Fiction Express - Backend Developer Position

Task 1

See README.md inside the zip file for instructions and information.

Task 2.1 - Microservices

Switching to a microservices architecture would involve separating the blog application into small independent services, each one running on its own. They could communicate through an API. This would be helpful if the app needed to handle vast amounts of data. For instance, if we notice that the user registrations are significantly increasing, we can scale up only a dedicated user service without the need of having to scale the entire application, which would be less time consuming and easier.

For the simple blog we have I would define three different microservices: user service, blog service (each with their own database), and a frontend service. Each one would handle its own functionalities, which would briefly be:

- **User service**: user credentials and roles.
- **Blog service**: blog post operations. Posting and deleting.
- **Frontend service**: provide a user interface that interacts with the backend (other microservices)

A system like this can be scaled horizontally which is more efficient and more scalable than a monolithic system. Updates can be deployed to individual services without affecting the entire system. Microservices offer agility to adapt to new features and changing loads. Another big advantage would be that a different tech stack can be used for each different one.

To centralize everything, we could use an API gateway which would facilitate load management and would make the services accessible from one point and would group everything and give a cleaner feel.

Task 2.2 - Non-relational databases

Non-relational databases into the blog app, specifically MongoDB or DynamoDB, have strengths in handling unstructured data that would be helpful. For example, the blog service would benefit from a non-relational database which is ideal for the varied nature of blog content.

Here's a simple schema for MongoDB:

User Collection:

```
{
  "_id": ObjectId("507f191e810c19729de860ea"),
  "first_name": "Ton",
  "last_name": "Vilà",
  "email": "ton.vila.r@gmail.com",
  "password": "hashed_password"
}
```

Post Collection:

```
{
  "_id": ObjectId("304f115e811c147wwwe860q3"),
  "title": "Fiction Express goes global",
  "content": "Lorem ipsum dolor sit amet, consectetur...",
  "date_posted": ISODate("2024-02-16T08:00:00Z"),
  "author_id": ObjectId("507f191e810c19729de860ea")
}
```

By using MongoDB, the schema remains flexible, so if I decide to add new features to the blog, like comments or tags, I can easily do so without major database restructuring. Also, both MongoDB and DynamoDB provide scalability perks, making them suitable for dealing with a high volume of data and user traffic, ensuring the blog remains performant and responsive as it grows.

Task 2.3 - Containerization with Docker

See README.md inside the zip file for instructions and information.

Task 3: AWS and Deployment (Task 2 - Docker Container)

Deploying the blog app on Amazon EC2 is a good idea. EC2 provides the flexibility to scale up or down based on our application's needs. This flexibility is useful for managing costs while handling varying loads, making it ideal for our blog application that might see fluctuating traffic. EC2 also gives us control of the computing environment, we can choose the operating system, configure the environment, and install any necessary software like Docker and Docker Compose, which we will need to run our containerized application.

Since EC2 is a pay-as-you-go service it ensures we're only paying for what we use, which is ideal for keeping costs low while ensuring our blog application remains highly available to our users.

My instructions on how I would approach deploying the application on AWS EC2:

1. **Prepare the Environment:**

- Start by creating an EC2 instance. I'd select a suitable instance type like t2.micro for starting, since it is included in the free tier. For the operating system, Amazon Linux AMI (Amazon Machine Image) is a good choice because it's optimized for AWS and comes with Docker available in its repositories.
- Next, set up the security group for the EC2 instance to allow inbound traffic on port 80 (HTTP), 443 (HTTPS) for web access, and 22 (SSH) for remote management.

2. **Installing Docker and Docker Compose:**

- After launching the EC2 instance, I'd SSH into it and install Docker to run containerized applications and Docker Compose to manage multi-container Docker applications, which our blog app is an example of.

3. **Deploying the Application:**

- With Docker set up, the next step is to get our application onto the EC2 instance. This could be done by cloning the app from my git repository.
- Once the application is on the EC2 instance, I'd navigate to the project directory and follow the instructions detailed in the repositories README.md to build, install, etc.

4. **Configuration Adjustments for Production:**

- Make that the Django settings are configured for production. This includes setting DEBUG to False, configuring the ALLOWED_HOSTS with the EC2 instance's public DNS or IP address, and setting up the database connection to use the MySQL service.

5. **Accessing the Deployed Application:**

- After the containers are up and running, the application should be accessible through the EC2 instance's public IP address or DNS name. This information can be found in the EC2 Management Console.