INTERNATIONAL
HELLENIC
UNIVERSITY

Anonymous auction using REST Services

Ioannis Ioannidis

SID: 3301130008

SCHOOL OF SCIENCE & TECHNOLOGY

A thesis submitted for the degree of

Master of Science (MSc) in Information and Communication Systems

December 2015

THESSALONIKI – GREECE

INTERNATIONAL
HELLENIC
UNIVERSITY

Anonymous auction using REST Services

Ioannis Ioannidis

SID: 3301130008

Supervisor: Prof. Apostolos Papadopoulos

Supervising Committee Members:

SCHOOL OF SCIENCE & TECHNOLOGY

A thesis submitted for the degree of

Master of Science (MSc) in Information and Communication Systems

December 2015

THESSALONIKI – GREECE

Abstract

Evolution of technology is running rampant, especially in mobile, IoT and cloud computing, therefore, there is a great need for developing contemporary web applications which will be alert to follow this evolution. In computer software design, Service-Oriented Architecture (SOA) is an architectural pattern in which application components provide services to other components via a communication protocol, typically over a network. SOA can be implemented at many different environments. The implementation of SOA in web environments is called Web Service.

In this thesis, we develop an online auction system, which is supported by web services and specifically by the newer standard, REST. The differentiation that this thesis provides, unlike other online auction systems, is the participants' identity concealment through a third party.

My sincere gratitude to my supervisor, Assistant Professor Apostolos Papadopoulos, who provided me an opportunity to work on this modern topic. Besides my supervisor, I would like to thank Dr Christos Berberidis for the continuous support in the last two years.

Ioannis Ioannidis

11/12/2015

Contents

1 Introduction

In previous years, monolithic applications were responsible for accomplishing the whole service. These applications were independent from other computing applications and the philosophy behind it was that, not only was the application responsible for a small task, but rather to perform every step needed to complete the whole function.

Software developers and programmers always strive to create software that is easy to maintain, easy to debug, as well as to be effortlessly extended and integrated with other systems. Throughout the years, various approaches in order to design and write software have been pursued. In the early stages of the software industry, programmers perceived that by organizing their code into modules, actually made it easier to maintain and reuse pieces of functionality. This was the beginning of building a collection of non-volatile resources, such as routines, classes, methods and documentations, known today as libraries. The next big idea in software design was object orientation. Object-oriented programming (OOP) is based on the concept of objects. An object is a software entity which has state (properties) and behavior (methods). Software objects are often used to model the real-world objects that you can find in everyday life. The implementation details of an object are hidden from the outside world (encapsulation) and the object can be changed as long as the object's signature remains the same. This has helped maintenance and scalability. However, object orientation was not enough. Trying to meliorate further the software industry, developers and programmers realized that the monolithic approach had many drawbacks. Some of them are the difficulty of making changes and improvements even in small parts of the application, the high maintenance costs as well as the huge set-up costs. On the other hand, by building small and independent applications with relatively limited functionality that could be plugged into many different software applications with different needs, led to the concept of software components and the term "service orientation". In this approach, software functionality is defined as a set of services. This is where Service Oriented Architecture (SOA) comes in.

1.1 Service Oriented Architecture

In computer software design, Service-Oriented Architecture (SOA) is an architectural pattern, in which application components provide services to other components via a communication protocol, typically over a network. The principles of service orientation are independent of any vendor, product or technology.[(1)]To put it differently, it is an architectural style for building business applications using loosely coupled services, which act like black boxes and can be well orchestrated in order to achieve functionality by linking them together. A service is a logical encapsulation of a clearly defined business functionality that operates independently from other services. Services should be self-contained, self-defined and communicate with each other using messages that are reliable and cross platform.

There are some basic principles that services in SOA implementation must follow:

- Loose coupling: Services must not have high dependency with each other.
- Abstraction: A service should hide its internal implementation from the outside world.
- Reusability: Services must be capable of being used again and again in several applications, instead of rewriting them.
- Statelessness: There must be no record of previous interactions and each interaction request has to be handled based entirely on information that comes with it.

1.1.1 Benefits of Service Oriented Architecture

There are many benefits by following the SOA style and methodology. The most important of them are mentioned below:

- *Platform independence*: SOA is based upon the use of services, which are available through standard technologies. The use of standard technologies reduces heterogeneity and is therefore the key to facilitate application integration. For instance, if an enterprise wants to extend its existing legacy applications and build additional functionality on top of that, with the approach of service orientation is feasible, even if this new service runs on different hardware, is written in different programming language or stores data in different format. It also helps an enterprise to integrate its applications with those of its partners.

- *Discrete developer roles*: Since a service has a distinct role, its implementation is independent from other services. Hence, developers who are in charge of a service can focus completely on implementing and maintaining that particular service without having to worry about other services.
- *Code reuse*: As it is already mentioned, SOA approach breaks down an application into small independent pieces of services. Services then can be reused in multiple applications. A direct consequence of that is the reduction of development cost.
- *Better testability*: It is reasonable that small and independent services are easier to test and debug than monolithic applications. This leads to more reliable software.
- *Parallel development*: Since services are independent of each other, they can be developed in parallel. This cuts down the software development life cycle markedly.
- *Better scalability*: A service can be easily moved to many servers. Moreover, there can be multiple instances of that service running on these different machines. This increases scalability.
- *Higher availability*: Due to the fact that you can have multiple instances of a service running on different servers, higher availability is ensured.

1.2 Web Services

SOA can be implemented at many different environments. The implementation of SOA in web environments is called Web Service. A web service is simply an application that is exposed to the internet and is accessible using standard web technologies. Basically, it is an online API (Application Programming Interface) that someone can call from his application in order to enhance its functionalities. The difference between a web application and a web service is that the former is intended for human consumption, usually accessed by a web browser and presented in HTML format, whilst the latter is meant for application level consumption and mainly present data in either XML or JSON format. Essentially, what a web service does is that allows two different applications, running on two different servers to be able to “talk” to each other. The main advantage of web services is that are interoperable and platform-hardware-location-programming language independent.

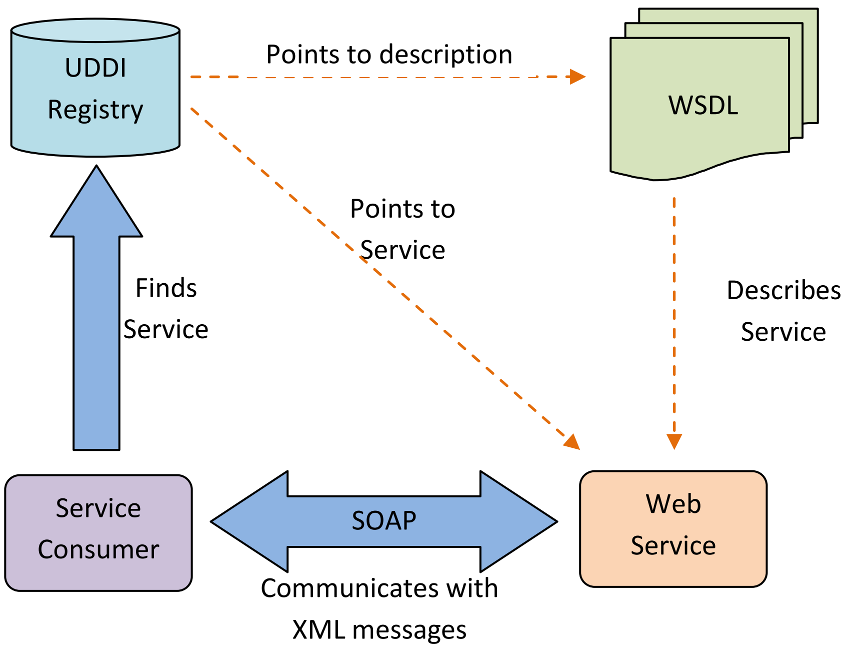
Primarily, there are two different types of web services. SOAP web services and REST web services. In Java world, JAX-WS (Java API for XML Web Services) is the specification that provides support for creating SOAP web services, while JAX-RS (Java API for RESTful Web Services) is the API that provides support in creating web services according to the Representational State Transfer (REST) architectural pattern.

1.3 Simple Object Access Protocol (SOAP)

SOAP is a standard-based web service access protocol which was firstly developed by Microsoft in order to take the place of older technologies like CORBA (Common Object Request Broker Architecture). Fundamentally, it is a set of rules for XML-based message exchange.

In order to have a better understanding of how this technology works, some terms need to be mentioned. Since we are talking about web services, it means that the data is exchanged over the web (internet). In order though, someone to be able to call a web service, he has to know what this web service does. In other words, an interface, a “contract”, must be shared among the potential consumers of that particular service. It is imperative that this interface be in a technology independent format. This format is an XML document which is called WSDL (Web Service Definition Language). What this document contains is very similar to what an interface contains in every object oriented programming language, such as the operations you can call, the arguments that they take, and the returned type they send back. However, in order to get the WSDL document you must firstly find it. This is where UDDI (Universal Description Discovery and Integration) comes into play. It is a directory model for web services, like yellow pages. In essence, it provides a registry mechanism for clients and servers to find each other. UDDI with SOAP and WSDL is thought as the three foundation standards of web services.

The figure below illustrates how SOAP web service technology works:



Picture 1-1[1]: SOAP Web Service Technology

1.4 Representational State Transfer (REST)

REST web services have gained widespread acceptance across the Web as a simpler alternative to SOAP. The first thing it should be mentioned is that REST is not a technology. It is an architectural style, by which you can design web services. Its concept is very closely related to HTTP (Hypertext Transfer Protocol). The reason why is due to its inventor, Roy Thomas Fielding, who was also one of the principal authors of the HTTP specification. In his academic dissertation, "Architectural Styles and the Design of Network-based Software Architectures", suggests that a concrete implementation of a REST web service should follow four basic design principles [(2)]:

- *Use HTTP methods explicitly*: REST requires from developers to use HTTP methods explicitly and in a way that is consistent with the protocol definition. In particular, to retrieve a resource from a server, use GET. To create a resource on the server, use POST. To change the state of a resource or to update it, use PUT. To remove a resource, use DELETE.

[1]Taken from www.novell.com

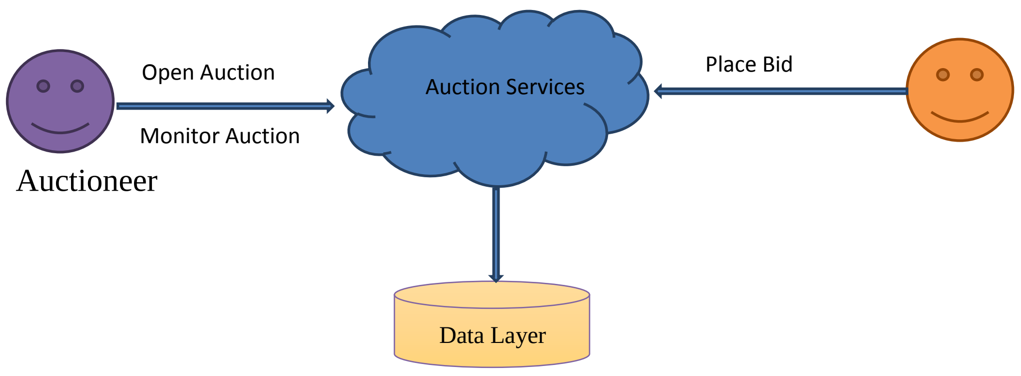
- *Be stateless*: Web service clients should send self-contained messages. That is, these messages should include within the HTTP headers and body all the parameters, context and data needed by the server-side component for generating a response. Furthermore, statelessness simplifies the design and implementation of server-side components.
- *Expose directory structure-like URIs*: Web service URIs should be intuitive to the point where they are easy to guess. These URIs, should be hierarchical, rooted at a single path, and their branches must expose the service's main areas. Moreover, a good practice is to have a static, resource-based URI, instead of action-based, for the reason that if a resource changes or the implementation of the service changes, the URI stays the same.
- *Transfer XML, JSON or both*: A resource representation typically reflects the current state of a resource. A RESTful web service client must be able to ask for data in a format that is more suitable for him (different representation of the same resource). As follows, the server has to support several MIME types. This approach allows the service to be used by a variety of clients, written in different programming languages, running on different platforms.

1.5 Goal of the Thesis

The goal of this dissertation is to develop an online auction system which is supported by web services and specifically by the newer standard, REST. This auction system can be designed as a set of different parts which are all capable to communicate with each other, and each one is responsible for a single task. The parties originally involved in an online auction are:

- The owner of the product/service (auctioneer).
- The prospective buyers who make offers for the product/service.
- The auction service which coordinates the auction. In our case it will be the part which implements the business logic, and the other two parts will communicate with it so as to participate in the auction.

The auction system can be presented by the following diagram:

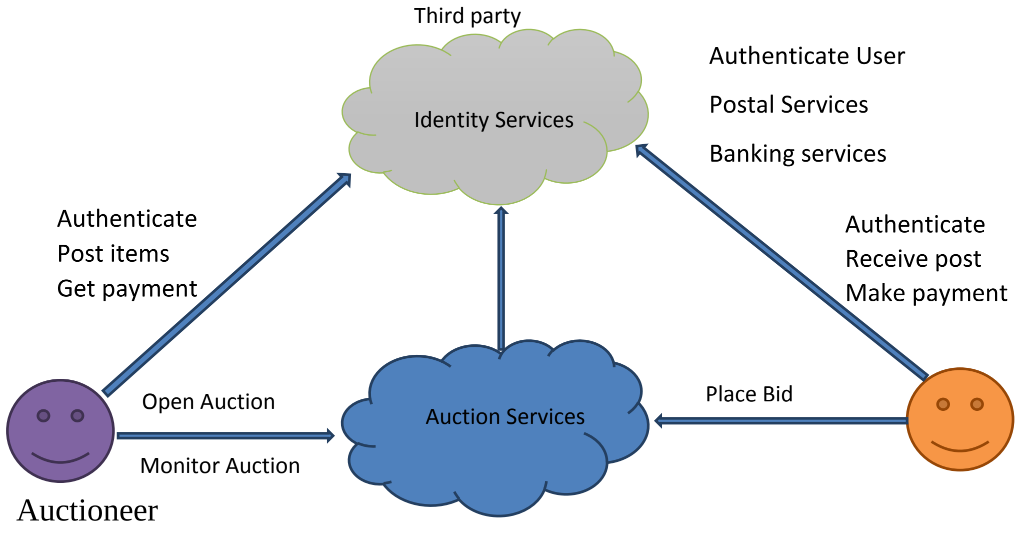


Picture 1-2: Representation of the Auction System

As it has been already pointed, one of the basic characteristics of REST web services is that the client-server communication is stateless. Hence, during the auction procedure, the auction service will not communicate directly with the potential buyers, but the stakeholders will send a request to the auction service and the latter will reply with a response.

The differentiation that this thesis provides, unlike other online auction systems, is the covering of the participants' identity through a third party. A common practice, especially when it comes to auction's products of great value, is that the parties involved wish to remain anonymous. This anonymity will be achieved by a third party, independent from the auction system, which plays a representative role. This is possible due to the messaging statelessness mentioned above.

Considering the previous scenario, when an auction is won by a certain buyer, he has to contact the auctioneer revealing his identity in order for the product to be sent to him, and also the auctioneer must do the same thing in order the buyer to make a deposit to his account. Nevertheless, in this case, neither the auctioneer nor the buyer can conceal their identities. That is why is necessary another independent party to be present for providing identity services, such as user authentication, postal and bank services. The new auction system is illustrated below.



Picture 1-3: Representation of the Final Auction System

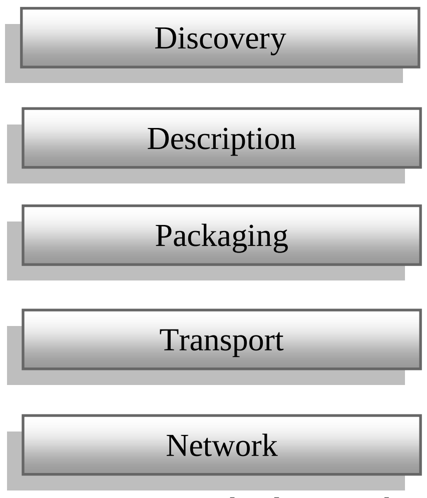
According to this diagram, the auctioneer will contact the Identity Server so as to identify himself to the Auction Service. The same will act the potential buyers. When the auction ends, Auction Server will send a request to Identity Service asking him to inform, both the winner of the auction and the seller of the product (auctioneer), about the auction outcome. As far as Identity Server is concerned, the notion of “auction” means nothing. The only thing he is aware of in this context, is to make a package transfer as well as to conduct a bank transaction. Hence, subsequently, Identity Server will withdraw money from the buyer’s account (winner of the auction), and put them in a “BLOCKED” state, which means that the money is not yet to the auctioneer’s account. No sooner has the package transfer been completed, than the money is deposited in his account. Only the third party (Identity) knows the actual address of the buyer and he will forward the package to him, while at the same time he will release the money for auctioneer’s behalf.

2 Literature Review

In the introductory part, it has been recognized the usefulness and the contribution that web services can provide in modern web development. Web services provide an abstract layer between the service consumer and the service provider, allowing cross-platform interoperability.

2.1 Simple Object Access Protocol (SOAP)

SOAP web service architecture is implemented through the layering of five types of technologies. These layers are organized and built one over the other [(3)].



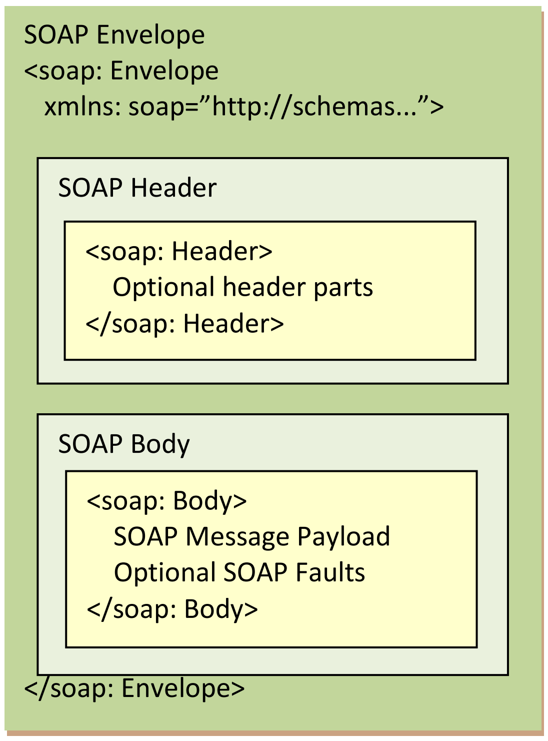
Picture 2-1: SOAP Technology Stack

The *Discovery* layer is the tool that allows a service consumer fetching the provider's service description. UDDI is the most well-known discovery tool. As soon as the web service is implemented, decisions such as which network, transport, and packaging protocols it will support, must be made. That way, the service consumer is able to use the service. WSDL is the standard for providing *Description*. This functionality is similar to an "interface" in object-oriented programming, except that WSDL is technology independent, as it is written in XML. In order for data to be moved around the network by the transport layer, a format that all parties can understand must be used for *Packaging* that data. SOAP protocol is a very common packaging format, built on XML. *Transport* layer is used for transferring data from one location to another. Technologies that are mainly used are: HTTP, TCP, FTP and SMTP. Lastly, the *Network* layer is exactly the same as the network layer in the TCP/IP Network Model.

2.1.1 SOAP Protocol

SOAP is an XML messaging protocol with a simple format, providing just a few conventions on how to structure headers and body in an XML message. SOAP is transport independent, thus, SOAP messages can be sent over several transport protocols, like HTTP, TCP and SMTP [(4)]. Since SOAP is XML, it should be noted that it is inextricably linked with XML standards like XML Schema and XML Namespaces.

A SOAP message consists of an envelope containing an optional header and a required body, as shown in the following figure.



Picture 2-2: SOAP Message Structure

The header block indicates how the message should be processed, while the body contains the actual message that is attempted to be exchanged. The XML syntax for expressing a SOAP message is based on the*http://www.w3.org/2001/06/soap-envelope* namespace. This XML namespace identifier points to an XML Schema that defines the structure of how a SOAP message should look like. A simple example that shows how a real SOAP message could look like, is given below [(3)]:

```
<s:Envelope xmlns:s="http://www.w3.org/2001/06/soap-envelope">
    <s:Header>
        <m:transaction xmlns:m="soap-transaction" s:mustUnderstand="true">
            <transactionID>1234</transactionID>
        </m:transaction>
    </s:Header>
    <s:Body>
        <n:purchaseOrder xmlns:n="urn:OrderService">
            <from><person>Christopher Robin</person>
                <dept>Accounting</dept></from>
            <to><person>Pooh Bear</person>
                <dept>Honey</dept></to>
            <order><quantity>1</quantity><item>Pooh Stick</item></order>
        </n:purchaseOrder>
    </s:Body>
</s:Envelope>
```

2.1.2 Web Services Description Language (WSDL)

The WSDL document is nothing but a simple XML document, which contains a set of definitions in order to describe a web service. For better understanding, it is helpful to list its main structure elements [(5)]:

Element	Description
<portType>	A set of operations supported by one or more endpoints. It describes a web service, the operations that can be performed, and the messages that are involved.
<types>	A container for data type definitions used by the web service. For maximum platform neutrality, WSDL uses XML Schema syntax to define data types.
<message>	A typed definition of the data being communicated. Each message can consist of one or more parts.
<binding>	A protocol and data format specification for a particular port type.

Table 1:WSDL Document Structure

The following XML sample shows the anatomy of a WSDL document:

```
<definitions ……>
        <types>
            “Data type definitions”
        </types>
        <message>
            “Definition of the data being communicated”
        </message>
        <portType>
            “Set of operations that take input and output messages”
        </portType>
        <binding>
            “What transport protocol is being used”
             “How the WS accepts requests and gives the response”
        </binding>
</definitions>
```

The <*definitions*> element is the root element of all WSDL documents. It defines the name of the web service. A piece of XML code showing the <definition> element is given below[2]:

```
<definitions name="HelloService"
targetNamespace="http://www.examples.com/wsdl/HelloService.wsdl"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://www.examples.com/wsdl/HelloService.wsdl"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  ..............................................
</definitions>
```

[2] http://www.tutorialspoint.com/

Once the WSDL document has been created and published, a service consumer must be able to “discover” it in order to use it. This is done with the help of UDDI which is a specification of web services’ registry. The UDDI registry is expressed in XML and allows a business to publicly record the services it provides. Subsequently, potential consumers of those services can locate them based on taxonomical information, such as what the service does or what industry the service targets [(3)]

2.2 Representational State Transfer (REST)

According to Roy Thomas Fielding, REST is a key architectural principle of the World Wide Web. In his doctoral dissertation, “*Architectural Styles and the Design of Network-based Software Architectures*”, he presented REST as a software architectural style for building scalable web services. Nowadays, it is strongly believed that web services can be implemented without using the SOAP approach, which is a technology of its own. In contrast, web basic technologies are good enough to be considered the default platform for distributed services. In other words, all you need is already there. What we want to avoid by using the web’s already built infrastructure is the unnecessary complexity and overhead that SOAP technology brings. In the following sections we will try to focus on both theoretical issues like what it means to be RESTful and why web services should be more RESTful instead of less, and practical issues such as how to design and implement RESTful web services.

2.2.1 Resource

A resource is everything that is worth being identified. Every “thing”, every “resource” gets an identifier. Usually, a resource is something that can be stored on a computer and represented as a stream of bytes. It can be a document like HTML, PDF, JSON or XML, a row in a database or the result of a running algorithm.

2.2.2 URIs

What makes the resource a “resource”, is the URI (Uniform Resource Identifier). The URI is the name and address of a resource [(6)]. Simply put, the job of a URI is to identify a resource or a collection of resources. By definition two different URIs should represent two different resources. However, sometimes it is possible the same resource to be referred by more than one URI. A good example is the following: If the current software release is 1.0.3, then *http://www.example.com/software/releases/1.0.3.tar.gz* and *http://www.example.com/software/releases/latest.tar.gz* will refer to the same file for a while. But the ideas behind those two URIs are different: one of them always points to a particular version, and the other points to whatever version is the newest at the time the client accesses it [(6)]. Nevertheless, the principles behind URIs are well described by Tim Berners-Lee in Axioms of Web Architecture [(7)]. A good practice behind constructing a URI is to think of it as it was a static web page with a particular structure (root, folders, sub-folders). It should not contain tech implementation details, for example *http://www.example.com/myapp/getMessages.do?id=10*, because these may change over time. The benefits of creating a URI following these simply rules is: high readability, easy to debug, easy for clients to construct more URIs. In respect to the previous example, a better way to write the URI would be *http://www.example.com/myapp/messages/10,* combining this URI with the appropriate HTTP method, depending on what you trying to do. However, there is no right or wrong way to create a URI. This is just a convention.

2.2.3 Statelessness

Statelessness means that every HTTP request is self-defined. In other words, when a client makes an HTTP request, it includes all the necessary information for the server to process that request. The server cannot “remember” information from previously sent requests. Because of that, it is easier to distribute a stateless application across load-balanced servers. Since two requests do not depend on each other, they can be processed by two different servers without having to coordinate them. This leads to more scalable applications. A stateless application is also easy to cache: a piece of software can decide whether or not to cache the result of an HTTP request just by looking at that one request [(6)]. The only way for a server to “remember” a client is by setting a *session*. A session can be maintained by a server with mainly two ways: Cookies and URL Rewriting (the session ID instead of being stored in the Cookie, is appended in the URL).

2.2.4 Representations

An application is split into resources. Yet, a resource is not the data. It is just the service designer’s idea of how to split up the data. A web server cannot send an idea. It has to send a series of bytes, in a specific file format. This is what is called the *representation* of the resource. In other words, a resource is a source of a representation, and a representation is the data about the current state of a resource, that is sent back to the client as a response [(6)]. A resource can be represented with various MIME types[3], like XML, JSON or HTML, among others. Strictly speaking, the server can provide the same resource in different representations. For instance, the same resource called "Person" can be represented both in XML and JSON as:

XML:

```
<Person>
    <ID>101</ID>
    <Name>Ioannis Ioannidis</Name>
    <Address>Aristotelous 13</Address>
    <City>Thessaloniki</City>
    <Country>Greece</Country>
</Person>
```

JSON:

```
{
    "ID": "101",
    "Name": "Ioannis Ioannidis",
    "Address": "Aristotelous 13",
    "City": "Thessaloniki",
    "Country": "Greece"
}
```

Interestingly, RESTful systems give you the ability to ask for data in a format that you can understand. If the server does not support the particular MIME type, it informs the client by sending the appropriate status code[4]. This procedure is called *Content Negotiation*.

[3] http://searchsoa.techtarget.com/definition/MIME
[4] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

2.2.5 Content Negotiation

Content negotiation is the process of choosing the most appropriate representation for a client, when there are manifold representations available by the server. Most of the times, content negotiation is associated with the practice of indicating media type preferences, however, it is also used to indicate preferences for different languages, character encoding and compression types. HTTP specifies two types of content negotiation. These are server-driven negotiation, which uses request headers to select a variant, and agent-driven negotiation which uses a distinct URI for each variant [(8)]. The reason why content negotiation is useful is due to the fact that by asking for a specific representation instead of getting a default one, client decreases the possibility to break because of an unknown and unmanageable representation, since the default one that the server sends, may change.

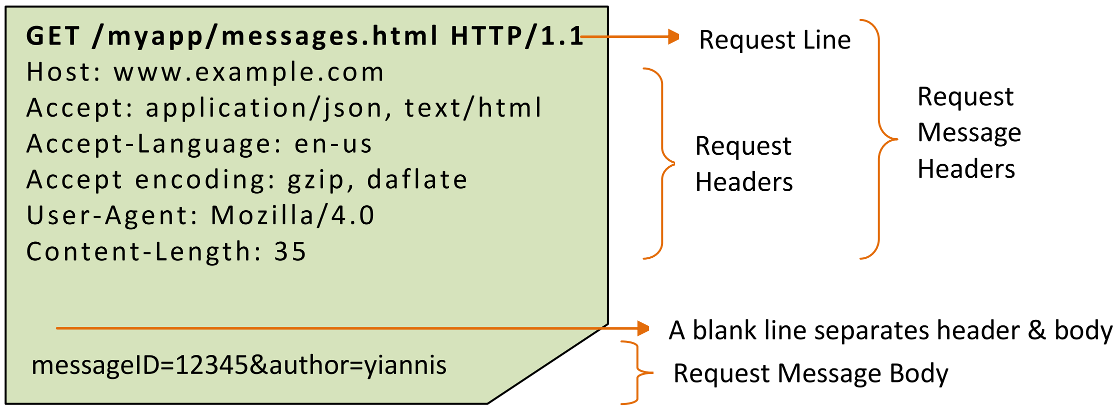
2.2.6 Uniform Interface

HTTP (Hypertext Transfer Protocol) is an application protocol for distributed, collaborative, hypermedia information systems [(9)]. It is the protocol that defines how messages are formatted and transmitted between a client and a server over the internet. REST take advantage of HTTP infrastructure by using its four basic verbs (methods):

- Retrieve a representation of a resource: *HTTP GET*
- Update an existing resource: *HTTP PUT*
- Create a new resource: *HTTP POST*
- Delete a resource: *HTTP DELETE*

In the REST world, these methods are called “operations”.

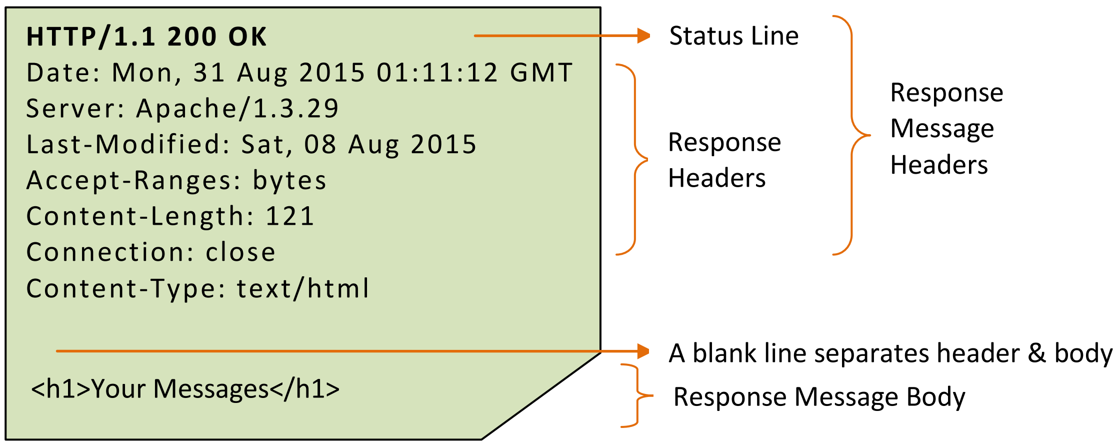
The syntax and the meaning of each operation do not change from application to application or from resource to resource. That is why HTTP is known as a *uniform interface* [(8)]. The following two figures illustrate how a client request message and a server response message look like.



```
GET /myapp/messages.html HTTP/1.1
Host: www.example.com
Accept: application/json, text/html
Accept-Language: en-us
Accept encoding: gzip, daflate
User-Agent: Mozilla/4.0
Content-Length: 35

messageID=12345&author=yiannis
```

Picture 2-3: HTTP GET Request

The first line of the request message describes the protocol, its version and the method used by the client. The next six lines are request headers, in a Key-Value form, that contain all necessary metadata for accessing the server successfully. By simply looking at these seven lines, any piece of software that understands HTTP (a web server, a proxy, etc) can decode not only the purpose of the request, but also how to parse the body of the message and send the appropriate response.



```
HTTP/1.1 200 OK
Date: Mon, 31 Aug 2015 01:11:12 GMT
Server: Apache/1.3.29
Last-Modified: Sat, 08 Aug 2015
Accept-Ranges: bytes
Content-Length: 121
Connection: close
Content-Type: text/html

<h1>Your Messages</h1>
```

Picture 2-4: HTTP GET Request

The first line of the response message denotes the HTTP version and the status code. *Status codes* are standard response codes given by servers in order to help the client to identify if there was an error or not when it tried to parse the request, and if there was an error, to give an indication about it. Status codes are split into five categories. Table 2-2 [(10)] shows these categories as well as the most important status code definitions. The remaining lines of the message, give information about the server and the sent message. In its body is included the requested representation.

Code	Definition
1xx	**Informational Codes**
100	CONTINUE – the client should continue with request
101	SWITCHING PROTOCOLS - the server will switch protocols as necessary
2xx	**Success Codes**
200	OK - the request was fulfilled
201	CREATED - following a *POST* command
202	ACCEPTED - accepted for processing, but processing is not completed
204	NO CONTENT - request received but no information exists to send back
3xx	**Redirection Codes**
301	MOVED PERMANENTLY - the data requested has a new location and the change is permanent
302	FOUND - the data requested has a different URL temporarily
304	NOT MODIFIED - the document has not been modified as expected
305	USE PROXY - The requested resource must be accessed through the specified proxy
307	TEMPORARY REDIRECT - the requested data resides temporarily at a new location
4xx	**Client Error Codes**
400	BAD REQUEST - syntax problem in the request or it could not be satisfied
401	UNAUTHORIZED - the client is not authorized to access data
402	PAYMENT REQUIRED - indicates a charging scheme is in effect
403	FORBIDDEN - access not required even with authorization
404	NOT FOUND - server could not find the given resource
415	UNSUPPORTED MEDIA TYPE - requested resource format is not supported
5xx	**Server Error Codes**
500	INTERNAL ERROR - the server could not fulfill the request because of an unexpected condition
501	NOT IMPLEMENTED - the sever does not support the facility requested
505	HTTP VERSION NOT SUPPORTED

Table 2: Status Codes Definitions

2.2.7 Idempotence

One way to classify the four HTTP methods could be by separating them between Read-Only methods and Write-Only methods (write something to the server). In such a way, the GET method would be in the first category, whilst POST, PUT and DELETE methods would be in the second.

Another way to categorize these methods is between *Idempotent* and *Non-Idempotent*. According to dictionary[5], idempotent is an operation that produces the same results, no matter how many times it is performed. In mathematics, an idempotent operation is one where f(f(x)) = f(x).For example, the abs() function is idempotent because abs(abs(x)) = abs(x) for all x. However, in computer science, an idempotent operation is the one that has no additional effect if it is called more than once with the same input parameters [(11)]. In other words, it is safe to make multiple repeated calls without having to worry about the impact on the server side. A client can make a GET request to the server several times, and every time it will get the same response. If a client sends a DELETE request to a specific resource to the server, that resource will be removed. If it sends the exact same request, the resource is still gone. If a client wants to update a resource, it should send a PUT request to the server, and the body of the request must contain the changes it wants to apply. It can resend the PUT request, and the resource state will not change again. On the other hand, if a client wants to create a new resource, it should send a POST request with the appropriate body, and the server will respond with a message that will contain, among others, the resource-ID for the resource that has just created, providing the way for that resource to be called later on. This time, however, if the client sends again the same POST request, the server will create a completely new resource with the same content.

Taking all these into account, we can safely assert that GET, PUT and DELETE are idempotent operations, whilst POST is Non-Idempotent. The reason why idempotence matters is because it lets a client make reliable HTTP requests over an unreliable network. For instance, if a client makes a GET request and something goes wrong, it is safe to make another one. If you make a PUT request and never get a response, just make another one. Even though your earlier request reached the server, a second request will have no additional effect [(6)]. As far as POST request is concerned, since it is Non-

[5] http://encyclopedia2.thefreedictionary.com

Idempotent, it must be used with caution, because unwanted side effects could be occurred on the server side.

It is highly recommended that each of the methods be used depending on the result you want to achieve. For example, a GET request to a URI like http://www.example.com/myapp/message/13/delete, is a bad practice. In that way, you are not fetching the resource, but rather modifying it. Moreover, it is easier for other developers to use your RESTful API. In particular, use GET for safe and idempotent information retrieval, POST for creating new resource, PUT for updating an existing resource or creating a new one in the case that the client is able to decide the resource's URI, and finally use DELETE for erasing a resource.

Yet, a confusing topic among developers is the difference between PUT and POST. Hence, it is worth mentioning their small declination. The key difference between them is that PUT is idempotent whereas POST is not. That is, POST should be used with more care. The second difference is that when you use PUT, you must always specify the complete URI of the resource, while with POST you have no such restriction. For example, if you want to update an existed resource, both these methods can do the job:

PUT *http://www.example.com/myapp/messages/13*

POST *http://www.example.com/myapp/messages/13*

Mistakenly, many developers think that REST does not allow POST to be used for updating a resource. On the other hand, the following PUT request would not work, if a client wanted to create a totally new resource, since PUT requires the complete URI.

PUT *http://www.example.com/myapp/messages/*

In such a case, use POST instead:

POST *http://www.example.com/myapp/messages/*

2.2.8 RESTful Web API Data Formats

There are several data formats provided by a RESTful Web API, such as HTML, XML, JSON, Text, CSV, JPEG. However, by far the most popular ones are XML and JSON, thus we will deal only with them, defining their serviceableness as well as their strengths and weaknesses.

According to W3C, *XML* (E**x**tensible **M**arkup **L**anguage) is a markup language that defines a set of rules for encoding documents in a format which is both human and machine-readable. It was designed to describe and exchange data, not to display it, and its tags are not predefined (you must define your own tags). It gained a lot of attention because it is technology independent.

JSON stands for **J**ava**S**cript **O**bject **N**otation and is a syntax for storing and exchanging data. Due to its lightweight format, it is considered by many developers and organizations a good alternative to XML. The common characteristics that both share are:

- Both are self-described, meaning that their values, tags and elements have names, which help human readability.
- Both are hierarchical, meaning that you can have nested values.
- Both can be parsed and used by almost every programming language.
- Both are technology independent.

As it is already noted, both XML and JSON are used in web APIs. Nevertheless, it is an open secret that JSON fits better in the new world of services. Among the developer community, JSON is growing its popularity, owing to its more compact and less verbose format. In general, XML is more complex than JSON. By default, XML requires complex features to be used along with it, like namespaces, schemas (for validation) and attributes. However, these features are not necessary for a successful web service. In the web service case, all that is requisite, is a straightforward data structure and a compact data exchange format that can be read and parsed fast, without having unnecessary overhead. To make it more obvious, let's give an example. Most of the times, a RESTful API client happens to be a browser. Technically speaking, a piece of JavaScript code which is running in the browser, which makes these REST API calls. Hence, this JavaScript client can easily convert these JSON responses to a JavaScript object, making the exchange and consuming of data effortless. It also deserves to be mentioned that companies like Twitter and Foursquare provide their API in JSON-only format.

2.2.9 HATEOAS

The acronym HATEOAS stands for **H**ypermedia **A**s **T**he **E**ngine **O**f **A**pplication **S**tate, and it is used in RESTful web services in order to make the API "discoverable". Simply put, hypertext it is used in order a client to find its way through the API, without any documentation, just like a website user does not need any documentation to discover locations and operations within it. The only thing that is needed is to go to the home page.

Briefly mention, the “things” that are transferred and exchanged in HTTP are called hypertext. Hypertext is a structured form of text, which has one interesting property. It contains logical links to other texts. These links are called hyperlinks, let us go from page to page and is the main reason why we can use a website effectively. This is the advantage of using HTTP. In a similar way, the best RESTful Web API is considered to be the one that needs no documentation in order to take advantage of its capabilities.

HATEOAS is a way to provide links to resources in the API response, so as the client does not have to deal with URI construction and the business flow. The hypermedia that is sent by the service as a response, drives the client interaction with the application state. Therefore, a REST client needs no prior knowledge about the URIs, which results in decoupling from the server.

The following example attempts to depict better the usefulness of HATEOAS[6] [(8)].

Here is a GET request to fetch an *account* resource, requesting details in a JSON representation:

```
GET /account/12345 HTTP/1.1
Host: somebank.gr
Accept: application/json
…….
…….
…….
…….
```

Picture 2-5: HTTP GET Request

Here is the response body which contains three links that the client probably wants to use later (make a deposit, a withdrawal or a transfer):

[6] http://restcookbook.com/Basics/hateoas/

```
{
   “Account_number”: “12345”,
   “balance”: 100.00,
    “links”: [
          {
           “href”: “http://somebank.gr/account/12345/deposit”,
            “rel”: “deposit”
          },
          {
            “href”: http://somebank.gr/account/12345/withdraw”,
            “rel”: “withdraw”
          },
          {
            “href”: “http://somebank.gr/account/12345/transfer”,
            “rel”: “transfer”
          }
          ]
}
```

Picture 2-6: HTTP Response Body in JSON

As it is shown, the response body not only has the account number and the corresponding balance, but also three linking URIs which the client could use in subsequent request.

- *href* attribute gives the complete URI that uniquely identifies the resource.
- *rel* describes the relationship of that link to this particular response message. That is, you add the “rel” attribute to make it clear what that link points to.

Even though the above response is in JSON, XML could be also used. HATEOAS does not make discriminations between them. Thus, for the sake of completeness, the response body is provided in XML format in the figure below:

```
<account>
    <account_number>12345</account_number>
    <balance>100.00</balance>
    <link rel=”deposit” href=”http://somebank.gr/account/12345/deposit”/>
    <link rel=”withdraw” href=”http://somebank.gr/account/12345/withdraw”/>
    <link rel=”transfer” href=”http://somebank.gr/account/12345/transfer”/>
</account>
```

Picture 2-7: HTTP Response Body in XML

2.2.10 Richardson Maturity Model

Due to the fact that REST is not a standard but rather an architectural style, is it possible to be in the position to say that some web API is fully RESTful or not? As a matter of fact, this is not a “Yes” or a “No” question. It would be extremely helpful if there was an API spectrum from “fully RESTful”, to “almost RESTful”, to “no RESTful at all”. Actually, there is a way to categorize a web API, and the way is a model developed by Leonard Richardson called *The Richardson Maturity Model*.

The Richardson Maturity Model is a way to grade your API according to some rules. The more strictly your API follows these rules, the higher its score is. The model has four levels (0-3), where level 3 designates a truly RESTful API, while level 0 defines a not RESTful API at all [(12)]. Consequently, every REST API belongs to one of the three levels (level 1, 2 or 3).

Level 0: The Swamp of POX

Level 0 uses its implementing protocol, which is normally HTTP, as a transport protocol [(13)]. Also, there is a single URL, which is called the *endpoint*, where the web service is exposed. That URL receives all the requests from the client. The way that the web service savvy what to do and what operations to perform (GET a resource, DELETE a resource, etc) is through the message that is sent by the client to that URL. The message, meaning the request body, contains all the operations that must be performed as well as the data that is needed for that operation.

Since the action is a part of the message itself, the common URL works perfectly. In fact, the same HTTP method can be used for each operation for the reason that all the details about the operation is in the request body. A typical example of this kind of web service is SOAP. Usually, the HTTP verb that is used is POST.

Level 0 is also called *the swamp of POX*. It refers to the common use of Plain Old XML for defining everything that the operation needs. Everything is defined in an XML. Thus, no HTTP concepts are necessary for the client-server communication.

Level 1: Resources

If you want to refine the previous model by introducing the concept of Resource URI, you will reach level 1 in the Richardson Maturity Model. This is the starting level for RESTful APIs. This level uses individual URIs, where each URI is the endpoint for one resource. Nonetheless, the information about the operation must still be inside the request because a single message URI needs to handle operations like adding a message, updating a message, deleting a message and so on. A single profile URI has to deal with all these operations. Hence, the request still contains what is necessary to be done. Basically, level 1 assigns different URISs to different resources, and sends the right request to the right resource URI.

Level 2: HTTP Verbs

The next step is to introduce different HTTP methods for different operations. An API in this case uses standard HTTP verbs, like GET, PUT, POST and DELETE in order to achieve different results upon the resource URI. In other words, the URI should specify the resource that is being operated upon, while HTTP methods determine what the operation is. Furthermore, it is highly important a better and more careful use of HTTP status codes. For instance, if a client's request is "delete that resource" and the request was successfully processed, is a better tactic, the subsequent response of the service to be *HTTP 204 No Content* and not just *HTTP 200 OK.* This is a convention that several developers usually follow. On the other hand, it would be detrimental if the web service sent *HTTP 200 OK,* assuming that something wrong happened on the server side during the processing. Given that these constraints are strictly followed, the API can be ranked as level 2.

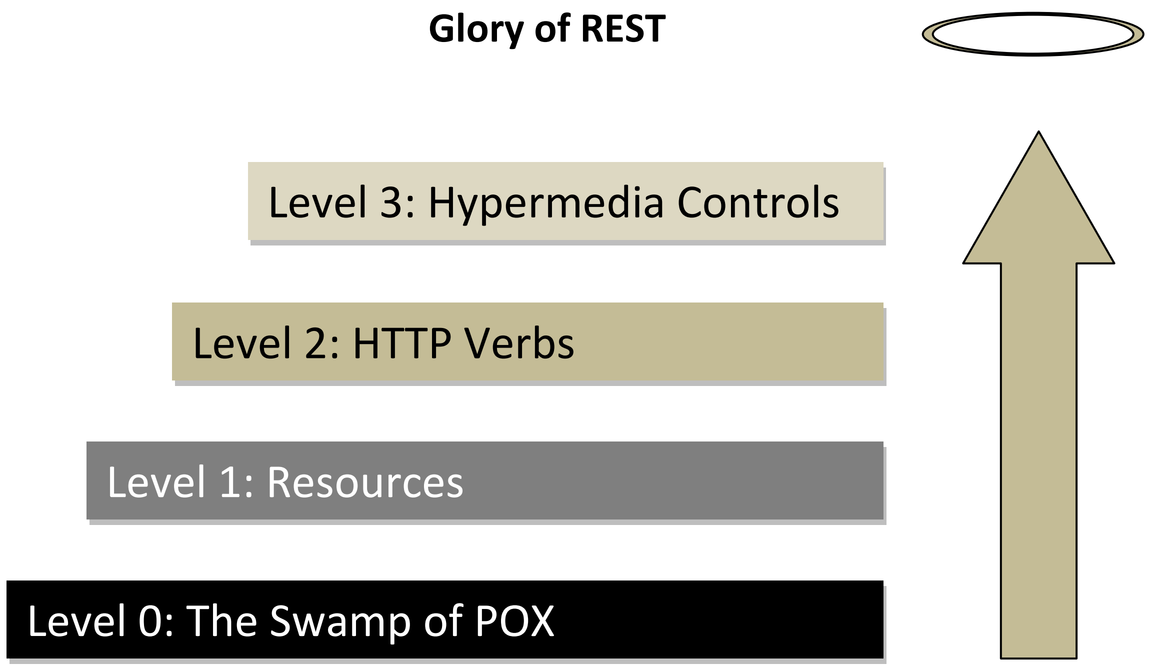
Level 3: Hypermedia Controls

Finally, level 3, the highest level, is when you implement HATEOAS. That is, the responses have links that drives the application state for the client. Consequently, the client does not have to be aware of the different API URIs. Everything that might be of interest to them, is sent in the response. When a web API is reached this level, is considered to be fully RESTful.

To conclude, the Richardson Maturity Model provides a good step by step way to perceive the basic ideas behind RESTful concepts. In essence, it is a guideline for designing RESTful APIs that helps us consider the kind of HTTP service we want to provide. Ultimately, we briefly state the topmost usefulness of each level [(13)]:

- Level 1 deals with the complexity of the one single endpoint by breaking it down into multiple resources.
- Level 2 implements a standard set of HTTP methods in order for similar operations to be handled in similar way, removing unnecessary variation.

- Level 3 enables a client to discover all the potentials of the API.



Picture 2-8: Steps towards REST

2.2.11 REST Security

As expected, since a RESTful API is a web-based application, security issues arise. Securing the system includes things like [(8)]:

- Ensure that only authenticated users have access to resources.
- Warrant the confidentiality and integrity of data, right from the moment it is collected until the time it is stored and later given to authorized users.
- Forbid unauthorized or malicious clients from corrupting resources.
- Maintain privacy.

First of all, it would be useful to briefly mention and clear a bit the terms Authentication, Authorization, and Encryption [(14)]:

Authentication answers the question “Who are you?” It is used by the server when it needs to know exactly who is accessing its information. The client then must prove its identity, usually by providing a username and a password. Authentication does not define the tasks that the client can do or the files it can access.

Authorization answers the question "What resources are you authorized to access?" It is the process by which a service resolves if the client has permission to use the requested resource. Authorization occurs after successful authentication. Usually, it is controlled at file system level.

Encryption involves the process of converting data into a cipher to prevent unauthorized access. Protocols such as SSH and TLS are usually used in transferring data between a client and a server. The data is encrypted before the data transmission.

In the beginning, basic authentication involves the client exchanging with the server a token to authenticate itself. As soon as the client sends a request to access a protected resource, the server returns status code "401 Unauthorized", which essentially means "Unauthenticated", along with a WWW-Authenticate header. Then, the client must send its credentials within the Authorization header. An example can better illustrate this process [(15)]:

```
GET /payment/1234 HTTP/1.1
Host: example.gr
…..
…..
…
```

Picture 2-9: Request for Accessing a Resource

```
401 Unauthorized
WWW-Authenticate: Basic realm="payments@example.gr"
…..
…..
…
```

Picture 2-10: Response Challenges the Client

This response challenges the client by asking him to give a Basic digest in order to access the resource in the realm "payments@example.gr". If the client knows the credentials for this realm, it hashes them with base64 encode, embeds them within the Authorization header and resends the request, as shown in the next figure.

```
GET /payment/1234 HTTP/1.1
Host: example.gr
Authorization: Basic ZuMjyvYXV0aA==
…..
…..
…..
```

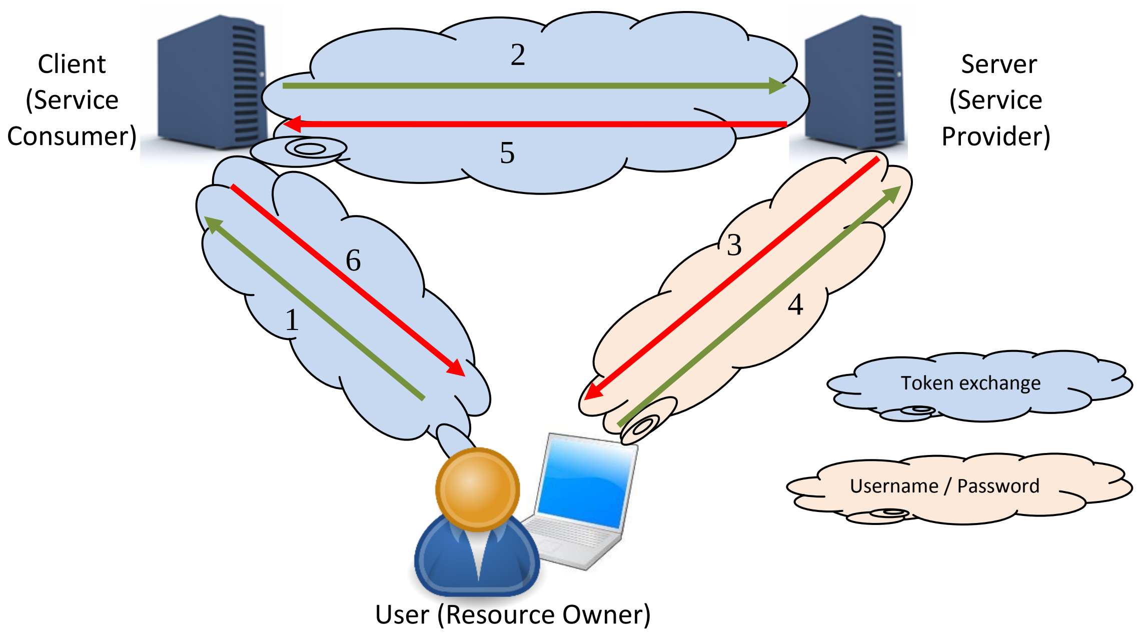
Picture 2-11 : Attempted Authorized Access to a Payment Resource

Certainly, if the client knows beforehand that the server requires Basic Authentication for a resource, it can originally provide the credentials along with the main message.

The problem with HTTP Basic authentication is that is not secure. Even though the credentials are not sent in plain text, Base64 encoded can be easily decoded. That is why Basic authentication should never be used without the TLS protocol. TLS has three phases (Handshake, Secure session, Channel setup), each of which must be completed before we can transfer representations via HTTPS [(15)]. After that, it is considered safe to transmit HTTP requests and responses.

Once the client has been verified, the service provider decides which operations are allowed. As it is already stated, this is called authorization. Authorization is often based on a username and password combination. After logging in, the system grants access to some of the functions and data. This has worked relatively well because usernames and passwords are often managed centrally in directory services. Nevertheless, it is not always possible or desirable to centralize and share credentials in a traditional way. When third parties provide services to another service provider, for instance, distributing usernames is normally undesirable and impractical. This is where OAuth comes in [(15)]. OAuth is an authorization protocol that allows you to approve one application interacting with another on your behalf without revealing your credentials. However, it could be also used for authentication. OAuth is ideal when a REST client do not own the data that he is trying to read. Although it is not perfect yet, OAuth is a huge improvement over HTTP Basic Authentication.

By illustrating the following figure we give an overview of how OAuth works. The steps for accomplishing the client authorization is attempted to be described in plain language[7].



Picture 2-12: OAuth Overview

1. The first step is that the user (the resource owner), asks the client (the service consumer) to carry out a certain task.
2. Then, the client tells the server (service provider) that a user wants to give him authorization for a specific task, and the server replies by passing him a token and a secret. A token is a unique string that plays the role of a username and a password, while the secret is used to prevent request forgery. The client uses the secret to sign each request so that the server can verify that is actually a legitimate party the one that asks for authorization.
3. After that, the client gives the token to the user and the user tells the server to authorize this request token. The server in turn, asks again the user if he sure about this authorization.
4. The resource owner responses positively and authenticates the client. Now the client has permission to use this access token. In essence, the service provider marks the request token as a valid one, so when the client requests access, it will be accepted (as long as it is signed with the shared secret).
5. At this point, consumer is authorized and asks the service provider to exchange the request token for an access token. The server passes one to him.

[7] http://blog.varonis.com/introduction-to-oauth/

6. Finally, the consumer has access to the protected resource and can send requests on client's behalf.

More technically speaking, the steps are as follows [(8)]:

The client needs the user's authorization in order to be able to accomplish a task. Therefore, he approaches the server to obtain an oauth_consumer_key and a secret. Most of the times, this is done manually, meaning that the service provider provides a web page for clients to register and obtain the consumer key and consumer secret. Moreover, the server must document the following URIs:

- A URI to obtain request tokens, for example: *https://www.example.gr/oauth/request_token*
- A URI to obtain a user's authorization, for example: *https://www.example.gr/oauth/authorize*
- A URI to obtain an access token, for example: *https://www.example.gr/oauth/access_token*

These requests involve the following parameters:

- *oauth_consumer_key:* This is the unique identifier issued by the server to each client.
- *oauth_signature_method*: This is the signing method used when computing a signature.
- *oauth_timestamp*: This is the number of seconds since January 1, 1970, 00:00:00 GMT.
- *oauth_nonce*: This is a random string parameter that helps servers prevent replay attacks
- *oauth_version*: This is the OAuth version

The first step for the client is to send a request to obtain a request token and a secret from the server. The signature in this request is based on the consumer secret that the client obtained along with the consumer key. The signature includes oauth_consumer_key, oauth_signature_method, oauth_timestamp, oauth_nonce, and oauth_version.

Using this signature, the client submits a request to the server to obtain a request token:

```
POST /request_token HTTP/1.1
Host: www.example.gr
Authorization: OAuth realm="http://www.example.gr/protected_resource",
            oauth_consumer_key=a1191fd420e0164c2f9aeac32ed35d23,
            oauth_nonce=109843dea839120a,
                        oauth_signature=d8e19bb988110380a72f6ca33b2ba5903272fe1,
            oauth_signature_method=HMAC-SHA1,
                        oauth_timestamp=1258308730,
                        oauth_version=1.0
Content-Length: 0
………
……….
……….
```

Picture 2-13: Request for Obtaining a Request Token

The response from the server contains a request token and a secret:

```
HTTP/1.1 200 OK
Content-Type: application/x-www-form-urlencoded

oauth_token=0e713d524f290676de8aff4073b1bb52e37f065c
            &oauth_token_secret=394bc633d4c93f79aa0539fd554937760f05987

……….
……….
……….
```

Picture 2-14: Response Containing Request Token and Secret

The *oauth_token* in this response is a request token that the client must use in order to get the user’s permission. The client then redirects the user to visit a resource URI on the server in order to grant authorization:

```
GET /oauth/authorize?oauth_token=0e713d524f290676de8aff4073b1bb52e37f065c HTTP/1.1
Host: www.example.gr
……….
……….
……….
```

Picture 2-15: Request for Obtaining Authorization

Due to the fact that OAuth allows granular access levels, the server might let the user select the resources that the client will be able to access. After that, the server directs the user back to the client so as the client to obtain the verification code provided by the server. The client then uses this code to acquire the access token by sending a new request to the server, including the appropriate Authentication header:

```
POST /access_token HTTP/1.1
Host: www.example.gr
Authorization: OAuth oauth_consumer_key="a1191fd420e0164c2f9aeac32ed35d23",
                                oauth_token="ad0d1c7a765c9e6e8b14e639c763177312d18e7e",
                                oauth_verifier="988786765423",
                                oauth_signature_method="RSA-SHA1",
                                oauth_signature="698d58fd3316304181e11c6eb8127ffea7e2df46",
                                oauth_timestamp="1258328458",
                                oauth_nonce="109843dea839120a",
                                oauth_version="1.0"
Content-Length: 0
…………
…………
```

Picture 2-16: Request for Obtaining an Access Token and a Secret

The response contains access token and token secret:

```
HTTP/1.1 200 OK
Content-Type: application/x-www-form-urlencoded
oauth_token=8d743f1165c7030177040ec70f16df8bc6f415c7
                     &oauth_token_secret=95aec3132c167ec2df818770dfbdbd0a8b2e105e
……….
……….
……….
```

Picture 2-17: Response Containing the Access Token and Secret

Finally, the client uses these credentials to build an Authorization header whenever he sends a request to access protected resources for that user.

2.2.12 REST Documentation

RESTful web services do not necessarily require documentation to help clients discover them. In fact, for some developers the best RESTful APIs are considered those that have no documentation at all. That is, the client must simply know a basic endpoint to the service and from there he could discover the service on his own by traversing the resources using links. However, in my point of view, there is no reason not to documenting your service. That way, you maximize the efficiency and the user experience of your service, while at the same time you minimize the confusion about what calls do what. Normally, the documentation for a REST API is not lengthy, since HTTP specification reveals how most of your API works.

Parts of the REST API that should be described in a human-readable manner (HTML format) are [(8)]:

- All resources and methods supported for each resource.
- Media types and representation formats for resources in requests and responses.
- The link relation that is used, its significance, HTTP method to be used, and resource that the link identifies.
- Query parameters used for all fixed URIs
- Authentication and security credentials for accessing protected resources

The table below is an example documentation of a RESTful API called "Messenger", which could be used in a social media application:

Service name: messenger

Root URI: http://www.example.gr/messenger/

Resource	Methods	URI	Description
Users	GET,POST,PUT, DELETE	http://www.example.gr/messenger/users/{UserID}	Contains information about a user {UserID} is optional Format: application/json Query Parameters: filters the results ……… ………
Messages	GET,POST,PUT, DELETE	http://www.example.gr/messenger/messages/{MessageID}	……. …….
Comments	GET,POST	http://www.example.gr/messenger/comments/{CommentID}	……. …….
Likes	GET,POST, DELETE	http://www.example.gr/messenger/likes/{LikeID}	……. …….

Table 3: RESTful API Documentation

2.3 SOAP vs REST

SOAP and REST can both be used to answer the question: “How can I build or access a web service?” Of course which of those answers this question the best, is not obvious. There are two sides of argument. One is that REST is better, and the other that SOAP is better. Most of the times, however, you need to choose one over the other depending on what you want to achieve.

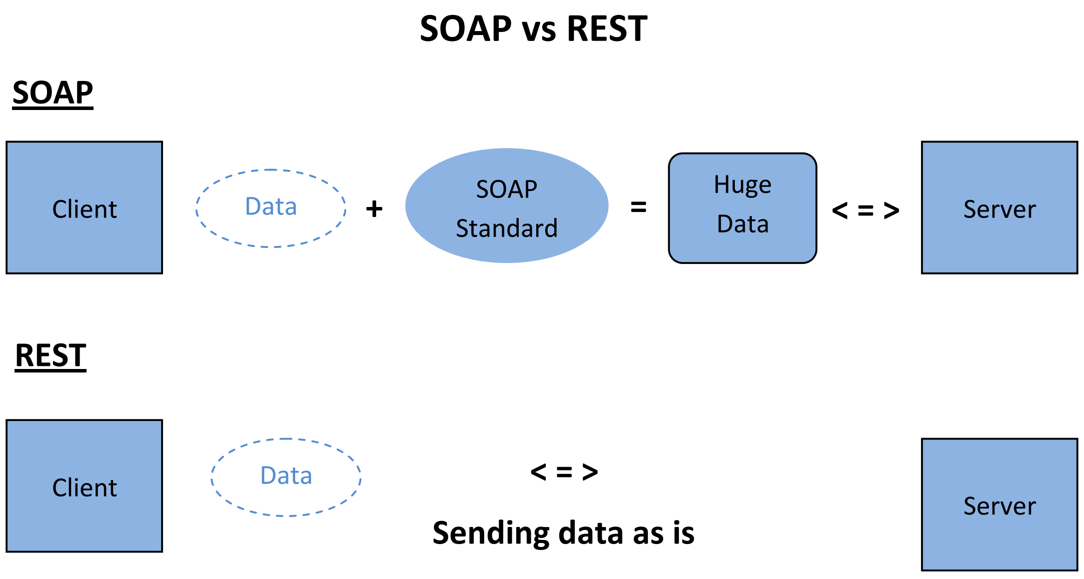
SOAP is better than REST:

SOAP is standardized. This standardization is convenient when a formal contract must be established to describe the interface that the web service offers. WSDL describes details such as messages, operations, bindings, and location of the web service. Furthermore, due to the fact that REST relies on HTTP and HTTP is synchronous, scalability issues are raised in REST because in order to scale, it is needed to have asynchronous messaging. On the other hand, SOAP is well suited when the application architecture needs to handle asynchronous processing and invocation, resulting in better scalability [(16)]. Over and above, REST is not reliable. Every messaging failure must be handled with retries, whereas SOAP has built-in error handling mechanisms.

REST is better than SOAP:

REST is lightweight, meaning that it does not require XML parsing. As a consequence, it fits better in technologies like mobile and IoT. Moreover, it consumes less bandwidth as it does not require a SOAP envelope for every message that goes to and from the service provider. In other words, the amount of data that is sent between a consumer and a producer is far less than in SOAP [(16)]. In addition, REST uses the underlying, already known technology of HTTP for transport, communication and security between clients and servers, which results in a shorter learning curve. It is the new trend and is not a coincidence that companies like Google, Twitter, Facebook and Yahoo use REST for exposing their web services.

To conclude, a rule of thumb is to use SOAP whenever you are publishing a complex API, whilst use REST when lightweight and simple transactions are needed. The following figure illustrates the overhead that SOAP carries compared to REST.



Picture 2-18:SOAP vs REST

3 Problem Definition

Auction is an economic process whose purpose is the selling or purchase of goods and services via a procedure known as bidding. Depending on the nature of the items that are for sale, as well as the way that the biddings are held, different auction structures might be either more efficient or more profitable to the seller in comparison with others.

3.1 Auction

There is a whole theory behind auctions, called "Auction Theory", which is an offshoot of economics which deals with how people should act in auction markets, how efficient an auction design is, and generally how to maximize your revenues. William Vickrey, who was a Canadian-born professor of economics and Nobel Laureate, first established the taxonomy of auctions based on the order in which the auctioneer quotes prices and the bidders tender their bids.[8] He established the four basic types of auctions that are widely used, even today. The distinction between these types of auctions comes from several different criteria, such as the action of the bidders and the structure of the auction itself. For instance, ascending or descending price auctions, auctions for single objects or for multiple objects, open auctions or sealed-bid auctions, etc. Analytically, the four types are:

1. *Ascending-bid auction*, also called *English auction*: This type of auction is arguably the most common. This kind of auction is carried out interactively in real time, with bidders present either physically or electronically. The seller gradually raises the price, while the other bidders drop out one by one [(17)].The auction stops when no other participant is willing to increase the previous highest bid, at which point the highest bidder pays their bid [(18)]. This type of auction is commonly used in the sales of rare paintings, used cars and houses.
2. *Descending-bid auctions*, also called *Dutch auctions*: This type of auction works in the exactly opposite way: the auctioneer starts at a very high price, and then lowers the price continuously. The first bidder who calls out that he accepts the

[8] http://www.nobelprize.org/nobel_prizes/economic-sciences/laureates/1996/vickrey-bio.html

current price, wins the object at that price [(19)]. These auctions took their name because flowers have long been sold in the Netherlands using this procedure. It is also used for perishable goods, such as fish and tobacco.

3. *First-price sealed-bid auctions* or *blind auction*: In this kind of auction, bidders submit simultaneous "sealed bids" to the seller. In other words, each bidder independently submits a single bid, without seeing others' bids. The terminology comes from the original format for such auctions, in which bids were written down and provided in sealed envelopes to the seller, who would then open them all together. The highest bidder wins the object and pays the value of his bid [(17)].
4. *Second-price sealed-bid auctions*, also called *Vickrey auctions*: This is identical to the sealed first-price auction except that the highest bidder wins the object, and pays the value of the second-highest bid. These auctions are commonly used in automated contexts, such as real-time bidding for online advertising.

3.1.1 When are Auctions Convenient?

Besides the four primary types of auctions that were mentioned above, there are many more variations. Actually, the range of auctions that exist is extremely wide and they can be used by many different people and organizations for buying almost anything. Briefly, we can state some auction types, such as car auctions, land and property auctions, antique and collectible auctions, title and insurance auctions, wine auctions.

Auctions are generally used by sellers in situations where they do not have a good estimate of the buyers' true values for an item, and where buyers do not know each other's values. In this case, some of the main auction formats can be used to elicit bids from buyers that reveal these values [(17)].

Known Values: Let's assume the case in which the seller and buyers know each other's values for an item. For instance, suppose that the seller is trying to sell the item which he estimates at x, and the potential buyer of the item estimates it in some larger number y. In this case, it can be said that there is a surplus of $y - x$ that can be generated by the sale of the item: it can go from someone who values it less (x) to someone who values it more (y). Should the seller knows the true values that the potential buyers define to the item, then he can just notify that the item is for sale at a fixed price slightly below y. Thus, the buyer with value y will buy the item, and the full value of the surplus will go to the seller. In other words, the seller has no need for an auction in this case. It must be stressed that in this example, the seller controls the mechanism of selling the item, which is beneficial to him. The reason why is because if the buyer believes this commitment, the item is going to be sold for a price just below *y*, thus the seller will reap almost all the surplus. On the other hand, if we gave the buyer the ability to control the mechanism with maximum value *y*, he could announce that he is willing to purchase the item for a price just above the larger of *x*. With this statement, the seller would still be willing to sell since the price would be above x, however, this time the largest percentage of the surplus would go to the buyer. In both cases, commitment by the seller or the buyer requires knowledge of everyone else's values. Furthermore, it is obvious that the commitment to a mechanism can shift the power in the transaction in favor of the seller or the buyer [(17)].

Unknown Values: In this case, buyers have independent and private values for the item. That is, each buyer knows how much he esteems the item, but he does not know how much the competitors value it, and his appraisal for it does not depend on others' value. Moreover, we have the case of *common values.* Assuming that the auctioned item is not intended to be consumed by the direct buyer, but it is planned to be resold, if he gets it. The item has an unknown but common value regardless of who gets it. This value is equal to how much revenue this future reselling of the item will generate. In this setting, the value each buyer specifies to the item, would be affected by the knowledge of the other buyers' valuations, since the buyers could use this knowledge to further refine their estimations of the common value [(17)].

3.1.2 Relationships between Different Auction Formats

As expected, bidders behave differently in interactive auctions, such as ascending-bid and descending-bid auctions, which take place in real time, and in sealed-bid auctions. The relationship between these different auctions are mentioned in the following two paragraphs.

Descending-Bid and *First-Price Auctions*: In the Descending-Bid auction, the seller is gradually decreasing the price from its high initial starting point. No participant reveals his intentions until finally someone accepts the bid and pays the current price. Consequently, participants know nothing while the auction is running. Hence, in that sense, the procedure is similar with the sealed-bid first-price auction, in which the item goes to the bidder with the highest bid value, and he pays the value of his bid in return for the item [(17)].

Ascending-Bid and *Second-Price Auctions*: In the Ascending-Bid auction, bidders progressively drop out as the seller steadily raises the price. The winner of the auction is the last bidder remaining, and he pays the value of his bid in return for the item. Nonetheless, two remarks should be pointed. Firstly, it makes no sense to stay in the auction after the price reaches or exceeds your defined true value, and secondly it also makes no sense to drop out before the price reaches the true value for that item. Thus, with these two simple indicators, a bidder knows when it is the right time to leave the auction. Yet, the rule for determining the outcome of an Ascending-Bid auction could be slightly changed as follows: The participant who made the highest bid is the one who stays in the longest, resulting in winning the item, but he pays the price at which the second-to last person dropped out. Thus, the item goes to the highest bidder at a price equal to the second-highest bid. This is exactly the rule that is used in the Sealed-Bid Second-Price auction, with the exception that the Ascending-Bid auction involves real-time interaction between the buyers and seller, whilst the Sealed-Bid version takes place through sealed bids that the seller opens and assesses [(17)].

3.1.3 Auction Terminology

Appraisal: An estimate of an item's worth, usually performed by an expert in that particular field

Auction House: The Company that operates the auction

Auction Fever: An emotional state elicited in the course of one or more auctions that causes a bidder to deviate from an initially chosen bidding strategy [(20)].

Bidding: The act of participating in an auction by offering to purchase an item for sale.

Buyout Price: A price that if accepted by a bidder, immediately ends the auction and awards the item to him/her.

Commission: A fee paid by a seller to the auction house; it is typically calculated as a percentage of the winning bid and deducted from the gross proceeds due to the seller.

Dummy Bid: A false bid, made by someone in collusion with the seller or auctioneer, in order to create a sense of increased interest in the item.

E-Bidding: Electronic bidding, whereby a person may make a bid without being physically present at an auction.

Lot: An item being sold.

Minimum Bid: The smallest bid that will be accepted.

Outbid: Prompt for bidding higher than another bidder.

Opening Bid: The first bid placed on a particular Lot.

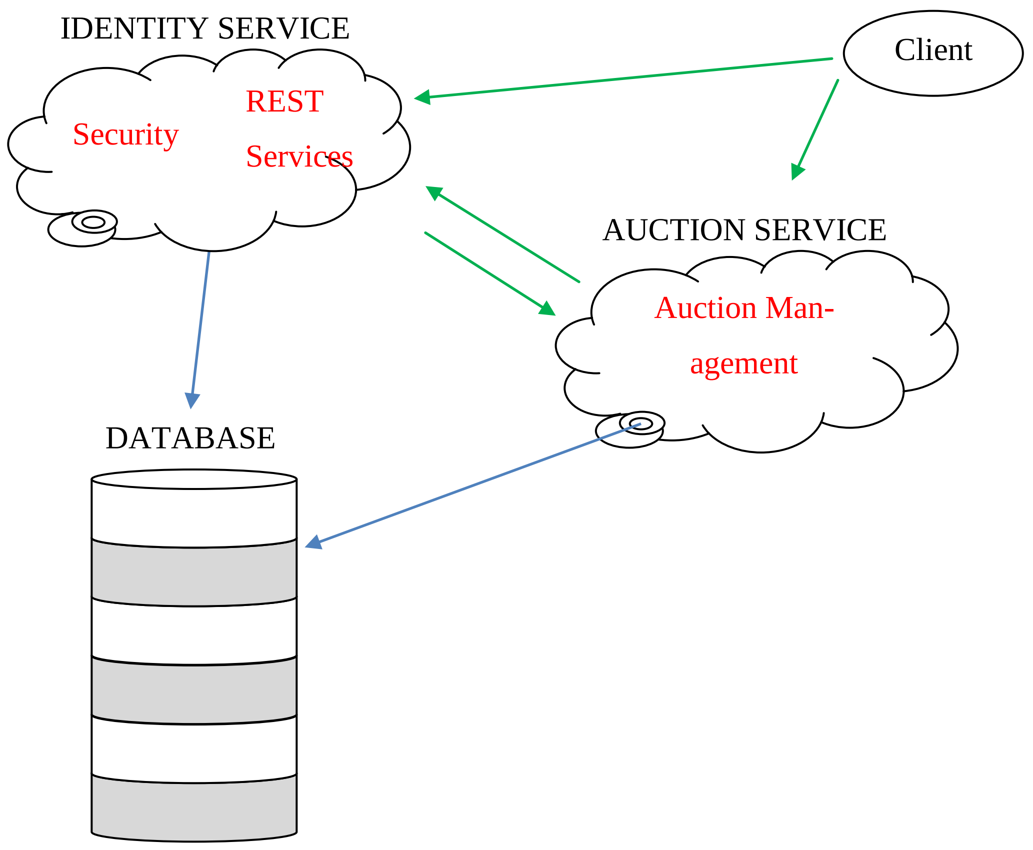
Proxy Bid: A bid placed by an authorized representative of a bidder who is not physically present at the auction.

Sealed bid: A submitted bid whose value is unknown to competitors.

Sniping: The act of placing a bid just before the end of a timed auction, thus giving other bidders no time to enter new bids.

3.2 Problem definition

In this thesis, we attempt to develop an online, ascending-bid auction system, which is supported by RESTful web services. The auction duration can be defined by the auctioneer, and when the auction closes, the highest bidder wins. The problem we try to tackle is to conceal participants' identity through a third party. It is common practice, especially with auction products of great value, that the involved partieswish to remain anonymous.The parties originally involved in an online auction are the auctioneer, the potential buyers of the auctioned product, and the auction service which coordinates the auction. Our contribution is the creation of a third party which provides the necessary identity services so as to cover the stakeholders' identity. That is why we could name this dissertation as *"3-Way Auction Service with 3rd Party Identity Service"*.



Picture 3-1: 3-Way Auction Service with 3rd Party Identity Service

As already mentioned in the introduction of the thesis, when an auction is won by a certain buyer, it is no longer requisite to reveal his identity due to the presence of this 3rd party, which will provide user authentication, postal and bank services. We briefly cite the scenario which was actually the motivation to design and develop our RESTful API.

The auctioneer opens and monitors the auction. As it is reasonable,he must already be authorized in the Identity Server, so as to identify himself to the Auction Service. The same will act the potential buyers. When the auction ends, Auction Server will send a request to Identity Server asking him to inform, both the winner of the auction and the seller of the product (auctioneer), about the auction outcome. As far as Identity Server is concerned, the notion of “auction” means nothing. The only thing he is aware of in this context, is to make a package transfer as well as to conduct a bank transaction. Hence, subsequently, Identity Service will withdraw money from the buyer’s account (winner of the auction), and put them in a “BLOCKED” state, which means that the money is not yet to the auctioneer’s account. No sooner has the package transfer been completed, than the money is deposited in his account. Only the third party (Identity) knows the actual address of the buyer and he will forward the package to him, while at the same time he will release the money for auctioneer’s behalf[9].

3.2.1 Why REST

The reason why we chose to use REST architecture for the implementation of our auction system is not only one, but rather many and varied. REST, except for the fact that is the new trend, it is also highly recommended when it is very important to minimize the coupling between the client and the server components, especially in a distributed application, like the one that we have tried to develop. This could be the case when the server is going to accept numerous requests. It might also be the case when the server is regularly updated, while the client software should remain intact.

Furthermore, following the REST architectural style, it is relatively easy to write web services, since the key concept is an interface that is already well known and widely used, namely the *URI*. Additionally, it is based on normal HTTP requests which enables intent to be inferred by the type of request being made. That is, GET for retrieving data, PUT for updating data, POST for writing new data and DELETE for removing data. All these concepts are already known, being used several years, and developers are familiar with them, resulting in a more standarized developing process, therefore, reduction of implementation time.

Another benefit of a RESTful API is that requests and responses can be short, thus, less overhead. This can be extremely useful in environments with limited bandwidth and resources. Also, any browser can be used, since REST is using HTTP verbs. On top of that, the messages can be in any user defined format, including JSON, which is much more light than XML, and works perfectly with JavaScript, an integral part of any modern browser.

As far as security is concerned, REST inherits security measures from the underlying transport layer. HTTPS secures the transmission of the message over the network and provides some assurance to the client about the identity of the server. In our case, since Identity Server provides Postal and Bank services, security is vital. In addition, through Basic Authentication, we provide extra security measures.

[9]Also see figure 1-3

Finally, due to the fact that RESTful APIs are stateless, communication messages can be cached leading to better performance and scalability. It is not a coincidence that REST has got lot of support from many companies. Some good examples are:

- Google - Google Maps, Google Adsense Search
- Yahoo! - All Yahoo API's Web Services, including Flickr
- Twitter
- Amazon
- Ebay

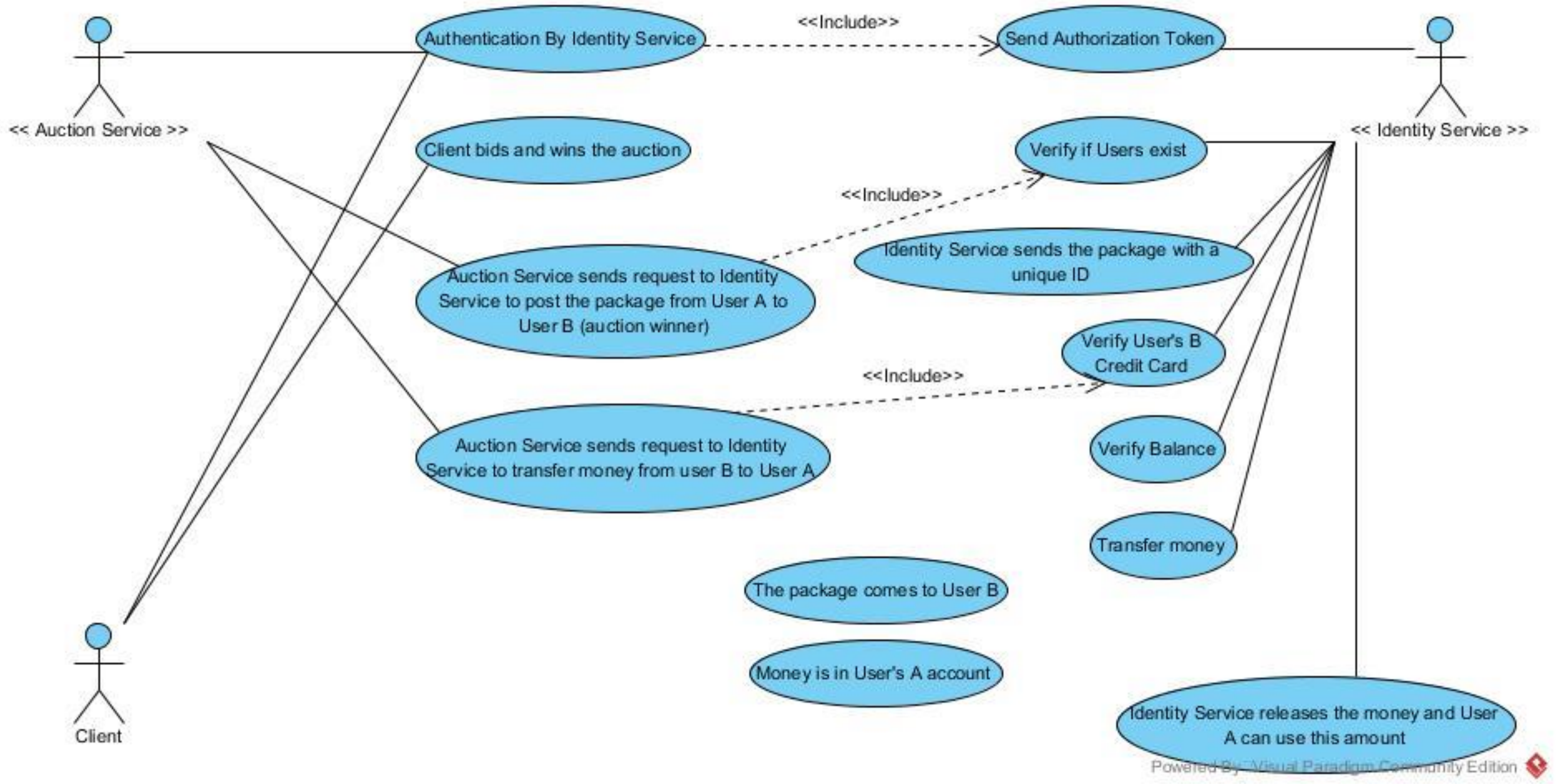
3.3 Use Case Diagram

A use case diagram is a visual representation of one or more use cases, depicting the actors that are involved, the relationship between the actors and the different use cases in which they participate. An actor, models a type of role played by an entity that interacts within the system. In other words, an actor is the agent who acts upon the use cases.Actors may represent roles played by human users or other systems. Use case diagram helps us determine the core functions of the system, from an external point of view.

To tackle any ambiguity, we will firstly present the use case of how an auctiontakes placeusing our REST API in plain text, and then as a diagram.

Use Case:	Auction participation
Overview:	A client bids for a product in order to win the auction
Actors:	Auction Server, Identity Server, Client
Pre-Conditions:	None
Main Success Scenario:	1. Auction Server and Clients must authenticate themselves in the Identity Service. 2. Identity Service sends to legitimate parties the appropriate authorization tokens. 3. Clients bid for a product until someone wins the auction. 4. Auction Server sends requst to Identity Server to post the won product from user A to user B (the winner of the auction). 5. Identity Server verifies that the users exist and send the package to user B with a unique ID (track number), in order for the package state to be tracked. 6. Auction Server sends request to Identity Server to transfer money from user B to user A. 7. Identity Server makes the necessary checks and then carries out the transaction. 8. The package comes to user B. 9. Money is in user’s A account. 10. Identity Server releases the money and user A has access to this amount.

Table 4: Use case



<< Auction Service >>
Client
<< Identity Service >>
Authentication By Identity Service
<<Include>>
Send Authorization Token
Client bids and wins the auction
Verify if Users exist
<<Include>>
Identity Service sends the package with a unique ID
Auction Service sends request to Identity Service to post the package from User A to User B (auction winner)
Verify User's B Credit Card
<<Include>>
Auction Service sends request to Identity Service to transfer money from user B to User A
Verify Balance
Transfer money
The package comes to User B
Money is in User's A account
Identity Service releases the money and User A can use this amount
Powered By: Visual Paradigm Community Edition

Picture 3-2: Use Case Diagram

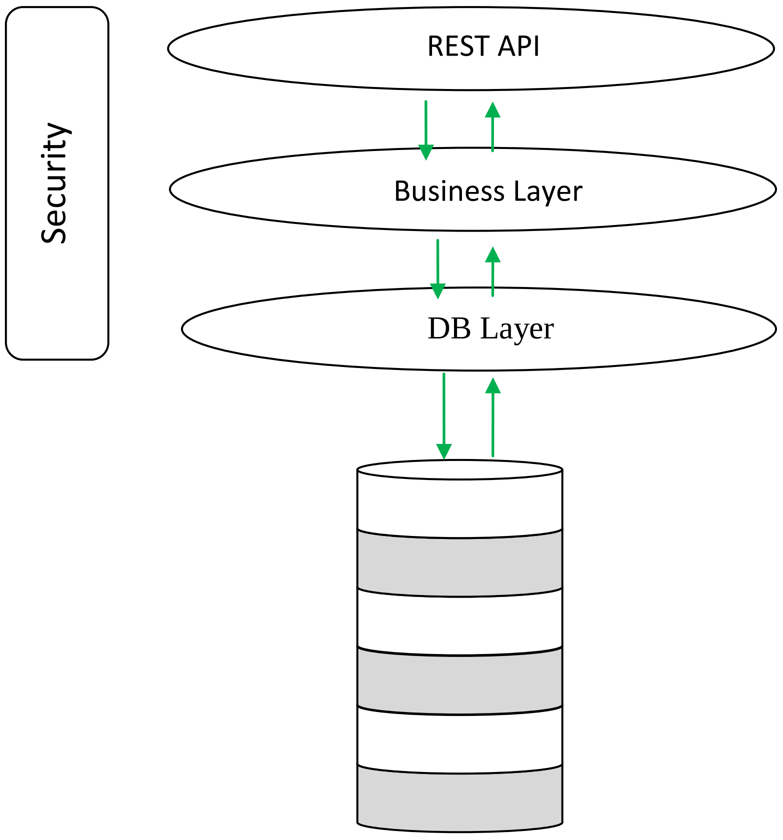
4 Implementation (Identity Server)

Based on the design considerations and the requirements of the previous chapters, this chapter presents a Java-based implementation of an auction RESTful API, supporting the two mainly media types, namely JSON and XML.

4.1 Identity Server

The solution is based on the JAX-RS implementation of Jersey. Jersey framework is an open source, production quality framework for developing RESTful web services in Java that provides support for JAX-RS APIs and serves as a JAX-RS Reference Implementation.[10] The core of this API is the *Identity Server,* which offers services such as Authentication, Postal and Bank services. For highest efficacy, robustness, clarity and maintenance, we designed and developed the whole project into layers. Each of the layer contains the same type of logic. Layering reduces the conceptual overhead, since services belonging to the same layer, address a smaller set of activities, thus, minimizes the impact of potential changes in one of the layer. In particular, we have developed three main layers. The first one is the front end layer API which communicates directly with the client. The second layer is the business layer which contains the business processing logic, such as how a user must be created or how security should be implemented, and the third one is the database layer which provides the database connectivity. This layer is responsible for exposing the data stored in the database to the business layer. This means that there is no directly communication between the front end layer and the database. The following figure illustrates all the layers.

[10]http://jersey.java.net

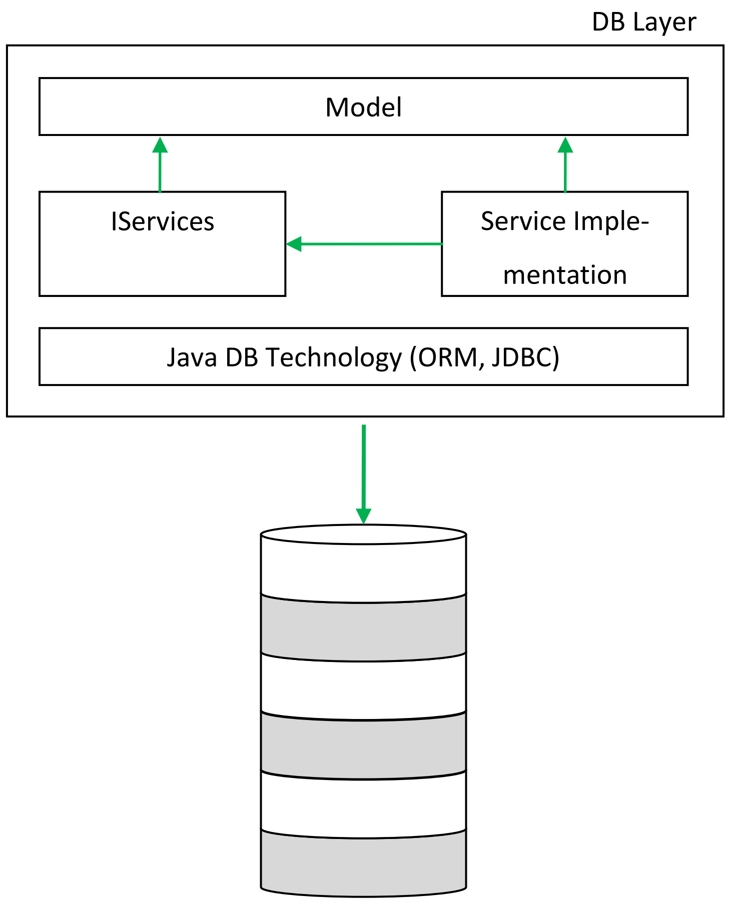


Picture 4-1: Layering Structure

4.2 Database Layering

In order for the API to execute its goal successfully, a connection to a database in which data will be persisted is mandatory. We chose MySQL as our RDBMS, because it is open source and it is developed under Oracle Corporation, the same company that runs Java. Thus, the communication between the Java programming language and MySQL database is particularly easy and efficient.

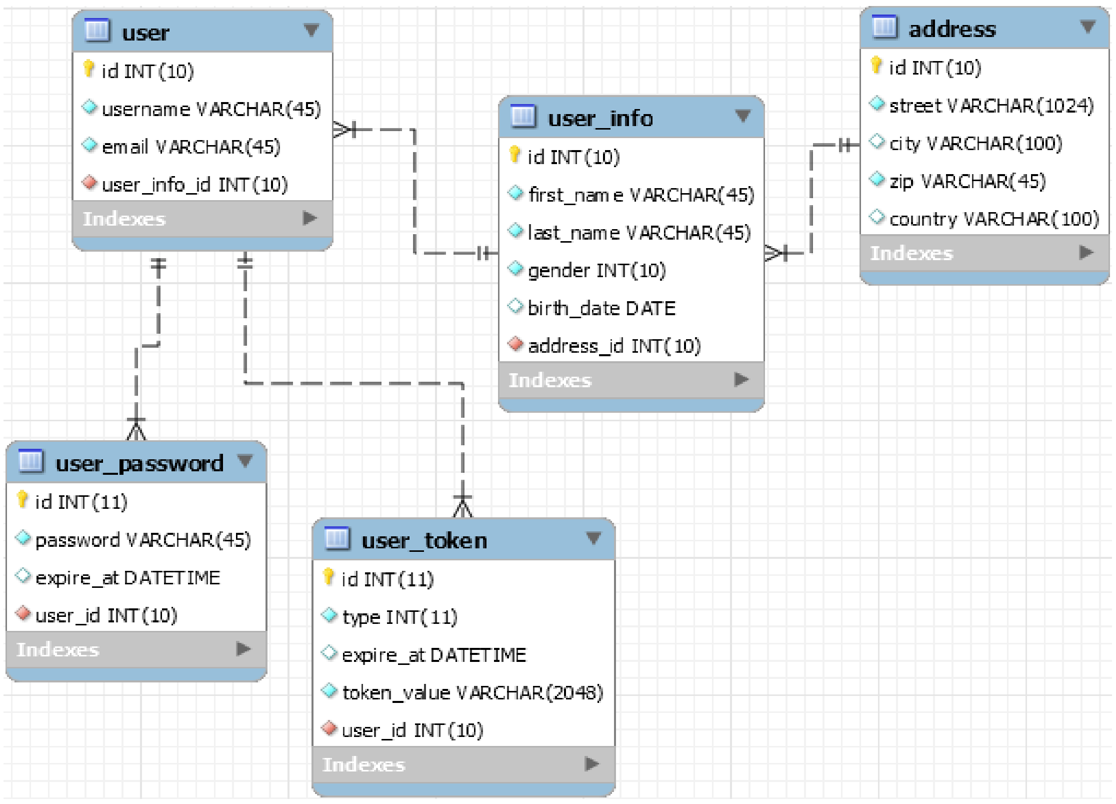
According to what we mentioned above, the DB layer is responsible for database connectivity. Let us now focus only on this layer.



Picture 4-2: DB Layer

The “Model” describes the data that must be persisted in the database like users, users’ info, their credentials, etc. “IServices” and the corresponding “Service implementation”, are essentially the controllers of the CRUD operations. It should be highlighted here that we prefer abstractions over implementations, as all senior developers recommend working with interfaces and not directly with implementations, exploiting the most of polymorphism.

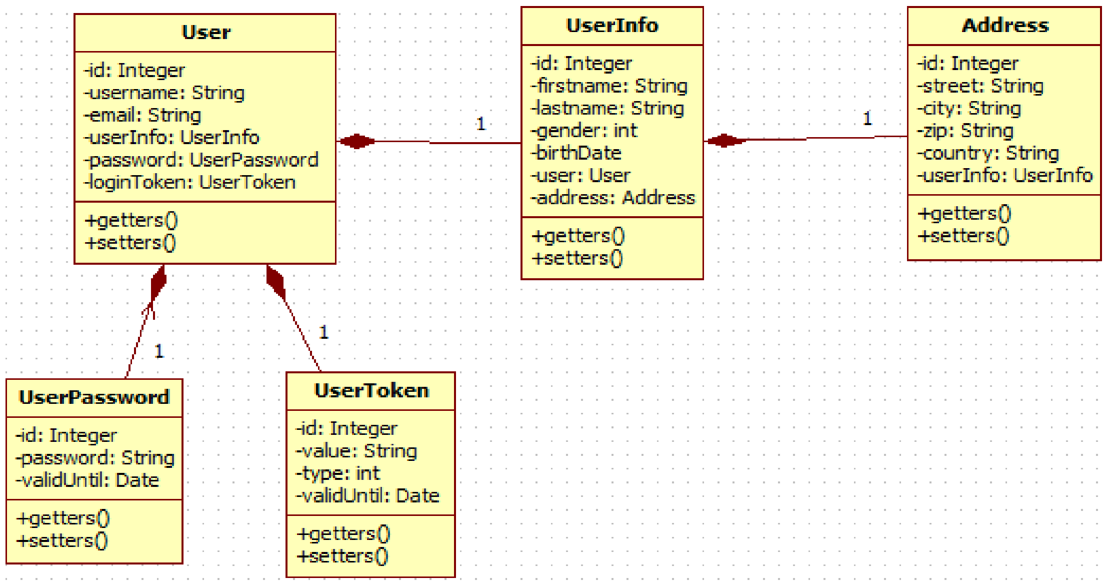
Initially, we created a database schema called “auction_db”, and there we created three tables: “user”, “user_info” and “address”. The reason why we created these three tables is to persist the three POJO (Plain Old Java Object) classes: “User”, “UserInfo” and “Address”, which are necessary for describing the users that are participating in the auction process. The tables, “user_password” and “user_token”, are necessary for ensuring the API security. The following figure illustrates the EER Diagram of this schema. This EER Diagram was designed with the MySQL Workbench tool.[11]



Picture 4-3: EER Diagram

The corresponding UML Class Diagram is shown below. The tool that was used is StarUML[12].

[11] https://www.mysql.com/products/workbench
[12] http://staruml.io



Picture 4-4: UML Class Diagram

4.2.1 ORM (Object Relational Mapping)

The hardest part of saving, retrieving, updating and deleting Java entities in a database, is the complicated SQL commands that must be executed. This is where *ORM* comes into play. ORM is a mechanism that is used to resolve the Java code and relational database mismatch. It is an ordinary library written in Java that encapsulates the code needed to manipulate the data, so that the user does not need to use SQL anymore, but directly a Java object. Behind the sense ORM uses the JDBC API. There are many ORM tools in Java, such as Hibernate, TopLink, JDO and EclipseLink. Nevertheless, the Java Persistence API, or *JPA*, is the standard persistence API, firstly introduced as part of the Java EE 5 platform. The reason why we decided to use JPA in this implementation, and not a specific persistence technologies that was previously mentioned, is because JPA is a standardized specification that helps a developer to build a persistence layer that is independent of any particular persistence provider.

4.2.2 Maven

In order to make the implementation process less cumbersome, we made use of *Maven*[13]. Maven's primary goal is to allow a developer to comprehend the complete state of a development effort in the shortest period of time. Maven is a lot of things, putting in one. First and foremost, is a build automation tool, which means that it helps us build

[13] https://maven.apache.org

our code in our development environment. Another role that Maven plays is the one of a project management tool, which helps us generate reports and manage dependencies. Maven is extremely useful, as it manages the multiple jars that are needed for the deployment and describes the necessary dependencies. It uses conventions for the build procedure, and only exceptions need to be written down. It is so powerful that all the well-known IDEs use these conventions. Maven allows a project to be built using an XML file called Project Object Model (POM), which describes the project structure, its dependencies and the versions on other external components, the building order, and the required plug-ins. It comes with pre-defined targets for performing certain well-defined tasks, such as compilation of code and its packaging. Maven automatically downloads Java libraries and Maven plug-ins from what is called “repositories”. A repository has two types of information. The first type of information is about the archetype information, which has the details about the different types of projects you want to create, as well as the folder structure and all the required information about creating a new project of that particular type. The second type of information is the dependency information, which is a list of all the jar files that you normally use, and the other jars that are dependent on the jars that you need. The main repository is the “Central Repository”. However, when you download an artifact, this is copied also in a local repository in your machine, and when is needed again, it does not have to be downloaded again from the Central Repository.

4.2.3 Java Entities

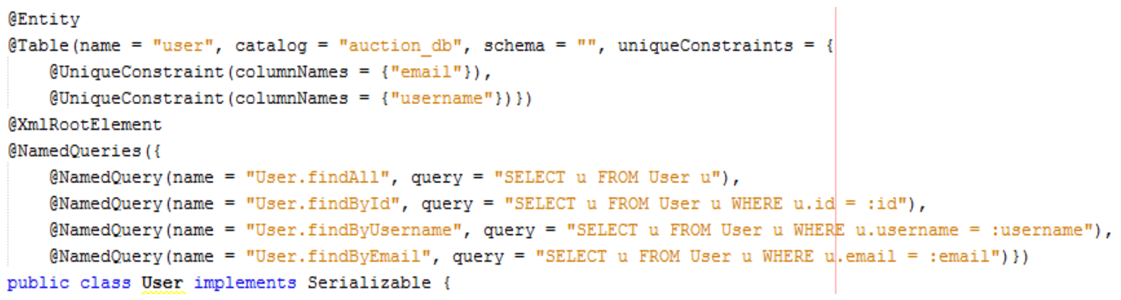
An entity is a lightweight persistence domain object. Typically, an entity represents a table in a relational database, and each entity instance corresponds to a row in that table. The primary programming artifact of an entity is the entity class. The persistent state of an entity is represented either through persistent fields or persistent properties. These fields or properties use object relational mapping annotations to map the entities and entity relationships to the relational data in the underlying data store [(16)].

An entity class must stick with the following requirements:

- The class must be annotated with the *javax.persistence.Entity* annotation.
- The class must have a public or protected, no-argument constructor. The class may have other constructors.
- The class must implement the *Serializable* interface.

- The class, the methods and the persistent instance variables must not be declared as *final*.
- Persistent instance variables must be declared *private* or *protected* and can only be accessed directly by the entity class's methods.

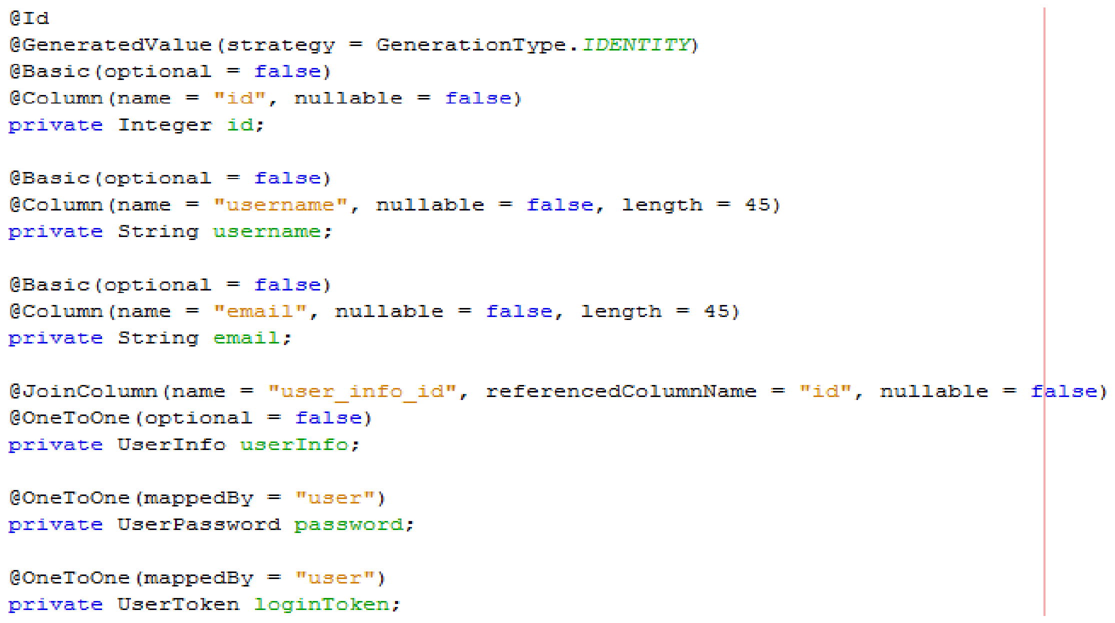
The next figure shows how the class *User* was developed, following the above mandatory requirements, plus some extra ones. All the other entities in the API are built in a similar way.



```
@Entity
@Table(name = "user", catalog = "auction_db", schema = "", uniqueConstraints = {
    @UniqueConstraint(columnNames = {"email"}),
    @UniqueConstraint(columnNames = {"username"})})
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "User.findAll", query = "SELECT u FROM User u"),
    @NamedQuery(name = "User.findById", query = "SELECT u FROM User u WHERE u.id = :id"),
    @NamedQuery(name = "User.findByUsername", query = "SELECT u FROM User u WHERE u.username = :username"),
    @NamedQuery(name = "User.findByEmail", query = "SELECT u FROM User u WHERE u.email = :email")})
public class User implements Serializable {
```

Picture 4-5: User Class

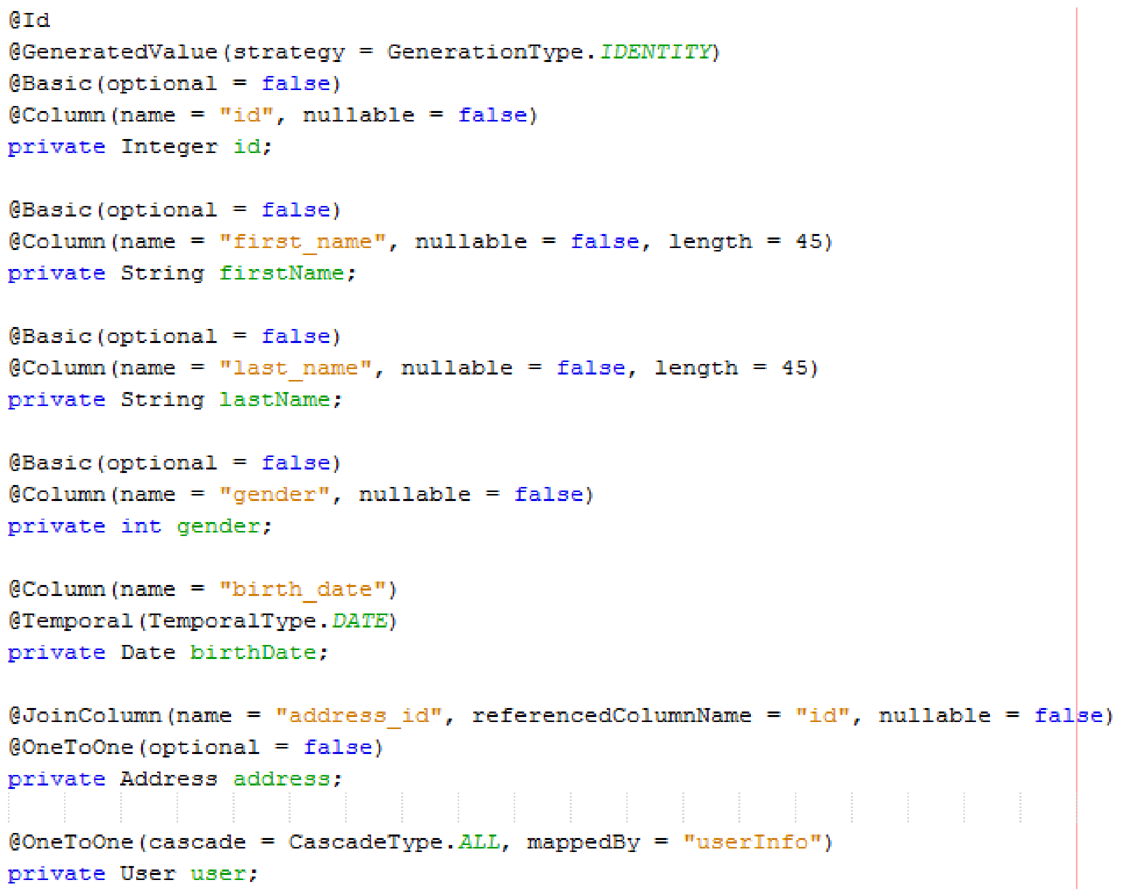
On the top it is the *@Entity* annotation which reveals that this class is intended to be persisted in a database. The *@Table* annotation is used for changing the default name of the table (which is the same as the Java Class), as well as for giving some metadata information about the table itself, such as the schema and constraints. The *@XmlRootElement* annotation is of paramount importance because without this annotation it would not be feasible for data persisted in the database to be parsed in XML or JSON format. Lastly, there are the *@NamedQueries* and *@NamedQuery* annotations. A named query is a pre-defined query with a predefined unchangeable query string. By using named queries instead of dynamic queries, code organization can be improved owing to a single place of writing them. In order to write named queries, JPQL (Java Persistence Query Language) is used. JPQL is defined in JPA specification and is not SQL, yet, is very similar to the syntax of SQL. The advantage of having SQL like syntax is that SQL is widely used by many developers, and thus there is no need to learn another language from scratch. The main difference between these two query languages is that SQL works with relational database tables, whilst JPQL works with Java objects. The next figure illustrates the instance variables of the class, which are essentially the ones that are persisted in the database and constitute the columns of the table.



```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Basic(optional = false)
@Column(name = "id", nullable = false)
private Integer id;

@Basic(optional = false)
@Column(name = "username", nullable = false, length = 45)
private String username;

@Basic(optional = false)
@Column(name = "email", nullable = false, length = 45)
private String email;

@JoinColumn(name = "user_info_id", referencedColumnName = "id", nullable = false)
@OneToOne(optional = false)
private UserInfo userInfo;

@OneToOne(mappedBy = "user")
private UserPassword password;

@OneToOne(mappedBy = "user")
private UserToken loginToken;
```

Picture 4-6: User Class Fields& Annotations

The *id* instance variable is annotated with the *@Id*, which means that this attribute is going to be treated as primary key in the user table. Furthermore, the *@GenaratedValue* annotation is used to auto-increment the primary key. The *@Column* annotation is used for renaming the default name of the column, and also for giving some extra metadata, such as whether a column can be null or not. The *userInfo* attribute has two annotations of great importance. *@JoinColum*n and *@OneToOne*. Both are critical for joining two tables and for declaring the cardinalities between the relationships. In particular, the former essentially says that the user table has a column called “user_info_id”, which is a foreign key, and links to the column “id” of the user_info table, which is its primary key. The latter says that each instance of that entity is related to a single instance of the other entity. In our case, a single user can have only one user info. On the other side, the UserInfo Class’s attributes are written as follows:



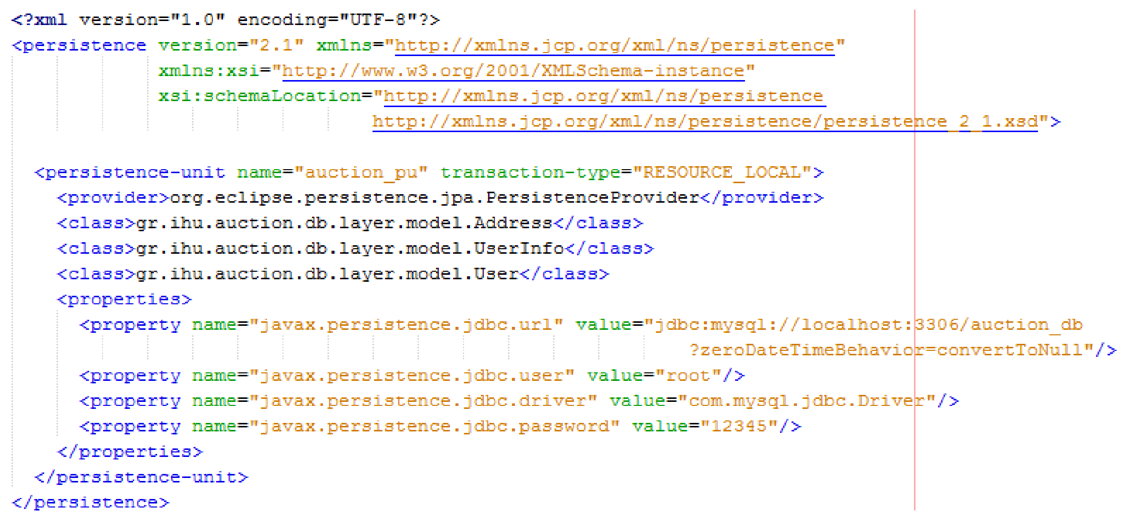
```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Basic(optional = false)
@Column(name = "id", nullable = false)
private Integer id;

@Basic(optional = false)
@Column(name = "first_name", nullable = false, length = 45)
private String firstName;

@Basic(optional = false)
@Column(name = "last_name", nullable = false, length = 45)
private String lastName;

@Basic(optional = false)
@Column(name = "gender", nullable = false)
private int gender;

@Column(name = "birth_date")
@Temporal(TemporalType.DATE)
private Date birthDate;

@JoinColumn(name = "address_id", referencedColumnName = "id", nullable = false)
@OneToOne(optional = false)
private Address address;

@OneToOne(cascade = CascadeType.ALL, mappedBy = "userInfo")
private User user;
```

Picture 4-7: UserInfo Class Fields& Annotations

The *user* attribute and its @OneToOne annotation, uses the *CascadeType.ALL* enumeration for saying that the persistence will cascade all *EntityManager* operations, such as PERSIST, REMOVE, REFRESH, MERGE and DETACH, to the relating entities. The *mappedBy* attribute signals to the persistence provider that the join column is in the "user_info" table. Finally, the *address* attribute and its annotations shows respectively that the "user_info" table is linked with the "address" table with exactly the same one-to-one relationship as the "user" table with the "user_info" table.

4.2.4 Persistence.xml

A JPA Persistence Unit is a logical grouping of user defined entity classes with related settings, such as the particular database and schema the user wants to connect to, as well as the corresponding credentials. A Persistence Unit is defined in an XML file called *persistence.xml*, which has to be located in the META-INF directory in the classpath. One persistence.xml file can include definitions for more than one persistence units. The persistence.xml of the "auction_db" schema would be as follows:



```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
                                 http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">

  <persistence-unit name="auction_pu" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>gr.ihu.auction.db.layer.model.Address</class>
    <class>gr.ihu.auction.db.layer.model.UserInfo</class>
    <class>gr.ihu.auction.db.layer.model.User</class>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/auction_db
                                                        ?zeroDateTimeBehavior=convertToNull"/>
      <property name="javax.persistence.jdbc.user" value="root"/>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.password" value="12345"/>
    </properties>
  </persistence-unit>
</persistence>
```

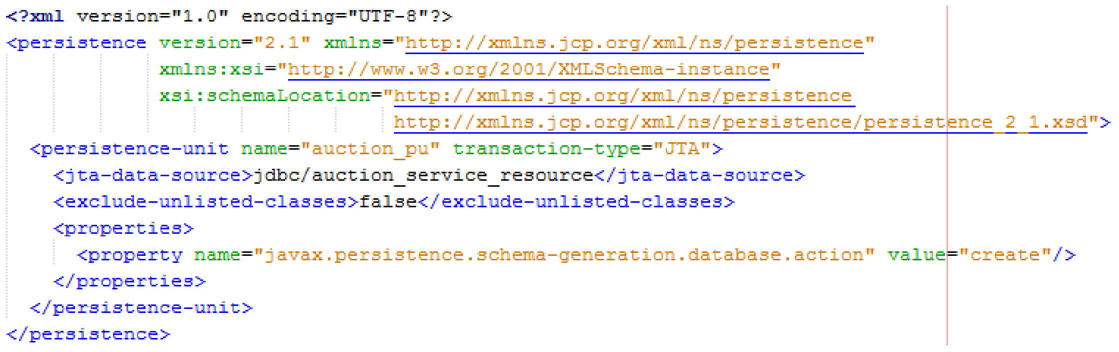
Picture 4-8: Persistence.xml File

It is worth noting the two attributes in the *persistence-unit* element. The name "auction_pu" identifies the persistence unit when instantiating an *EntityManagerFactory*[14]. In other words, it is the name that must be used when you want to deal with this persistence-unit. The transaction-type="RESOURCE_LOCAL" indicates that you as a developer are responsible for creating an *EntityManager*[15], opening and closing a session every time you want to make a change in database, and generally doing all the tedious and monotonous job.

However, due to the fact that we are developing a web application, we would like the server to provide us with some automated functionalities. More specifically, we have deployed our web service in the *GlassFish* server[16], which includes, among other, an *EJB* container. EJB (Enterprise Java Beans) technology is the server-side component architecture for Java Platform Enterprise Edition (Java EE). EJB technology enables rapid and simplified development of distributed, transactional, secure and portable applications based on Java technology [(16)]. EJB container provides some lower level services, such as Life Cycle Management, Database pooling, Transactions and Security. With these services, the developer focuses only on the business logic of the application, leaving the rest to the container.

[14] The EntityManagerFactory interface is used by the application to obtain an application-managed entity manager.
[15]EntityManager API is used for CRUD operations among entities
[16] https://glassfish.java.net

As expected, the persistence.xml file in this situation must be altered so as to comply with the new conditions. The next figure displays these changes.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
                                  http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="auction_pu" transaction-type="JTA">
    <jta-data-source>jdbc/auction_service_resource</jta-data-source>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties>
      <property name="javax.persistence.schema-generation.database.action" value="create"/>
    </properties>
  </persistence-unit>
</persistence>
```

Picture 4-9: Altered Persistence.xml File

The seemingly basic change is in the persistence-unit element, where the transaction-type attribute has value "*JTA*". This value indicates that the EJB container will supply us with an EntityManager, and additionally will take care all the transaction operations. Of course, in order for all these automated operations to work properly, we needed to make the appropriate settings in the GlassFish server. We visited the admin console page at port 4848, and from there, we created a JDBC connection pool, named "auction_service_mysql_pool", with all the necessary information for the server, like the database driver name, its URL, the maximum number of connections that can be created to satisfy client requests, the username and the password. In parallel, we created a JDBC Resource which points to the above connection pool. The corresponding JNDI[17] name is "jdbc/auction_service_resource". The "jta-data-source" element specify the global JNDI name of the data source to be used by the container.

4.3 Services

Having said and done all that, we now have built the database layer which is a fundamental part of our RESTful API. Yet, the core modules of this API are the services themselves. These are the ones that would provide the additional functionalities in an exterior application. These are the ones that actually make real the SOA architectural pattern, whose principles should be followed when a new software is designed and planned to be developed.

[17] http://www.oracle.com/technetwork/java/jndi/index.html

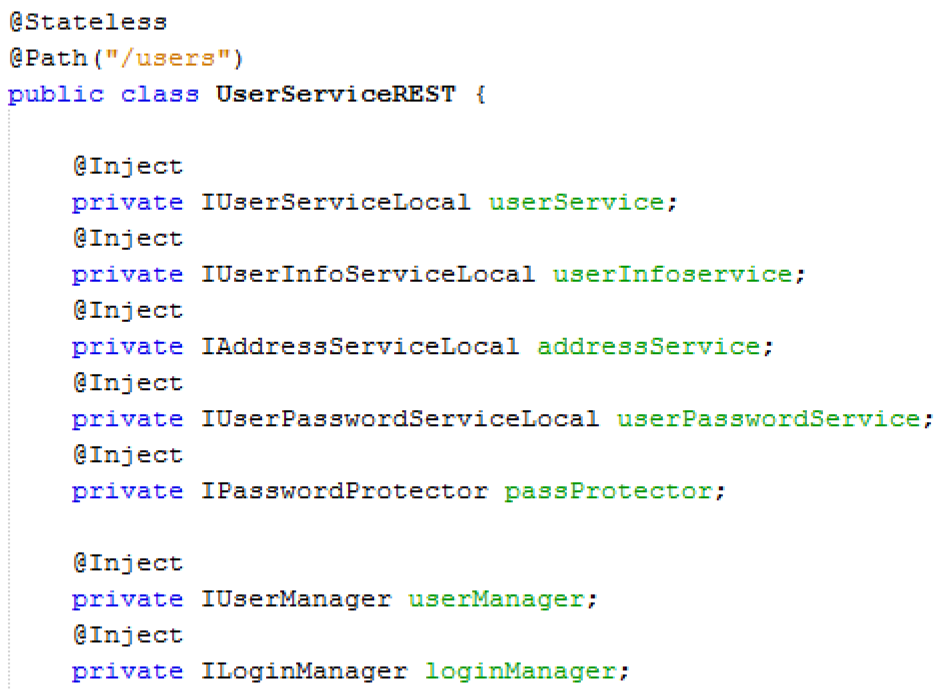
4.3.1 Users

Broadly speaking, in order for Jersey to be aware of what class is mapped with a particular resource, what Java method should be called when a client sends a specific HTTP method, and in what format must send the response back, three annotations have to be utilized.

The *@Path* annotation identifies the URI path to which the resource responds, and is specified at the class level (or method level if you want to access nested resources) of a resource [(16)]. In other words, it binds a URI pattern to a Java class or a Java method. The *@GET* annotation, and correspondingly all the other annotations for the HTTP methods, binds an HTTP GET request with a specific Java method. Finally, the *@Produces* annotation gives the insight to Jersey to send back the response in a specific format.

Apart from these annotations, due to the fact that our RESTful API makes use of an EJB container, *@Stateless* annotation is necessary, since it is used to mark the class as Stateless Session Bean. Stateless Session Beans are business objects that do not have state associated with them. This is the exact behavior we want to achieve, since one of the basic principle of REST services is that the client-server communication should be stateless.

Thereafter, in the next figures we present the first service that we developed. Along the way, we will explain every piece of code in this Java class, including how the facilities of the EJB container works hand in hand with Jersey. It is noted that the basic URL: *localhost:8080/integrated-auction-service/* forms the application context, just like all the web applications on the internet. Of course, in a real application the domain name it would not be “localhost”. The RESTful API accepts request to the URL: *localhost:8080/integrated-auction-service/rs/.* After that point are mapped all the several services that we have created. Obviously, the service below is bind to the URL: *localhost:8080/integrated-auction-service/rs/users*

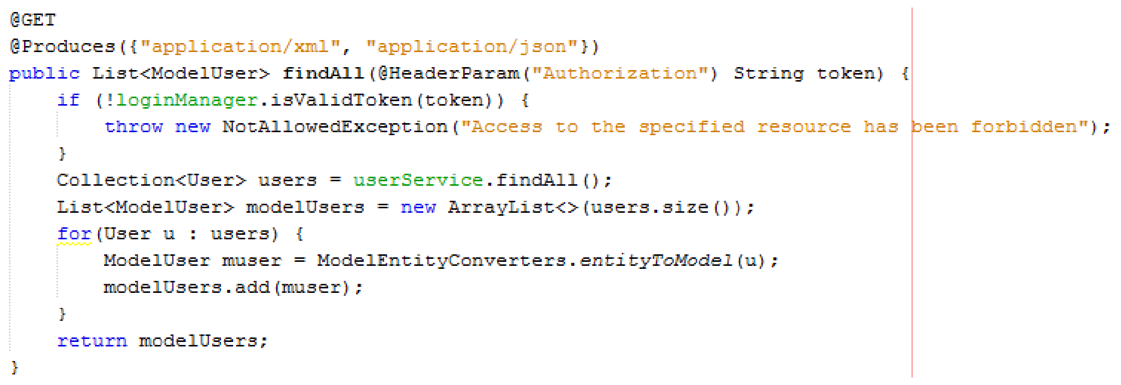


```
@Stateless
@Path("/users")
public class UserServiceREST {

    @Inject
    private IUserServiceLocal userService;
    @Inject
    private IUserInfoServiceLocal userInfoservice;
    @Inject
    private IAddressServiceLocal addressService;
    @Inject
    private IUserPasswordServiceLocal userPasswordService;
    @Inject
    private IPasswordProtector passProtector;

    @Inject
    private IUserManager userManager;
    @Inject
    private ILoginManager loginManager;
```

Picture 4-10: Users REST Service

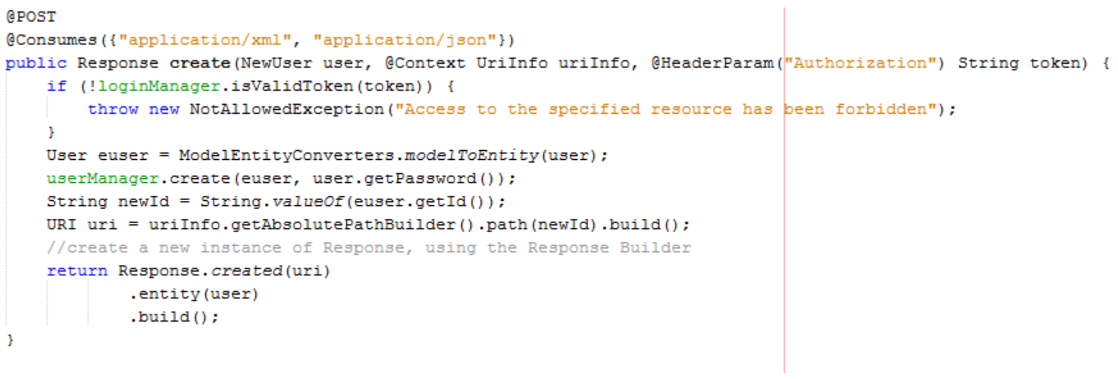
The *@Inject* annotation is used to inject all the requisite Stateless EJBs into the JAX-RS Web Service. In essence, it injects the dependencies and leave the container do the job. The first five dependencies are for database transactions, while the last two has to do with the business logic and the security mechanism of the RESTful API.

The following figures illustrate all the java methods that are bind with the various HTTP methods. All the parts of the code that are associated with the security features of the API, will be discussed in the next section.



```
@GET
@Produces({"application/xml", "application/json"})
public List<ModelUser> findAll(@HeaderParam("Authorization") String token) {
    if (!loginManager.isValidToken(token)) {
        throw new NotAllowedException("Access to the specified resource has been forbidden");
    }
    Collection<User> users = userService.findAll();
    List<ModelUser> modelUsers = new ArrayList<>(users.size());
    for(User u : users) {
        ModelUser muser = ModelEntityConverters.entityToModel(u);
        modelUsers.add(muser);
    }
    return modelUsers;
}
```

Picture 4-11: GET Request in the Users Service

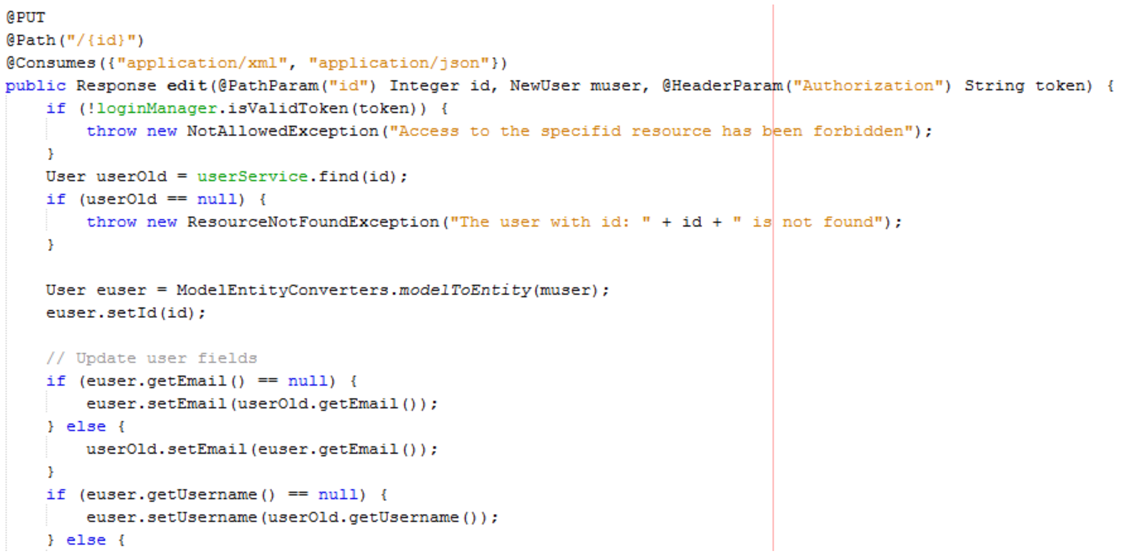


```
@POST
@Consumes({"application/xml", "application/json"})
public Response create(NewUser user, @Context UriInfo uriInfo, @HeaderParam("Authorization") String token) {
    if (!loginManager.isValidToken(token)) {
        throw new NotAllowedException("Access to the specified resource has been forbidden");
    }
    User euser = ModelEntityConverters.modelToEntity(user);
    userManager.create(euser, user.getPassword());
    String newId = String.valueOf(euser.getId());
    URI uri = uriInfo.getAbsolutePathBuilder().path(newId).build();
    //create a new instance of Response, using the Response Builder
    return Response.created(uri)
            .entity(user)
            .build();
}
```

Picture 4-12: POST Request in the Users Service

The *@Consumes* annotation in the above figure signifies that the body of a POST request in the RESTful API must be in either XML or JSON format. Remarkably, the *UriInfo* interface provides methods to enable you to find or build URI information of a request.

The next picture shows a part of the method that handles the PUT request.

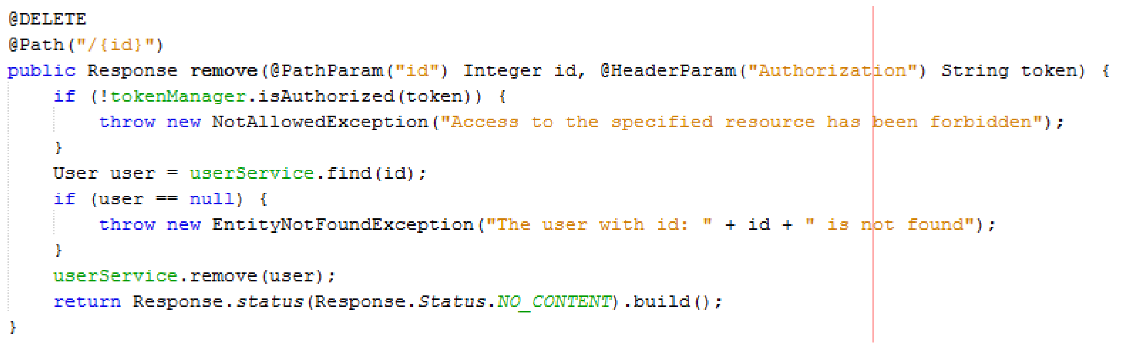


```
@PUT
@Path("/{id}")
@Consumes({"application/xml", "application/json"})
public Response edit(@PathParam("id") Integer id, NewUser muser, @HeaderParam("Authorization") String token) {
    if (!loginManager.isValidToken(token)) {
        throw new NotAllowedException("Access to the specifid resource has been forbidden");
    }
    User userOld = userService.find(id);
    if (userOld == null) {
        throw new ResourceNotFoundException("The user with id: " + id + " is not found");
    }

    User euser = ModelEntityConverters.modelToEntity(muser);
    euser.setId(id);

    // Update user fields
    if (euser.getEmail() == null) {
        euser.setEmail(userOld.getEmail());
    } else {
        userOld.setEmail(euser.getEmail());
    }
    if (euser.getUsername() == null) {
        euser.setUsername(userOld.getUsername());
    } else {
```

Picture 4-13: PUT Request in the Users Service

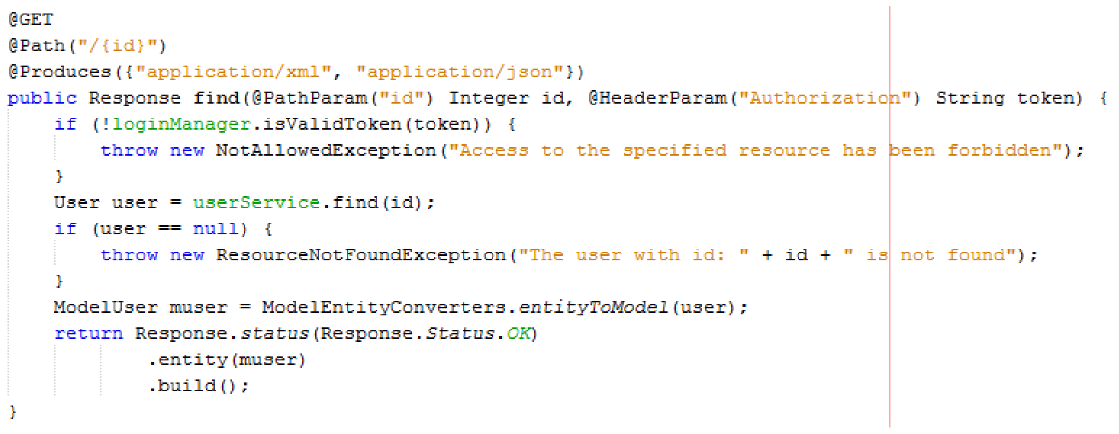
It is noteworthy that in this method we are dealing for the first time with a nested resource, meaning that the URL path *localhost:8080/integrated-auction-service/rs/users* can be embedded with user-typed variables. These variables are substituted at runtime in order for a resource to respond to a request, based on the substituted URI. Variables are denoted by curly braces { } [(16)]. More specifically, we use *@PathParam* to inject the value of URI variable that is defined in @Path expression, into the Java method.

All the next figures make use of @PathParam annotation in order for all the functionalities of the RESTful API to be covered.



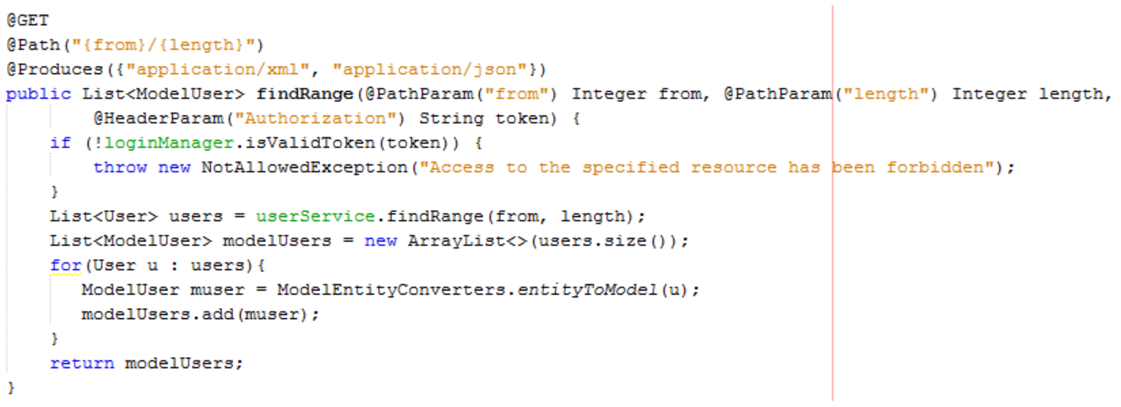
```
@DELETE
@Path("/{id}")
public Response remove(@PathParam("id") Integer id, @HeaderParam("Authorization") String token) {
    if (!tokenManager.isAuthorized(token)) {
        throw new NotAllowedException("Access to the specified resource has been forbidden");
    }
    User user = userService.find(id);
    if (user == null) {
        throw new EntityNotFoundException("The user with id: " + id + " is not found");
    }
    userService.remove(user);
    return Response.status(Response.Status.NO_CONTENT).build();
}
```

Picture 4-14: DELETE Request in the Users Service



```
@GET
@Path("/{id}")
@Produces({"application/xml", "application/json"})
public Response find(@PathParam("id") Integer id, @HeaderParam("Authorization") String token) {
    if (!loginManager.isValidToken(token)) {
        throw new NotAllowedException("Access to the specified resource has been forbidden");
    }
    User user = userService.find(id);
    if (user == null) {
        throw new ResourceNotFoundException("The user with id: " + id + " is not found");
    }
    ModelUser muser = ModelEntityConverters.entityToModel(user);
    return Response.status(Response.Status.OK)
            .entity(muser)
            .build();
}
```

Picture 4-15: GET By Id Request in the Users Service

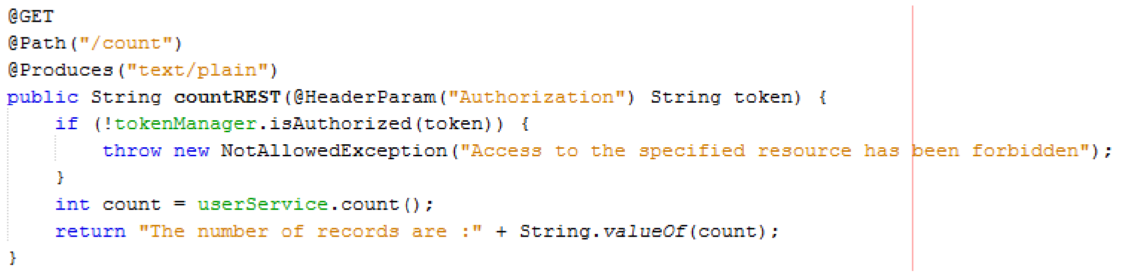


```
@GET
@Path("{from}/{length}")
@Produces({"application/xml", "application/json"})
public List<ModelUser> findRange(@PathParam("from") Integer from, @PathParam("length") Integer length,
        @HeaderParam("Authorization") String token) {
    if (!loginManager.isValidToken(token)) {
        throw new NotAllowedException("Access to the specified resource has been forbidden");
    }
    List<User> users = userService.findRange(from, length);
    List<ModelUser> modelUsers = new ArrayList<>(users.size());
    for(User u : users){
        ModelUser muser = ModelEntityConverters.entityToModel(u);
        modelUsers.add(muser);
    }
    return modelUsers;
}
```

Picture 4-16: GET With Range Request in the Users Service

In the above picture, it is shown that is possible to use more than one variables in the @Path annotation.

Last but not least, it is presented the functionality which gives a client the ability to count the records of those that are participating in the auction.



```
@GET
@Path("/count")
@Produces("text/plain")
public String countREST(@HeaderParam("Authorization") String token) {
    if (!tokenManager.isAuthorized(token)) {
        throw new NotAllowedException("Access to the specified resource has been forbidden");
    }
    int count = userService.count();
    return "The number of records are :" + String.valueOf(count);
}
```

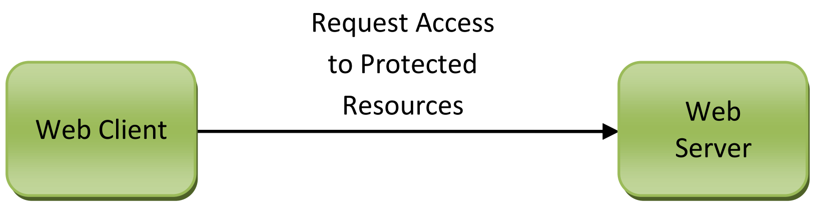
Picture 4-17: GET With Range Request in the Users Service

It should be noted that this method sends back the response not in XML or JSON format, but in plain text (@Produces("text/plain")).

4.3.2 Login

Security is a crucial part of any web application nowadays. In fact, it is inextricably linked with whether a web application would be successful or not. This RESTful API could not be the exception, and therefore we tried to provide some basic security features for all the protected resources. Before explaining practically how we implemented them in Java, let us first give an overview of how our security mechanism works.

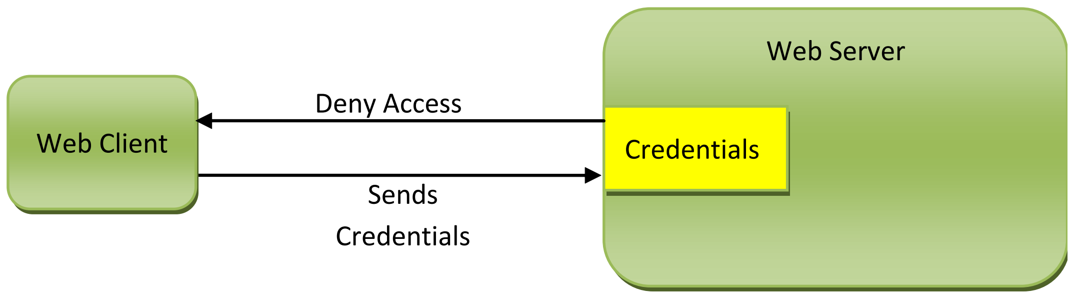
Step 1: Initial Request



Picture 4-18: Initial Request

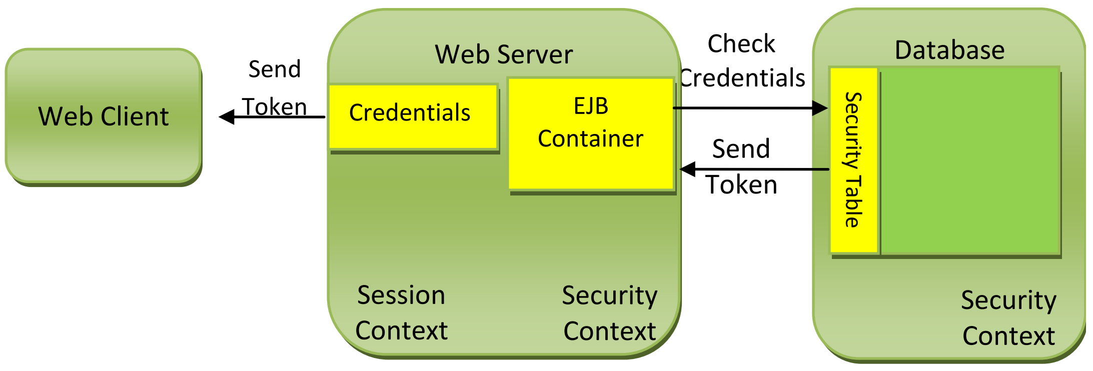
Let’s assume that the client is not yet authenticated to the application environment.

Step 2: Access Forbidden



Picture 4-19: Access Forbidden

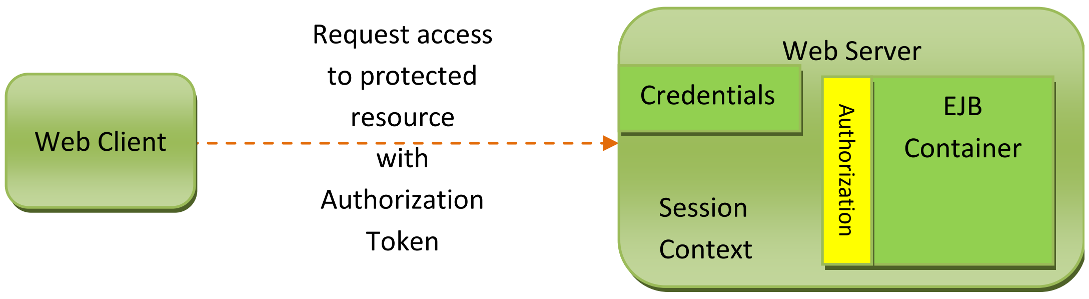
Step 3: Credential Checking & Token Sending



Picture 4-20: Credentials Checking &Token sending

As soon as the client sends his credentials, the server goes to the database and looks up whether they match or not. If the credentials are valid and there is already a token value for that user, it sends the token to the server and the server in turn sends it back to the client. If the credentials are valid but there is no token registered, the web server creates one, saves it in the database and then sends it back to the client.

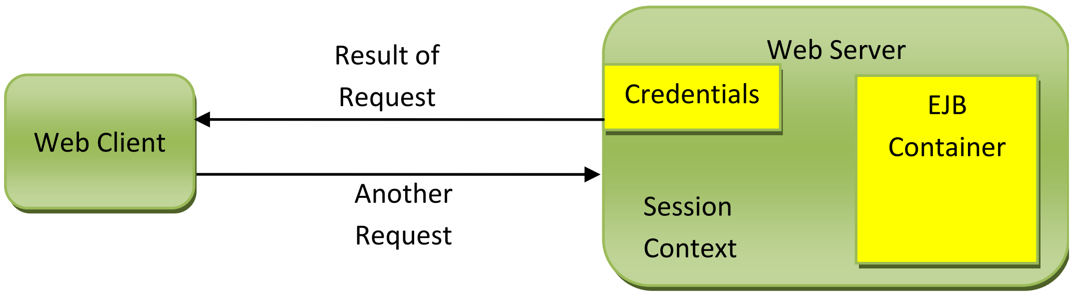
Step 4: Request Access to Protected Resources with Token



Picture 4-21: Request Access to Protected Resources with Token

The client sends again a request to the web server, but this time within the header of the request also sends the authorization token.

Step 5: Fulfilling the Original Request



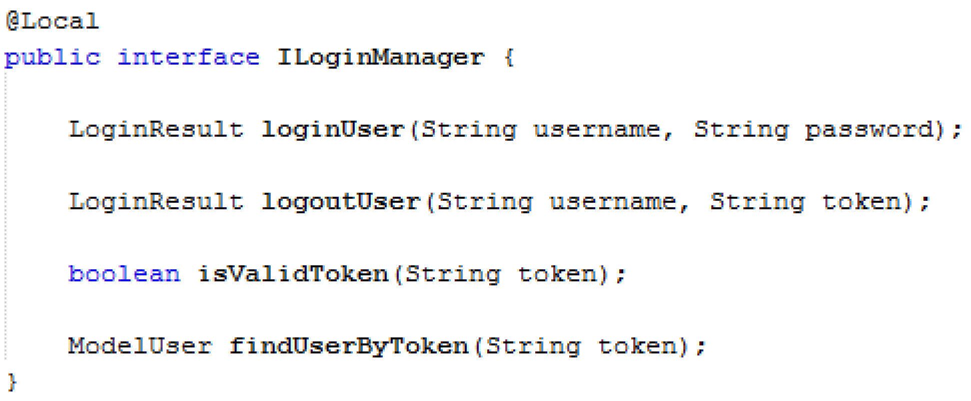
Picture 4-22: Fulfilling the Original Request

If the user is authenticated, the web server returns the result of the original URL request. After that, the client can further use the services that the web server provides (in our case, GlassFish), by sending the required requests.

Having given the overview of how the security mechanism in our RESTful API works, we can now briefly demonstrate how the *login* service has actually been implemented in Java. The following annotations are already known, since they were discussed in the previous section.

The whole security feature is, once again, architected having in mind the layering design principle. Therefore, in the business layer we created, among others, a Java class called *LoginManager* which stands between the REST API layer and the DB layer, and its responsibility is to manage our business logic and essentially define the rules of how a client should be authenticated by the system. The LoginManager class implements the *ILoginManager* interface which indicates all the operations that must be implemented.

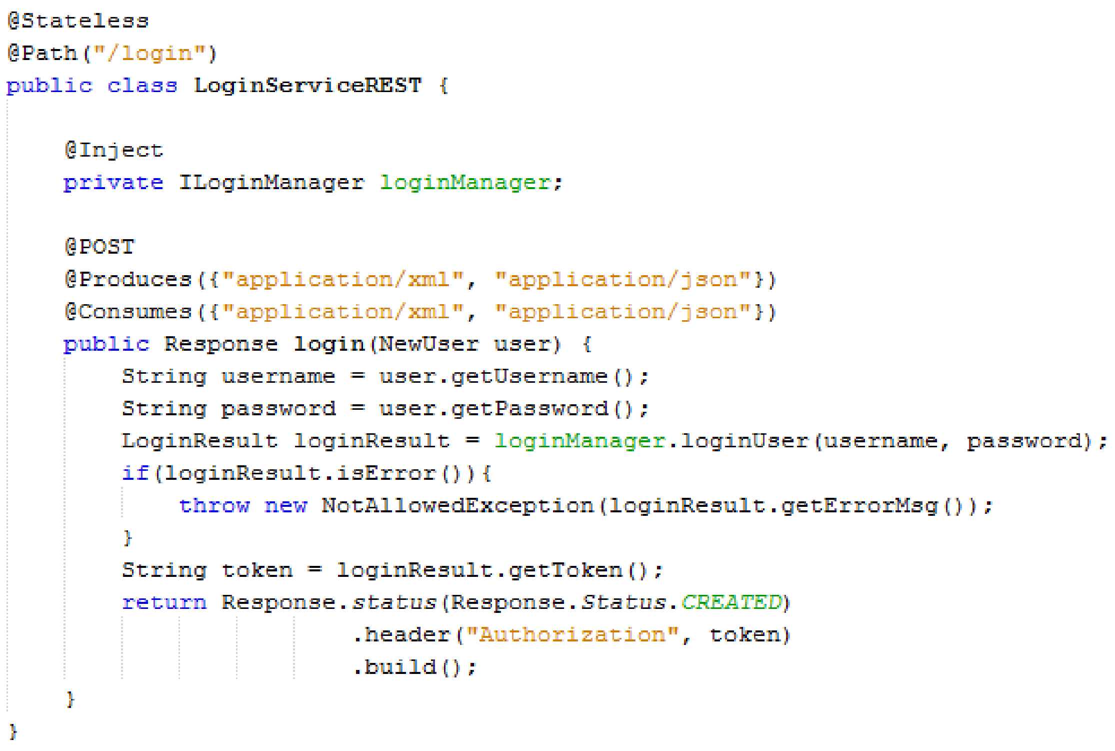
In the next two figures we present the aforementioned interface as well as the Login Service class, in order to give an insight of how the Identity Server actually manages security issues.



```
@Local
public interface ILoginManager {

    LoginResult loginUser(String username, String password);

    LoginResult logoutUser(String username, String token);

    boolean isValidToken(String token);

    ModelUser findUserByToken(String token);
}
```

Picture 4-23: LoginManager Interface

Picture 4-23 denotes that the Login Manager stipulates that certain conditions must be met in order for a stakeholder to login in the Identity Server and make use of its services. The *@Local* annotation, defines the local interface of the Bean. Picture 4-24 shows the actual implementation of the Login REST Service.



```
@Stateless
@Path("/login")
public class LoginServiceREST {

    @Inject
    private ILoginManager loginManager;

    @POST
    @Produces({"application/xml", "application/json"})
    @Consumes({"application/xml", "application/json"})
    public Response login(NewUser user) {
        String username = user.getUsername();
        String password = user.getPassword();
        LoginResult loginResult = loginManager.loginUser(username, password);
        if(loginResult.isError()){
            throw new NotAllowedException(loginResult.getErrorMsg());
        }
        String token = loginResult.getToken();
        return Response.status(Response.Status.CREATED)
                        .header("Authorization", token)
                        .build();
    }
}
```

Picture 4-24: Login Service

One last annotation that is significant to be noted is the *@HeaderParam*. This annotation is used in all the methods in all the REST services, so as Jersey to be able to read the HTTP request header. Identity Server gets the value of the “Authorization” request header, which is actually the authorization token, and checks whether a particular user should have access to its services or not.

4.3.3 Postal

Another kind of service that Identity Server provides, is the Postal Service. This time, let us first describe its characteristics and usefulness through a *sequence diagram*. A sequence diagram is a kind of interaction diagram. The main purpose of an interaction diagram is to visualize the interactive behavior of the system. It is one of the thirteen UML type diagrams and emphasizes on modelling the collaboration of objects based on a time sequence. It shows how the objects interact with others in a particular scenario of a use case. A sequence diagram can be drawn in five steps:

- Step 1: Define who will initiate the interaction
- Step 2: Draw the first message being passed to the sub-system
- Step 3: Draw other messages being passed to other sub-systems
- Step 4: Draw return messages going back to the original sender (actor)
- Step 5: Send message to objects outside the scope of the diagram (anonymous actors)

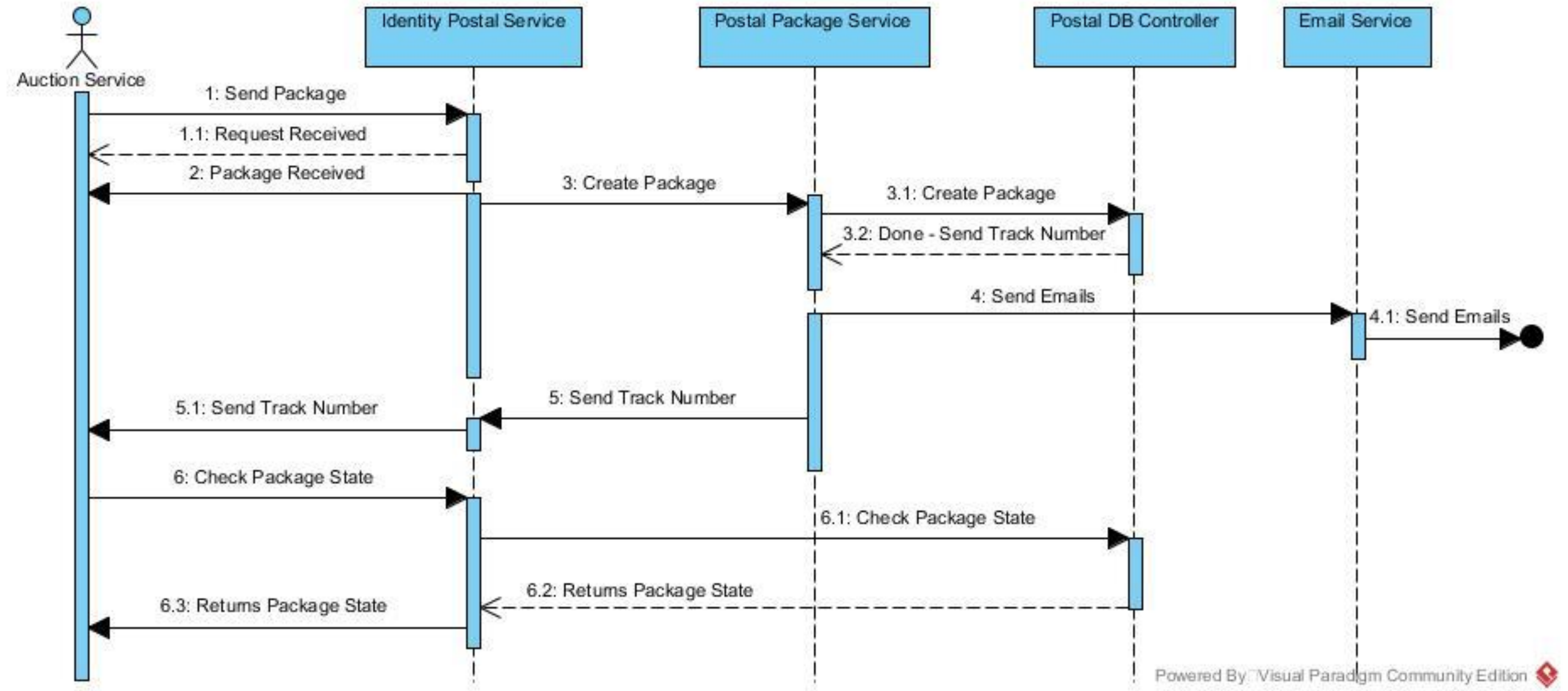
Firstly, we will present a business use case. We will go through all the steps that elapsing from the initial request to the Postal Service up to the completion of its job, and then we will provide the corresponding sequence diagram. We take for granted in this phase that whoever sends a request to Identity Postal Service, has the requisite credentials.

- Step 1: Auction Service sends a request to Identity Postal Service so as the latter to send a package from User A to User B.
- Step 2: Identity Service sends a confirmation that the request is received.
- Step 3: The package itself is received, and Identity Postal Service sends the corresponding confirmation.
- Step 4: Identity Postal Service apprise Postal Package Service (Business Layer) to create the package.

- Step 5: Postal Package Service informs Package DB Controller (DB Layer) that a package must be created and be persisted in the database.
- Step 6: Package DB Controller creates a record in the database for that package, and sends back to Postal Package Service a confirmation message, as well as the package’s track number. This number will be used later by Auction Service for checking the current state of the package. That is, getting information about the package state in any specific point of time.
- Step 7: Postal Package Service sends email to the interested parties, informing them about its actions.
- Step 8: Postal Package Service sends the track number to Identity Postal Service.
- Step 9: Identity Postal Service sends the track number to Auction Service. Auction Service can now check the package state by sending the appropriate track number to the Identity Postal Service.
- Step 10: Auction Service sends request to Identity Postal Service asking for the current state of the package with a specified track number.
- Step 11: Identity Postal Service asks Postal DB Controller to find the package state of that track number.
- Step 12: If the package state is found, DB Controller returns it.
- Step 13: Identity Postal Service sends to Auction Service the current state of the package.

It follows the aforementioned sequence diagram, which illustrates all the above steps. At the top of the diagram are the objects/roles of the system lined up in a row. In our case, the Auction Server, Identity Postal Service, Postal Package Service, Package DB Controller and Email Service. Messages are being passed horizontally between these objects, and even more importantly, the order in which the interaction occurs, moves from top to bottom in the diagram. The software tool that is used is visual-paradigm[18].

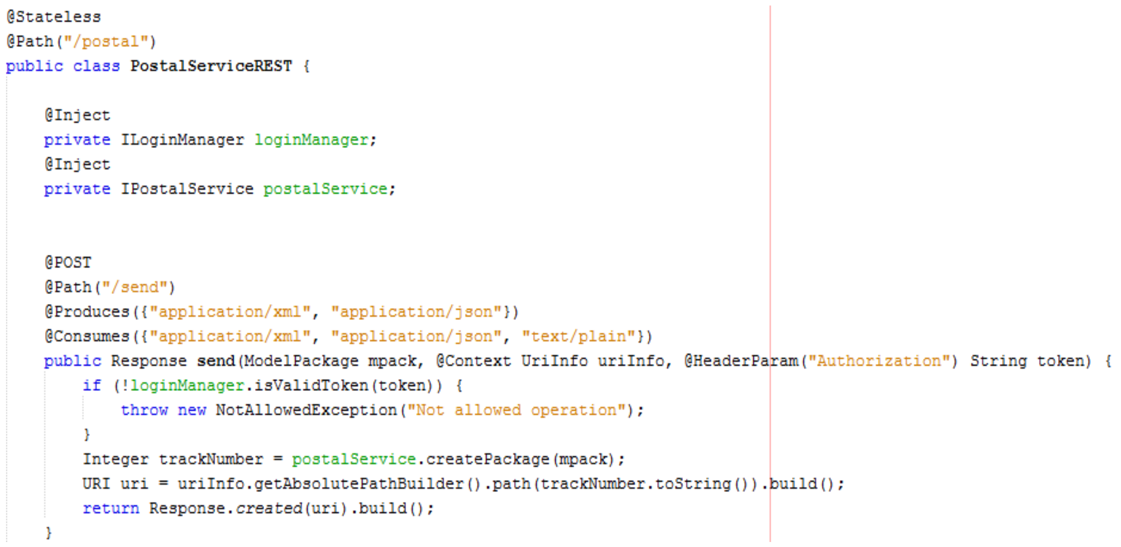
[18]http://www.visual-paradigm.com



Picture 4-25: Postal Sequence Diagram

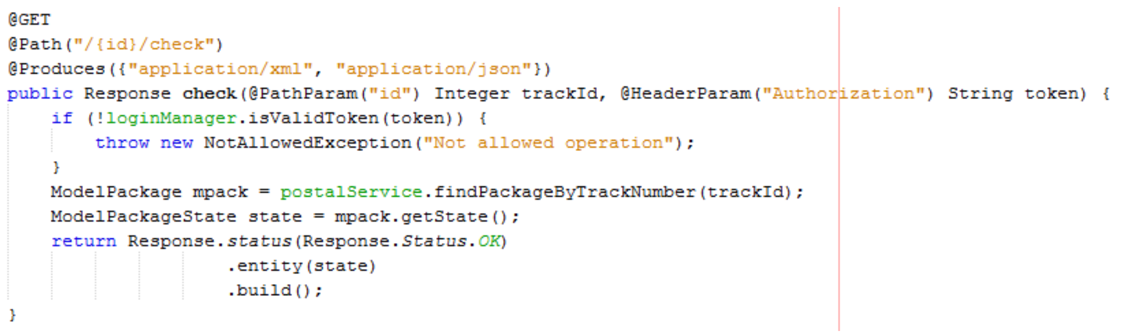
Lastly, it is briefly shown how we practically implemented the most important features of the Postal Service, which are the Postal REST Service, the Postal Package Service and the Email Service.

The Postal REST Service gives you the opportunity to send a package from User A to User B and to check the package state.



```java
@Stateless
@Path("/postal")
public class PostalServiceREST {

    @Inject
    private ILoginManager loginManager;
    @Inject
    private IPostalService postalService;


    @POST
    @Path("/send")
    @Produces({"application/xml", "application/json"})
    @Consumes({"application/xml", "application/json", "text/plain"})
    public Response send(ModelPackage mpack, @Context UriInfo uriInfo, @HeaderParam("Authorization") String token) {
        if (!loginManager.isValidToken(token)) {
            throw new NotAllowedException("Not allowed operation");
        }
        Integer trackNumber = postalService.createPackage(mpack);
        URI uri = uriInfo.getAbsolutePathBuilder().path(trackNumber.toString()).build();
        return Response.created(uri).build();
    }
```

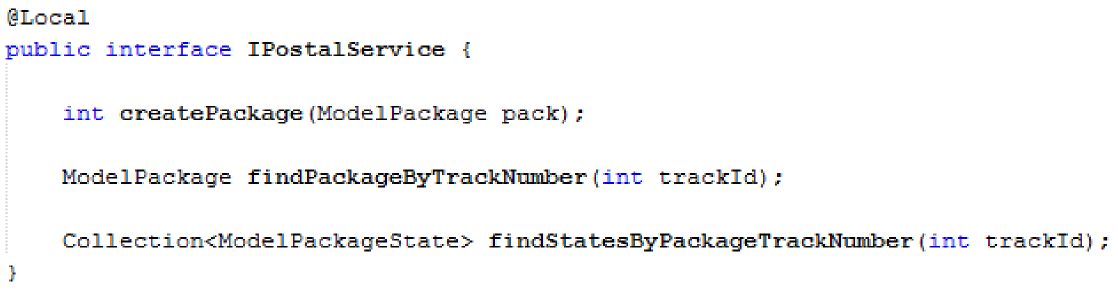
Picture 4-26: Postal REST Service (Send Package)



```java
@GET
@Path("/{id}/check")
@Produces({"application/xml", "application/json"})
public Response check(@PathParam("id") Integer trackId, @HeaderParam("Authorization") String token) {
    if (!loginManager.isValidToken(token)) {
        throw new NotAllowedException("Not allowed operation");
    }
    ModelPackage mpack = postalService.findPackageByTrackNumber(trackId);
    ModelPackageState state = mpack.getState();
    return Response.status(Response.Status.OK)
                    .entity(state)
                    .build();
}
```

Picture 4-27: Postal REST Service (Check Package State)

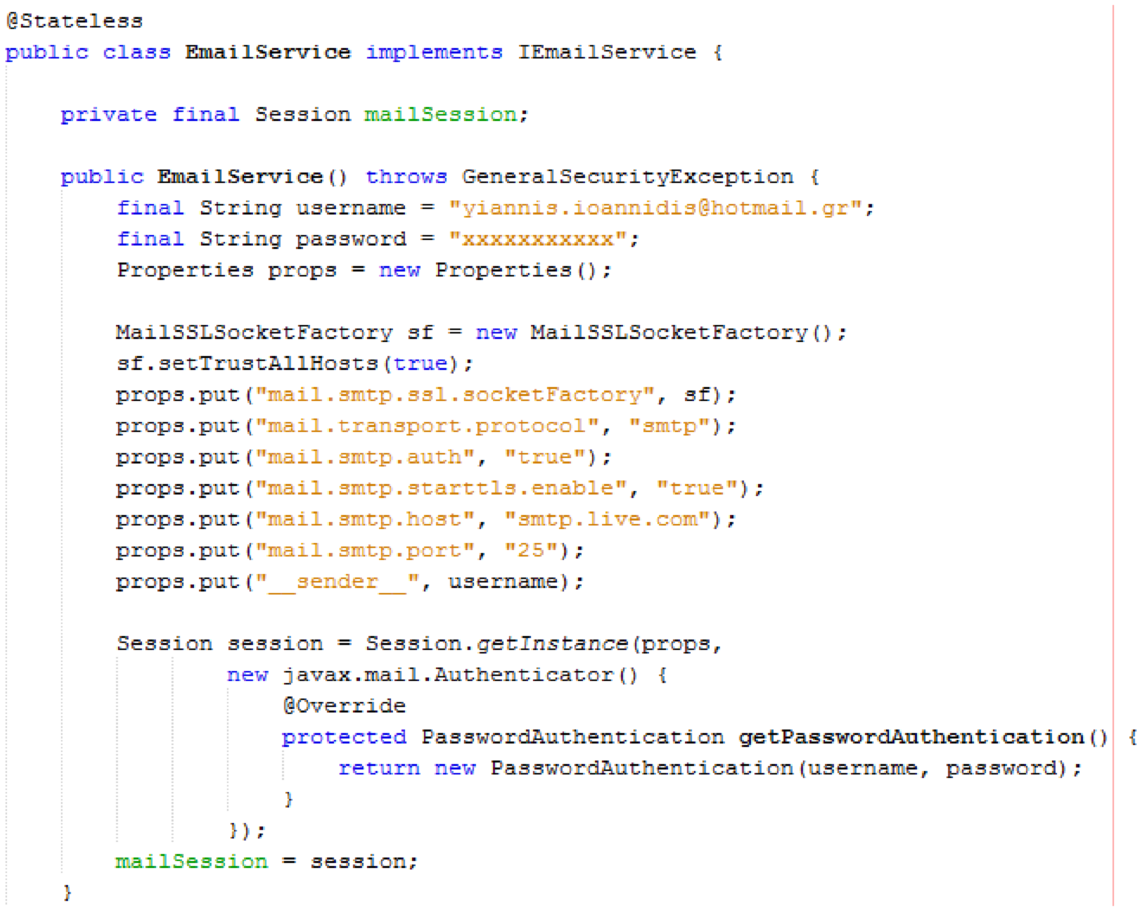
The next figure illustrates an interface called *IPostalService* which is implemented by the Postal Package Service, showing what operations must be managed. These operations are used by the Postal REST Service.



```
@Local
public interface IPostalService {

    int createPackage(ModelPackage pack);

    ModelPackage findPackageByTrackNumber(int trackId);

    Collection<ModelPackageState> findStatesByPackageTrackNumber(int trackId);
}
```

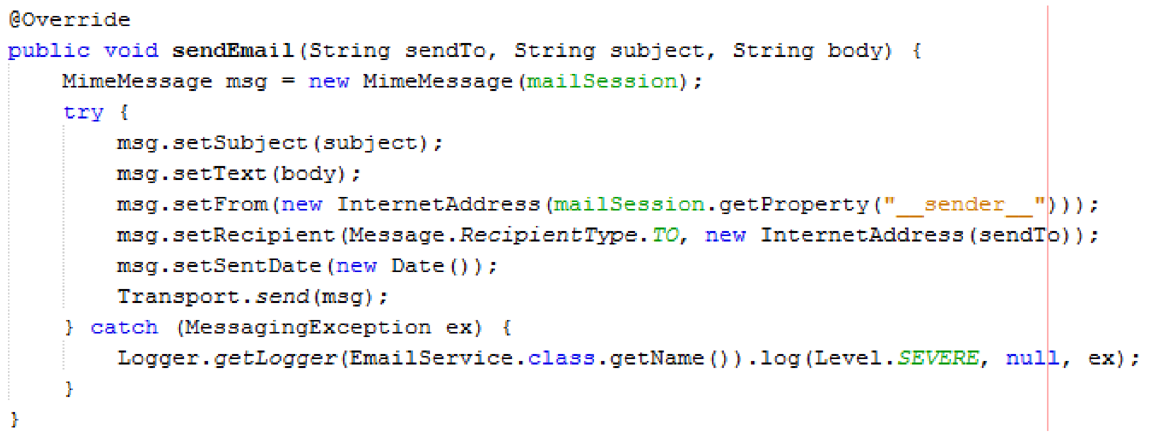
Picture 4-28: IPostalService Interface

Sending emails with Java is pretty easy. The following two figures are self-explanatory, and show the implementation of the EmailService class. The first one shows how the necessary properties are defined and created within the constructor of the class, while the second one displays how Java actually sends an email.



```
@Stateless
public class EmailService implements IEmailService {

    private final Session mailSession;

    public EmailService() throws GeneralSecurityException {
        final String username = "yiannis.ioannidis@hotmail.gr";
        final String password = "xxxxxxxxxxx";
        Properties props = new Properties();

        MailSSLSocketFactory sf = new MailSSLSocketFactory();
        sf.setTrustAllHosts(true);
        props.put("mail.smtp.ssl.socketFactory", sf);
        props.put("mail.transport.protocol", "smtp");
        props.put("mail.smtp.auth", "true");
        props.put("mail.smtp.starttls.enable", "true");
        props.put("mail.smtp.host", "smtp.live.com");
        props.put("mail.smtp.port", "25");
        props.put("__sender__", username);

        Session session = Session.getInstance(props,
                new javax.mail.Authenticator() {
                    @Override
                    protected PasswordAuthentication getPasswordAuthentication() {
                        return new PasswordAuthentication(username, password);
                    }
                });
        mailSession = session;
    }
```

Picture 4-29: Email Service Class

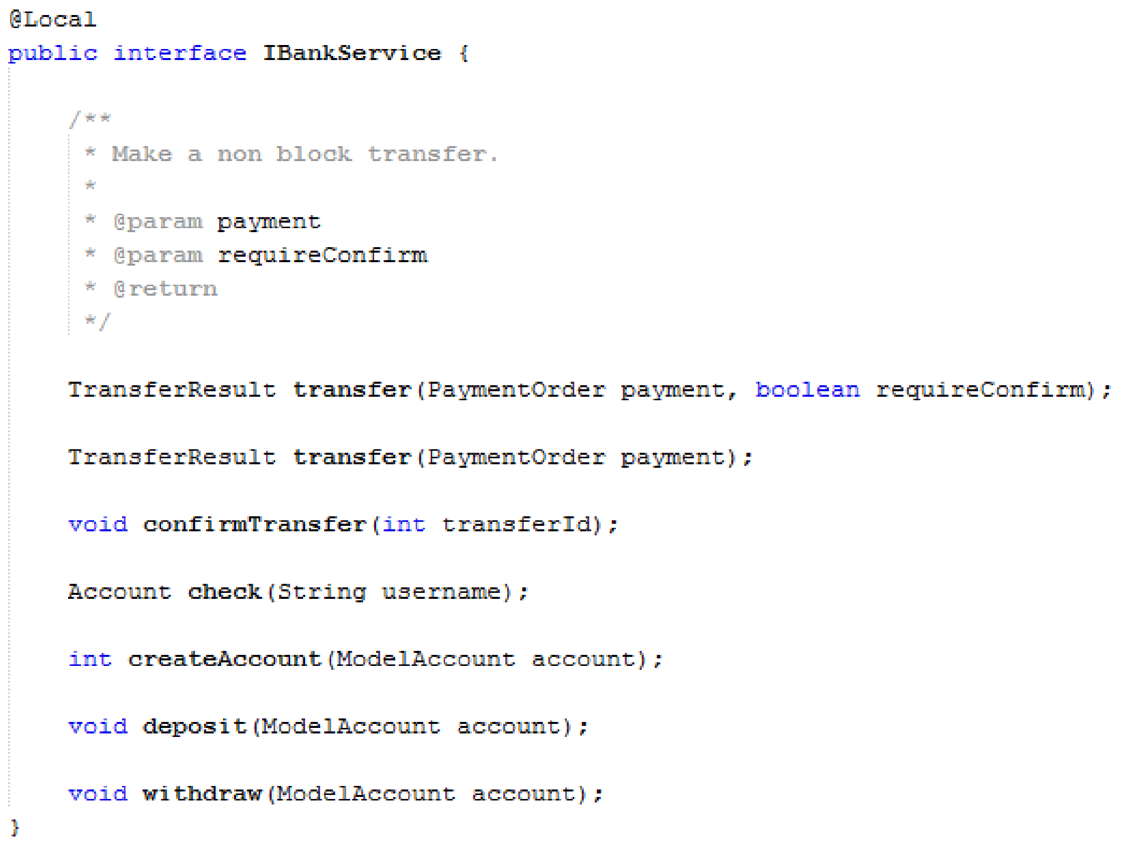


```java
@Override
public void sendEmail(String sendTo, String subject, String body) {
    MimeMessage msg = new MimeMessage(mailSession);
    try {
        msg.setSubject(subject);
        msg.setText(body);
        msg.setFrom(new InternetAddress(mailSession.getProperty("__sender__")));
        msg.setRecipient(Message.RecipientType.TO, new InternetAddress(sendTo));
        msg.setSentDate(new Date());
        Transport.send(msg);
    } catch (MessagingException ex) {
        Logger.getLogger(EmailService.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

Picture 4-30: Email Service Class

4.3.4 Bank

The last service that Identity Server offers is the Bank service. Even though its main job is to properly transfer money from User A to User B, it also provides services such as, create a user account, check an account, make a deposit and make a withdrawal. The next Java interface illustrates these operations.



```java
@Local
public interface IBankService {

    /**
     * Make a non block transfer.
     *
     * @param payment
     * @param requireConfirm
     * @return
     */

    TransferResult transfer(PaymentOrder payment, boolean requireConfirm);

    TransferResult transfer(PaymentOrder payment);

    void confirmTransfer(int transferId);

    Account check(String username);

    int createAccount(ModelAccount account);

    void deposit(ModelAccount account);

    void withdraw(ModelAccount account);
}
```

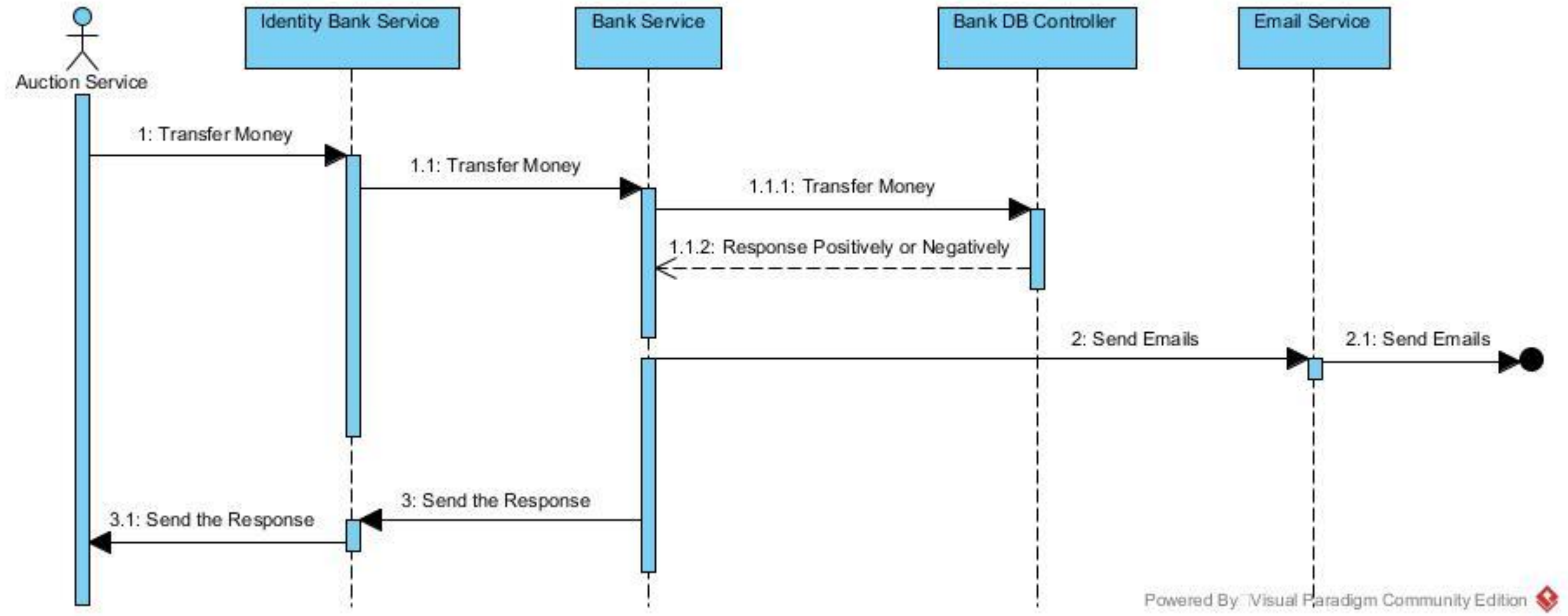
Picture 4-31: Bank Service Operations

As we have marked earlier, when a “transfer” request comes to the REST Bank Service, the money that is being withdrawn from the winner of the auction is put in a “BLOCKED” state, until the package transfer procedure is completed. At that point of time, a “confirm” request reaches the Bank Service in order to confirm that the transfer transaction is now legitimate to be accomplished, the money is unblocked and finally is deposited in auctioneer’s account.

As has been done in the Postal Service, we are going to introduce a business use case with all the steps starting from receiving a request by Auction Server, and ending with a positive or negative response message.

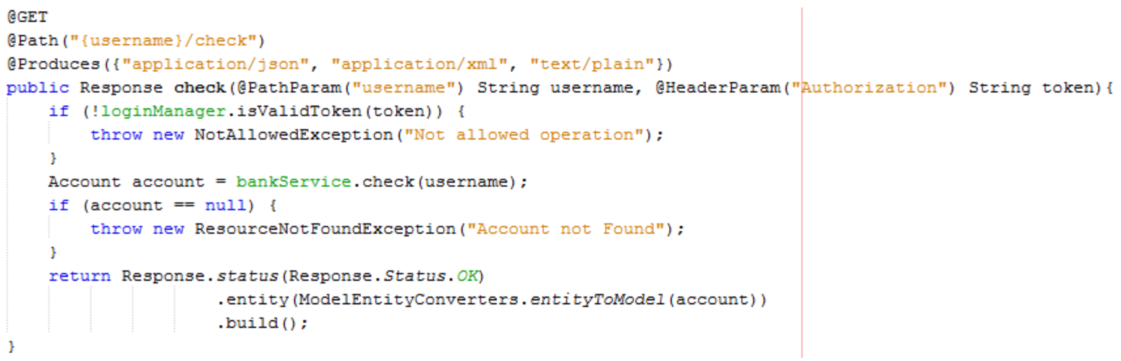
- Step 1: Auction Service sends a request to Identity Bank Service for transferring a certain amount of money from User A to User B.
- Step 2: Identity Bank Service (REST) asks Bank Service (Business Layer) to carry out the transaction.
- Step 3: Bank Service sends the request to Bank DB Controller.
- Step 4: Bank DB Controller checks whether User A has enough money in his account to successfully accomplish the transaction, and sends back the corresponding message.
- Step 5: Bank Service sends emails to stakeholders informing them about the result of the transaction.
- Step 6: Bank Service sends the aforementioned response to Identity Service.
- Step 7: Identity Service sends the response to Auction Service.

It follows the sequence diagram which illustrates the above use case.



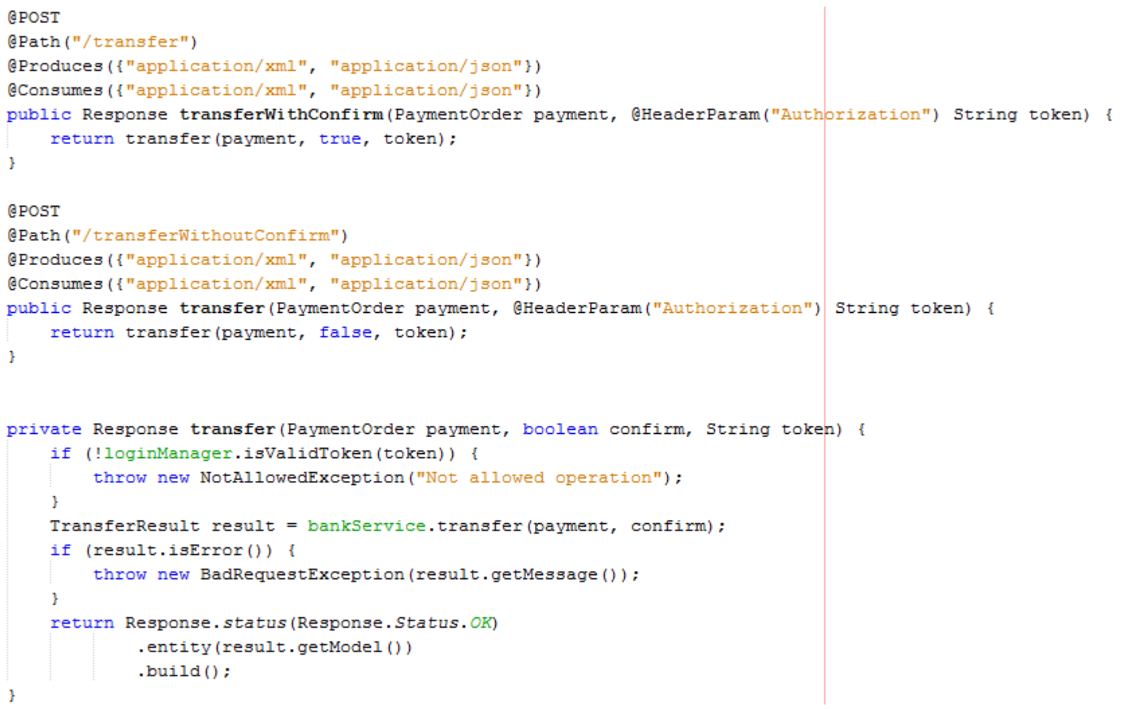
Picture 4-32: Bank Sequence Diagram

To conclude, in the next two figures, we display portion of the Java implementation of the Bank REST Service.



```java
@GET
@Path("{username}/check")
@Produces({"application/json", "application/xml", "text/plain"})
public Response check(@PathParam("username") String username, @HeaderParam("Authorization") String token){
    if (!loginManager.isValidToken(token)) {
        throw new NotAllowedException("Not allowed operation");
    }
    Account account = bankService.check(username);
    if (account == null) {
        throw new ResourceNotFoundException("Account not Found");
    }
    return Response.status(Response.Status.OK)
                    .entity(ModelEntityConverters.entityToModel(account))
                    .build();
}
```

Picture 4-33: Check Account



```java
@POST
@Path("/transfer")
@Produces({"application/xml", "application/json"})
@Consumes({"application/xml", "application/json"})
public Response transferWithConfirm(PaymentOrder payment, @HeaderParam("Authorization") String token) {
    return transfer(payment, true, token);
}

@POST
@Path("/transferWithoutConfirm")
@Produces({"application/xml", "application/json"})
@Consumes({"application/xml", "application/json"})
public Response transfer(PaymentOrder payment, @HeaderParam("Authorization") String token) {
    return transfer(payment, false, token);
}


private Response transfer(PaymentOrder payment, boolean confirm, String token) {
    if (!loginManager.isValidToken(token)) {
        throw new NotAllowedException("Not allowed operation");
    }
    TransferResult result = bankService.transfer(payment, confirm);
    if (result.isError()) {
        throw new BadRequestException(result.getMessage());
    }
    return Response.status(Response.Status.OK)
            .entity(result.getModel())
            .build();
}
```

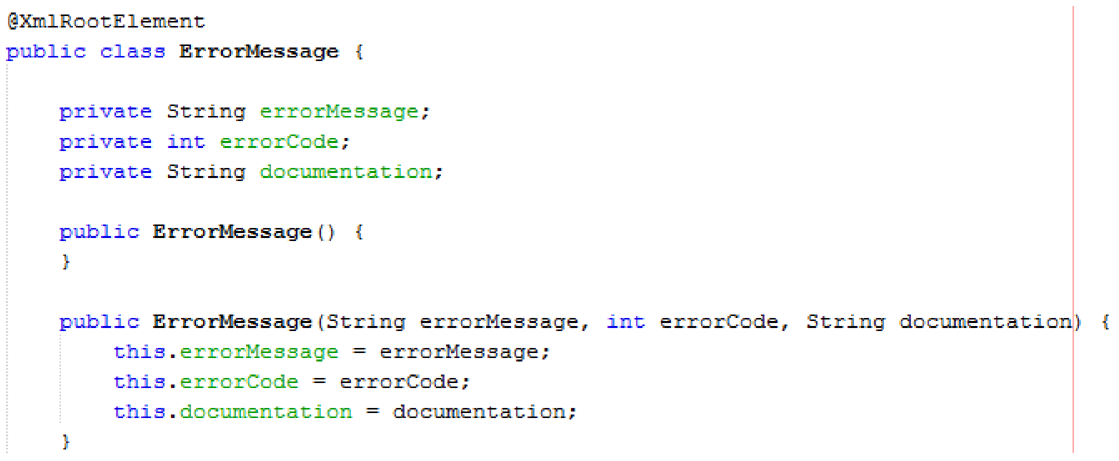
Picture 4-34: Transfer Money

4.4 Handling Exceptions

So far, we discussed about the happy path scenario, meaning that everything works as expected. We know what response will be returned, what the status code is going to be, etc. The important question that must be answered though, is what you as a developer want to send back as a response, when an exception in thrown. For this scenario, a helpful way to approach error handling in RESTful APIs is to create an error message that will be sent to the client in a nice XML or JSON format, and not allow some random HTML page that the GlassFish server throws up to be displayed.

We created two kind of exceptions for this API. The first one for database-related exceptions, and the second one for all the other scenario exceptions. For practical reasons, we will explain only one type of exception, the *NotAllowedException*, type which was randomly chosen. This exception is thrown when a client is not authenticated to access protected resources. All the other exceptions were built in the same way and work similarly.

The first step was to prepare the error message that would be sent. This message includes information about the cause of the exception, an error code, as well as additional guidelines about how to troubleshoot that exception. A part of this ErrorMessage class is displayed in the following figure.



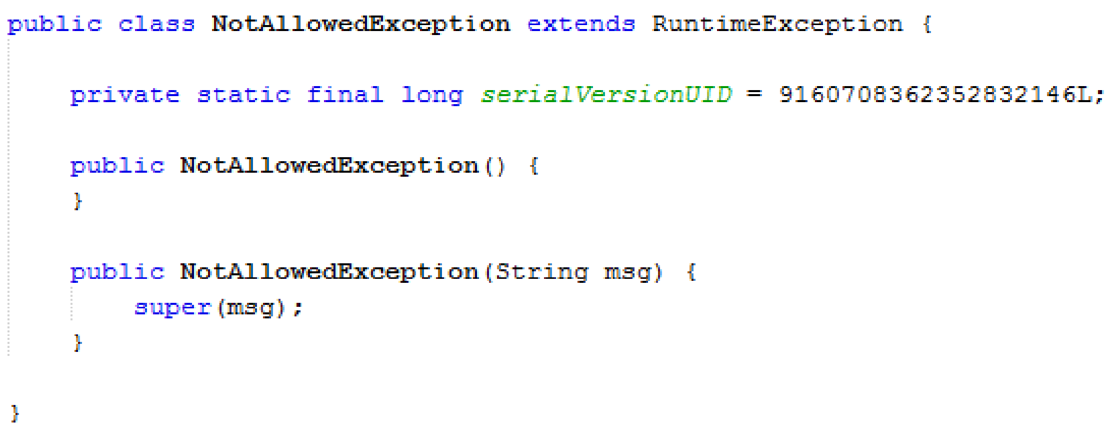
```
@XmlRootElement
public class ErrorMessage {

    private String errorMessage;
    private int errorCode;
    private String documentation;

    public ErrorMessage() {
    }

    public ErrorMessage(String errorMessage, int errorCode, String documentation) {
        this.errorMessage = errorMessage;
        this.errorCode = errorCode;
        this.documentation = documentation;
    }
```

Picture 4-35: Part of the ErrorMessage Class

The class in annotated with *@XmlRootElement* in order for the message to be converted to either XML or JSON. The class comprises three instance variables, two constructors, plus the corresponding “getters” and “setters”.

The second step was to map this error message to an exception. In other words, when some particular exception is thrown, concrete respond must be returned. In essence, we are mapping an exception to a response.

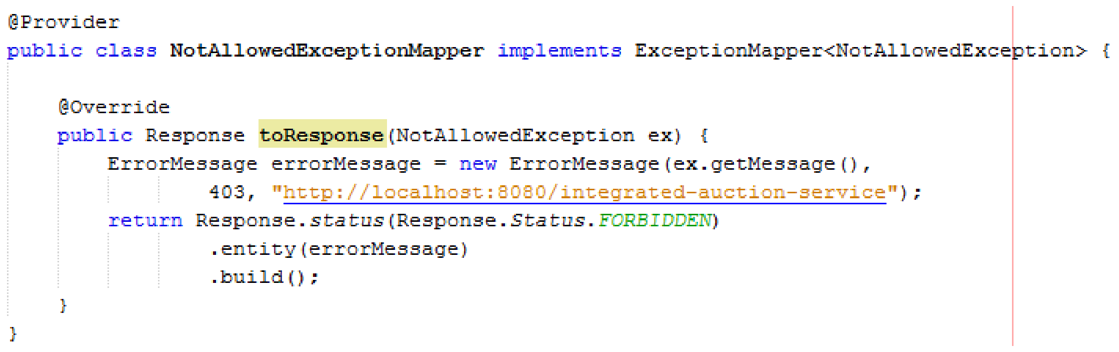
In order to fulfill the second step, we had to create the class *NotAllowedException*, which extends the class *RuntimeException*. It is a simply class with just two constructors.



```java
public class NotAllowedException extends RuntimeException {

    private static final long serialVersionUID = 9160708362352832146L;

    public NotAllowedException() {
    }

    public NotAllowedException(String msg) {
        super(msg);
    }

}
```

Picture 4-36: NotAllowedException Class

After that, the only thing that was left to do was to map this exception to the appropriate response. The way that this mapping can be done in JAX-RS is by using a Generic interface called *ExceptionMapper*. Hence, we needed to create one more class, which we called *NotAllowedExceptionMapper* that implements the ExceptionMapper interface. The Generic type of the interface, must be the exception that you want this exception mapper to map. In this case, the exception is the NotAllowedException. This interface has only one method that must be overridden. This method is called *toResponse*, takes as an argument the exception you want to throw, and returns a Response. The next figure displays the exact implementation of the NotAllowedExceptionMapper class.

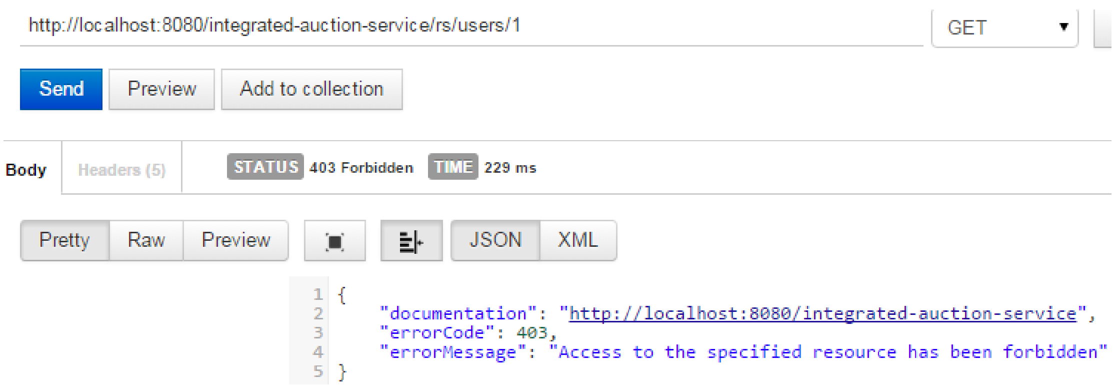


```java
@Provider
public class NotAllowedExceptionMapper implements ExceptionMapper<NotAllowedException> {

    @Override
    public Response toResponse(NotAllowedException ex) {
        ErrorMessage errorMessage = new ErrorMessage(ex.getMessage(),
                403, "http://localhost:8080/integrated-auction-service");
        return Response.status(Response.Status.FORBIDDEN)
                .entity(errorMessage)
                .build();
    }
}
```

Picture 4-37: NotAllowedExceptionMapper Class

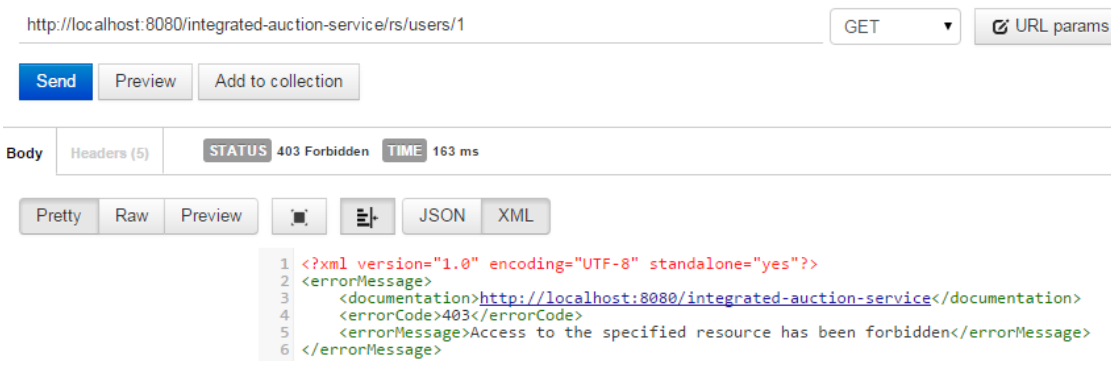
In the body of that class, we actually wrote code in order to create the specific response that would be send back to a client when that exception is thrown. The last thing to do was to annotate the class with *@Provider* annotation. This annotation registers the class in JAX-RS, so as JAX-RS to know about its presence.

To conclude, let us demonstrate what will be the response returned to a client when the NotAllowedException is thrown.



```
{
    "documentation": "http://localhost:8080/integrated-auction-service",
    "errorCode": 403,
    "errorMessage": "Access to the specified resource has been forbidden"
}
```

Picture 4-38: Response Message in JSON



```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<errorMessage>
    <documentation>http://localhost:8080/integrated-auction-service</documentation>
    <errorCode>403</errorCode>
    <errorMessage>Access to the specified resource has been forbidden</errorMessage>
</errorMessage>
```

Picture 4-39: Response Message in XML

In both cases, a non-authenticated client tried to access protected resources, and Jersey returned a message with an error code, the cause of the problem, as well as a documentation link for troubleshooting.

5 Implementation (Auction Server)

Having implemented all the Identity Server's services in the previous chapter, alongside with all the implementation details that exist behind the scenes, little are the worth-mentioning topics that remain to be included in this chapter.

5.1 Auction Server

In this chapter, we are going to touch on the Auction Service responsibilities together with the development of a RESTful Web Service Client, explaining why it is so much of importance to our RESTful API.

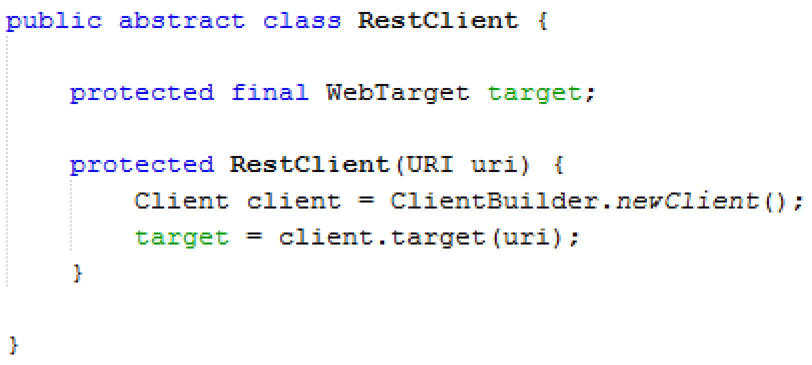
5.1.1 RESTful Client

In this section, we introduce the concept of a RESTful Client and attempt to explain why it is so useful, both in RESTful web services in general, but also in our API. More specifically, we deal with the JAX-RS client API, which is a Java based API that makes it particularly easy for RESTful web services, exposed via HTTP protocol, to be consumed. The goal of a client API is threefold:

- Wrap a key constraint of the REST architectural style, that is, the Uniform Interface Constraint, and map data elements as client-side Java objects.
- Facilitates the consumption of RESTful web services exposed over HTTP, just like the JAX-RS server-side API facilitates the development of RESTful web services.
- Share mutual concepts and design theory of the JAX-RS API between the server and the client side programming models [(21)].

In our case, client API is extremely serviceable and useful, since it is a vital part of Auction Server, which is presented in the next section. In particular, each and every request that reaches Auction Server must be checked for its validity by Identity Server. In order for this process to be seamlessly, without awareness of the outer world, Auction Server uses a client API to send requests to Identity Server, in order for the two systems to communicate internally. As a result, if some client sends request to the Auction Server asking, for instance, to open a new auction for his behalf, Auction Server uses client API to check the token validity, and then Auction replies to external client properly. This audit is happening automatically, without the aforementioned client knowing about this procedure. The only thing that the client gets, is a negative or a positive response concerning his initial request.

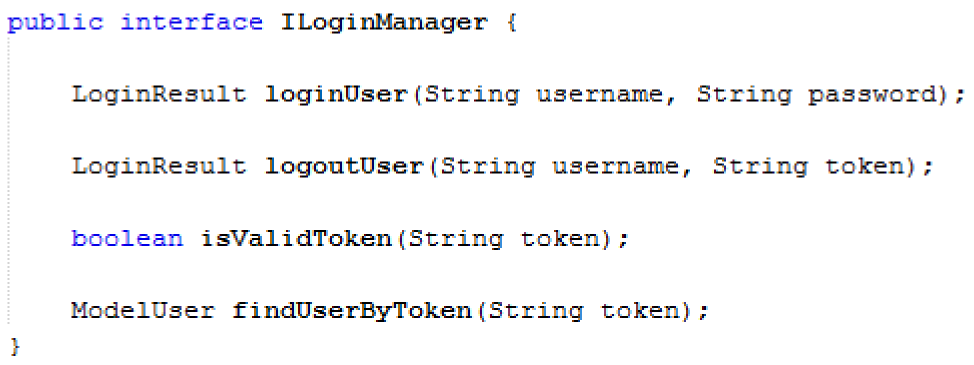
In the rest of this section, we attempt to give an insight into the client API and how it works. First of all, a resource in the JAX-RS client API is represented by an instance of a Java class called *WebTarget*, which encapsulates a URI. All HTTP verbs can be invoked based on that Java class [(21)]. However, in order to utilize the client API, it is firstly necessary to build an instance of a *Client* using a Java abstract class called *ClientBuilder*. Hence, in our API we created an abstract class called *RestClient*, to facilitate the creation of a RESTful client. The figure below illustrates this class.



```
public abstract class RestClient {

    protected final WebTarget target;

    protected RestClient(URI uri) {
        Client client = ClientBuilder.newClient();
        target = client.target(uri);
    }

}
```

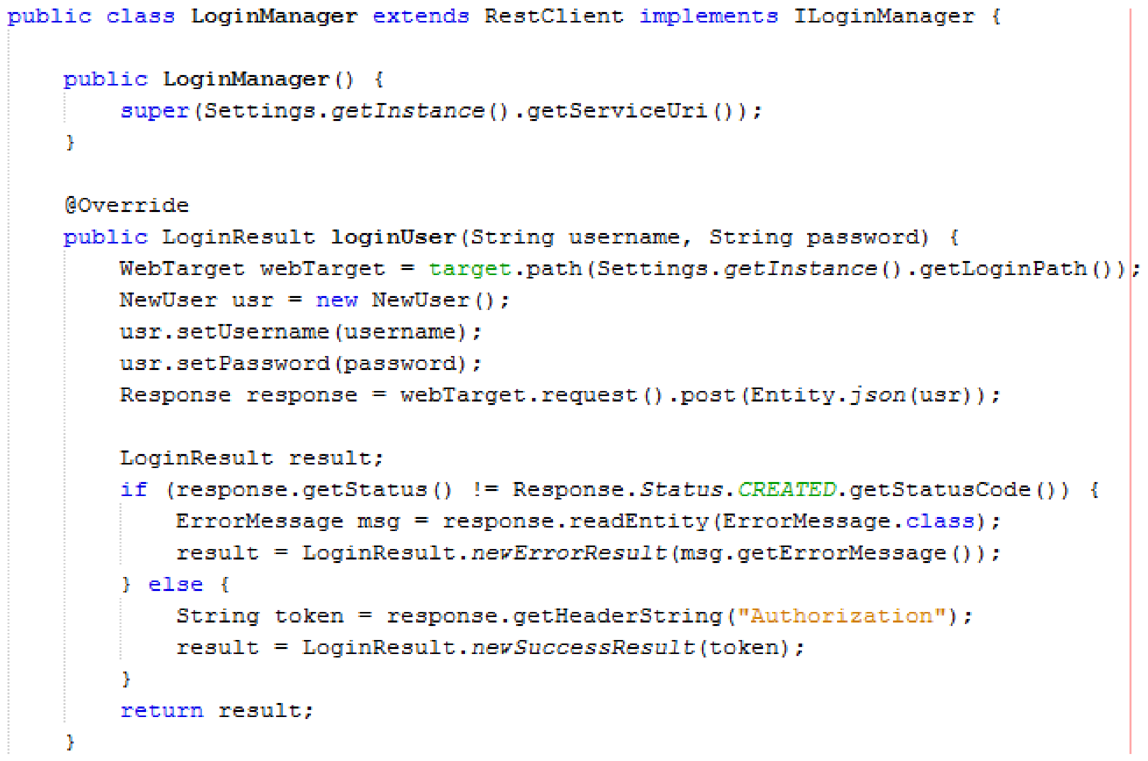
Picture 5-1: Rest Client Abstract Class

In its constructor it takes a URI, which is essentially the targeted resource. We instantiate a Client instance and then create a WebTarget from it. Moreover, as it is reasonable, our RESTful client has its own Login logic in order to communicate and be authenticated by Identity Server. The operations that is able to perform is given by the next Java interface, and the implementation by the corresponding class, which also *extends* the RestClient abstract class.



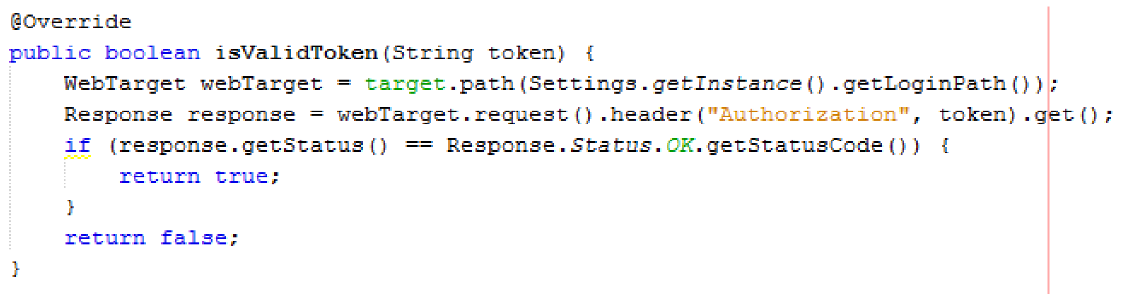
```
public interface ILoginManager {

    LoginResult loginUser(String username, String password);

    LoginResult logoutUser(String username, String token);

    boolean isValidToken(String token);

    ModelUser findUserByToken(String token);
}
```

Picture 5-2: Client Login Logic Interface



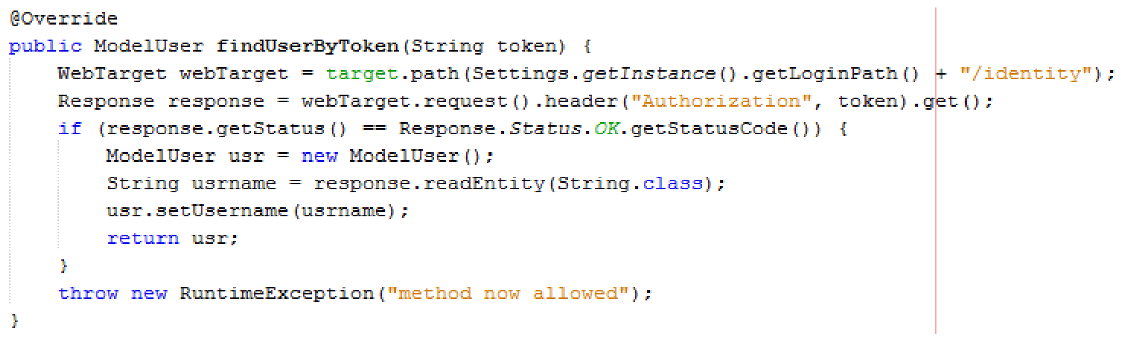
```
public class LoginManager extends RestClient implements ILoginManager {

    public LoginManager() {
        super(Settings.getInstance().getServiceUri());
    }

    @Override
    public LoginResult loginUser(String username, String password) {
        WebTarget webTarget = target.path(Settings.getInstance().getLoginPath());
        NewUser usr = new NewUser();
        usr.setUsername(username);
        usr.setPassword(password);
        Response response = webTarget.request().post(Entity.json(usr));

        LoginResult result;
        if (response.getStatus() != Response.Status.CREATED.getStatusCode()) {
            ErrorMessage msg = response.readEntity(ErrorMessage.class);
            result = LoginResult.newErrorResult(msg.getErrorMessage());
        } else {
            String token = response.getHeaderString("Authorization");
            result = LoginResult.newSuccessResult(token);
        }
        return result;
    }
```

Picture 5-3: Client Login Logic Implementation



```
@Override
public boolean isValidToken(String token) {
    WebTarget webTarget = target.path(Settings.getInstance().getLoginPath());
    Response response = webTarget.request().header("Authorization", token).get();
    if (response.getStatus() == Response.Status.OK.getStatusCode()) {
        return true;
    }
    return false;
}
```

Picture 5-4: Client Login Logic Implementation



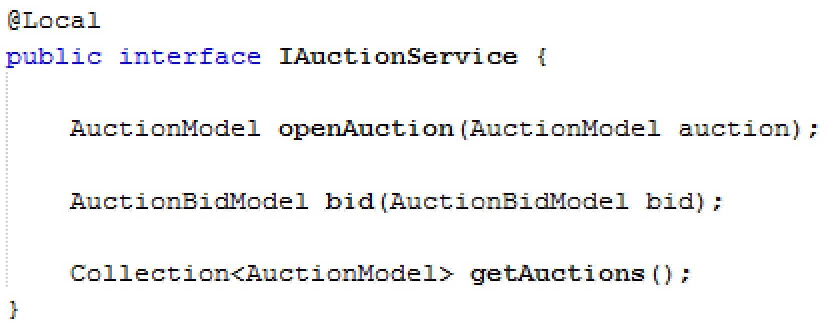
```java
@Override
public ModelUser findUserByToken(String token) {
    WebTarget webTarget = target.path(Settings.getInstance().getLoginPath() + "/identity");
    Response response = webTarget.request().header("Authorization", token).get();
    if (response.getStatus() == Response.Status.OK.getStatusCode()) {
        ModelUser usr = new ModelUser();
        String usrname = response.readEntity(String.class);
        usr.setUsername(usrname);
        return usr;
    }
    throw new RuntimeException("method now allowed");
}
```

Picture 5-5: Client Login Logic Implementation

In the next section, it is shown how exactly our RESTful client is used by the Auction Service, in order to send requests to Identity's Login REST service.

5.1.2 Auction Service

The role of Auction Server is highly targeted. It accepts request by clients, either for creating a new auction or for placing a new bid in an already running auction. The latter case also involves the ability of the Auction server to give to all the potential bidders, all the open auctions. The next Java interface illustrates these services.



```java
@Local
public interface IAuctionService {

    AuctionModel openAuction(AuctionModel auction);

    AuctionBidModel bid(AuctionBidModel bid);

    Collection<AuctionModel> getAuctions();
}
```

Picture 5-6: Auction Service Operations

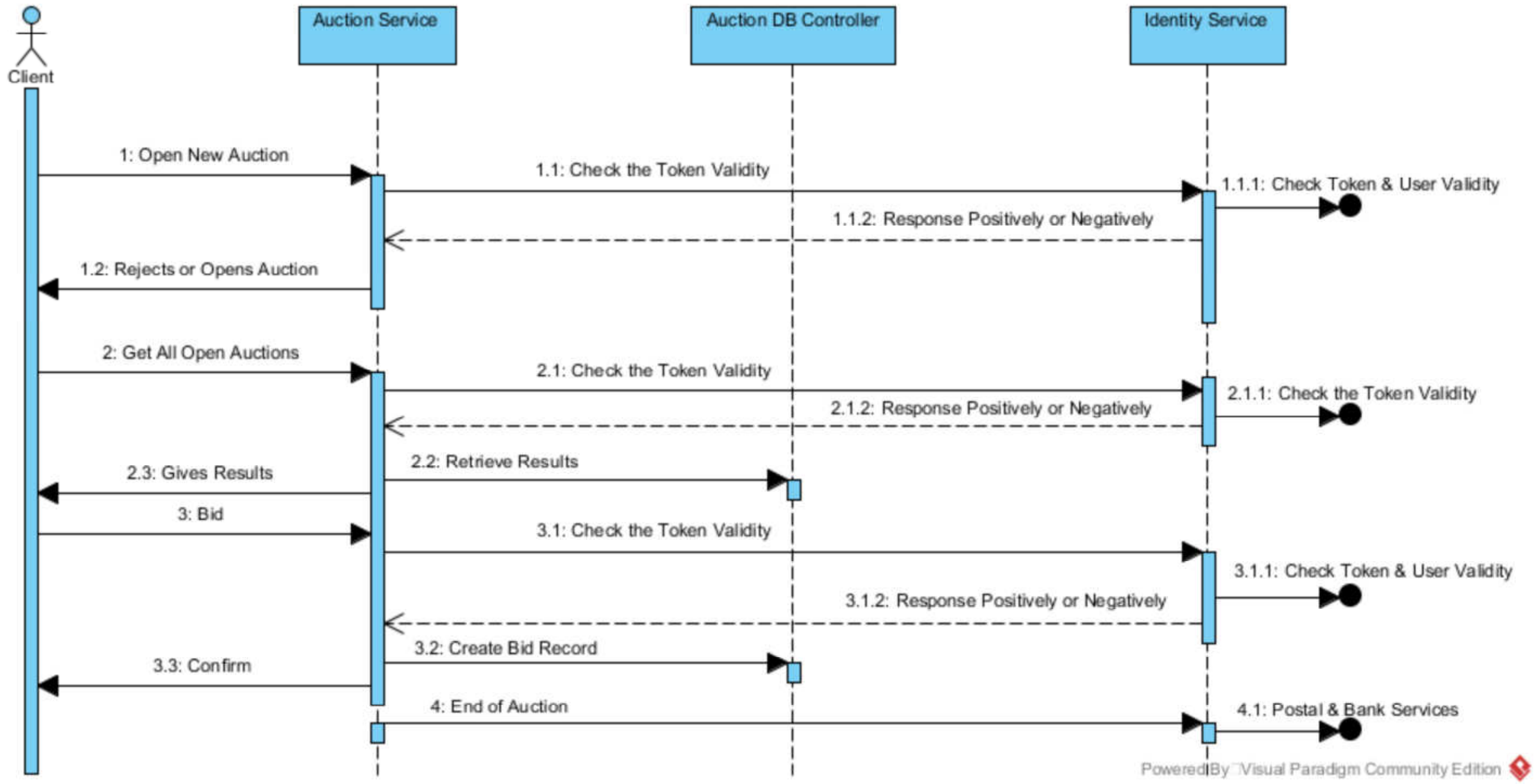
For better comprehension, we introduce yet another use case with all the steps involved, starting from receiving a request by a client, and ending with a positive or negative response message by Auction Server.

- Step 1: A client (auctioneer) sends a proper request to Auction Server to open (create) a new auction.
- Step 2: Auction Server checks the validity of the request's token by sending a subsequent request to Identity Server.

- Step 3: Identity Server checks the token as well as the user of that token, and replies accordingly.
- Step 4: If credentials do not match, Auction Server rejects the client’s request. If, however, everything is as it should, Auction Server opens a new auction on his behalf.
- Step 5: Consumers of the service (other clients) are able to send requests and scan all the open auctions (the available ones).
- Step 6: Legitimate clients bid for an auction product.
- Step 7: When an auction ends, Auction Server sends the appropriate requests to Identity Server, in order to take care the rest actions/services that are mandatory for the whole auction procedure to be completed.

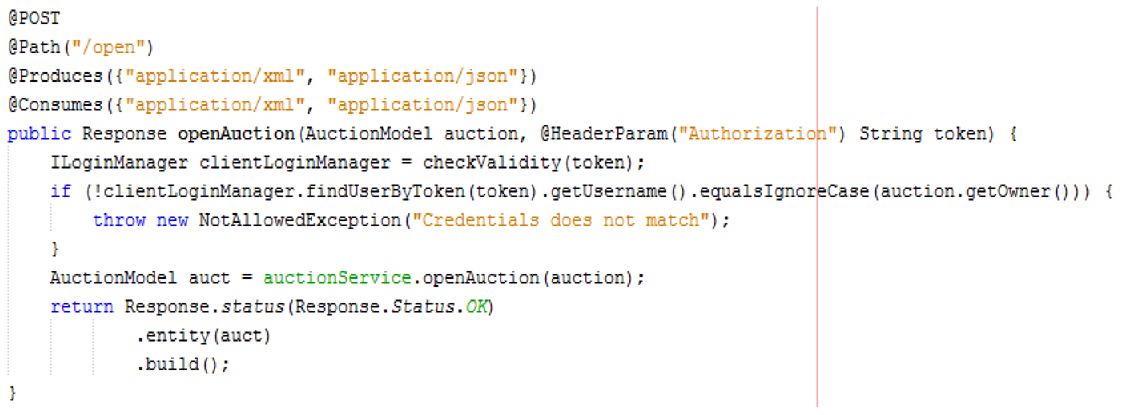
The following sequence diagram illustrates the above use case

.


Client
Auction Service
Auction DB Controller
Identity Service
1: Open New Auction
1.1: Check the Token Validity
1.1.1: Check Token & User Validity
1.1.2: Response Positively or Negatively
1.2: Rejects or Opens Auction
2: Get All Open Auctions
2.1: Check the Token Validity
2.1.1: Check the Token Validity
2.1.2: Response Positively or Negatively
2.2: Retrieve Results
2.3: Gives Results
3: Bid
3.1: Check the Token Validity
3.1.1: Check Token & User Validity
3.1.2: Response Positively or Negatively
3.2: Create Bid Record
3.3: Confirm
4: End of Auction
4.1: Postal & Bank Services
Powered By Visual Paradigm Community Edition

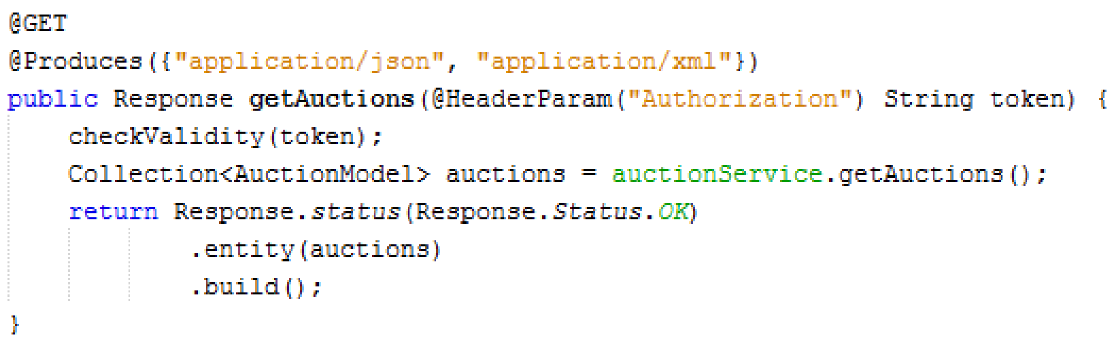
Picture 5-7: Auction Sequence Diagram

We will close this section by providing the Java implementation of the Auction REST Service. It is noteworthy how Auction REST Service uses the RESTful client in order to seamlessly validate user's credentials.



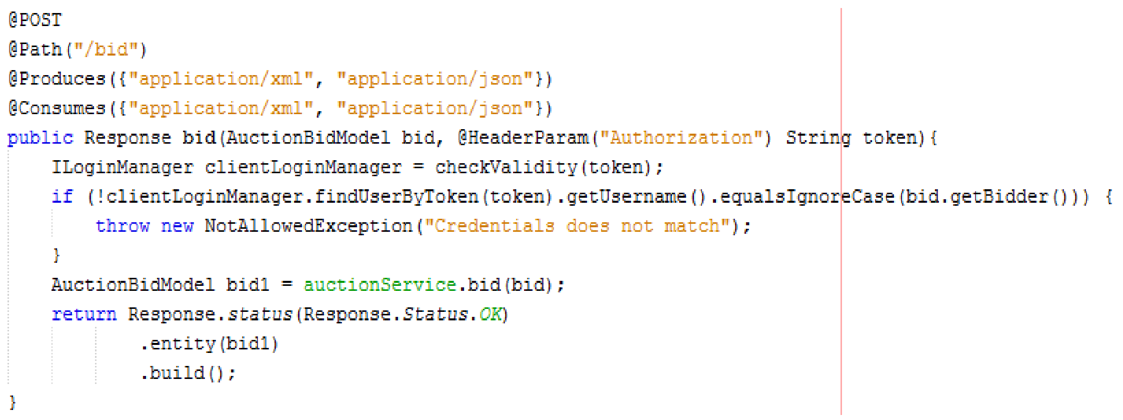
```java
@POST
@Path("/open")
@Produces({"application/xml", "application/json"})
@Consumes({"application/xml", "application/json"})
public Response openAuction(AuctionModel auction, @HeaderParam("Authorization") String token) {
    ILoginManager clientLoginManager = checkValidity(token);
    if (!clientLoginManager.findUserByToken(token).getUsername().equalsIgnoreCase(auction.getOwner())) {
        throw new NotAllowedException("Credentials does not match");
    }
    AuctionModel auct = auctionService.openAuction(auction);
    return Response.status(Response.Status.OK)
            .entity(auct)
            .build();
}
```

Picture 5-8: Open Auction



```java
@GET
@Produces({"application/json", "application/xml"})
public Response getAuctions(@HeaderParam("Authorization") String token) {
    checkValidity(token);
    Collection<AuctionModel> auctions = auctionService.getAuctions();
    return Response.status(Response.Status.OK)
            .entity(auctions)
            .build();
}
```

Picture 5-9: Get All Open Auctions



```java
@POST
@Path("/bid")
@Produces({"application/xml", "application/json"})
@Consumes({"application/xml", "application/json"})
public Response bid(AuctionBidModel bid, @HeaderParam("Authorization") String token){
    ILoginManager clientLoginManager = checkValidity(token);
    if (!clientLoginManager.findUserByToken(token).getUsername().equalsIgnoreCase(bid.getBidder())) {
        throw new NotAllowedException("Credentials does not match");
    }
    AuctionBidModel bid1 = auctionService.bid(bid);
    return Response.status(Response.Status.OK)
            .entity(bid1)
            .build();
}
```

Picture 5-10: Bid for an Auction Product

6 Conclusions

In this thesis we tried to develop an integrated web service, which could be a complementary part of an already deployed web application. We have used the JAX-RS specification, which is an API that provides support for creating web services according to the Representational State Transfer (REST) architectural pattern. The nice thing with JAX-RS is that it bridges the gap between the HTTP world and the Java world by mapping key terms of the RESTful architectural style, such as HTTP methods, with Java objects.

The reason why we chose to use the RESTful architectural style is due to the fact that we wanted to create a loose coupling web service, which could be easily become part of another web system, especially when this web application is regularly updated, while it is vital the consumers of these services keep using them insusceptible. Another attempt was to create a server-client communication, that is not so verbose as XML, thus, less overhead, in order to be effortlessly used in environments with limited bandwidth and resources.

Throughout the development of our RESTful web service, we made use of the whole Java Platform, Enterprise Edition 7, and its capabilities. The main point we have tried to make is that by using the Java platform and the tools that it provides, it is relatively easy for a developer to create from scratch a fully operational RESTful API.

The initial research was spent on evaluating and understanding the reasons why REST services is so much a trend nowadays, and why great companies like Google, Twitter, Facebook and Yahooprefer them. Consequently, the comparison between REST and SOAP web services was inevitably highlighted. It should be underlined that even though the development process encountered difficulties, the Java platform proved to be a valuable and reliable ally.

7 Bibliography

1. **microsoft.com.** *Chapter 1: Service Oriented Architecture.* s.l. : Msdn.microsoft.com.

2. **Thomas, Fielding Roy.** *Architectural Styles and the Design of Network-based Software Architectures.* s.l. : University of California, Irvine.

3. **James Snell, Doug Tidwell, Pavel Kulchenko.** *Programming Web Services with SOAP.* s.l. : O'Reilly Media, 2001.

4. **Francisco Curbera, William A. Nagy, Sanjiva Weerawarana.** *Web Services: Why and How.* s.l. : IBM T.J. Watson Research Center, 2001.

5. **http://www.w3schools.com/.** http://www.w3schools.com/. [Online]

6. **Richardson, Leonard and Ruby, Sam.** *RESTful Web Services.* s.l. : O'Reilly Media, 2007. 978-0-596-52926-0.

7. **http://www.w3.org/DesignIssues/Axioms.** http://www.w3.org/DesignIssues/Axioms. [Online]

8. **Allamaraju, Subbu.** *RESTful Web Services Cookbook,Solutions for Improving Scalability and Simplicity.* s.l. : O'Reilly Media, 2010.

9. **Fielding, Roy T., et al.** *Hypertext Transfer Protocol -- HTTP/1.1. IETF.* . 1999. RFC 2616.

10. **http://www.w3.org.** http://www.w3.org. [Online]

11. **Hewgill, Greg.** www.stackoverflow.com. [Online]

12. **http://restcookbook.com/Miscellaneous/richardsonmaturitymodel/.** *http://restcookbook.com/Miscellaneous/richardsonmaturitymodel/.* [Online]

13. **http://martinfowler.com/articles/richardsonMaturityModel.html.** *http://martinfowler.com/articles/richardsonMaturityModel.html.* [Online]

14. **http://www.bu.edu/tech/about/security-resources/bestpractice/auth/.** *http://www.bu.edu/tech/about/security-resources/bestpractice/auth/.* [Online]

15. **Jim Webber, Savas Parastatidis, Ian Robinson.** *REST in Practice, Hypermedia and Systems Architecture.* s.l. : O'Reilly Media, 2010.

16. **http://docs.oracle.com/.** http://docs.oracle.com/. [Online]

17. **Kleinberg, David Easley and Jon.** *Networks, Crowds, and Markets: Reasoning about a Highly Connected World.* s.l. : Cambridge University Press, 2010.

18. **McAfee, Dinesh Satam and McMillan, Dinesh.** *“Auctions and Bidding”, Journal of Economic Literature* . s.l. : American Economic Association, 1987.

19. **Klemperer, Paul.** *Auctions: Theory and Practice.* s.l. : Princeton University Press.

20. **Adam, M.T.P., et al.** *"Understanding auction fever: A framework for emotional bidding".* 2011.

21. **https://jersey.java.net/.** https://jersey.java.net/. [Online]

8 Appendix

8.1 Documentation

Service name: integrated-auction-service

Root URI: http://localhost:8080/integrated-auction-service/rs/

Resource	Methods	URI	Description
Users	GET	*/users/{userID}*	/{userID} is optional. /{from}/{length} is optional /count is optional Format: application/json, application/xml, text/plain
	POST	/users/*{from}/{length}*	GET: retrieve all users or a specific one
	PUT	/users/*count*	POST: create a new user
	DELETE		PUT: update the specified user. (use of: /{userID})
			DELETE: delete the specified user. (use of: /{userID})
			GET: retrieve users with range (use of: /{from}/{length})
			GET: count all the users (use of: /count)
User token	POST	*/login/identity*	/identity is optional
	GET		POST: provide token for legitimate users. (User sends his username & password and if they are valid, a token is returned)
			GET: check if a user is authenticated
			GET: return user by token (use of: /identity)

Package, Package state	POST GET	/postal/send /postal/{packageId}/check	Format: application/json, applica-tion/xml
			POST: send a package from user A to user B. (use of: /send)
			GET: check the package state of a particular package. (use of: /{packageId}/check)
Account, Transfer transac-tion	POST GET	/bank/transfer /bank/{username}/checkAccount /bank/create /bank/deposit /bank/withdraw /bank/{bankTransferId}/{packageId}/check	Format: application/json, applica-tion/xml, text/plain
			POST: transfer money from Ac-count A to Account B. (use of: /transfer)
			GET: check an account. (use of: /{username}/checkAccount)
			POST: create an account. (use of: /create)
			POST: deposit money. (use of: /deposit)
			GET: check if the specified package has delivered. If so, unblock the money in the specified bank trans-action. (use of: /{bankTransferId}/{packageId}/check)
Auction, Auction bid	POST GET	/auction/open /auction/bid /auction	POST: withdraw money. (use of: /withdraw)
			Format: application/json, applica-tion/xml
			POST: open a new auction. (use of: /open)
			POST: place a bid. (use of: /bid)
			GET: get all the open auctions

Table 5: Integrated-Auction-Service API Documentation

8.2 Source Code

The complete source code of our RESTful API is available at:

https://github.com/yiannis13/thesis-repo

8.3 Tools & Technologies

To recapitulate, the main Java related tools and technologies that made this RESTful API feasible are as follows:

- The chosen Integrated Development Environment (IDE), in order to simplify the whole development process, was *NetBeans*. It could be argued that this is the official Java IDE, since it is provided by Oracle Corporation, the same company that runs Java. It is free, open source and cross-platform.
- The ORM (Object Relational Mapping) tool that was selected in order to easily query and manipulate data from the database (MySQL in this case), was *JPA*. JPA is a standardized specification that helps a developer to build a DB layer that is independent of any particular persistence provider.
- *GlassFish* Server. Since web services is a kind of web application, it was needed a robust application server in order to deploy and run our RESTful API. GlassFish is open-source, free and is also sponsored by Oracle Corporation. It supports, among others, Enterprise JavaBeans for dependency injections and automatic database transaction capabilities, enabling developers to focus only on the business logic of an application. Literally speaking, everything runs inside GlassFish.
- *Maven* played a vital role in almost every step in this implementation. It managed the countless jars and libraries needed, by automatically downloading all the dependencies that were pointed out in the pom.xml file. Without Maven, the development process would be far more cumbersome.
- For the data storage we chose to use *MySQL*, because it is open source, is developed under Oracle Corporation, which is the same company that runs Java, and therefore, the communication between the Java programming language and MySQL database would be particularly easy and efficient. For making our live easier, we also used MySQL *Workbench*, which is a Graphical User Interface

(GUI) tool that provides SQL development, administration, database design, and maintenance for the MySQL RDBMS.

- For UML class diagrams, use case diagrams and sequence diagrams, we chose to use *StarUML* and *Visual Paradigm*.