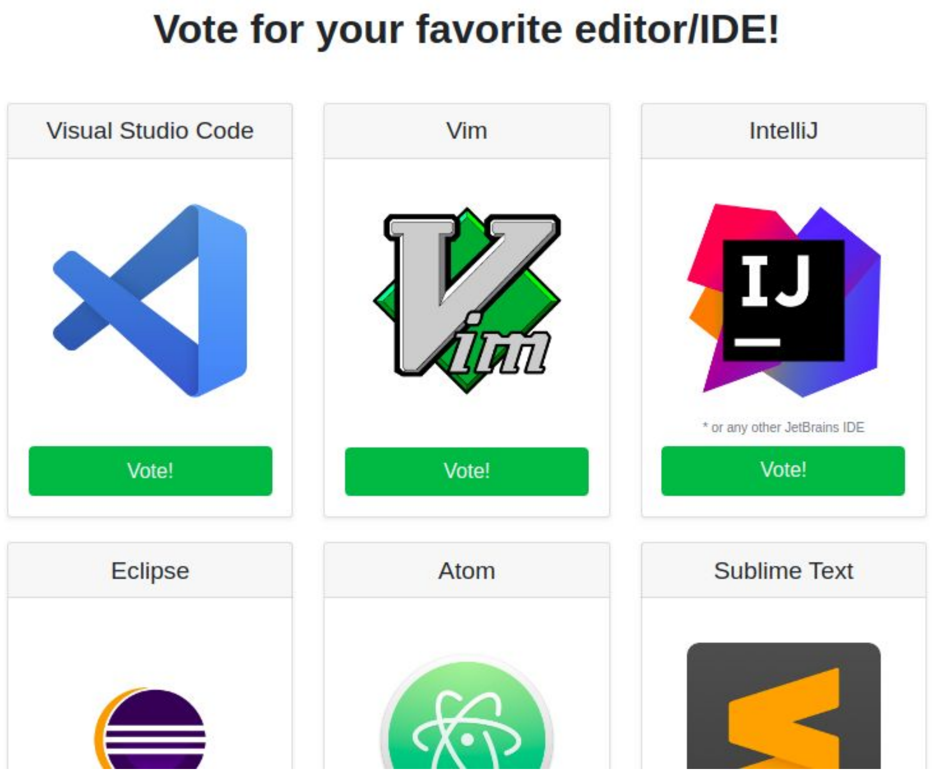
Decoupled SQS Application

Architecting a decoupled application on AWS



The application

The goal was to create an application where users could vote for their favorite text editor or programming IDE.

The entire infrastructure was written as a code, using CloudFormation.

This project was created as part of the **ProjectGallery Challenge** from Bertelsmann Udacity Cloud Scholarship

The architecture

The architecture was planned to be a **cost-efficient and scalable** solution for the voting app, while decoupling the front-end actions and the back-end database storing.

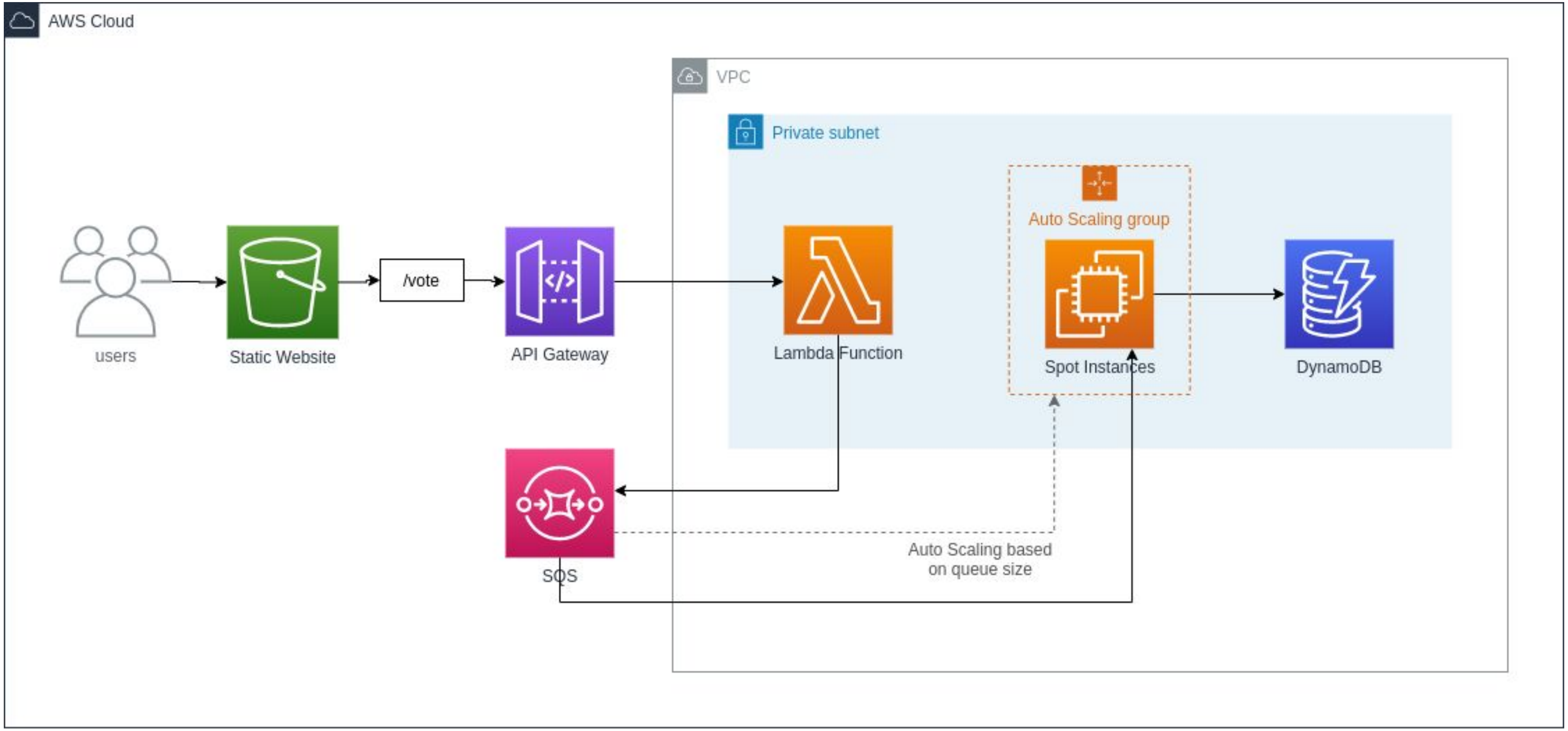
In this example the benefits may not be very clear, but if your back-end does some slow processing like fraud checking before finishing an online order, it’s a good practice to decouple it from the front-end to give your users a quick response.

The architecture

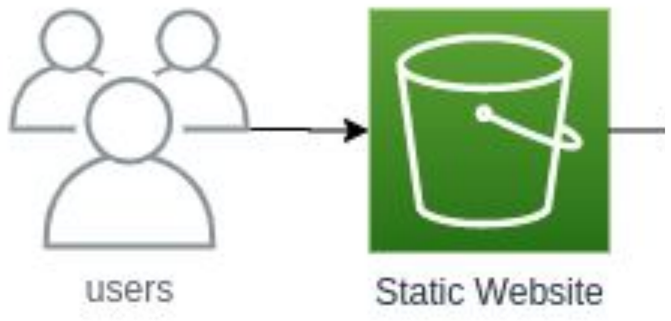
The project's repo on GitHub is:

https://github.com/GusAntoniassi/decoupled-sqs-application

The architecture



The front-end



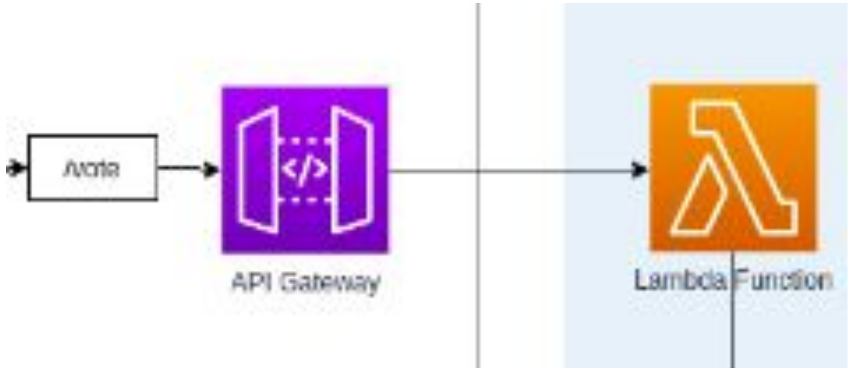
➔ **Written in HTML + CSS + JS**
Used Bootstrap for styling, and jQuery for easier AJAX requests..

➔ **Hosted on Amazon S3**
HTML is served to the users from an S3 bucket configured as a static website.

➔ **API integration**
Calls API Gateway via AJAX

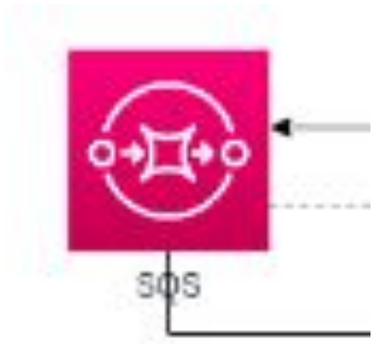
➔ **Source code available at**
https://github.com/GusAntoniassi/decoupled-sqs-application-static-website

The API back-end



- ➔ **API Gateway**
  Configured to accept a POST request on the /vote endpoint
- ➔ **CORS**
  API Gateway includes CORS headers for the front-end AJAX integration work
- ➔ **Lambda Function**
  Written in Python, sends a message to SQS with the IDE the user voted on
- ➔ **Source code available at**
  https://github.com/GusAntoniassi/decoupled-sqs-application-voting-api-lambda

The SQS queue

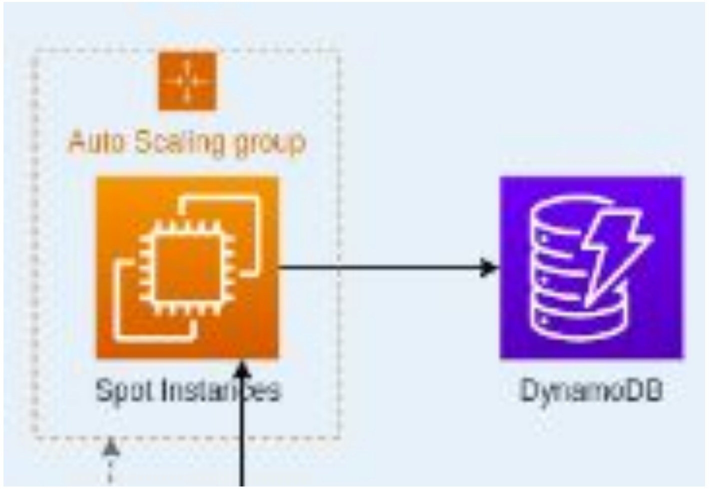


➔ **SQS Standard Queue**
Since we don't care about the order of the votes, a FIFO queue wasn't needed

➔ **Receives messages**
From the front-end, with the ID of the application the user voted on

➔ **Messages are read**
By the Python code on the EC2 spot instances that run on Auto-Scaling

The processing back-end



- ➔ **EC2 Spot Instances**
  For cost-efficiency, the application uses a fleet of EC2 t3.nano spot instances
- ➔ **Processing code**
  Reads the SQS queue and stores the vote on AWS DynamoDB.
- ➔ **Source code available at**
  https://github.com/GusAntoniassi/decoupled-sqs-application-processing-code

1. Course concepts

The following concepts from the course were used in this project:

- ➔ **Cloud Computing**
- ➔ **Compute Service (EC2 & Lambda)**
- ➔ **Storage (S3, DynamoDB**
- ➔ **Elasticity (EC2 Auto Scaling)**
- ➔ **Messaging (SQS)**
- ➔ **Infrastructure as Code (CloudFormation)**

2. Lessons learned

Some of the lessons learned during the development of this project:

- ➔ **CloudFormation**
  I learned a lot about how CloudFormation works, its functions and way of working.
- ➔ **Limitations**
  CloudFormation has a couple of limitations, like: you can't upload files to S3 buckets, and you can't use Lambda code from Git. I had to use a lot of shell scripting to deploy the entire application with a single command.

2. Lessons learned

Some of the lessons learned during the development of this project:

➔ **SQS**

While I had already worked with creating and managing SQS, I had no experience with the development part, like sending and receiving messages.

➔ **AWS Go SDK documentation**

Initially my idea was to do the lambda and processing code in Go, but the AWS SDK documentation I found wasn't very good in my opinion. Python's Boto3 docs were way better, with example and clear explanations.

3. Improvements

Due to time constraints, there are a lot of improvement points for this project:

- **Improve documentation**
  Add README for all the repos, document the shell script better.
- **Deploy Docker image**
  I think the deploy script will only work on Linux (and maybe Mac). A good way to solve this would be to use a Docker image to wrap the deployment script.

3. Improvements

Due to time constraints, there are a lot of improvement points for this project:

➔ **Smarter queue scaling**

Currently the Spot fleet will scale up if there are 10 messages in the queue, and scale down if there are 0. But doing it this way we can't have 0 instances running if the queue is empty.

My idea is to create custom CloudWatch metrics with the number of instances and the messages in queue, to allow for better calculations and scaling alarms.

Thanks for reading!

CloudFormation repo URL:
https://github.com/GusAntoniassi/decoupled-sqs-application

If you're a Bertelsmann Scholarship student, I'm @GusAntoniassi on Slack!