A Thesis/Project/Dissertation Report

on

SECURING AND SIMPLIFYING FILE SHARING WITH BLOCKCHAIN

Submitted in partial fulfillment of the

requirement for the award of the degree of

BTech Computer Science Engineering

GALGOTIAS UNIVERSITY

(Established under Galgotias University Uttar Pradesh Act No. 14 of 2011)

Under The Supervision of
Dr. Shobha Tyagi: Associate Professor

Submitted By

Danish Jamal
18SCSE1180045, 18021180046

Ayush Tiwari
18SCSE1140036, 18021140082

SCHOOL OF COMPUTING SCIENCE AND ENGINEERING DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING GALGOTIAS UNIVERSITY, GREATER NOIDA
INDIA December, 2021

Table of Contents

Acronyms

B.Tech.	Bachelor of Technology
M.Tech.	Master of Technology
BCA	Bachelor of Computer Applications
MCA	Master of Computer Applications
B.Sc. (CS)	Bachelor of Science in Computer Science
M.Sc. (CS)	Master of Science in Computer Science
SCSE	School of Computing Science and Engineering

Abstract

In today’s world, where data is more valuable than anything, for an estimate 2.5 quintillion bytes of data is produced every day (that’s 2.5 followed by a staggering 18 zeros!) hence comes a responsibility of managing and securing it. Along with the huge quantity of data comes a need of sharing it, and spreading information securely over the internet. In most of the today’s technology all the sharing services uses the on-premise database for storing users’ data and provide different strategies for securing and avoiding file tampering which to some extent provide user security but still has the big loop hole of single point of failure and doesn’t guarantee 100% fault tolerance against file data.

We propose the idea of making the file sharing decentralized, where we don’t need to focus on different strategies over securing user files and data because that can be handled by blockchain. Our system will be responsible for user identification and providing different filter and sharing options where user can have full control over managing the files and related sharing options. With this solution we can focus more over business logic and providing better user experience without the need of designing the complex security mechanism.

With all those proposed solution comes a great responsibility of implementation and choosing favorable tech stack to support our idea and deliver great user experience without compromising the system performance. In the proposed system, all files are decoupled from the blockchain and stored in the Inter Planetary File System (IPFS) in a distributed manner. To further reduce storage requirements on the user side, the proposed system contains a cloud storage for managing and controlling users’ data.

As a result of the proposed system, user can experience the power of easiness when sharing files over internet without compromising the security. Proposed system will provide and android app and a website for clients to interact with our system.

The proposed system addresses the need of securing files using decentralized environment along with the easiness of file sharing and full control over file and sharing management.

Introduction

In this modern era where technology has taken over most of our day to day schedule, privacy and integrity remains a major concern. There is a race going on between tech giants for the most important currency ie. data. It won’t be too difficult to conclude that none of our data remains private and also we don’t actually own our data. We use drives and drop-boxes to store our confidential information on cloud, so that it can be accessible from anywhere. These data get stored on centralised servers owned by tech giants like google, Microsoft , Apple etc. Have you ever considered what would happen if these servers goes down? Or if google denies the user to use its servers? Obviously there are legal actions that could be taken but wouldn’t it be more better if all our data gets stored in many different locations rather than at a single place.

A decentralised file sharing system could serve as an initial step to share data using decentralised network. Decentralisation implies whatever data that’s present on the network is held by all the participants of the network. There is no central authority controlling the system. With the introduction of blockchain technology, decentralisation is revolutionising the web 2.0 to web 3.0. The blockchain has some inherent benefits such as decentralisation, immutability and audibility. These properties allows the construction of decentralised file sharing system. This system could also have the possibility of including liquidity and tokenising the files. Cloud storage and distributed storage are considered as appropriate storage options to store and share the files. A good alternative to central cloud servers is IPFS(Inter Planetary File System) and we’ll be using IPFS for our file sharing platform. IPFS is content addressable storage, with this we can forget about 404 error. In brief, IPFS returns a hash value for every information we upload on it. All the data is stored in decentralised way on IPFS and using the hash value, we can access our data.

The project focuses on utilising IPFS to create a platform where user can quickly upload and share a file, the file could be media, document or even a simple text. On uploading a file, the user receives a short url which he can share with people whom he want share his file. When the receiver opens the url, the corresponding file will be automatically downloaded to his system. The best part of this system is, there is an abstraction layer between IPFS and the user. The user can never directly access the hash provided by IPFS. The platform will also monitor links created by user, size of file uploaded and general queries to improve UX. It removes the complexity of google drives and all the data that we have to provide to google in-order to use their service. In our system, user just uploads the file, gets a link and he is good to go.

1.2 Formulation of problem

As technology has progressed, the internet has become a vast and complex web of data and files that communicate using the HTTP. Due to increased traffic, the sheer volume of information transmitted has become enormous, HTTP has started to crack under this strain. For example, each time we load a web page, HTTP is used to retrieve content from centralized servers. If the content involves transmitting large files, it may consume a lot of bandwidth. If a server is taken down, a website might still exist but with missing pieces, such as images or graphic files.
Furthermore, due to a reliance on centralized servers, HTTP makes it easy to introduce censorship.
Torrenting is the best-known solution by the general public. Torrenting has been used as a way of distributing much larger files, such as audio and video, over the internet to overcome the challenges of using HTTP.
However, the earlier versions of file sharing protocols also have some limitations. Nodes are generally run by volunteers. They can choose to stop volunteering their services, meaning that there's no guarantee there will always be enough people to host files.
Moreover, we require login/signup to create new files for sharing and that requires sending the user data to the central server. In other words, we are exchanging our data in order to be able to access file sharing services.
Using blockchain technology is a way to create robust decentralized file sharing networks where participants are incentivized to continue contributing. A token-based reward system ensures there are always enough nodes providing their services to the network. User does not have to create any sort of account for uploading, accessing or sharing his files.

The major problems leading to development of decentralized sharing system includes:

- Central servers controlling the storage of files in the existing system.
- Increasing importance of user data and concerns about privacy
- Data theft and cyber-attacks leading to data leaks
- Need of revolutionary technologies to provide more security to user data on cloud

1.2.1 Tools and technologies used

We have used and worked on following technologies:

- IPFS :- To store the data of files uploaded by the user
- DynamoDb: To store user and file metadata and hash values received by the IPFS
- React: For building client side web application
- Node.js : For writing lambda functions to handle queries and file rederivation
- AWS Cognito :- To provide authentication interface and create user pool
- AWS API Gateway :- To server as a middleware for running lambda functions based on API endpoints
- Android Native development using Kotlin: For building android client application. Based on Google’s material guidelines and uses latest android architecture pattern for optimal performance and modular development

Literature Survey

Blockchain technology has been introduced by an author using the alias of Satoshi Nakamoto and lead to a series of developments that shaped decentralized applications later on. Blockchain is a decentralized technology which is built around a data structure that provides a verifiable, immutable, distributed ledger mechanism. The second generation of Blockchain technology enabled users
to build decentralized applications (dApps) using smart contracts running on a decentralized virtual machine (Ethereum VM). dApps are verifiable, autonomous, secure and stable applications. They lead to development of applications that does not need a third party to establish a trust mechanism between the users of the application.

There are two major decentralized file sharing service providers. BitTorrent was developed in 2001 as a peer-to-peer file sharing protocol and was acquired by Tron in July 2018. By that time, BitTorrent had reached 100 million monthly active users around the globe.
BitTorrent announced the launch of the BitTorrent File System, or BTFS, based on the Tron network. The launch of BTFS addresses two needs within the decentralized file storage segment. Firstly, it introduces incentivization to BitTorrent's peer-to-peer network, allowing participants to be rewarded in tokens for their contributions.
Secondly, it provides a decentralized file storage solution to decentralized applications running on a blockchain. File storage on a blockchain is expensive, meaning that many developers default to centralized solutions. BTFS aims to address this gap, introducing decentralized file storage that's both cost-effective and accessible. BTFS is live now.

IPFS
Interplanetary File System (IPFS) is a distributed file system which uses content-addressable naming convention. The contents of a file are hashed that are used to address them universally. The files can not be modified once they are created. IPFS does not have an access management layer since it serves to the public domain. A file can be accessed by anybody that know its content name. IPFS provides an alternative for public services such as HTTP and FTP.
Blockchain frameworks such as Ethereum has limited capacity to store data within their distributed ledgers. When a block is added to the ledger it will not be deleted forever. Therefore for decentralized application that require storing large amounts of data external solutions must be used. As the decentralized applications should not depend on centrally control systems, using centalized servers or privately owned distributed solutions such as cloud servers will com-

primise the privacy of the application. For this reason IPFS and similar systems are the proper solutions for storing large amounts of data.
Access control of IPFS has not been implemented natively. Therefore applications that are going to use IPFS generally built custom mechanisms for access control or use it as it is, meaning that their files are public. There are a number of attempts to build frameworks over IPFS and blockchain such as IPFS is also suggested to be used for storing transactions in a blockchain as in. The

primary difference with our framework with earlier work is that our framework is designed with the needs of private data

IPFS, aims to solve a similar need. Protocol Labs launched IPFS in 2015 as a peer-to-peer file sharing protocol. Protocol Labs has also been developing Filecoin, its own blockchain layer, to complement IPFS.
BTFS is integrated into decentralized video streaming network DLive, enabling fast and censorship-resistant livestreaming. This marks yet another milestone and example wherein decentralized file sharing proves to have unlimited potential.
Namecoin (Haferkorn and Quintana Diaz, 2015) is an open-source blockchain technology that implements a decentralised version of DNS. The main benefits of a decentralised DNS approach are security, censorship resistance, efficiency, and privacy. Alexandria (The Decentralized Library of Alexandria, 2015) is an open-source blockchain-based project that provides a secure and decentralised library of any kind of media while allowing the freedom of speech. Both systems may be enhanced utilising digital identity services which can confirm an individual's identities (e.g. using pseudonyms), enabling security and anonymity in a standardised verification model (Swan, 2015, Zhang et al., 2017).
In Zyskind et al. (2015a) the authors propose a decentralised P2P blockchain-based platform that comprises three types of entities: (i) users, which interact with the applications; (ii) services, which provide such applications and process users' personal data for operational and business-related reasons; and (iii) nodes, entities that receive rewards in exchange for maintaining the blockchain. Since only hash pointers are stored, users have control over their data.

In our user model, there are three user types: data owners,data creators, and data requesters. The data owners are the owners of the sensitive data. For electronic health records, patients are the data owners. The information stored in files are directly linked to the owners. Our model currently does not support co-ownership. However there might be cases where a medical test contains sensitive information for more than one person such as DNA compatibility test for marriage. We plan to address co-ownership in the future, where access rights are managed by more than one data owner. The second user type is data creator who has the right to create files that contains sensitive information. These will be mostly medical, financial or legal institutions where personal data are gener-

ated. The data owners can also be data creators as well. For example, by filling up a questionaire, a data owner can create sensitive data which is not necessarily require involvement of an institution. Data creators should interact with data owners to create a file. They do not have any right after the creation of the file accept the initial read access permission. The third user type is data requester. Access to files are given to these users by the data owner. There is only one type of permission: read. The permission can be given for a specific period of time or it can be revoked by the data owner later on. Since the files are stored in a public server, the access right is also combined with encryption to ensure that the data is confidential to the rest of the world.
Access rights can be given by only data owner. There are basically three types of permissions: read, revoke and create. When a data creator wants to create a file, create permission is requested from the data owner. Once a file has been created, it will never be deleted or its contents are modified. The files of a data owner constitutes a history record of an individual. Hence it is essential to relate all of the files owned by an individual. When an access right is given to a data requester, all files of the data owner can be read by the requester. Partial sharing of a user's history is also possible. However in that case, the data requester is notified that the data does not cover the full history. The current implementation of the framework does not support partial sharing yet.
All permission requests and grants are recorded in blockchain. Therefore, for an individual it is possible to keep track of the users who had read the sensitive files. This will provide a full traceability of access history of users and enable a transparent log of accesses. The permission requests and grants are handled by smart contracts and recorded in blockchain. The request may have a timeout period so the permission can be revoked automatically. If there is no timeout for the grant, the permission can be revoked by the data owner by another smart Contract.

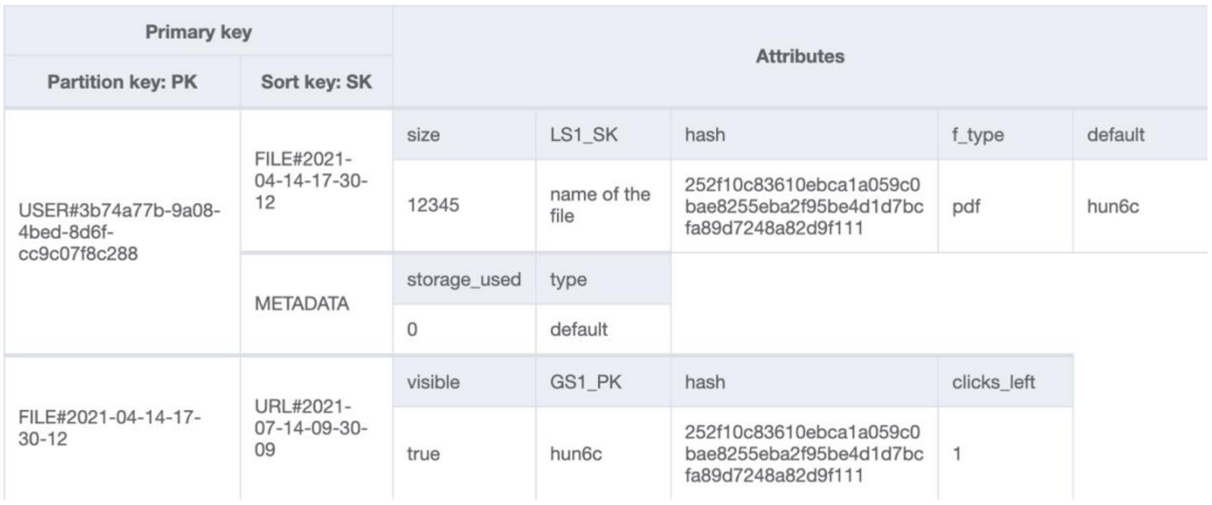
There is no personal identity in Ethereum network. All entities in our framework are represented by their digital signatures their wallets. The wallets in a blockchain platform provides account information and history of all transactions for a given individual. However the wallet addresses and real personalities are not explicitly tied into each other. Therefore even tough all sharing history of an individual are recorded on blockchain, it is not possible to reveal the real identity of a person by third parties.
Our framework use Metamask browser extension as the wallet. The users of the platform can login into their wallets and start using the framework over browsers. The following diagram displays the general structure of the framework. As see in the figure our framework acts as a layer over IPFS. Even though the files are stored in IPFS users cannot access them directly. Each data owner can see their history data, permission logs and access logs. The transaction regarding the access permissions and actual reads are stored in Ethereum blockchain.

The proposed framework relies on IPFS and Ethereum network. The framework is implemented as a higher level layer over IPFS and Ethereum.
shows the architectural components of the system. The users are identified by their private keys in the framework. There are three modules that takes care of access rights, logging and direct accesses.

When a new file is added into system or when a right is requested, users interacts Access Control Module. Logging Module is used to report logs of the file system. Direct Access Module is used when a data requester tries to read a file. Interaction with each module is done through smart contracts that are executed in Ethereum VM and file operations are performed in IPFS.
The files of a data owner shares a meta-data file which contains information about the access rights, cryptographic keys, and other file related information. When a data requester is granted a read right, a copy of the files are encrypted by a secret key and the secret key is encrypted by public key of the data requester. The resulting ciphertext stored in the meta-data file. When the requester wants to read files, the secret key of the files are revealed through a smart contract as well. Once a user gets the secret key, the files can be read until the read right of the requester is revoked. When the right is revoked, the files are encrypted again by a new secret key. As the content-names of files depend on their contents in IPFS, once a file is encrypted with another key, the name of the file changes and becomes unavailable to public.

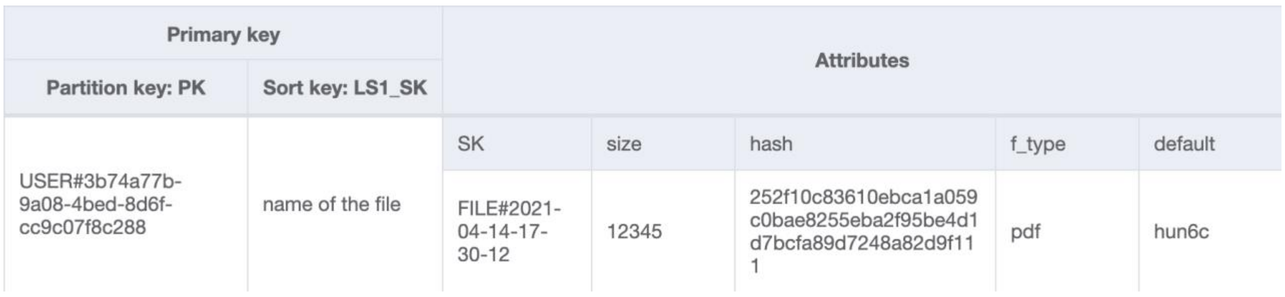
List Of Tables

Main Table Design: Used for storing data to be accessed directly using entity id.



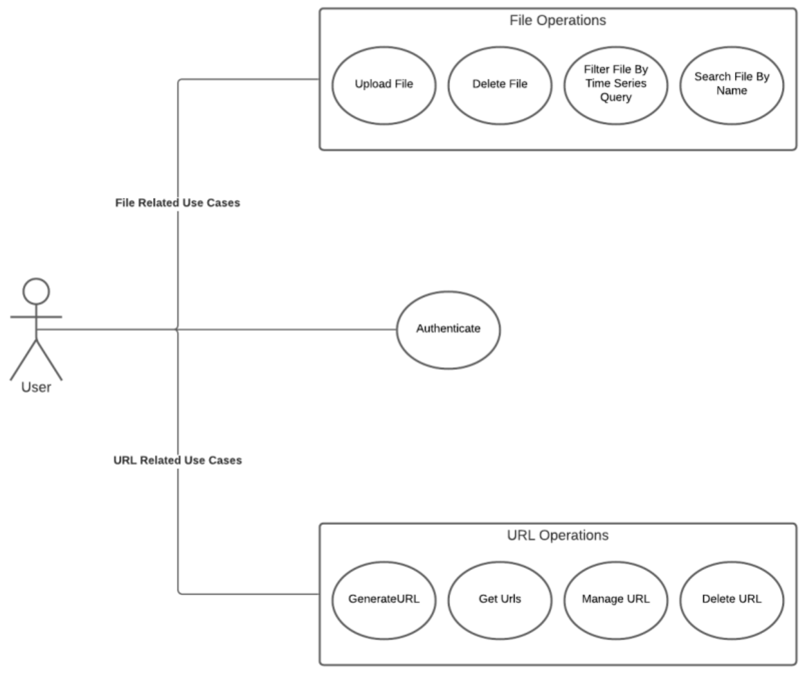
Primary key		Attributes				
Partition key: PK	Sort key: SK					
USER#3b74a77b-9a08-4bed-8d6f-cc9c07f8c288	FILE#2021-04-14-17-30-12	size	LS1_SK	hash	f_type	default
		12345	name of the file	252f10c83610ebca1a059c0bae8255eba2f95be4d1d7bcfa89d7248a82d9f111	pdf	hun6c
	METADATA	storage_used	type			
		0	default			
FILE#2021-04-14-17-30-12	URL#2021-07-14-09-30-09	visible	GS1_PK	hash	clicks_left	
		true	hun6c	252f10c83610ebca1a059c0bae8255eba2f95be4d1d7bcfa89d7248a82d9f111	1	

Local Secondary Index: Used for querying files with their name. Index the table based on name of the file for faster access and optimum performance.



Primary key		Attributes				
Partition key: PK	Sort key: LS1_SK					
USER#3b74a77b-9a08-4bed-8d6f-cc9c07f8c288	name of the file	SK	size	hash	f_type	default
		FILE#2021-04-14-17-30-12	12345	252f10c83610ebca1a059c0bae8255eba2f95be4d1d7bcfa89d7248a82d9f111	pdf	hun6c

Global Secondary Index: Used for direct access to the URL. Creates a index for short URL so that we can directly access file hash based on the short URL without the need of entity id and any filtering options.

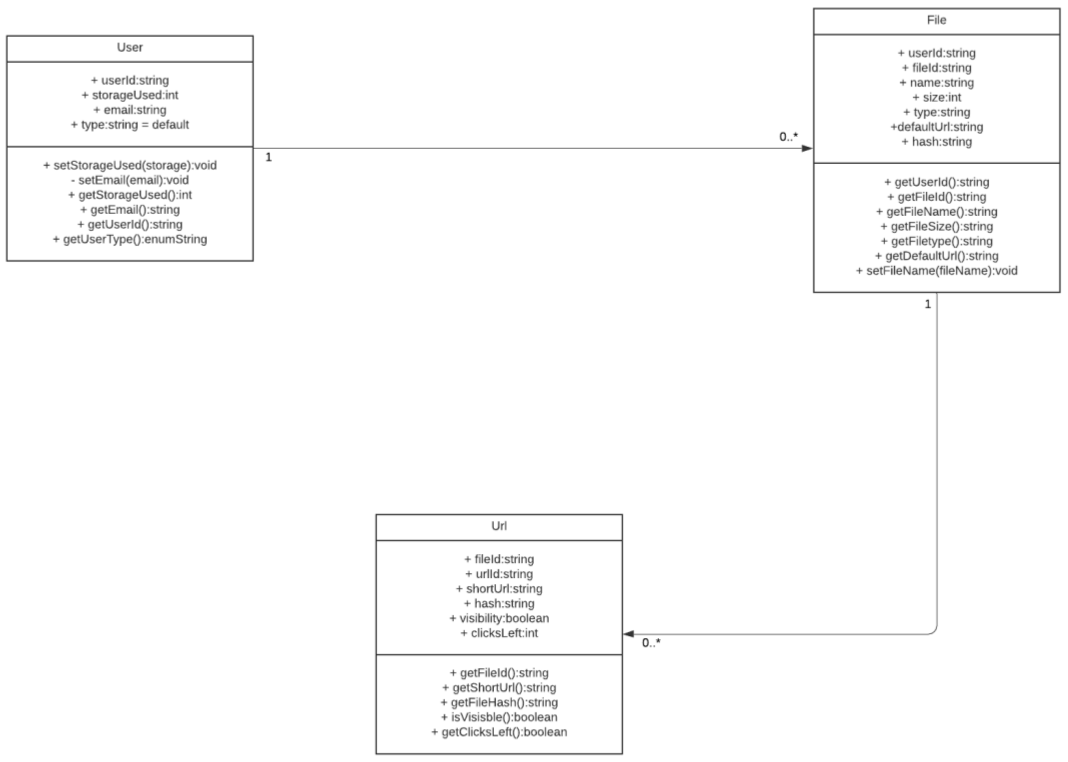


Use Case Diagram

Our system's use case can be broadly divided into three parts:

1. File Related
2. Authentication
3. URL Related

Where each broader category contains CRUD use cases along with some custom use cases.



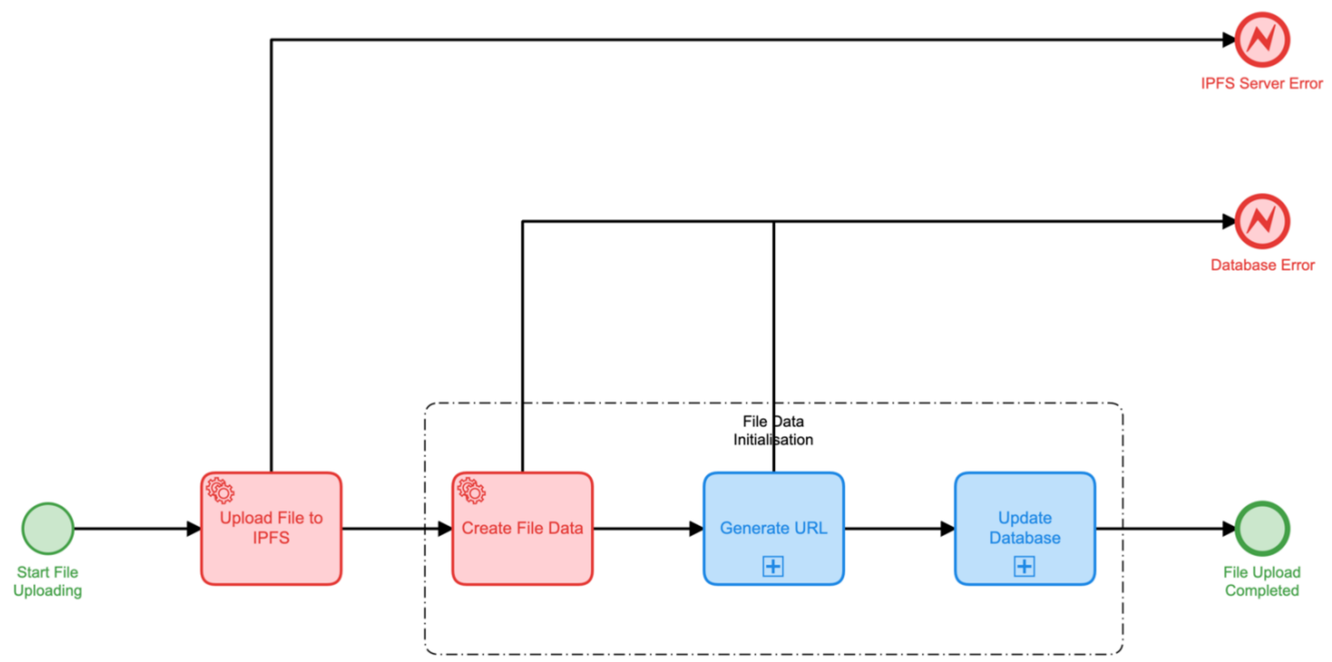
Class Diagram

The proposed system mainly has 3 classes:

1. User
2. File
3. URL

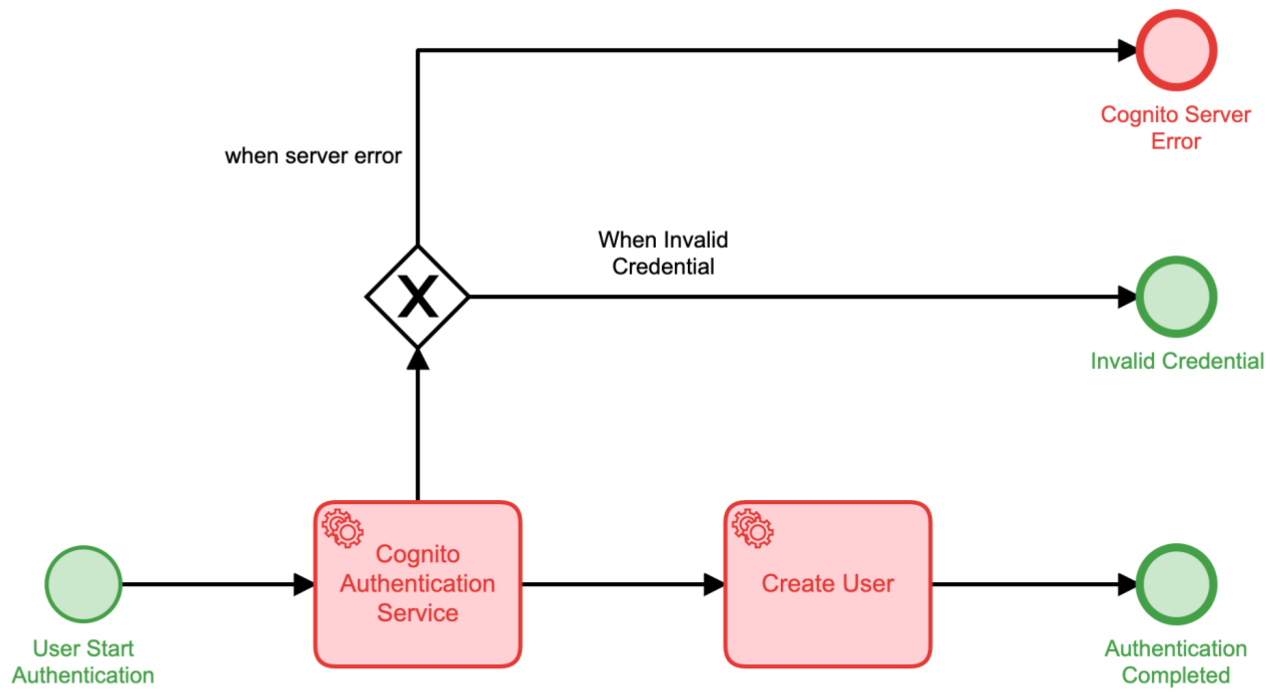
Where the class relation can be defined as:

1. User to File -> 1 to Many
2. File to URL -> Many to Many
3. User to URL -> Many to Many (Indirect)



BPMN Diagram for file upload

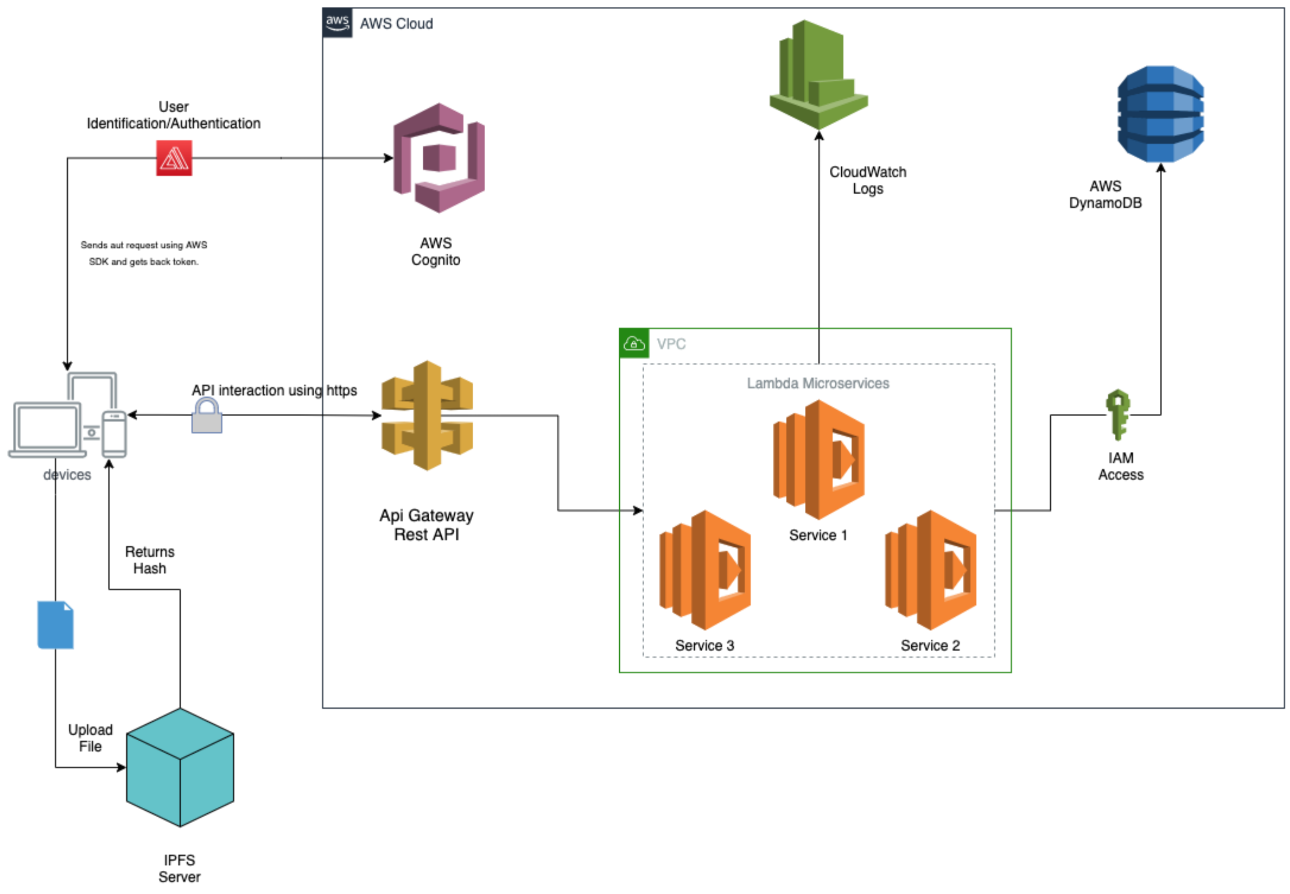
This diagram shows the business flow for the file upload. Each of the boxes with settings icon on it represents a service and boxes with plus icon on it represents the sub-task within the group. As we can see in File Data Initialisation block we have "Create File Data" microservice along with two sub task. Hence first the file is uploaded in IPFS server which is part of "Upload File to IPFS" microservice next if it succeeds the flow reaches to "Create File Data" block which is responsible for creating file meta data and updating the local database.



BPMN Diagram for user authentication

This diagram shows the business flow for the user authentication. Each of the boxes with settings icon on it represents a service and boxes with plus icon on it represents the sub-task within the group. Since we are Cognito as our authentication service we need not need to take care of JWT or any such strategies hence this flow is pretty simple and straightforward. Once user send the authentication request it is redirected to Cognito and once the user is authenticated the request is send to “Create User” microservice which is responsible for adding user meta data into our primary database. Since these data will be frequently accessed in the client side application and these data are the base for setting up the database index structure with proper configuration for optimal performance of the database.

Modules Description



System Architecture

This is the high-level system architecture of our proposed solution. We are using AWS cloud services since it is one of the widely used cloud provider and also it provides free starter resource for dev testing and building.
We are using cloud watch for monitoring all the logs. All the logs of the system go at one place and is organized in a topic fashion which makes it our ideal choice for logging, hence in case anything goes wrong we can directly log into cloud watch rather than monitoring each services individually in the system.
Apart form that we are using IAM(Identity Access Management) provided by AWS for the interacting between different components withing the system. Hence we don't need to open any external port for accessing the services making the system secure from external interference. Along with that all the request send to API gateway is secured using HTTPs making the request secure.
All the user identification part is handled by Cognito hence we can focus more on business logic rather than implementing authentication strategies and managing user database.

Our system architecture has 4 major components:

1. IPFS: Responsible for storing files in decentralized network.
2. Cognito: Provide user identification and authentication.
3. API Gateway: Exposes services to client using rest endpoint.
4. Microservices: Different services having separate concerns.
    a. May interact with database, in our case DynamoDB.

Our design architecture is divided into 4 modules.

1. AWS Lambda

“Serverless” has been the buzz word for several years now, with many applications choosing to implement the serverless approach. The term originated from the idea that the infrastructure used to run your backend code does not need to be provisioned and managed by you and your team. This significantly lessens the time it takes to get your application production-ready as well as the time and effort required to maintain your infrastructure. In 2014, Amazon Web Services released a product that would eventually become a gem in the wide pool of serverless solutions; that product is known as Lambda. In this article, we’ll take a look at why Lambda is worth your attention as well as the disadvantages you’ll want to consider, we’ll walk through the most prominent features of this service and explore its inner workings.

What is it?

As a brief overview, AWS Lambda is a function-based computing service that takes the efforts of provisioning and maintaining its infrastructure out of your hands. With Lambda, you don’t need to worry about scaling your infrastructure and removing unnecessary resources as this is all handled for you. We’ll take a deeper dive into how this service works, but first let’s take a look at why this tool is a worthy addition to your stack.

Advantages

Many of the advantages of using AWS Lambda relates to the advantages of adopting the serverless-approach in general. As mentioned in the intro, a major benefit of going serverless is the time and effort saved from creating and maintaining your infrastructure. AWS provisions and manages the infrastructure your Lambda functions run on, scales the instances to handle times of excessive load, and implements proper logging and error handling. Anyone that’s been involved in the creation or maintenance of infrastructure will understand the gravity of this advantage. Not only is there a large amount of time involved in building a system that suits the needs of your application, there is also a considerable amount of time required to maintain that system as your application evolves. Time saved means quicker time to market for your application, greater agility as your team is able to most faster, and more time spent on more important tasks such as bug fixes or new features.

As for why AWS Lambda is one of the most popular serverless solutions, AWS has done a very good job of ensuring Lambda accommodates for applications at scale as well as applications in early stages. For applications with large amounts of load, AWS allows you to run your Lambda function simultaneously with other Lambda functions; meaning, you won’t need to worry about clogged up queues. Not only that, multiple instances of the same Lambda function can be provisioned and run at the same time. Both advantages ensures that no matter how much load your application is under, Lambda will be able to handle it. Another advantage of using AWS Lambda is that you only pay for what you need; accomodating for applications that are not yet at scale or have widely differing loads. AWS charges you for the number of requests your Lambda functions recieve and the time it takes to execute those requests per 100ms. Despite its wide array of advantages, there isn’t a single solution that exists without its share of disadvantages and AWS Lambda is no exception.

Disadvantages

Moving the task of maintaining your infrastructure away from your team and in the hands of a provider results in less control and flexibility, which is the biggest disadvantage of the serverless approach. On top of that, services that help implement the serverless approach come with their own set of infrastructure-related limitations; in Lambda’s case, these limitations are the following:

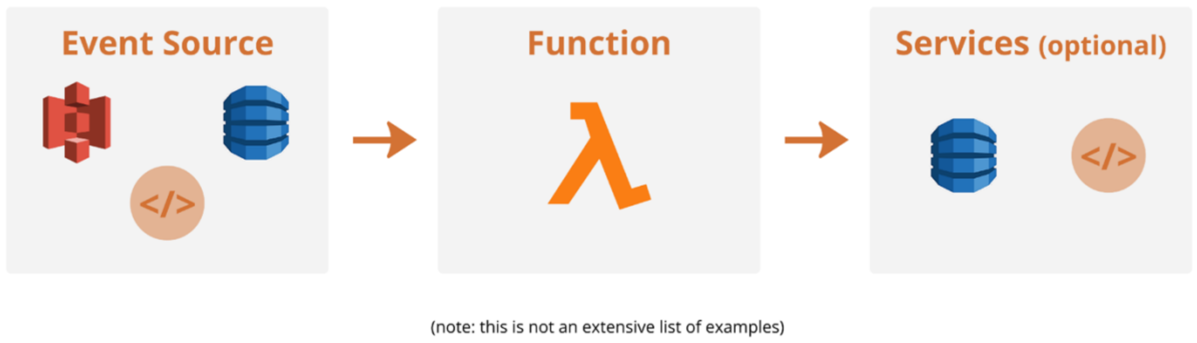
- Functions will timeout after 15 minutes.
- The amount of RAM available ranges from 128MB to 3008MB with a 64MB increment between each option.
- The Lambda code should not exceed 250mb in size, and the zipped version should be no larger than 50mb
- There is a limit of 1,000 requests that can run concurrently, any request above this limit will be throttled and will need to wait for other functions to finish running.

Whether or not these limitations will impact your application is dependant on the nature of your Lambda functions; usually, the solution is to refactor your Lambda functions to improve their efficiency. If any of these limitations begin to impact your Lambda functions, the first thing to do is to investigate why and whether your functions could be improved. For example, is the reason your function is timing out is because there's inefficient algorithms involved? Are there any unnecessary dependencies in your Lambda code, causing its size to exceed the limit?

Cost was mentioned in our list of advantages, but although you only pay for what your application requires this does not necessarily result in a cost effective solution; during times of high load, the cost of the same infrastructure on AWS EC2 or other services may be cheaper. The price of other services based on your application's needs should be considered especially if your application experiences consistently high load. The final disadvantage worth mentioning is the small latency time between when an event occurs and when the function runs. This small latency times only occurs in some cases — during a *cold start*. In most cases, these latency times are so miniscule that it's hardly an issue but it's still worth considering if your application is already bordering towards potential load problems; I'll talk about cold starts in more detail in a later section.

How does it work?

If you've decided that Lambda may be a worthy addition to your stack, the next step is to understand the inner workings of a Lambda function. On a very basic level, serverless applications are made up of 2 or 3 components; these are event sources, functions and (in some cases) services. An event source encapsulates anything that can invoke a function, such as uploads to an S3 bucket, changes to data state or requests to an endpoint. When any one of the designated events occurs, your Lambda function will run in its own container. The resources allocated to that container and the number of containers used is determined by the size of the data and the computational requirements of the function, this is all handled by AWS. Once the request is completed, your Lambda function will either return a result back to the invocation source or a connected service, or it could make changes to a connected service (such as a database).



AWS Lambda Flow Chart Diagram | created by Author

Before you can run a Lambda function, you'll need to create one and to successfully do so, you'll need a basic understanding of what's involved. A

Lambda function consists of 3 or 4 parts; the handler function, the event object, the context object and in some cases a callback function. The handler function is the function that will be executed upon invocation, this can either be async or non-async. Asynchronous functions take an event object and a context object whereas non-asynchronous functions take both these objects and a callback function. The event object contains data that was sent when the function's event was triggered, this includes information such as the request body and the uri; the data that is passed through depends on the invocation service. The context object contains runtime information such as the function name, function version and log group. The callback function is only passed through to synchronous handlers and it takes two arguments: an error and a response. Once the Lambda function is created and pushed up to AWS, it is compressed along with its dependencies and stored in an S3 bucket.

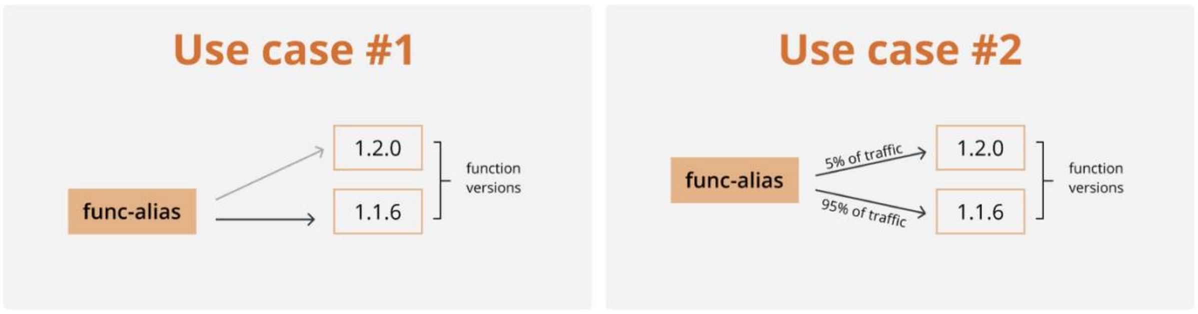
For non-async handlers, function execution continues until the event loop is empty or the function times out.

Layers

Once you start to build your Lambda functions you'll notice that there's pieces of logic that could be shared between multiple functions, this is when layers can come in handy. Layers allow you to reuse code across several functions without needing an additional invocation. Once you've identified a piece of code that could be reused, implement it as a layer and attach it to the functions that need it. A layer is created in the same fashion as a Lambda function, with slight configuration changes which depend on the method you've chosen to deploy your functions (via AWS's GUI, using the serverless framework, or AWS's CLI tool). This is also true for adding a layer to a function; it can easily be done via AWS's GUI, by adding a few extra lines to the *serverless.yml* file if you're using the serverless framework, or with a single command using their CLI tool. The use of layers can help improve the maintainability and cleanliness of your Lambda functions as you'll be able to significantly lessen the amount of code necessary for each function as well as the amount of duplicate code across your application.

Versioning and Aliases

AWS not only allows you to save different versions of your Lambda functions but also allows these versions to coexist and run at the same time, this gives consumers of your functions the flexibility to upgrade to newer versions as they please. Aliases are used as a pointer to a particular version of a Lambda function, there is a long list of use cases in which they can be utilized in; two of which is worth mentioning. Firstly, instead of updating the version of a function everywhere it's called, you could use an alias in these areas and update the version that the alias points to. Secondly, aliases have the ability to point to two versions and give you the flexibility to determine the percentage of traffic to be sent to each version. This can be very useful if you and your team wanted to test a new version of a function with a small percentage of your traffic before releasing the new version universally.



Alias Use Cases Diagram | Created by Author

Permissions

Relatively speaking, your Lambda functions are considerably secure by default; your function can’t talk to other services nor can it be invoked by any client, you’ll have to enable it to do so. Permissions surrounding your Lambda functions fall into two buckets: execution policies and resource-based policies. Execution policies determine which services and resources a Lambda function has access to, as well as which event sources can trigger a Lambda function to run. Resource-based policies grant other accounts and AWS services access to your Lambda resources, these include functions, versions, aliases and layer versions.

Resilience

AWS helps to ensure that your Lambda function is able to handle faults without impacting your entire application using a set of features they’ve included into Lambda. The most notable features have already been mentioned in this article and those are Lambda’s scalability, versioning and ability to run concurrently. A couple of other features that contribute to the service’s resilience is their use of multiple availability zones and the ability to reserve concurrency. By default, AWS runs your Lambda functions in multiple availability zones, this ensures that your functions are not impacted if a single zone is down; the same cannot be said for services such as EC2 where this behaviour must be set by the developer. With Lambda, developers have the ability to set reserved concurrency for a particular function which ensures that it can always scale to (but not exceed past) a set number of concurrent invocations despite the number of requests other functions are consuming — note that AWS will still adhere to the upper limit of 1,000 requests, which means requests for other functions will be throttled.

Cold Starts

Cold starts occur when a function has been idle for a long enough period of time that its container has been completely terminated. A new container is provisioned when the function is invoked resulting in a small amount of latency. At times, an idle Lambda container is available to pick up new requests; if this is the case, provisioning a new container isn't necessary — this is called a *warm start*. The period of time a Lambda function can be idle for before it gets terminated isn't well-documented but an experiment in 2017 found that most functions were terminated after 45–60 minutes of inactivity; potentially earlier if resources are needed by other customers. The amount of time it takes for a function to start up is influenced by its scripting language, whether the function is outside of a VPC (if it is, start up time will be faster), how big the package size is and how much memory is allocated to the function. Whether or not your application will likely experience cold starts depends on the amount of variation between your load levels. A fairly constant amount of load will mean that your application will require the same number of containers most of the time, which results in more warm starts as a container will likely be available for most requests.

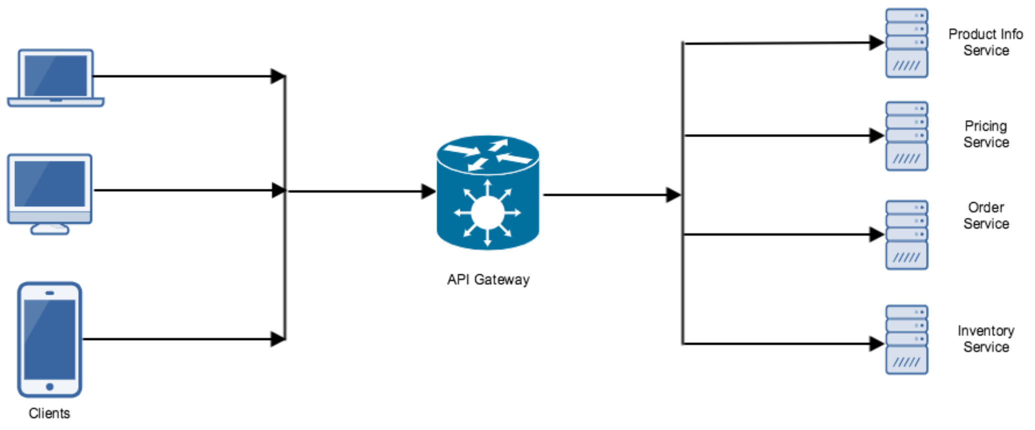
This module mainly focuses on getting the functionality of lambdas ready. Its core part include 2 minor sub modules first is getting the list of lambdas required for making the system resilient enough to handle all the request and also be economical. Secondly get the functionality of all the services be ready. We have built custom python script for building all the lambdas artifacts. The deploying part is still done manually but we are looking for the solution to automate it.
Below is the link to the current progress over the lambdas.
https://github.com/code-gambit/VT-lambdas

2. AWS API Gateway

In microservices architecture, there are several services running each designed for a very specific component of the system. When clients (Mobile Apps, Web Apps or Third party applications) communicates directly with these microservices then many problems arise.

1. **The granularity of APIs** provided by microservices is often different than what the client needs. Microservice API's are very generic and granular in nature where each returns only a portion of data for functionality. A single operation might require call to multiple services. This can result in multiple round trip network call between client and servers, adding significant latency.
2. **Network performance** is different for different kind of clients, like the mobile network is slower and high latency. WAN is slower than LAN. Making multiple network calls from clients creates an inconsistent experience.
3. It can result in **complex client code**. A client needs to keep track of multiple endpoints (host + port) and handle failures from the services in a resilient way.
4. It also creates a **tight coupling** between the client and the backend. The client needs to know how individual services are decomposed. It becomes harder to add a *new service* or refactor *existing services*.
5. Each client facing services must **implement common functionalities** like authorization and authentication, SSL, API rate limiting, access control and etc.
6. Services must only use **client friendly protocols** like HTTP or WebSocket. This limits the choice of communication protocols for services.

There comes an **API gateway** which can help to address these challenges. It decouples clients from services. An **API Gateway** sits between clients and services and a single entry point for all clients requests. It takes all requests from clients and then routes them to the appropriate microservice with request routing, composition and protocol translation.



Typically it handles requests by invoking multiple services and aggregating the results and sending it back to the client. API Gateway provides the following benefits:

1. **Isolates the clients** from how the applications are partitioned into microservices and solve the problem of determining the locations of service instances.
2. API gateway can **aggregate multiple individual requests** into a single request. This pattern applies when a single operation requires calls to multiple backend services. The client sends one request to the gateway. The gateway dispatches requests to the various backend services, and then aggregates the results and sends them back to the client. This helps to reduce chattiness between the client and the backend.
3. It also **improves client performance** and user experience by avoiding multiple round trips between client and server. Also, Multiple calls made by API gateway are running in the same network, it will be more performant than it was executed from the client.
4. **Simplifies** the client by moving the logic for calling multiple services from the client to API Gateway.
5. Allow services to use **non web friendly protocol** by translating standard web friendly API protocol to whatever protocols are used internally.
6. Gateway can be used to **offload the common functionality** from individual services.It can be useful to consolidate these functions into one place, rather than making every service responsible for implementing them. This is particularly true for features that requires specialized skills to implement correctly, such as authentication and authorization. functionalities which can be offloaded are :

- SSL termination
- Authentication

- IP whitelisting
- Client rate limiting
- Logging and monitoring
- Response caching

It includes getting all the ingresses ready which will be responsible for authentication and redirecting it to the respective microservice. API gateway also has many security feature like API keys and stagging services. We are using single API key at the gateway layer for dev testing once the testing is completed we will be generating another production API key and it will be static long secret. API model can be found at this (https://github.com/code-gambit/VT-lambdas/blob/development/Api/model.md) link.

3. Database Design and Configuration

DynamoDB is a managed NoSQL database provided by AWS, and it is a highly scalable and reliable database. We can scale from 10 to 1000 transactions per second (tps) in couple of seconds. Since it is a managed service, we don’t need to worry about the underlying hardware, servers or operating system. Data stored in the DynamoDB is redundantly copied across multiple Availability Zones so by default it provides protection for data loss due to underlying hardware failures.

DynamoDB organises data as tables and each table contains several items (rows) and each item has Keys and Attributes (columns). The tables in the DynamoDB are non relational and non schema based. This means table joins are not supported at DB level. For most use cases, we don’t need table joins and those fit really well with DynamoDB.

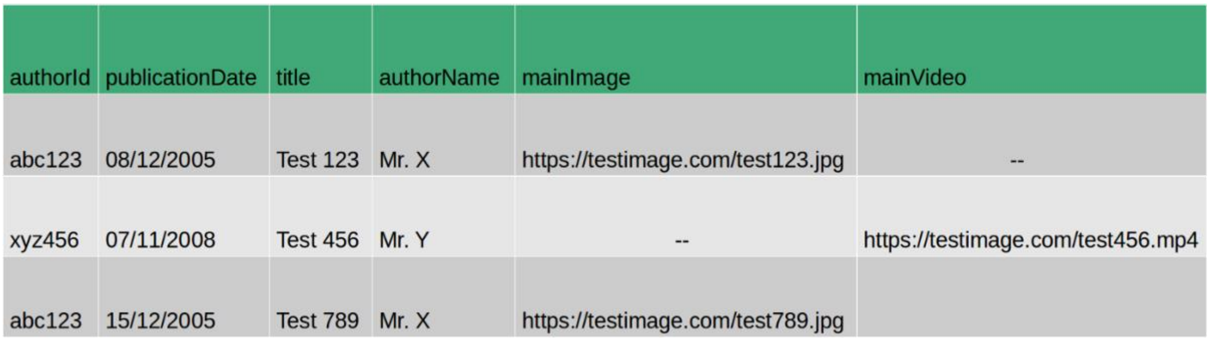
Now let us discuss DynamoDB terminology.

Tables

When we create the table we specify Partition Key and an optional Sort Key, we can’t change these later but rest of the attributes (columns) of item (row) can change. Also each item can have different set of attributes.

Example

Let us say we want to store articles from various authors in DynamoDB:



authorId	publicationDate	title	authorName	mainImage	mainVideo
abc123	08/12/2005	Test 123	Mr. X	https://testimage.com/test123.jpg	--
xyz456	07/11/2008	Test 456	Mr. Y	--	https://testimage.com/test456.mp4
abc123	15/12/2005	Test 789	Mr. X	https://testimage.com/test789.jpg	

Article_Table
Table Keys
Partition Key : authorId
Sort Key : publicationDate

Item 1 has mainImage but not mainVideo attribute, where as Item 2 has mainVideo but not mainImage attribute. This is possible in DynamoDB as it is non-schema based.

Item1 and Item3 have same partition key abc123 and with different sort keys, which is possible in dynamoDB.

There is no limit on how many items can be stored in a table.

Item

A DynamoDB item is nothing but a row in the table. We can change any attribute of an item except its keys: partition key or sort key, these keys are an identification for an item; if we have to change these keys, then the only option is to delete an item and create it again.

Data Types

DynamoDB supports different data types for attributes of an item, they can be mainly categorised into the following:

- Scalar Types : Number, String, Binary, Boolean and Null.
- Document Types : List and Map
- Set Types : Number Set, String Set, and Binary Set.

Partition Key

This key is mandatory for the DynamoDB table and item. DynamoDB partitions the items using this key, that's why this key is also called as the partition key and some times is also referred as a Hash Key.

Sort key

This key can be used in conjunction with the Partition key but it is not mandatory. This is useful while querying the data relating a Partition key. We can use several different filter functions on the sort key such as begins with, between etc. Some times it is also referred to as a Range Key.

Primary Key

Primary key is just a combination of both Partition key and Sort Key.

Batch APIs

BatchGetItem : This can be used to fetch items from different tables using Partition Key and Sort Key. In a single BatchGetItem call, we can fetch up to 16MB data or 100 items.

BatchGetItem can be performed only on tables not on secondary indexes.

BatchWriteItem: This can be used to delete or put items on one or more tables in DynamoDB in one call. We can write up to 16 MB data, which can be 25 put and delete requests.

BatchWriteItem cannot update items, for that use UpdateItem API call

Query

To query table we must pass partition key so selecting proper partition key for the table is important. Query operation will return all items that are matched with partition key of the table. Sort Key is further useful to filter and sort items but it is optional.

Scan

Scan operation does't require Partition Key or Sort Key to fetch results from the table. As the name suggests, it scans an entire table to filter results based on attribute values passed as filters.

Pagination

DynamoDB Query/Scan results return maximum of 1MB response size so if our request matches an items of size more than 1MB, it gets automatically divided into pages. In such cases DynamoDB returns a special response parameter "LastEvaluatedKey" and this can be used to fetch next page results. Please note we need to pass the value of "LastEvaluatedKey" as "ExclusiveStartKey" in the next request to DynamoDB.

In some cases we might want to fix page size to number such as 10 or 20 results per page. In those cases we can use the "Limit" parameter. Please note if the results matching to the "Limit" is more than 1MB then DynamoDB only returns subset of the results which fits to 1MB limit.

Sorting

When we use Query/Scan operation on a DynamoDB table, then by default the results are sorted based on Sort Key value of the table. Incase we want that results in reverse order then we need to pass "ScanIndexForward" as "false" in query/scan request parameters. If the data type of Sort key is a number, then the results will be in a numeric order, otherwise, results will be in UTF-8 bytes. By default sort order is ascending. To get results in a descending order, pass “ScanIndexForward” as “false”.

How to call DynamoDB APIs programmatically ?

AWS SDK for DynamoDB is supported for several languages, which can be used to interact with DynamoDB API seamlessly. We can also use DynamoDB using AWS CLI.

Pricing
Last but not least… let us see how is DyanamoDB is priced.
Free Tier :
Throughput limit : 200 million requests per month (25 read and 25 write capacity units)
Stream limit : 2.5 million read requests per month.
Storage : 25GB of indexed data storage.
Note : Above free tier does not end after 12 months.

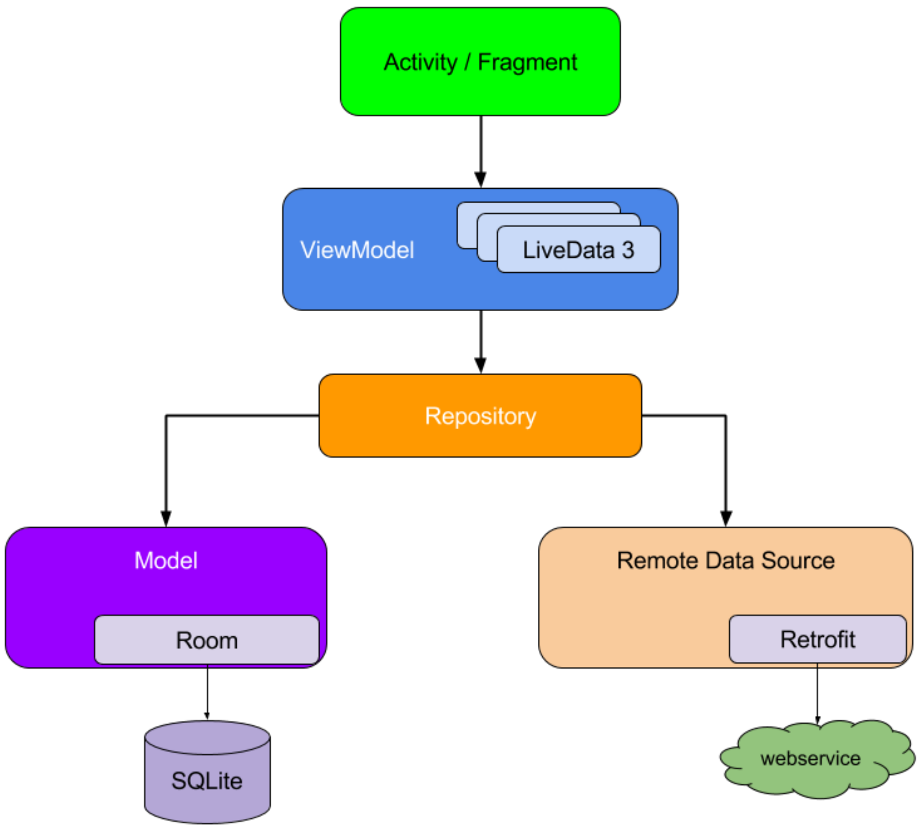
This module main motive is to get the database configuration ready. The high-level database design is already discussed in List Of Tables.
We are using DynamoDB as our primary database since it is the part of the AWS architecture. The most challenging part was to get the indexes ready since we need to minimize the indexes along with achieving our system design requirements. We have used NoSQL data modeler for modeling of the database and tested it against the data and got the optimal performance and decent cost graphs.

4. Client Application

Android Client

Android client is a android based client side application. It uses some of the latest android libraries and features. The application in based on MVVM architecture, what is MVVM architecture will be explained later. Apart from that the application is based on latest googles latest material UI/UX guidelines. Some of the key components on android application are listed below.

1. MVVM



There are mainly 3 components of MVVM architecture, namely:
Activity/Fragment: These are the classes in which we place the different views with which the user interacts. So, this is a UI part or we can say View of this architecture.
ViewModel: A ViewModel class is a class where we perform functions either which are related to the business logic or

independent of the UI.

But why do we need a ViewModel, I mean that there can another class in which we put our business logic and then use that class in our view (Activity/Fragment).

In the above case, the class will be bound to the View's lifecycle, and if a case occurs where the view is destroyed by the user's action or some device event that is out of the developer's hand. In that case, the data is will be destroyed with that. For example, your app may include a list of users in one of its activities. When the activity is re-created for a configuration change, the new activity has to re-fetch the list of users. To avoid this kind of scenarios, we make use of ViewModel by storing such data in its variables.
Repository: In this class, we perform the tasks which are related to the data sources which can be an API or a local DataBase. Since we get the data that flows in the app from the repository, we can say that this our Model.

2. Room Persistent Library

Room is a new way to create a database in your android apps, it is much similar OrmLite.
The core framework provides built-in support for working with raw SQL content. Although these APIs are powerful, they are fairly low-level and require a great deal of time and effort to use:
There is no compile-time verification of raw SQL queries.
As your schema changes, you need to update the affected SQL queries manually. This process can be time consuming and error prone.
You need to write lots of boilerplate code to convert between SQL queries and Java data objects.
Room takes care of these concerns for you while providing an abstraction layer over SQLite.
Database, Entity, DAO
There are three major components in Room:
Entity represents data for a single table row, constructed using an annotated java data object. Each entity is persisted into its own table.
DAO (Data Access Object) defines the method that access the database, using annotation to bind SQL to each method.

Database is a holder class that uses annotation to define the list of entities and database version. This class content defines the list of DAOs.

3. Caching in Android

   Whenever the user accesses the application in offline mode, the data is dispatched into the view, it can either be a fragment or an activity. If there is no data or the data is insufficient in the disk as a cache, then it should fetch the data over the network. It checks if there is a need to log in (if the user logouts, then re-login would be required). It re-authenticates, if successful then it fetches the data, but it failed, then it prompts the user to re-authenticate. Once the credentials are matched, then it fetches the data over the network. If the fetch phase is failed, then it prompts the user. Otherwise, if successful, then the data is stored automatically into the local storage. It then refreshes the view.

4. Retrofit

   Retrofit is a REST Client for Java and Android. REST Client is a method or a tool to invoke a REST service API that is exposed for communication by any system or service provider. For example: if an API is exposed to get real-time traffic information about a route from Google, the software/tool that invokes the Google traffic API is called the REST client. It makes it relatively easy to retrieve and upload JSON (or other structured data) via a REST-based webservice.

5. Dependency Inject with HILT

   Dependency injection (DI) is a technique widely used in programming and well suited to Android development, where dependencies are provided to a class instead of creating them itself. By following DI principles, you lay the groundwork for good app architecture, greater code reusability, and ease of testing. Have you ever tried manual dependency injection in your app? Even with many of the existing dependency injection libraries today, it requires a lot of boilerplate code as your project becomes larger, since you have to construct every class and its dependencies by hand, and create containers to reuse and manage dependencies.

By following DI principles, you lay the groundwork for good app architecture, greater code reusability, and ease of testing.
The new Hilt library defines a standard way to do DI in your application by providing containers for every Android class in your project and managing their lifecycles automatically for you.
Hilt is built on top of the popular DI library Dagger so benefits from the compile time correctness, runtime performance, scalability, and Android Studio support that Dagger provides. Due to this, Dagger's seen great adoption on 30% of top 10k apps of the Google Play Store. However, because of the compile time code generation, expect a build time increase.
Since many Android framework classes are instantiated by the OS itself, there's an associated boilerplate when using Dagger in Android apps. Unlike Dagger, Hilt is integrated with Jetpack libraries and Android framework classes and removes most of that boilerplate to let you focus on just the important parts of defining and injecting bindings without worrying about managing all of the Dagger setup and wiring. It automatically generates and provides:
Components for integrating Android framework classes with Dagger that you would otherwise need to create by hand.
Scope annotations for the components that Hilt generates automatically.
Predefined bindings and qualifiers.
Best of all, as Dagger and Hilt can coexist together, apps can be migrated on an as-needed basis.

6. Timber

While developing android applications we as developers tend to put in a lot of logs with in different different priorities for this sake android sdk comes in with a utility class Log, which does have utility methods to log our messages with different priorities the most common methods that we use for different cases like we uses Log.e(TAG,"message") in the cases we want to show some error (this usually appears in red color in the logcat window of android studio), or Log.d(TAG, "message") when we want to print some message for the purpose of debugging, like value of some data. Usually when the application development is completed and its the time to release the app on the play store, we need to remove all Log statement from the app, so that none of the application data such as user information, hidden application data, auth-

tokens are available to user in logcat as plain text and this becomes a cumbersome task
There are several ways to tackle this problem
1. Condition based logging
In this we can make a public static boolean variable in Application class like in the below code snippet.
ApplicationController.java
and use this variables value as check for logging like the code snippet below
MainActivity.java
and use this boolean variable at every place where you want to put a log message, and once you are ready to release the app, just set isDebug = false in the application class, but this approach puts in an extra line of code, which is completely unnecessary and can be avoided.
This simply can be achieved using android's in built build configuration, BuildConfig.DEBUG can be used instead of that boolean variable for checking if the build type is release or debug.
2. ProGuard
This approach is all also a solution to this problem, you can render the Log class ineffective and can remove those statement in the release build by putting in
into the proguard rules file.

7. Chucker

While we all have shifted to the new era of remote work, debugging bugs and issues are becoming a pain to a lot of teams now, and hence the QA is banging your head that there is some issue in your android app.
Checking API logs is a bit difficult when you are doing a lot of requests parallelly or when you do not have a logcat in front of you but the APIs are failing due to some reason, that's where Chucker helps you.
Motivation behind
At Oye! Rickshaw, we were having too much effort in testing as everybody is working remotely and the QA team has to reach out to multiple developers/teams at the same time to understand why aren't features behaving as expected and what is the cause of issues?
We began searching for a tool which could provide a solution that could solve this problem for everyone across the different teams from wherever mobile app traverse before the final release, and

found ‘Chucker’ as a perfect solution to our problem and hence the team could be self-dependent and get the issues resolved without poking extra heads.
Chucker comes in with a lot of pros for us:
API Monitoring in real-time while using the app
QA Team can report more bugs too easily
Backend testing made easier

Web Client

Web client is a web based client side application. It uses some of the latest web libraries and features. The application in based on flux architecture, what is flux architecture will be explained later. Apart from that the application is build using react and is based on googles latest material UI/UX guidelines. Some of the key components on web application are listed below.

1. React

   React is a JavaScript library (not a framework) that creates user interfaces (UIs) in a predictable and efficient way using **declarative code**. You can use it to help build single page applications and mobile apps, or to build complex apps if you utilise it with other libraries.

2. Context API

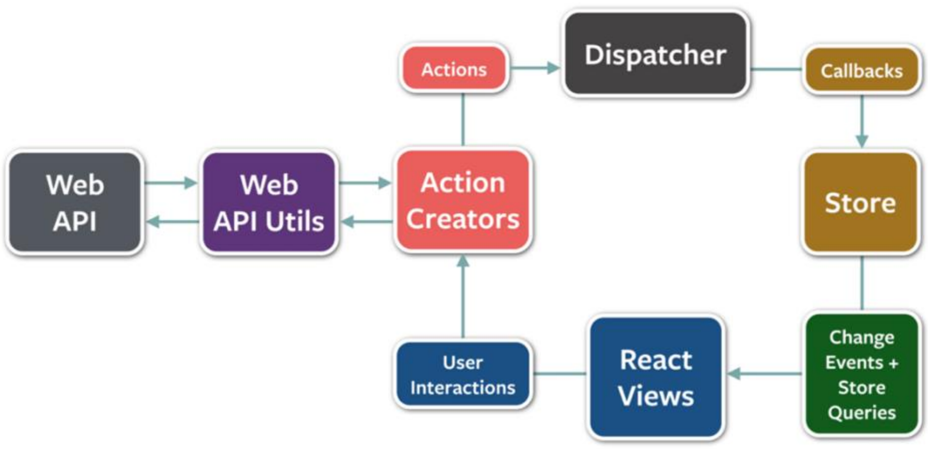
   React Context API is a way to essentially create global variables that can be passed around in a React app. This is the alternative to “prop drilling”, or passing props from grandparent to parent to child, and so on. Context is often touted as a simpler, lighter solution to using Redux for state management.
   But before moving forward, lets see why we even need a state management tool and what state management is.
   State Management:
   In all applications, you have to keep the different parts of the UI in sync. Now, what I mean by that is if there is a part of the UI with which the user interacts, there can be a need for the other part of the UI to change accordingly, and in a large application where you might have several components, you will have to write a lot of code to get all this set up. Say that you are making an e-commerce store, where

on clicking the add to basket, you want the basket count to go up by one. To make all this possible, you have to maintain the state properly, and this is where the state management tools come in handy. If you are making a single page web application, for a basic portfolio website or something, then you generally will not deal with state management issues, but if you are making a really big application, or even a relatively big application, then I am sure that you should use some kind of state management tool. Some examples here will be Redux, the Context API, Flux, or MobX.

3. Circle CI

Continuous integration is a practice that encourages developers to integrate their code into a master branch of a shared repository. Instead of building out features in isolation and integrating them at the end of a development cycle, code is integrated with the shared repository by each developer multiple times.
CircleCI is the continuous integration & delivery platform that helps the development teams to release code rapidly and automate the build, test, and deploy. CircleCI can be configured to run very complex pipelines efficiently with caching, docker layer caching, resource classes and many more. After repositories on GitHub or Bitbucket are authorized and added as a project to circleci.com, every code triggers CircleCI runs jobs. CircleCI also sends an email notification of success or failure after the tests complete.

4. Flux



Flux is an architectural pattern proposed by Facebook for building SPAs. It suggests to split the application into the following parts:

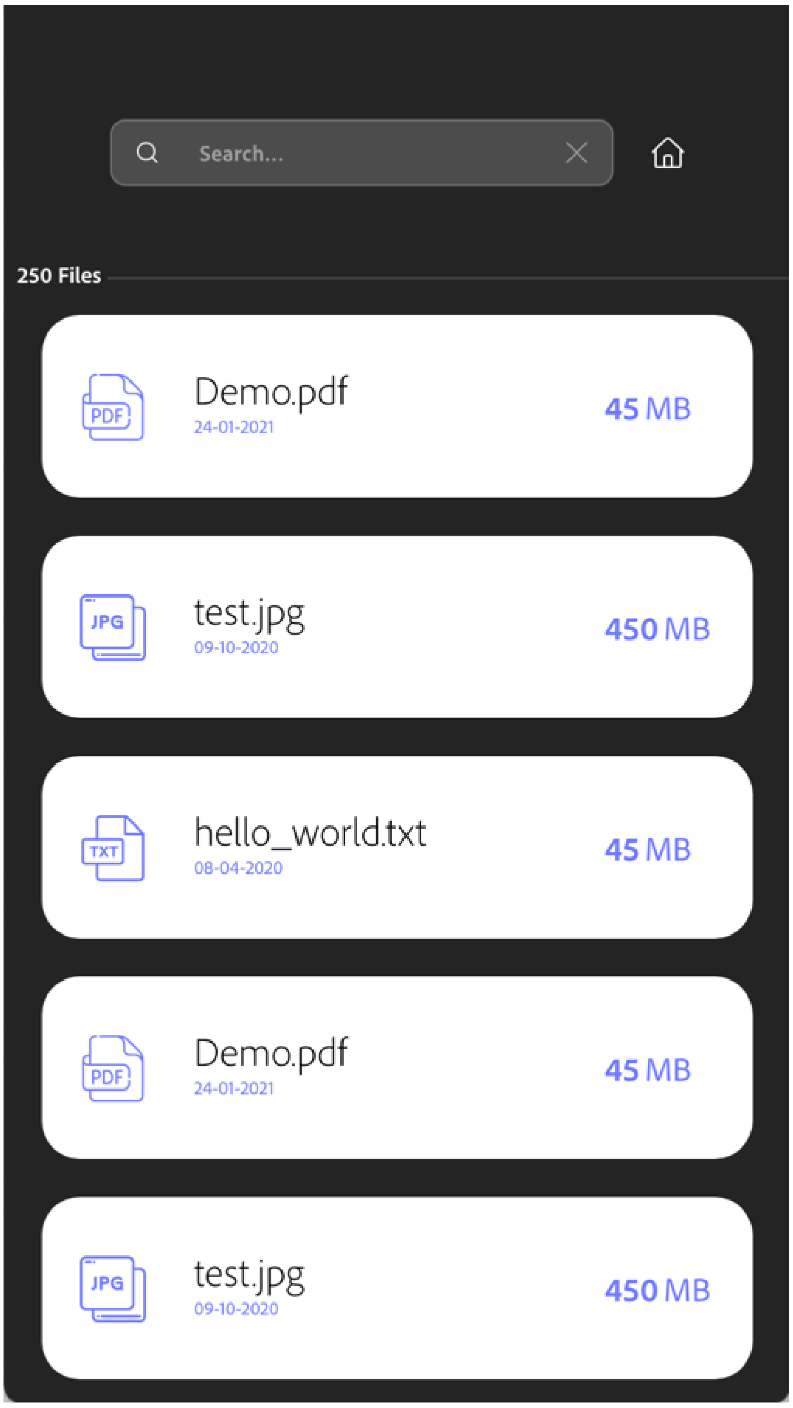
- Stores
- Dispatcher
- Views
- Action / Action Creators

The client application is the vital part of the proposed idea and need to be hosted as a service which can be easily accessed by user across the globe. The client application itself is divided into two sub modules first is web client and another is android client. Links to both of the projects can be found below. In both the client application we researched the recent trends and tried to implement them in the UI. Both the application is based on material UI and uses latest dependencies.

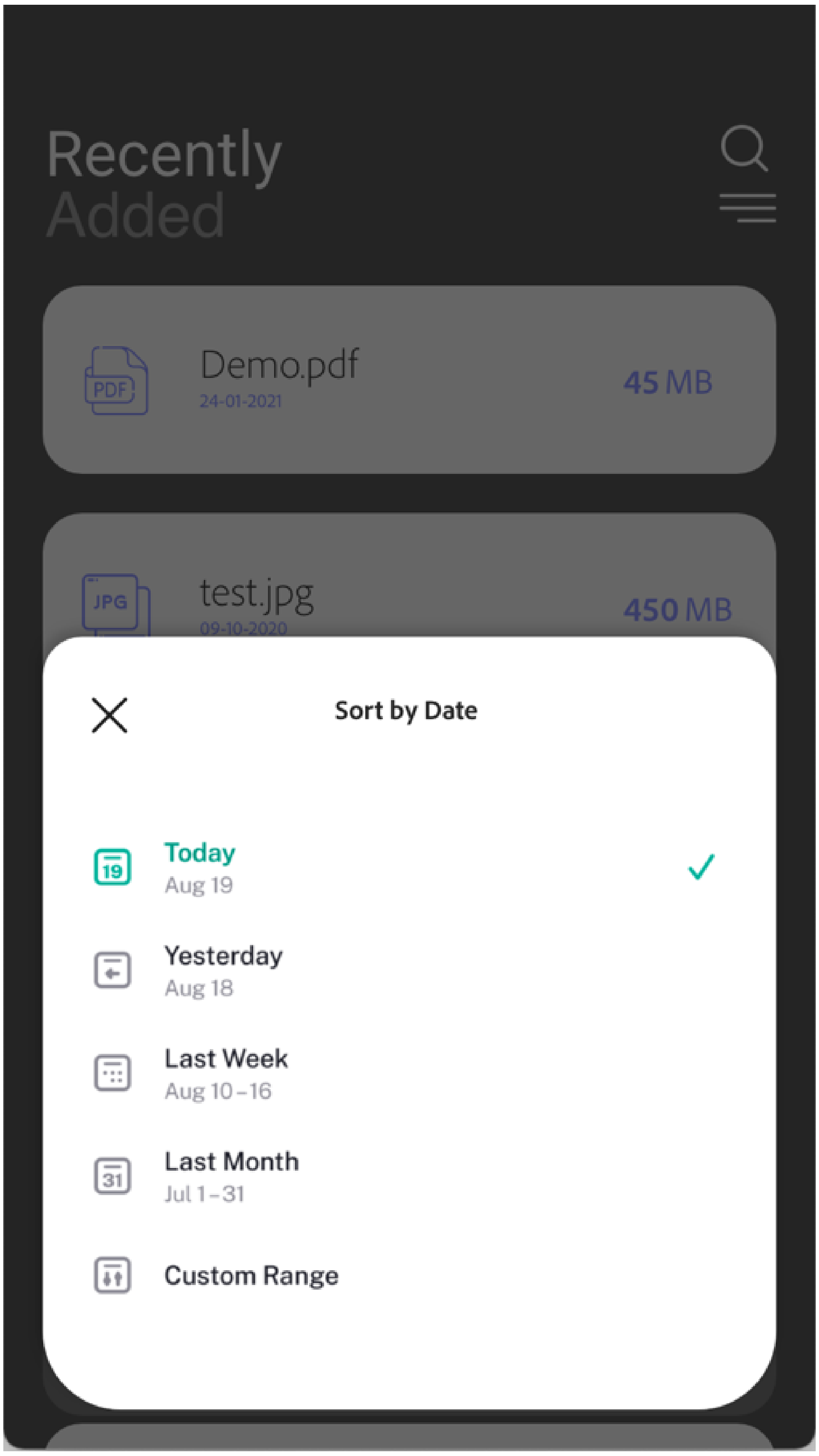
https://github.com/code-gambit/VT-AndroidClient
https://github.com/code-gambit/VT-lambdas

Screenshot

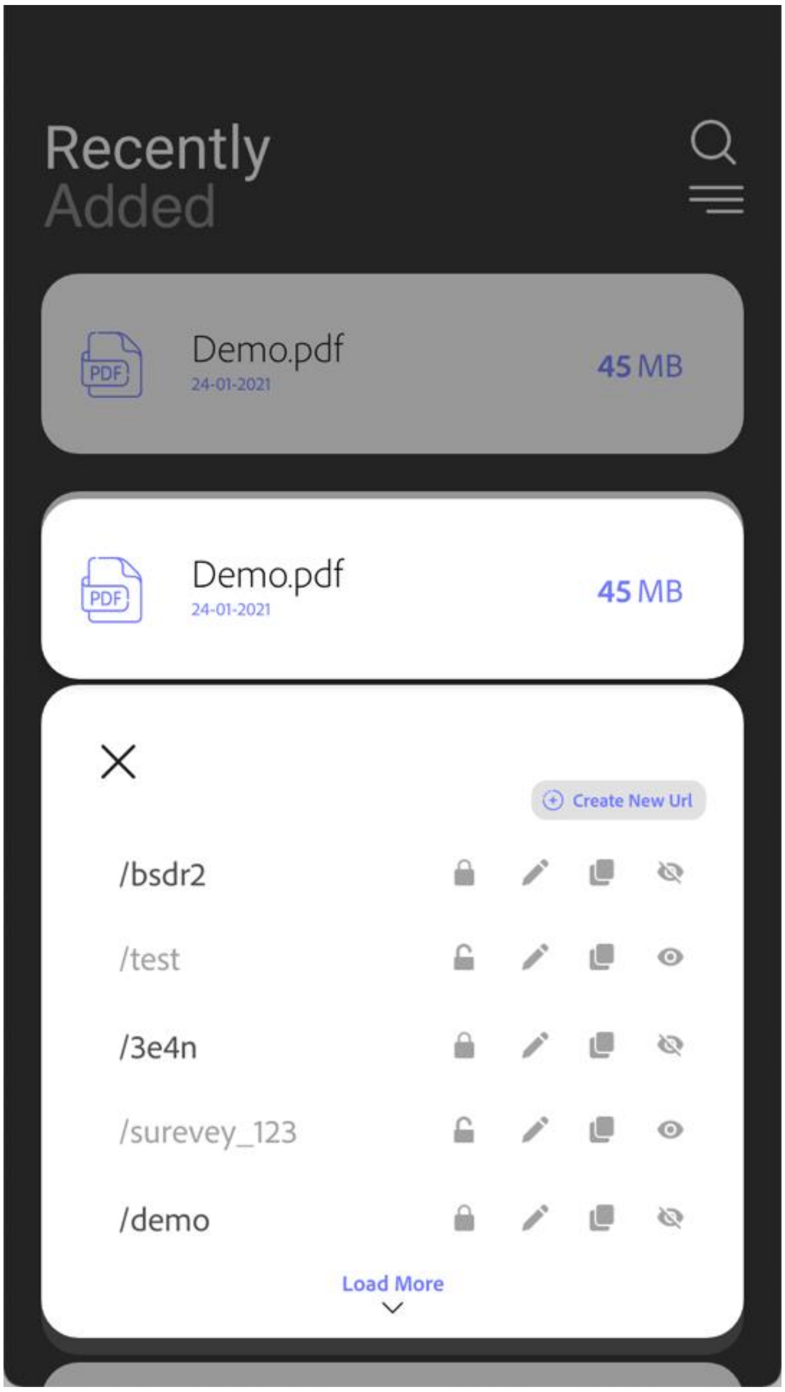
Also below are some of the screenshots of the application.



This screenshot shows the option for search. Using this feature user can search for the specific file name. Our search implementation matches the start of the text rather than in between the text. But for searching user has to enter at least 3 alphabets. User can also type phrases or multiple words for searching.

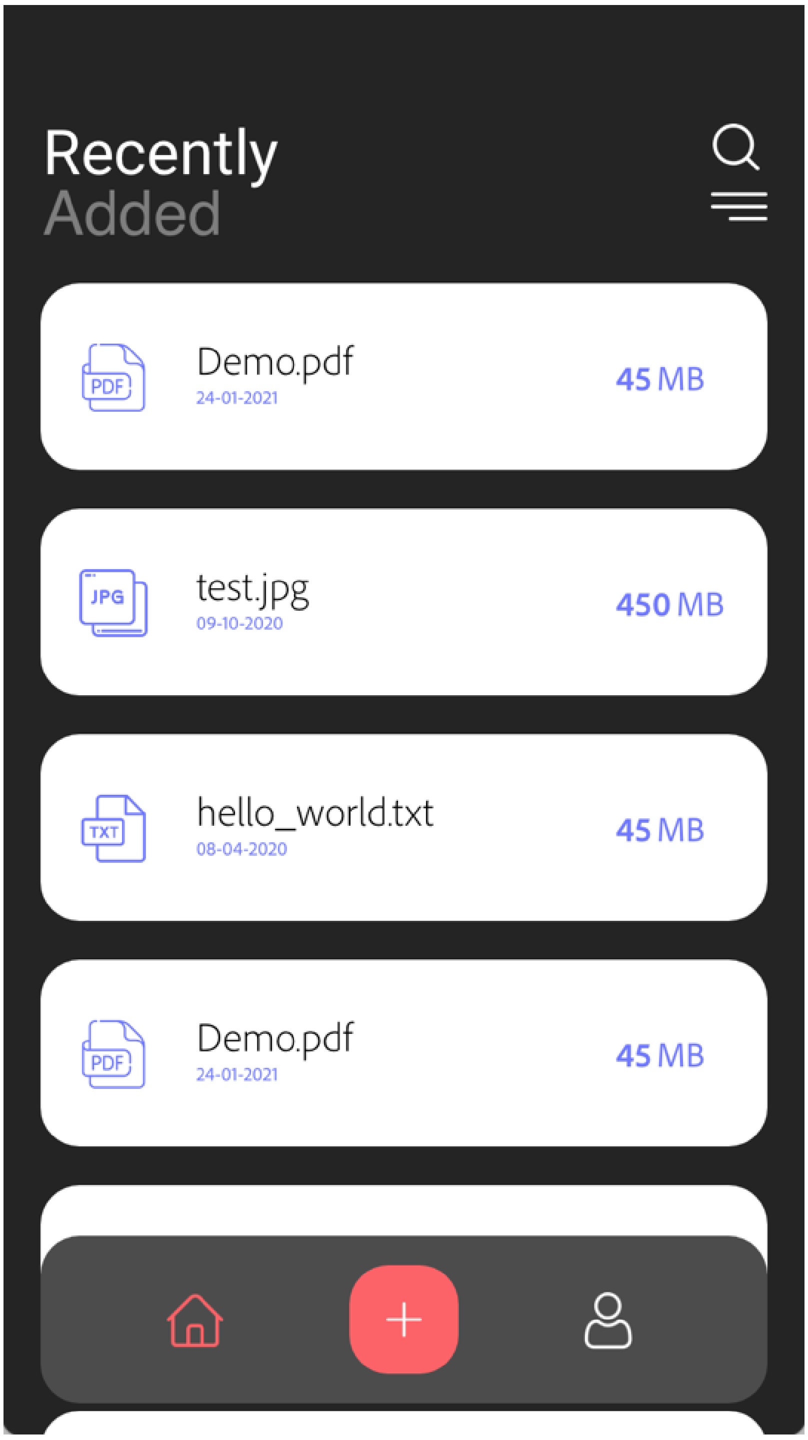


This screenshot shows the date filter option. We have provided the user with full customizable time series filter where user can filter the files based on the date range. When user clicks on the filter icon on main page the bottom sheet open ups with the date filters like, today, yesterday, last week, last month and custom range. All the specific filters are generated in runtime, whereas Custom Range provides user with a input field to enter start and end date.



This is the detail view of single file with a collapsible bottom sheet. The bottom list shows all the URLs corresponding to the current file. Along with that it also has a different quick toggle for controlling the URLs accessibility and visibility, these toggles provide features like edit click count, make URL hidden/un-hidden, delete URL etc.

Apart from the URL related it also provide a quick information about the file like file name, file size, date uploaded etc.



Recently Added

Demo.pdf 24-01-2021 45 MB

test.jpg 09-10-2020 450 MB

hello_world.txt 08-04-2020 45 MB

Demo.pdf 24-01-2021 45 MB

This is the home page of the Android application, which is responsible for showing the recent uploaded files for the user to quickly access it. Along with that bottom bar with minimalistic design provide user's the scope to navigate between different parts of the application. The plus icon in the center of the navigation bar is the main highlight of the page as it is responsible for uploading different types of files.

Results and Conclusion

In this paper, a blockchain-based paper review system is proposed to solve the problems of current peer-review systems. The proposed framework provides a decentralized solution addressing file-sharing systems, and the lowest level of URL customization while sharing over the network. The paper also illustrates AWS system architecture and BPMN(Business Process And Model Notation) diagram for user authentication and file sharing workflow. Furthermore, In this paper, we have attached all the necessary points and information or we can say the workflow of the system. We have also explained the need and goods of the system. So that every person can understand it easily. File sharing is the essential demand of today's internet-driven world and in this paper, we have comprehensively discussed a different aspect of the file-sharing system along with the system proposal which is efficient enough to fulfill users' needs along with security and user-friendliness.

Reference

[1]
https://www.researchgate.net/publication/335232056_A_Decentralized_File_Sharing_Framework_for_Sensitive_Data
Accessed on: 3rd Dec 2021

[2] https://cointelegraph.com/explained/decentralized-file-sharing-explained
Accessed on: 3rd Dec 2021

[3] https://medium.com/hackernoon/research-on-decentralized-file-storage-and-sharing-on-the-blockchain-f3a224c4c85b
Accessed on: 3rd Dec 2021

[4] https://www.researchgate.net/publication/336372929_A_Blockchain-based_File-sharing_System_for_Academic_Paper_Review
Accessed on: 3rd Dec 2021

[5] " International Journal of Advanced Research", Volume 3, March 2015.

[6] " Ian Somerville , Software Engineering ", Third Edition, Pearson Education.

[7] " Object-Oriented System Development ", Third Edition, Tata McGraw Hill Edition.

[8] "Journal of network" , volume7 , no.10, October 2012.

[9]. https://medium.com/hackernoon/research-on-decentralized-file-storage-and-sharing-on-the-blockchain-f3a224c4c85b
Accessed on: 3rd Dec 2021

[10] Decentralized File Storing and Sharing System using Blockchain and IPFS
Journal: International Research Journal of Engineering and Technology (IRJET)
Volume: 07 Issue: 05 | May 2020

[11]
https://www.researchgate.net/publication/330758943_Asterism_Decentralized_File_Sharing_Application_for_Mobile_Devices
Accessed on: Dec 03 2021

[12] Decentralized file sharing application in a global approach
Author: K. Mohan Krishna, M. Kranthi Kiran, P. Kiran Kumar, M. Tejasmika, K. Ravi Teja Reddy, K. Sandeep
Pub. Date19 March, 2019
Paper IDV5I2-1287
PublisherIJARIIT
EditionVolume-5, Issue-2, 2019