# Intro

I have created a basic branch which does only what the pdf describes – accept a post request with customer details and either returns an ACCEPT or REJECT based on the criteria. However, this document and the files attached cover an extended solution which has future proofing in mind and will be able to host multiple, more advanced solutions.

The service can host customer, policy and ‘customer policy’ information which can store data on which customer has which policies, as well as store the information on why a customer was rejected for a specific policy, instead of losing that information with the simpler service. When a customer is rejected for a policy, this request is still considered valid and is saved, but the policy is not activated, and rejection information is retained.

# Design Overview

The service is designed to host credit policies using a Django application with a RESTful API. The core components include models for Customer, Policy, and CustomerPolicy, along with associated serializers, views, and validation. The service provides endpoints for creating and managing the models.

The service is designed with Django due to its powerful model support. Future proofing was a key concern in the design of the service; with Django you can modify the models with new fields and relationships and migrate the databases to them with ease which would add support for more complicated credit policies.

Models:

* Customer: Stores information about customers, such as income, debt to income ratio, age, and payment remarks.
* Policy: Represents policies with constraints on income, debt, payment remarks, and age.
* CustomerPolicy: Links customers and policies, indicating whether a policy is accepted or rejected, listing reasons for rejections.

All models include basic validations such as ensuring correct datatypes for decimal fields, or ensuring minimum/maximum value validations (for example: not allowing an age past 101)

### API Endpoints:

* Customers: http://localhost:8000/api/customers/
* Policies: http://localhost:8000/api/policies/
* CustomerPolicies: http://localhost:8000/api/customer\_policies/

All endpoints support GET, POST, and DELETE requests.

Database:

The service utilizes a sqlite3 database which was chosen as it is the default database for Django and is included in python which allows me to easily demo this project without unnecessary requirements and setup steps. I acknowledge that in a production environment sqlite3 would not be the correct choice and I would opt for PostgreSQL which is a more powerful and feature rich alternative.

My reasoning behind choosing a SQL vs a NoSQL database was:

* SQL databases are designed for structured data with well-defined relationships between entities. In a service where customers, policies, and their relationships are clearly defined, a SQL database allows you to model these relationships using tables and foreign keys.
* SQL databases adhere to the ACID (Atomicity, Consistency, Isolation, Durability) properties, ensuring transactional consistency and reliability, which are crucial when dealing with financial or sensitive information, such as customer details and policies.

### Admin Interface:

The project includes an administration interface which can be used to manage all models. This can be accessed at: http://localhost:8000/admin/

### Testing

The service includes a suite of Model and API unit tests which test all validation features of the models, and test creating all models directly via Django, and the API.

# Setup

Tested on Python 3.10.0.

1. Navigate to the project directory.
2. Create a virtual environment: python -m venv env
3. Source the virtual environment:
   1. Windows: .\env\scripts\activate
   2. Linux/OSX: source env/bin/activate
4. Install requirements: pip install -r requirements.txt
5. Create migrations: python manage.py makemigrations
6. Run migrations: python manage.py migrate
7. Create superuser for admin page: python manage.py createsuperuser
8. Run server: python manage.py runserver

The service is now up and running and requests can be made to the API endpoints described earlier. For example requests, see example\_requests.md.

### Unit Tests:

With the service stopped: python manage.py test --no-input

### Helper Scripts:

I have included helper scripts to upload sample data to the database or clear it. Note that the service needs to be running for these to work.

These can be found in the helpers folder.

To create sample date run: python upload\_sample\_data.py

To clear the DB, run: python clean\_db.py (ensure your PYTHONPATH is in the project root)

### Dockerfile

A dockerfile is included to run the project. To run:

1. docker build -t anyfin\_project .
2. docker run -p 8000:8000 anyfin\_project

To create superuser or run helper scripts:

1. docker exec -it <container\_name> bash
2. python manage.py createsuperuser
3. exit