Ricardo Vano
ricardo.vano@gmail.com
https://www.linkedin.com/in/ricardovano

Engineering Manager - System Design Exercise

Functional Requirements

- Get authorization from users to access their banking data;
- Get banking data from Bank API;
- Provide credit score to partners;
- Store the anonymized data in a database for future analytical purposes.

Non-Functional Requirements

- Satisfy a 10 credit score request/second;
- Get prepared to scale to 100 credit score requests/second.

Question: For how long a Credit Score is valid?
Considering the Lending Company could request the same Credit Score in a period of time, I included a cache server for this data.

1. System design

For this project, I used a serverless architecture, to keep the team small and ship faster, at the same time provide scalability as we need.

Technologies

Identity Server: Amazon Cognito. Fully managed service, that scales to hundreds of millions of users, using Oauth2 workflow;

API Gateway: Amazon API Gateway, which scales and implements Load Balance, Security and Monitoring;

Backend: Python, running in containers in AWS Fargate. Scale as needed. Cheaper than AWS Lambda and easier than EC2;

Frontend: Django running in AWS Fargate to scale as needed. Amazon CloudFront to store static content and provide a faster experience to the end user;

Database: Amazon RDS running MySQL or PostgreSQL instances. A managed database could scale as we need. SQL can guarantee ACID compliance and it is easier for Analytics rather than NoSQL.

Message Broker: Kafka. With High-throughput and fault tolerance, it could be a better option to RabbitMQ.

Cache: Memcached considering the Credit Score has a simple structure and expires in a few minutes/hours. If needed a more rich data structure, could be considered Redis as an alternative.

Protocol: HTTP instead of gRCP. There is no performance gain using gRPC because of the delay of the Credit Score service and Bank API. Beyond that, HTTP offers simplicity, compatibility and a widespread adoption.

Synchronous requests: Server-Sent Events (SSE). Considering the high request volume of 100 requests per second, SSE is more efficient than Long Pooling, because there is no need to open and close connections, reducing overhead on the server. Although SSE provides a higher level of abstraction compared to raw sockets, with more scalability, proxy support and compatibility.
To implement SSE, it would be more appropriate to have separate endpoints for Server-Sent Events (SSE) and asynchronous requests.

Diagram (System Design v1.2.png)

*Attached diagram in System Design v1.2.png file

Detailed description of the diagram

Steps 1, 2 and 3: Lending Company initiates authorization, redirecting the user to ACME's authorization endpoint, passing the necessary parameters in the URL query string, such as:

- **client_id**: The client ID of the Lending Company application;
- **redirect_uri**: The URI where ACME will redirect the user after authorization;
- **scope**: The requested scope of access to the banking data;
- **response_type**: Set to "code" to indicate the authorization code flow.

Steps 4, 5 and 6: ACME presents the user with a login screen to authenticate themselves with their banking credentials. After successful authentication, ACME displays a consent screen to explain the requested permissions and scope of access to the user's banking data. The user reviews the details and grants authorization to the Lending Company.

Steps 7 and 8: ACME generates an authorization code from the identity server and redirects the user back to the Lending Company's specified redirect_uri, appending an authorization code as a query parameter. For example:
https://lendingcompany.com/callback?code=AUTHORIZATION_CODE.

Step 9: The Lending Company exchanges the authorization code sent by ACME to the redirect_uri for an access token, making a POST request to ACME's token endpoint, providing the necessary parameters in the request body:

- **grant_type**: Set to "authorization_code" to indicate the exchange of an authorization code;
- **code**: The authorization code received from ACME;
- **client_id**: The client ID of the Lending Company application;
- **client_secret**: The client secret of the Lending Company application;

- **redirect_uri**: The same redirect URI used in the initial authorization request.

Step 10: ACME validates the authorization code. If the validation is successful, ACME generates an access token for the Lending Company.

Steps 11, 12 and 13: The Lending Company includes the obtained access token in API requests to ACME's banking data endpoint. ACME validates the access token for each request to ensure authentication and authorization.

Steps 14: ACME checks the permissions associated with the Lending Company.

Step 15: ACME sends a permission ID that allows Lending Company to request a Credit Score of a specific user.

Step 16: Using the permission ID, Lending Company asks for a Credit Score.

Steps 17 and 18: The ACME service will validate if this specific Credit Score is still in cache. If so, get data from cache.

Step 19 e 20: ACME validates permission, credentials and scope, then retrieves data from Bank API using stored credentials.

Step 21: ACME sends the data to a message broker.

Step 22: Anonymization service gets data from Message Broker when available, anonymizes and sends to SQL database.

Step 23: Generate Credit Score Service gets data from Message Broker when available and sends to cache, to be available for Lending Company.

Step 24: ACME will send Credit Score. If the Lending Company chose the synchronous approach, ACME can respond directly with the result in the response. If the Lending Company opted for the asynchronous approach, ACME can send the results to the Lending Company's provided webhook URL.

2. Scale from 10 credit score requests/second to 100 request/second

Using fully managed services on AWs, scales it is easy and fast:

Services in AWS Fargate: Create an Auto Scaling Group. An ASG enables automatic scaling based on predefined scaling policies.

Kafka: Use Amazon EC2 Auto Scaling to automatically add or remove broker instances based on predefined scaling policies.

Database:

- Vertical scaling: Increasing the instance size (CPU and memory). There is a minimum down time.
- Horizontal scaling: Multi-AZ Deployment provides automatic failover in case of a primary instance failure.
- Sharding: Partitioning the data across multiple RDS instances.
- Storage: Vertical scaling. Amazon RDS supports automatic storage scaling.

API Gateway: AWS API Gateway automatically scales horizontally to handle increased traffic. It dynamically provisions additional instances of the API Gateway service as needed.

3. Milestones

Milestone 1: Project Setup and Infrastructure

- Set up the development environment, including necessary tools and frameworks;
- Configure AWS infrastructure, including VPC, subnets, security groups, and IAM roles;
- Create an Amazon RDS instance for the database;
- Set up the necessary networking components, such as API Gateway and load balancers.

Milestone 2: User Authorization and Data Retrieval

- Implement the authentication and authorization flow using OAuth2;
- Integrate with the Identity Server to handle user login and consent;
- Develop endpoints to receive valid access tokens from partners/customers;
- Implement the logic to verify permissions and retrieve banking data using the Bank API;
- Store the retrieved data securely in the database;
- Integrate Server-Sent Events (SSE) for synchronous responses.

Milestone 3: Credit Score Calculation

- Design and implement the credit score calculation algorithm;
- Develop the necessary APIs to initiate the credit score calculation process;
- Implement the synchronous and asynchronous options for credit score requests;
- Generate the credit score and store it in the database.

Milestone 4: Integration with Kafka and Anonymization Service

- Set up Kafka clusters in AWS to handle data streaming and messaging;
- Implement message producers to send data retrieved from the Bank API to Kafka topics;
- Develop consumer services to process Kafka messages and feed data to the anonymization service;
- Integrate with the anonymization service to anonymize sensitive data securely;
- Store the anonymized data in the database for analytical purposes.

Milestone 5: Deployment and Scaling

- Create deployment scripts and automation to deploy the system to AWS Fargate.
- Configure auto-scaling policies for Fargate tasks based on workload demands.
- Set up monitoring and alerting using Amazon CloudWatch for system metrics and performance.
- Conduct comprehensive testing and optimization for scaling the system to handle increased load.

- Document the deployment process and provide guidelines for future scaling and maintenance.

4. Team

Fullstack Engineers: 3 (Python and Django)
SRE Engineer: 1 (Cloud AWS Engineer)