CSC409 A3: r/place clone

Members

zhan6104

dongshu4

Software Used

AWS CLI

AWS SAM CLI

AWS Tools

CloudFormation

We use CloudFormation to model and set up our AWS resources using yaml templates that describes all the AWS resources we want, and CloudFormation takes care of provisioning and configuring those resources.

We are also using AWS SAM to build our Serverless Application Model that automatically handles the code upload for us (in addition to auto resoucre allocation).

IAM

IAM is a core part of our stack, we use IAM to set permissions for each roles and create developer accounts for accessing resources, and also API keys for CLI applications we use (eg. aws-cli, sam-cli).

IAM rules are also defined in our CloudFormation templates to enable access rights from lambda functions to DynamoDB and API Gateway.

S3 bucket policy and CloudFront permissions are also set by our CloudFormation templates.

Lambda

We use Lambda, in conjunction with API Gateway, to manage the websocket conenctions (conenct, broadcast, disconnect), and interacts with DynamoDB for persistent data storage (connected users, update history, borad info, etc.) and dynamically query for the data we need. All requests sent from users will be forwarded to and handled by one of the three lambda functions we have.

This resouce is managed by AWS Serverless Stack.

WebSocket API (API Gateway)

API Gateway WebSocket APIs are bidirectional. A client can send messages to a service, and services can independently send messages to clients. This bidirectional behavior enables richer client/service interactions because services can push data to clients without requiring clients to make an explicit request.

We use WebSocket API to handled the websocket connections from users, and data will be forwarded to responding Lambda functions.

DynamoDB

DynamoDB is a key-value and document database that supports tables of virtually any size with horizontal scaling. DynamoDB scales to more than 10 trillion requests per day and with tables that have more than ten million read and write requests per second and petabytes of data storage.

DynamoDB Accelerator (DAX) is an in-memory cache that delivers fast read performance for your tables at scale by enabling you to use a fully managed in-memory cache.

We use DynamoDB to store all history update information, pixel colors and connection IDs.

CloudWatch

Amazon CloudWatch collects and visualizes real-time logs, metrics, and event data in automated dashboards to streamline your infrastructure and application maintenance.

All our function outputs and logs will be stored in CloudWatch and can be retrived at anytime.

S3

We use Amazon S3 for two main purposes:

1. Storing the static front-end website and serve as origin for Amazon CloudFront.
2. Holding tamplate files and source codes for serverless function, which will be used in CloudFormation and SAM Deploy.

CloudFront

We use Amazon CloudFront to securely deliver the front-end website with low latency and high transfer speeds.

CloufFront will use the S3 bucket as origin to distribute resources acros its infurstructure.

Spam Protection

This is achieved at server side as each connection will have a rate limit (1 update per 5 minutes) when sending drawing information. The frontend javascript will prevent users from sending more than 1 draw requests in 5 minutes. This also prevents bots from spamming requests to occupy our bandwidth.

Even if the user modifies our client, the lambda will not accept new update requests sent by users during their cool down time and it will not be broadcasted other users, and the user who requested the update will be notified through websocket with the wait time.

Broadcast Updates

This is handled using AWS Lambda in part with API Gateway to broadcast the information to all clients still connected.

When our lambda received a legitamate draw update, it will scan the dynamodb table, where we maintain to keep track of all active clients, to find all currently connected clients and broadcast the update.

An entry will be inserted to the connections table when a new connection opens. If we find a stale connection or a connection closes, the table of connections will be updated to reflect the change. This table is useful for our serverless design because we can keep track of the current connections.

Scalability

Since our application is cloud-native (all achieved using AWS lambda and API Gateway), the scalability aspect of out service is completely handled by AWS.

Our applications shines in the aspect of scalability as it can ultilize AWS's global infrastructure and its potential to dynamically scale up or down our services. Our system's performance will stay the same no matter how many users or concurrent requests we have as reponse time won't slow down as our database grows thanks to highly scalable Websocket API and DynamoDB's single-digit millisecond performance, nearly unlimited throughput and storage, built-in memory cache (DAX), and automatic multi-Region replication, as well as lambda function's automatic scaling, multi-AZ and load balancing.

Vertical

We can simply increase the memory limit of our function as a cloud formation setting when deploying the stack, and AWS will also allocates CPU power proportional to the amount of memory provisioned (up to 10 GB of memory and 6 vCPU cores).

The same thing goes to our dynamoDB and the front-end static site.

DynamoDB is a key-value and document database that supports tables of virtually any size with horizontal scaling. DynamoDB scales to more than 10 trillion requests per day and with tables that have more than ten million read and write requests per second and petabytes of data storage.

Amazon S3 was designed from the ground up to handle traffic for any internet application. It can be dynamically scaled according to demand and will be billed on a pay-per-use model.

Horizontal

As traffic increases, Lambda increases the number of concurrent executions of our functions. When a function is first invoked, the Lambda service creates an instance of the function and runs the handler method to process the event. After completion, the function remains available for a period of time to process subsequent events. If other events arrive while the function is busy, Lambda creates more instances of the function to handle these requests concurrently.

For an initial burst of traffic, the cumulative concurrency in a Region can reach between 500 and 3000 instances per minute, depending upon the Region. After this initial burst, functions can scale by an additional 500 instances per minute.

Current soft limit: 1000 concurrent executions (set by AWS, can increase upon request).

The same thing goes to our dynamoDB and the front-end static site.

DynamoDB instantly accommodates the workloads as they ramp up or down to any previously reached traffic level. If a workload's traffic level hits a new peak, DynamoDB adapts rapidly to accommodate the workload.

Our Amazon S3 bucket is already been served from multiple AZ, and can be configured to use Amazon S3 Replication which enables automatic, asynchronous copying of objects across Amazon S3 buckets (that can be in other region).

Availability

AWS Lambda runs our function in multiple Availability Zones to ensure that it is available to process events in case of a service interruption in a single zone.

Our front-end website also run on AWS infurstructre (S3 and CloudFront). Our S3 bucket objects (source codes) are automatically stored across multiple devices spanning a minimum of three Availability Zones, each separated by miles across an AWS Region. S3 Standard storage class is designed for 99.99% availability.

We also use CloudFront to accelerate the requests from users.

In addition to the support of AWS global infrastructure, Amazon CloudFront offers an *origin failover* feature to help support our data resiliency needs. CloudFront is a global service that delivers contents through a worldwide network of data centers called *edge locations* or *points of presence* (POPs). If contents are not already cached in an edge location, CloudFront retrieves it from an origin (S3 in this case) for the definitive version of the content.

Security

Our application is cloud-native, and all resources are accessed through AWS SDK with proper roles/permisions set (no VPC/subnet/EC2 involved). Our database is only accessible from our developer accounts and Lambda functions. The permission are precisely set as each Lambda function will only have access to the table they need (not the entire db). We only have a websocket (which is handled by API Gateway) exposed to the public, and invalid requests will be rejected either by API Gateway or our Lambda functions.

Our S3 buckets are also set to be only accessible by the cloudfront user using IAM role.

Since we have no VPC nor subnets, we do not need to worry about the inbound rule, and the same also applys to outbound rules.

Design Decisions

We designed our application to be cloud-native to take full advantage of AWS's insfurstrcutes. All codes are deployed serverless-ly to reduce the hassle of scaling and mataining the servers and databases.

Our approach also reduce the cost of running the application. All AWS services we used can be set to use pay-per-use billing model, which means our system won't cost much and takes less resources (if not none for most cases) if the userbase is small, and can also automatically scale up without any intervention while requests start to grow.

To achieve serverless, we have chosen to use Lambda+API Gateway to handle websockets and S3+CloudFront to handle static contents.

We chose to use DynamoDB for its fast response time, and it's also secure because IAM access is required to interact with DB; the permission can be set on a per-table basis. It is also easy to access DB given the Amazon SDK is powerful and easy to use.

We choose to use DynamoDB because it is build with DAX in-memory cache that delivers fast read performance our tables at scale by enabling us to use a fully managed in-memory cache. This also means that we do not need to use a Redis cluster for data caching.

Weakness and Strength

Weakness

- Unencrypted websocket messages

  Our system uses plain text websocket to communicate between servers to clients, which can be subject to MITM attacks.

  To midigate this, we can improve by using end-to-end encryption when sending messages using websocket.

- Lack of user identification

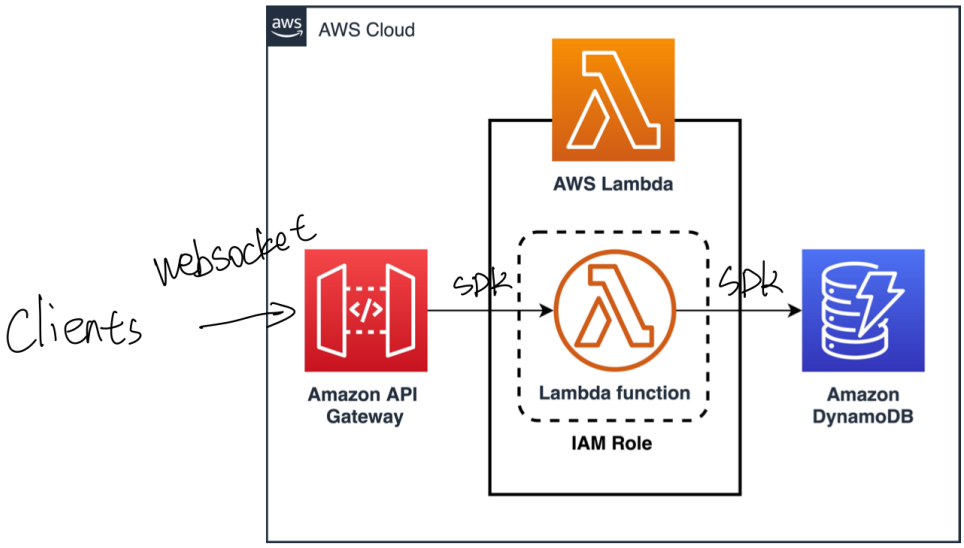
  Our system is subject to DoS attacks as we do not have a user system reddit has. We are currently using connection IDs to identify each unique user and that's not a reliable indicator when handling spam messages.

Strength

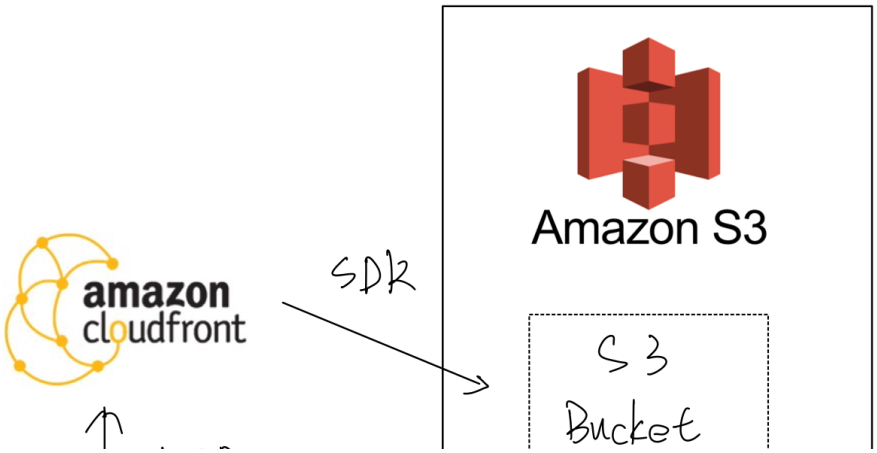
- Low latency & Low reponse time
- High Scalability & Availability (Performance at any scale)
- Highly secure, only necessary resource exposed

Diagrams

Serverless Lambda (websockets)



frontend Static website:

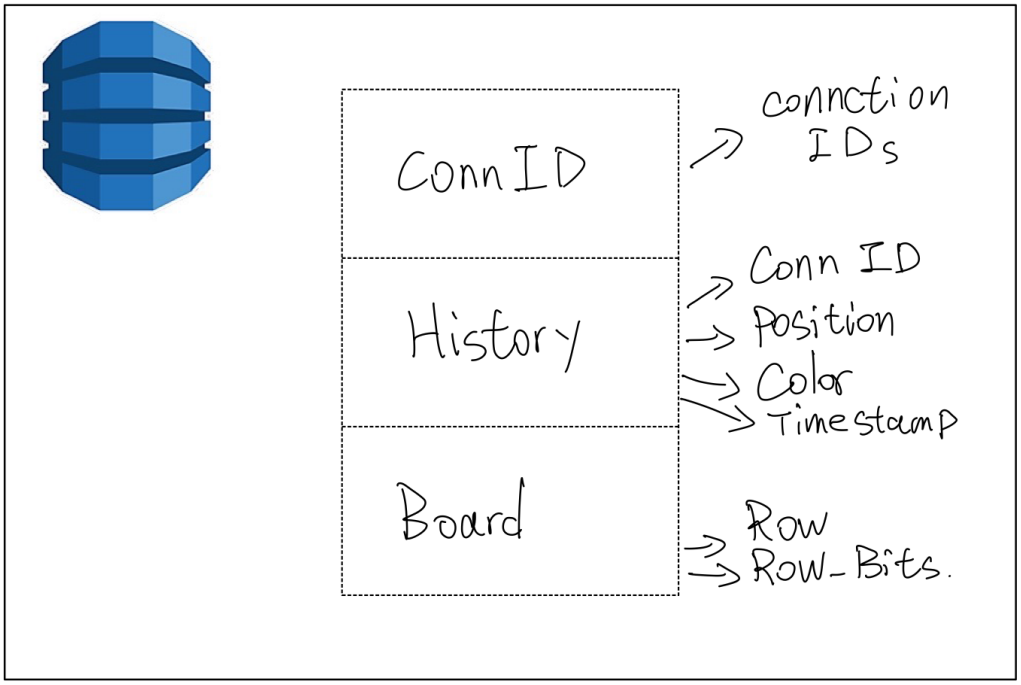


HTTP

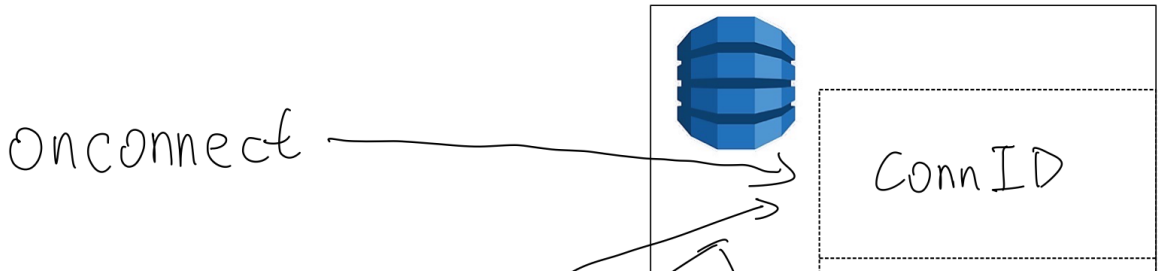
Users

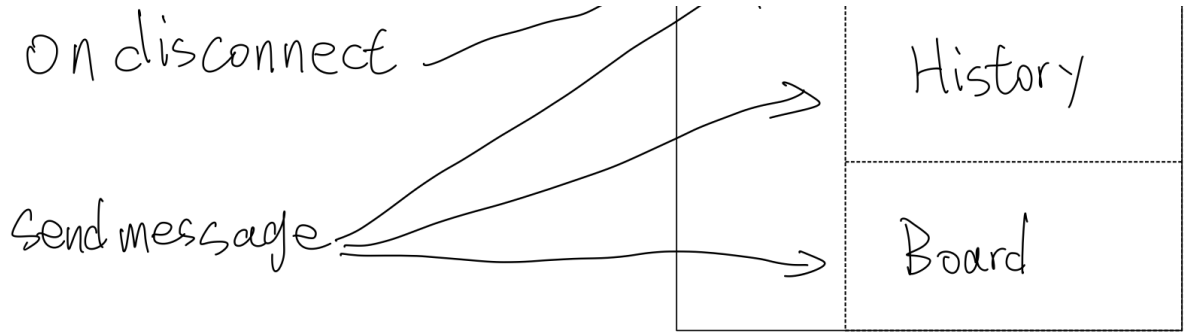
IAM Role

DynamoDB:



Lambda function access:





Dataflow :

User

↕ HTTP

CloudFront

↑↓ Amazon SDK

S3 Bucket

Web client

↓↑ websocket

API Gateway

↓↑ Amazon SDK

Lambda Functions

↓↑ Amazon SDK

DynamoDB

References

AWS Documentation

Acknowledgement

This application is forked and modified from simple-websockets-chat-app