The design of a RESTful web-service

NADIA MOHEDANO TROYANO

KTH Electrical Engineering

Master's Degree Project
Stockholm, Sweden

XR-EE-LCN 2010:003

KUNGLIGA TEKNISKA HÖGSKOLAN
SCHOOL OF ELECTRICAL ENGINEERING
TNSSM

MASTER THESIS

THE DESIGN OF A RESTFUL WEB-SERVICE

STUDENT: Nadia Mohedano Troyano

EXTERNAL SUPERVISOR: Torbjörn Törnkvist, Klarna AB

EXAMINER: Viktoria Fodor

DATE: June 2010

Acknowledgements

Firstly I would like to thank my academic advisor Professor Viktoria Fodor.

I would like to specially thank *Klarna AB* for the opportunity they gave me and the resources that had put on me to make this project. Thanks to Torbjörn Törnkvist, my supervisor at Klarna, for his constructive advice, and always fast and profitable feedback when I delivered my problems to him. His continuous support and guidance was essential in the development of this thesis.

I would also like to thank my parents, José Joaquín and Ángela, and my brother Jose Ángel for their unconditional moral support although not being here. And specially Jordi, who has been always by my side and who has always given me good advice. And also thanks to Luis, Mari Carmen, Laura, Policarpa, Hipolita and Hipamia.

Nadia Mohedano Troyano

In a well-designed RESTful service, everything does what it says.

Contents

1 Introduction **5**

1.1 Financial systems today 5

1.2 Goals 6

1.3 Outline of the Thesis 6

2 REST **7**

2.1 Background on WEB Services 7

2.2 REST 9

2.2.1 Resources 9

2.2.2 Addressability 10

2.2.3 Statelessness 11

2.2.4 Connectedness 11

2.2.5 Uniform interface 12

2.2.6 Cacheable 13

2.3 Why REST 15

3 Klarna environment **16**

3.1 Workflow 16

3.2 Erlang 16

3.2.1 Erlang's Characteristics 17

3.3 XML-RPC 18

3.4 The Klarna system 19

3.5 Goals 20

4 Designing the REST API **22**

4.1 Resources 22

4.2 Addressability 23

4.3 Statelessness 24

4.4 Connectedness 24

4.5 Uniform Interface 25

4.6 Cacheability 26

4.7 The REST API structure . . . 27
4.8 Sequence Diagrams . . . 29
4.8.1 Get invoices . . . 29
4.8.2 Get invoice . . . 29
4.8.3 Create invoice . . . 31
4.8.4 Update/Modify invoice . . . 31
4.8.5 Delete invoice . . . 31
4.8.6 Search invoices . . . 35

5 Prototype **37**
5.1 Introduction . . . 37
5.2 CouchDB . . . 38
5.3 Mochiweb . . . 38
5.4 The prototype . . . 38

6 Prototype integration **43**
6.1 Mnesia . . . 43
6.2 Yaws . . . 44
6.3 Prototype integration . . . 45
6.4 Testing with Curl . . . 46
6.5 Testing throguh GUI . . . 49

7 Conclusions **55**
7.1 Future work . . . 55

List of Figures

2.1 Two resources, the same URI. 10
2.2 Two different URIs refer to the same data. 10
2.3 RESTful service well connected 12

3.1 The Klarna System 20

4.1 Cache process 27
4.2 Klarna System with the REST API 27
4.3 Design of the structure of the REST API 28
4.4 Get invoices sequence 30
4.5 Get invoice sequence 32
4.6 Create invoice sequence 33
4.7 Modify invoice sequence 34
4.8 Search invoices sequence 36

5.1 Prototype 38
5.2 Function run_controller from the dispatch module 39
5.3 Function dispatch from a controller module 40
5.4 Function controller_top from the invoices_controller module 41
5.5 Function controller_invoice from the invoices_controller module . . . 42

6.1 Prototype inside the Klarna System 45
6.2 Detailed prototype inside the Klarna System 46
6.3 User Interface Home Page 50
6.4 New Invoice Page 50
6.5 New Person Form 50
6.6 Retrieved List of Persons 51
6.7 Search List of Persons From Sweden 51
6.8 Search List of Persons From Sweden (maximum 2 persons) 52
6.9 Get Invoice 52
6.10 Get Estore 53
6.11 Update Estore 54

Chapter 1

Introduction

This chapter starts with the motivation for doing this thesis which includes the problems presented by financial systems and web-based systems. Following it is introduced Klarna AB, the company where the thesis has been developed, and the goals in general terms are presented.

1.1 Financial systems today

A financial system allows the transfer of money between savers and borrowers. It comprises a set of complex and closely interconnected financial institutions, markets, services and transactions. By transcaction is understood the event or process initiated or invoked by a user or computer program, regarded as a single unit of work and requirering a record to be generated for processing in a database. In this environment, a transaction should be regarded as a single unit of work and must either be processed in their totality or rejected as a failed transaction. Furthermore, a process that records every transaction has to be established since it is mandatory for the financial company to be able to demonstrate what funds were received and how funds were spent.

Most of these financial systems are nowadays web-based systems which refers to those applications or services that are resident on a server that is accessible using a Web browser and is therefore accessible from anywhere in the world via the Web [1]. These web-based financial systems have many complex requirements involving 100% availability, being able to properly handle huge amount of transactions and using standard communication protocols to automatically interact with other financial systems.

Klarna AB, the company where this thesis is carried out, is a financial institution created in 2005 under the name Kreditor. It deals with more than 2.000.000 transacations per year, each of them having an average value around 1.000 SEK and it directly interacts with banks, e-commerce shops and persons. The transactions are performed in real-time (3 sec latency) and the availability goal is zero planned downtime and less than a minute downtime at hardware failure.

1.2 Goals

As of today, most of the financial systems provides an XML-RPC API used for computer to computer communication and a GUI API used for human to computer communication. This represents an issue given the fact that in most cases the same services are provided by both APIs. This means that almost double amount of code is needed and when refactoring or applying some change, double amount of work is required. Furthermore, each one of the APIs themselves have some disadvantages.

An investigation of the REST architectural style will be carried out to find out if it would be a proper solution for this issue. The investigation has the following goals:

- Understand the principles of REST.
- Get an understanding of what kind of data is handled by the financial systems.
- Devise a method to structure the access of the data using a REST API.
- Implement a RESTful API following that method.
- Show how the API can be used by persons and computers.

1.3 Outline of the Thesis

Chapter 2 starts explaining where the term REST fits today with a brief explanation of the background on Web Services. The term REST and its characteristics are introduced.

Chapter 3 explains the environment and structure of the Klarna system. Moreover, a description of Erlang, the programming language that will be used to develop the thesis, is provided and the thesis' goals are presented again in a more technical way.

Chapter 4 describes the design process of the RESTful API, how the REST principles are applied and how the API will be structured. In addition, a set of sequence diagrams in high level are presented to show how the REST API will behave and how the different modules that are part of it will interact.

Chapter 5 explains the development process of the initial prototype and the technologies used to build it.

Chapter 6 describes how the definitive REST API is implemented by integrating the initial prototype in Klarna's system replacing some technologies by the ones used by the Klarna's system. Furthermore, a test process is carried out to check the correct behaviour of the API by accessing it through a set of computer-to-computer requests and through a simple GUI.

Chapter 7 summarize the work done on the thesis, shows the conclusions and the future work.

Chapter 2

REST

This chapter starts with the origins of REST, explains the alternatives to use REST and the problems these alternatives present. It also introduces the fundamental ideas of REST, the main characteristics and why to consider it.

2.1 Background on WEB Services

In November 1990, Sir Tim Berners-Lee, actual Director of the World Wide Web Consortium, helped by Robert Cailliau, published a formal proposal presenting what we now know as the World Wide Web. Since then, the World-Wide Web global information initiative has been using HTTP.

The Hypertext Transfer Protocol (HTTP), according to the IETF specification [2], is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, object-oriented protocol which can be used for many tasks through the extension of its request methods. A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred.

Berners-Lee was one of the authors of the HTTP/1.0 [3] and HTTP/1.1 [2][4] standard specification. Another author was Roy Thomas Fielding who, at the same time the RFC 2068[4] was being developed, honed his model down to a set of principles, properties, and constraints[5]. Here was born the term REpresentational State Transfer (REST), introduced in 2000 in his doctoral dissertation [6].

Considering that the World Wide Web uses HTTP and the relation between REST and HTTP, REST can be considered as the set of principles underlying the Web. Today's ”web service” architectures have forgotten or ignore the most important feature that makes the Web successful: its simplicity. They propose a heavyweight architecture for distributed object access, similar to COM or CORBA [7]. REST tries to be faithful to the Web principles.

The W3C [1], through the WSA (Web Services Architecture) document [9] defines a Web service as a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

The WSA does not specify how Web services are implemented, and imposes no restriction on how Web services might be combined. It just describes the minimal characteristics that are common to all Web services, and a number of characteristics that are needed by many of them. The WS-I (Web Services Interoperability) organization instead mandates both SOAP and WSDL in their definition of a web service.

According to the W3C there are two classes of web services, REST-compliant Web services, and arbitrary Web services (Big Web Services according to [7]). Both classes use URIs to identify resources and use Web protocols (such as HTTP and SOAP 1.2) and XML data formats for messaging. SOAP 1.2 can be used in a consitent manner with REST.

To extend Web Services capabilities there are a number of standards, protocols, specifications mostly built on top of HTTP. These are known as the WS-* stack and include WS-Notification, WS-Security, WSDL and SOAP [7]. This WS-* stack is the one that has evolved so fast as it has complicated the simplicity of the Web. The WS stack delivers interoperability for both the Remote Procedure Call (RPC) and messaging integration styles. The most representative example of the RPC-style is the XML-RPC protocol (see 3.3), these days a legacy protocol.

SOAP once stood for Simple Object Access Protocol, now it is just an acronym that according to [10] is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. It uses XML (Extensible Markup Language) technologies to define an extensible messaging framework providing a message construct that can be exchanged over a variety of underlying protocols. SOAP 1.1, which is considered as the XML-RPC's larger sibling, exchanges messages over HTTP 1.1.

The SOAP messages are used as an envelope where the application encloses whatever information needs to be sent [11]. The envelope contains a header and a body. XML Schema is used to describe the structure of the SOAP message, so that SOAP engines at the two endpoints can marshall and unmarshall the message content and route it to the appropriate implementation.

[1] The World Wide Web Consortium (W3C) is the main international standards organization for the World Wide Web (WWW or W3). Founded and headed by Sir Tim Berners-Lee, it is constituted of member organizations which maintain full-time staff for the purpose of working together in the development of standards for the WWW. It also engages in education and outreach, develops software and serves as an open forum for discussion about the Web [8].

The WSDL (Web Services Description Language) Version 2.0 [12] provides a model and an XML format for describing Web services. WSDL 2.0 enables one to separate the description of the abstract functionality offered by a service from concrete details of a service description such as ”how” and ”where” that functionality is offered. This specification defines a language for describing the abstract functionality of a service as well as a framework for describing the concrete details of a service description. It also defines the conformance criteria for documents in this language.

2.2 REST

As a consequence of the increasing complexity of the Big Web services, the RESTful ”style” has been brought as an alternative solution to implement remote procedure calls across the Web. Furthermore, as cited on [7]:

”The web is based on resources, but Big Web Services don’t expose resources. The Web is based on URIs and links, but a typical Big Web Service exposes one URI and zero links. The Web is based on HTTP, and Big Web Services hardly use HTTP’s features at all. This isn’t academic hair-splitting, because it means Big Web Services don’t get the benefits of resource-oriented web services. They’re not addressable, cacheable, or well connected, and they don’t respect any uniform interface. (Many of them are stateless, though.) They’re opaque, and understanding one doesn’t help you understand the next one. In practice, they also tend to have interoperability problems when serving a variety of clients.”

Representational state transfer (REST) is an architectural style for distributed hypermedia systems such as the World Wide Web. To implement this RESTful architecture there is a set of guidelines called ROA, Resource-Oriented Architecture which are the rules to be followed for designing RESTful web services [6].

The REST architectural style is based on the following principles [7][13][14]: Resources, Addressability, Statelessness, Connectedness and Uniform Interface and Cacheability.

2.2.1 Resources

Everything that a service provides is a resource (customers, cars, pictures, invoices, e-stores, etc.). A resource may be a physical object or an abstract concept. Every Resource will have at least one URI (a unique id). If a piece of information does not have a URI, it is not a resource and it is not on the Web.

Two resources cannot have the same URI (see Figure 2.1) but two different URIs can point to the same data at the same moment (see Figure 2.2). For example, we can have an URI to identify the last_version and another for the particular version_1.3. When the last version is the specific version 1.3, both URIs are pointing to the same data. Some months later, when the new version is released, the last_version URI will not be the same as the version_1.3 URI anymore.

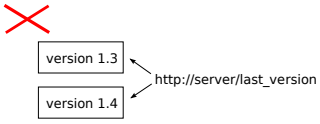


Figure 2.1: Two resources, the same URI.

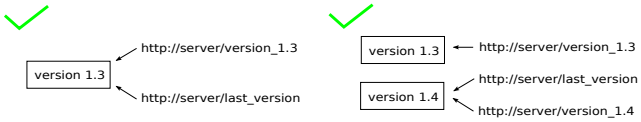


Figure 2.2: Two different URIs refer to the same data.

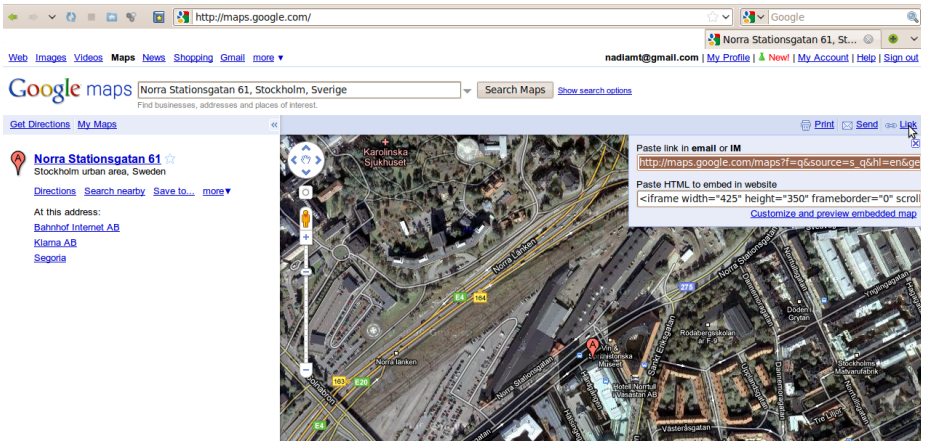
The URI should have a structure. With that it will be predictable for the client and it will be posible for him to create its own entry into the service. That is not a RESTful rule, it is a good web design rule.

A resource can be represented in any format, any media type, for example: XML, XHTML, JSON or CSV representation. Each representation will have its own address but avoiding the format for example: path/page3.xml, path/page3.html and so on. The Content-Type header tells the client the representation needed to display the entity-body. On the human web, a web browser tries to display it inline, and if not possible run an external program. On the programmable web, a web service client uses this to decide which parser to apply to the entity-body.

2.2.2 Addressability

Every resource should be addressable. A resource is addressable by means of URIs so we will have as many URIs as resources we have. With this addressability we can bookmark an specific page, you can mail the URI to someone else, it allows cacheability (the first time someone requests a URI, the cache will save a copy. The next time someone request that same URI, the cache will serve the saved copy).

Consider the web application Google Maps. We type ”Norra Stationsgatan 60, Stockholm” and we obtain the view of the address we have just typed in. The URI of the site ”http://maps.google.se/” does not change. That does not mean that Google maps is not addressable, it means that the web application that we are using is not, but instead, the web service is. If we want to send a friend this map, we need to ask the application for the specific URI and it will show us an URI with all the information that our friend needs to see the same image we are looking at. Without addressability we cannot send the map we are looking at to a friend, we need to do a screenshot or explain to him the steps he has to follow.



2.2.3 Statelessness

Every HTTP request should happen in complete isolation. It means that when the client makes an HTTP request all the information required to process that request must be present. The service should never rely on information from previous requests. If some data from the previous request is needed, the client have to send it again since the server does not keep any information about the client.

This makes the system more reliable, simpler and horizontally scalable[2]. A client does a request to a server A; the client does another request but just in that moment the server A fails. Another server will serve the request since all the information that it needs to serve it is given when doing the request. It means that a web server is replaceable easily by another, easily fail-over makes the system scalable.

In a system with a load balancer (LB), the LB just has to take into account which server will serve better the request not if that server has served the requests from that user before or not. This kind of LB becomes much simpler to implement.

2.2.4 Connectedness

RESTful services should guide clients from one state to another by sending links in the representation. Representations are hypermedia, documents that contain not just data, but links to other resources. Human web is well connected but programmable web was not until now. All the resources should be connected and linked. It allows the client to discover the interface by traversing hyperlinks between each of its resource representations as can be seen on Figure 2.3 which shows an example of connectedness where each of the rectangles represents a resource.

[2]Since the server does not keep any data from the client, any server can serve any request, so more servers can be easily added

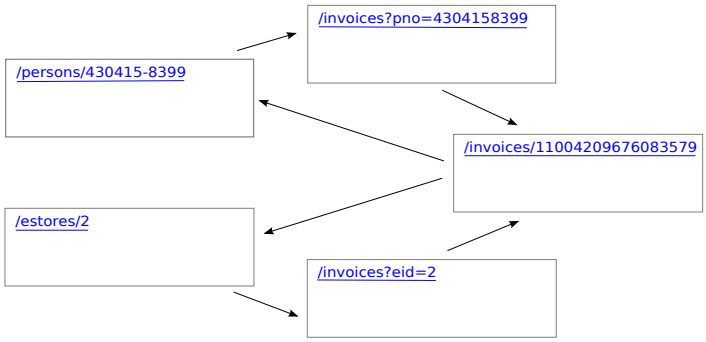


Figure 2.3: RESTful service well connected

2.2.5 Uniform interface

The client interacts via the defined HTTP methods: GET, POST, PUT, DELETE. These methods define the things that can be done to a resource.

HTTP GET

The GET method means retrieve whatever information (in the form of an entity) is identified by the Request-URI [2]. In the case of RESTful web services, this information identified by the URI is a representation of a resource.

GET is a safe and idempotent method. Safe means that it does not change the state of the server, it only retrieves information so it should not have side effects. Idempotent means that it can be applied multiple identical requests having the same effect as a single request. Since the HTTP protocol is a stateless protocol, being prescribed as safe should also be idempotent. Being idempotent allows to safely repeat GET on resources in failure cases.

If the request succeeded, and the resulting resource is returned in the message body the appropriate response status is 200 (OK). If the resource does not exist the response status is 404 (Not Found).

If the request message includes a Range header field, the semantics of the GET method change to a ”partial GET”. It reduces unnecesasry network usage by allowing partially-retrieved entities to be completed without transferring data already held by the client.

HTTP POST

The POST method is used to request that a new resource need to be created with the data enclosed in the request as a new subordinate of the URI given.

If the action performed does not result in a resource that can be identified by an URI, the appropriate response status is 200 (OK). If a resource has been created, the response should be 201 (Created) and contain an entity which describes the status of the request and refers to the new resource, and a Location header.

HTTP PUT

The PUT method requests that the enclosed entity be stored under the supplied Request-URI [2].

If the URI refers to an already existing resource, the enclosed entity should be considered as a modified version of the one on the server. If it does not point to an existing resource, and that URI is capable of being defined as a new resource by the requesting user agent, the origin server can create the resource with that URI. If a new resource is created, the response status is 201 (Created). If an existing resource is modified, a 200 (OK) response codes should be sent to indicate successful completion of the request. If the resource could not be created or modified, an appropriate error response should be given that reflects the nature of the problem.

The main difference between POST and PUT is reflected in the different meaning of the request. The URI in a PUT request identifies the entity enclosed with the request (the client indicates in the URI the id of the resource). In contrast, the URI in a POST request identifies the resource that will handle the enclosed entity. Moreover, the PUT method is idempotent, as well as the GET method, since making more than one PUT request to a given URI should have the same effect as making only one, but the POST method is not.

HTTP DELETE

The DELETE method requests that the server deletes the resource identified by the request. It is also an idempotent method as the PUT and GET methods.

A successful response should be 200 (OK) if the action has been performed, if the action could not be performed, an appropriate error response should be given that reflects the nature of the problem.

2.2.6 Cacheable

The resources should be cachable whenever possible with an expiration date and time. Cacheability offers better user experience since the load on the server is decreased and with that, a better response and loading time is achieved.

There are two types of caching mechanisms in HTTP: Expiration and Validation. According to RFC2616 [2], the goal of caching in HTTP/1.1 is to eliminate the need to send requests in many cases, which reduces the number of network round-trips required for many operations and, to eliminate the need to send full responses in many other cases, which reduces network bandwidth requirements. The former is done through the ”Expiration” mechanism; the latter through the ”Validation” mechanism.

The responses to the POST method are not cacheable, unless the response includes appropriate Cache-Control or Expires header fields (Expiration). The conditional GET method is intended to allow cached entities to be refreshed without requiring multiple requests or transferring data already held. The GET method becomes conditional GET if the request messages includes an Etag / If-None-Match, Last-Modified / If-Modified-Since.

Expiration

Expiration is a way for the server to say how long a requested resource stays fresh for. It is excellent for resources that change very rarely (i.e. images) or at known time.

- **Expires header**

  The HTTP Expires header just says the date and time when the resource will become stale. It requires the server's and the client's clock to be synchronized.

- **Cache-Control Header**

  HTTP 1.1 replaced the Expires header by this one which solves the syncronization problem from the Expires header and offers more flexibility. The cache-control header has a number of clauses that can be used to control the way the client caches the resource [15].

Validation

Validation allows a client to ask the server whether a cached version of a resource is still fresh.

- **Last-Modified / If-Modified-Since** This header makes conditional HTTP GET possible. When doing a GET request, the Last-Modified header value tells the client the last time the representation changed. The client can keep track of this date and send it in the If-Modified-Since header of a future request. When the server receives a GET request that has the If-Modified-Since header defined, it checks that the resource has not changed from the time specified in the mentioned header. If it has not, the server will send a response code of 304 (”Not modified”) with no entity-body. If it has, then the server will realize that given the fact that the If-Modified-Since date is previous to the one that the server internally has. Therefore, the server will send the new entity-body and will fill the Last-Modified header with the time in which the resource was lastly modified.

  Since Last-Modified has a one-second accuracy it is not accurate enough. Occasionally a conditional HTTP GET gives the wrong result only taking into account the If-Modified-Since header. For this reason it is also advisable to use the ETag and If-None-Match headers.

- **ETag / If-None-Match** The value of ETag is used to determine if a representation has changed. If the respresentation changes, the ETag should also change. The ETag header ought to be sent in response to GET requests. The value of ETag can be used as the value of a If-None-Match header in a future conditional GET request. If the representation has not changed, the condition fails since the ETag is the same. The server sends a response code of 304 (”Not Modified”) with no entity-body, so the server can save time and bandwidth not sending it. If the ETag has changed, the condition succeeds and the server sends the new entity-body.

The ETag header is a second line of defense, the conditional GET main drivers are the Last-Modified and the If-Modified-Since response headers. If the representation changes twice in one second, the Last-Modified header will not change but it will do it the ETag header.

2.3 Why REST

Given the principles explained in the last section, a set of benefits can be obtained following the REST architectural style.

Firstly, each physical object or abstract concept at which we can refer is identified as a unique resource. This implies that each of them can be quickly accesed by just one request once they are identified. This results in a better navigability and experience for the user and less load for the server since each resource can be accessed with just one request.

Secondly, due to the statelessness principle, REST makes the system really scalable since the servers do not keep any information from any of the clients. This means that every single machine that is part of the system is able to receive any kind of request from any of the clients and generate a proper response. Other effects of this principle are that complex load balancers are not needed anymore and that the clients that are using the system when a server goes down do not notice it, since the system becomes fault-tolerant and recoverable.

Furthermore, with a no RESTful web service it can be really difficult to discover the interface because the resources may not be addressable, they may not be connected between each other and the HTTP methods used in each request may not follow a standard pattern. Following the REST style, the client is able to guess how the interface is designed and go directly to the information needed using a specific URI or it may navigate through the representations using the links they contain resulting in a really good user experience.

Lastly, cacheability, which can be easily achieved by using the HTTP headers properly, reduces the load on the servers and improves the response time which is translated again in a better user experience.

All these benefits make REST an interesting and attractive architectural style that is worth to try out since it can really improve a web service in many ways.

Chapter 3

Klarna environment

This chapter starts with a brief explanation of the Klarna workflow and the most important technologies used by the Klarna system. The programming language Erlang and the XML-RPC protocol are among those technologies. Next, the structure of the current system at Klarna is described. Lastly, the objectives of the thesis are explained in a more technical way.

3.1 Workflow

Klarna is a payment supplier for e-commerce currently available for Swedish, Finnish, Norwegian, Danish and German consumers. Klarna's payment solutions increases sales for the e-stores by offering safe and simple payment to the end-customers through invoices.

The basic workflow is the following: an end-customer does a purchase on an e-store and chooses Klarna as the payment option. The Klarna system receives the request, stores the data of the end-customer and the purchase and pays the purchase to the e-store at the same moment the purchasing process ends. When the end-customer receives the goods purchased, Klarna sends an invoice to the end-customer address and that end-customer is expected to execute the payment.

3.2 Erlang

The Klarna system is developed using the Erlang programming language. In this section is explained the history and the main characteristics of this language.

In the 80s Joe Armstrong, Robert Virding, Claes Wikström and Mike Williams, at the Ericsson Computer Science Laboratories, were assigned to the task of investigating which programming language would be the most suitable for programming the next generation of telecommunication products. After some years of experiments

they decided to develop their own language, a declarative language designed for programming concurrent and distributed systems [16].

Erlang is a functional language that was influenced by functional languages such as ML, concurrent languages such as CHILL, as well as the Prolog logic programming language among others [17]. In 1995, after years of development and prototyping, an enough mature version was released to be used in major projects. Due to this, in 1996 was released the Open Telecom Platform (OTP), a framework that provides a group of libraries for building large-scale, fault-tolerant, distributed Erlang applications.

Originally Erlang was a propietary language within Ericsson, but it was released as open source in 1998 to ensure its independence from a single vendor and to increase the awareness of the language. Nowadays, many applications are written in Erlang: e.g. CouchDB, a distributed, fault-tolerant and schema-free document-oriented database accessible via a RESTful HTTP/JSON API; ejabberd, an XMPP (Extensible Messaging and Presence Protocol) application server; and Mochiweb, a library that provides support for building lightweight HTTP servers.

3.2.1 Erlang's Characteristics

Erlang's main strength is support for concurrency. Processes cannot interfere with each other inadvertently since each Erlang process executes in its own memory space and owns its own heap and stack and they communicate with each other via messagepassing. The process creation time is of the order of microseconds and independent of the number of concurrently existing processes.

The single assignment property makes much easier to understand and predict the behaviour of a program since variables can only be bound once.

The pattern matching property in Erlang is used to assign values to variables, to control the execution flow of programs or to extract values from compound data types. According to [19], pattern matching is the act of checking for the presence of the constituents of a given pattern rigidly specified. Often it is possible to give alternative patterns that are tried one by one, which yields a powerful conditional programming construct.

Erlang also allows code to be changed in a running system, it is called hot code swapping. This is of great use in 'non-stop' systems, telephone exchanges, air traffic control systems, etc., where the systems cannot be halted to make changes in the software.

Due to a set of simple but powerful error-handling mechanisms and exception monitoring constructs, used to built library modules with robustness designed into their core, Erlang makes the system robust. Letting these libraries handle the errors the programs become shorter, easier to understand, and will contain fewer bugs.

The last property, but not the less important, is the distribution property, which Erlang has incorporated into its syntax and semantics. It makes easier to built distributed systems and applications written for a single processor can be ported to run on a network of processors without difficulty. The default distribution mode is

based on TCP/IP, which allows a node on a heterogeneous network to connect to any other node running on any operating system as long as these nodes are connected through a TCP/IP network. Operations such as clustering, load balancing, the addition of hardware and nodes, communication, and reliability come with very little overhead and code.

3.3 XML-RPC

XML-RPC's[20] heritage comes from RPC and the World Wide Web. It reuses infrastructure that was originally created for communications between humans to support communications between programs on computers.

Although the Web was initially a tool for human-to-human communications, it evolved into a sophisticated interface for human-to-computer interaction, and is also moving into increasingly complex computer-to-computer communications.

HTML was really successful but only really useful for transactions presenting information to people. For that reason, the World Wide Web Consortium (W3C), hosted the development of eXtensible Markup Language (XML), a markup language that provides more flexibility for communications between programs than HTML.

XML provides a vocabulary for describing Remote Procedure Calls (RPC), which are then transmitted between computers using the transport protocol of the Web, the HTTP protocol. Because XML-RPC is layered on top of HTTP, it inherits its inefficiencies, which affect on its use in large-scale, high-speed applications where systems must scale to millions of transactions at a time, keeping response time to a minimum. But on systems which response time is not critical, XML-RPC can simplify development tremendously and make it easier for different types of computers to communicate. It allows a developer to build services without having to know in advance the characteristics of the client or server on the other end of the connection.

The use of HTTP means that XML-RPC requests must be both synchronous and stateless. An XML-RPC request is always followed by the XML-RPC response because both the response and the request must occur on the same HTTP connection. It is possible to implement an asynchronous system, but the overhead of creating it is probably prohibitive in most cases. In general, synchronous requests are capable of fulfilling many processes' requirements.

Being stateless means that no context is preserved from one request to the next. XML-RPC itself has no way to treat them as anything other than two isolated, unconnected incidents. This avoids the sometimes massive overhead involved in maintaining state across multiple systems. XML-RPC does not provide support for preservation of state, but a stateful system on top of XML-RPC can be implemented.

Having explained the main points of this technology, the disadvantages that it presents are provided in the following list:

- An XML-RPC request includes the action to execute and the parameters of the action in the body of the HTTP request when HTTP already has facilities for making requests and returning responses.
- An XML-RPC API needs to define its own error codes when would be much easier to use the HTTP error states by using the HTTP protocol properly.
- With this type of API there is a lack of client-side caching and authentication, functionalities that HTTP provides, since it is not possible to implement them in systems when using XML-RPC.

3.4 The Klarna system

The Klarna system is written in Erlang. It was chosen because it provides many features that a financial computer system should have such as:

- fault-tolerance which is needed to build a system that automatically recovers itself after a crash without causing the system to stop. This is essential in financial systems since they should be always available.
- hot code swapping which allows the upgrade of the code that the system is currently using without having to stop it.
- easy to use concurrency that allows multiple processes to run at the same time in multicore processors which leads to better performance.
- easy to handle huge amoung of transactions thanks to the distribution property which is essential to build an scalable system.

As of today, the Klarna system provides an XML-RPC API used for computer to computer communication and a GUI API used for human to computer communication. Regarding the structure of the system, it has three big layers that are shown on figure 3.1: a data storage layer that uses Mnesia (see 6.1) and an abstraction layer on top of it called KDB; a business layer that contains most part of the business logic of the system; a top layer that contains some business logic and the mentioned XML-RPC and GUI APIs.

As mentioned in 1.2, many of the services are provided by both APIs and that results in almost double amount of code and double amount of work when refactoring some of those services. Furthermore, each one of the APIs themselves have some disadvantages: the XML-RPC disadvantages were explained in 3.3 and the GUI API ones are described next:

- It just provides HTML as returning data type. It should provide other data types in order to make possible to communicate with other clients that only accept JSON, XML, etc. as returning data types.
- The URIs used are not intuitive for the users and do not provide discoverability of the system.

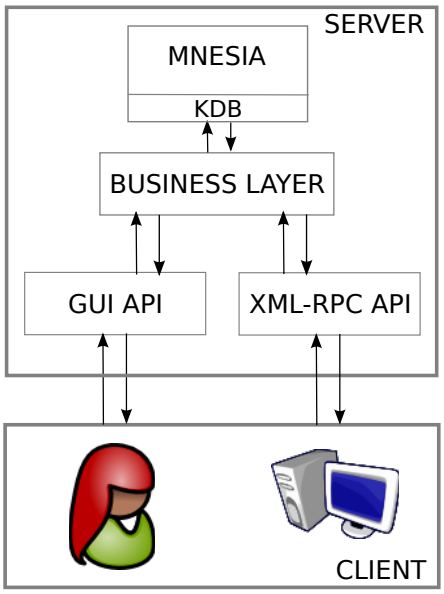


Figure 3.1: The Klarna System

- Cacheability functionality of the HTTP protocol is not used due to the current design of the GUI API and that limits the scalability of the web service.
- The code handling the production of the HTML is mixed with the code dealing with the business logic.

3.5 Goals

Taking into account all the issues and disadvantages mentioned, an investigation of the REST architectural style will be carried out to find out if it would be a proper solution. The plan is to replace the XML-RPC API and the GUI API with a unique and generic REST API that handles all the services. The following goals have been defined for the investigation:

- Understand the principles of REST, what properties, benefits and disadvantages it has and how it can be applied and used to solve the described issues.
- Get an understanding of what kind of data is handled by the Klarna system, which are the most important resources and what information they contain.
- Devise a method to structure the access of the data using a REST API and design how it will be built, which modules will make up it, what each module will do and how they will interact between each other.
- Implement the RESTful API following the results from the design phase, including first an initial prototype and then the integration of this prototype in the Klarna system.

- Test the API by accessing it through a simple proof of concept GUI and through the command line to check that both a human and a computer can make use of it.

Chapter 4

Designing the REST API

This chapter describes the design process of the REST API, that is, describes the structure that the REST API will have, how it will be built, which modules will make up it, what each module will do and how they will interact between each other. Moreover, is explained how the REST principles are going to be properly applied so that the API can be considered RESTful. Furthermore, an explanation of the workflow for the main use cases is provided.

4.1 Resources

The first activity of the design process is to identify the resources that are going to be accessible through the API. This has been achieved by getting an outline of how the Klarna system behaves and also talking and discussing about it with software developers at Klarna who work daily with the system. It has been decided, for the sake of simplicity, to just select the three main resources stored in the Klarna system: *invoice*, *e-store* and *person*. This decision has been made because it is not mandatory to provide all the resources to show if a REST API fits or not in the Klarna environment. Supporting the most important ones is enough and the results will be more clear.

Next, is provided an explanation of what each resource represents. Is also described what data they contain and what parameters are needed to create them as of today in the Klarna system:

- **Person**
  The resource *person* identifies the end-user, the person who buys something in an e-store and pays it using the Klarna payment service. To create an instance of this resource two parameters are needed: the personnummmer[1] and the id of an e-store, which is an old requirement of the Klarna system.

[1]The personnummer is the Swedish national identification number. It is personal and it covers the total resident population of Sweden. The number is used by authorities, by health care, schools and universities, banks and insurance companies.

A person is identified in the system by his or her personal number and this identity is also used to request the official information of the person to a credit company to check the person's reliability. Among this official information there are basic fields like first name, last name and address that are then stored within the person resource.

The following design decisions have been made regarding the REST API that will be built: the person resource is made up by personal number, e-store id, first name, last name and address; users are allowed to edit all fields but the personal number; clients are able to search through the set of persons providing criterias containing personal number, first name, last name and country.

- **e-store**
  The resource *e-store* represents an e-commerce shop that is a Klarna customer and therefore offers Klarna payment solutions as a way of payments to its end-customers. The parameteres needed to create a resource of this type are name of e-store, organisation number, address, post number, city, country, telephone, webpage and e-mail. When an e-store is created it is assigned to a unique e-store identifier (eid) which is automatically generated. In the REST API: the e-store resource is made up by the fields entered by the person using the API when creating it plus the e-store identifier; clients can edit all the fields of the e-store but the eid; clients can search through all the e-stores by introducing a specific criteria involving eid, name and country.

- **Invoice**
  The resource *invoice* represents a purchase done by an end-customer through an e-commerce shop using Klarna payment solutions. It is created by introducing on the system the id of the e-store where the purchase has been done, the personal number of the person who has done the purchase, a list of the goods purchased and the country. When the invoice is created, an invoice number is automatically generated as an identifier which refers uniquely to the invoice. In the REST API: the invoice resource is made up by the fields introduced by the client plus the invoice identifier; clients can edit all the fields but the invoice number; clients can search through the set of invoices by applying different criterias involving invoice number, the estore identifier, the personal number and country.

4.2 Addressability

One of the main principles of REST is that each resource should be addressable by a unique URI as said in Section 2.2.2. To achieve this requirement in the REST API a unique and predictable URI is going to be assigned to each resource:

- The first part of the URI shows which type of resource the client is addressing to. Therefore, the first part is going to be either */invoices*, */estores* or */persons*. Those URIs represent the set of invoices, e-stores and persons respectively.

- If the URI has a second part, it represents the identifier of the resource to which an action is applied. Therefore, an action applied to the URI */invoices/123* will be addressed to the invoice with invoice number 123.

With this URI structure is achieved a simple, understandable and easy to use way of directly addressing a specific resource.

4.3 Statelessness

As explained in Section 2.2.3, every HTTP request should happen in complete isolation and no information about the client should be stored in server from one request to another. Currently, when a user logs in to the Klarna GUI, a session state is stored in the server that received the request. This state contains information such as language, last invoice searched, last person searched, etc. This causes the Klarna system not to be easily scalable.

In the REST API that is being designed, no information per session is going to be stored in the server side. It is a simple task since there is no real need to store the mentioned information:

- Authentication can be achieved by standard HTTP authentication, but it is not going to be implemented in this version of the API since it would not prove anything new.
- It is avoidable to store in the session state which language the client wants the information presented on. This could be achieved by adding a parameter language in the URI of the request like */invoices/123?lang=en* or by storing this preference in the data storage layer. Again, this is not going to be implemented in the REST API during the thesis since it is about implementation details.

To sum up, the REST API is going to be completely stateless following the REST principles which will result in a more scalable system.

4.4 Connectedness

A well connected application allows the user to discover by his own the interface. The resources must be connected to other resources and its representation should offer link to those other resources. This feature is one of the most simple to provide in the REST API being designed. Connectedness is going to be achieved by:

- Each list of resources such as *invoice*, *e-store* and *person* contains link to the resources listed.
- Each invoice resource contains a link to the person that did the purchase and a link to the e-store where the purchase was done.

- Each person resource contains a set of links to his or her last invoices.
- Each e-store resource contains a set of links to its last invoices.

4.5 Uniform Interface

Most of the APIs provided by web-based systems today just allow GET and POST methods to execute different actions over their resources (retrieve, create, update or delete a resource). On the other hand, a RESTful API should use the defined GET, POST, PUT and DELETE HTTP methods properly as explained in section 2.2.5. Thus, in the REST API being defined, the four methods are used to execute the function they were designed for.

Next, the description about the effects an HTTP request will have taking into consideration the four methods and the URIs described in section 4.2 is presented:

- **/resource**
  - GET: shows a limited list of the resources on the system. If the URI contains a query, the list of resources returned will meet the criteria specified in the URI. If the response is in XHTML content type, it contains links to create a new resource and search for resources.
  - POST: adds a new resource of the specific type to the system taking the information embedded in the body of the request.
  - PUT: not appliable
  - DELETE: not appliable
- **/resource/X (X = resource identifier)**
  - GET: return the information of the resource with identity X in the content type requested by the user. If the response is in XTHML content type, it contains a link to update the given resource.
  - POST: not appliable
  - PUT: updates the information of the resource with identity X taking the information embedded in the body of the request.
  - DELETE: not appliable since it should not be possible to remove any kind of resource from the Klarna system. Klarna should keep all the persons, e-stores and invoices stored in the system for legal reasons.

When the user is a human browsing the GUI, he or she needs an easy way to create, update and search resources. Therefore, three extra URIs are provided in this case which are presented next:

- **/resource/search**

  - GET: returns a form to search for resources. The fields the form has are the ones matching the resource field that the user wants to search. When submitting the form, an HTTP GET request will be sent to an URI that will include the search criteria. An example could be */estores?name=Klarna+AB*.
  - POST: not appliable
  - PUT: not appliable
  - DELETE: not appliable

- **/resource/new**
  - GET: returns a form to create a resource. The form contain a specific list of fields depending on which resource the user is creating. When submitting the form, a POST request will be sent to */resource* with the value of the fields in the body of the request.
  - POST: not appliable
  - PUT: not appliable
  - DELETE: not appliable

- **/resource/X/update**
  - GET: returns a form with a list of fields already filled with the current data of a given resource that will help the human user to easyly update the information of that resource. The user will edit some fields and then submit the form. When that happens, an HTTP GET request will be sent to */resource/X*.
  - POST: not appliable
  - PUT: not appliable
  - DELETE: not appliable

4.6 Cacheability

Cacheability is one of the most important principles to build an scalable system. The purpose is to cache as many resources as possible to lower the load of the servers. It has been decided that lists of resources such as */invoices* or */persons* are not cacheable since they tend to quickly change, but given the case that a set of resources is not that dynamic it would make sense to cache it.

On the other hand, resources such as */invoices/123* or */persons/841017-1234* are cacheable and a big profit is going to be obtained from this decision. To cache this resources, the Validation mechanism is going to be used. Each time a client retrieves the invoice with id 123 using the GET method, the server will check if the resource has been modified since the version that the client has available. The server will be able to do this since it will store the last modification date per each single resource in the system. If the resource has not been modified, the system

will just return the correct HTTP 'Not Modified' 304 code with an empty body, which means that the amount of work to do in the server side is minimal. On the other hand, if the resource has been modified the server will have to generate the body of the response and override the last modification date with the current date.

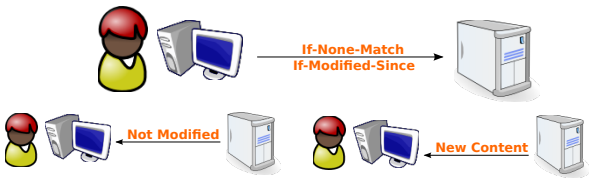


Figure 4.1: Cache process

4.7 The REST API structure

With the REST API in place the Klarna system still has three layers. The data storage layer and the business layer remain unmodified but the XML-RCP API and the GUI API are replaced with the REST API. Both human and computer clients will interact with the same API as shown in Figure 4.2.

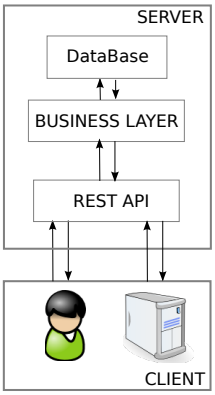


Figure 4.2: Klarna System with the REST API

Next, the design of the structure of the REST API is presented in Figure 4.3. The purpose of this structure is to make it as generic, abstract and divided into small modules as possible. Following this path is obtained a REST API that perfectly fits in totally different kind of systems, regardless of the technologies used on the system.

Each of the modules appearing in Figure 4.3 are explained next:

- The *dispatcher* is the module that receives the HTTP requests from the client, analyzes them and forwards them to the appropiate controller module depending on which URI the request was sent to. Thus, if the request is done

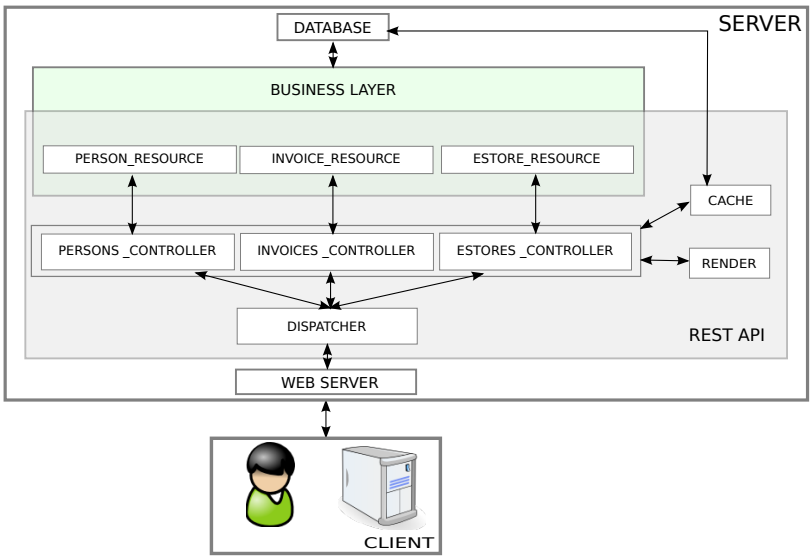


Figure 4.3: Design of the structure of the REST API

against */invoices/123* the dispatcher will forward this request to the invoices controller. After that it will wait for the controller to send a result back, it will build an HTTP response with that result and send it to the client.

- The *controllers* are resource specific modules and there is one of them per each type of resource in the system. They receive the request from the dispatcher and do some error detection. It checks if the URI is correct, if the method sent in the request can be applied to the URI, if the body of the request is correct and so on. If the request is not correct, it sends back to the dispatcher the appropiate HTTP error. Otherwise, it controls the rest of the flow that the request will go through and that involves caching, interacting with the business layer, rendering the result in the correct content type and return it back to the dispatcher.

- The *xxx_resources* modules are also resource specific and there is one of them per each kind of resource in the system. They handle the basic logic business flow of each action that can be applied to the resource. They are called from the controllers, then they interact with the rest of the business layer and perform some action to the data storage layer. Finally they return a result back to the controller.

- The *cache* module handles cache issues for all the resources becoming a really generic and useful module. Its purpose is to store the last modification date of each resource and also to decide if a resource has been modified when receiving a GET request from the client. It then interacts with the controllers and the data storage layer.

- The *render* module handles the transformation of data coming from the business layer into data formatted in the content type that the client asked for.

XML, XHTML and JSON formats are going to be supported.

The main benefits of the described structure are:

- It is modularized into small pieces and every piece has its own specific and defined purpose.
- It is generic and abstract since there are modules like *dispatcher* and *cache* that are adapted to many different kind of resources.
- It is really scalable since adding a new resource is as simple as adding a controller and a small business module for each new resource.
- The contoller modules take care of the flow of an HTTP request being capable of handling HTTP errors, different content types, caching and also interacting with the business layer in a really simple way.

4.8 Sequence Diagrams

In this section is shown a set of sequence diagrams that describe the workflow for the main use cases. Given the fact that the flow is really similar between the three type of resources in the system, is just provided for the invoice resource.

4.8.1 Get invoices

Figure 4.4 shows the flow when a client needs to obtain a list of links to navigate to the invoices created in the system. The client sends an HTTP GET request to the URI */invoices*. The request is received by the *dispatcher* which extracts the URI and the request headers and forwards it to the appropiate controller taking into account the URI. In this case the controller chosen is *invoices_controller*. The controller, by analyzing the URI and the get method, decides that it should call the *invoice_resource* and this will just ask to the data storage layer for the list of invoices. The list arrives to the controller which asks the *invoice_render* to render the list in the content type asked by the client. Finally, the response is sent back from the controller to the dispatcher which generates the HTTP response and sends it to the client.

4.8.2 Get invoice

Figure 4.5 shows the flow when a client needs to obtain the data of a single invoice created in the system. The client sends an HTTP GET request to the URI */invoices/1234* where 1234 represents the identifier of the resource to which the action is applied. The request is received by the *dispatcher* which extracts the URI and the request headers and forwards it to the appropiate controller taking into account the URI. In that case the controller chosen is *invoices_controller*. The controller, by analyzing the URI and the method, decides that it should call the *cache*

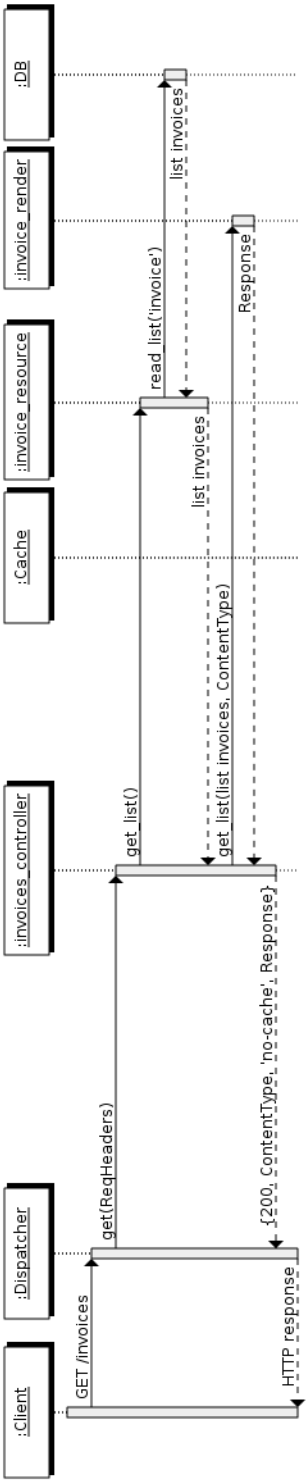


Figure 4.4: Get invoices sequence

module and this checks if the invoice has been modified since the last time the client retrieved it. If it has not been modified, the *cache* module returns a *Not Modified* message. That message arrives to the controller which asks the *invoice_render* to render the message in the content type asked by the client. Finally, the response is sent back from the controller to the dispatcher which generates the HTTP response and sends it to the client. If it has been modified, the controller decides that it should call the *invoice_resource* and this will just ask to the data storage layer for the invoice data. The data arrives to the controller which asks the *invoice_render* to render the invoice data in the content type asked by the client. Finally, the response is sent back from the controller to the dispatcher which generates the HTTP response and sends it to the client.

4.8.3 Create invoice

Figure 4.6 shows the flow when a client needs to add an invoice in the system. The client sends an HTTP POST request to the URI */invoices* including on the body the data of the new invoice. The request is received by the *dispatcher* which extracts the URI , the request headers and the body of the request where the data to create the new invoice is included, and forwards it to the appropiate controller taking into account the URI. In that case the controller chosen is *invoices_controller*. The controller calls the *invoice_resource* and this will generate the invoice and write it to the data storage layer. The generated invoice arrives to the controller which asks the *invoice_render* to render the invoice in the content type asked by the client. The controller will call the *cache* module in order to set the Cache Headers to do the resource cacheable. Finally, the response is sent back from the controller to the dispatcher which generates the HTTP response and sends it to the client.

4.8.4 Update/Modify invoice

On figure 4.7 it is shown the flow when a client needs to update an invoice in the system. The client sends an HTTP PUT request to the URI */invoices/1234* including on the body the data that wants to update, where 1234 represents the identifier of the resource to which the action is applied. The request is received by the *dispatcher* which extracts the URI, the request headers and the body of the request where the new data is included, and forwards it to the appropiate controller taking into account the URI. In this case the controller chosen is *invoices_controller*. The controller calls the *invoice_resource* and this will update the invoice and write it to the database. The updated invoice arrives to the controller which asks the *invoice_render* to render the invoice in the content type asked by the client. Finally, the response is sent back from the controller to the dispatcher which generates the HTTP response and sends it to the client.

4.8.5 Delete invoice

In the specific case of Klarna it is not allowed to delete any resource.

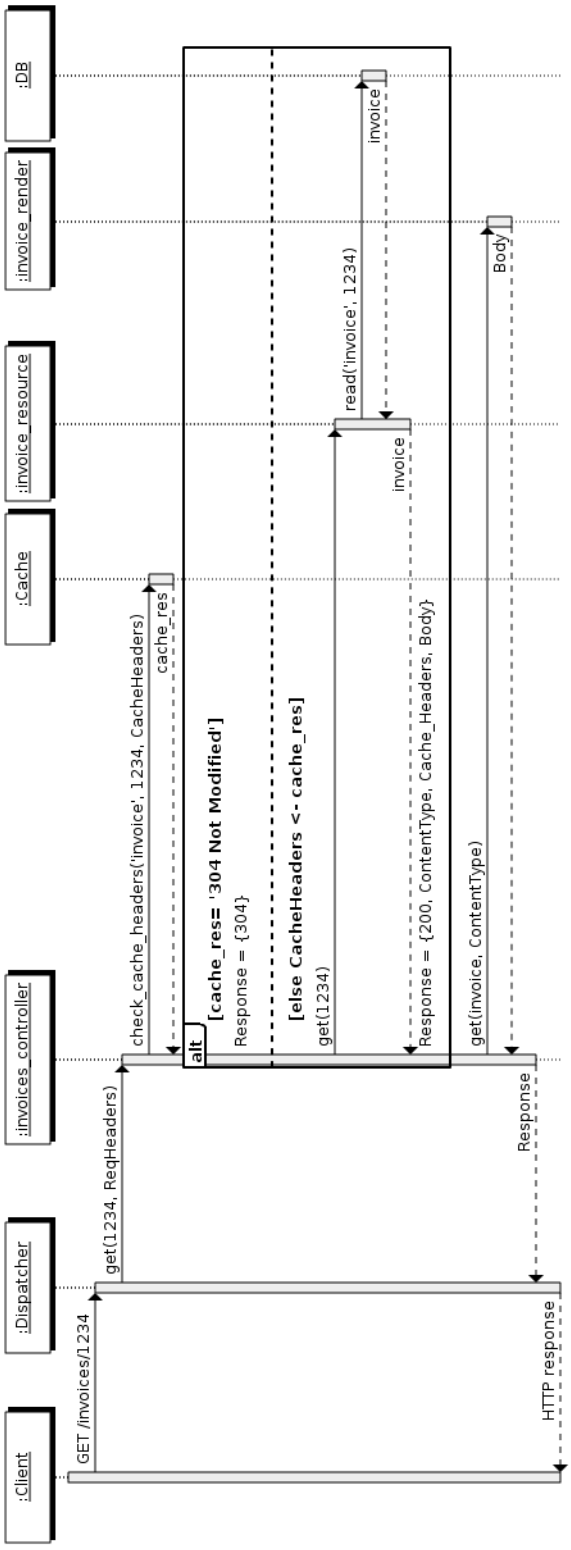


Figure 4.5: Get invoice sequence

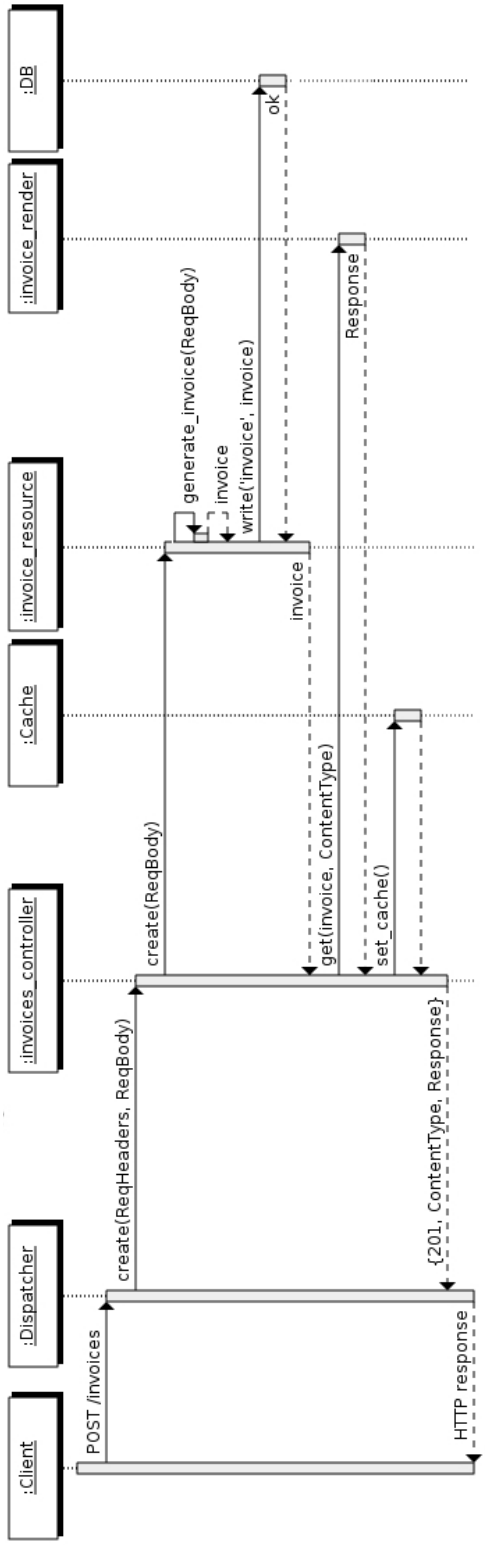


Figure 4.6: Create invoice sequence

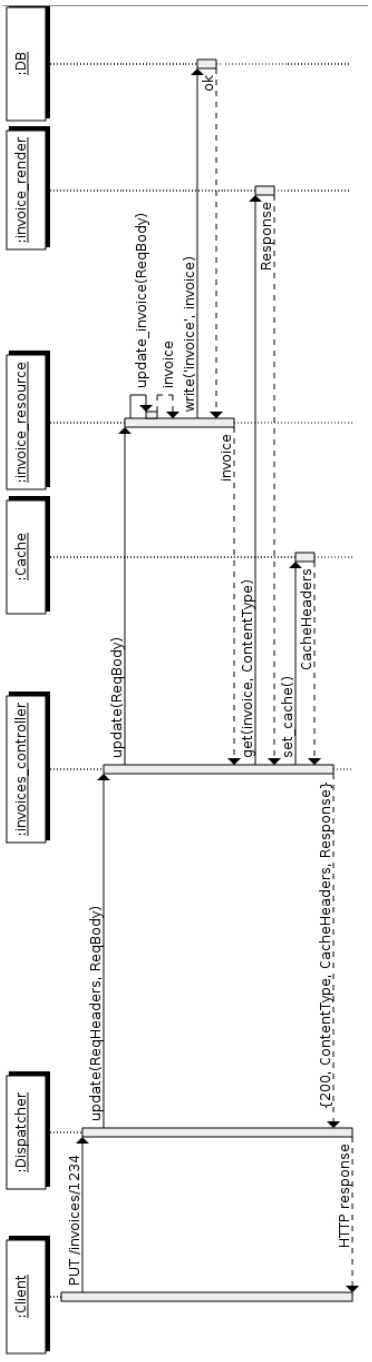


Figure 4.7: Modify invoice sequence

4.8.6 Search invoices

Search invoices is the same as Get invoices but with some restrictions indicated on the body of the request. Figure 4.8 shows the flow when a client needs to obtain a list of links to navigate to the invoices in the system that fits these restrictions. He/she/it does a HTTP GET request to the URI */invoices*. The request is received by the *dispatcher* which extracts the URI, the request headers and the body of the request where it is indicated which restrictions this invoices should fit, and forwards it to the appropiate controller taking into account the URI. In that case the controller chosen is *invoices_controller*. The controller, by analyzing the URI and the method, decides that it should call the *invoice_resource* and this will just ask to the data storage layer for the list of invoices. The list arrives to the controller which asks the *invoice_render* to render the list in the content type asked by the client. Finally, the response is sent back from the controller to the dispatcher which generates the HTTP response and sends it to the client.

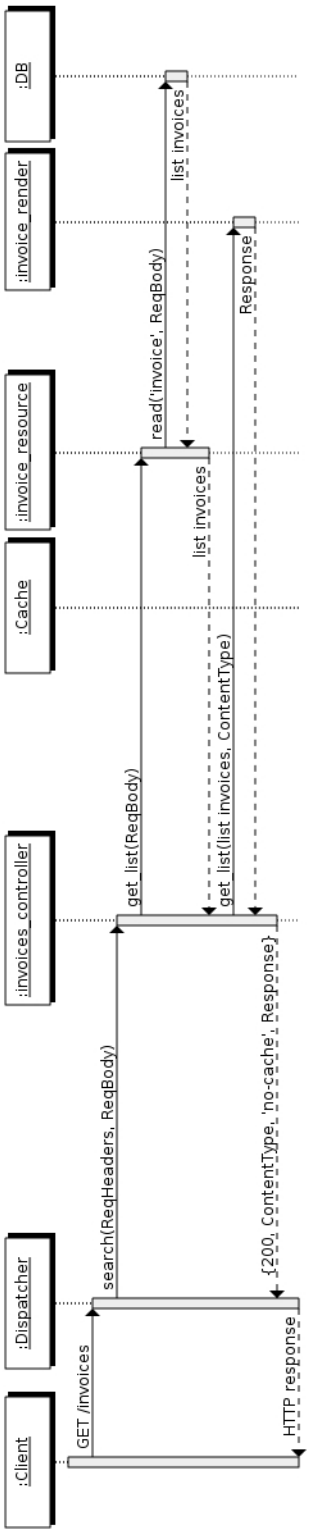


Figure 4.8: Search invoices sequence

Chapter 5

Prototype

This chapter introduces the techonologies used in the initial prototype that has been developed and why they have been chosen. Then, is explained the development process of the prototype and is provided a small set of pieces of code to show the simplicity that implementing a REST API involves.

5.1 Introduction

When dealing with technical concepts as REST, it is much better to learn about them by creating an isolated and small prototype to avoid external problems and just focus on the real concept that is being checked. This is the reason why it has been decided to first implement a prototype instead of directly trying to build an application that tightly interacts with a huge and complex system such as the Klarna system.

Developing this prototype some exploratory work in new technologies has been done and it has been used to build the mentioned prototype. The technologies chosen are CouchDB, that acts as data storage layer, and it has been selected because it is written in Erlang and because is a RESTful database; and Mochiweb that acts as the web server and it has been selected because it is also written in Erlang and it is a really light-weight and easy to use web server. Figure 5.1 shows the structure of this prototype which lacks business layer because there will not be any business logic in it. There is also a module called CDB that acts as an abstraction layer for CouchDB.

The next sections provide a description on these two technologies.

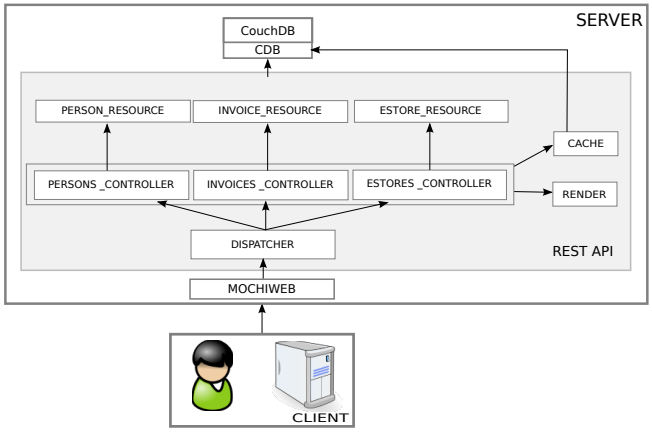


Figure 5.1: Prototype

5.2 CouchDB

CouchDB is not a relational database or an object-oriented database and is not intended to replace them. CouchDB is a distributed, fault-tolerant and schema-free document-oriented database accessible via a RESTful HTTP/JSON API. It is written in Erlang and it provides robusticity, incremental replication with bi-directional conflict detection and resolution, and can be queried and indexed in a MapReduce fashion using the JavaScript language.

5.3 Mochiweb

Mochiweb is an Erlang library for building lightweight HTTP servers [21]. Unlike the inet library, which is a handy Erlang library but not really a fully capable Web server, or Yaws, which is closer to Apache and very capable Web server but with many of its own ideas about how to build Web systems, Mochiweb is really flexible and easily adaptable to REST HTTP server.

Mochiweb does the heavy lifting on the outward-facing HTTP connections, leaving the developer to write his application-specific server logic. It provides useful abstractions over the request and response, and manages the client connections and other details of pushing data between the code and the clients.

5.4 The prototype

The prototype has been developed strictly following the design described in the previous chapter and it has been proved to be a good and easy to implement design. Having mentioned that, it is also important to say that some difficulties have been found during the implementation process:

- Some integration problems with CouchDB appeared. Specifically, the difficulties happened when trying to update a document and retrieve a query result when there were both lists and strings in the data. The problem turned out to a be misunderstanding of the JSON library.
- When trying to render XHTML, the SGTE template engine library written in Erlang has been used to transform data in Erlang tuples to XHTML. The documentation of this library is really short and unclear and that resulted in longer implementation time.
- When receiving a PUT HTTP request, there has been the need of creating a function that checks the correctness of the changed fields and merges the current fields with the updated ones. This should apparently be a simple task but has turned out not to be that simple.

When writing the code it has been always taken into account the best practises of developing in general and developing Erlang specifically. It has been also a goal to produce neat and simple code with really short functions that are easy to debug.

In general, the implementation process has been easy and smooth and that is in part because of Erlang and also because of the simplicity of the REST style itself. It has been simple to extract data from the request like the URI, the method, the content type and the body and then build a clear and generic flow that handles all kind of requests and takes care of error handling.

In the rest of the section is provided a set of small pieces of code that shows how the main functionality of the REST API looks in Erlang to prove the simpleness of implementing a REST API.

Figure 5.2 shows how the *dispatcher* module handles the synchronous call to the appropiate controller and then generates the HTTP response in a way or another depending on the result received from the controller.

```
run_controller(Controller, Args) ->
    case (catch apply(Controller, dispatch, Args)) of
        {'EXIT', _} ->
            [{status, ?NOT_FOUND}, {content, "text/html", "Not Found"}];
        {Status, ContentType, Data} ->
            [{status, Status}, {content, ContentType ++ ";" ++ ?CHARSET, Data}];
        {Status, ContentType, "no-cache", Data} ->
            [{status, Status},
             {content, ContentType ++ ";" ++ ?CHARSET, Data},
             {header, {cache_control, "no-cache, must-revalidate"}}];
        {Status, ContentType, {LastModified, ETag}, Data} ->
            [{status, Status},
             {content, ContentType ++ ";" ++ ?CHARSET, Data},
             {header, {cache_control, "no-cache, must-revalidate"}},
             {header, "Last-Modified:" ++ LastModified},
             {header, "ETag:" ++ ETag}];
        ?NOT_MODIFIED ->
            [{status, ?NOT_MODIFIED}]
    end.
```

Figure 5.2: Function run_controller from the dispatch module

Figure 5.3 shows the main function of any controller. This function decides which subfunction will handle the request depending on the URI specified in the request.

```
dispatch({_Req, _ResContentType, Path, _Meth} = Args) ->
    F = case Path of
            "" ->
                top;
            ["new"] ->
                new;
            ["search"] ->
                search;
            [_, "update"] ->
                update;
            [_] ->
                invoice;
            _ ->
                erlang:error(bad_uri)
    end,
    apply(?MODULE, F, [Args]).
```

Figure 5.3: Function dispatch from a controller module

Figure 5.4 shows the function in the controller that handles the */invoices* URI. It has four clauses, each one of them taking care of one HTTP method. As can be seen, caching, rendering and HTTP errors are handled by this small function.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% /invoices
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
top({Req, ResContentType, _Path, get}) ->
    QueryString = Req:parse_query(Req),
    try
        Data = invoice_resource:get_list(QueryString),
        Response = invoice_render:get_list(Data, ResContentType),
        {?OK, ResContentType, "no-cache", Response}
    catch
        throw:bad_request->
            {?BAD_REQUEST, "text/plain", "Bad request"}
    end;
top({Req, ResContentType, _Path, post}) ->
    Body =  Req:parse_post(Req),
    try
        Invno = invoice_resource:create(Body),
        Response = invoice_render:create(ResContentType),
        rest_cache:set_validation(invoice, Invno),
        {?OK, ResContentType, Response}
    catch
        throw:bad_request ->
            {?BAD_REQUEST, "text/plain", "Bad request"};
        error:_ ->
            {?INT_ERROR, "text/plain", "Internal server error"};
        throw:server_error ->
            {?INT_ERROR, "text/plain", "Internal server error"}
    end;
top({_Req, _ResContentType, _Path, delete}) ->
    {?BAD_METHOD, "text/plain", "Bad method"};
top({_Req, _ResContentType, _Path, put}) ->
    {?BAD_METHOD, "text/plain", "Bad method"}.
```

Figure 5.4: Function controller_top from the invoices_controller module

Figure 5.5 shows the function in the controller that handles the */invoices/invoice* URI. As can be seen, the structure of the function is exactly the same as it is for the previous case which talks good about the generity and abstraction of the code.

```
%% /invoices/invoice
invoice({Req, ResContentType, [Id], get}) ->
    try
        Headers = Req:get(headers),
        case rest_cache:get_validation(invoice, Id, Headers) of
            ?NOT_MODIFIED ->
                ?NOT_MODIFIED;
            CacheHeaders ->
                Data = invoice_resource:get(Id),
                Response = invoice_render:get(Data, ResContentType),
                {?OK, ResContentType, CacheHeaders, Response}
        end
    catch
        throw:bad_uri ->
            {?NOT_FOUND, "text/plain", "Not found"}
    end;
invoice({_Req, _ResContentType, _Id, post}) ->
    {?BAD_METHOD, "text/plain", "Bad method"};
invoice({_Req, ResContentType, [_Id], delete}) ->
    {?BAD_METHOD, "text/plain", "Bad method"};
invoice({Req, ResContentType, [Id], put}) ->
    Body = Req:parse_post(),
    try
        invoice_resource:update(Path, Body),
        Response = invoice_render:update(ResContentType),
        CacheHeaders = rest_cache:set_validation(invoice, Id),
        {?OK, ResContentType, CacheHeaders, Response}
    catch
        throw:bad_uri ->
            {?NOT_FOUND, "text/plain", "Not found"};
        throw:bad_request ->
            {?BAD_REQUEST, "text/plain", "Bad request"};
        throw:server_error ->
            {?INT_ERROR, "text/plain", "Internal server error"}
    end.
```

Figure 5.5: Function controller_invoice from the invoices_controller module

Chapter 6

Prototype integration

This chapter describes the integration process of the implemented prototype into the Klarna system. It starts with a brief explanation about the techonologies used in that system such as the Mnesia database and the Yaws webserver. Furthermore, a description of how the REST API has been tested is provided.

6.1 Mnesia

Mnesia is a distributed Telecommunications DataBase Management System (DBMS), appropriate for telecommunications applications and other Erlang applications which require continuous operation and soft real-time properties. The implementation is based on features in the Erlang programming language, in which Mnesia is embedded. Mnesia is considered an extension and an application of Erlang.

Mnesia was created by Håkan Mattson, Hans Nilsson and Claes Wikström to offer the functionality required for the implementation of fault tolerant telecommunications systems in nonstop systems and to accomplish the requirements on the DBMS to run in the same address space as the application. With this, Mnesia tries to address all of the data management issues required for typical telecommunications systems, not addressed by traditional commercial DBMSs.

According to [22] Mnesia was designed with the typical data management problems of telecommunications applications in mind:

- Fast realtime key/value lookup
- Complicated non realtime queries mainly for operation and maintenance
- Distributed data due to distributed applications
- High fault tolerance
- Dynamic re configuration
- Complex objects

Mnesia provides the following features which combine to produce a fault-tolerant, distributed database management system written in Erlang:

- **Schema manipulation routines.** Description of all tables are kept in a DB schema. It is possible to reconfigure the Mnesia schema at runtime without stopping the system, since Mnesia is intended for nonstop applications, and without being noticed by the application running at the same time.
- **A specifically designed and powerful query language.** Apart from traffic processing, telecommunications systems contain substantial amounts of operational and maintenance (O & M) code which requirements [22] which can be accomplish by
- **Fast real time data searches.** For some systems is really important to do a lookup operation efficiently. Mnesia allows the user to handle complex values efficiently and naturally in order to the data to be structured and stored in such a way so that relevant data can be accessed in a single lookup operation. Mnesia allows the user to user arbitrarily complex objects both as attribute values but also as key values in the database.
- **Fault tolerance.** Both the DBMS and also the application must be able to continue to provide its services even if errors occur (nonstop systems). The DBMS must provide the application designers with mechanisms whereby a sufficiently fault tolerant system can be designed. Mnesia tables can be moved or replicated to several nodes to improve fault tolerance being coherently kept on disc as well as in main memory. Mnesia can recover partially from complete disasters and distinguish safely data from garbage.
- **Distribution and location transparency.** Mnesia provides location transparency as well as the ability to explicitly locate data. Programs can be written without knowledge of the actual location of data since their address table names and the system itself keeps track of table locations.
- **Transactions and ACID.** DBMSs have ACID (Atomicity, Consistency, Isolation and Durability) properties, implemented in Mnesia by means of transactions, writeahead logging and recovery.

6.2 Yaws

Yaws, Yet Another Web Server, is an Erlang web server written in Erlang. It can operate as an embedded webserver in another erlang application or just runs as a regular webserver daemon, which is the default mode [23].

Using Erlang makes it able to handle a very large number of concurrent connections due to the fact that Yaws uses Erlang's lightweight threading system. A load test was conducted in 2002 [24] comparing Yaws and Apache in which Yaws supported 20 times the number of concurrent connections supported by Apache.

6.3 Prototype integration

The prototype build in Chapter 5 has been integrated in the Klarna system to properly check if it would be a good solution for Klarna. During the integration process, the prototype has just suffered some minor changes but its environment has changed completely as can be seen in Figure 6.1.

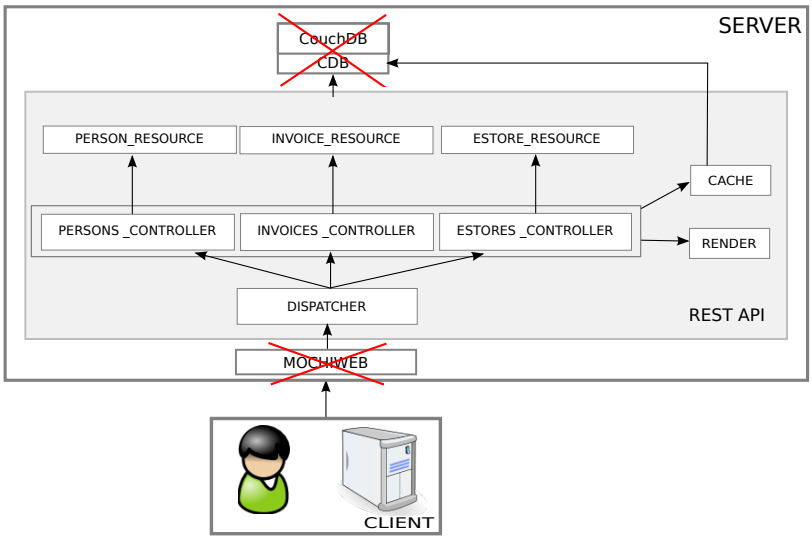


Figure 6.1: Prototype inside the Klarna System

The first big change has been replacing CouchDB with Mnesia. It has not been a complex task since the implementation of the prototype was done in a generic and database independent way. What has been done to achieve this is replace all the calls to CDB module with calls to KDB module which a module in the Klarna system that talks with Mnesia.

The second big change has been another technology replacement, in this case use Yaws instead of Mochiweb. This change has been even easier given the fact that the Mochiweb funcionalities used in the prototype work really similar in Yaws. It has been a straightforward task.

Another important modification has been cache handling. In the prototype there were some CouchDB views that took care of this task, but in the Klarna system this freedom does not exist. After some research in the code of the Klarna system it has been decided to use a group of Mnesia tables that are currently used to store many different kind of data related to the different resources in the system.

The last and most complex change has been interacting with the Klarna business layer when adding and updating invoices, e-stores and persons. It has been really difficult to find out which functions are supposed to be called when doing this actions since the code base of the Klarna system is huge and maybe not well structured. But after some investigation and help from the supervisor of the thesis it has been achieved.

In general, the integration process has not been complex and the genericity and abstraction of the initial prototype has been really helpful in this phase.

The REST API fully integrated in the Klarna system is shown in Figure 6.2.

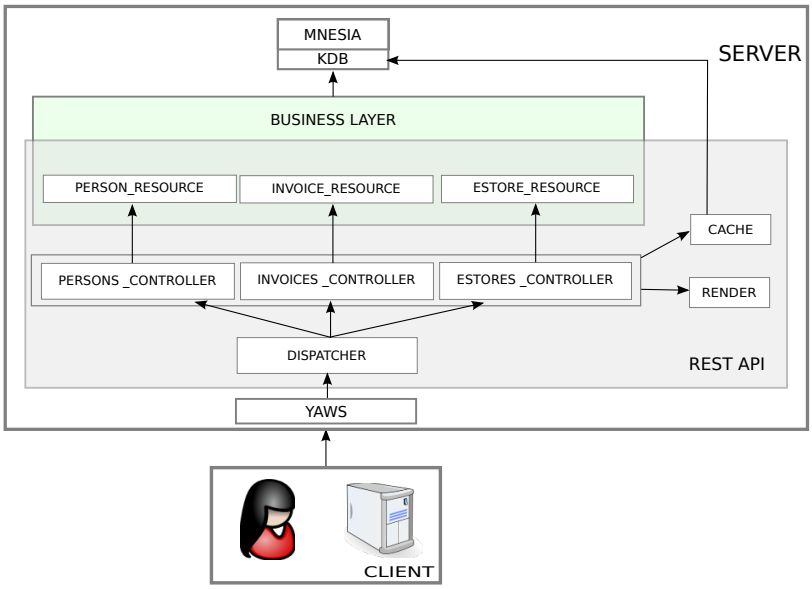


Figure 6.2: Detailed prototype inside the Klarna System

The next sections describe how the REST API has been tested in the two environments needed: computer-to-computer and human-to-computer interaction.

6.4 Testing with Curl

To test if the REST API inside the Klarna system properly interacts with other machines the Curl tool has been used. Curl is a command line tool for transferring data with URL syntax, supporting FTP, FTPS, HTTP, HTTPS, SCP, SFTP, TFTP, TELNET, DICT, LDAP, LDAPS, FILE, IMAP, SMTP, POP3 and RTSP. Curl supports SSL certificates, HTTP POST, HTTP PUT, FTP uploading, HTTP form based upload, proxies, cookies, user+password authentication (Basic, Digest, NTLM, Negotiate, kerberos...), file transfer resume, proxy tunneling and a busload of other useful tricks [25].

Curl has a specific syntax that allows a human person to create, retrieve and update a resource as a machine would do. The next line shows the format of the commands that have been used to test the system designed.

```
curl -i -k -H headers -X method -d data
```

Where,
-i includes the HTTP-header in the output. The HTTP-header includes things like server-name, date of the document, HTTP-version, etc.

-k explicitly allows curl to perform "insecure" SSL connections and transfers.

-H allows the user to define HTTP headers.

And,
-d sends the specified data in an HTTP POST request.

Using curl, a set of different requests has been sent to the server to check how the REST API behaves when interacting with computer clients. These requests involve different HTTP methods, different URIs and also different content-type expected, in other words, it will be checked that the server can generate responses both in JSON and XML. Next, a list of all the curl commands executed to send requests and the responses received by each one are shown:

```
Create person:
curl -i -k -H "Accept: application/xml" -X POST -d "eid=2"
-d "pno=YYMMDDXXXX" http://klarnarestful/persons/

Response:
HTTP/1.1 200 OK
Server: Yaws/1.87 Yet Another Web Server
Date: Thu, 20 May 2010 13:51:50 GMT
Content-Length: 44
Content-Type: application/xml;charset=iso-8859-1

<?xml version="1.0"?><response>ok</response>
```

```
Get list of invoices:
curl -i -k -H "Accept: application/json"
-X GET http://klarnarestful/invoices/

Response:
HTTP/1.1 200 OK
Server: Yaws/1.87 Yet Another Web Server
Cache-Control: no-cache, must-revalidate
Date: Thu, 20 May 2010 13:20:29 GMT
Content-Length: 141
Content-Type: application/json;charset=iso-8859-1

[{"link":"/rest/invoices/11004209676083579","invno":11004209676083579},
{"link":"/rest/invoices/11004239782295973","invno":11004239782295973}]
```

```
Get invoice:
curl -i -k -H "Accept: application/xml"
-X GET http://klarnarestful/persons/5311096845

Response:
HTTP/1.1 200 OK
Server: Yaws/1.87 Yet Another Web Server
Cache-Control: no-cache, must-revalidate
Date: Thu, 20 May 2010 13:18:09 GMT
Content-Length: 253
Content-Type: application/xml;charset=iso-8859-1
ETag: ï¿½y\[=ï¿½|ï¿½iï¿½ï¿½kp
Last-Modified: Wed, 21 Apr 2010 12:49:05 GMT

<?xml version="1.0"?>
<person><invoices_link>/rest/inivoices?pno=531109-6845</invoices_link>
<pno>531109-6845</pno><fname>Maud</fname><lname>Johansson</lname>
<street>Sveavagen</street><zip>12149</zip><city>Johanneshov</city>
<country>209</country></person>
```

```
Search resource:
curl -i -k -H "Accept: application/json"
"http://klarnarestful/persons?country=209"

Response:
HTTP/1.1 200 OK
Server: Yaws/1.87 Yet Another Web Server
Cache-Control: no-cache, must-revalidate
Date: Thu, 20 May 2010 13:59:08 GMT
Content-Length: 365
Content-Type: application/json;charset=iso-8859-1

[{"link":"/rest/persons/YYMMDD-XXXX","fname":"Nadia","lname":"Mohedano"},
{"link":"/rest/persons/430415-8399","fname":"Karl","lname":"Lidin"},
{"link":"/rest/persons/672021-7931","fname":"Bjï¿½rnligan AB","lname":""},
{"link":"/rest/persons/531109-6845","fname":"Maud","lname":"Johansson"},
{"link":"/rest/persons/602031-0139","fname":"Kalle Anka AB","lname":""}]
```

```
Update person:
curl -i -k -H "Accept: application/json" -X PUT -d "zip=13526"
http://klarnarestful/persons/6020310139

Response:
HTTP/1.1 200 OK
Server: Yaws/1.87 Yet Another Web Server
Cache-Control: no-cache, must-revalidate
Date: Thu, 20 May 2010 14:04:22 GMT
Content-Length: 11
Content-Type: application/json;charset=iso-8859-1
ETag: ï¿½ï¿½ï¿½mrz>ï¿½ï¿½ï¿½"ï¿½E
Last-Modified: Thu, 20 May 2010 12:04:22 GMT

{"ok":true}
```

The results obtained are really satisfactory. The built REST API is perfectly capable of receving requests from other computers and generate a responses that they can understand. Furthermore, it is also capable of generating the responses both in XML and JSON. In order to be sure that the code on the response is correct JSON and XML, that code has been validated in web based validators for JSON and XML.

Thus, it has been proved that the current XMLRPC API that Klarna uses to interact with banks and e-stores through computer-to-computer communication can be replaced by the REST API implemented.

6.5 Testing throguh GUI

The user interface has been created to test that it is simple and user friendly for human users to access the REST API and that allows them to retrieve, create and update resources. Next, it is shown a sequence of events that a human user could do browsing the GUI:

Figure 6.3 shows the home page of the GUI. The header remains the same on all pages and it contains Klarna's logo on the left side, which is linked to Klarna's webpage, and a list of links to the home page, list of e-stores, invoices and persons.

Then the user would for example click on the invoices link and in the new invoice link and the user will see what is shown in Figure 6.4. This page, as most of the pages in this GUI, contains a right menu to improve the experience of the user and allows him or her to quickly navigate to different parts of the GUI. The main box of the page shows a simple form to create an invoice adding e-store id, person number and a list of goods. When clicking the Create button an HTTP POST request will be sent to *https://klarnarestful/invoices*.

Figure 6.5 shows the page that handles the creation of a new person. As said on Chapter 4 the only data the user has to introduce is the person number and the id of the e-store the person is linked to.

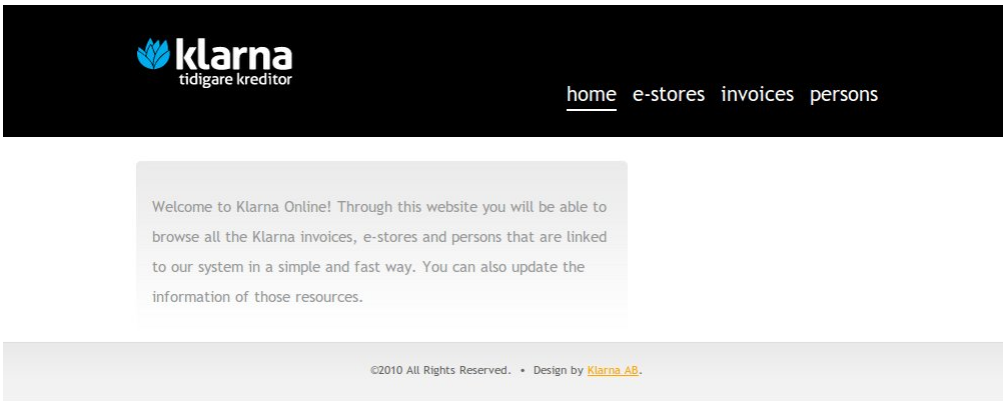


Figure 6.3: User Interface Home Page [http://klarnarestful]

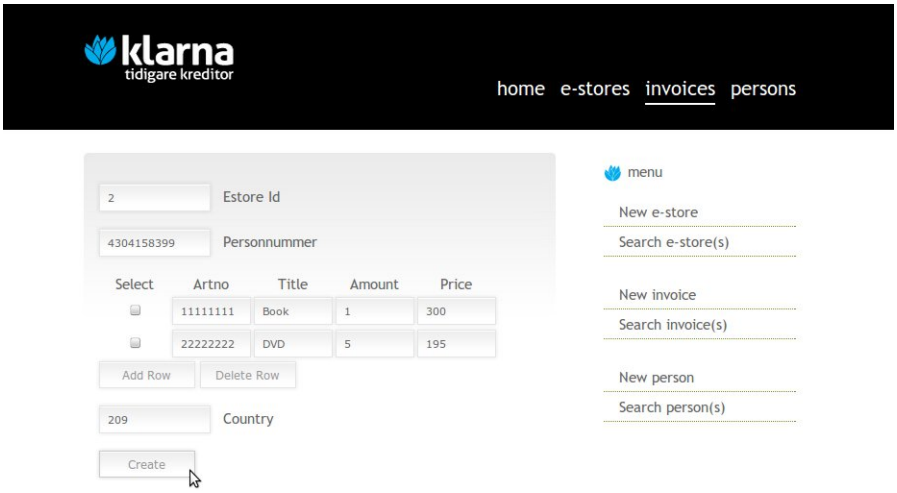


Figure 6.4: New Invoice Page [http://klarnarestful/invoices/new]

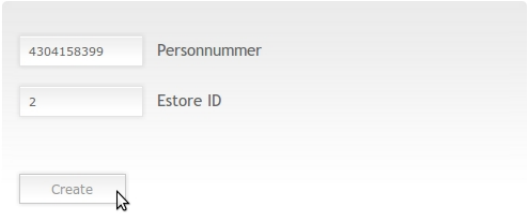


Figure 6.5: New Person Form [http://klarnarestul/persons/new]

Figure 6.6 shows the page that displays the list of last persons created in the system. The right menu and the header are not shown since they look exactly the same as in the previous pages. Each name in the list is a link to the person's page.

Figure 6.6: Retrieved List of Persons [http://klarnarestful/persons]

Figure 6.7 shows the result of a search among the set of persons in the system. The criteria is to select all the persons whose address is in country 209 (Sweden).

Figure 6.7: Search List of Persons From Sweden [http://klarnarestful/person?country=209]

Figure 6.8 shows another search result in the person resources but with slightly different criteria and that is to have a limit of 2 matching results.

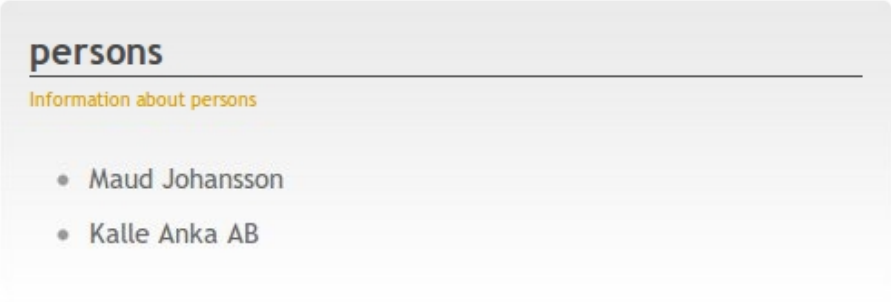


Figure 6.8: Search List of Persons From Sweden (maximum 2 persons) [http://klarnarestful/person?country=209&limit=2]

Figure 6.9 shows the information of a specific invoice including the goods list, a link to the person that made the purchase *http://klarnarestful/persons/SE430415-8399* and a link to the e-store in which the purchase was done *http://klarnarestful/estores/2*. This again show the connectedness of the REST API. There is also a Modify button that will lead the user to an update form to edit the data of the invoice.

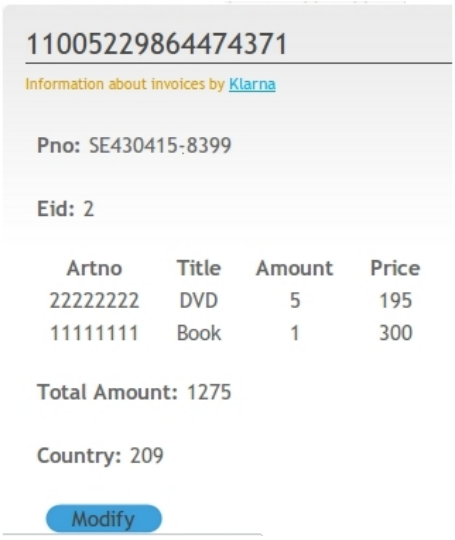


Figure 6.9: Get Invoice [http://klarnarestful/invoices/11005229864474371]

Figure 6.10 shows how the data of the e-store with identity 4 is shown in the GUI. As the invoice page, it also contains a Modify button to update the estore. Below the information of the e-store, there is useful link to the list of all the invoices of this e-store *http://klarnarestful/invoices?eid=4*.

Figure 6.10: Get Estore [http://klarnarestful/estores/4]

Lastly, Figure 6.11 shows the form that allows the user to update an e-store. The form is already filled with the current information of the resource and the user can edit what is needed. When clicking the update button, a HTTP PUT request will be sent to *http://klarnarestful/estores/4*.

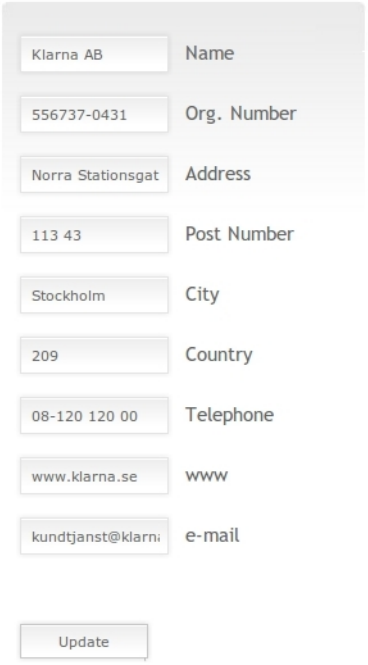


Figure 6.11: Update Estore [http://klarnarestful/estores/4/update]

The implementation process of this simple GUI has last more than a week but it has worth it. It has been proved that a REST API is not just easy to develop, but it is also easy to use for human users. It cannot be forgotten that many REST principles provide better usability for the site and hence better experience when browsing it and users get a high benefit from that.

Chapter 7

Conclusions

Most of the financial systems are nowadays web-based systems accessible from anywhere in the world via the Web. In this project, a RESTful WebService have been implemented previous design of the RESTful API, and the results obtained are really satisfactory. Thus, it can be said that these web-based financial systems have many complex requirements which are easier to accomplish with REST.

After doing the investigation about the REST architectural style it can be said that it would be a proper solution. It has been replaced the XML-RPC API and the GUI API with a unique and generic REST API that handles all the services. Furthermore, the integration of the prototype designed in the Klarna system as have been demonstrated on Chapter 6 works properly being accessed through a simple proof of concept GUI and through the command line. Thus, it has been checked that both a human and a computer can make use of it and that the data can be returned in different representation types. Furthermore, a really generic API has been built and it seems obvious that it can be reused for any kind of web-based system since it is really generic. Lastly, it has been proved that once one understand the main concepts of REST it is not a complex task to implement such API as it appeared at the beginning.

7.1 Future work

Lots of improvements and extra implementation can be added in the current API:

- The current system do not cover every possible action handled by the Klarna system and do not provide all the resources that are needed by it. This activity will most likely take plenty of time but it would be mandatory if this solution is to be applied in the real system.
- Authentication handling should be added to the API so that just some specific users have rights to create and update resources.

- HTTP Secure support should be added to achieve a more secure platform that is mandatory when payment transactions are involved.
- Some testing in terms of performance (response time, load on the server) should be done.

Bibliography

[1] What is a web based system? http://wiki.answers.com/.

[2] R. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. IETF RFC 2616, June 1999.

[3] T. Berners-Lee et al. *Hypertext Transfer Protocol - HTTP/1.0*. IETF RFC 1945, May 1996.

[4] R. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. IETF RFC 2068, January 1997.

[5] R. Fielding, editor. *RFC for REST*. REST Discussion Mailing List, 2006.

[6] R. Fielding. *Architectural Styles and The Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[7] L. Richardson, S. Ruby, et al. *Restful Web Services*. O'Reilly, 1st edition, May 2007.

[8] World wide web consortium. http://en.wikipedia.org/, May 2010.

[9] D. Booth et al. *Web Services Architecture*. W3C, February 2004.

[10] M. Gudgin et al. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. W3C Recommendation. W3C, April 2007.

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

[12] E. Christensen et al. *Web Services Description Language (WSDL) 1.1*. W3C Note. W3C, March 2001.

[13] C. Pautasso, O. Zimmermann, and F. Leymann. *RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision*. IW3C2, April 2008.

[14] Restful webservices. http://www.slideshare.net/gouthamrv/restful-services-2477903, Dec 2008.

[15] P. James. Http caching. http://www.peej.co.uk/articles/http-caching.html, Sep 2006.

[16] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent programming in Erlang*. Prentice Hall, 2nd edition, 1996.

[17] F. Cesarini and S. Thompson. *Erlang Programming - A Concurrent Approach to Software Development*. O'Reilly, June 2009.

[18] B. Goldberg. Functional programming languages. *ACM Computing Surveys*, 28(1):249–251, March 1996.

[19] Pattern matching. http://en.wikipedia.org/, April 2010.

[20] S. St.Laurent, J. Johnston, E. Dumbill, and D. Winer. *Programming Web Services with XML-RPC*. O'Reilly, June 2001.

[21] What is mochiweb for? http://erlang.2086793.n4.nabble.com/.

[22] H. Mattson, H. Nilsson, and C. Wikström. *Mnesia A Distributed Robust DBMS for Telecommunications Applications*.

[23] Yaws, yet another webserver. http://yaws.hyber.org.

[24] A. Ghodsi. Apache vs. yaws. http://www.sics.se/ joe/apachevsyaws.html, January 2007.

[25] curl. http://curl.haxx.se/.