### **INTRODUCTION**

**1.1 Theoretical Background**

Zopper is an hyper local e-commerce marketplace where consumers find products and check the offline prices. Zopper lets users search for products across categories and check prices and place an order from nearby stores. The projects I have worked on are:-

* Setting up a simple Digital-Wallet system which can keep track of all transactions performed by a customer or a retailer.
* Implemented an Enterprise Service Databus(ESB) called AESOP which was made by Flipkart and manufactured by LinkedIn.
* Implemented a ‘Source from Zopper’ functionality in which Zopper will work with a network of regional and national level distributors to make their pricing and inventory available to retailers in real time.
* Implemented the desktop version for Zopper.
* Implemented a dedicated queueing system to process orders in large number for Large Format Retailers(LFR) like croma,etc.

The concepts required to be grasped by me for the project completion were:

**1.1.1 Django Framework**

Django is a high-level Python Web framework which encourages rapid development and clean, pragmatic design. It is used and built by experienced developers which takes care of Web development. We chose Django framework as we needed to work in different applications which is provided in Django framework as it separates a project into individual applications.

**1.1.2 DjangoRest Framework**

REST (Representational State Transfer) is a style which is utilized for web

development services. The REST is preferred over SOAP as it consumes lesser

bandwidth and can be used easily over internet. We preferred REST over other

technologies like SOAP because:

● RESTful services are more scalable than SOAP.

● It is easier to make use of REST as it can be used from within applications much faster. REST not only saves time but also money.

● We needed a style which is fast enough as well as cheap. REST uses a smaller

message format than SOAP which makes it much faster than traditional SOAP.

● REST is preferred for use over the Open Internet/Web thus it is a better choice for web scale applications, and certainly for cloud-based platforms.

**1.1.3 HTML, CSS, JAVASCRIPT**

HTML is the standard markup language which can be used to create web pages. It describes the content of a webpage and looks are added using CSS which has elements to make the page look more presentable. Javascript is supported by all the web browsers and support object-oriented, imperative, and functional programming styles.

**1.1.4 SQL**

Structured query language, we used SQL as a language for moving data into and out of a RDBMS.

**1.1.5 Java**

Apache Maven is a software project management and comprehension tool coded in JAVA. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.It allows a developer to comprehend the complete state of a development effort in the shortest period of time.

**1.1.5 Other Tools**

The other tools which were required in this project are:-

* Apache Zookeeper
* Apache Kafka
* Maven
* Redis(A NoSQL key-value datastore)
* Twemproxy
* Django Celery
* Queueing systems such as RabbitMQ

**CHAPTER - DIGITAL WALLET**

**2.1 Introduction**

A digital wallet refers to an electronic device that allows an individual to make electronic commerce transactions. This can include purchasing items on-line with a computer or using a smartphone to purchase something at a store. An individual's bank account can also be linked to the digital wallet. They might also have their driver’s license, health card, loyalty card(s) and other ID documents stored on the phone. The credentials can be passed to a merchant’s terminal wirelessly via near field communication(NFC). Increasingly, digital wallets are being made not just for basic financial transactions but to also authenticate the holder's credentials. For example, a digital-wallet could potentially verify the age of the buyer to the store while purchasing alcohol.

**2.2 Need at Zopper**

As an e-commerce company, Zopper introduced it’s own wallet system to reduce the hassle of filling out long forms every time an item is purchased on our platform. Consumers are not required to fill out order forms on each site when they purchase an item because the information has already been stored and is automatically updated and entered into the order fields across merchant sites when using a digital wallet. Consumers also benefit when using digital wallets because their information is encrypted or protected by a private software code; merchants benefit by receiving protection against fraud. They provide means to transfer confidential information and money securely as they do not pass any payment card details to the website. This Digital-Wallet was made exclusively to record to transactions of stores.

**2.3 Design**

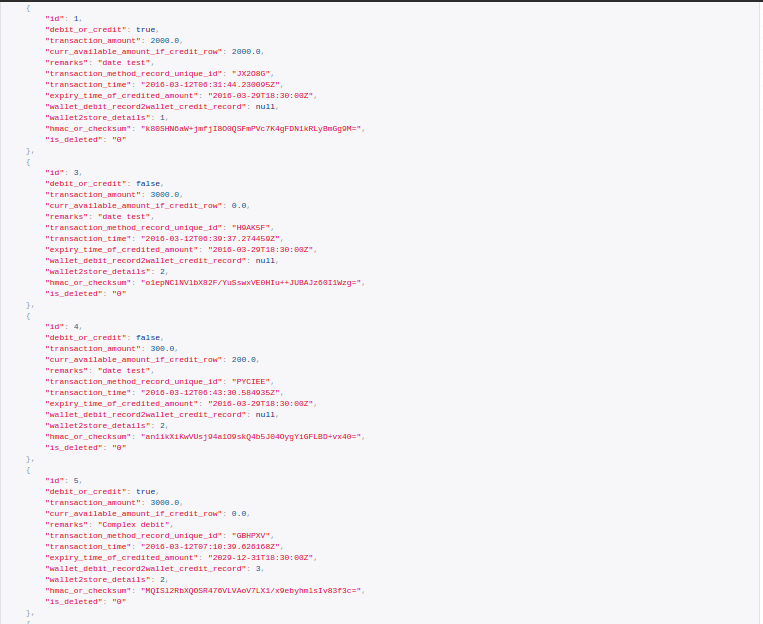
The Digital-Wallet consists of the following features:-

* Debit\_or\_credit :- Boolean field which depicts if a transaction if a credit(True) or a debit(False)
* *Transaction\_amount* :- Specifies the amount for every transaction.
* *Curr\_available\_amount\_if\_credit\_row* :- Specifies the amount in the current record if it is a credit transaction.
* *Transaction\_method\_record\_unique\_id* :- A unique custom-generated ID for every transaction to identify each transaction.
* *Wallet\_debit\_record2wallet\_credit\_record* :- A foreign key reference to itself, this field is used to link each debit transaction to a credit transaction to ensure that there is no ambiguity in the records kept by the company.
* *Wallet2store\_details* :- A unique number used to map a wallet to a particular store.
* *Hmac\_or\_checksum* :- A hashed value computed using SHA-256 algorithm to check for unauthorized transactions such as manual addition of credit to wallet,etc.

**2.4 Implementation**

The wallet has been implemented using DjangoREST framework to create RESTful API’s which can be easily consumed by any other application. The API’s are mapped to certain URL’s and can be accessed using GET,POST and PUT methods. The fields to be passed in the POST request are:-

*Store\_ID,Amount,Method(Debit or Credit),Remarks,expiry\_time(only for credit)*



**Figure 2.1**

**GET function to list all available wallets**

### 

### **CHAPTER - AESOP**

**3.1 Introduction**

AESOP in a Enterprise Software Bus(ESB) build by Flipkart on top of the LinkedIn Databus and Google Open Replicator. Overall, it is a complete *Change Data Capture System.* In Internet architectures, data systems are typically categorized into:-

1. *Source-of-truth systems* that serve as primary stores for all the user generated data.
2. *Derived data stores* or indexes which serve reads and other complex quenes.

The data in these secondary stores is often derived from the primary data through custom transformations, sometimes involving complex processing driven by business logic. Similarly data in caching tiers is derived from reads against the primary data store, but needs to get invalidated or refreshed when the primary data is changed. The requirement for these kind of data architectures is the need to reliably capture, flow and process primary data changes.

AESOP is a keen observer of changes that can also relay change events reliably to interested parties. It provides useful infrastructure for building Eventually Consistent data sources and systems.

**3.2 Need for a Databus**

Large scale internet systems often use a combination of relational (SQL) and non-relational (NoSQL) data stores. Contrary to product claims, it is hard to find a single data store that meets common read-write patterns of on-line applications. Different databases try to optimize for specific workload patterns and data durability, consistency guarantees - use Memory buffer pools, Write-ahead logs, optimize for Flash storage etc. These data stores are not operated in isolation and need to share data and updates on it - for e.g. a high performance memory based KV data cache might need to be updated when data in the source-of-truth RDBMS or Columnar database changes.

Data from the primary store is used to feed more than just business decisions. At Zopper, it also feeds real-time search indexes, real-time retailer pricing indexes, cache coherency, Database Read Replicas,real-time seller ranking etc... These are examples of Zopper’s near-real-time data needs. Old systems use the ETL(Extract, Transform and Load) technology to propagate these changes. The industry typically uses ETL to run nightly jobs to give executives a view of the previous day's, week's, month's, year's business performance.

It is difficult to ensure uniform and consistent data replication across all sources. This problem further increases as the number of sources grow due to addition of several factors such as latency,etc.

Polyglot persistence is key to building highly scalable,consistent and dependable systems. There are a variety of data stores used for any large scale project. Ensuring consistency,persistence and availability between these varied data stores is difficult. Two of the main challenges for polyglot persistance are:-

1. *Caching/Serving Layer Challenges*

* The strategies used are Low Cache TTL’s, Lazy Caching and High Request Concurrency. In Lazy Caching, we begin with an empty cache and as the user requests start coming in, we load data into the cache from the primary data stores. To keep the data fresh(i.e up-to date), low TTL’s are set on the cache to increase consistency. If the requests increase for a particular resource that is not already loaded in the cache, all requests will access the same primary data store, creating the Thundering Herds problem. It will also expose the availability of the primary data store.
* The cache size can be increased. Distribution of data in cache also has to be implemented. Due to this, multiple copies of the same data will exist. Due to this, it will be difficult to implement write-through mechanisms to maintain consistency.

2. *Eventual Consistency*

It refers to the gradual process of all updates being replicated in all the database sources

Pros:

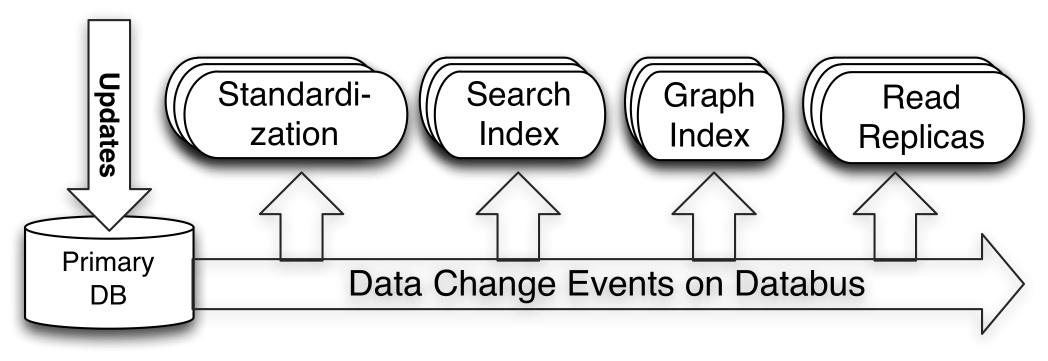
* Scale reads through multiple replicas.
* Higher overall data availability.

Cons:

* Reads return live data before convergence.i.e two simultaneous requests to the same resource may return two different values.

Achieving eventual consistency requires *atleast once* delivery guarantee of updates to all replicas.

The AESOP transport layer provides end-to-end latencies in milliseconds and handles throughput of thousands of change events per second per server while supporting infinite lookback capabilities and rich subscription functionality.

**Figure 3.1**

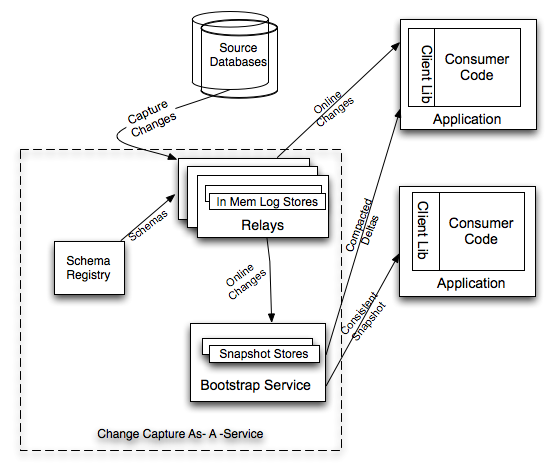
**Databus Introduction**

As shown above, systems such as Search Index and Read Replicas act as Databus consumers using the client library. When a write occurs to a primary OLTP database, the relays connected to that database pull the change into the relay. The databus consumer embedded in the search index or cache pulls it from the relay (or bootstrap) and updates the index or cache as the case may be. This keeps the index up to date with the state of the source database.

The features offered by AESOP are:-

* **Source-independent:** AESOP supports change data capture from multiple sources including Oracle and MySQL. The Oracle adapter is included in our open-source release. We plan to open source the MySQL adapter soon.
* **Scalable and highly available:** AESOP scales to thousands of consumers and transactional data sources while being highly available.
* **Transactional in-order delivery:** AESOP preserves transactional guarantees of the source database and delivers change events grouped in transactions, in source commit order.
* **Low latency and rich subscription:** AESOP delivers events to consumers within milliseconds of the changes being available from the source. Consumers can also retrieve specific partitions of the stream using server-side filtering in Databus.
* **Infinite lookback:** One of the most innovative components of AESOP is the ability to support infinite lookback for consumers. When a consumer needs to generate a downstream copy of the entire data (for example a new search index), it can do so without putting any additional load on the primary OLTP database. This also helps consumers when they fall significantly behind the source database.

**3.3 Design of AESOP**



**Figure 3.2**

**Outline of AESOP**

The main components of AESOP are:-

1. *Producers:-* They comprise of the data sources which store the primary data for the application. Currently the producers include MySQL database and HBase. These data stores support a kind of transaction log. The producers use the log-mining approach. Write Ahead Logs(WAL) ensures
   1. Each modification done is flushed to disk.
   2. Log records are stored in order.
2. *Relay:-* The Relay is a data store agnostic memory-mapped circular ring buffer. It contains the data update events which are processed from the binlogs generated in Avro Format. It also maintains subscriptions for multiple consumers.

It is similar to a broker in a pub-sub system. The relay has been advanced

AESOP for configurability,metrics collection and provide an administrator console.

1. *Consumers:-* They consist of subscribers running at different rates. They register themselves on the relay, they get the data delivered to them, transform the map and finally deliver the data to their respective destinations.

They are basically sinks for all the change events. Consumers have been advanced in AESOP to accomodate bootstraping,configurability and data transformation.

On an abstract level, AESOP is like a traditional publish-subscribe system. Because of the design of consumers, we can possibly have data generated from a SQL data store such as MySQL and transfer the same data to a NoSQL datastore such as Redis,MongoDB,etc.

Aesop uses the **log mining approach** of detecting data changes. It also uses the infrastructure components of Databus, mostly for serving change events. The concept of Event Producer, Relay, Event Buffer, Bootstrap, Event Consumer and System Change Number(SCN) are quite appealing and used mostly as-is in Aesop.

Aesop extends support for change detection on HBase data store by implementing an Event Producer based off NGDATA hbase-sep. For MySQL, Aesop builds on the producer implementation available in Databus which internally uses Google Open Replicator.

The log mining producer implementations leverage master-slave replication support available in databases. Such producers may be called "Push" producers where changes are pushed from the master to slave (the Aesop Event Producer).

The log mining approach has limitations if data is distributed across database tables (when updates are not part of a single transaction) where-in it is hard to correlate multiple updates to a single change. It also has limitations when data is distributed across types of data stores - for e.g. between an RDBMS and a Document database. Aesop addresses this problem by implementing a "Pull" producer that uses an application-provided "Iterator" API to periodically scan the entire datastore and detect changes between scan cycles. This implementation is based off *Netflix Zeno.*

ChangeEvent.png

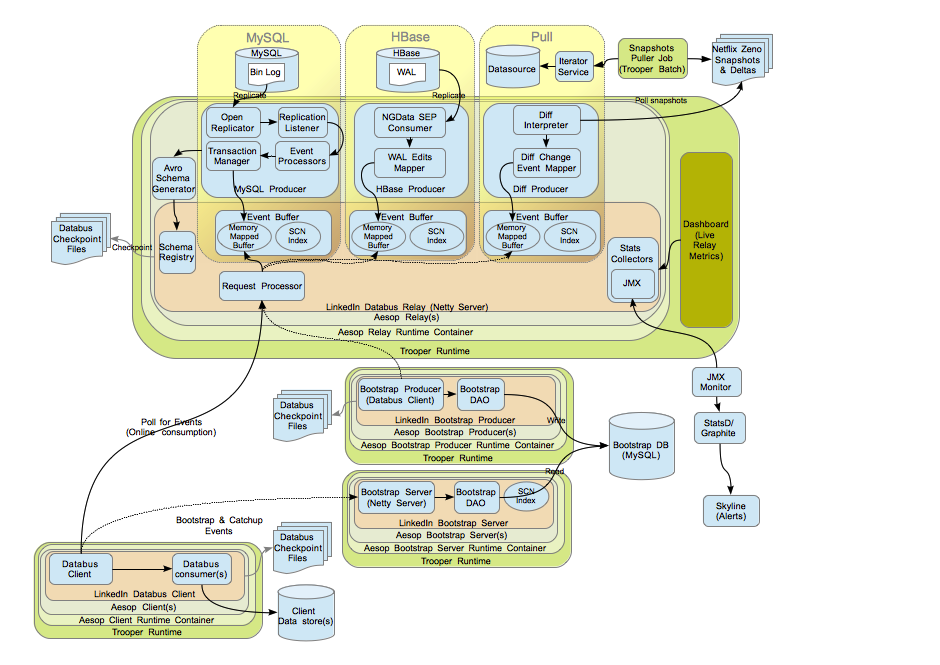
**Figure 3.3**

**Change propagation employing both ‘Push’ and ‘Pull’ producers**  
(e.g. HBase WAL edits listener,  
 e.g. MySQL Replication listener)

*Technical Stack of AESOP*

* LinkedIn Databus
* Google Open Replicator
* Netflix Zeno
* Apache Helix,Zookeeper and Kafka
* ElasticSearch

*Logical Architecture of AESOP*



**Figure 3.4**

**Logical Architecture**

**3.4 Change Propagation Steps**

The typical change propagation flow in the pipeline is as follows:

* An Aesop Relay is started with one or more configured Event producers.The Trooper runtime picks the configuration files and loads Relay and Event Producer instances defined in these files. Each such configuration file is managed as a separate deployment module by Trooper.
  + Each Relay's configuration is defined by an instance of *RelayConfig*. It contains information about such details as listening port, Event buffer allocation policy, Location for storing checkpoints etc.
  + An Event Producer is associated with a Relay using source configuration i.e.*PhysicalSourceConfig* containing one or more *LogicalSourceConfig* definitions. The Relay configuration file contains these definitions.
  + The Relay instance is created by a Factory i.e. *DefaultRelayFactory*. This factory takes a relay configuration and one or more instances of *ProducerRegistration* that associates an Event producer with a source configuration.
* Each Event producer is started by the Relay after initialization using the last seen SCN (System Change Number). The Event producer implementation executes cycles of event sourcing/production and at the end of each cycle, checkpoints the last seen SCN.
* The event format varies by source type and usually requires a mapper for transforming it from the source format (bin log or WAL) to the Avro defined type. The Avro type and schema is picked up from the Relay's schema registry based on the type name specified in *LogicalSourceConfig*. The mapped Avro objects are appended to the Event buffer created by the Relay that started the Event producer. The change events are now ready for consumption.
* A Aesop Client is started with one or more configured Event consumers. The Trooper runtime picks the configuration files and loads Client and Event Consumer instances defined in these files. Each such configuration file is managed as a separate deployment module by Trooper.
  + Each Client's configuration is defined by an instance of *ClientConfig*. It contains information about such details as listening port, properties for data puller, Location for storing checkpoints etc.
  + An Event Consumer is associated with a Client using *RelayClientConfig* that points to a running relay and the Avro change event type that the consumer is interested in . The Client configuration file contains these definitions.
  + The Client instance is created by a Factory i.e. *DefaultClientFactory*. This factory takes a client configuration and one or more instances of *ConsumerRegistration* that associates an Event consumer with a source configuration.
  + The Client configuration may optionally also have a *BootstrapClientConfig*. This configuration refers to a Bootstrap server that may be used by the client for cold starts or when it falls off the relay. The client consumes snapshots from the Bootstrap server to catchup with the Relay and then switches to online consumption mode.
* Each Event consumer is started by the Client after initialization using the last seen SCN (System Change Number). The Event consumer listens to a callback event for each change event consumed off the Relay or Bootstrap server. The event payload is of the Avro change event type.
* Each consumer may process the change event differently. Real consumers in a change propagation pipeline might write it to a downstream data store, log file or secondary index to enable searching.
* The Bootstrap producer is a special kind of Event consumer that writes the change events to the MySQL snapshots database. This database is used by the Bootstrap server to serve data to consumers as described above.
* The Aesop and Databus statistics collectors collect metrics and expose them via JMX. The dashboard uses these metrics to provide a real-time view of the change propagation pipeline. Some users of Aesop also consume the JMX metrics directly and push these to time series databases such as StatsD/Graphite and enable alerting using tools like Skyline.

**3.5 Benefits of Design Abstraction**

The component separation and design abstraction has these benefits:

* Ability to reuse existing Producers, Relays and Client interfaces and implementations to create a change propagation pipeline where only event mapping and transformations need to be implemented.
* Configuration driven deployment of multiple producers, relays or consumers on a server container instance. Life-cycle isolation across multiple components deployed on the same runtime.
* Interface abstraction for most components. Enables creation and use of database specific change event producers (e.g. Postgres) and Consumers (e.g. indexing MySQL data on Lucene/Solr/Elastic Search)
* Start with Pull (snapshot based) change propagation and move to Push (replication based) implementation when data freshness matters. Move back and forth between the two implementations while preserving the consumer interface.
* Live monitoring using the Relay dashboard and detailed metrics via JMX.

**3.6 Architecture at Zopper**

architecture3.png

**Figure 3.4**

**AESOP Architecture at Zopper**

The setup of AESOP is as follows:-

1. There are two databases that are being monitored for change events i.e Merchant and Accounts. The avro schema has been generated for the tables we need the monitor in the above databases, namely store\_product\_price in Merchant DB ,which stores all the details about on the Zopper platform, and orders\_orderitem and orders\_ordersummary in Accounts DB which store all the details for each order that is generated.
2. The Kafka Client Consumer has been implemented as the consumer for the change events. They register themselves on the relay to receive these events.
3. The client consumer internally uses fault-tolerant, consistent and high availability clusters of Zookeeper and Kafka. Crucial data such as last consumed SCN,topics for each data source,information needed by the Kafka brokers,etc. Kafka is a stream data platform that captures streams of events or data changes and feeds these to other data systems such as relational databases, key-value stores, Hadoop, or the data warehouse. Kafka creates topics for each database in Zookeeper. The change events for
4. The ‘custom python client’ subscribes to the above topics created by kafka.In the case of Zopper,the python client stores information about the orders in the Redis Database Cluster.
5. Redis Database is a in-memory key-value datastore. The cluster stores information about the products and the all the orders that are placed on the Zopper platform. In the case of products, the key stored in redis in the format *primarykey\_tablename*. The primary key is the product ID. The value for each key in a list with 2 fields which the first field consists of the entire JSON dump containing the product details and the second field consists of a *Pagination* number which is used to track whether the products have been synced to ElasticSearch and the Bugatti servers(based on Apache Solr) of Zopper

In case of orders\_orderitem and orders\_orderitemsummary, the key is the same i.e *primarykey\_tablename* . The value of each key stores information about each order such as delivery status,etc.

1. The final component of the setup is the custom python client. This client reads data from the redis datastore. The redis datastore is used because AESOP does not propagate old values i.e values of fields before they were updated. Hence,redis stores the old values for these fields. The client compares these old values to the new one’s coming in from the Kafka topics and take decision based on some business logic. For eg, the redis cluster stores the lowest price in the current region for a given product. The logic implemented in the client would first checked the updated price it received with this existing value in the redis cluster, and if it is lower, it will update the value in the cluster and synchronize this price with the Elastic and Bugatti servers, which are used to serve the product prices on the consumer application.

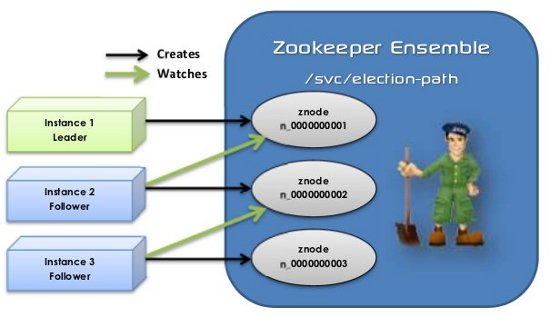
**3.6 Software Components**

**3.6.1 Apache Zookeeper**

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications. Each time they are implemented there is a lot of work that goes into fixing the bugs and race conditions that are inevitable. Because of the difficulty of implementing these kinds of services, applications initially usually skimp on them ,which make them brittle in the presence of change and difficult to manage. Even when done correctly, different implementations of these services lead to management complexity when the applications are deployed.

For reliable ZooKeeper service, we have deployed ZooKeeper in a cluster known as an *ensemble*. As long as a majority of the ensemble are up, the service will be available. Because Zookeeper requires a majority, we have used use an odd number of machines. For example, with four machines ZooKeeper can only handle the failure of a single machine; if two machines fail, the remaining two machines do not constitute a majority. However, with five machines ZooKeeper can handle the failure of two machines.

Zookeeper comes with an internal LeaderElection mechanism. Once the leader fails, it checks if the quorum condition is satisfied. All participants of the election process create an ephemeral-sequential node on the same election path. The node with the smallest sequence number is the leader. Each “follower” node listens to the node with the next lower sequence number to prevent a herding effect when the leader goes away. In effect this creates a linked list of nodes. When a node’s local leader dies it goes to election either find a smaller node or becoming the leader if it has the lowest sequence number.



**Figure 3.5**

**Zookeeper Cluster at Zopper**

**3.6.2 Apache Kafka**

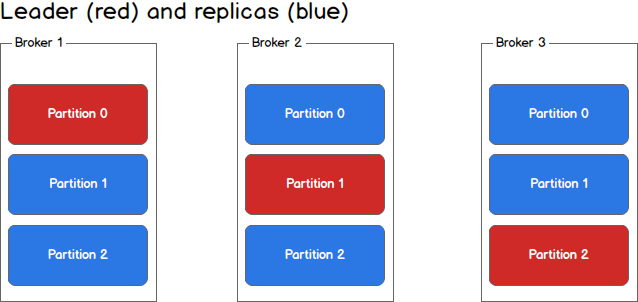
Kafka is one of those systems that is very simple to describe at a high level, but has an incredible depth of technical detai. In summary, *Kafka is a distributed publish-subscribe messaging system that is designed to be fast, scalable, and durable*.

Like many publish-subscribe messaging systems, Kafka maintains feeds of messages in topics. Producers write data to topics and consumers read from topics. Since Kafka is a distributed system, topics are partitioned and replicated across multiple nodes.

Messages are simply byte arrays and the developers can use them to store any object in any format – with String, JSON, and Avro the most common.Within Zopper, we have chosen to store it in JSON format over Avro because of the sheer simplicity of JSON. It is possible to attach a key to each message, in which case the producer guarantees that all messages with the same key will arrive to the same partition. When consuming from a topic, it is possible to configure a consumer group with multiple consumers. Each consumer in a consumer group will read messages from a unique subset of partitions in each topic they subscribe to, so each message is delivered to one consumer in the group, and all messages with the same key arrive at the same consumer.

Kafka treats each topic partition as a *log* (an ordered set of messages). Each message in a partition is assigned a unique offset. Kafka does not attempt to track which messages were read by each consumer and only retain unread messages; rather, Kafka retains all messages for a set amount of time, and consumers are responsible to track their location in each log. Consequently, Kafka can support a large number of consumers and retain large amounts of data with very little overhead.

Each broker holds a number of partitions and each of these partitions can be either a leader or a replica for a topic. All writes and reads to a topic go through the leader and the leader coordinates updating replicas with new data. If a leader fails, a replica takes over as the new leader.



**Figure 3.6**

**Kafka Cluster at Zopper**

At Zopper, we have implemented a three-server multi-broker cluster system. This ensures high availability and fault tolerance. With three brokers, the kafka system can handle two server crashes,ensuring that the performance of applications at Zopper is not affected.

**3.6.3 Apache Helix**

Apache Helix is a generic *cluster management* framework used for the automatic management of partitioned, replicated and distributed resources hosted on a cluster of nodes. Helix automates reassignment of resources in the face of node failure and recovery, cluster expansion, and reconfiguration.

**Cluster Management**

A distributed system typically runs on multiple nodes for the following reasons:

* scalability
* fault tolerance
* load balancing

Each node performs one or more of the primary functions of the cluster, such as storing and serving data, producing and consuming data streams, and so on. Once configured for your system, Helix acts as the global brain for the system. It is designed to make decisions that cannot be made in isolation. Examples of such decisions that require global knowledge and coordination:

* scheduling of maintenance tasks, such as backups, garbage collection, file consolidation, index rebuilds
* repartitioning of data or resources across the cluster
* informing dependent systems of changes so they can react appropriately to cluster changes
* throttling system tasks and changes

While it is possible to integrate these functions into the distributed system, it complicates the code. Helix has abstracted common cluster management tasks, enabling the system builder to model the desired behavior with a declarative state model, and let Helix manage the coordination. The result is less new code to write, and a robust, highly operable system

Helix has been used internally and modified to suit the needs of Zopper.

**3.6.4 ElasticSearch**

Elasticsearch is a search server based on Lucene. It provides a distributed, multitenant-capable full-text search engine with an HTTP web interface and schema-free JSON documents. Elasticsearch is developed in Java and is released as open source under the terms of the Apache License.

Elasticsearch can be used to search all kinds of documents. It provides scalable search, has near real-time search, and supports multitenancy. Elasticsearch is distributed, which means that indices can be divided into shards and each shard can have zero or more replicas. Each node hosts one or more shards, and acts as a coordinator to delegate operations to the correct shard(s). Rebalancing and routing are done automatically.

Elasticsearch uses Lucene and tries to make all its features available through the JSON and Java API. It supports facetting and percolating which can be useful for notifying if new documents match for registered queries.

*ElasticSearch at Zopper*

**3.6.4 Bugatti**

Bugatti in an internal engine developed on top of Apache Solr. Solr is highly reliable, scalable and fault tolerant, providing distributed indexing, replication and load-balanced querying, automated failover and recovery, centralized configuration and more. Solr powers the search and navigation features of many of the world's largest internet sites.

As Zoper is a hyperlocal-ecommerce company, we need to keep track of the lowest price of products in a region at any given time. This price is served to the consumer application via Bugatti.

**3.6.5 Redis Cluster**

Redis is an open source (BSD licensed), in-memory **data structure store**, used as database, cache and message broker. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps,hyperloglogs and geospatial indexes with radius queries. Redis has built-in replication, Lua scripting, LRU eviction,transactions and different levels of on-disk persistence.

It supports *atomic operations* on these types, like appending to a string; incrementing the value in a hash; pushing an element to a list; computing set intersection, union and difference; or getting the member with highest ranking in a sorted set.

In order to achieve its outstanding performance, Redis works with an *in-memory dataset*. Depending on the use case, we can persist it either by dumping the dataset to disk every once in awhile, or by appending each command to a log. Persistence can be optionally disabled, if you just need a feature-rich, networked, in-memory cache.

Redis also supports trivial-to-setup master-slave asynchronous replication, with very fast non-blocking first synchronization, auto-reconnection with partial resynchronization on net split.

*Need at Zopper*

The main reason why Redis is needed as Zopper is because AESOP does not propagate old values with the records that it sends to the consumers. Hence, there exists no way to compare values and take some business decisions. To solve this problem, we run the AESOP system on the older logs(say logs from January 2016) and fill up the redis-cache with these values. Then these values are compared with the incoming new values to perform certain tasks and take critical business decisions.

*Redis Architecture at Zopper*

redis.png

**Figure 3.7**

**Redis Master-Slave with twemproxy cluster**

*Master-Slave Replication*

Redis replication is a very simple to use and configure master-slave replication that allows slave Redis servers to be exact copies of master servers. Redis uses *asynchronous replication*.

Slaves are able to accept connections from other slaves. Aside from connecting a number of slaves to the same master, slaves can also be connected to other slaves in a graph-like structure.

Redis replication is non-blocking on the master side. This means that the master will continue to handle queries when one or more slaves perform the initial synchronization.

Replication is also non-blocking on the slave side. While the slave is performing the initial synchronization, it can handle queries using the old version of the dataset. However, after the initial sync, the old dataset must be deleted and the new one must be loaded. The slave will block incoming connections during this brief window.

Once a slave is set up, upon connection it sends a SYNC command. It doesn't matter if it's the first time it has connected or if it's a reconnection.

The master then starts background saving, and starts to buffer all new commands received that will modify the dataset. When the background saving is complete, the master transfers the database file to the slave, which saves it on disk, and then loads it into memory. The master will then send to the slave all buffered commands. This is done as a stream of commands and is in the same format of the Redis protocol itself.

The slaves of all masters at Zopper have been set to a read-only mode. This ensures that there is no way some data can exist on the slave and not on the master. All the read commands are split evenly between the master and slave, but the write commands can be performed only through the master.

*Redis Sentinel*

Redis Sentinel provides high availability for Redis. In practical terms this means that using Sentinel you can create a Redis deployment that resists without human intervention to certain kind of failures.

Sentinel provides:-

* *Monitoring* : Sentinel constantly checks if the master and slave instances are working as expected.
* *Notification* : Sentinel can notify the system administrator, another computer programs, via an API, that something is wrong with one of the monitored Redis instances.
* *Automatic**failover* : If a master is not working as expected, Sentinel can start a failover process where a slave is promoted to master, the other additional slaves are reconfigured to use the new master, and the applications using the Redis server informed about the new address to use when connecting.
* *Configuration provider***:** Sentinel acts as a source of authority for clients service discovery: clients connect to Sentinels in order to ask for the address of the current Redis master responsible for a given service. If a failover occurs, Sentinels will report the new address.

In Zopper, each master-slave server has a sentinel server running. The sentinel servers discover other sentinels running on the same network and start communicating with each other. Once a master server is down, sentinel broadcasts a *SDOWN* message. It then pings the master for the time specified in the redis configuration. It sentinel still can’t connect with the master, it sends a *ODOWN* message and then the slave is promoted to the role of the master.

*TWEMPROXY*

Twemproxy is a fast, light-weight proxy for redis. As Redis Cluster is a unstable feature, we needed to cluster the multiple redis-servers running at Zopper to make it scalable to meet the fast growing demands. To solve this issue,we have used Twemproxy.

Twemproxy has the following functionalities:-

* It works as a proxy between clients and many Redis instances.
* It is able to automatically shard data among the configured Redis instances.
* It supports consistent hashing with different strategies and hashing functions.

The failover scenario is handled by the sentinel servers. Hence, by using twemproxy, we have built a highly scalable,fault-tolerant and highly available redis cluster. The redis cluster is an independent module, hence it can be used throughout the organization.

**3.7 Changes at Zopper**

Following are the changes to AESOP that have been implemented at Zopper:-

1. AESOP relay was designed to support reading binlogs from only one database at a time. We have provided support for a single relay server to read binlogs from multiple databases at a time.
2. The Kafka-Client-Consumer was designed to read update events from a single database source. We have provided support to consume update events from multiple sources.
3. Provided scripts in BASH and python to monitor the status of AESOP relay and client consumers.
4. Scripts to monitor the status of each cluster and notify DevOps team incase any component stops functioning.
5. Developed the STOP\_EVENT processor for Google Open Replicator which is used internally in AESOP to read events from mysql binlogs. After the addition of this processor, the relay can be started from any binlog index number, ensuring that there is no loss of events incase the relay stops working.
6. Integrated scripts to send out mails and alerts to concerned people incase the MySQL service goes down.
7. Developed custom *TransportClient* in ElasticSearch to support current version 2.2 in AESOP. Through this client, business logic specific to zopper is executed.
8. Upgraded the AESOP build to compile with the latest Java jdk-1.8. This ensures that there is up-to-date technology running at Zopper.
9. Automated complete deployment of AESOP to any system through Ansible.

**3.8 Automating Deployment through Ansible**

Ansible is a configuration management tool that is used to provision servers and automate deployment of applications into production. It is needed because handcrafted servers:-

* Are very hard to maintain.
* They are cost and time intensive.
* Setup is not very easily reproducible.
* They are very buggy and error-prone.

Ansible has the following advantages:-

* There are no master or agent servers.
* It is coded in YAML,making it easy to learn and understand.
* It primarily uses ‘SSH’ for all connections. Hence, all Unix commands will work as it is on ansible.

At Zopper, Ansible is a critical component of any application deployment. We have written Ansible code to automatically:-

* Setup all the required packages for AESOP.
* Clone and configure the AESOP relay and client consumer on each server.
* Install and configure Zookeeper servers and make them function in the Leader-Follower mode(ensemble)
* Install and configure Kafka multi-broker system which internally uses the Zookeeper ensemble.
* Install and configure Redis servers, set master-slave replication, configure sentinel servers for each Redis instance, and finally configure Twemproxy to make the redis instances function like a single cluster.

### **CHAPTER - SOURCE FROM ZOPPER**

**4.1 Introduction**

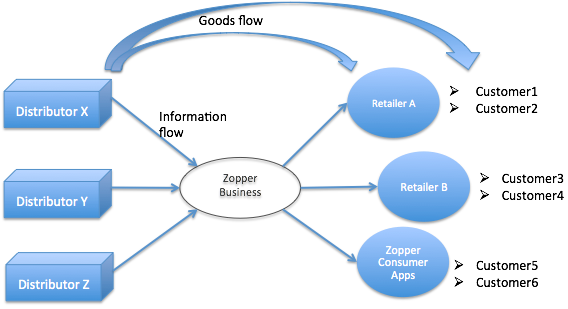
Retailers typically are able to store and carry a very small set of products because of constraints such as capital, warehouse space etc.. Consumers on the other hand are becoming more and more choosy about what products they are looking for, especially with easy availability of product information, reviews etc. on internet.

This is especially true for retailers in electronics, home appliances, mobiles categories. Retailers also struggle with large volume of orders for the same product as they cannot stock more than a certain number of units at any given point in time.

To solve this, Zopper wants to provide a service to retailers where in they can query (in real or near real time) and source a specific product from a wide selection of products which are stocked by a set of distributors associated with Zopper.

**4.2 Process Flow**

The system that has been developed at Zopper functions like below:-



**Figure 4.1**

**System Flow Diagram**

When a customer walks-in to a retailer’s shop looking for a specific item, the retailer will query the Zopper Business, to check the availability and price of the item.

If the item is available and at a price (that includes zopper commission, delivery charges and taxes) and delivery SLA which is acceptable to retailer, the retailer can then place an order for the item to be delivered to his designated address (which could be customer’s address).

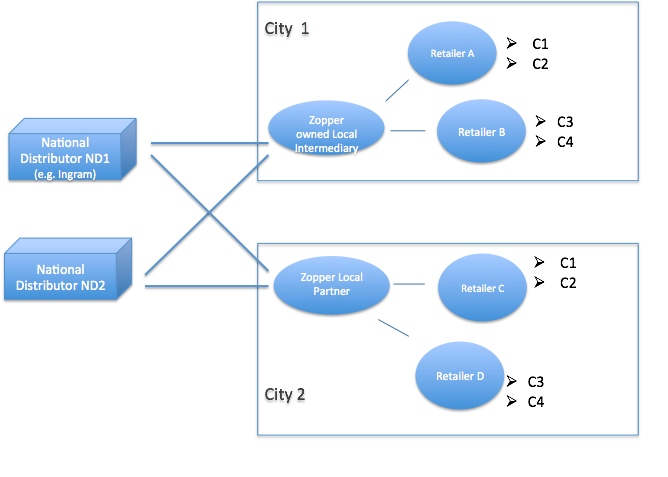
The payment has to be made in advance for these orders to Zopper business. No returns and cancellations are allowed on these orders.

Delivery to the given address will be arranged by Zopper using either in-house teams or one of it’s delivery partners.

Billing will be done from the distributor to the retailer directly, and Zopper will charge it’s service commission plus service tax on the commission.

Alternatively, for sourcing from certain bigger distributors, it might be the case that the distributor is not willing to bill directly to retailer. In this case, the distributor will bill first to a Zopper intermediary or chosen partner (this could be a different entity in each region). Then the said intermediary will create an invoice on the name of retailer.

The system can look as shown below in the second case. However, the flow of goods will still happen directly from distributor to an address specified by the retailer.



**Figure 4.2**

**Catalog Extension Flow Diagram**

**4.3 Use Cases**

This part describes the use cases for which support has been provided in Catalog Extension:-

1. *Stock Synchronization:* 
   1. Provided a catalog system, functioning to the one that already exists at zopper.
   2. Integration with chosen local distributors - we get real time (or near real time) visibility of stock availability from them.
   3. Integration with Ingram (national level distributor) – to get visibility of their stocks on our platform.
2. *Retailer flows:*
3. System for Retailers to query the stock availability,
4. Order a particular item from stock.
5. Payment systems to charge retailer(using retailer wallet) for placing the order & using the service.
6. *Billing* - For both the retailer and the distributor,, using the appropriate intermediary (e.g. Waterworks)
7. *Delivery* - Setting up logistic partner(s) to pick items from the distributor and deliver to retailer. This is including a delivery management process\system
8. *Payment* to Distributors - Systems for making payouts made to the distributors.
9. *Grievance handling* :- This will be to address any concerns or complaints the retailers have against the distributors and vice-versa.

## **4.3.1. Stock Synchronization**

Zopper Consumer Catalog has been used as the master catalog for Zopper business as well. It is not a copy of Zopper Consumer Catalog, but the same catalog has been extended to support requirements of Zopper business.

For a given distributor onboarded on Zopper business, we do the following: -

1. Store the mapping of the SKU(Product ID on distributor side) IDs of distributor to PIDs(Product IDs on Zopper platform) in Zopper catalog. New products as encountered\discovered in product feed\API pushes from distributors are sent to Zopper catalog team for updating the mapping)
2. We store a list of products (and quantity/commitment if available depending on type of data available from distributor) along with prices (this does not include Zopper commissions and taxes) which are available from distributor.
3. Update the list described in ‘b.’ at a predefined frequency or event depending on the agreement\integration with distributor. This is acheived using Django Celery, which supports asynchronous tasks.

## **4.3.2. Retailer Workflows**

The retailer should be able to perform the following actions from the seller dashboard:-

1. *Search*

Search for availability of a product on Zopper business with part or full productname and/or Model No. as query parameter.

Further retailer should be able to filter search results by one or more of following filters:-

* Category (selected from a dropdown)
* Brand (brand filter can only be selected after selecting category filter)

The filter can be applied before or after a search. The search will rerun with filters applied if filters are applied after searching first.

On clicking search, list of top 20 products (Product name, image) matching the search query should be displayed. Each tuple is clickable, clicking which opens up product’s price and availability details inline on the page.

*Search Suggestions:*

After typing 3 or more characters in the search query in the given search box, retailer should be able to see top (at max 20) matching product names results as suggestions below my search box. On clicking any of these price & availability information of the selected product will be displayed.

### *b. Price\Availability details of a product*

When retailer clicks to view the price\availability of a product, retailer will be prompted to select the nearest delivery locality corresponding to a delivery address. The dropdown will contain same localities as are currently displayed on Zopper consumer side corresponding to the region of the retailer.

Locality once selected will be cached for the given search session. Retailer will be asked for a locality only if performs a new search.

If product is available from any of the distributors onboarded for the given region, Product will be shown as available.

Then a list of final prices applicable for retailer are shown to retailer, along with estimated delivery time (to be calculated based on distance between distributor warehouse, and locality entered by Seller, and our agreements with distributor, and delivery partners for said distance)

Price shown should include delivery charges, zopper commissions and any taxes (VAT in case of intermediary billing, service tax in case of direct billing).

If there are more than one distributors with Product available, the least price and corresponding delivery time will be shown on top.

All other options where estimated delivery time is less than the least price option will also be shown below it, with an ‘Order now’ button against each option.

### *c. Checkout process*

When the retailer clicks on order now, the item from the corresponding distributor is blocked for a period of 5 minutes (meaning the inventory of the item is reduced by 1). The retailer has to finish rest of his order flow in next 5 minutes, otherwise he will have to start afresh.

*Delivery address*

Retailer is then prompted for a delivery address. Retailer can select either his shop’s registered address with Zopper or enter a new address.

*Payment*

Then retailer makes payment using any of the online methods supported (CC,DC, Netbanking, Retailer Wallet). In case Retailer wallet does not have sufficient balance to cover the cost of item, rest of the payment is collected from any of the other methods. If retailer selects wallet, the system automatically calculates the maximum amount that can be taken from wallet, and prompts him to choose to pay the rest of the amount from any of the existing methods.

No flexibility has to be given to adjust the share of wallet and other payment method. Amount is deducted from retailer wallet, as soon as the control leaves zopper systems and goes to payment gateways for collecting rest of the amount. If the transaction fails on the payment gateway for rest of the amount, the wallet transaction is also reverted.

On successful payment, the retailer is shown a success message along with the details of his order i.e.

1. A unique order ID (This is unique across retailers, across time)
2. Product name
3. The net money charged from him
4. Estimated delivery time
5. Delivery address
6. An option to view invoice and commission invoice (explained below in billing)

### *d. Order Tracking*

All orders placed by sellers in the past will be available in a panel called “View past purchase orders” with above mentioned details for each order, along with order placement date (and sorted by it in reverse chronological order), and an order status.

**4.3.3. Billing**

There are two billing methods to be supported.

*Case 1*: In case Billing will be done from the distributor to the retailer directly, clicking on Invoice link will just show the retailer a message that the actual invoice will be provided with the goods or uploaded in the system and will be available at a later date.

On clicking “commission invoice”, Commission invoice of Zopper with service tax will be displayed along with an option to print.

Case2: If the distributor is of type which doesn’t bill directly to retailer. In this case, an invoice will be generated on behalf of Zopper intermediary for the retailer’s region and displayed to the retailer with an option to print it.

On clicking “commission invoice”, “Commission invoice is not applicable” message will be displayed.

## **4.3.4. Delivery**

Orders placed through above system by retailers should be passed on to the corresponding distributor depending on the integrations in place for each retailer, so that distributor can make order available for delivery along with invoice.

Simultaneously, order details should also be passed to the Zopper team managing “Delivery scheduling” for retailer purchase orders.

Third-Party-Logistic(3PL) partner or delivery person from zopper will receive the order details, and make the pickup and delivery as applicable. Delivery partner\person will need to update the system to inform a delivery has been done. Order status will change to “Delivered” and will reflect against the order status on retailer’s panel.

## **4.3.5. Payment to distributors**

A report is to be generated for each distributor based on our agreement containing details of which all orders Zopper needs to pay them for since last payment. This needs to be sent to finance team directly to remit the amounts.

## **4.4 System Requirement Specification**

**4.4.1 Capability**

The system that is being built should be capable of handling the entire product inventory of all distributors, which can exceed one million units at any given point in time. The system should be able to accommodate new national-level distributors and also list and update their products in near real time complexity. It should also be scalable to meet the ever-growing needs of the Zopper business.

**4.4.2 Reliability**

The system that is being built will contribute a major-portion of Zopper’s business. Hence, it should be extremely reliable. The system should have very minimal or no downtime. There should also be software developed to continuously monitor the status of the servers and the databases incase any server/application stops responding and/or crashes.

**4.4.3 Sustainability**

As this system will be a major-contributor to increasing the number of orders for a retailer, the system should be able to sustain the large number of incoming orders for stores in a particular region. It should also be able to respond to these ordes and synchronize everything with the databases present at Zopper to prevent any ambiguity.

**4.4.4 Security**

The system must be a fail-proof and tightly secured system as distributors can misuse it to give false information about the products they have and the exact stock of the products stored in their warehouses. Also, distributors should only be able to lend their inventory to stores that have been allocated to them. Measures must be taken to ensure that every step/transaction goes through the Zopper Business entity, so that it can be properly recorded for future purposes.

**4.4.5 Components**

Following were the components used to implement the system for Catalog Extension:-

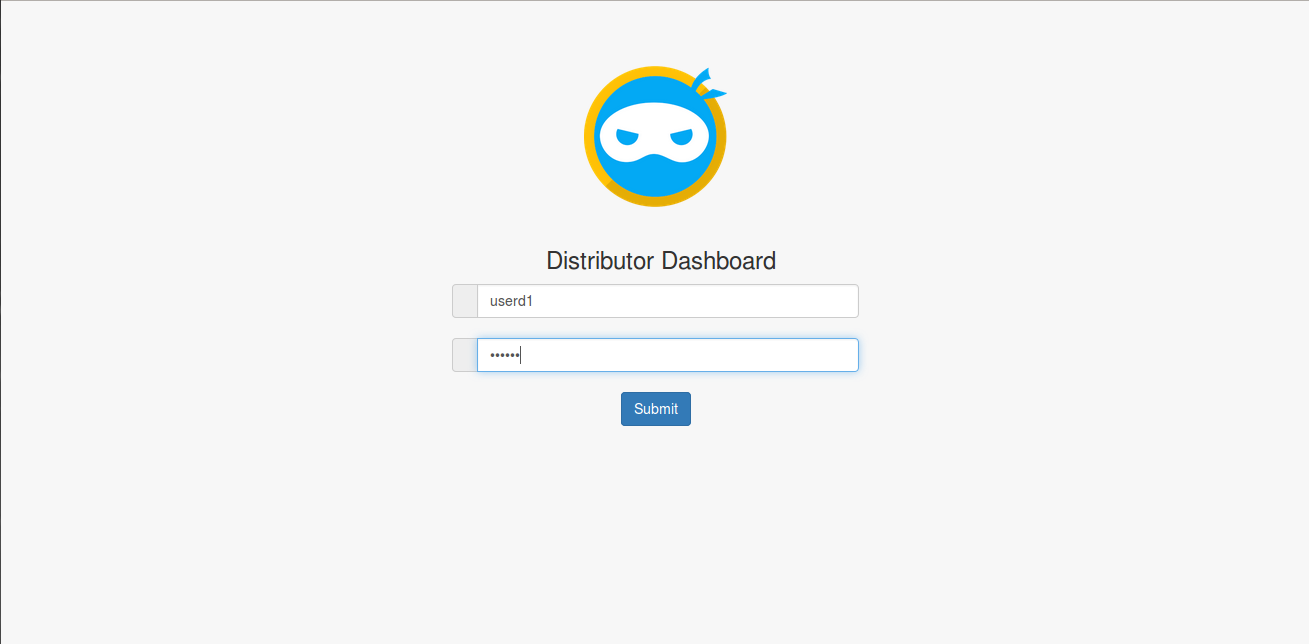
* Five AWS instances running Ubuntu 14.04 with 8GB RAM and 400GB HDD. These instances were used to web application servers,database servers,payment gateway servers,etc.
* Django Framework
* DjangoREST framework which was used to create RESTful state API’s. These API’s were then consumed by the dashboard and the Android Application to retrieve data about the products and other data such as stores attached to a distributor,etc.
* DjangoCelery(DjCelery) along with RabbitMQ as the broker was used to carry out asynchronous tasks such as sending emails about the orders to retailers, updating the inventory of the distributors,etc.
* ElasticSearch and Bugatti(Based on Apache Solr) was used to deliver product prices in the android application and also to support the autocomplete feature.
* Ansible was used to deploy the entire project on staging for testing purposes and then on production.

## **4.5 Design and Implementation**

The steps to onboard a new national distributor onto Zopper as part of Catalog Extension are as follows:-

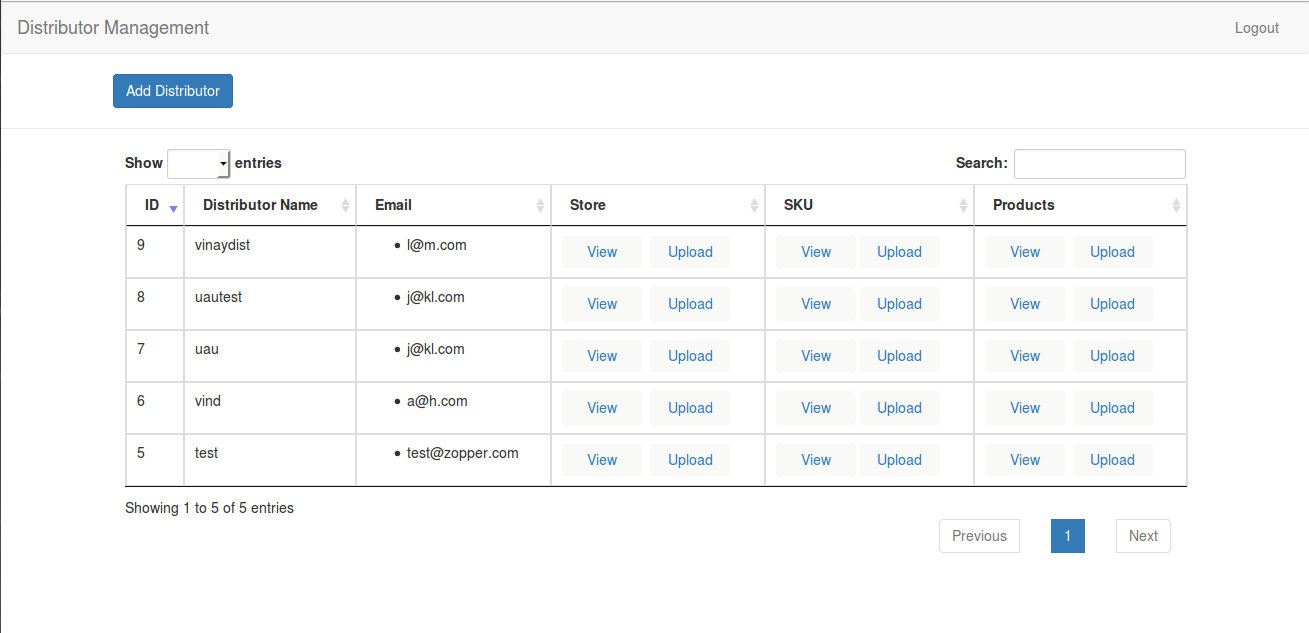
1. Login Credentials are generated by the Zopper team for the people who will upload the details for the distributor. This part generates a token for the current login session. Hence, it will block access to people who are not authorized.
2. Once the login is successful, the user is directed to the distributor screen.
3. Here, the user will click on the ‘Add Distributor’ button to add a new distributor. After the entering the required details, the distributor is registered on the platform.
4. Once the registration is complete, on the home screen the user will be able to see the distributor name. Now the user has to upload the stores, SKU mappings and the Product Mapping in the same order.
5. All the data about the distributors are uploaded in an Excel Sheet format(.xlsx). The reason Zopper has chosen this format is because it is easy for distributors to upload the details of the products in their warehouse in a single go via excel sheets than using API calls for each product.
6. The Excel sheet contains details about the store such as name,address,latitude and longitude,etc. Once this is uploaded, the Excel sheet is parsed each record is synced into the database asynchronously using Django Celery tasks. Mails are used to notify concerned stakeholders about the result of the above task.
7. SKU and Zopper Product Mapping(ZPID) is a crucial step for implementing distributors as they will have a different product ID compared to the one stored in the Zopper database. Zopper generates unique ZPIDs for each SKU. The details for this are stored in an Excel sheet and uploaded via the same dashboard for each distributor.
8. The last step is the upload products which will be accessible to each store. This is done through an Excel sheet that contains detials about the product such as product ID,product name,specifications,store ID to which this product will be accessible etc. Once the upload is successful, the excel sheet is parsed using a custom Python parser. The records are then sent into Celery, which performs asynchronous tasks. For Catalog Extension, a new index was created in ElasticSearch Called distributor\_price. All the details about products that were available from the distributor were synced to this index. The Celery tasks validates all the business logic implemented for the products. Once they are validated, the products are synced to ElasticSearch and Bugatti Engines.

## **4.6 Screenshots**



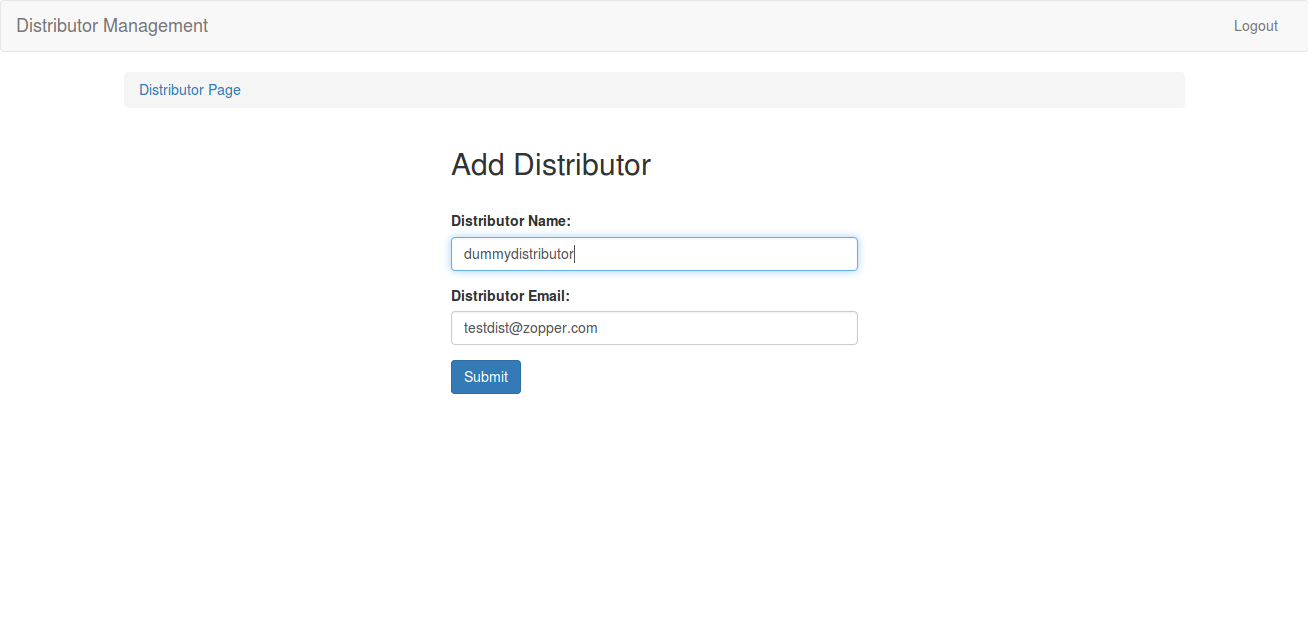
**Figure 4.3**

**Login Page for Merchant Dashboard**



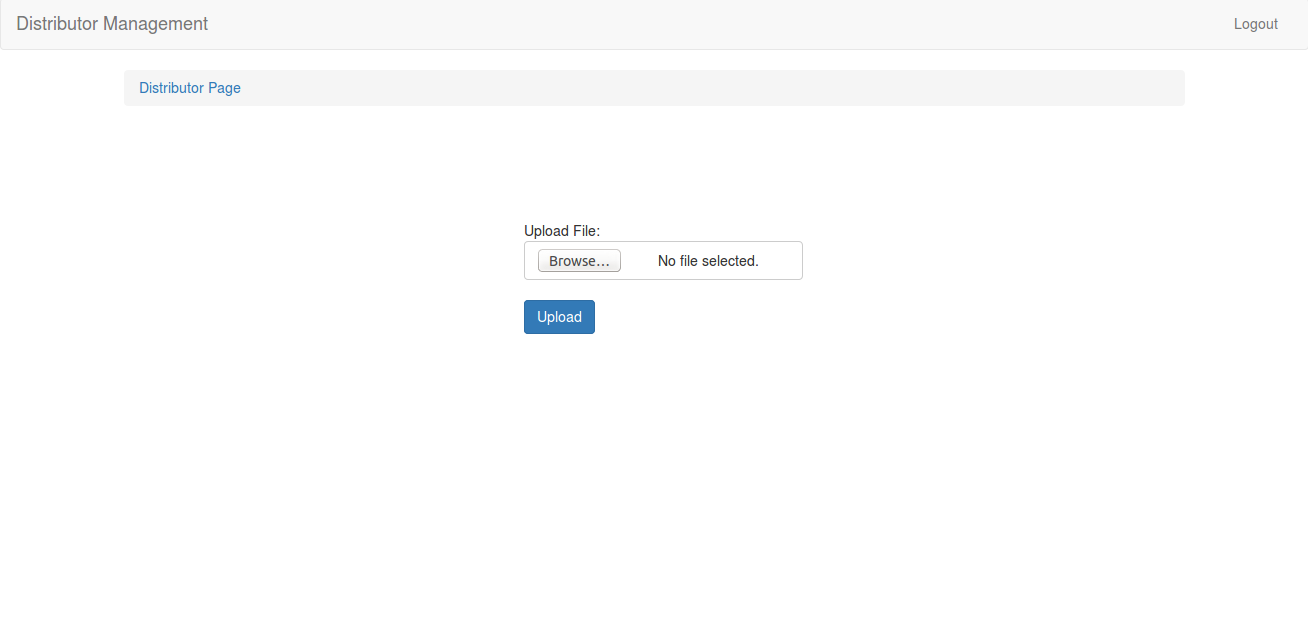
**Figure 4.4**

**Home page**



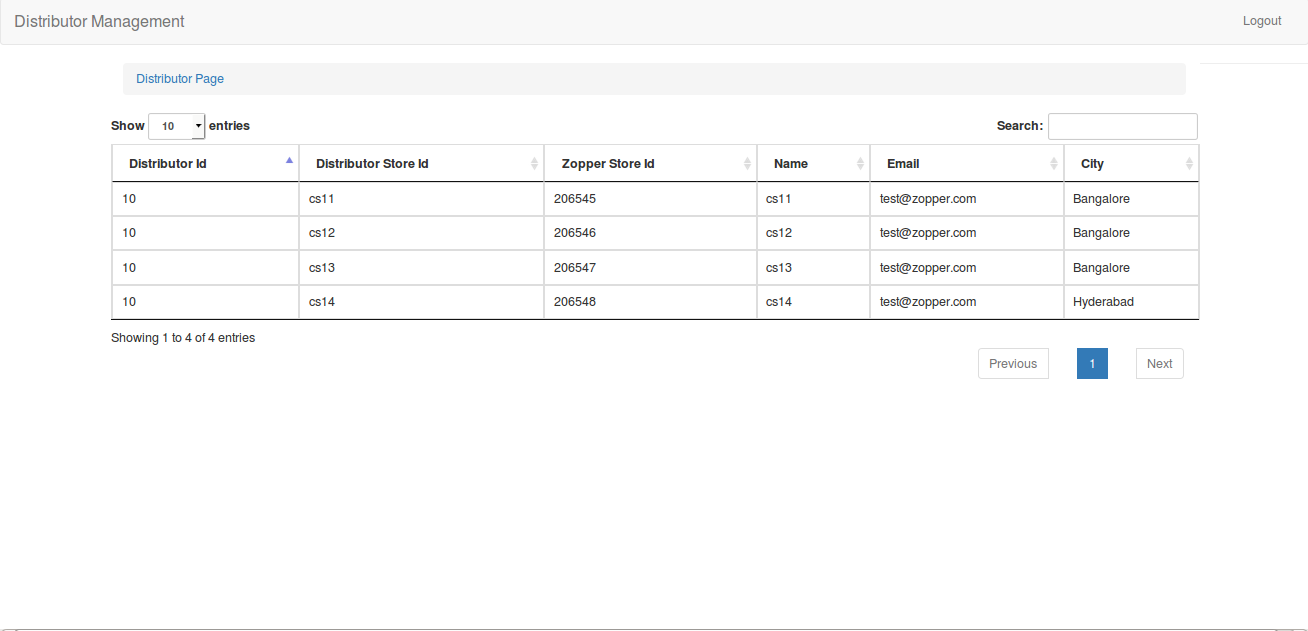
**Figure 4.5**

**Distributor Add Page**



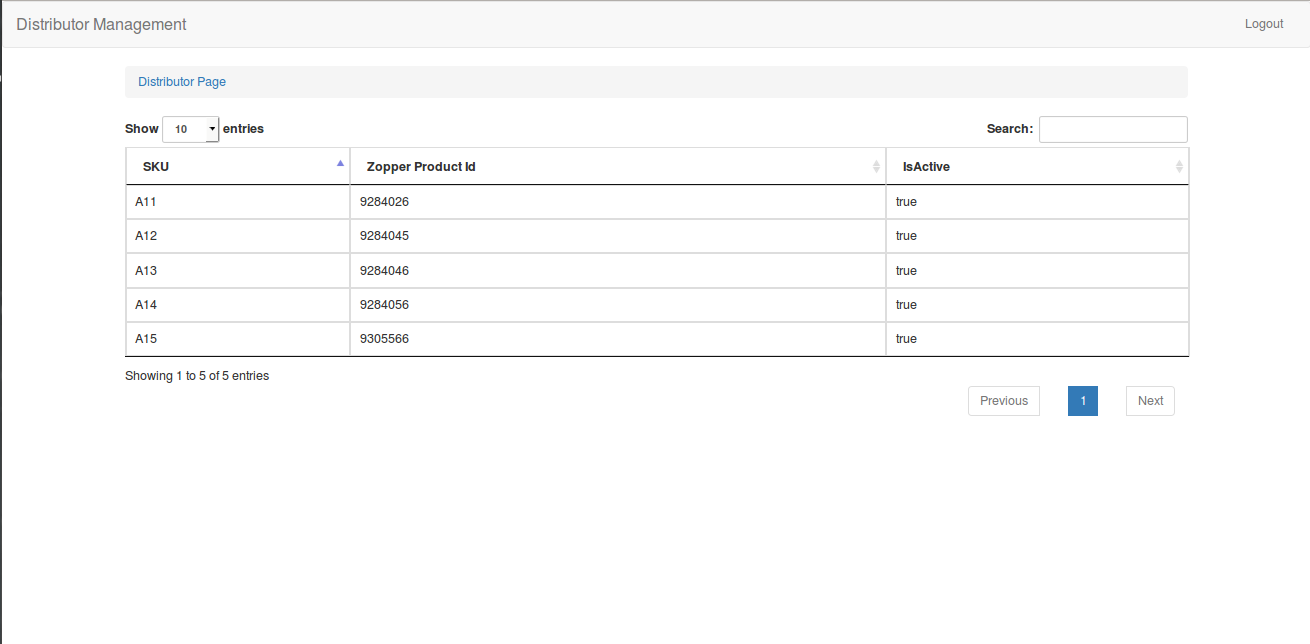
**Figure 4.6**

**Upload Page**



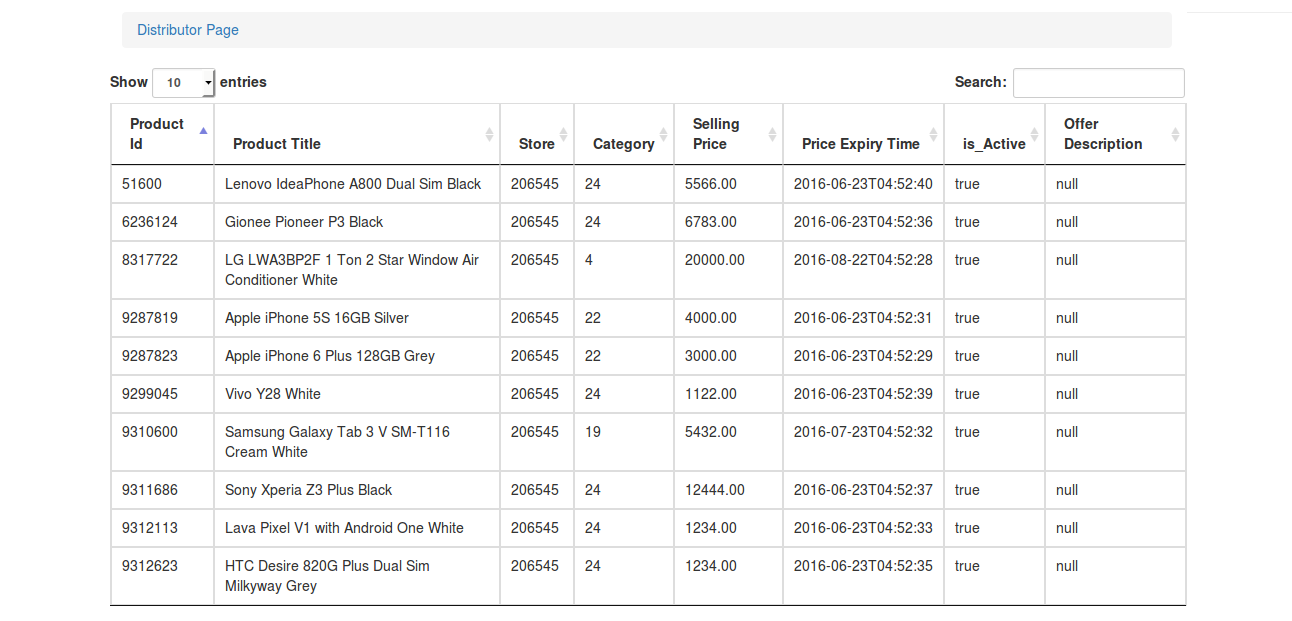
**Figure 4.7**

**Store List Page**



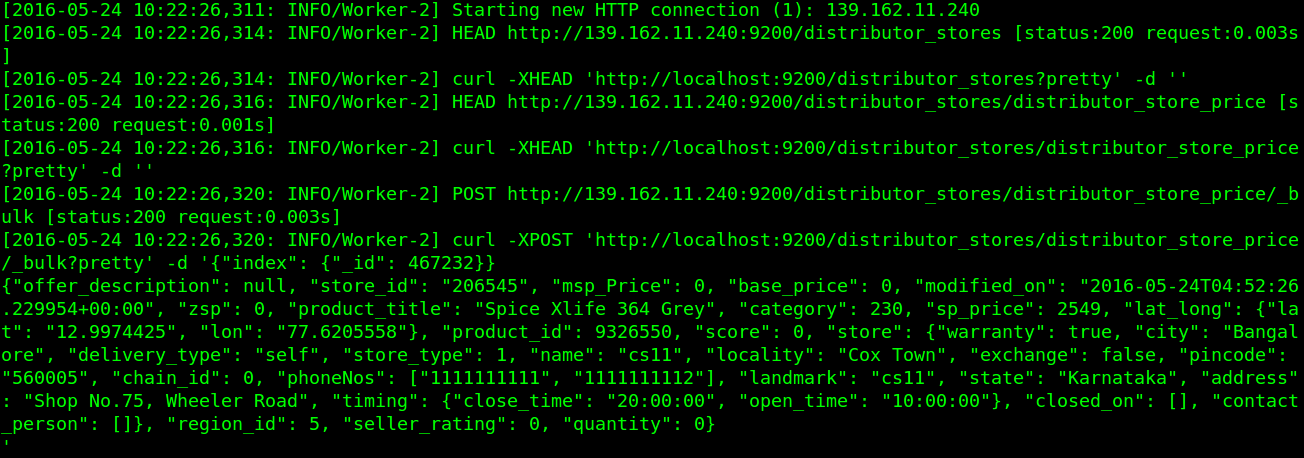
**Figure 4.8**

**SKU List Page**



**Figure 4.9**

**Product List Page**



**Figure 4.10**

**Synchronizing products to ElasticSearch**

**4.7 Conclusion**

Source from Zopper i.e Catalog Extension was a major project for Zopper. It was similar to setting up a miniature Zopper, hence it has touched and impacted every subsystem that exists within Zopper.

From a national perspective, this project has helped increase Zopper’s prescence throughout the country. Through national distributors, we can ensure that the consumer demands of fast delivery and good quality products are satisfied. It also helps increasing the number of orders coming in for any retailer, thereby helping them increase their visibility in the region.

In the future, Zopper aims at acquiring more national distributors which will further increase the supply of products to meet the never ending demand of consumers. The system will be scaled out to accommodate every store in India by the end of 2017.

**CHAPTER: DESKTOP WEBSITE FOR ZOPPER**

**5.1 Introduction**

Zopper started out as a site for reviewing products across the country. It was soon converted into a hyperlocal e-commerce platform which brings local retailers onto it’s platform and helps them trade goods and services.

Due to the rate at which Zopper is growing, a web-platform was needed for the website to address the needs of consumers.

Hence, a solution called ZINAFAS was proposed. This has since gone through many iterations and has been the building block for building the zopper website.

**5.2 Features**

The website that we have proposed has the following unique features which will differentiate it from the existing e-commerce website:-

1. *Single Page Shopping*

Users will be able to see product Specifications, Local Trusted stores, Service Centers, Reviews, Cart with Shipping and payment details in a single page. This will ensure that the users do not get distracted with unnecessary content. This will also help to streamline the process.

2. *Friends Reviews*

User will have a facility to get the reviews from his buddies, friends and Family members before buying the product. This is an important feature as feedback from friends/family plays a major role in influencing a user’s decision to buy a product or not. It will also ensure that the users are actively participating in the discussion.

This process will also bring the friends of a user to the platform of Zopper, thereby increasing the traffic to the website and contributing to the growth of Zopper.

3. *Trusted Stores & Service Centers*

Trust of Local Stores and Knowledge of Nearest Service Centers is a unique feature. There is a dynamic map that has been placed on the website. This map lists the stores and service centres present for a particular product. This will have a huge impact on the users decision to buy the product as one always looks for a product which has quality service centres so that servicing the product will be convenient. We have also listed stores which come with Zopper Assure which is an add-on warranty consumers can buy along with their products for a very low price. This warranty will give users confidence that the product is worth buying and it will be a stable product.

4. *Convenience of Online Shopping*

Easy to shop, No Multipage Clutter, Select Products, Get Reviewed and Check Out with Products with in Seconds.

**5.3 Problems Addressed**

Existing e-commerce website face the following problems:-

1. *Multipage clutter*:- As the product details are spread over multiple pages, the consumer get lost in a clutter of webpages. This clutter distracts him and hence he loses his interest in the product, thereby causing the website to lose valuable consumers.
2. *No trusted Suggestions or Reviews:-* Online websites are flooded with fake reviews and suggestions from third parties that just want to promote their products on the website.
3. *No proper knowledge of Seller and Service Centers:-* Consumers do not have adequate knowledge about the service centres for each product. There is lack of trust between the consumer and the retailer due to unknown reputation of the retailer and the website on which the product is being sold.
4. *Concerns about quality and warranty of product:-*

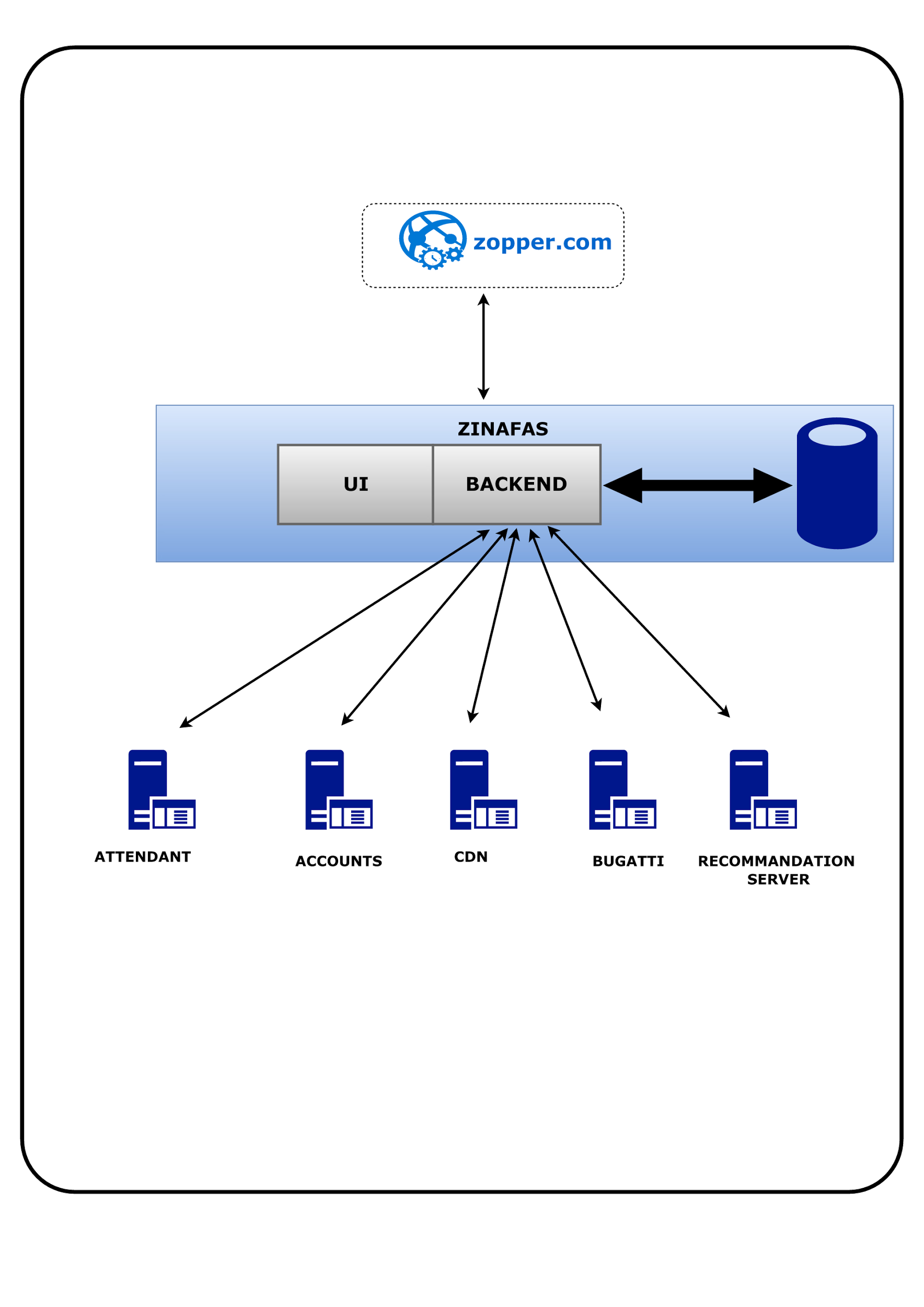
The consumers do not trust the quality and the warranty provided for the products as there is no organization to guarantee any of the above products.

**5.4 Solutions Proposed**

The solutions that have been proposed are as follows:-

1. *Easy single page shopping:-* Zopper has implemented a single-page--checkout system in which the process from the product description to the final checkout from the cart is restricted to a single page, thereby streamlining the process of buying the product.
2. *Reviews from Experts, Friends and Family:-* The review system is unique to Zopper as it helps user take real-time reviews from their friends and family about a product. This review largely influences their decision to buy the product. Also having reviews from trusted expert reviews provides valuable insight to users.
3. *Trusted Local Stores Information of Service Centers:-* Providing verified information about service centres ensures that the users feel comfortable buying the product as they as confident about the quality of service provided through Zopper Service Centres.
4. *Zopper Warranty:-* This is a new initiative by zopper which guarantees protection against manufacturing damages for all products brought on the zopper platform.

**5.5 Technical Architecture**



**Figure 5.1**

**Technical Architecture**

The components of the technical architecture are as follows:-

1. *Attendant:-* This is an internal system developed at Zopper using Django and DjangoREST framework with Celery for asynchronous tasks. This system is used to serve the API for all tasks related to stores and product such as onboarding a new store on the Zopper platform, updating the inventory of each store,etc.
2. *Accounts:-* This is an internal system at Zopper which is used to keep track of all financial and authentication needs of Zopper. The details of dispatched orders, along with order details and the consumer details such as consumer name,banking details,etc. As stored securely in this system. It is also used to authenticate users and set permissions for each user.
3. *CDN:-* A Content Delivery Network(CDN) has been implemented in the system to supply the applications with static data such as images of the products,etc.
4. *Bugatti:-* Bugatti is an internal system in Zopper that has been built on Apache Solr. This system is use to fetch and serve the prices and details of each product available on the Zopper platform. It also caches some images of the product that are rendered on the website.
5. *Recommendation Server:-* The recommendation system has been built using minimal machine learning algorithms on the specifications of the products. This system is used to give valid and attractive recommendations to users based on their location and the product that they are currently interested in.

The above systems are used to feed data to the backend of the website which has been developed using Django and DjangoREST framework. This backend application in turn abstracts and provides data which is used by the UI to display the relevant information on the website. The UI layer will continously query the backend layer for any data/information it needs either for the products or for information about the user.

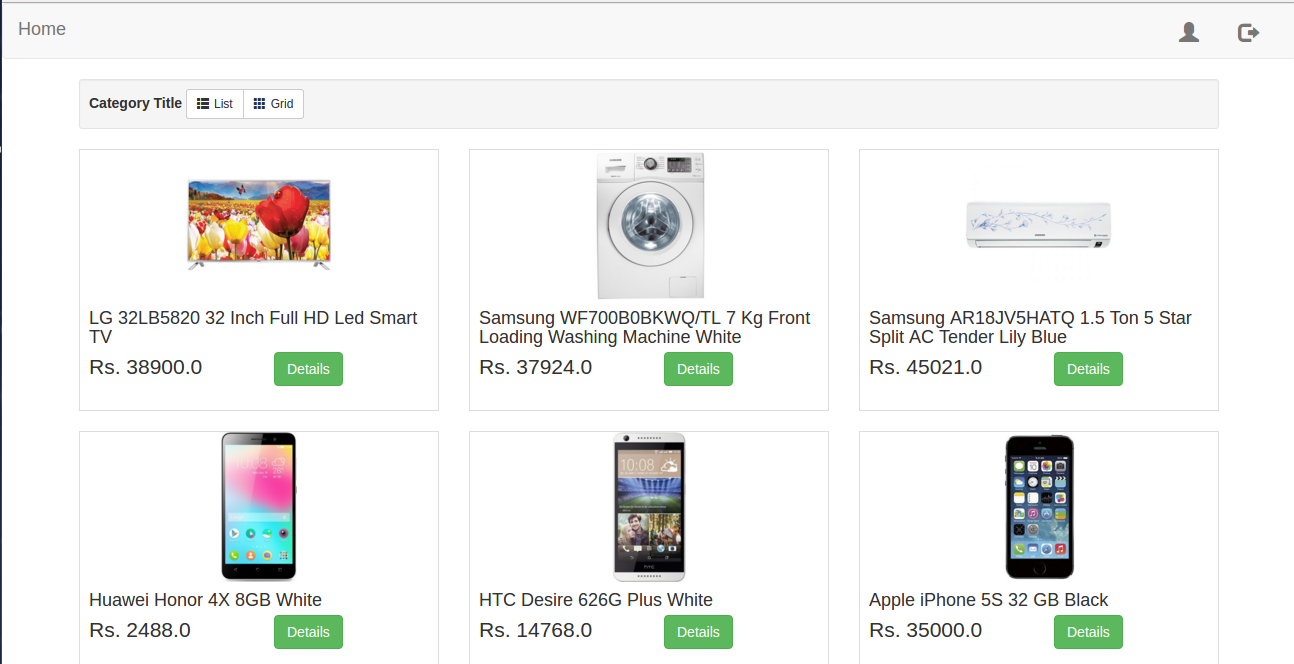
**5.6 Process Flow Diagram**



**Figure 5.2**

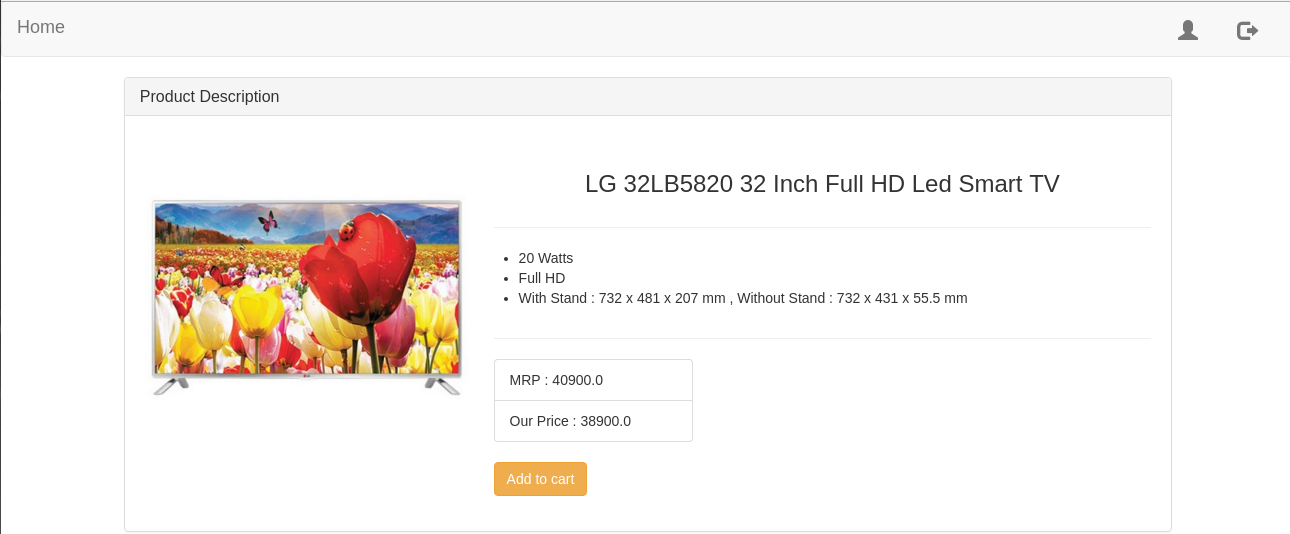
**Process Flow Diagram**

**5.7 Screenshots**



**Figure 5.3**

**Home Screen**

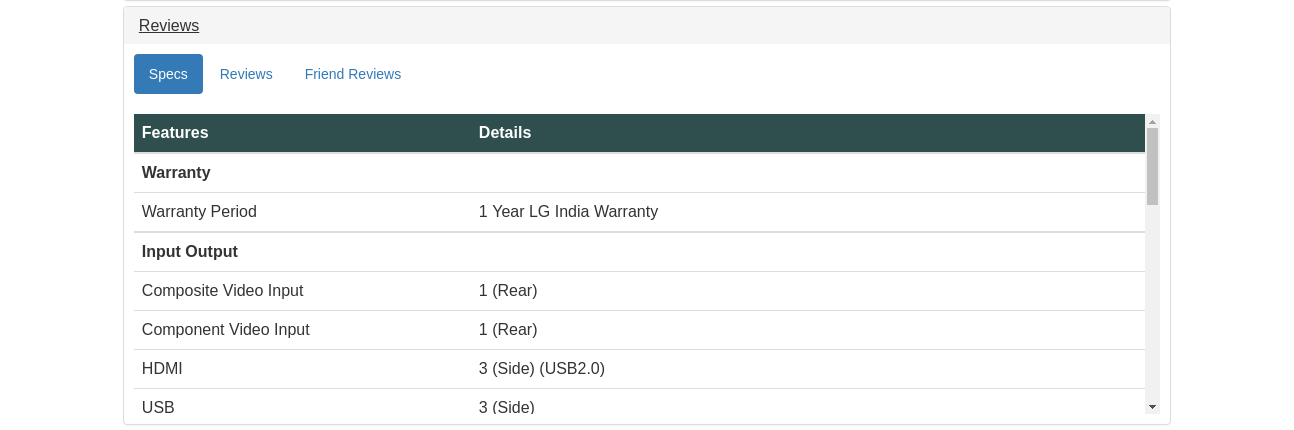


**Figure 5.4**

**Product Screen**

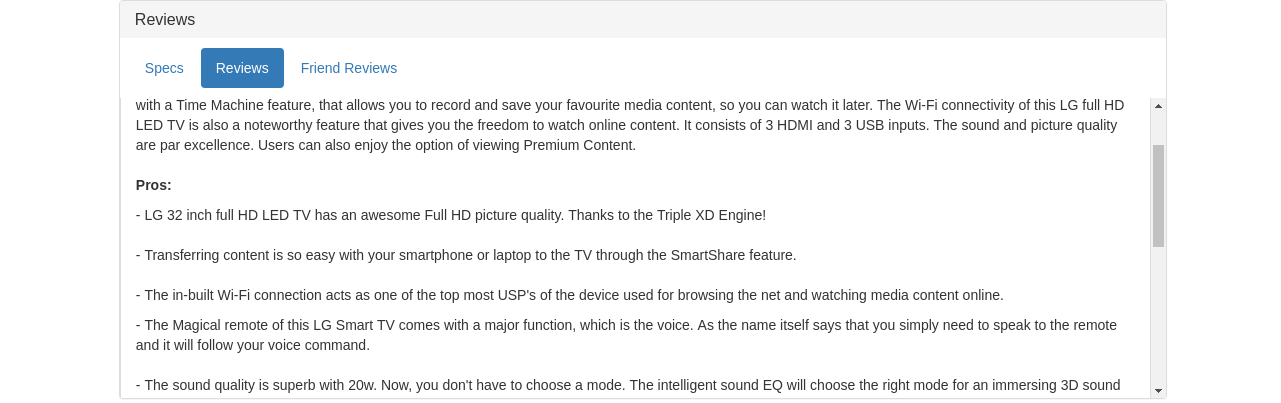
**Figure 5.5**

**Dynamic map**



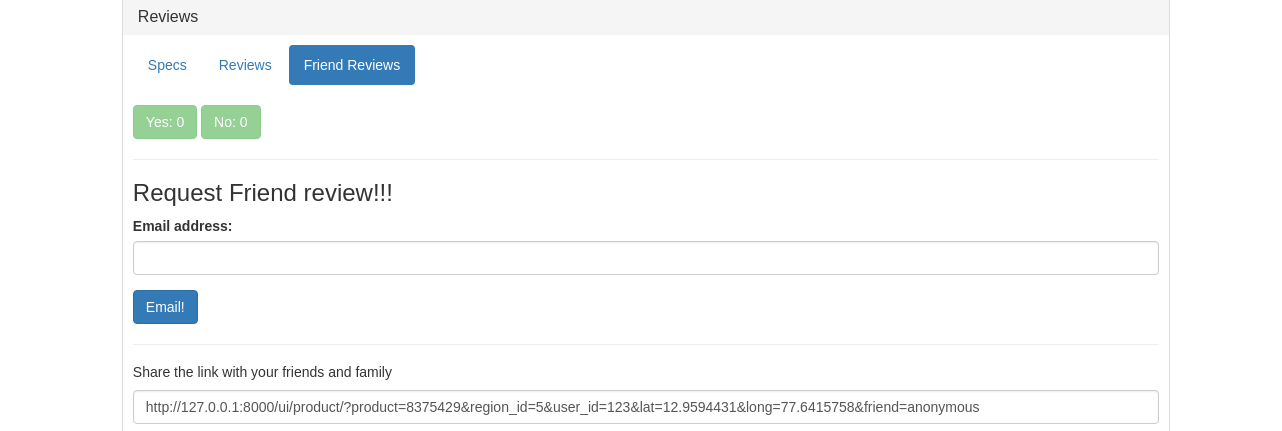
**Figure 5.6**

**Product details**



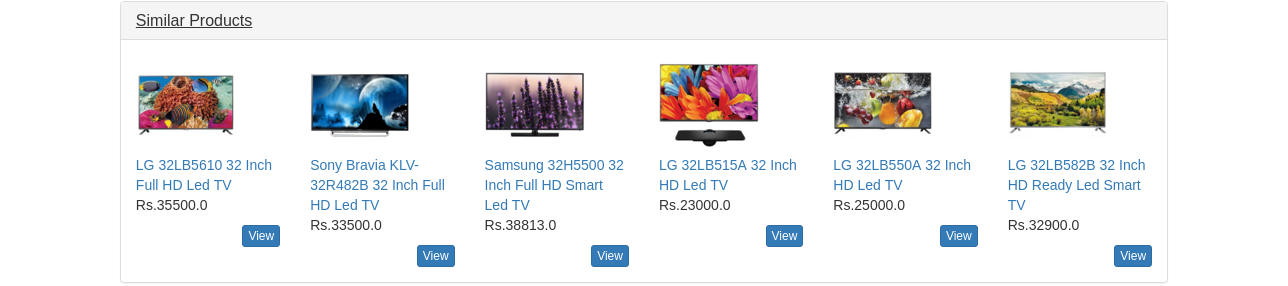
**Figure 5.7**

**Expert Reviews**



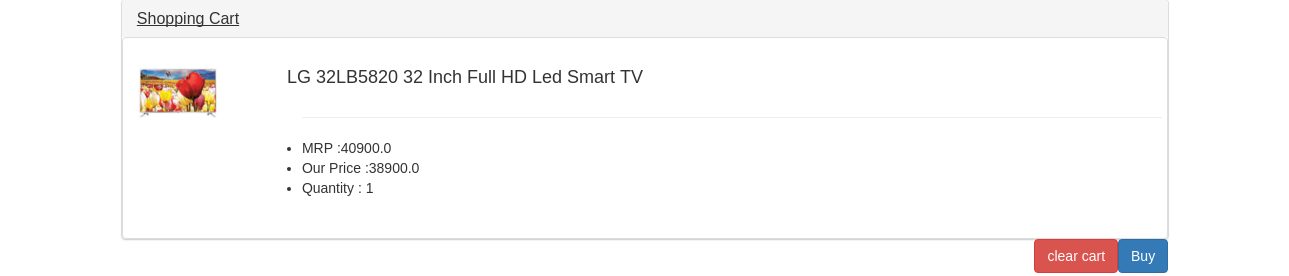
**Figure 5.8**

**Friend Review System**



**Figure 5.9**

**Similar Products**



**Figure 5.10**

**Dynamic cart**

**5.8 Conclusion**

The ZINAFAS project has been a major source of inspiration for the Zopper website. Many core components of ZINAFAS have been used in making the Zopper website, most importantly the review system and the dynamic map system which lists nearby store and service centre details.

The website will be deployed into production in the upcoming weeks and multiple features will be implemented such as a real-time chat system for the consumers,bidding-system for retailers on products sourced from national distributors,etc.

**REFERENCES**

1. <https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying>
2. <https://github.com/Flipkart/aesop>
3. <https://github.com/linkedin/databus>
4. https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-sharma.pdf
5. <http://docs.ansible.com/ansible/index.html>
6. <http://www.django-rest-framework.org/>
7. <https://docs.djangoproject.com/en/1.9/>
8. <http://docs.celeryproject.org/en/latest/>
9. <https://www.rabbitmq.com/documentation.html>
10. <http://kafka.apache.org/documentation.html>
11. <https://zookeeper.apache.org/doc/trunk/>
12. <http://helix.apache.org/javadocs/0.6.5/reference/packages.html>
13. <http://redis.io/documentation>
14. <http://highscalability.com/>
15. <https://github.com/twitter/twemproxy>
16. <http://maven.apache.org/guides/>
17. <http://gradle.org/documentation/>