Vendr Backend Tech Spec

Overview:

Vendr is a product browsing application that allows users to have a store browsing experience digitally. It incorporates standard features such as user and account creation and “messenger” type communication. The standout aspect is the product recommendation that showcases other users’ created products based on location and preferences in a “tinder-like” fashion. Users can then utilize the messaging feature of the app to discuss the selling of said item and come to an agreement of the cost and buying location. The app should also provide opportunities for users to link their socials to legitimize themselves as sellers and leave reviews on each other for a better selling and buying experience.

What can the user do and app features:

1. Browse location based and tag based products that have 1-5 images per product, descriptions and price
2. Submit a new product
3. Save an item for future conversation with the seller
4. Have an account, profile images, names and credentials
5. Have a business account
6. Have reviews attached to them by other users and review other users
7. Link their account with another social media
8. Add specific tags to narrow the given products
9. Message other users when offering to buy a product
10. Receive messages from users when someone wants to buy their product and message back
11. Email and Phone verification
12. Hype calculations for product’s promotion
13. Item deletion after a certain period of time
14. Paying for product promotion (increasing item hype)

The complexity of the app is high, thus we will need to make compromises and b-line for specific features if we want to have a functional app off of which we can build our other features.

Green is for “must have”, yellow is “nice to have”, and red is for features that can come after the initial creation of the app. With the amount of people working on the backend, there is no way we can have 5 & 6 in by April 15th.
Thus the services and the dynamo diagrams and breakdown won’t hold information on those particular features until we come around to working on them (talking about **socials connection** and **user reviews**)

We must first prioritize the creation of an account, then the creation of new products on the platform + the browsing experience of said products and only then focus on messaging.

If we had more engineers we could work on each of those in parallel, but due to the size of the team it can only be done one by one.

The app will be built on top of the AWS cloud provider as it is most stable, relatively cheap under the free tier and provides the infrastructure that we need for the creation of the Vendr app. The products we will use to maintain our application:

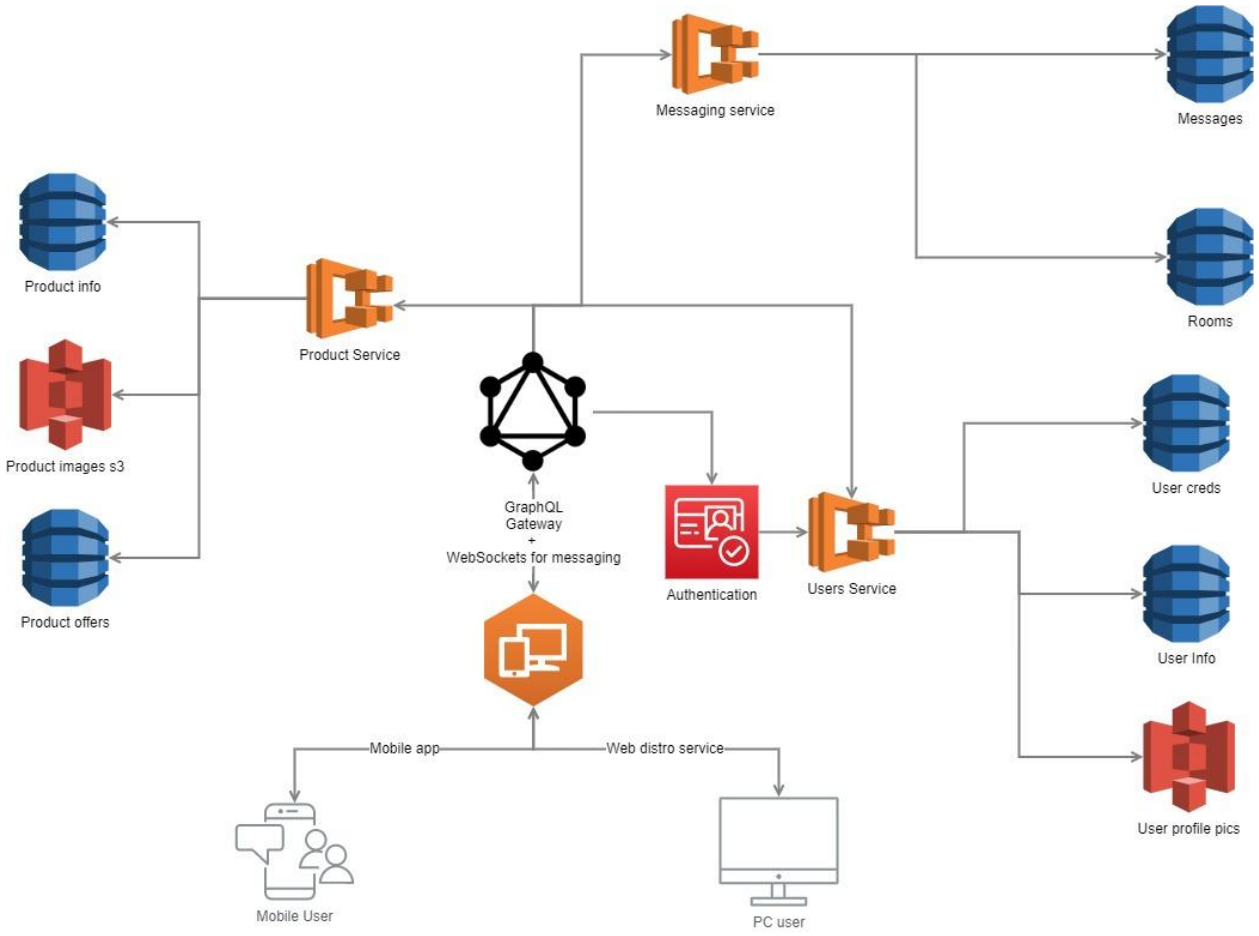
- ECS + Fargate for deploying our microservices
- Cloudwatch for backend monitoring and logging
- DynamoDB for databases
- S3 buckets for image file storage
- VPC + route53 for networking and routing

We will be using the copilot cli to abstract the creation of certain resources (VPC and Networking in particular) as we do not have the time or team size to make our own cli for this. But it uses the cloudformation aws service meaning we will remain in the aws ecosystem.

For our applications gateway we will be using GraphQL (the gophers version) as we are targeting both mobile and web. GraphQL also provides a good mutation and query abstraction layer and allows you to request only the data that you need, making the requests coming to our services far less costly and efficient.

All of the micro services will be built on top of Golang and utilize Twirp for handling schema generation, RPCs and client gen.

Diagram:



Breakdown:

Hosting

Before we begin working on our API, we need to first create a hosting platform to distribute our service. As part of that work, anytime someone makes a merge into master we should build the application and deploy it to the hosting zone.

This involves creating a pipeline and an ECS service that will be distributing our application.

GraphQL gateway

Schema:
Auth Token will be required to perform certain operations

Mutations:
Login()
Signup()
UpdateProfile()
CreateProduct()
UpdateProduct()
DeleteProduct()
SendMessage()
CreateRoom()

Queries:
GetUserInfo()
GetProduct()
ListProducts()
ListMessages()
ListRooms()

WebSockets + PubSub

The majority of the work will be handled through GQL subscriptions.

The subscription form looks like this for chat rooms:

room.%room-id%

Since the room ids will be unique, that's all we require for updating the web sockets for connected users in the room. If a user is not within the room and not connected to web sockets, we will use the *messages service* to store the message and they will be able to see new messages once back in the room and list unread messages.

Users Service

==========

RPC Schema:

Login(password, userdata) AuthKey
Signup(password, userdata) AuthKey
GetUser(user-id) User
GetUsers(user-ids) Users // TBD, doesn't seem to be important right now
//Change UpdateProfile(user-id, password, userdata) User
Userdata - {
    First/Last name
    Email
    Password
    Phone #
    Birthdate
    Bio
    Based Address [AdressLine1+ AdressLine2 + City + State + Zipcode] (long, late)
    Profile img (array)
    isVerifiedUser (boolean)
}
Shopdata - {
    Shop name
    First/Last name
    Email
    Password
    Phone #
    Birthdate
    Bio
    Based Address [AdressLine1+ AdressLine2 + City + State + Zipcode] (long, late)
    Profile img array
    isVerifiedUser (boolean)
    isVerifiedShop (boolean)
}
==========

User Creds DB Schema:

User-id string (hash key)
Pwd-hash string
Email-hash string
Phone-hash string

==========

User Info DB Schema:

User-id string (hash key)
BasedAddress-hash string
CurrentLocation string
Birthday string
Bio string
Shopname string - Shopdata ONLY
ShowPhone boolean
ShowAddressLines boolean
IsEmailVerified boolean
IsPhoneVerified boolean
IsVerifiedUser boolean
IsVerifiedShop boolean - Shopdata ONLY

Socials json // TBD. This is not going to exist during the first pass of the product

==========

User Image storage:

User-id string (hash key)
Img array associated with above

==========

Product Service

==========

RPC Schema:

GetProduct(product-id) Product
ListProducts(user-id, filterBy) Products
CreateProduct(product-data) Product
UpdateProduct(product-data) Product
DeleteProduct(product-id, user-id) Product

filterBy - {
    ByOwnerID
    ByTags
    ByPricing
    ByDistance
    ByCondition

}

Product-data - {
    Images
    Owner-id
    LocalPickup (boolean)
    Title
    Description
    Price
    ~~Maximum Distance~~
    Condition (new||used||notSpecified)
    Location
    Tags
}

==========

Product Images storage:

Product-id string (hash key)
Img array associated with above

==========

Product Info DB Schema:

Product-id string (hash key)
Owner-id string (sort key) (user-id)
Location location (sort key)
Tags string (tag1|tag2|tag3)
Price number
Condition (new||used||notSpecified)
Title string
Description string
CreatedAt Timestamp

==========

Product Tags DB Schema:

Tag string (hash key)
Product-id string (sort key)

==========

Product Offers DB Schema:

Offer-id string (hash key)
Buyer-id string (sort key) (user-id)

Seller-id string (sort key) (user-id)
Product-id string
==========

Messaging Service

TODO: Take a look at the automod schema + rooms schema to double check
==========
RPC Schema:
SendMessage(user-id, room-id, text) Message
ListMessages(room-id, cursor) Messages
CreateRoom(seller-id, buyer-id, product-id) Room // happens when you swipe right
ListRooms(user-id, filterBy) Rooms

filterBy {
    ByBuyer
    BySeller
}
==========
Rooms DB Schema:
Room-id string (hash key)
Owner-id string (sort key)
Buyer-id string (sort key)
Product-id string
==========
Messages DB Schema:
Message-id string (hash key)
Room-id string (sort key)
SentAt Timestamp
Text string
==========

...

Story points:

Point Cost	Task Description
1	~~Setup AWS account~~
0.5	~~Transition the frontend repository into a private one~~
2	~~Create the docker file and deploy infra for hosting the frontend~~
2	~~Setup a pipeline for frontend deployments through copilot~~
1	~~Create a temporary graphql service to be used to deploy to fargate~~
2	~~Setup the infra for the graphql api gateway using copilot and deploy the temporary graphql service to it~~
1	~~Create the temporary twirp users service that will be used to deploy to fargate~~
1.5	Setup infra for the users service which will also include the s3 and ~~dynamo creation~~
2	~~Set up a comms VPC channel between graphql and users service to verify proper communication between them only~~
3	Implement the schema designs and logic of the users service
2	Implement the schema designs and logic of the graphql gateway for the users portion
1	Create a temporary product service that will be deployed fargate
1	Create infra for product service including the dynamos and s3
3	Implement the design and logic of the product service
2	Implement the design and logic of the of the graphql gateway for the product portion
1	Research graphql subscriptions further for web socket connections
1	Create the temporary messages twirp service that will be used for fargate deployment

1	Create the infra and deploy the messaging temp service. This includes the dynamo tables
2	Implement the designed messaging service and logic
2	Setup subscriptions via graphql for websockets and pubsub
2	Implement the schema portion of the graphql api for the messaging service

34.5 points total.
+8.5 for margin of error on the count.
Estimation: 43 points.
A minimum of **3.6 points** must be delivered each week to see through the delivery of the backend on time. I'm struggling to see the possibility of that as I can devote a maximum of 2 hours per day during the week and may be double during the weekend and it may be very possible that we might be a month late for this.

What infra we will pay for:

All infra is hosted on **us-east-1**

- 10x ECS to Fargate instances (1 per dev stage - beta and production)
    - GQL gateway
    - Messages service
    - Product service
    - Users service
    - Frontend hosting
- 2x s3 Buckets (share the buckets across staging and production)
- 12x dynamo tables (1 per dev stage - beta and production)

We can start out with just the beta stage and add the production environments when we will plan to go public. This way we can

We will also utilize **fargate** for all of our service as it will be a lot cheaper to handle instead of maintaining EC2 instances