**ISM 6930: Big Data and Ecommerce**

Fall 2015

**Big Data Reference Architecture**

**On demand Online Video Streaming Portal (Netflix)**

**Manohar Lahoti**

**Tanay Bhalerao**

Contents

[Introduction: 1](#_Toc434585415)

[Design Parameters: 3](#_Toc434585416)

[Performance: 3](#_Toc434585417)

[Scale: 3](#_Toc434585418)

[Availability: 4](#_Toc434585419)

[Reference Business consideration: NETFLIX 6](#_Toc434585420)

[Functional and Technical Viewpoints 7](#_Toc434585421)

[Functional view of the design parameters: 7](#_Toc434585422)

[Technical view of the design parameters: 7](#_Toc434585423)

[Big Data Architectural Design for an On-demand online streaming video portal 9](#_Toc434585424)

[Architecture: 10](#_Toc434585425)

[Storage: 11](#_Toc434585426)

[Event driven Computation Framework: 12](#_Toc434585427)

[Content Delivery Network: 13](#_Toc434585428)

[NoSQL: 14](#_Toc434585429)

[Computational Engine: 15](#_Toc434585430)

[Real Time Computation System: 16](#_Toc434585431)

[Batch Processing Computation System: 16](#_Toc434585432)

[Caching System: 17](#_Toc434585433)

[Messaging System (Pub-Sub Architecture): 18](#_Toc434585434)

[Runtime Environment for Web Applications: 20](#_Toc434585435)

# Introduction:

On- demand video streaming is an interesting ecommerce business with millions of prospective consumers. These applications are massively complicated systems which need to be available for their consumers at all points of time. They need to deliver the large video content at a high speed at different geographical locations. And last but not the least they need to ensure the full customer satisfaction through interesting recommendations, quick feedback systems, a large library to make a choice from and secured!

If we plot this business across the parameters of speed and scale, then the complexity and problems of this application fall in the region where both the hardware and the software can fail. This is a zone where the application is facing a rapidly changing demand and a large volume of consumers to sustain.

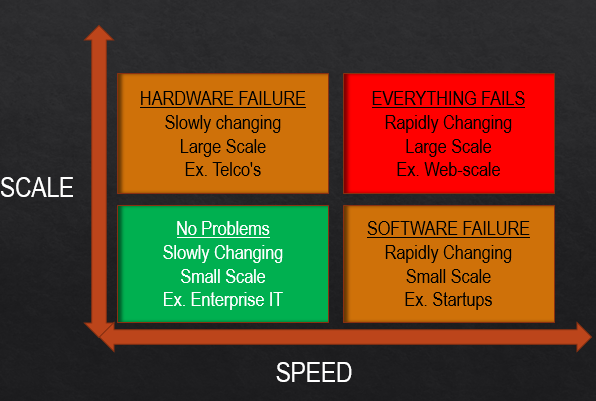
So we are designing a platform for on-demand video streaming which is:

**Web-scale:** Millions of transactions/customers. With web-scale you have both the need and the resources to be able to reach for a kind of robustness, redundancy and availability which needs to be handled.

**Across the country (U.S.):** Serves customer across the big country. There is a peak time across the country at all the times. There is no good time for a downtime for database maintenance, backup or schema change.

**Highly-available:** Since the business needs to be highly available then ideally the service should always be up and running.

**Consumer-facing:** The systems iterates continuously by making changes and building features that consumers care about. And those changes are potentially open to bugs, failures and faults.



Looking at the above assumption graph which talks about scale and speed of change. The bottom quadrant deals with systems but doesn’t serve millions of customers (may be 1000s of customers) and the changes might happen on a monthly or a weekly basis. It continues to work since it is not experiencing sudden growth or changes. As we move up the scale axis, we need to deploy the application on more and more hardware (scaling up) which in turn opens new opportunities for the hardware to fail. For example, a telco application has a huge scale but the requirements don’t change that often. Along the bottom axis, we can think of a startup who is growing rapidly and learning what their customer and business needs and making those changes in their systems which are again open to bugs and failures. Usually software faults are predominant in this space.

Then we move to web-scale- millions of subscribers, rapid iterations, lots of change, and lots of hardware. This is where everything fails and we are designing a reference architecture considering such challenges.

# Design Parameters:

The main goals for our platform are availability, scale and performance.

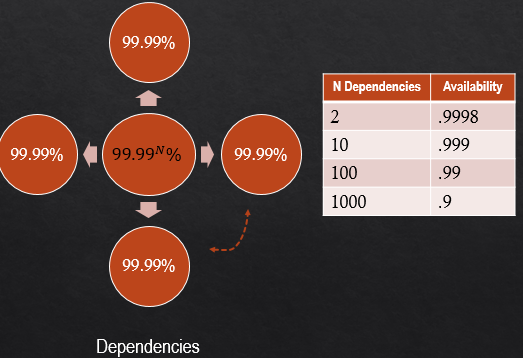
## Performance:

If we consider performance, if an on demand video streaming platform takes around 20-30 seconds to come up on your screen or that home page to load. Even if the performance is improved by 1-2 seconds the scale which it is applying to can literally save one human life time every day. When the user sits in front of his TV to decide which platform to use to watch a movie – HBO, Netflix, Hulu etc. The moment of truth is to captivate the user’s decision in those 30-40 seconds of time to use our portal. That is a way we can deliver value. Even a small performance hike across a huge scaled platform can lead to a huge business benefit. Performance can also be considered as showing the user the content he would love to watch and improving these recommendation algorithms even by 1 % can have a huge impact. It has the potential to deliver 500,000 hours/day of additional value.

## Scale:

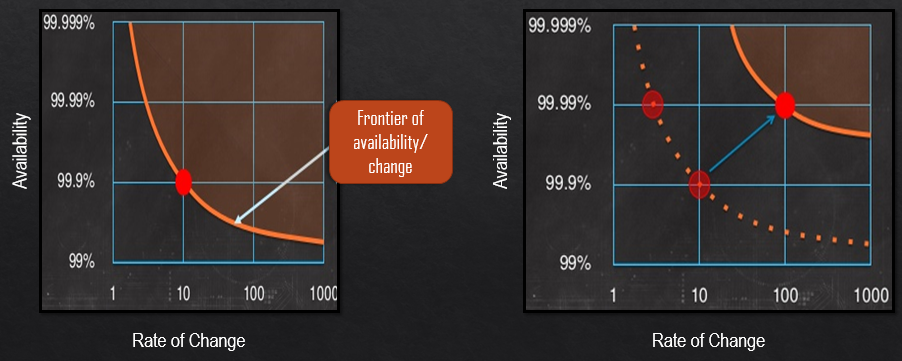
Considering the vast landmass of the United States we want to deliver our content to each of the states with millions of users. Along with the users the volume of the content (digital media) is increasing every day with almost doubling every year. Inspite of all the scale we don’t want a huge cost.

## Availability:



It is a dream to achieve 4 9’s in the availability quadrant in terms of serving content to the subscribers even during peak hours. That accounts to approximately having a downtime of 3 minutes a quarter. Disappointed customers calling for availability issues cost much more. If we consider an availability compound. If we have 2 subsystems each of which have 99.99% availability and they are independent. Then our overall system will have an availability of 99.992%. But if we have around 1000 subsystems then we will have 99.991000% which is around 90% available. This can be damaging to the business. If I want to achieve 99.99% availability for my overall system then I need 99.9999% of availability for each of my subsystems. So to break this model we need to isolate these dependencies to avoid cascading failures and avoid propagation of failures in the system.

But sometimes availability, scale and performance are not enough we need to consider the iteration and rate of change. Since we need to push new features, perform A/B Tests, optimize, scale up, and remove redundant features etc. This may account to around 1000 changes a day and every change gives us the opportunity to fail. So if we plot the availability and the rate of change, we realize that we need to shift the curve outwards to avoid any changes causing availability issues.



# Reference Business consideration: NETFLIX

Netflix is an on demand online media streaming. It allows the users to choose from thousands of movie titles as well as full television series on demand through computers, mobile devices, tablets and Netflix ready televisions. It provides recommendations on the basis of your previous using experience. A user can rate the TV shows and movies. Based on the ratings Netflix will recommend more similar movie titles and TV shows. It divides the movies into various sections based on movie genres, new releases, and popularity across Netflix, as well as repeat watching. Instant queue feature allows a user to watch a movie/TV series at a later stage.

There are around 3 million titles with around 60+ million active subscribers in 40+ countries. On an average it has around 10+billion hours of content every month. 1/3 of the downstream traffic in North America during peak hours. 61% of the users binge-watch shows at least every few weeks. 90% of the users engage themselves in watching the original content delivered by Netflix. Netflix needs about 100 to 150 TB of storage per server and the total size of the content library is around 1 petabyte. With all the above statistics Netflix has been able to achieve around 63% votes from customers to be extremely or very satisfied.

It was an interesting business and technical model to look at. It provides design insights to deliver a similar robust and scaled platform which is highly available.

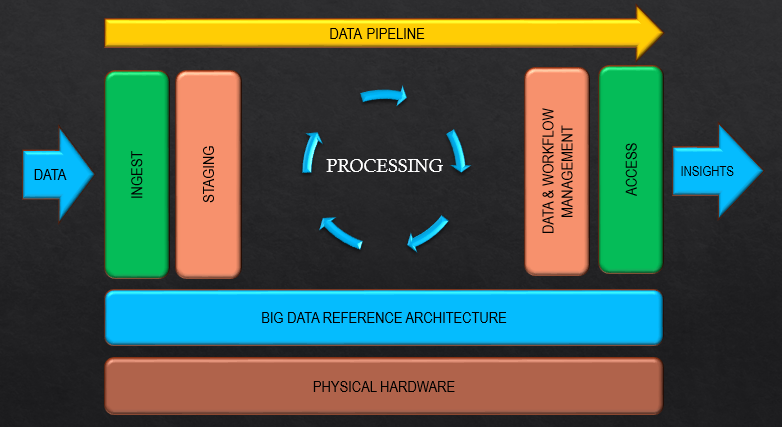
## Functional and Technical Viewpoints

### **Functional view of the design parameters**:

* Uninterrupted viewing of videos even during failure
* Handle large data files in various formats
* Concurrent views of different or the same videos
* Handle different devices for content delivery
* Personalized recommendations for every user

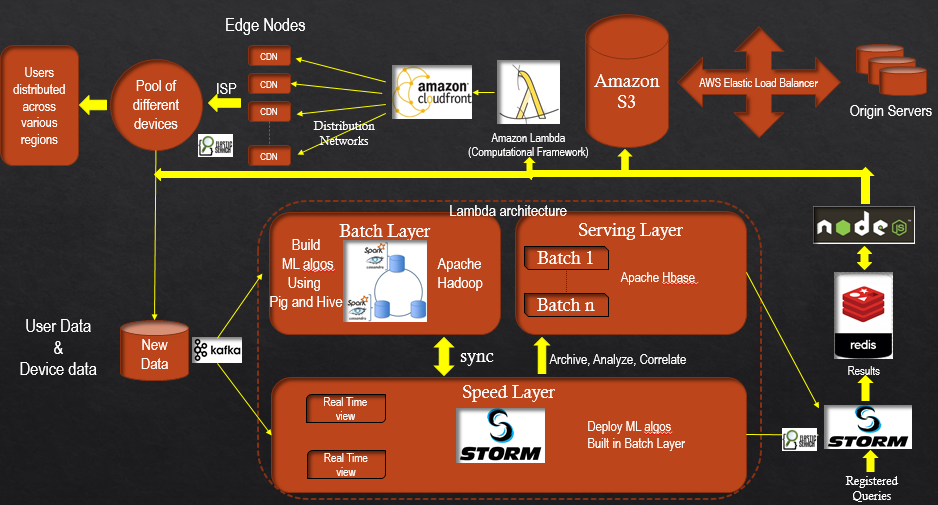
### Technical view of the design parameters:

* **File System:** Distributed file systems which provide storage, fault tolerance, scalability, reliability, and availability.
* **Data Store:** Key-Value, Document, Column and Graph to store data.
* **Resource Manager:** provide resource management capabilities and support schedulers for high utilization and throughput.
* **Coordination:** systems that manage state, distributed coordination, consensus and lock management.
* **Computational Framework:** highly specialized compute frameworks for Streaming, Interactive, Real Time, Batch and Iterative Graph (BSP) processing.
* **Data Analytics:** Analytical (consumption) tools and libraries, which support exploratory, descriptive, predictive, statistical analysis and machine learning.
* **Data Integration:** these include not only the orchestration tools for managing pipelines but also metadata management.
* **Operational Framework:** these provide scalable frameworks for monitoring & benchmarking.



## Big Data Architectural Design for an On-demand online streaming video portal

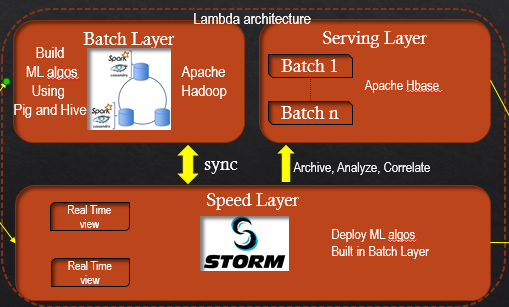
This is the reference architecture which we have designed for implementing the on-demand online streaming video which will have the best scalable architecture, highly available and real time recommendation system

****

### Architecture:

We choose the lambda architecture which is generic, scalable and fault-tolerant data processing architecture. It aims to satisfy the needs for a robust system that is fault-tolerant, both against hardware failures and human mistakes, being able to serve a wide range of workloads and use cases, and in which low-latency reads and updates are required. The resulting system should be linearly scalable, and it should scale out rather than up.

**Following is the functioning and the benefits of the lambda architecture:**

* All data entering the system is dispatched to both the batch layer and the speed layer for processing.
* The batch layer has two functions: (i) managing the master dataset (an immutable, append-only set of raw data), and (ii) to pre-compute the batch views.
* The serving layer indexes the batch views so that they can be queried in low-latency, ad-hoc way.
* The speed layer compensates for the high latency of updates to the serving layer and deals with recent data only.
* Any incoming query can be answered by merging results from batch views and real-time views.

### Storage:

We are using Amazon S3.

**Amazon S3:** Amazon Simple Storage Service is safe, secure and highly scalable object storage in the cloud. It can be used to store and retrieve any amounts of data at anytime from anywhere on the web. This will avoid us in investing a lot of money in buying and managing hardware. Also it will help us in avoiding costs in paying for hardware which is not in use. Amazon S3 provides us with pay as you use facility which can save us a lot of money. We will be using Amazon S3 for all our static content as well as for backups for faster retrieval. It allows us to do versioning which is very much needed for a web-scale application. Thus allowing us to roll back to a previous version while performing changes. They provide standards-based REST and SOAP web services API which can allow us to programmatically store, retrieve and manage our data. Also AWS has regions all over the world which can be utilized to significantly reduce the latency to the end users. We have an option in Amazon S3 to optimize latency, minimize costs and address regulatory requirements.

The alternatives which we evaluated were Google cloud storage and Microsoft azure. We chose Amazon S3 since it was our technology stack consisted of amazon product which lets free data transfer along with the added benefits of durability, scalability and secured transfers.

### Event driven Computation Framework:

We are using Amazon Lambda.

**Amazon Lambda:** AWS Lambda lets us run code without provisioning or managing servers. We need to pay only when the code is triggered and it processes data in real time. It helps in handling:

* Any changes in data
* Shift in systems date
* Peaks in demand
* Some actions

The event driven nature is very helpful for us to automate the scaling up and deployment of the servers. It eliminates computing hardware setup.

The alternatives which we evaluated were Apache Stackstorm and Azure webjobs. Even though these products provide many alternatives and flexibility, our usecases of handling a large data plus a good customer service along with an intuitive management console and a cheaper price compels us to use AWS Lambda.

### Content Delivery Network:

We are using Amazon Cloudfront.

**Amazon Cloudfront:** Considering that our website is streaming large video files and known to encounter heavy traffic at random hours and situations. We need to serve the consumer with the lowest latency. Amazon cloudfront is the content delivery network we have chosen for our service which is spread across the U.S.

Benefits of Amazon cloudfront:

* Around 40 edge locations. The point of presence is good for our website.
* The average response time is around 120ms with the fastest response being 80ms and slowest being 159ms.
* It can help us in targeted advertising to be computed on the fly based on cookie or query string data.
* There are no platform fees and it is a pay per use service. Also there is not minimum commitment.
* Also it provides access through SSL and also provides versioning of data.
* It helps us in providing information about the devices, locations, detailed cache stats reports, most popular objects and other operational real time metrics.

After comparing with similar services, Akamai is an expensive enterprise software which doesn’t provide pay per use. Akamai CDN has a better PoP these days but with the internet architecture innovations this dependency has reduced. The setup of Akamai is complex compared to Amazon.

### NoSQL:

We are using Apache Cassandra.

**Apache Cassandra:** It is a fully distributed, highly scalable database which allows users to create online applications are always available and can process large amounts of data in real time. It can also be used for read-intensive database for large scale business intelligence applications. This offers no single point of failure. It is a column oriented database.

We are using apache Cassandra to store all the critical user data. It is used along with apache spark for better performance. It is part of the lambda architecture and forms the batch layer where all the machine learning algorithms for our recommendation system are built on the basis of the user and device data. This is done through running Hadoop map reduce jobs on the data.

Also analytics are built on the data stored in Cassandra.

We chose Cassandra over other options like MongoDB, CouchDB, Bigtable because as per the CAP theorem we were more focused on making our system available and partition tolerant over being consistent. Eventual consistency was a reasonable tradeoff for us.

### Computational Engine:

We are using Apache Spark.

**Apache Spark:** Apache spark is a processing engine that enables applications in Hadoop clusters to run upto 100 times fasters in memory and even 10 times faster when running on disk. It is good for recommender systems where we need to profile and provide each individual user.

We compared the Apache spark engine with Hadoop and realized the following benefits:

* Spark has designed to run on top of Hadoop and it’s an alternative to the traditional batch map/reduce model that can be used for real time stream data processing and fast interactive queries.
* Spark uses Resilient Distributed datasets (RDDs): This provides fault tolerance. It removes the need for replication to achieve fault tolerance.
* It is a lot faster than Hadoop
* It helps in combining SQL, streaming and complex analytics.
* Spark runs on diverse data sources including HDFS, Cassandra, Hbase and S3

All these benefits satisfy our business usecases as well as support our technology stack.

### Real Time Computation System:

We are using Apache Storm.

**Apache Storm**: is used as a real time computation system to reliably process unbounded streams of data. Storm is used for real time processing similar to Hadoop for batch processing.

We are using storm in the speed layer to transform the unstructured data as it flows/enters a system into a desired format for further analysis. Apache storms helps us do task parallel computing.

We compared apache storm and apache spark streaming for our usecases. We came to a conclusion that Storm is a good choice for sub-second latency and no data loss. Spark streaming is better if we need stateful computation, with a guarantee that each event is processed only once. Also storm is known to be run in production for a longer time than Spark Streaming.

### Batch Processing Computation System:

We are using the Hive Queries and Pig scripts on Hadoop for computation in batch processing

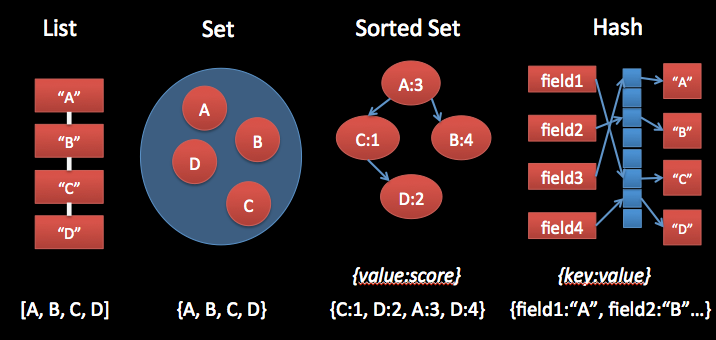
**Hive Queries and Pig scripts on Hadoop** can give us the appropriate environment to implement the ETL on the data in batches. Innovative reports and dashboards can be designed to get the insight from the data. Hadoop is the best platform for distributed computation for huge dataset. We can also implement the machine learning and data mining algorithms on the data to create the predictive models which can be used for our recommendation engine. For the data analysis and machine learning algorithm, we can use Mahout or H2O libraries on Hadoop

### Caching System:

We are using Redis implementing in-memory caching system.

**Redis:** It is an open source advanced key-value store. Redis is often known as the data structure server since keys can contain strings, hashes, lists, sets and sorted sets. These all 6 types of complex data structures are being used to store the complex metadata about the titles, movies and other media related information. Redis is an in-memory storage, superfast, persistence cache system which supports the complex data types. Redis has provided the facility of data replication and sharding. It can also be distributed which does not give any instance for single point failure.

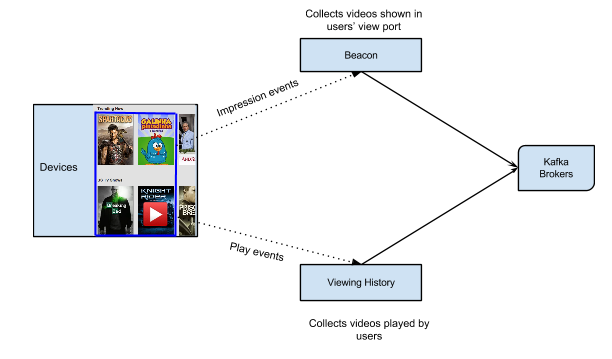
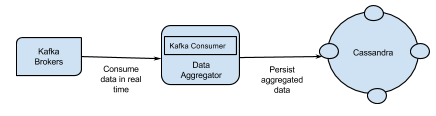
We compared Redis with Memcachd as in-memory cache storage was our moto. Memcachd is also key-value store but Memcachd is effective and efficient for simple keys. It does not work effectively for complex keys like map and hashes. Hence that is the primary reason to choose Redis over Memcachd. Another reason to choose Redis over Memcachd is the provision of better Data Replication and Data persistence. Redis provides the advanced and efficient data replication and data persistence.

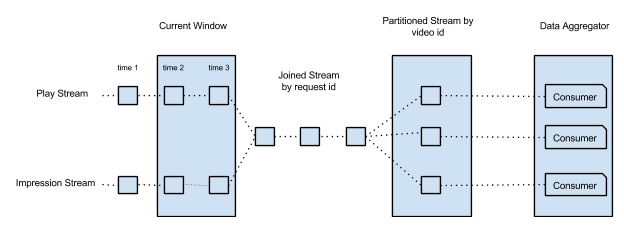


### Messaging System (Pub-Sub Architecture):

We are using Kafka for distributed messaging system

**Kafka**: Apache Kafka is publish-subscribe messaging system which can also be considered as distributed commit log. We have to generate the data pipeline for on demand video streaming data as number of users can view the multiple videos simultaneously which will create the humungous clicks per minute. So it is important to publish, collect, aggregate and move the data through pipeline so that we can manage the scenario where the input data is coming very fast compared to the consumers who accepts the data. Hence we have to streamline the data using persistent messaging system.

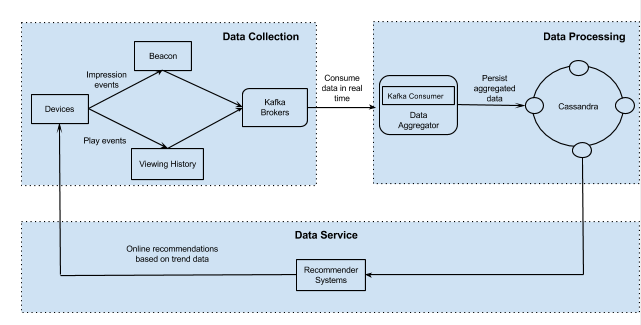




Apache Kafka differs from RabbitMQ and other traditional messaging system as:

* Distributed system which is very easy to scale out.
* High throughput for both publishing and subscribing.
* Multi-subscribers and balances the consumers during failure.
* Messages persistence on disk and hence can be used for batched consumption like ETL, in addition to real time applications.

After bringing all the components which we have discussed before, we can get the overall messaging architecture as below



### Runtime Environment for Web Applications:

We are using Node.js as a runtime environment for web applications

**Node.js:** It is an open source which provides us the runtime environment for developing the server side web applications. Nod.js is an event driven module which uses only single thread for completing the tasks. Because of single thread, it is monolithic and is recognized as a non-blocking IO model which is used for real time data processing for multiple users. It avoids the communication and synchronization overhead by using the single thread loop. JavaScript is an inevitable part of Node.js and it has been widely used hence Node.js provides the easy way to develop the web applications. Node.js is fast and reduces the development effort and cost considerably. Node.js is highly scalable which can support a large number of users without using the multi-threading. This makes the Node.js quite unique as we do not need to deal with concurrency issues

We have chosen the Node.js as the runtime environment because our on demand video streaming applications will have millions of users who will be simultaneously using the application in real time. As node.js is highly scalable, non-blocking IO, single threaded and event driven model, it will work quite efficiently to run the web applications on server side. PHP, Pearl, Ruby all these runtime environments need the multiple threads and synchronization to manage the large number of tasks and users. They have to deal with the frequent context switching. There is a high cost associated with the context switching. Hence to avoid all these issues and to provide the best efficient web application solution, we have used Node.js