Zebrands Backend Technical Test

URL of deployed project: http://3.129.97.67/api/docs#/

Analysis of requirements

Observations from raw requirements

- Built a basic catalog system to manage products.
- System needs to have two type of users:
    - Admins:
        - Can create, update, delete products
        - Can create, update delete other admin users
    - Anonymous users:
        - Can get product information
- Special requirements:
    - Whenever and admin user makes a change in product, we need to notify all other admins about the change
        - Via email or other mechanism
    - Keep track of the number of times every single product is queried by an **anonymous user**
        - Will be used to build reports in the future

Summary

- Catalog system to manage products
    - User roles and permissions
        - Admins can:
            - create, update, delete products
            - create, update, delete admins
        - Anonymous users:
            - get products
                - (either all or just one)
    - Special requirements:
        - Notify other admins whenever an admin changes a product
            - Via email or other mechanism
        - Track number of times an ANONYMOUS USER queried a product

User stories

1. As an Admin, I want to manage other admins, so that I can assess who has admin permissions to change the catalog system
    1. Use cases:
        1. 1.1 - Create users
            - Acceptance Criteria:
                - Only admins should be able to perform this
                - After execution, a new user with the specified username and encrypted password should be created in the database
                - If a user already exists, it should not create a user in the database and throw an error
        2. 1.2 - Update users
            - Acceptance Criteria:
                - Only admins should be able to perform this
                - After execution, the user should be updated with new data
                - After execution, if user is superuser, it should fail
        3. 1.3 - Delete users
            - Acceptance Criteria:
                - Only admins should be able to perform this
                - After execution, the user should not exist in the database
                - After execution, if user is superuser, it should fail
2. As an Admin, I want to manage the products, so that I can adapt the product to current market and tendencies
    1. Use cases:
        1. 2.1 - Create products
            - Acceptance Criteria:
                - Only admins should be able to perform this
                - After execution, a new product should be created in the database with the specified parameters
        2. 2.2 - Update products
            - Acceptance Criteria:
                - Only admins should be able to perform this
                - After execution, the product should be updated with the new data
        3. 2.3 - Delete products
            - Acceptance Criteria:
                - Only admins should be able to perform this
                - After execution, the product should not exist in the database
3. As an Admin, I want to receive notifications on any change of products, so that I can track the history of the products
    1. Use cases:
        1. 3.1 - Send email notification on change of products

        - Acceptance Criteria:
            - After an execution of create, update or delete on products, an email should be sent to all the admins
4. As an Anonymous user, I want to view the products so that I can know if what I want is available for purchase
    1. Use cases:
        1. 4.1 - Read products
            - Acceptance Criteria:
                - After execution, users will get all the products in the database
                - Users should be able to see one product
                    - For every individual product seen, the views counter should increase
5. As an Anonymous user with admin account, I want to sign in to the catalog system, so that I can perform admin actions
    1. User cases:
        1. 5.1 - Login
            - Acceptance Criteria
                - After execution, it should create a new refresh token and store it with the user
                - After execution, it should create a new access token
                - After execution, it should return the new refresh token and access token
                - Refresh token and access token should expire after certain time has passed
        2. 5.2 - Refresh token
            - Acceptance Criteria
                - After execution, it should revoke the previous refresh token and return a new access token
                - After execution, if refresh token does not match with user or was not found in database, it should throw an error

Nice to haves:

- Tests for code
- Containerize app
- Deploy API to real environment
- Use AWS SES or another 3rd party API to implement notification system
- Provide API documentation (ideally, autogenerated from code)
- Propose an architecture design and explanation on how to scale it in the future

Design of solution

Stack

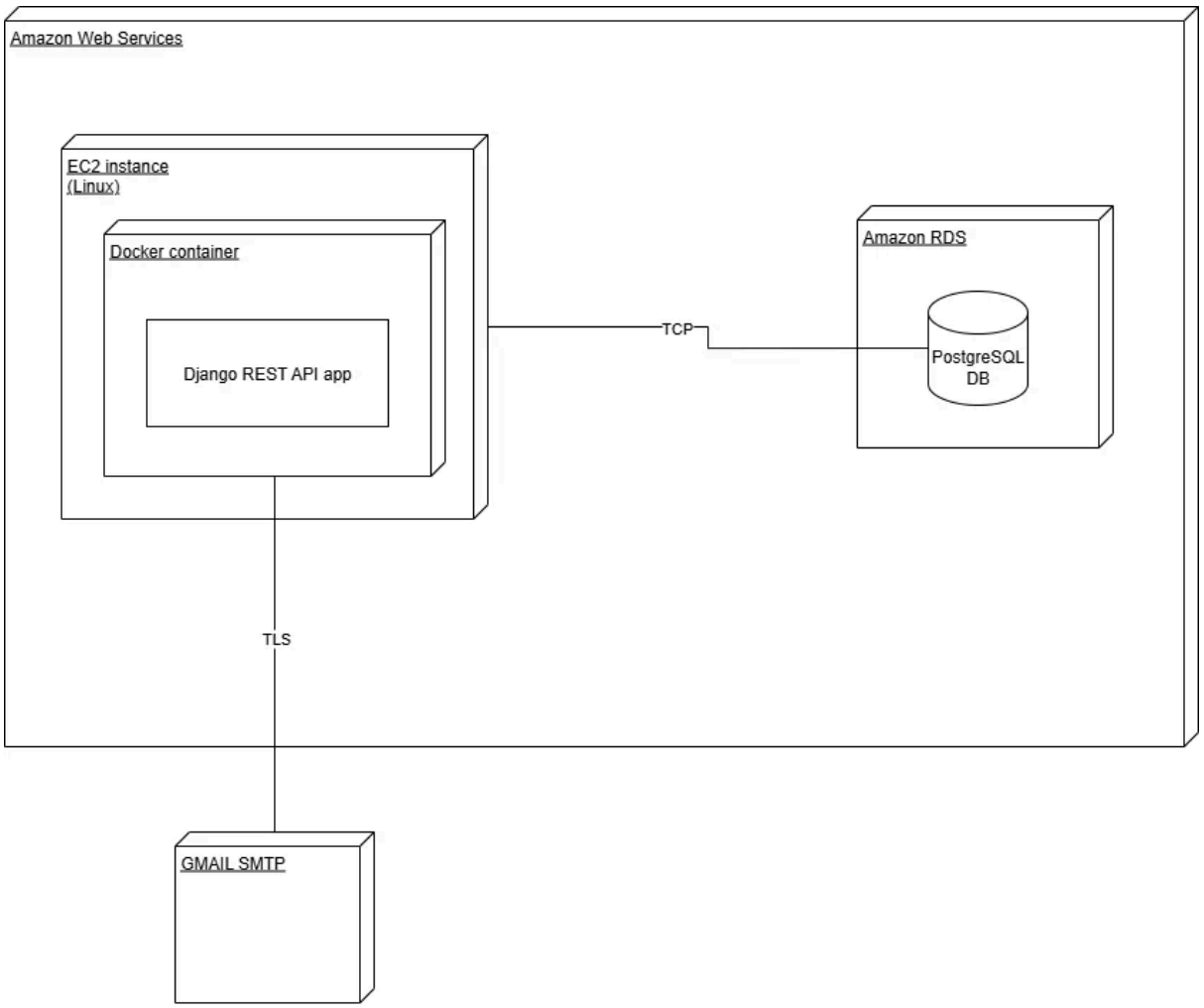
- Docker for containerization of the app
- UV as python package manager
- Django as REST API framework
    - Python 3.10 for compatibility with most of current python libraries
    - `unittest` for automation tests
- GMAIL SMTP for email notifications
- PostgreSQL as main database

Data requirements

- Products
    - SKU: uuid of the product [primary key]
    - name: string
    - price: float
    - brand: string
    - views: integer
- Users
    - handled by Django authentication module

Architecture diagram

- AWS EC2 instance with the docker container
    - This will be used as the main server that holds the django app
- Amazon RDS using a PostgreSQL Database
- Gmail SMTP: The current email provider used to send the emails

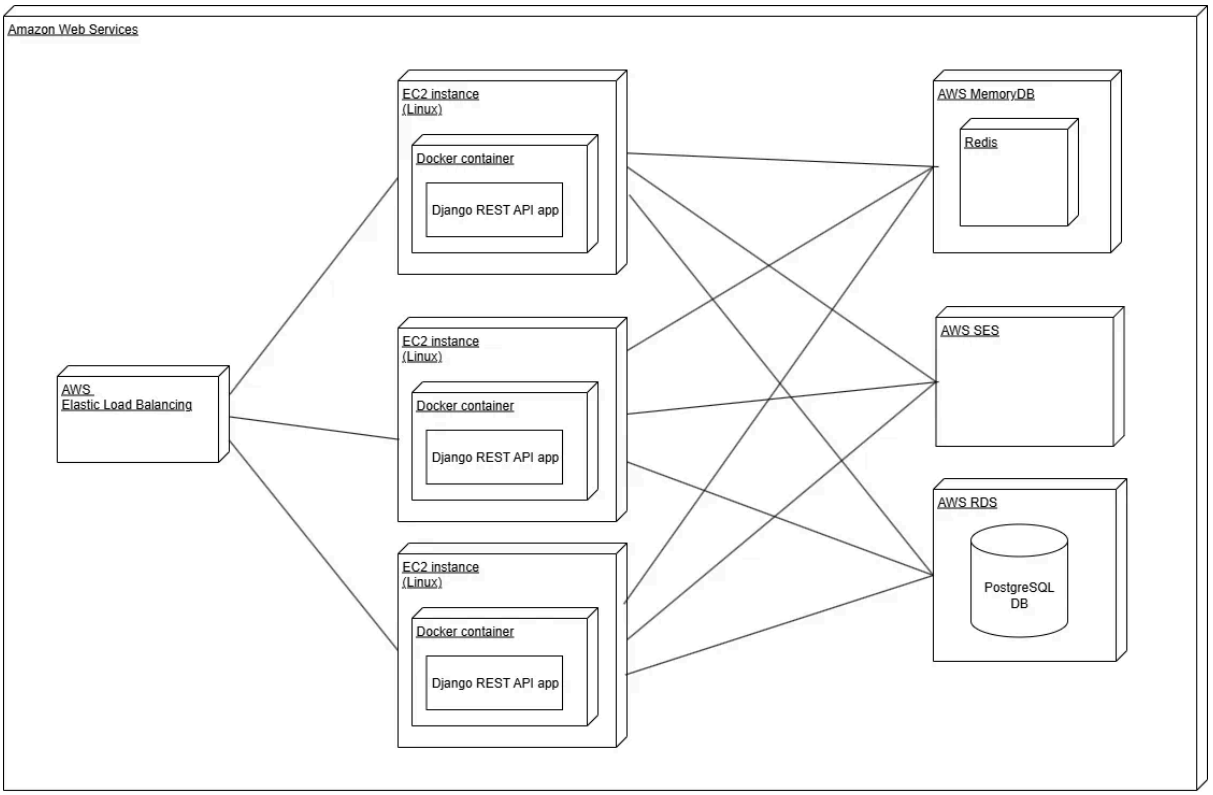


Disadvantages of current architecture

- One single point of failure
    - If the EC2 fails, the whole service is down
    - On deployments of new versions, the service is down for the time it takes to deploy
- No caching
    - Since it is a catalog of products, it will perform a lot of read operations to the database
    - There is no caching to avoid more database increase
- Email notification service
    - It is using GMAIL SMTP with personal email. This is not scalable due to the limit of emails that can be sent
        - Limit is 500 emails per day

Scaled architecture diagram

- This architecture solves the above problems
    - Multiple instances of the application so there is not just one single point of failure
        - Load balancer to redirect the user to the correct server
    - Use of AWS SES as a scalable solution of email notifications
        - The limit of emails is 50,000 per day
    - Use of Redis in AWS MemoryDB to return read operations from cache
        - This will reduce the amount of queries that the database needs to make



This architecture will solve the problems mentioned in the original architecture, however, we can modify it depending on what is important

- We could add more instances of the database to prevent one single point of failure if the database fails or gets deleted however:
    - If we decide to do this, we need to consider what is more valuable to for the application:
        - Consistency: The data is consistent
        - Availability: The data is always available
        - Partition tolerance: The system keeps working even if there is delay or dropped data
    - Based on the CAP Theorem it is impossible to have all three of them, we can only have two of this properties in our distributed system.

- Depending on what we need for the production environment we can either go with:
    - Consistency and availability (CA): Data is consistent and always available → One single instance of the database
    - Consistency and partition tolerance (CP): Multiple instances of database but upon failure block write operations to maintain consistency
    - Availability and partition tolerance (AP): Multiple instances of database and allow write operations upon failure which will reduce consistency