Web Services: Architectural Styles and Design Considerations for REST API

S. Ganesh Kumar, Sanjay Shanbhag, G Rohit

Abstract: ***The issue of sharing real time data with others is paramount, especially in today's digital age. We have lots of applications generating tons of data every minute. Each and every row of such data is useful either to the data generator or to a third party application. But we need a standard set of protocols to share data between applications over a network. This is where Application Programming Interfaces comes in. By using web services any application can share and read information automatically from other applications without human interference. This greatly advances the data sharing between applications, hence improving the services, productivity and user experience. We identified that soccer and sports in general pumps large amount of data into the internet, with no proper way to leverage it. Hence, we will be building a system, that collects this massive data from 1993 all the way up to the present time and will continue to do so automatically as long as there is data to mine, and a REST API on top of it so that the client/developers can access this data in a slick, automated, efficient and a fast way. This paper deals with various ways of leveraging data available online as well as an in detail comparison of the two major types of web services namely: SOAP & REST. It then goes on to detail the architectural styles and the design considerations to build REST API from scratch.***

Index Terms: Web Services, API, REST, SOAP, JSON, XML, Service Oriented Architecture

I. INTRODUCTION

Sports is an entertainment field, which a lot of people follow. Soccer in itself the most popular game on this planet with a huge fan following. [1] There are literally 100's of leagues around the world with countless teams participating in them. Talking about statistics, soccer is a major data generator.

There are close to 20 matches happening in a week in a single league. Also, there are close to 8 major leagues across the world and at least 16 minor leagues. [1] Every match comes with statistics such as Goals, Cards, Possession fouls, etc and to top it all off they come with half-time statistics as well. [2] So we can see the amount of data that is being pumped into the internet every week. This type of data is gold to data scientists who could use it to build interesting algorithms for data analysis and predictions. It could also be used to build awesome applications. [3]

Revised Manuscript Received on May 04, 2019.

S. Ganesh Kumar, Associate Professor, Computer Science & Engineering, SRM University.

Sanjay Shanbhag, Student, Department of Computer Science & Engineering, SRM University.

G. Rohit, Student, Department of Computer Science & Engineering, SRM University.

But if one intends to leverage this data into, let's say an application that plots individual team's performance over a certain period of time or even a simple application that lets the analysts out there play around with this data, how would they do it? Sure, this data is out there for anyone and everyone to see. [4] But, how does one get their hands on the entire data set spanning years and continuously keep doing so every day? Even if they do, this data is extremely large in both size and sheer amount of data within that it is so very easy to get lost within the millions of rows and columns that is the Microsoft Excel. Navigating through this humongous data is a problem in itself and one that many developers don't like to get into. Also, one needs to consider the pumped data of this magnitude is not so easy to retrieve. One needs to apply data mining techniques to continuously keep retrieving data into their own data store. So what are the ways of leveraging the data?

II. DIFFERENT WAYS OF LEVERAGING DATA AVAILABLE ONLINE

1. Create one's own database

If one is technically sound, sure. Seems like the best bet at first, but then again one needs to constantly keep updating the database as and when the new data arrives, or build a sophisticated automated system to do this job automatically in the background. This can be done if they have got considerable resources at hand such as servers, hosting, server space, cron job support, security, etc, and not to mention the programming capability to successfully build the data mining system to continuously update the data store. Also, even though this data exists on their own servers performing extensive queries and retrieving user specific results is much slower when compared with a technology like Web Services, which ill form the next part of our discussion.

2. Web Services

Web Services have been used since many years as an interface between applications. Web Services such as Simple Object Access Protocol (SOAP) and Representational State Transfer (REST) are extremely useful in data transfer and comparatively faster. Creating a Web Service saves the hassle for all developers as the tedious work such as building the database, automating the system and even performing SQL or other database queries is taken away. With such hassles out of the way they can just jump into building what they intended to and leverage this data using our proposed web service.

This is a win-win for everybody as there is one centralized version of the data that anybody and everybody can use in a simple and easier way.

III. WEB SERVICES: SOAP & REST

Now that we have established the need for web services, it is time to get a little deeper into its types, how they work, and we will also talk about the particular type that we have chosen to go with and why.

There are basically two main types of data transferring Web Services, namely SOAP and REST. We will be looking at both in detail below. The data transfer through web services can be done in two ways:

Extensible Markup Language (XML): a standard used to describe data in a flexible way.

Fig. 1 A simple XML response

JavaScript Object Notation (JSON): a language independent format which is comparatively less verbose and which is easy to generate and analyze.

Both of these have advantages and disadvantages of their own with many developers preferring one over the other. The system that we are about to build will have both standards readily available and it will be up to the developers' discretion to choose the system that they are comfortable with and the system that is compatible with their applications.

But, as shown through Fig 1 and Fig 2, JSON is obviously less verbose and easier to understand than its XML counterpart.

Fig. 2 A simple JSON Response

Hence, it should come as no surprise that JSON is the more preferred response format for many developers.

Simple Object Access Protocol as well as Representational State Transfer are both extremely useful and have advantages disadvantages of their own. SOAP is a standardized pro-tocol, governed by the strict rules and guidelines set and maintained by the W3C. Whereas, REST on the other hand is more of an architectural style, only giving certain guidelines for the developers to implement. Its rules aren't set in stones and it leaves the developer with the choice to implement the lower level build in their own way.

We have already established that SOAP service is slow, the reason for that being the strict protocol like nature of the service. The security, error-handling and authorization, etc are built into the protocol making the requests as well as responses bulky.

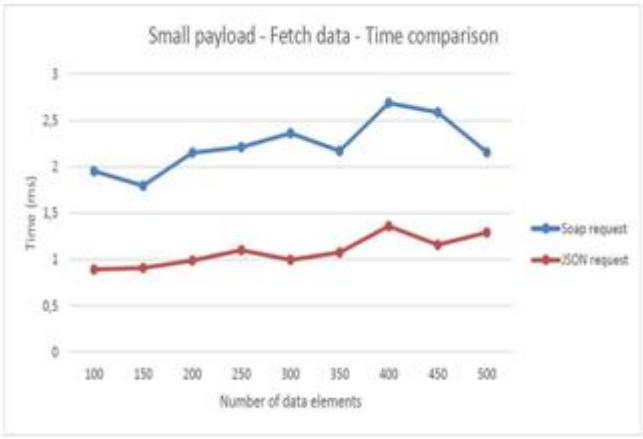

Fig. 3 The response time difference between REST & SOAP

But REST doesn't assume point-to-point direct contact and JSON responses are extremely light weighted. This makes the REST comparatively less secure when compared to SOAP but also light weighted. This difference in the response times and size can be seen in the figure Fig 3 above. [5]

REST supports many formats such as XML, JSON, plain text as well as HTML when compared to SOAP's XML only.

Fig. 4 An example of a SOAP response

This gives an obvious advantage to REST as clients as well as service developers do not need to get used to the complicated structure of XML but can work with the format that they are comfortable with.

Figure 1, Figure 2 & Figure 4 depicts the sample XML response, a sample JSON response and a sample SOAP response respectively. [6]

REST works on top of HTTP and inherits all its main operations such as GET, POST, PUT and DELETE. SOAP on the other hand is capable of working with various application layer protocols such as HTTP, TCP, UDP or SMTP.

Given all these differences it may seem as though SOAP seems to be at a slight disadvantages. And you may be right to some extent in thinking so, as REST was designed to offset the disadvantages of SOAP. This is the reason why many web services that have been developed in the recent times have been REST. But that doesn't mean that SOAP is completely useless in today's age. [7] On the contrary, systems that require high security will and should use SOAP. A popular example is PayPal's Public API that uses SOAP protocols to validate credit card numbers and transactions. [8]

But in our case we need not build extreme security into the data transmission in itself. We need security, make no mistake, but we need no extreme security in data transmission, hence we can make do with REST's HTTP protocols. Also, since our API will be used by application developers and data analysts, it is imperative that the service is fast to facilitate the seamless transmission of data. As already established, REST is faster than SOAP giving us another reason to choose REST over SOAP.

IV. BUILDING REST

We have chosen REST as our preferred web service and will now look into the design considerations involved in building a RESTful API from scratch.

	SOAP	REST
Meaning	Simple Object Access Protocol	Representational State Transfer
Approach	Function Driven	Data Driven
State	Stateless by default, but stateful is SOAP API is possible.	Stateless (No session elements)
Design	Standardized Protocol	Architectural Style with loose rules
Caching	API calls cannot be cached	API Calls can be cached
Message Format	Only XML	XML, JSON, Plain text and HTML
Security	WS-Security with SSL support	Supports SSL and HTTPS
Transfer Protocols	HTTP, SMTP, UDP, etc	Only HTTP

Fig. 5 Summary of key differences between SOAP & REST

While implementing the REST service we will keep in mind the following 4 architectural constraints: [9]

Uniform Interface- The API needs to have only one endpoint for each resource.

Stateless- There should be no server side sessions, or logins. All the data required by the service including authorization details should be sent in the request itself.

Client-Server Separation- As name suggests the API intends to separate the client and server. So the client will only be able to interact with the data in request-response way.

Cacheable resources- This API will allow the clients to cache certain responses. This will be in conjunction with the versions so that the client will be able to request newer versions without getting back the cached responses every time.

V. DESIGN CONSIDERATIONS

There are certain design considerations that we will have to look into so that API doesn't crash or is mishandled.

The first thing that we have to consider is to build an automated system to keep our database fresh and up to date. As the soccer data is ever-increasing and changing, we have to constantly keep a check and make sure that the latest data matches our version. We will build the automated system using machine learning & data mining algorithms as well as server cron jobs.

The second design consideration is security. It was mentioned earlier that security isn't paramount in our case as this is a data based service rather than a transaction based one. This is indeed true and this is what made us choose REST over SOAP in the first place. But this security does not correspond to the response transmission security but the security of the system in itself. Data centric platforms and services are often targets for hacks, and we will be building stringent security into the system to curb this. Any developer/client who intends to use the API will need to be in the possession of a unique token. This token will be provided by the system after a simple sign up process. Every request to the API will need to be made with this token, and the response will only be sent back to the genuine and safe users.

DDOS attacks are common and can put down any service if not well shielded. We will be implementing access throttling into our API. Each access token will be limited to about 10000 (a tentative figure) requests an hour. If an access token and in turn a user exceeds this figure before the completion of the hour, their access will be revoked for the remainder of the time. That access is of course granted back after the completion of the said time. This prevents the API from being misused and dominated by certain users, thus denying and delaying the response times of others due to server loads. This in conjunction with the token authorization method makes sure that DDOS attacks aren't theoretically possible.

The last design consideration is documentation. We know that without proper documentation, it is very difficult to use the API due to its complicated request states and the nature of the data. Thus, we will be releasing a detailed documentation on our API endpoint with request and response examples for each request and response type. This alongside this paper should give enough understanding of the API to any developer willing to use it.

VI. IMPLEMENTATION

Implementing and actually building the API with the above said guidelines and considerations is a humongous task in itself as we started from scratch.

We decided to go with the MEAN stack as our preferred development stack. The individual components of this stack are as described below:

MongoDB: Document database – used by the back-end application to store its data as JSON (JavaScript Object Notation) documents

Express (referred to as Express.js): Back-end web application framework running on top of Node.js

Angular (formerly Angular.js): Front-end web app framework; runs the JavaScript code in the user's browser, allowing the application UI to be dynamic Node.js: JavaScript runtime environment – lets one implement application back-endin JavaScript. Runs the Chrome's V8 engine outside the browser. [10]

Node is becoming the popular choice for API development as it along with Express makes it simpler to define and work with routes.

We mined the data of football matches from the season 1993-94 to 2018-19 (current season) and build automated system that keeps our mongo database updated as and when new matches are played. The database also included the details about each and every competition supported by the API and also details about each and every team/club that has ever participated in the said leagues. As of at the time of writing this, we support 22 leagues from around Europe which includes all the 8 major leagues and 16 minor leagues. We plan to include leagues from North and South America in the future.

As already mentioned, this API works in the form of a HTTP request-response format. For example, a sample request to our API would look something like this:

```
1  sampleDomain.com/api/v1/{API_KEY}/
       competitions
```

Code Snippet 1.

For a valid request this would return the details about the competitions supported.

Here's a sample response:

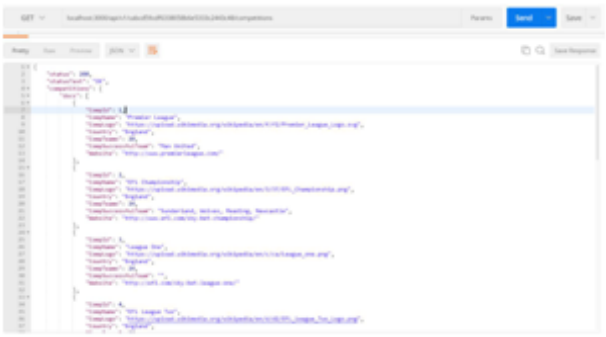

Fig. 6 Sample response when run on localhost with Postman

Similarly, there are other routes such as,

```
1  sampleDomain.com/api/v1/{API_KEY}/
       standings/8/1415
```

Code Snippet 2. Request sample for the league table for the Spanish top flight for the season 2014-15

There are many other routes, each of which has been extensively documented on our documentation page. These requests can be used in conjunction with each other in our API to retrieve any and all things football.

Since Node is becoming a very popular choice as the server side language of many programmers, we have build a native library on top of node to make it easier for developers to work with our API. This library is hosted on npmjs.com, free to download and use. This library makes it extremely simple to make requests and receive responses from our API. For example, in node making a request the vanilla way can look something like this:

```
const express = require('express');
const bodyParser = require('body-parser');
const mongoose = require('mongoose');

var app = express();

app.use(bodyParser.urlencoded({ extended: false }))

app.get('/soccerapi.com/competitions/1415', (req, res) => {
  req.fetch().then((data) => {
    if (!data) {
      throw new Error('No Data Found!');
    }
    console.log(data);
  }, (err) => {
    res.status(400).send('Something Went Wrong!');
  });
});
```

Fig. 7 Sample API request in NODE JS

Note that this is not the de-facto way of making requests in Node. This is just one of the countless ways in which developers could do it.

Now, this is what the code essentially boils down to by using the native library:

```
const soccerapi = require('soccer-api');
const API_KEY = 'API_KEY';

var data = soccerapi.getCompetitions(API_KEY, 1415);
console.log(data);
```

Fig. 8 Sample Node request with our library

The node library is not a must, to use the API, but it exists only to make life a little bit easier.

VII. FUTURE WORKS AND CONCLUSION

Using Web Services to transmit data, automate systems and creating application to application interface is very important, more so in today's world where everything is digital. So much data is generated every second and sharing

the data between applications is no longer manually possible. This is the reason why web services and API 's in particular has seen tremendous growth. All the major internet players have their own web services allowing developers to leverage their technology. Wikipedia, Google, PayPal, Twitter all have their own API's to provide a host of services to developers across the world.

We identified that sports pumps large amount of data into the internet, with no proper way to leverage it. Hence, we will be building a system, that collects this massive economic data from 1993 all the way up to the present time and will continue to do so automatically as long as there is data to mine and REST API on top of it so that the client/developers can access this data in a slick, automated, efficient and a fast way.

This API will always be a work in progress and will never be 'fully' complete. There will always be bugs to squash and new features to implement. Also, the data isn't exactly generic, meaning its structure could change anytime, hence this API will be built with constant change as priority. This API will be easily able to adapt to changes without hindering the client experience or having them to change their understanding of the system.

When coming to the data in itself, we could implement our own algorithms to predict the results for upcoming fixtures. With so much historic as well as current data at our disposal it becomes quite hard to resist the urge to implement our own algorithms to predict future results. This could be done with the help of Machine Learning supervised algorithms such as Linear Regression, KNN and Random Forests. This could be combined with our API as a new resource which again saves the clients a lot of time as they don't have to spend their own resources to build algorithms for predictions. This adds a lot of value to this API and many clients & developers may find this particular feature appealing to them.

REFERENCES

1. "Association of Football." [Online]. Available: https://en.wikipedia.org/ wiki/Association_football
2. "Match Statistics." [Online]. Available: https://www.google.com/ search?q=chelsea&rlz=1C1CHBD_enIN727IN727&oq=chelsea&aqs= chrome..69i57j69i60j69i61j69i60j69i59l2.1010j0j9&sourceid=chrome & ie=UTF-8#sie=t;/m/023fb;2;/m/02_tc;mt;fp;1;;
3. "REST." [Online]. Available: https://en.wikipedia.org/wiki/ Representational_state_transfer
4. "API Intro." [Online]. Available: https://en.wikipedia.org/wiki/ Application_programming_interface
5. J. Makonnen, "Performance and usage comparison between REST and SOAP web services," November 2017.
6. "Atlassian Developers." [Online]. Available: https://developer.atlassian. com/server/crowd/
7. C. Garcia and R. Abilio, "Systems Integrations Using Web Services."
8. A. Monus, "Raygun - SOAP & REST." [Online]. Available: https://raygun.com/blog/soap-vs-rest-vs-json/
9. J. Hradil and V. Skilinak, "Practical Implementation of REST API."
10. "MEAN Stack." [Online]. Available: http://mean.io/
11. B. Jaya Kaviya and G. Selvakumar, "Survey on RESTful web service composition"
12. S. Ganesh Kumar , K.Vivekanandan " Self-Directed Web Services Composition: A Platform"
13. Anil Dudhe and S. S. Sherekar, "Performance of SOAP and REST web services in Cloud Environment"
14. Min Choi, Young-Silk Jeong and Jong Hyuk Park, "Improving performance through REST Open API grouping for wireless sensor network"
15. S. Ganesh Kumar, K.Vivekanandan2015. "ODMM – An Ontology-Based Deep Mining Method to Cluster the Content from WEB Servers"
16. Ganesh Kumar. S.," Resisting Web Proxy-Based HTTP Attacks through Cross-Site Scripting"

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication