Proposed Design and Implementation for RESTful Web Server

Mou'ath Hourani, Qusai Shambour, Ahmad Al-Zubidy and Ali Al-Smadi
Software Engineering Department, Faculty of Information Technology
Al-Ahliyya Amman University, PO Box 19328, Amman, Jordan
{mouath.hourani@ammanu.edu.jo}, {Q.Shambour@ammanu.edu.jo}{a.jonline@yahoo.com}, {ali.a-smadi@hotmail.com}

***Abstract*— In this paper, we propose a framework to design and implement a novel RESTful Web server architecture. The proposed RESTful Web server architecture is a lightweight server that will be less taxing on system resources and will therefore handle a greater volume of requests. Furthermore, in contrast to the usual servers that supports most websites, the proposed RESTful Web server architecture is used for hosting RESTful-based Web Services. Moreover, the results obtained from our experiments show that the proposed architecture demonstrates high performance while maintaining proper stability. Our Web server is programmed in Java using a simple yet flexible web application framework that fulfills the needs of modern web application development. It also provides developers with a rapid and cost-effective method for implementing, deploying and serving a web API (Application Programming Interface).**

***Index Terms*—Service-Oriented Architecture (SOA), REST, RESTful, Web Server, Web Services, Web API.**

I. INTRODUCTION

Software architecture is an abstraction of the run-time elements of a software system. It can be defined by the configuration of its elements. Such elements (components, connectors, and data) are constrained in their function and relationships in order to achieve a required set of architectural properties (e.g., scalability, reliability, reusability) [1]. A coordinated set of such architectural constraints is called an architectural style [1].

Web service based applications have been widely applied in a variety of domains with the development of Service-oriented architecture (SOA) [2, 3]. SOA is an architectural style that guides all aspects of creating and using services throughout their lifecycle, as well as defining and providing the infrastructure that allows heterogeneous applications to exchange data. This communication usually involves the involvement in business processes, which are loosely coupled to their underlying implementations. SOA represents a model in which functionality is decomposed into separate units (services) that can be spread over a network and can be united together and reused to create business applications [4]. SOA allows the creation of systems using reusable components with well-defined service interfaces, these components can be published as discoverable services over the Internet based on their capabilities [5]. Ninety-two percent of companies say their SOA initiatives met or exceeded business unit objectives, while only eight percent say they did not. Additionally, SOA market is growing 17% a year to reach $10 billion by 2015 [6].

Currently, two architectural styles are commonly discussed in the context of SOA: firstly, the Simple Object-Access Protocol (SOAP) styles and related standards (e.g., WSDL). Secondly, styles based on the Representational State Transfer (REST) with loosely coupled designs similar to resources of the World Wide Web. Although the REST vs. SOAP debate is mostly ignored in academia, the SOA community is still arguing about the pros and cons of each style [1, 7-9]. However, REST's simplicity, beside its natural fit over HTTP, has contributed to its status as the best method to achieve a desired result for Web 2.0 applications in terms of exposing their data [10].

The REST style architecture is a main contributor to the Web's success. REST describes how the web as a large scale distributed hypermedia systems, have to operate to make the most of beneficial properties, including scalability, modifiability, performance, simplicity, and reliability. To retain usability in the face of increasing growth and expansion into new domains, the Web must maintain the benefits of the RESTful design. However, even though REST principles have been known for more than a decade, developing systems that conform to them is difficult [11].

In this paper, we propose a framework architecture to design and implement a RESTful Web server that conforms to REST constraints/principles. The proposed RESTful Web server as a lightweight server will be able to handle more requests since it is less taxing on system resources, and provides high performance, and stability. This paper is organized as follows. Section 2 provides background information on REST style architecture and an overview of the most popular Web servers. In Section 3, we propose our RESTful Web server including its structure, requirements and components. Section 4 illustrates the evaluation and results. Finally, conclusions and directions for future study are provided in Section 5.

II. BACKGROUND AND LITERATURE REVIEW

doi:10.4304/jsw.9.5.1071-1080

This section reviews literature related to this study. First, an overview of the REST style architecture is presented. We then provide a review of its benefits. Finally, an overview of the most popular Web servers is presented.

A. REST-Style Architecture

REST style architecture consists of clients and servers. Clients initiate requests and servers process them and return responses. Requests and responses are built around the transfer of resources' representations [1, 12]. A resource can be any meaningful information that can be named. A representation of a resource is a document that captures the intended state of a resource [1, 12]. REST style architecture addresses four main goals [1, 12]:

- Scalability of component interactions;
- Generality of interfaces;
- Independent deployment of components; and
- Intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.

REST achieves these goals by implementing the following principles [1, 12-14], which are:

1. Client-Server

Separation of concerns is the principle behind the client-server constraints in which clients are separated from servers by a uniform interface. By separating the user interface concerns from the data storage concerns, the portability of the user interface across multiple platforms is improved. Also, this improves the servers' scalability.

2. Stateless

The concept of stateless means that the client-server communication is constrained by no client context being stored on the server between requests. Each client request has to be fully self-descriptive, is considered in isolation, and is interpreted only in context of the current resource state. Application state is maintained by the clients. This makes servers more visible for monitoring and more reliable in the face of network failures.

3. Cacheable

Clients are able to cache responses in order to improve network efficiency. Cache constraints require that the data within a response to a request be implicitly or explicitly defined as cacheable or non-cacheable to prevent clients reusing inappropriate data in response to further requests. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests. This constraint improves scalability and performance.

4. Layered system

The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond the immediate layer with which they are interacting. Intermediaries can also be used to improve system scalability by enabling load balancing of services across multiple networks and processors. They may also enforce security policies.

5. Uniform Interface

The concept of the Uniform Interface explicates that all communication between client and the server is conducted using the fixed operation set provided by HTTP: GET, PUT, POST, DELETE. A GET method retrieves the current representation of the requested resource from the server to the client. The PUT method allows the client to change the state of the resource by modifying its representation and transferring it back to the server. The POST method creates a resource on the server. DELETE removes a resource from the server.

Fulfilling these REST style architectural constraints will enable any kind of distributed hypermedia system to have desirable evolving properties, such as performance, scalability, simplicity, modifiability, visibility, portability and reliability. In addition, conforming to the above constraints is generally referred to as being "RESTful". If a service violates any of the required constraints, it cannot be considered RESTful [1, 12-14].

B. Benefits of the REST-Style Architecture

The main benefits of REST style architecture are [15-19]:

- The REST it is a simple yet effective technology that can be used to create web services. Without needing any toolkit, developers need to know the Hypertext Transfer Protocol (HTTP), HyperText Markup Language (HTML) and Extensible Markup Language (XML) in order to implement web services.
- Also, using standard formats as HTML or XML ensures compatibility over time.
- Requests and responses through the REST interface can be short. Thus, in terms of bandwidth usage, REST is light.
- REST developers can easily create and modify an URI to access different Web resources.
- With the support for caching, REST provides improved response times and server loadings due to the totally stateless operation of the REST approach.
- By reducing the need to maintain communication state, REST improves server scalability. This means that initial and subsequent requests can be handled in different servers.
- Since a single browser can access any resource and application, REST demands less client-side software to be written than other approaches.
- With the use of hyperlinks in content, a separate resource discovery mechanism is not needed.
- Less memory consumption than others.
- Possibility to distribute queries across more than one server.

In terms of testing and troubleshooting. It is easy to test and troubleshoot an HTTP REST API since one can construct a call with nothing more than a browser and check the response inside the browser window itself. No

troubleshooting tools are required to generate a request / response cycle.

C. Related Literature: Most Popular Web Servers

According to the Septemper 2013 Netcraft Web Server Survey (http://news.netcraft.com/archives/category/web-server-survey/), three servers are currently widely used on the Web: the Apache Server from the Apache Software Foundation (http://httpd.apache.org/), Internet Information Server (IIS) from Microsoft (http://www.iis.net/) and the Nginx Web Server (http://nginx.org/). Apache is the first most popular Web Server in the world with a market share of 47%. IIS is the second most popular Web Server with a market share of 22%, whereas the Nginx come in the third place with a market share of 15%. However, this doesn't automatically mean that Apache is better than other Web servers. A reason might be that Apache has been around longer than others, giving Apache an edge since it's already been integrated into a lot of systems.

Apache is the most popular Web server and one of the most successful open-source projects of all time. Apache has proven to be a very stable, scalable, secure, cross-platform, and flexible Web server in which it facilitates interface customization. However, its configuration process is complicated and requires profound special knowledge. Additionally, there are Web servers that are more lightweight and faster than Apache. Although Apache scales well and can handle high loads, other Web servers might be more appropriate for serving pure static content [20, 21].

Unlike Apache which is a free open-source Web server, Internet Information Server (IIS) is a Web server created by Microsoft in which it comes with the server versions of Windows and cannot be downloaded separately. IIS has a modular architecture. Modules, also called extensions, can be added or removed individually so that only modules required for specific functionality have to be installed. IIS includes native modules as part of the full installation. These modules are individual features that the server uses to process requests. In addition, IIS includes enhanced security features. These features are: client certificate mapping, IP security, request filtering and URL authorization. However, IIS runs only on Windows unlike Apache which runs on almost every operating system [22].

Although Apache is an excellent open-source Web server, Nginx Web server can be considered as an alternative with the same functionality, a simpler configuration, better performance and efficiency. Nginx is a high-performance Web server and reverse proxy designed to use very few system resources. Nginx was first conceived to be an HTTP server. It was created by Igor Sysoev to solve the C10K problem, described by Daniel Kegel at http://www.kegel.com/c10k.html, of designing a Web server to handle 10,000 simultaneous connections. Nginx is able to do this through its event-based connection-handling mechanism, and will use the OS-appropriate event mechanism in order to achieve this goal. Like Apache, Nginx is used by some of the largest Web sites in the US, including WordPress, Hulu and MochiMedia. Nginx is the third-most-popular Web server, and it is currently serving more than 112 million Web sites [23, 24].

III. The Proposed RESTful Web Server Architecture

Although there are many widely available RESTful web service API's in java and other programming languages (e.g, Jersey, JBoss and Restlet), which eventually lead to RESTful web services complying with the main REST constraints [11, 13, 25, 26], however, (1) developing RESTful web services is still a key challenge due to the lack of software development frameworks that support all REST constraints; (2) most of the RESTful web services are typically expected to run over the same server which is used to power full blown websites, while our proposed server is a standalone RESTful Web server concerned with hosting RESTful web services created through our own APIs.

To overcome the above drawbacks, this paper proposes the design and implementation of a fully lightweight RESTful Web server that follows all the RESTful constraints and features, including:

- Provide a real 100% clean URL (Uniform Resource Locator).
- Dedicated to deploy and host only RESTful web services.
- Providing an auto generated Web Application Description Language (WADL) document for each service.
- Automatically provide the user with all the available resources for a given service.

A. Proposed Web Server Architecture

The proposed Web server uses a multi-threaded architecture, by utilizing threads to serve requests, this approach basically associates each incoming connection to be handled by a dedicated thread, while enabling the various threads to easily share data structures. The memory requirements for this type of architecture is relatively little, as the server initializes a dynamic thread pool at start up, the size of the pool varies with workload intensity. When load increases, so will the pool size, allowing more requests to be processed concurrently, leading to reduction of the incoming queue size. When the load is low, the number of threads reduces to free up memory.

B. Proposed Web Server Requirements

- Usability of the Server: The server must be easy to use, and not with a lot of configuration and complex operation, the server must do almost everything automatically, and with little or no skills required to manage it.
- Compliance: The system is tested to be complaint on the client-side with all the major browsers like Microsoft Internet Explorer 6+, Firefox 2+, Google Chrome 3+, Apple Safari 3+, and any client that support the HTTP 1.1.

- Maintainability: The system is highly structured and follows the formal Object Oriented Programming (OOP) paradigm of clear unit separation, so by following this document's instructions a next developer will not find it difficult to alter functionality or fix a bug.
- Performance and Response Time: The server must response to the request as fast as possible even when there are a lot of threads in the same time, the max response time should be 3000.
- Security and Survivability: One of the most important features is the security of our server, and how it maintain everything under control even under in an attack situation.
- High Level of Abstraction: Since we are using the REST architecture style, there is no need to know all the technical details of how the service work, REST standard protocols handles all required contacts.

C. Proposed Web Server Components

The server consists of several external and internal components, integrated in a solid architecture to fulfill their duties and work together as a one unit, each component is responsible for one or more transaction, or simply process data and move it on to another component in order to get the job done. Some components are meant to have several instances running at the same time to serve certain concurrency needs.

All internal and external components are integrated to insure the rapid and lightweight transactions, starting from deploying services, accepting several connections, processing requests and finally returning requested resources to clients on the fly. Before taking the server apart and analyzing its components, a basic understanding of the server's process cycle, as shown by Figure 1, would clarify the required information to understand the significance and functionality of each unit and how it interacts with other units.

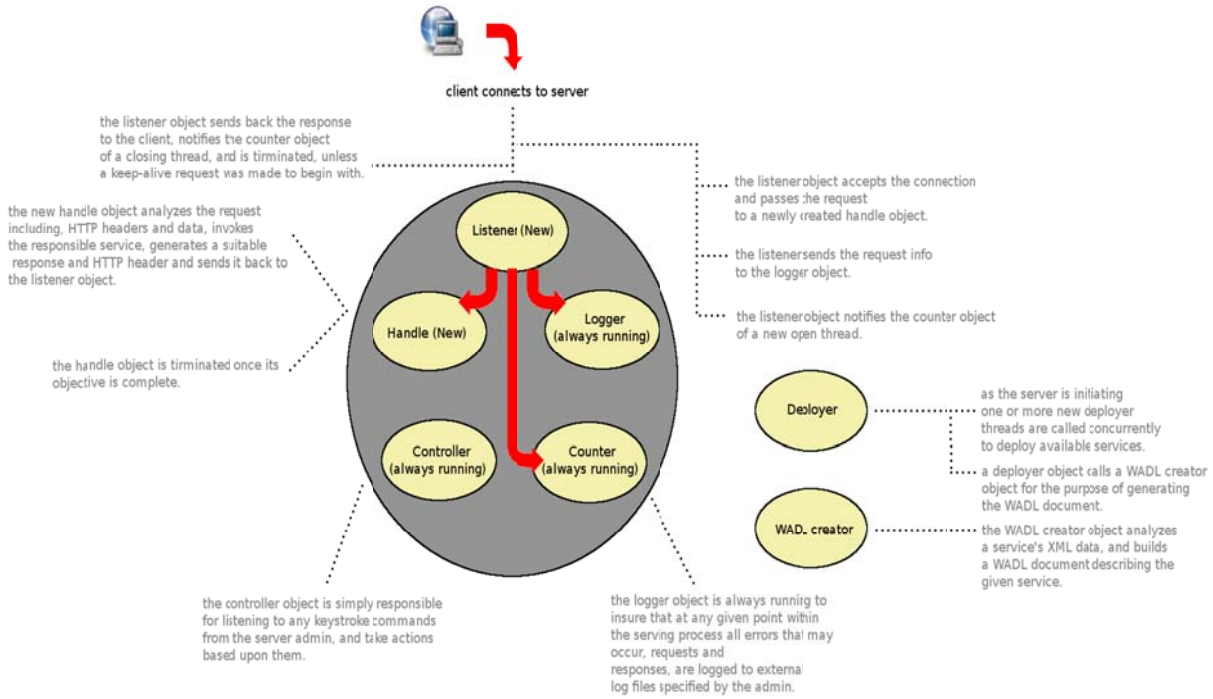

Figure 1. A comprehensive overview of the server's components.

1) External Components

a) Main external components

i. *Deployer:* As soon as the server is initiated, one of the very first actions it takes is calling one or more deployer threads (as needed, or specified in the server's configuration). The deployer is responsible for making a given service ready for serving by several means which include, compiling Java source code and calling the WADL creator to generate a WADL document. The deployer object starts by deleting any old deployment folder for the service, and creates a new deployment folder to be accessed by the server when needed. The deployer object creates a folder for Java compiled classes to be copied there. The deployer object analyzes the manifest file for the given service and creates Java classes according to resources defined for this service.

All source code classes are compiled via the Java Development Kit (JDK) pre-installed on the system, and deleted after compilation which leaves only compiled executable byte-code classes. At this point the deployer object checks whether the compilation process has been executed successfully or not, and returns to the

server any errors. If the compilation process was successful, the deployer calls a WADL creator object to generate a WADL document for the given service. After the deployer has completed all tasks, it returns to the caller a success or fail state accompanied with failure details that may have occurred at any point.

ii. *WADL Creator*: WADL creator object is called to generate a WADL document for a given service corresponding to the international WADL document standards. The WADL creator first reads the service's manifest file and determines whether it is a valid one or not. Information required to generate the document is extracted from the manifest file, and a WADL document is created and information is parsed to it in XML. After all tasks are completed the WADL creator object returns a success or failure state back to the caller accompanied with failure details that may have occurred at any point within the process.

b) Supporting External Components

The supporting external components are used to respond to certain methods called by some of the server's components under the Windows operating system, as these methods are natively supported under the Linux operating system, anyhow they had to be satisfied when running the server under Windows. Further details address these issues and explain how these components support the server's processes under Windows.

i. *rm*: under the Linux operating system, the "rm" command is used to remove a file or folder, and could be used with other arguments, however under the Windows operating system this command is not defined, while a similar command is available, it was seen for the best to create an external package that would simulate this command under windows. A number of components within the server call the system command "rm", which is not an issue under Linux, using this package ensures that is command will also be carried out under Windows just as so. Once called, the rm object reads the call's arguments and if valid, will delete a file or folder as specified by the call.

ii. Javac: The java JDK compiler under the Linux operating system can be called by the command "javac" accompanied by other arguments, however under the Windows operating system the full path for this command should be stated. As some components of the server make calls to the Java compiler, the call would work flawlessly under Linux, anyhow it would not work under Windows. javac component locates the Java JDK compiler on the system (if any found), passes the arguments to the compiler, insuring that any calls using the command "javac" under Windows would be passed to the compiler with the full proper path.

iii. cp: under the Linux operating system, the "cp" command is used to copy a file or a folder, the command is accompanied with some arguments. Under the Windows operating system, a similar command is available, however the exact same command does not exist. Some of the server's components call this system command, anyhow it would not work under Windows, which is why this component exists in order to simulate the functionality of this command under Windows. The cp object reads a call with its arguments, and is responsible for completing the required copy action based on the call's arguments.

2) Internal Components

a) Listener

The Listener object is considered to be the lowest level component in the server, as well as being the most substantial. Its main objective is to receive a connection from a client, decide whether to accept or deny it, create a new handle object to process the request.

When a new listener object is initiated, it listens to a given port for any new connection, once a connection request is made, the listener object stops listening, and initiates a new listener object to listen for any new connection requests, and notifies the counter object of one new connection is being handled. The listener checks the client's IP against a predefined list of denied IP addresses. If this client is not found in the list, its connection to the server is granted, and a data stream is now opened between the client and server for incoming payloads. The listener waits for a predefined amount of time for the client to make a request. Once a request is made, the listener analyzes the HTTP header, determines if the request is valid, by checking the HTTP request method (valid methods are OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE, JEFF, CATS) [27]. Other information is extracted from the HTTP request header to be added to the log information. A couple of header items "x-forwarded-for" and "via" (if exist in the HTTP header) are scanned for IP addresses, and if found are checked against the denied IP, which may lead to the fact that a denied client is making a connection to the server through a proxy server. In this case the listener will close and terminate the connection.

At this point, the listener object creates a handle object to process the request and give back a suitable response, once the newly created handle object returns the response, the listener returns the response to the client by dividing the response into several packets (depends on the size of the response and the network buffer size at the time). After the

response has been delivered to the client, the listener notifies the running logger object of the transaction made, then waits for further requests or terminates, depending on what action was requested by the client in the HTTP header item "connection". The maximum number of allowed "keep-alive" requests is predefined by the admin in the server's configuration. At any point the listener has completed its tasks and just before it is terminated, notifies the counter object that it has completed its work, thus one connection is done with.

b) Handle

Generating a suitable response, along with a HTTP response header is the main functionality of the server, this is the responsibility of the handle object. Once a new handle object is created and a request is passed to it, the first thing the handle checks for in order to determine the response, is the HTTP request method. If the request involves invoking a service, the handle will search for the service, invoke its resources, using the Java JRE pre-installed on the system, returns back the response, adds to it a suitable HTTP header depending on the request and response. Returns the whole response to the caller (a listener object), and then the handle object is terminated. The handle object is built to be able to generate a suitable response in every possible case (e.g requested resource does not exist, HTTP request version is not supported, uniform resource identifier (URI) is too long, no response found, wrong method is used, or bad syntax is used to reach a specified resource, etc.).

c) Logger

Each complete transaction within the server (connection, handling, response) is logged to an external log file, corresponding to the World Wide Web Consortium (W3C) log file format standard. The logger object is always running as a concurrent thread. Its main objective is to receive notifications of transactions with their full details and write them to the log file. The logger object has a queue of transactions waiting to be written to the log file, whenever the logger receives a new transaction notification, it is added to this queue to be written once its turn is due. Two logger objects run within the server, the first is to log transactions, the other is to log any errors that may occur. Separate log files are used for these purposes, which can be defined by the admin in the server's configuration.

d) Counter

In order to limit the maximum number of active connections made to the server, the counter object which is always running as a concurrent thread keeps count on the current number of concurrent connections currently being handled by the server. A new connection to the server cannot be initiated if the number of current connections has reached the maximum allowed connections. The number of maximum allowed connections to the server is defined by the admin before starting the server in the server's configuration.

e) Controller

The controller component is always running as a concurrent thread, it has one objective, which is simply to wait for any keystrokes from the admin, if the captured keystroke is a "q" character, the controller object will end the server's process.

D. XML File Schema

In order to build a web service to run on the server, an XML file (manifest.xml) should be created for this service, containing a description of the service and its resources. A Java class (Index.java) should also be created for the service, where all the resources server-side code will be held. The XML file should comply with certain rules in order for the server to be able to successfully read and identify resources within this service. These rules are:

- All tags within this file should be placed within a <service> tag.
- The service base attribute tag (<base>) must be defined before adding any resource tags. Where the base defines the URL of the service on the server (e.g, <base>http://localhost:8080/f2c/</base>).
- After the base tag is added, a <resources> tag can be placed now to start adding resources to the service.
- For a group of resources within a <resources> tag, a <method> tag should be added to let the server know of the suitable method clients can reach these resources (e.g, <method>GET</method>).
- For each resource, a <resource> tag must be added.
- For each resource, a <name> tag must be included to identify the name of this resource (e.g, <name>celsius</name>).
- Another tag should be added for the resource, which is the <action> tag, which tells the server which Java method to invoke when a client calls this resource (e.g, <action>@converttocelsius</action>).
- A <produces> tag should also be added for a resource, in order to let the server know what MIME type to respond with to the client (e.g, <produces>application/xml</produces>)
- The <description> tag is an optional tag, where the developer could add a comment describing this resource's work (e.g, <description>returns celsius degree, from fahrenheit degree</description>).
- If a resource is designed to parse one or more parameter, a <parameter> tag should be added for each parameter.
- For each parameter, a <pname> tag should be added to include the name of this parameter (e.g, <pname>fdegree</pname>).
- A <ptype> tag should also be included to determine the type of this parameter (Int, String, Double, Boolean) (e.g, <ptype>int</ptype>).
- A <default> tag is optional, which includes a default value for this parameter, for the client to

consider when parsing a value (e.g, <default>38</default>).

- The <option> tag is also optional, where it holds an optional value for the client to consider for this parameter. One or more <option> tag may be added for a parameter (e.g, <option>100</option>).

The following Table shows an XML file containing a description of a service "f2c"(returns Celsius degree from Fahrenheit degree):

TABLE I. AN EXAMPLE OF AN XML FILE

```
<?xml version="1.0" encoding="UTF-8"?>
<service>
    <base>http://localhost:8080/f2c/</base>
     <resources>
          <method>GET</method>
          <resource>
               <name>celsius</name>
               <action>@converttocelsius</action>
               <parameter>
                    <pname>fdegree</pname>
                  <ptype>int</ptype>
                  <default>38</default>
                  <option>100</option>
                  <option>50</option>
              </parameter>
 <produces>application/xml</produces>
               <description>returns Celsius degree,
                    from Fahrenheit
degree</description>
          </resource>
     </resources>
</service>
```

E. Main Java Class Constraints

- Each method defined within the class should have a string return type.
- A method should contain each parameter defined in the XML file, to be parsed as String.
- Every method should also parse three static variables (remoteAddr, requestURI, String reqBody) all of type String, which can be used within the method.

The following Table shows the "Index.java" class written as an example for the previous XML file example that was specified in Table I:

TABLE II. AN EXAMPLE OF A JAVA INDEX CLASS

```
public class Index {
public Index() {}
//@GET
//produces application/xml
public String converttocelsius(String fdegree,
String remoteAddr, String requestURI, String
reqBody) {
Response response = new Response();
double cel = (Double.parseDouble(fdegree) - 32)
* 5 / 9;
response.print("<celsius>"+cel+"</celsius>");
return response.getResponse();
}
}
```

Note: the XML file, java index class, and any other external java classes should be all put together in the same directory, which has the name of the service. The service's directory is to be put into the server's "htdocs" directory.

IV. TESTING AND RESULTS

For testing purposes, we need to make sure that every output of the Web server meets the requirements, and we would perform a load and performance test. Our test case would be testing the same service that we built and used as example to convert from Fahrenheit to Celsius. Unfortunately, we were unable to compare our results with other Web servers like Apache or Nginx since our Web server support clean URL whereas Apache and Nginx don't support a real clean URL, which would be a problem. SoapUI 4.5 is one of the famous SOAP and REST testing software, and that is what we will use in our tests.

To begin our testing, we need first to initiate a soapUI project as shown by Fig. 2.

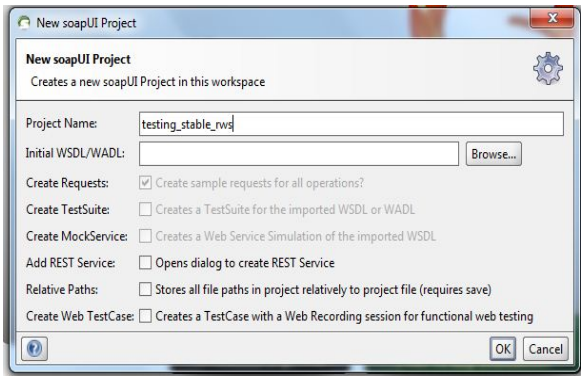

Figure 2. Project Creation.

Now we add the WADL URL to the project, by 'Right Click on the Project → Add WADL' as shown by Fig. 3.

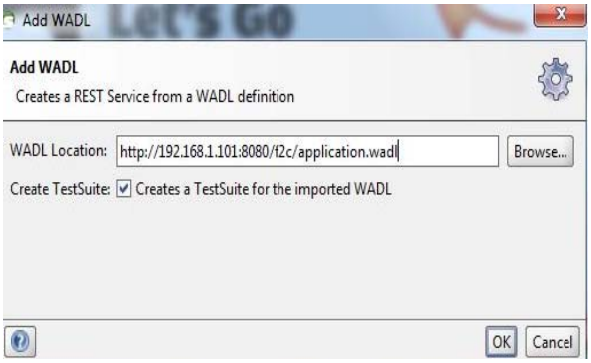

Figure 3. Linking WADL.

In Fig. 4 we can notice that the soapUI has defined all the resources of that service, and methods correctly just by reading the WADL document of the service, which would even prove that the server created the WADL document correctly based on the standard of the documentation.

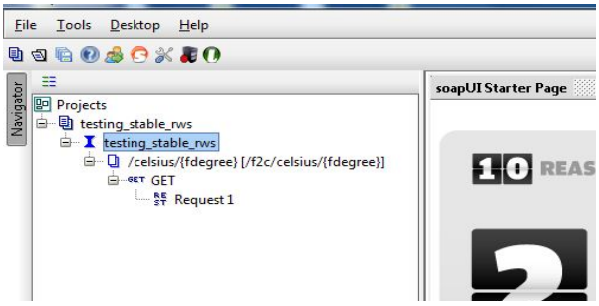

Figure 4. Project Tree.

Looking into Figures 5 and 6, the method that sends a request to the server to convert Fahrenheit degree as example “80 Fahrenheit” to Celsius Degree is invoked. The results were correct in both XML and HTML formats, with average response time (250ms), which is quiet fast.

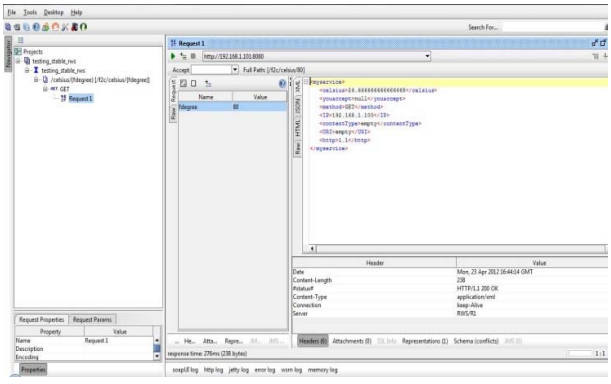

Figure 5. Converting Example – XML Respone.

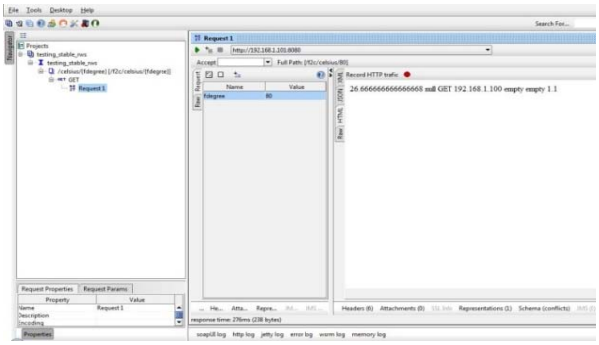

Figure 6. Converting Example – HTML Respone.

A load test on the server is conducted, in which we started with a simple test of 5 threads in 60 seconds, and the results were good as expected. The results, as shown by Fig. 7, were as follows: average response time (226.66ms), minimum time (166ms), maximum time (701ms), and with zero errors.

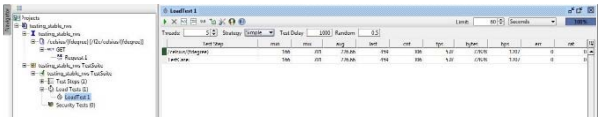

Figure 7. Load Test Results - Simple

The system monitor in Linux Operating System is used to monitor our resources, and the results were great, with average of 25% of the CPU usage as depicted by Figures 8 and 9.

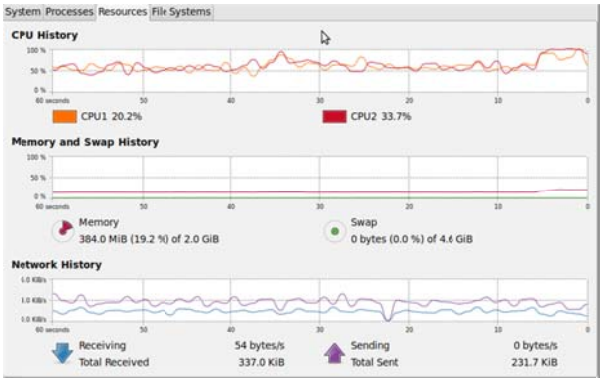

Figure 8. Load Test Results – Simple – Server Status During Test 1.

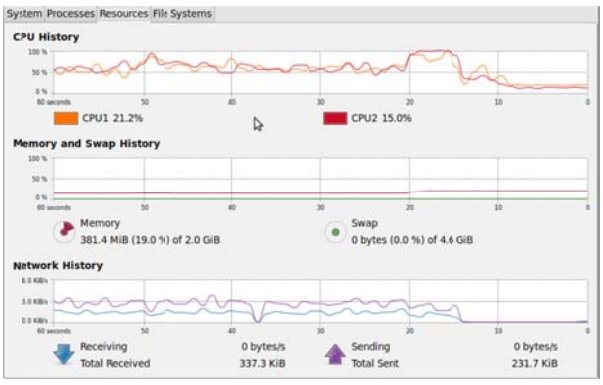

Figure 9. Load Test Results – Simple – Server Status During Test 2.

Bearing in mind the good results from the 5 threads test, we decided to take it to the next level and created

a test case with 1000 threads in 60 seconds, the results were more than great where the server handled all the 1000 threads correctly. The results were as follows: average response time (265ms), minimum time (202ms), maximum time (387ms), and again with zero errors. The results shown by Figures 10, 11, 12 and 13.

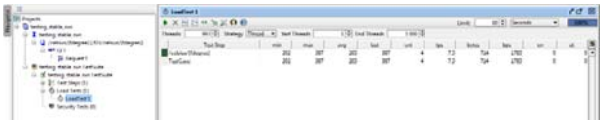

Figure 10. Load Test Results – Advanced 1.

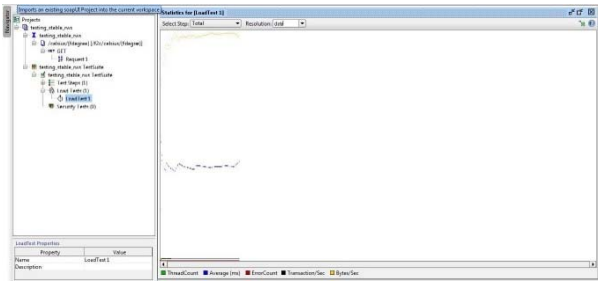

Figure 11. Load Test Results – Advanced 2.

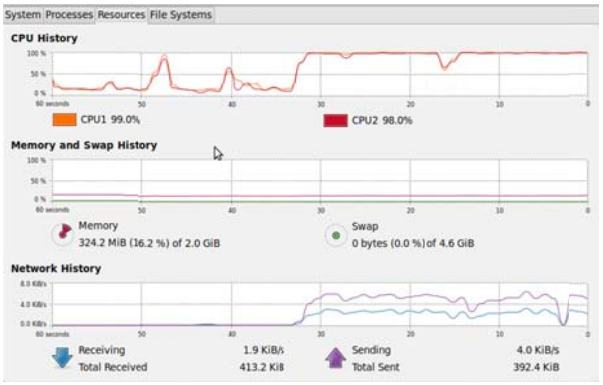

Figure 12. Load Test Results – Advanced – Server Status During Test 1.

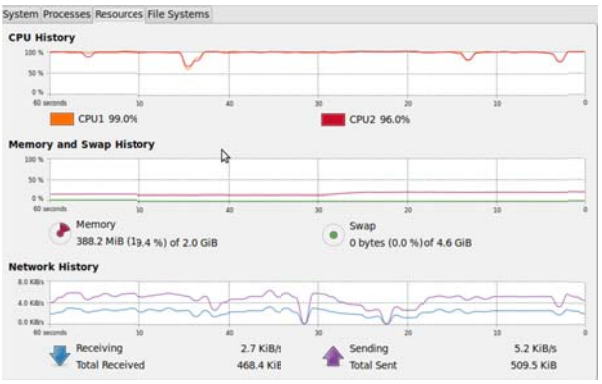

Figure 13. Load Test Results – Advanced – Server Status During Test 2.

To sum up, the results were more than perfect for the proposed RESTful Web server in which the error log recorded zero errors, and the access log file recorded all the transactions and it was all successful.

NOTE: The server was running on Linux Ubuntu 10.04 (Dual Core 1.7 GHz, 2GB DDR2), and the testing conducted on Windows 7 (Core i5, 4GB DDR3). The server minimum requirements are ARM1176JZF-S 700 MHz processor and 512MB SDRAM memory.

V. Conclusion and Future Work

In this paper, we have presented a RESTful Web server that can be used for hosting REST-based web services instead of using the ordinary servers that are used for Full-Blown websites. The proposed RESTful Web server provides high performance and stability since it is less demanding on system resources. It also provides developers with a rapid and cost-effective method for implementing, deploying and serving their web APIs which will satisfy the REST style architectural constraints. However, there are some security challenges concerning the RESTful style architecture that have not been considered in our current implementation. These challenges reduces the scalability of the REST style architecture for applications and services [28]. Therefore, our future work will focus on addressing and solving some of the security challenges in our implementation. Furthermore, detailed literature review for different available Web servers, testing their performance comparing the results with our proposed architecture will be carried out.

References

[1] J. Becker, M. Matzner, and O. Müller, "Comparing Architectural Styles for Service-Oriented Architectures - a REST vs. SOAP Case Study," *Information Systems Development*, G. A. Papadopoulos, W. Wojtkowski, G. Wojtkowski *et al.*, eds., pp. 207-215, US: Springer, 2010.

[2] T. Kaewkiriya, R. Saga, and H. Tsuji, "Framework for Distributed e-Learning Management System," *Journal of Computers,* vol. 8, no. 7, 2013.

[3] C. Liu, and D. Liu, "QoS-oriented Web Service Framework by Mixed Programming Techniques," *Journal of Computers,* vol. 8, no. 7, 2013.

[4] N. Laranjeiro, M. Vieira, and H. Madeira, "A robustness testing approach for SOAP Web services," *Journal of Internet Services and Applications,* vol. 3, no. 2, pp. 215-232, 2012. http://dx.doi.org/10.1007/s13174-012-0062-2.

[5] H. Xu, A. Reddyreddy, and D. F. Fitch, "Defending Against XML-Based Attacks Using State-Based XML Firewall," *Journal of Computers,* vol. 6, no. 11, 2011.

[6] ZDNet. "SOA and cloud by the numbers: what the data tells us so far," [viewed 27 November, 2011]; http://www.zdnet.com/blog/service-oriented/soa-and-cloud-by-the-numbers-what-the-data-tells-us-so-far/3974.

[7] P. Prescod, "Roots of the Rest/Soap Debate.," Proceedings of the 2002 Extreme Markup Languages Conferences, Quebec, Canada, 2002.

[8] M. zur Muehlen, J. V. Nickerson, and K. D. Swenson, "Developing web services choreography standards- the case of REST vs. SOAP," *Decision Support Systems,* vol. 40, no. 1, pp. 9-29, 2005. http://dx.doi.org/10.1016/j.dss.2004.04.008.

[9] G. Mulligan, and D. Gracanin, "A comparison of SOAP and REST implementations of a service based interaction independence middleware framework," Proceedings of the 2009 WinterSimulation Conference

(WSC), Austin, Texas, pp. 1423-1432, 2009. http://dx.doi.org/10.1109/wsc.2009.5429290.

[10] R. Battle, and E. Benson, "Bridging the semantic Web and Web 2.0 with Representational State Transfer (REST)," *Web Semantics: Science, Services and Agents on the World Wide Web,* vol. 6, no. 1, pp. 61-69, 2008. http://dx.doi.org/10.1016/j.websem.2007.11.002.

[11] I. Zuzak, and S. Schreier, "ArRESTed Development: Guidelines for Designing REST Frameworks," *IEEE Internet Computing,* vol. 16, no. 4, pp. 26-35, 2012. http://dx.doi.org/10.1109/mic.2012.60.

[12] N. S. Bhuvaneswari, and S. Sujatha, *Integrating Soa and Web Services*: River Publishers, 2011.

[13] L. Richardson, and S. Ruby, *RESTful Web Services*: O'Reilly Media, 2008.

[14] R. T. Fiedling, "Architectural Styles and the Design of Network-Based Software Architectures," University of California, Irvine, 2000.

[15] D. N. Darji, and N. B. Thakkar, "Comparative Study on the Features of Different Web Services Protocols," *International Journal of Research in Computer Application & Management,* vol. 2, no. 9, pp. 102-106, 2012.

[16] InfoQ. "REST and SOAP: When Should I Use Each (or Both)?," [viewed 26 March, 2012]; http://www.infoq.com/articles/rest-soap-when-to-use-each.

[17] Geeknizer. "REST vs. SOAP - The Right WebService," [viewed 26 March, 2012]; http://geeknizer.com/rest-vs-soap-using-http-choosing-the-right-webservice-protocol/.

[18] Ajaxonomy. "Web Services, Part 1: SOAP vs. REST," [viewed 26 March, 2012]; http://ajaxonomy.com/2008/xml/web-services-part-1-soap-vs-rest.

[19] K. Jucyte. "Web service implementation with SOAP and REST," [viewed 26 March, 2012]; http://rudar.ruc.dk/bitstream/1800/2108/1/Web%20services%20-%20SOAP%20%26%20REST.pdf.

[20] D. L. Ridruejo, and D. Lopez, *Sams Teach Yourself Apache 2 in 24 Hours*: Sams Publishing, 2002.

[21] G.-h. Li, H. Zheng, and G.-z. Li, "Building a Secure Web Server Based on OpenSSL and Apache," Proceedings of the 2010 International Conference on E-Business and E-Government (ICEE), Guangzhou, pp. 1307-1310, 2010. http://dx.doi.org/10.1109/icee.2010.334.

[22] Wikipedia. "Internet Information Services," [viewed Septemper 2, 2013]; http://en.wikipedia.org/wiki/Internet_Information_Services.

[23] D. Aivaliotis, *Mastering Nginx*, UK: Packt Publishing, 2013.

[24] W. Reese, "Nginx: the high-performance web server and reverse proxy," *Linux Journal,* vol. 2008, no. 173, 2008.

[25] H. J. Li, "Research on Restful Web Services in Java," *Applied Mechanics and Materials,* vol. 135-136, pp. 806-808, 2012.

[26]L. Hongjun, "RESTful Web service frameworks in Java," Proceedings of the 2011 IEEE International Conference on Signal Processing, Communications and Computing (ICSPCC), Xi'an, pp. 1-4, 2011. http://dx.doi.org/10.1109/icspcc.2011.6061739.

[27] J. G. R. Fielding, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, *Hypertext Transfer Protocol--HTTP/1.1*, RFC 2616, Internet Engineering Task Force, 1999.

[28] D. Forsberg, "RESTful Security," Nokia Research Center, Helsinki, 2009, [viewed 06 January, 2013]; w2spconf.com/2009/papers/s4p3.pdf.