http://www.artima.com/intv/domain2.html

[12] R. E. Johnson and B. Foote, “Designing reusable classes,” *Journal of Object-Oriented Programming, Volume 1, Number 2*, 1988.

[13] K. Ramirez, “Ioc container face-off,” 2005. [Online]. Available: http://today.java.net/pub/a/today/2005/02/10/ioc.html

[14] M. Fowler, “Inversion of control,” 2005. [Online]. Available: http://martinfowler.com/bliki/InversionOfControl.html

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns.* Addison Wesley, 1995.

[16] G. Seshadri, “Understanding javaserver pages model 2 architecture,” 1999. [Online]. Available: http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html

[17] Wikipedia, “Create, read, update and delete,” 2007. [Online]. Available: http://en.wikipedia.org/wiki/Create,_read,_update_and_delete

[18] J. Nandhakumar and J. Avison, “The fiction of methodological development: a field study of information systems development,” *Information Technology People 12(2): 179-191*, 1999.

[19] D. P. Truex, R. Baskerville, and J. Travis, “Methodical systems development: The deferred meaning of systems development methods,” *Accounting, Management and Information Technology 10: 53-79*, 2000.

[20] P. Abrahamsson, O. Salo, J. Warsta, and J. Ronkainen, “Agile software development methods, review and analysis,” *VTT Publications 478*, 2002. [Online]. Available: www.vtt.fi/inf/pdf/publications/2002/P478.pdf

[21] K. Schwaber and M. Beedle, *Agile Software Development With Scrum.* Prentice-Hall, 2002.

[22] K. Beck, “Embracing change with extreme programming,” *IEEE Computer, vol. 32*, 1999.

[23] A. Hunt and D. Thomas, *The Pragmatic Programmer.* Addison Wesley, 2000.

[24] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Manifesto for agile software development," 2001. [Online]. Available: http://agilemanifesto.org/

[25] P. Abrahamsson, J. Warsta, M. Siponen, and J. Ronkainen, "New directions on agile methods: A comparative analysis," *IEEE 5-1877-X/03*, 2003.

[26] P. Prescod, "Roots of the rest/soap debate," 2002. [Online]. Available: http://www.prescod.net/rest/rest_vs_soap_overview

[27] T. Berners-Lee, R. Fielding, U. Irvine, and L. Masinter, "Uniform resource identifiers (uri): Generic syntax," IETF, RFC, 1998. [Online]. Available: http://www.ietf.org/rfc/rfc2396.txt

[28] Amazon, "Amazon simple storage service developer guide (api version 2006-03-01)," 2006. [Online]. Available: http://docs.amazonwebservices.com/AmazonS3/2006-03-01/

[29] Google, "Google products (apis)," 2008. [Online]. Available: http://code.google.com/more/#label=APIs

[30] D. Winer, "Xml-rpc," xmlrpc.com, Specification, 1999. [Online]. Available: http://www.xmlrpc.com/spec

[31] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon, "Soap version 1.2 part 1 messaging framework (second edition)," World Wide Web Consortium, Recommendation, 2007. [Online]. Available: http://www.w3.org/TR/soap12-part1

[32] ——, "Soap version 1.2 part 2: Adjuncts (second edition)," World Wide Web Consortium, Recommendation, 2007. [Online]. Available: http://www.w3.org/TR/soap12-part2/

[33] Y. Shohoud, "Rpc/literal and freedom of choice," 2003. [Online]. Available: http://msdn.microsoft.com/en-us/library/ms996466.aspx

[34] L. Clement, A. Hately, C. von Riegen, and T. Rogers, "Uddi version 3.0.2," OASIS, Specification, 2004.

[35] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services: Concepts, Architectures and Applications.* Springer, 2004.

[36] K. Ballinger, D. Ehnebuske, M. Gudgin, M. Nottingham, and P. Yendluri, "Basic profile version 1.0," The Web Services-Interoperability Organization, Specification, 2004. [Online]. Available: http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html

[37] D. Box, E. Christensen, F. Curbera, D. Ferguson, J. Frey, M. Hadley, C. Kaler, D. Langworthy, F. Leymann, B. Lovering, S. Lucco, S. Millet, N. Mukhi, M. Nottingham, D. Orchard, J. Shewchuk, E. Sindambiwe, T. Storey, S. Weerawarana, and S. Winkler, "Web services addressing (ws-addressing)," World Wide Web Consortium, Submission, 2004. [Online]. Available: http://www.w3.org/Submission/ws-addressing/

[38] A. Wiggins, "Rest-client," Website, 2008. [Online]. Available: http://github.com/adamwiggins/rest-client/tree/master

[39] J. Nunemaker, "httparty," Website, 2008. [Online]. Available: http://github.com/jnunemaker/httparty/tree/master

[40] J. Forder, "Actionwebservice in ruby on rails," 2006. [Online]. Available: http://wiki.rubyonrails.org/rails/pages/ActionWebService

[41] M. Neumann, "xmlrpc4r - xml-rpc for ruby," Website, 2002. [Online]. Available: http://www.fantasy-coders.de/ruby/xmlrpc4r/

[42] H. Nakamura, "soap4r," Website, 2000. [Online]. Available: http://dev.ctor.org/soap4r

[43] WSO2, "Wso2 web services framework for ruby," Website, 2008. [Online]. Available: http://wso2.com/products/wsfruby/

[44] R. K. zuir Yin, *Case study research: design and methods, 3rd edition.* Sage Publications, 2003.

[45] T. Fuller and S. Morgan, "Data replication as an enterprise soa antipattern," 2006. [Online]. Available: http://msdn2.microsoft.com/en-us/library/bb245678.aspx

[46] D. Raggett, A. L. Hors, and I. Jacobs, "W3c html 4.01 specification," World Wide Web Consortium, Recommendation, 1999. [Online]. Available: http://www.w3.org/TR/REC-html40/

[47] D. Winer, "Rss 2.0," Berkman Center, Specification, 2003. [Online]. Available: http://cyber.law.harvard.edu/rss/rss.html

[48] Apple, "Podcasting faq," 2007. [Online]. Available: http://docs.info.apple.com/article.html?artnum=301880

[49] D. Winer, "Opml 1.0," OPML, Specification, 2007. [Online]. Available: http://www.OPML.org/spec

[50] Apple, "Tips for podcast fans," 2008. [Online]. Available: http://www.apple.com/itunes/store/podcaststips.html

[51] J. Kraemer, "Act as ferret," 2008. [Online]. Available: http://projects.jkraemer.net/acts_as_ferret/wiki

[52] E. Weaver, "Has many polymorphs," 2008. [Online]. Available: http://blog.evanweaver.com/files/doc/fauna/has_many_polymorphs/

[53] "World usability day," Website, 2008. [Online]. Available: http://www.worldusabilityday.org

[54] "Eventbrite," Website, 2008. [Online]. Available: http://www.eventbrite.com/

[55] "Thumbalizr," Website, 2008. [Online]. Available: http://www.thumbalizr.com/

[56] "Iso/iec 9126 : Information technology - software product evaluation - quality characteristics and guidelines for their use," ISO/IEC, Reference, 1991. [Online]. Available: http://www.cse.dcu.ie/essiscope/sm2/9126ref.html

Index