BankSystem

Test Environment

- **OS**: Windows10 22H2
- **Programming Language**: Python3.9
  - Django 4.2
  - mysqlclient 2.1.1
- **IDE**: PyCharm 2023.1

Execution

Set database

```
# in file 'BankSystem/setting.py'
# Set the properties of database so that it can connect with MySQL
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'BankSystem',
        'USER': 'root',
        'PASSWORD': 'xxx',
        'HOST': '127.0.0.1',
        'PORT': 3306
    }
}
```

Then create a new scheme named “ `banksystem` ” in MySQL.

Initializing environment

```
python manage.py makemigrations
python manage.py migrate
```

Create superuser

create superuser for initial work, e.g. adding new employees.

```
python manage.py createsuperuser --username=<your user name>
```

Please enter the password same with the username!!!

Start server

```
python manage.py runserver
```

Client

Open browser and visit `http://127.0.0.1:8000`

You need to log in before visiting or editing anything. If you don’t have a account, please look back to step `Create superuser` and log in as superuser.

In this system, you can log in with ID as **customer**, **employee** or **superuser**.

Design Overview

Structure

The bank system consists of two modules: the frontend and the backend.

- **The frontend** is based on open-source bootstrap template
    - It utilize JavaScript to visit and render the data resource via corresponding URL.
    - It create/edit/delete data by visiting corresponding URL, which is displayed as several buttons.
- **The backend** is implemented by **Django Rest Framework**.
    - It communicate with the frontend via HTTP1.1 protocol.
    - It provides interface `/api/{table_name}` for frontend to gain or create data by GET or POST.
    - It provides interface `/api/{table_name}/{primary_key}` for frontend to edit or delete data by PUT or DELETE.

File Origanization

Backend

The main files and their functions are listed as follow

```
BankManagement
├── __init__.py
├── admin.py
├── apps.py
├── migrations
├── models.py           # Definition about the table
├── serializers.py      # Serializating data of table
├── tests.py
├── urls.py             # Configuration about automatic URL router
└── views.py            # Respondance to requisition of the frontend
```

Frontend

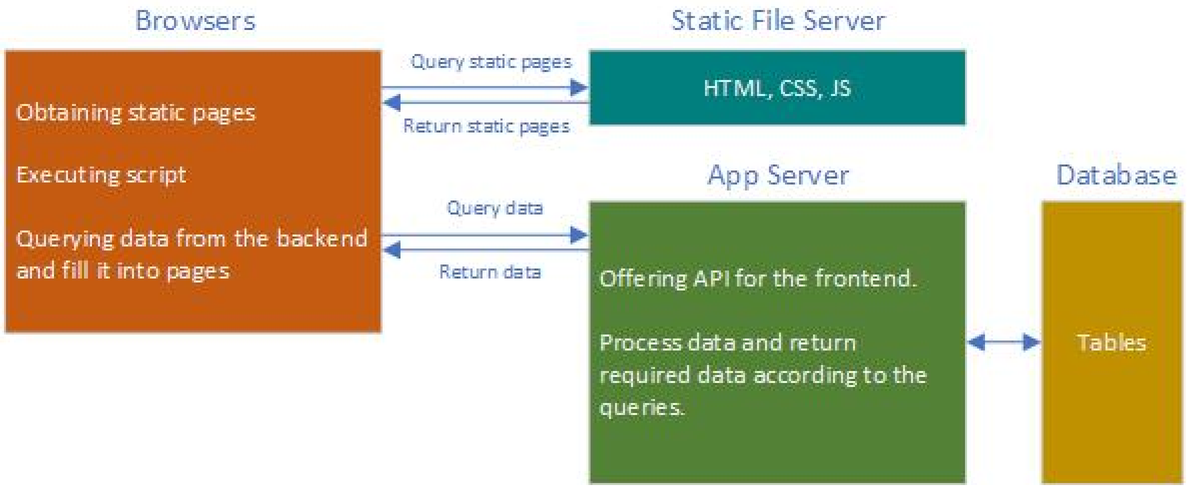
The main files and their functions are listed as followed:

```
BankFrontend
├── __init__.py
├── __pycache__
├── admin.py
├── apps.py
├── migrations
├── models.py
├── templates                           # The file of templates
│   └── BankFrontend
│       ├── dist
│       │   ├── 400.html
│       │   ├── 404.html
│       │   ├── 500.html
│       │   ├── checkaccounts.html
│       │   ├── customers.html
│       │   ├── employees.html
│       │   ├── index.html
│       │   ├── savingaccounts.html
│       │   └── tables.html
│       └── index.html
├── tests.py
├── urls.py                             # Definition of URL available to the frontend.
└── views.py                            # Renders of templates visited.
```

The static files required by the fronted are listed as followed:

```
static
├── dist
│   ├── assets
│   │   ├── demo
│   │   └── img
│   ├── css
│   └── js
├── scripts
└── src
    ├── assets
    │   ├── demo
    │   └── img
    ├── js
    ├── pug
    │   ├── layouts
    │   │   └── includes
    │   │       ├── head
    │   │       └── navigation
    │   └── pages
    │       └── includes
    └── scss
        ├── layout
        ├── navigation
        │   └── sidenav
        └── variables
```

Workflow Overview



Detail

Models / Tables

Customer

The definition of table:

```python
# models.py

class Customer(models.Model):
    Customer_ID = models.CharField('Customer_ID', primary_key=True, max_length=18)
    Customer_Name = models.CharField('Customer_Name', max_length=MAX_CHAR_LEN, blank=False)
    Customer_Phone_Number = models.DecimalField('Customer_Phone_Number', max_digits=11, decimal_places=0, blank=False)

    class Meta:
        db_table = 'Customer'
```

The definition of serializer:

```python
# serializers.py

class CustomerSerializer(serializers.ModelSerializer):
    class Meta:
        model = Customer
        fields = ('Customer_ID', 'Customer_Name', 'Customer_Phone_Number')
```

It is supported to create/edit/delete information of customers.

- `list`

  Get data of customer(s)

  For queries from customer, just return information relative this customer.

  For queries from superuser or employee, return all information.

```
def list(self, request):
    role = request.session.get('role')

    if role == 'customer':
        id = request.session.get('user_id')
        queryset = Customer.objects.filter(pk=id)
    else:
        queryset = Customer.objects.all()

    serializer = CustomerSerializer(queryset, many=True)
    return Response(serializer.data)
```

- `create`

  To create a new entry about customer, we need to get the data from query serialized.

  If the data is valid, create new customer entry in database and new user entry in ‘auth’ module of django, and finally return positive response.

  Return “Bad request” if encountering any invalid data or invalid operation.

```
def create(self, request):
```

```python
    serializer = CustomerSerializer(data=request.data)

    if serializer.is_valid():
        id = request.data.get('Customer_ID')
        if User.objects.filter(username=id):
            user = authenticate(username=id)
            if user and not user.is_active:
                user = User.objects.get(username=id)
                user.is_active = True
                user.save()
            else:
                return Response({
                    'status': 'Bad request',
                    'message': 'User exists',
                }, status=status.HTTP_400_BAD_REQUEST)
        else:
            User.objects.create_user(username=id, password=id)

        Customer.objects.create(**serializer.validated_data)
        return Response({
            'status': 'Success',
            'message': 'Create new Customer Successfully'}, status=status.HTTP_201_CREATED)

    return Response({
        'status': 'Bad request',
        'message': 'Invalid data',
    }, status=status.HTTP_400_BAD_REQUEST)
```

- `update`

  Updating customer’s information acquires insurance of transaction properties.

  Firstly, it look up the database for object with the same primary key.

  Then, it update information of customer by row granularity(all field of customer entry: Customer_Name, Customer_Phone_Number)

  Finally, it response the client with successful signal if not encountering anything exceptional.

```
@transaction.atomic
    def update(self, request, pk=None):
        queryset = Customer.objects.filter(pk=pk)
        if not queryset.exists():
            return Response({
                'status': 'Failed',
                'message': 'Customer not exist'}, status=status.HTTP_400_BAD_REQUEST)
        if pk != request.data.get("Customer_ID"):
            return Response({
                'status': 'Failed',
                'message': 'Could not change Customer_ID'}, status=status.HTTP_400_BAD_REQUEST)
        with transaction.atomic():
            queryset.update(
                Customer_Name=request.data.get("Customer_Name") if request.data.get("Customer_Name") else queryset[
                    0].Customer_Name,
                Customer_Phone_Number=request.data.get("Customer_Phone_Number") if request.data.get(
                    "Customer_Phone_Number") else queryset[0].Customer_Phone_Number,
            )
        return Response({
            'status': 'Success',
            'message': 'Update data Successfully'}, status=status.HTTP_200_OK)
```

- `destroy`

  To destroy a entry of customer.

  Firstly, it look up the database for object with the same primary key.

  If the data is valid, delete the customer in database and set the user entry as ‘inactive’ in ‘auth’ module, and finally return positive response.

  Return “Bad request” if encountering any invalid data or invalid operation.

```
def destroy(self, request, pk=None):
  queryset = Customer.objects.all()
  customer = get_object_or_404(queryset, pk=pk)
  try:
      user = User.objects.get(username=customer.Customer_ID)
      user.is_active = False
      user.save()
      customer.delete()
  except ProtectedError as e:
      return Response({
          'status': 'Failed',
          'message': 'Could not delete'}, status=status.HTTP_400_BAD_REQUEST)
```

```
    return Response({
        'status': 'Success',
        'message': 'Delete data Successfully'}, status=status.HTTP_200_OK)
```

Employee

The definition of employee table.

There are three fields about employee:

- `Employee_ID` the unique ID of employee object
- `Employee_Name` the name of employee
- `Employee_Hire_Date` the date on which the employee is hired

```
# models.py

class Employee(models.Model):
    Employee_ID = models.CharField('Employee_ID', max_length=18, primary_key=True)
    Employee_Name = models.CharField('Employee_Name', max_length=MAX_CHAR_LEN, blank=False)
    Employee_Hire_Date = models.DateTimeField('Employee_Hire_Date', blank=False)

    class Meta:
        db_table = 'Employee'
```

The definition of serializer of employee object:

```python
# serializers.py

class EmployeeSerializer(serializers.ModelSerializer):
    class Meta:
        model = Employee
        fields = ('Employee_ID', 'Employee_Name',
                  'Employee_Hire_Date')
```

The bank system support create new entry about employee. The process is just like that of customer table.

- At first, it tries to serialize the data from query
- Then, if the data is valid and there is not existed user with the same ID, it create new entry of employee.
- If anything invalid happens, return ‘Bad request’

```python
class EmployeeViewSet(viewsets.ReadOnlyModelViewSet):
    permission_classes = (AllowAny,)
    queryset = Employee.objects.all()
    serializer_class = EmployeeSerializer

    def create(self, request):
        serializer = EmployeeSerializer(data=request.data)

        if serializer.is_valid():
            if User.objects.filter(username=request.data.get('Employee_ID')):
                return Response({
                    'status': 'Bad request',
                    'message': 'User exists',
                }, status=status.HTTP_400_BAD_REQUEST)
            else:
                User.objects.create_user(
                    username=request.data.get('Employee_ID'),
                    password=request.data.get('Employee_ID'),
                    is_staff=True
                )

                Employee.objects.create(**serializer.validated_data)
            return Response({
                'status': 'Success',
                'message': 'Create new Employee Successfully'}, status=status.HTTP_201_CREATED)

        return Response({
            'status': 'Bad request',
            'message': 'Invalid data',
        }, status=status.HTTP_400_BAD_REQUEST)
```

CheckAccount

The definition of check account table.

There are three fields about check account:

- `CAccount_ID` the unique ID of check account object
- `CAccount_Balance` the amount of check account

- `CAccount_Open_Date` the date on which this check account was created, which is filled in by server automatically

```python
# models.py

class CheckAccount(models.Model):
    CAccount_ID = models.CharField('CAccount_ID', primary_key=True, max_length=MAX_CHAR_LEN)
    CAccount_Balance = models.DecimalField('CAccount_Balance', max_digits=20, decimal_places=2, blank=False)
    CAccount_Open_Date = models.DateTimeField('CAccount_Open_Date', blank=False)

    class Meta:
        db_table = 'CheckAccount'
```

The definition of serializer of check account object:

```python
# serializers.py

class CheckAccountSerializer(serializers.ModelSerializer):
    class Meta:
        model = CheckAccount
        fields = ('CAccount_ID', 'CAccount_Balance', 'CAccount_Open_Date')
```

It is supported to create/edit/delete information of check accounts.

- `list`

  Get data of check account(s)

  For queries from customer, just return information relative this customer(Looking up the table CustomerToCA and get all information about specific customer by Customer_ID).

  For queries from superuser or employee, return all information.

```
def list(self, request):
    role = request.session.get('role')
    if role == 'customer':
        id = request.session.get('user_id')
        cond_queryset = CustomerToCA.objects.filter(Customer_ID=id)
        queryset = CheckAccount.objects.filter(CAccount_ID__in=cond_queryset.values('CAccount_ID'))
    else:
        queryset = CheckAccount.objects.all()

    serializer = CheckAccountSerializer(queryset, many=True)
    return Response(serializer.data)
```

- `create`

  It requires three properties to create a new check account: `Customer_ID`, `CAccount_Balance`, `CAccount_ID` and properties of transaction is ensured.

  After checking validation of query, it look up the Customer table for a entry with the same `Customer_ID` with that of query.

Then it get the datetime and create new entry in table **CheckAccount** and **CustomerToCA**

If encountering anything invalid, it returns ‘Bad Request’

```
@transaction.atomic
def create(self, request):
    checkaccount = request.data.copy()

    try:
        checkaccount.pop('Customer_ID')
    except KeyError as e:
        return Response({
            'status': 'Failed',
            'message': 'Customer_ID is required'}, status=status.HTTP_400_BAD_REQUEST)

    ca_to_customer = request.data.copy()

    try:
        ca_to_customer.pop('CAccount_Balance')
    except KeyError as e:
        return Response({
            'status': 'Failed',
            'message': 'More information is required'}, status=status.HTTP_400_BAD_REQUEST)

    queryset = Customer.objects.filter(pk=request.data.get('Customer_ID'))
    if not queryset.exists():
        return Response({
            'status': 'Failed',
            'message': 'Customer not exist'}, status=status.HTTP_400_BAD_REQUEST)

    try:
        with transaction.atomic():
            checkaccount['CAccount_Open_Date'] = datetime.datetime.now()
            ca_serializer = CheckAccountSerializer(data=checkaccount)

            if ca_serializer.is_valid():
                CheckAccount.objects.create(**ca_serializer.validated_data)


                ca_to_customer['CAccount_Last_Access_Date'] = datetime.datetime.now()

                ca_to_customer_serializer = CustomerToCASerializer(data=ca_to_customer)
                if ca_to_customer_serializer.is_valid():
                    CustomerToCA.objects.create(**ca_to_customer_serializer.validated_data)

    except IntegrityError as e:
        return Response({
            'status': 'Bad request',
            'message': str(e),
        }, status=status.HTTP_400_BAD_REQUEST)

    return Response({
        'status': 'Success',
        'message': 'Create new Check Account Successfully'}, status=status.HTTP_201_CREATED)
```

- `update`

  The update of CheckAccount is similar to creating.

  Look up the table CustomerToCA to get field `CAccount_ID` and edit the corresponding entry in table CheckAccount.

  Return ‘Bad Request’ if meeting anything invalid.

```
@transaction.atomic
def update(self, request, pk=None):
    # Only balance and overdraft are allowed to modify
    queryset = CheckAccount.objects.filter(pk=pk)
    if not queryset.exists():
        return Response({
            'status': 'Failed',
            'message': 'Check Account not exist'}, status=status.HTTP_400_BAD_REQUEST)
    if pk != request.data.get("CAccount_ID"):
        return Response({
            'status': 'Failed',
            'message': 'Could not change CAccount_ID'}, status=status.HTTP_400_BAD_REQUEST)
    try:
        with transaction.atomic():
             queryset.update(
                CAccount_ID=pk,
                CAccount_Balance=request.data.get('CAccount_Balance') if request.data.get('CAccount_Balance') else
                queryset[0].CAccount_Balance,
            )
            queryset = CustomerToCA.objects.filter(CAccount_ID=pk)
            queryset.update(
                CAccount_Last_Access_Date=datetime.datetime.now()
            )
    except IntegrityError as e:
        return Response({
            'status': 'Bad request',
            'message': str(e),
        }, status=status.HTTP_400_BAD_REQUEST)

    return Response({
        'status': 'Success',
        'message': 'Update Check Account Successfully'}, status=status.HTTP_200_OK)
```

- `destroy`

  Look up for entry with the same `CAccount_ID` in table CheckAccount and remove it.

  Return ‘Bad Request’ if meeting anything invalid.

```
@transaction.atomic
def destroy(self, request, pk=None):
    queryset = CheckAccount.objects.all()
    checkaccount = get_object_or_404(queryset, pk=pk)
    queryset = CustomerToCA.objects.all()
    customer_to_ca = get_list_or_404(queryset, CAccount_ID=pk)
    try:
        with transaction.atomic():
            for obj in customer_to_ca:
                obj.delete()
            checkaccount.delete()
    except IntegrityError as e:
        return Response({
            'status': 'Bad request',
            'message': str(e),
        }, status=status.HTTP_400_BAD_REQUEST)

    return Response({
        'status': 'Success',
        'message': 'Delete Check Account Successfully'}, status=status.HTTP_200_OK)
```

SavingAccount

The definition of saving account table.

There are three fields about saving account:

- `SAccount_ID` the unique ID of saving account object
- `SAccount_Balance` the amount of saving account

- `SAccount_Open_Date` the date on which this saving account was created, which is filled in by server automatically

```python
# models.py

class SavingAccount(models.Model):
    SAccount_ID = models.CharField('SAccount_ID', primary_key=True, max_length=MAX_CHAR_LEN)
    SAccount_Balance = models.DecimalField('SAccount_Balance', max_digits=20, decimal_places=2, blank=False)
    SAccount_Open_Date = models.DateTimeField('SAccount_Open_Date', blank=False)

    class Meta:
        db_table = 'SavingAccount'
```

The definition of serializer of saving account object:

```python
# serializers.py

class SavingAccountSerializer(serializers.ModelSerializer):
    class Meta:
        model = SavingAccount
        fields = ('SAccount_ID', 'SAccount_Balance', 'SAccount_Open_Date')
```

It is supported to create/edit/delete information of saving accounts.

- `list`

  Get data of saving account(s)

  For queries from customer, just return information relative this customer(Looking up the table CustomerToSA and get all information about specific customer by Customer_ID).

  For queries from superuser or employee, return all information.

  ```
  def list(self, request):
      role = request.session.get('role')

      if role == 'customer':
          id = request.session.get('user_id')
          cond_queryset = CustomerToSA.objects.filter(Customer_ID=id)
          queryset = SavingAccount.objects.filter(SAccount_ID__in=cond_queryset.values('SAccount_ID'))
      else:
          queryset = SavingAccount.objects.all()

      serializer = SavingAccountSerializer(queryset, many=True)
      return Response(serializer.data)
  ```

- `create`

  It requires three properties to create a new saving account: `Customer_ID`, `SAccount_Balance`, `SAccount_ID` and properties of transaction is ensured.

After checking validation of query, it look up the Customer table for a entry with the same `Customer_ID` with that of query.

Then it get the datetime and create new entry in table **SavingAccount** and **CustomerToSA**

If encountering anything invalid, it returns ‘Bad Request’

```python
def create(self, request):
    savingaccount = request.data.copy()

    try:
        savingaccount.pop('Customer_ID')
    except KeyError as e:
        return Response({
            'status': 'Failed',
            'message': 'Customer_ID is required'}, status=status.HTTP_400_BAD_REQUEST)
    try:
        sa_to_customer = request.data.copy()
        sa_to_customer.pop('SAccount_Balance')
    except KeyError as e:
        return Response({
            'status': 'Failed',
            'message': 'More information is required'}, status=status.HTTP_400_BAD_REQUEST)

    queryset = Customer.objects.filter(
        pk=request.data.get('Customer_ID'))
    if not queryset.exists():
        return Response({
            'status': 'Failed',
            'message': 'Customer not exist'}, status=status.HTTP_406_NOT_ACCEPTABLE)

    savingaccount['SAccount_Open_Date'] = datetime.datetime.now()
    sa_serializer = SavingAccountSerializer(data=savingaccount)

    try:
        with transaction.atomic():
            if sa_serializer.is_valid():
                SavingAccount.objects.create(**sa_serializer.validated_data)
                sa_to_customer['SAccount_Last_Access_Date'] = datetime.datetime.now()

                sa_to_customer_serializer = CustomerToSASerializer(
                    data=sa_to_customer)
                if sa_to_customer_serializer.is_valid():
                    CustomerToSA.objects.create(
                        **sa_to_customer_serializer.validated_data)
    except IntegrityError as e:
        return Response({
            'status': 'Bad request',
            'message': str(e),
        }, status=status.HTTP_400_BAD_REQUEST)

    return Response({
        'status': 'Success',
        'message': 'Create new Saving Account Successfully'}, status=status.HTTP_201_CREATED)
```

- `update`

  The update of SavingAccount is similar to creating.

  Look up the table CustomerToSA to get field `SAccount_ID` and edit the corresponding entry in table SavingAccount.

  There are two kinds of operations:

  - **Edit**: Update the whole entry of a single account
  - **Transfer**: Transfer specific number of balance to another account.

  Return 'Bad Request' if meeting anything invalid.

```
@transaction.atomic
def update(self, request, pk=None):
    # Only balance and overdraft are allowed to modify
    queryset = SavingAccount.objects.filter(pk=pk)
    if not queryset.exists():
        return Response({
            'status': 'Failed',
            'message': 'Check Account not exist'}, status=status.HTTP_400_BAD_REQUEST)

    if pk != request.data.get("SAccount_ID"):
        return Response({
            'status': 'Failed',
            'message': 'Could not change SAccount_ID'}, status=status.HTTP_400_BAD_REQUEST)

    try:
        with transaction.atomic():
            if request.data.get('ope') == 'edit':
                queryset.update(
                    SAccount_ID=pk,
                        SAccount_Balance=request.data.get('SAccount_Balance') if request.data.get(
                        'SAccount_Balance') else
                    queryset[0].SAccount_Balance,
                )
                queryset = CustomerToSA.objects.filter(SAccount_ID=pk)
                queryset.update(
                    SAccount_Last_Access_Date=datetime.datetime.now()
                )
            elif request.data.get('ope') == 'transfer':
                target_queryset = SavingAccount.objects.filter(pk=request.data.get('Target_SAccount_ID'))
                if not target_queryset.exists():
                    return Response({
                        'status': 'Failed',
                        'message': 'Target Check Account not exist'}, status=status.HTTP_400_BAD_REQUEST)

                transfer_amount = int(request.data.get('Transfer_Amount'))
                if transfer_amount <= 0:
                    return Response({
                        'status': 'Failed',
                        'message': 'Invalid transfer amount'}, status=status.HTTP_400_BAD_REQUEST)

                queryset.update(
                    SAccount_Balance=queryset[0].SAccount_Balance - transfer_amount
                )
                target_queryset.update(
                    SAccount_Balance=target_queryset[0].SAccount_Balance + transfer_amount
                )
                queryset = CustomerToSA.objects.filter(SAccount_ID=pk)
                queryset.update(
                    SAccount_Last_Access_Date=datetime.datetime.now()
                )
                target_queryset = CustomerToSA.objects.filter(SAccount_ID=request.data.get('Target_SAccount_ID'))
                target_queryset.update(
                    SAccount_Last_Access_Date=datetime.datetime.now()
                )

    except IntegrityError as e:
        return Response({
            'status': 'Bad request',
            'message': str(e),
        }, status=status.HTTP_400_BAD_REQUEST)

    return Response({
        'status': 'Success',
        'message': 'Update Check Account Successfully'}, status=status.HTTP_200_OK)
```

- `destroy`

  Look up for entry with the same `SAccount_ID` in table SavingAccount and remove it.

  Return ‘Bad Request’ if meeting anything invalid.

```python
@transaction.atomic
def destroy(self, request, pk=None):
    queryset = SavingAccount.objects.all()
    savingaccount = get_object_or_404(queryset, pk=pk)
    queryset = CustomerToSA.objects.all()
    customer_to_sa = get_list_or_404(queryset, SAccount_ID=pk)
    try:
        with transaction.atomic():
            for obj in customer_to_sa:
                obj.delete()
            savingaccount.delete()
    except IntegrityError as e:
        return Response({
            'status': 'Bad request',
            'message': str(e),
        }, status=status.HTTP_400_BAD_REQUEST)
    return Response({
        'status': 'Success',
        'message': 'Delete Saving Account Successfully'}, status=status.HTTP_200_OK)
```

CustomerToCA

```python
# models.py

class CustomerToCA(models.Model):
    CAccount_Last_Access_Date = models.DateTimeField('CAccount_Last_Access_Date', auto_now=True)

    CAccount_ID = models.ForeignKey(CheckAccount, on_delete=models.CASCADE)
    Customer_ID = models.ForeignKey(Customer, on_delete=models.PROTECT)

    class Meta:
        db_table = 'CustomerToCA'
        constraints = [
            models.UniqueConstraint(fields=['Customer_ID', 'CAccount_Open_Bank_Name'], name='One customer is only allowed to open one CA in one bank'),
            models.UniqueConstraint(fields=['CAccount_ID', 'Customer_ID'], name='CustomerToCA Fake Primary Key')
        ]
```

This is the relationship between customer and its check account:

- A customer is only allowed to have a check account
- The last access date is filled by server automatically

CustomerToSA

```python
# models.py

class CustomerToSA(models.Model):
    SAccount_Last_Access_Date = models.DateTimeField('SAccount_Last_Access_Date', auto_now=True)

    SAccount_ID = models.ForeignKey(SavingAccount, on_delete=models.CASCADE)
    Customer_ID = models.ForeignKey(Customer, on_delete=models.PROTECT)

    class Meta:
        db_table = 'CustomerToSA'
        constraints = [
            models.UniqueConstraint(fields=['Customer_ID', 'SAccount_Open_Bank_Name'], name='One customer is only allowed to open one SA in one bank'),
            models.UniqueConstraint(fields=['SAccount_ID', 'Customer_ID'], name='CustomerToSA Fake Primary Key')
        ]
```

This is the relationship between customer and its saving account:

- A customer is only allowed to have a saving account
- The last access date is filled by server automatically

The frontend

The frontend is implemented by Django app.

- The frontend will render corresponding templates when visiting specific URL. Then JavaScript will gain data via URL and fill forms
- The browser will jump to `/dist/index.html` when visiting `index.html` . Each URL is relative to a HTML file. All static files is stored in directory `static`
- Browser visit URL and gain data from the backend by **ajax**.