

School of Computing and Engineering

Final Year Project

Abstract

*Project Title:*

Designing and implementing

RESTful Web Services

*Undertaken By:*

Khan Najam ul Asre (21303779)

*Submitted On:*

??????

*Supervisor:*

??????

*Second Marker:*

??????

Contents

[1 Introduction 3](#_Toc512718937)

[1.1 Background 3](#_Toc512718938)

[1.1.1 Monolithic Applications 3](#_Toc512718939)

[1.1.2 Information Technology 3](#_Toc512718940)

[1.1.3 Distributed Systems 4](#_Toc512718941)

[1.1.4 Rise of Computing Devices 4](#_Toc512718942)

[1.1.5 Heterogeneity 5](#_Toc512718943)

[1.1.6 System Agility 5](#_Toc512718944)

[1.1.7 Cloud Computing 5](#_Toc512718945)

[1.1.8 Service Oriented Architecture (SOA) 6](#_Toc512718946)

[1.1.9 Microservices Architecture (MSA) 6](#_Toc512718947)

[1.1.10 Architectural Attributes 7](#_Toc512718948)

[1.2 Motivation 8](#_Toc512718949)

[1.3 Aims and Objectives 9](#_Toc512718950)

[1.3.1 Aim 9](#_Toc512718951)

[1.3.2 Objectives 9](#_Toc512718952)

[2 REST – A Better Style for Architecting Modern Applications 10](#_Toc512718953)

[2.1 Why REST 10](#_Toc512718954)

[2.1.1 Interoperability 10](#_Toc512718955)

[2.1.2 Network-based API vs. Library-based API 10](#_Toc512718956)

[2.1.3 Devices 10](#_Toc512718957)

[2.1.4 The Cloud 10](#_Toc512718958)

[2.2 Why Distributed Systems Fail 10](#_Toc512718959)

[2.2.1 Requirements-Driven Architecture 10](#_Toc512718960)

[2.2.2 Fallacies of Distributed Systems 10](#_Toc512718961)

[2.3 How REST Mitigates Failures 12](#_Toc512718962)

[2.3.1 Constraint-driven Architecture 12](#_Toc512718963)

# Introduction

## Background

### Monolithic Applications

Early computers had limited processing, storage and communication capabilities. Despite, they were too expensive and too large in size for individuals to own or maintain. Hence, they were used only within the large organisations. The software systems were monolithic where functionally distinct concepts, e.g. data persistence and retrieval, business logic, user interface, error handling and logging, were strongly interwoven without any clear boundaries or architectural separation. Business requirements would change rarely and hence the software systems. It was very common that a version of software remained useful for the business for years, so monolithic applications worked really well.



Figure ‎1‑1Monolithic Architecture

### Information Technology

Over the last few decades, computing technology has seen dramatic advancement. Computing devices are becoming smaller and smaller in physical size, cheaper in cost but growing in computational power, data storage capacity and communication capabilities. The introduction of the Internet provided a global communication infrastructure. These factors together gave birth to the Information Technology which involves “the development, maintenance and use of computer systems, software and networks for the processing and distribution of data”[[1]](#footnote-1).

### Distributed Systems

The powerful devices and global communication network infrastructure revolutionised the business information and management systems. It was now possible for organisations to have geographically isolated locations with “autonomous computers, connected through a network and distribution middleware, which enabled computers to coordinate their activities and to share the resources of the system, so that users perceive the system as a single, integrated computing facility.”[[2]](#footnote-2)



Figure ‎1‑2 Distributed Software Architecture

### Rise of Computing Devices

Once thought to be of the interest of large enterprises, computing technology has now become a household commodity. With the introduction of smart hand-held devices computing devices are now personal pocket-items. This has changed software requirements of the business organisations. The market has become competitive. The businesses have to reach vast customer-base across the globe.



Figure ‎1‑3 Computing/Smart devices

### Heterogeneity

Lots of devices and manufacturers means lots of operating platforms and lots of software development frameworks. This poses the challenge of interoperability. Today’s ideal software systems have to be platform independent and capable of communicating and working with systems built using various frameworks and running on various platforms. As organisation’s customer base grows so does the need for system interoperability, to ensure that business is able to reach customer owning different devices running on different platform.

### System Agility

To keep going alongside the competitors, organisations have to change their marketing strategy and product presentation quickly and continuously. This requires the software systems that are agile and responsive, that can be changed quickly with or without the need of redeployment; or support Continuous Integration and Delivery.

### Cloud Computing

Traditionally businesses hosted their own on-premises computing infrastructure. For stable software systems this was feasible both financially and technically. But as the need grew for system agility companies start looking at maintenance and upgradation of on-site computing infrastructure as a continuous financial and technical pressure. This motivated the introduction of cloud computing where specialist organisation hosted and managed computing infrastructure which and be leased by the other business organisations. This shifted the responsibility of system maintenance and upgradation from consumer organisations to the cloud providers. Cloud offered kind of elastic resources that can grow or shrink on demand. The consumer organisations have to pay only what they consume. This is why a huge number of organisations have moved to cloud over last decade and process of migration to cloud still continuous. Although cloud offered a scalable infrastructure it does not come out of the box. The software architecture has to be cloud friendly to take full advantage of scalable cloud infrastructure.

### Service Oriented Architecture (SOA)

Unfortunately, monolithic applications lacked the architectural separation of concerns, therefore unable to become distributed systems. Monolithic software systems are mostly built for single platform using single framework. They are difficult, even impossible sometimes, to be changed or scaled. System designed to run on a single platform lack the interoperability, therefore reaching customer with devices running on different platform is not possible. Monolithic systems therefore failed to meet aforementioned challenges. They cannot take advantage of scalable cloud infrastructure. This lead the software architects to favour software systems composed of small, self-contained, independent and interoperable components rather than a giant monolithic system. Service oriented architecture (SOA) was one answer to such problems. SOA is a “system architecture in which application functions are built as components (services) that are loosely coupled and well-defined to support interoperability and to improve flexibility and reuse”[[3]](#footnote-3).



Figure ‎1‑4 Service Oriented Architecture

### Microservices Architecture (MSA)

“Microservices Architecture is basically Service Oriented Architecture done well”[[4]](#footnote-4). Microservices Architecture “is a method of developing software applications as a suit of independently deployable, small, modular services in which each service runs a unique process and communicates through a well-defined, lightweight mechanism to serve a business goal”[[5]](#footnote-5). Being able to meet almost all challenges of modern businesses, in the recent years MSA for many organisations has become a preferred architecture of creating enterprise applications. Martin Fowler notes Netflix, Amazon, eBay, Twitter, UK Government Digital Services and many other applications and websites have evolved from monolithic to MSA[[6]](#footnote-6).



Figure ‎1‑5 Microservices Architecture

### Architectural Attributes

Now that we have established some of the requirements of the modern software systems and challenges involved thereby, we sum up the properties or attributes of and ideal software architecture that supports distributed applications development in a way that they are easy to integrate and response to changes in the business quickly.

1. **Interoperability:** The application components should have ability to communicate and interoperate with each other regardless of the language tools and frameworks they are built with or platforms that they are running on. This will open up the vast possibility of integrating components built with different tools and technologies.
2. **Scalability:** The application should be able to scale itself should the need arise. This means that introduction of more infrastructural resources should proportionally enhance the system’s capacity to compute, communicate and response to the requests. This will ensure that businesses are capable of meeting quickly a sudden rise in demand.
3. **Evolvability:** The different component of the software system should be able to evolve independently. This will allow new features can be added quickly to keep up with the business’s market demands. Independent evolvability reduces the deployment and regression testing cost as only the changed components need to be tested and deployed.
4. **Visibility:** It should be possible that system performance and failures are visible without need for exposure of internal implementation details. This enables the addition of monitoring tools, load balancers and intelligent gateways without fearing the disclosure of proprietary estate.
5. **Reliability:** The system should be able to recover from full or partial failure and support graceful degradation should the need arise.
6. **Efficiency:** The system should be able to make efficient use of resources. It should be able to keep the server load to the minimum so the server component can manage their own resources efficiently.
7. **Performance:** Systems should be able to deliver responses to the requests quickly with minimum delay and improve overall perceived performance of the application.
8. **Manageability:** Systems should be easily manageable and should support the introduction of management tools.

## Motivation

Distributed Systems are advanced level of the architecture that started from Inter Process Communication (IPC). Different software vendors introduced, from time to time, different frameworks and tools. For example, Microsoft Component Object Model (COM) introduced in 1993 provided IPC and served as basis for some future Microsoft technologies. Java Remote Method Invocation (RMI) and Microsoft .Net Remoting next examples of technologies that worked provided Remote Procedure Call (RPC) capabilities thus influencing software systems to be composed of distributed processes. Enterprise Java Beans (EJB) and Microsoft Windows Communication Foundation (WCF) were further advancements that provided some level of scalability and facilitated development of Distributed Enterprise Systems. Almost all of these technologies were proprietary and hence could not inter-operate. SOAP based web services provided a unified standard that enabled interoperability between software sub systems built with different language tools/frameworks and running on different platform to communicate, coordinate and integrate in logically unified systems. SOAP based web services provided the interoperability but their very nature still had RPC mindset. Server publish service contract using Web Services Definition Language (WSDL). Clients shape themselves to conform to such service contracts. A change in service contract would mean modification and redeployment of all clients – a painstaking and expensive process that inhibited the system agility. None of these technologies elegantly and fully addressed the requirements and challenges mentioned in ‎1.1 above.

In 2000 Roy Thomas Fielding, in his PhD dissertation “Architectural Styles and the Design of Network-based Software Architectures”[[7]](#footnote-7) presented the concept Representational State Transfer (REST). Roy has been working on the WEB and the REST for over six years prior to the publication of his dissertation. Fielding was one of the main developers of HTTP Standard (RFC2616[[8]](#footnote-8)) and URI Specifications (RFC3986[[9]](#footnote-9)) and he developed REST to describe the architectural concepts behind the design of the Web. REST address all of the concerns mentioned above and provided standardised solutions to all of them. This is why RESTful services have now been recognised as generally the most useful methods to provide data-intensive services for web and mobile application development. A large number of business organisations have switched their application architectures to REST and rest are moving towards REST quickly. Therefore, there is strong technical as well as career motivation behind choosing this project to study RESTful architecture in depth and detail.

Next chapters of this report describe REST Architecture in detail and present a sample application architected in REST style to demonstrate the concepts.

## Aims and Objectives

### Aim

The aim of this project is to study and understand in greater detail the concept of RESTful  
architectural pattern of designing web services, thereby evaluating the RESTful services compared to  
other technologies like SOAP services and RPC.

### Objectives

1. To carry out in depth study of REST principles in order to understand how RESTful services differ from other options.
2. To build a data intensive web API software using REST architectural style to help  
   understanding and evaluation.
3. To gain a detailed understanding of the nature of HTTP protocol and how HTTP and RESTful  
   architectures are related to each other.
4. To make use of at least one HTTP client testing tool to gain a practical experience of how  
   HTTP (and thereby RESTful) communication works.
5. To understand the software project life cycle.

# REST – A Better Style for Architecting Modern Applications

## Why REST

The Chapter ‎1 already presents an abstract vision of what lead towards REST. Here we take a rather concrete account of some of key motivations to favour REST more.

### Interoperability

Interoperability refers to the ability to integrate various functional components built using different language tools and frameworks and running on different platforms. For example, a popular news website may have various elements that perform various functions. It may be providing search facility using Google integration, Advertisements provided by Ad Host integration, comments manged by integrating Disqus and sharing using integration of Facebook and Twitter. All these service providers need not be running on the same platform or built using single framework, therefore cannot make any assumptions about each other. Such wide integration requires a mechanism that is simple, consistent and reliable. This makes REST a good fit for the integration of heterogenous system.

### Network-based API vs. Library-based API

Rest comes up with the idea of network-based API rather than the Library-based API. Library-based API is mostly built using certain tools and framework that can run on a single or very few platforms. Network-based API on the other hand is not dependant upon a single platform or development framework. All it requires is to implement certain constraints that provide a standardised way of communication. If implemented correctly, such API offers unlimited interoperability between heterogeneous systems.

### Devices

The last decade saw a huge rise in smart devices. These devices may range from smartphone and tablets to in-car navigation to smart-tv and smart home air conditioning system. The businesses need to reach to the maximum devices. All devices need not be running web application. They have their own operating systems and native applications. In-car navigations system may request traffic data corresponding to GPS coordinates. The smart air conditioning controller may communicate with services provided by local meteorological office to maintain suitable in-house environment. Devices and services are provided by different vendors. They may be upgraded and evolve independent of each other and REST offers such independent evolvability.

### The Cloud

Rise in devices and increased interoperability poses another challenge to the software developers of systems that subscribe capabilities from other service providers and/or publish their own capabilities for other consumers. This opens up the possibility of dramatic rise in number of consumers. Over-subscription may push systems to the limits. To avoid such chaos, organisations tend to favour Cloud that provides elastic infrastructure that has ability to shrink or expand on demand, and they are only charged what they used. This provides financial benefits as well as scalability. But scalability of the could is not something out-of-the-box thing. “It is critical to build a scalable architecture in order to take advantage of a scalable infrastructure.[[10]](#footnote-10)” REST offers such scalable architecture.

## Why Distributed Systems Fail

Due to the rise in versatile smart devices, applications however small and simple tend to deviate from monolithic design and begin to resemble distributed systems. The Chapter ‎1 on Introduction describes in details the characteristics, requirements and challenges of modern software systems. It concludes that REST is a better approach for building modern distributed systems. Here we briefly discuss the key factors contributing to the failure of distributed systems.

### Requirements-Driven Architecture

Traditional RPC based architectures tend to take requirement driven approach. Business requirements are identified and software are designed to fulfil those requirements. When business requirements grow, design is grown to cover those new requirements. Such software designs are conceived and tested in a controlled environment and then deployed in the real environment. It is only after the deployment to the real environment that the limitations of the real environment are discovered that reduce the usability of the design, and eventually resulted in system failure. In requirements driven approach the architecture of the application is shaped to conform the business domain and result is an architecture that is tightly moulded within the shape of business.

### Fallacies of Distributed Systems

In 1994, at Sun Microsystems, Peter Deutsch identified seven assumptions that most of the architects of the distributed system tend to make. In 1997, James Gosling added another such fallacy[[11]](#footnote-11). Howard Dierking, a Pluralsight author, identifies yet another fallacy[[12]](#footnote-12). Unfortunately, these assumptions prove wrong in the long run, hence causing the system to fail. In literature these assumptions are known as Fallacies of Distributed Systems and are listed below (7 from Peter Deutsch, one from James Gosling and one from Dierking):

#### Network Reliability

Fallacy is that network is reliable. Obviously, this is not true. Power failure, hardware failure, people tripping over the cable – a whole lot of reasons to compromise network reliability.

#### Latency

Fallacy is that latency is zero. In one local area it might not be seen as a problem. But what if user is on other side of the globe? Even in case of local users, they might be using mobile devices with delayed response.

#### Bandwidth

Fallacy is that bandwidth is infinite. Although we now have much greater bandwidth then we ever had, but it is still finite. This is particularly true in the case of mobile device, where even there is large bandwidth, user may be charged for the bandwidth they use.

#### Security

The fallacy is that network is secure. Not all networks secure by default. And then this is the most overlooked aspect in practice of software development.

#### Network Topology

The fallacy is that the network topology never changes. In the modern Internet world, this is absolutely untrue. Servers and intermediaries keep moving. DNS, IP address and URLs keep changing. Even the relative paths and query strings at the server keep changing. Topology also regularly changed when applications are scaled and more hardware is added.

#### Administration

The fallacy is that there is one administrator. Obviously, this is not true in the case of applications distributed over the network. For example, when a remote or third-party service fail, local administrators have now access or control to diagnose the problem.

#### Transport Cost

The fallacy is that the transport cost is zero. This is also overlooked aspect during the development of the application. It is assumed that setting up a hardware and network infrastructure has zero cost or at least is one off cost. While in fact such cost is regular. Things like maintenance needs, upgradation, load balancing, bandwidth cost need for scalability contribute to regular cost.

#### Heterogeneous Network

The fallacy is that all nodes on the network is same. This is also false. We have seen already that in the modern world of computing, network and devices are not some. They are heterogeneous, particularly with the rise of variety of mobile devices.

#### Complexity

This fallacy assume that consumers of our service have enough domain and context knowledge of our service such that they will use our services correctly. This is obviously not true. In today’s Internet world, publisher and consumer may not know each other or may not have proper level of technical support available.

## How REST Mitigates Failures

In his dissertation, Fielding identifies the forces to be consider which are closely inline with the fallacies discussed above. He took into consideration such forces while defining the REST.

### Constraint-driven Architecture

Contrary to requirement driven design REST on the other hand advocates constraint-driven architecture. It begins with identifying the forces that could impact the system usability. It then defines and apply constraints on the architecture design such that the impact of those forces can be eliminated or at least minimised. REST, therefore, requires the software developer to map business domain on to the architecture design. The resulted product is a design that works with rather than against those forces.

1. https://www.merriam-webster.com/dictionary/information%20technology [↑](#footnote-ref-1)
2. http://www0.cs.ucl.ac.uk/staff/ucacwxe/lectures/ds98-99/dsee3.pdf [↑](#footnote-ref-2)
3. Service-Oriented Architecture (SOA) Compass: Business Value, Planning, and Enterprise Roadmap Bieberstein, Bose, Fiammante, Jones, & Shah, 2006 [↑](#footnote-ref-3)
4. https://www.pluralsight.com/courses/microservices-architecture [↑](#footnote-ref-4)
5. https://smartbear.com/learn/api-design/what-are-microservices/ [↑](#footnote-ref-5)
6. https://martinfowler.com/articles/microservices.html [↑](#footnote-ref-6)
7. https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding\_dissertation.pdf [↑](#footnote-ref-7)
8. https://tools.ietf.org/html/rfc2616 [↑](#footnote-ref-8)
9. https://tools.ietf.org/html/rfc3986 [↑](#footnote-ref-9)
10. https://media.amazonwebservices.com/AWS\_Cloud\_Best\_Practices.pdf [↑](#footnote-ref-10)
11. http://java.sys-con.com/node/38665 [↑](#footnote-ref-11)
12. https://app.pluralsight.com/library/courses/rest-fundamentals [↑](#footnote-ref-12)