Budgeter

Documentation

Product Name: Budgeter

Product Version: 1.0

Current Phase: Development

Current Date: 06.10.2022

Part I

User Documentation

Product Overview

Budgeter is a simple and user-friendly budgeting tool that helps users to keep track of their financial habits, whether it is about spending or earning money.

Budgeter is divided into three separate applications, the authenticator, the budgeter and the wallet. Its modularity allows for re-usability and ease of maintenance. New features are easy to be implemented, so future releases ought to extend the functionalities and further improve the user experience.

The included documentation presents the main functionalities of Budgeter, provides thorough installation instructions and gives tips on how to use the application's main dashboard.

The user documentation is followed by the system documentation and the source code, where technical aspects can be found. The former includes details on the application backlog, information architecture, user-flow scheme and models included in the project.

Product Objectives

Budgeter aims to provide an intuitive application that enables users to record and manage their day-to-day financial transactions, including expenditures and earnings.

The user interface is built taking into account simplicity, ease of access and rich information. The application's main dashboard brings to the user useful information at a glance, intuitive controls and a minimalistic design.

The included technical documentation details the user stories behind the application's development, as well as the user journey. The rolled-out features help to reliably accomplish all basic objectives encountered across the possible steps a user may take in managing daily finances.

1. **Keep track of your financial habits:** Create, read, update and delete budget entries, or simply navigate through past transactions and learn where all you money goes and comes from.
2. **Filter & search with ease:** Filter entries based on categories and transaction types.
3. **Find everything you need:** Look up entries using key words, such as names, types, categories and notes.
4. **See what's going on:** See statistics at a glance and rewind the previous six months by looking at cash-flow trends.
5. **Manage account & settings:** Make your account your very own and change regional settings.

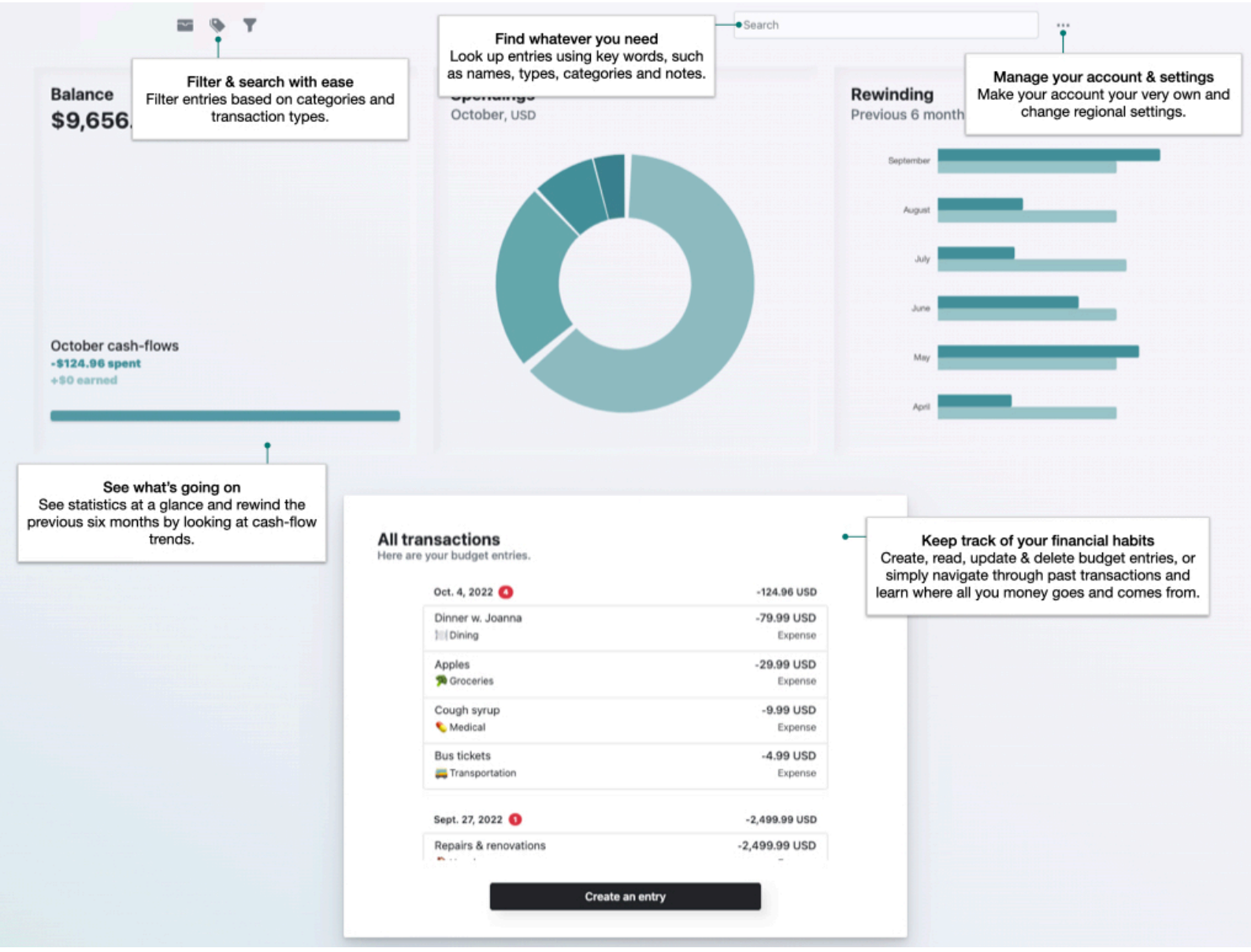
Installation Guide

Follow the steps below to set-up Budgeter in the source-code editor and prepare the dummy database.

```
# Clone the repository and open the project's folder
$ git clone https://github.com/andreicsandor/project-rainier.git


# Set up the virtual environment and install the requirements
$ python3 -m venv venv
$ source venv/bin/activate
$ pip install -r requirements.txt


# Make the initial migrations
$ cd rainier
$ python manage.py makemigrations
$ python manage.py makemigrations authenticator
$ python manage.py makemigrations budgeter
$ python manage.py makemigrations wallet
$ python manage.py migrate authenticator
$ python manage.py migrate budgeter
$ python manage.py migrate wallet
$ python manage.py migrate


# Create the superuser
$ python manage.py createsuperuser


# Load the initial fixtures into the database
$ python manage.py loaddata type
$ python manage.py loaddata category
$ python manage.py loaddata currency
$ python manage.py loaddata profile
$ python manage.py loaddata transaction


# Run the server
$ python manage.py runserver
```

Quick Start Guide

Jump right into Budgeter's dashboard and access all features as per the guidelines below.



Part II

System Documentation

User Story Map

Below is presented the application backlog and each user story which denotes the most typical interactions one can expect while using a budgeting application. We highlight the available functions in the *minimum viable product* (MVP) and planned features for future releases.

USER ACTIVITIES	**Application & settings**		**Transactions**			
USER STORIES	**Interaction with application**	**Settings**	**Entries management**	**Filter & search entries**	**Visualise statistics**
USER TASKS					
Release 1	Manage the app from a desktop web browser	Edit account details	Create a new entry	Apply a quick filter by type	See balance details
	Manage the app from a mobile web browser	Edit preferences	Update & delete an existing entry	Apply a quick filter by category	See current month analysis
	Manage the app from a tablet web browser			Apply advanced filters	See previous 6 months analysis
				Search using basic keywords	
Release 2		Change password	Delete multiple entries	Search using personalised keywords	Select month for spendings card
		Recover password		Apply multiple filters	
Release 3			Set budgets for each category	Apply filters & search simultaneously	See dynamic summary cards based on applied filters
					See budgets analysis

Product Map

The following map is a visual scheme of the information architecture and helps visualise the structure of the application. The main pages of the web application are *Authentication*, *Settings*, *Dashboard* and *Editor*.

Budget tracker

Authentication

Log-in

- Username and password fields
- Sign-up option

Sign-up

- Username, name, e-mail address and password inputs
- Log-in option

Initial set-up

- Currency selection

Settings

Account

- Username, name and e-mail address update

Preferences

- Currency selection

Dashboard

Menu bar

- Type filter
- Category filter
- Advanced filter
- Search bar
- Settings

Header cards

- Balance overview & current month cash-flows
- Current month spendings overview
- Previous 6 months overview

Transactions card

- Budget entries list
- Create new entry functionality

Advanced filter panel

- Type filter
- Cascading category filter
- Date filter

Editor

Create entry

- Select entry type
- Set entry name
- Set transaction amount
- Select date
- Select entry category
- Add note

Update entry

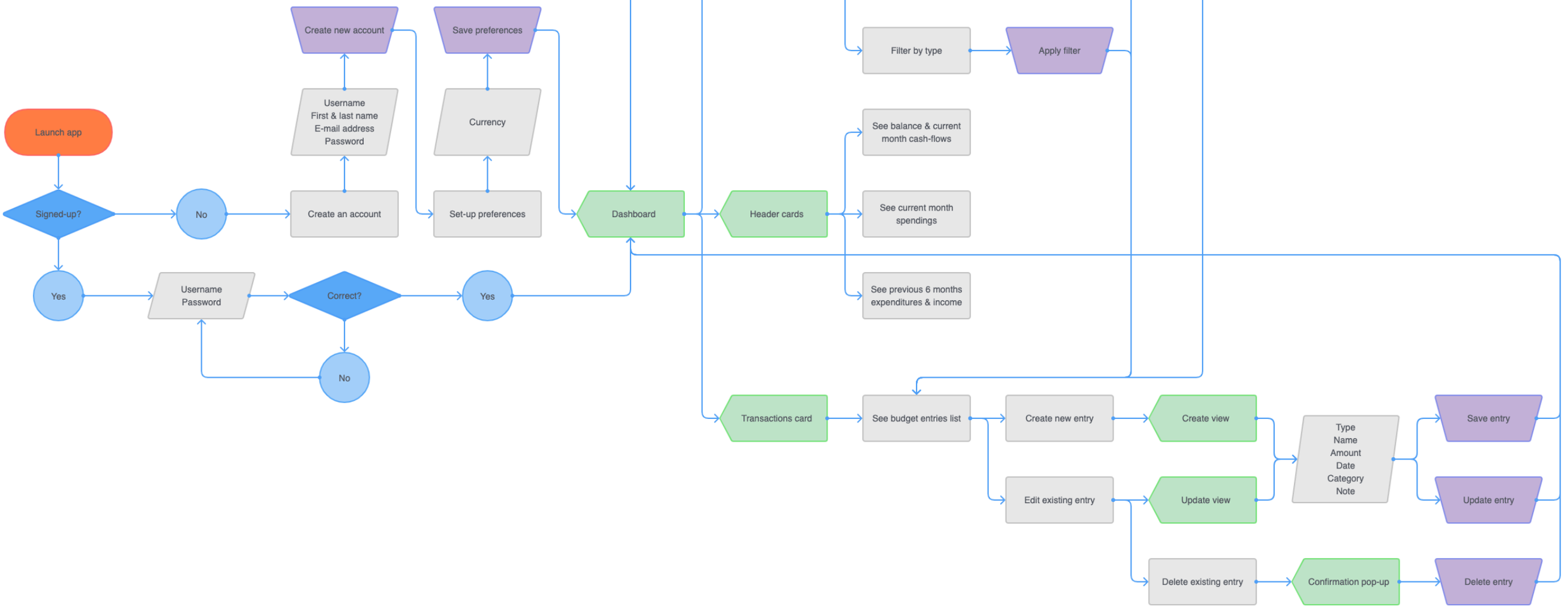
- Change entry type
- Edit entry name
- Edit transaction amount
- Change date
- Change entry category
- Edit note
- Delete functionality

Delete entry

- Confirmation pop-up

User Journey Scheme

This sub-section presents the user flow scheme and depicts all the possible steps a user may take while using the budgeting app. The scheme starts with the *authentication process* and covers the user movement logic across the *dashboard interaction*, *CRUD operations*, *account & preferences management,* and *signing out process*.


Launch app
Signed-up?
No
Create an account
Create new account
Username
First & last name
E-mail address
Password
Yes
Username
Password
Correct?
No
Yes
Set-up preferences
Save preferences
Currency
Dashboard
Menu bar
Expand options
Log-out
Exit app
Edit settings
Account page
Username
First & last name
E-mail address
Save changes
Edit preferences
Preferences page
Currency
Update preferences
Filter by advanced criteria
Advanced filter panel
Apply filters
Search entries
Search keyword
Type
Category
Date
Filter by category
Apply filter
Apply filter
Filter by type
Apply filter
Header cards
See balance & current month cash-flows
See current month spendings
See previous 6 months expenditures & income
Transactions card
See budget entries list
Create new entry
Create view
Edit existing entry
Update view
Delete existing entry
Confirmation pop-up
Type
Name
Amount
Date
Category
Note
Save entry
Update entry
Delete entry

Database Scheme

The project functionalities are factored into three separate apps, the *Authenticator*, *Budgeter* and *Wallet.* Each application serves its specific processes and related scenarios, while also fetching the relevant data from its associated models.

The entire database comprises six models: *User*, *Profile*, *Currency*, *Type*, *Category* and *Transaction*.

User: Related to multiple records in the Transaction table.
Many-to-one relationship between Transaction and User.

Related to individual records in the Profile table.
One-to-one relationship between Profile and User.

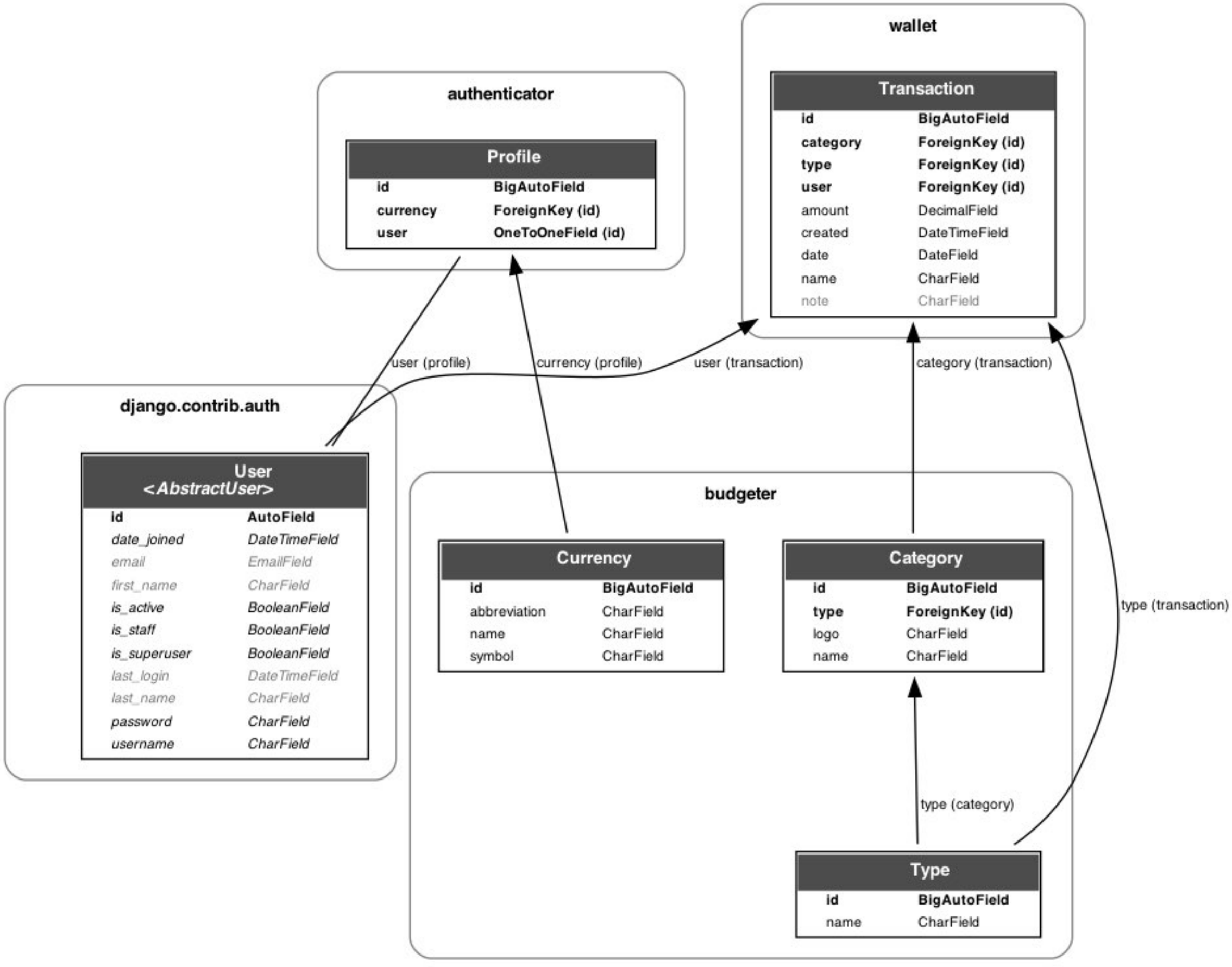
Profile: Extends the base User model and contains user-specific settings, such as preferred currency.
One-to-one relationship between User and Profile.

Currency: Related to multiple records in the Profile table.
Many-to-one relationship between Profile and Currency.

Type: Related to multiple records in the Transaction table.
Many-to-one relationship between Transaction and Type.

Related to multiple records in the Category table.
Many-to-one relationship between Category and Type.

Category: Related to multiple records in the Transaction table.
Many-to-one relationship between Transaction and Category.


wallet
Transaction
id BigAutoField
category ForeignKey (id)
type ForeignKey (id)
user ForeignKey (id)
amount DecimalField
created DateTimeField
date DateField
name CharField
note CharField
authenticator
Profile
id BigAutoField
currency ForeignKey (id)
user OneToOneField (id)
user (profile)
currency (profile)
user (transaction)
category (transaction)
type (transaction)
django.contrib.auth
User
<AbstractUser>
id AutoField
date_joined DateTimeField
email EmailField
first_name CharField
is_active BooleanField
is_staff BooleanField
is_superuser BooleanField
last_login DateTimeField
last_name CharField
password CharField
username CharField
budgeter
Currency
id BigAutoField
abbreviation CharField
name CharField
symbol CharField
Category
id BigAutoField
type ForeignKey (id)
logo CharField
name CharField
type (category)
Type
id BigAutoField
name CharField

Part III

Source Code

This section contains the project's back-end source code for the 3 main applications. Each sub-section takes a look at the forms.py, models.py, views.py and services.py files.

\

Authenticator

forms.py

```
from django import forms
from django.contrib.auth.forms import UserChangeForm, UserCreationForm
from django.contrib.auth.models import User
from authenticator.models import Profile
from budgeter.models import Currency

class SignUpForm(UserCreationForm):
    username = forms.CharField(widget=forms.TextInput(attrs = {"type": "text", "class": "form-
control", "placeholder": "Username"}))
    first_name = forms.CharField(widget=forms.TextInput(attrs = {"type": "text", "class":
"form-control", "placeholder": "First Name"}))
    last_name = forms.CharField(widget=forms.TextInput(attrs = {"type": "text", "class":
"form-control", "placeholder": "Last Name"}))
    email = forms.EmailField(max_length=200, widget=forms.EmailInput(attrs = {"type": "email",
"class": "form-control", "placeholder": "jappleseed@mail.com"}))
    password1 = forms.CharField(max_length=200, widget=forms.PasswordInput(attrs = {"class":
"form-control", "placeholder": "••••••••"}))
    password2 = forms.CharField(max_length=200, widget=forms.PasswordInput(attrs = {"class":
"form-control", "placeholder": "••••••••"}))

    class Meta:
        model = User
        fields = ('username', 'first_name', 'last_name', 'email', 'password1', 'password2')

class LoginForm(forms.Form):
    username = forms.CharField(widget=forms.TextInput(attrs = {"type": "text", "class": "form-
control", "placeholder": "Username"}))
    password = forms.CharField(max_length=200, widget=forms.PasswordInput(attrs = {"class":
"form-control", "placeholder": "••••••••"}))

class UserForm(UserChangeForm):
    username = forms.CharField(widget=forms.TextInput(attrs = {"type": "text", "class": "form-
control", "placeholder": "Username"}))
    first_name = forms.CharField(widget=forms.TextInput(attrs = {"type": "text", "class":
"form-control", "placeholder": "First Name"}))
    last_name = forms.CharField(widget=forms.TextInput(attrs = {"type": "text", "class":
"form-control", "placeholder": "Last Name"}))
    email = forms.EmailField(max_length=200, widget=forms.EmailInput(attrs = {"type": "email",
"class": "form-control", "placeholder": "jappleseed@mail.com"}))

    class Meta:
        model = User
        fields = ('username', 'first_name', 'last_name', 'email')

class ProfileForm(forms.ModelForm):
    currency = forms.ModelChoiceField(queryset=Currency.objects.all(), empty_label="Pick your
prefered currency...", widget=forms.Select(attrs = {"class" : "form-select"}))

    class Meta:
        model = Profile
        fields = ('currency',)
```

models.py

```
from django.db import models
from django.contrib.auth.models import User
from budgeter.models import Currency

class Profile(models.Model):
    """
    Adds custom fields to the parent user class.
    """
    user = models.OneToOneField(User, null=True, on_delete=models.CASCADE)
    currency = models.ForeignKey(Currency, null=True, on_delete=models.CASCADE)

    def __str__(self):
        return f"{self.user}"
```

```
    def ProfileCurrency(self):
        """Retrieves the user's currency of choice."""
        return Currency.CurrencyDetails(self.currency)

    class Meta:
        verbose_name = 'Profile'
        verbose_name_plural = 'Profiles'
```

views.py

```
from django.contrib.auth import authenticate, login, logout
from django.contrib.auth.decorators import login_required
from django.contrib.auth.models import User
from django.core.exceptions import ObjectDoesNotExist
from django.shortcuts import redirect, render
from django.views import View
from authenticator.forms import LoginForm, ProfileForm, SignUpForm, UserForm
from authenticator.models import Profile

class Utilities(View):
    """
    Represents a child class for views.

    Displays the account management, preferences and
    initial settings configuration pages and forms.
    """

    @login_required
    def setup(request):
        """Displays the initial settings configuration page and form."""

        # Prevents the user from revisiting the initial configuration page
        try:
            if Profile.objects.get(user=request.user.id):
                return redirect('.')
        except ObjectDoesNotExist:
            if request.method == "POST":
                form = ProfileForm(request.POST)
                if form.is_valid():
                    obj = form.save(commit=False)
                    obj.user = User.objects.get(pk=request.user.id)
                    obj.save()
                    return redirect('.')
            else:
                form = ProfileForm()

        context = {"form": form}

        return render(request, "configure.html", context)

    @login_required
    def account(request):
        """Displays the account management page and form."""

        # Prevents the user from skipping the initial configuration step.
        try:
            if Profile.objects.get(user=request.user.id):
                pass
        except ObjectDoesNotExist:
            return redirect('/configure')

        user = User.objects.get(pk=request.user.id)
        form = UserForm(instance=user)
        if request.method == "POST":
            form = UserForm(request.POST, instance=user)
            if form.is_valid():
                obj = form.save(commit=False)
                obj.save()
                return redirect('/')

        context = {"form": form}

        return render(request, "account.html", context)

    @login_required
    def preferences(request):
        """Displays the preferences page and form."""
```

```
        # Prevents the user from skipping the initial configuration step.
        try:
            if Profile.objects.get(user=request.user.id):
                pass
        except ObjectDoesNotExist:
            return redirect('/configure')

        user = User.objects.get(pk=request.user.id)
        profile = Profile.objects.get(user=user)
        form = ProfileForm(instance=profile)
        if request.method == "POST":
            form = ProfileForm(request.POST, instance=profile)
            if form.is_valid():
                obj = form.save(commit=False)
                obj.save()
                return redirect('/account')

        context = {"form": form}

        return render(request, "preferences.html", context)

def signup_view(request):
    """Displays the form and creates a new user account."""

    if request.user.is_authenticated:
        return redirect('.')
    elif request.method == "POST":
        form = SignUpForm(request.POST)
        if form.is_valid():
            form.save()
            username = form.cleaned_data['username']
            password = form.cleaned_data['password1']
            user = authenticate(username=username, password=password)
            login(request, user)
            return redirect('/configure')
    else:
        form = SignUpForm()

    context = {"form": form}

    return render(request, "signup.html", context)

def login_view(request):
    """Displays the form and logs in the user."""

    if request.user.is_authenticated:
        return redirect('.')
    elif request.method == "POST":
        form = LoginForm(request.POST)
        if form.is_valid():
            username = form.cleaned_data['username']
            password = form.cleaned_data['password']
            user = authenticate(request, username=username, password=password)
            if user:
                login(request, user)
                return redirect('/')
    else:
        form = LoginForm()

    context = {"form": form}

    return render(request, "login.html", context)

def logout_view(request):
    """Displays the form and logs out the user."""

    logout(request)
    return redirect(‘/‘)
```

Budgeter

models.py

```python
from django.db import models

class Currency(models.Model):
    """
    Represents the preferred currency for transactions.
    """
    name = models.CharField(max_length=50)
    abbreviation = models.CharField(max_length=50)
    symbol = models.CharField(max_length=50)

    def __str__(self):
        return f"{self.symbol} {self.name}, {self.abbreviation}"

    def CurrencyDetails(self):
        """Returns the currency details."""
        return self.abbreviation, self.symbol

    def CurrencyList(self):
        """Retrieves the available currency types."""
        return self.objects.all().order_by('name')

    class Meta:
        verbose_name = 'Currency'
        verbose_name_plural = 'Currencies'


class Type(models.Model):
    """
    Represents the type of the transaction for the individual budget entries.
    """
    name = models.CharField(max_length=50)

    def __str__(self):
        return f"{self.name}"

    def TypeList(self):
        """Retrieves the available transaction types."""
        return self.objects.all().order_by('name')

    class Meta:
        verbose_name = 'Type'
        verbose_name_plural = 'Types'


class Category(models.Model):
    """
    Represents the categories for the individual budget entries.
    Both expenses and income categories are grouped together.
    """
    name = models.CharField(max_length=50)
    logo = models.CharField(max_length=50, default=None)
    type = models.ForeignKey(Type, on_delete=models.CASCADE)

    def __str__(self):
        return f"{self.logo} {self.name}"

    def CategoryList(self):
        """Retrieves the available transaction categories."""
        categories_expenses = self.objects.filter(type="1").order_by('name')
        categories_income = self.objects.filter(type="2").order_by('name')
        return categories_expenses, categories_income

    def CategoryType(self):
        """Returns the category type formatted as a string."""
        return str(self.type)

    class Meta:
        verbose_name = 'Category'
        verbose_name_plural = 'Categories'
```

Wallet

forms.py

```
from django import forms
from wallet.models import Category, Transaction, Type

class TransactionForm(forms.ModelForm):
    type = forms.ModelChoiceField(queryset=Type.objects.all(), empty_label="Select type...",
widget=forms.Select(attrs = {"class" : "form-select"}))
    name = forms.CharField(widget=forms.TextInput(attrs = {"type": "text", "class": "form-
control", "placeholder": "Name"}))
    amount = forms.DecimalField(min_value=0, max_digits=11, decimal_places=2,
widget=forms.NumberInput(attrs = {"type": "number", "class": "form-control", "placeholder":
"Amount"}))
    date = forms.DateField(widget=forms.DateInput(format='%m/%d/%Y', attrs = {"class": "form-
control", "id": "input-date", "name": "input-date"}))
    category = forms.ModelChoiceField(queryset=Type.objects.all(), empty_label="Select
category...", widget=forms.Select(attrs = {"class" : "form-select"}))
    note = forms.CharField(required=False, widget=forms.TextInput(attrs = {"type": "text",
"class": "form-control", "placeholder": "Note"}))

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.fields['category'].queryset = Category.objects.none()

        if 'type' in self.data:
            try:
                type_id = int(self.data.get('type'))
                self.fields['category'].queryset =
Category.objects.filter(type_id=type_id).order_by('name')
            except (ValueError, TypeError):
                pass
        elif self.instance.pk:
            self.fields['category'].queryset =
self.instance.type.category_set.order_by('name')

    class Meta:
        model = Transaction
        fields = ('type', 'name', 'amount', 'date',  'category', 'note')
```

models.py

```
import datetime
from django.contrib.auth.models import User
from django.db import models
from django.utils.timezone import localtime
from budgeter.models import Category, Type

class Transaction(models.Model):
    """
    Represents the individual budget entries that can be either
    expenses or income streams.

    All types of transactions are grouped together and Custom Model
    managers are used to filter between expenses and income.
    """
    user = models.ForeignKey(User, null=True, on_delete=models.CASCADE)
    created = models.DateTimeField(default=localtime)
    date = models.DateField(default=datetime.date.today)
    type = models.ForeignKey(Type, on_delete=models.CASCADE)
    name = models.CharField(max_length=50)
    amount = models.DecimalField(max_digits=11, decimal_places=2)
    category = models.ForeignKey(Category, on_delete=models.CASCADE)
    note = models.CharField(max_length=255, blank=True)

    def __str__(self):
        return f"{self.user} - {self.date} - {self.category}, {self.type} - {self.name},
{self.amount}USD"

    def TransactionMonth(self):
        """Returns the formatted month of the transaction."""
        return self.date.strftime("%B")
```

```
    def TransactionType(self):
        """Returns the type of the transaction formatted as a string."""
        return str(self.type)

    def TransactionCategory(self):
        """Returns the category of the transaction formatted as a string."""
        return str(self.category)

    class Meta:
        verbose_name = 'Transaction'
        verbose_name_plural = 'Transactions'
```

services.py

```
from django.contrib.auth.decorators import login_required
from django.contrib.auth.models import User
from django.core.exceptions import ObjectDoesNotExist
from django.shortcuts import redirect
from authenticator.models import Profile
from budgeter.models import Category, Type
from wallet.models import Transaction

# Retrieves & groups the common and user-specific data

@login_required
def finder(request):
    """Returns the corresponding entries for the applied search & filter criteria."""

    # Unfiltered user-specific entries
    qs = get_entries(request)

    query_type = request.GET.get('input-type')
    query_category_search = request.GET.get('input-category-search')
    query_category = request.GET.get('input-category')
    query_type_advanced = request.GET.get('input-type-advanced')
    query_category_advanced = request.GET.get('input-category-advanced')
    query_date = request.GET.get('input-date')
    query_search = request.GET.get('input-search')

    # Checks the corresponding match for the type filter
    if query_type is not None:
        qs = filter_type(qs, query_type)
    # Checks the corresponding match for the category search input
    if query_category_search is not None:
        qs = search_category(qs, query_category_search)
    # Checks the corresponding match for the category filter
    if query_category is not None:
        qs = filter_category(qs, query_category)
    # Checks the corresponding match for the advanced filter
    if query_type_advanced is None or query_type_advanced == "All":
        if query_category_advanced is None or query_category_advanced == "All":
            if query_date is not None:
                qs = filter_date(qs, query_date)
        else:
            if query_date is not None:
                qs_category = filter_category_advanced(qs, query_category_advanced)
                qs_date = qs = filter_date(qs, query_date)
                qs = qs_category & qs_date
    else:
        if query_category_advanced is None or query_category_advanced == "All":
            if query_date is not None:
                qs_type = filter_type_advanced(qs, query_type_advanced)
                qs_date = filter_date(qs, query_date)
                qs = qs_type & qs_date
        else:
            if query_date is not None:
                qs_type = filter_type_advanced(qs, query_type_advanced)
                qs_category = filter_category_advanced(qs, query_category_advanced)
                qs_date = filter_date(qs, query_date)
                qs = qs_type & qs_category & qs_date

    # Checks the corresponding match for the search input
    if query_search is not None:
        qs = search_all(qs, query_search)

    return qs

@login_required
```

```
def get_entries(request):
    """Retrieves the user's set of entries."""
    user = User.objects.get(pk=request.user.id)
    data = Transaction.objects.filter(user=user).order_by('-date', '-amount', 'name')
    return data

def group_entries(data):
    """Groups the entries into expenses & income."""
    expenses = data.filter(type="1").order_by('-amount')
    income = data.filter(type="2").order_by('-amount')
    return expenses, income

# Searches & filters the user's data

def filter_type(qs, query_type):
    """Filters the entries based on the type criteria."""
    qs = qs.filter(type__name__iexact=query_type)
    return qs

def filter_category(qs, query_category):
    """Filters the entries based on the category criteria."""
    qs = qs.filter(category__name__iexact=query_category)
    return qs

def search_category(qs, query_category_search):
    """Filters the entries based on the category search query."""
    qs = qs.filter(category__name__icontains=query_category_search)
    return qs

def filter_type_advanced(qs, query_type_advanced):
    """Filters the entries based on the advanced type criteria."""
    qs = qs.filter(type__name__iexact=query_type_advanced)
    return qs

def filter_category_advanced(qs, query_category_advanced):
    """Filters the entries based on the advanced category criteria."""
    qs = qs.filter(category__name__iexact=query_category_advanced)
    return qs

def filter_date(qs, query_date):
    """Filters the entries based on the date range."""
    start_date, end_date = format_date(query_date)
    qs = qs.filter(date__range=[start_date, end_date])
    return qs

def search_all(qs, query_search):
    """Filters the entries based on the search query."""
    if check_type(query_search):
        qs = qs.filter(type__name__icontains=query_search)
    elif check_category(query_search):
        qs = qs.filter(category__name__icontains=query_search)
    elif check_note(query_search):
        qs = qs.filter(note__icontains=query_search)
    else:
        qs = qs.filter(name__icontains=query_search)
    return qs

# Validates & checks if the search query parameters are valid

def check_type(param):
    """Validates the query parameters for entry types."""
    types = list(dict.fromkeys([item.name for item in Type.objects.all()]))
    return param.title() in types

def check_category(param):
    """Validates the query parameters for entry categories."""
    categories = list(dict.fromkeys([item.name for item in Category.objects.all()]))
    return param.title() in categories

def check_note(param):
    """Validates the query parameters for entry notes."""
    notes = list(dict.fromkeys([item.note for item in Transaction.objects.all()]))
    text = []
    for note in notes:
        text.extend(note.lower().split())
    return param.lower() in text

def format_date(date):
    """Formats the date into the proper format"""
    # Splits the input into two dates
    elements = date.split(" - ")
```

```
    # Formats the start date
    elements[0] = elements[0].split("/")
    start_date = f"{elements[0][2]}-{elements[0][0]}-{elements[0][1]}"
    # Formats the end date
    elements[1] = elements[1].split("/")
    end_date = f"{elements[1][2]}-{elements[1][0]}-{elements[1][1]}"
    return start_date, end_date

# User & profile configuration tool

def check_configuration(request):
    """Prevents the user from skipping the initial configuration step."""
    try:
        if Profile.objects.get(user=request.user.id):
            pass
    except ObjectDoesNotExist:
        return redirect('/configure')
```

views.py

```
import datetime
from django.contrib.auth.decorators import login_required
from django.contrib.auth.models import User
from django.core.exceptions import ObjectDoesNotExist
from django.shortcuts import redirect, render
from django.views import View
from authenticator.models import Profile
from budgeter.models import Category, Type
from wallet.forms import TransactionForm
from wallet.models import Transaction
from wallet.services import finder, get_entries, group_entries

class Viewer(View):
    """
    Represents a child class for views.

    Displays the entries, queries, user-specific statistics and
    main forms through multiple screens.
    """

    @login_required
    def dashboard(request):
        """Displays the corresponding entries & statistics for the current user."""

        # Prevents the user from skipping the initial configuration step.
        try:
            if Profile.objects.get(user=request.user.id):
                pass
        except ObjectDoesNotExist:
            return redirect('/configure')

        # Common data
        types = Type.TypeList(Type)
        categories_expenses, categories_income = Category.CategoryList(Category)

        # User-specific data
        profile = Profile.objects.get(user=request.user.id)
        currency_short, currency_symbol = Profile.ProfileCurrency(profile)
        entries = get_entries(request)
        expenses, income = group_entries(entries)

        # Filtered entries
        query = finder(request)

        # Computes the user's balance
        balance_total = 0
        for entry in expenses:
            balance_total -= entry.amount
        for entry in income:
            balance_total += entry.amount

        # Counts the no. of entries per day & computes the daily expenditure
        days = list(dict.fromkeys([item.date for item in query]))
        balance_daily = dict.fromkeys(days, 0)
        counter_daily = dict.fromkeys(days, 0)
        for day in days:
            for entry in query:
                if entry.date == day:
```

```
                    counter_daily[day] += 1
                    if entry in expenses:
                        balance_daily[day] -= round(float(entry.amount), 2)
                    if entry in income:
                        balance_daily[day] += round(float(entry.amount), 2)
        statistics = []
        for count, total in zip(counter_daily.values(), balance_daily.values()):
            pair = [count, total]
            statistics.append(pair)
        summary_daily = dict(zip(days, statistics))

        # Computes the absolute expenses & income for the current month
        expenses_current = 0
        income_current = 0
        day_current = datetime.datetime.now()
        month_current = day_current.strftime("%B")
        for entry in expenses:
            if Transaction.TransactionMonth(entry) == month_current:
                expenses_current += entry.amount
        for entry in income:
            if Transaction.TransactionMonth(entry) == month_current:
                income_current += entry.amount

        # Computes the relative expenses & income for the current month
        try:
            expenses_relative = round(expenses_current / (expenses_current + income_current),
2) * 100
        except ZeroDivisionError:
            expenses_relative = 0
        try:
            income_relative = round(income_current / (expenses_current + income_current), 2) *
100
        except ZeroDivisionError:
            income_relative = 0

        # Computes the expenses & income for the previous 6 months
        months_previous = []
        month_current_start = day_current.replace(day=1)
        for _ in range(6):
            months_previous_start = (month_current_start -
datetime.timedelta(days=1)).replace(day=1)
            month_current_start = months_previous_start
            months_previous.append(month_current_start.strftime("%B"))
        expenses_previous = dict.fromkeys(months_previous, 0)
        for month in months_previous:
            for entry in expenses:
                if Transaction.TransactionMonth(entry) == month:
                    expenses_previous[month] += entry.amount
        income_previous = dict.fromkeys(months_previous, 0)
        for month in months_previous:
            for entry in income:
                if Transaction.TransactionMonth(entry) == month:
                    income_previous[month] += entry.amount

        # Generates the chart data for the current expenses overview
        labels_expenses_current = [category.name for category in categories_expenses]
        data_expenses_current = dict.fromkeys(labels_expenses_current, 0)
        for entry in expenses:
            if month_current == Transaction.TransactionMonth(entry):
                # Ignores the category logo
                category = Transaction.TransactionCategory(entry).replace(" ", "")[1:]
                data_expenses_current[category] += entry.amount
        for key, value in dict(data_expenses_current).items():
            if value == 0:
                del data_expenses_current[key]
        labels_expenses_current = list(data_expenses_current.keys())
        values_expenses_current = list(data_expenses_current.values())

        # Generates the chart data for the previous expenses overview
        labels_entries_previous = list(income_previous.keys())
        values_expenses_previous = list(expenses_previous.values())
        values_income_previous = list(income_previous.values())

        context = {
            "types": types,
            "categories_expenses": categories_expenses,
            "categories_income": categories_income,
            "currency_short": currency_short,
            "currency_symbol": currency_symbol,
            "entries": query,
            "balance_total": balance_total,
```

```
        "summary_daily": summary_daily,
        "month_current": month_current,
        "expenses_current": expenses_current,
        "income_current": income_current,
        "expenses_relative": expenses_relative,
        "income_relative": income_relative,
        "expenses_previous": expenses_previous,
        "income_previous": income_previous,
        "labels_expenses_current": labels_expenses_current,
        "values_expenses_current": values_expenses_current,
        "labels_entries_previous": labels_entries_previous,
        "values_expenses_previous": values_expenses_previous,
        "values_income_previous": values_income_previous
    }

    return render(request, "home.html", context)

@login_required
def creator(request):
    """Creates a new transaction entry."""

    # Prevents the user from skipping the initial configuration step.
    try:
        if Profile.objects.get(user=request.user.id):
            pass
    except ObjectDoesNotExist:
        return redirect('/configure')

    # User-specific data
    profile = Profile.objects.get(user=request.user.id)
    currency_short, currency_symbol = Profile.ProfileCurrency(profile)

    if request.method == "POST":
        form = TransactionForm(request.POST)
        if form.is_valid():
            obj = form.save(commit=False)
            obj.user = User.objects.get(pk=request.user.id)
            obj.save()
            return redirect('/')
    else:
        form = TransactionForm()

    context = {
        "form": form,
        "currency_short": currency_short,
        "currency_symbol": currency_symbol
    }

    return render(request, "create.html", context)

@login_required
def editor(request, pk):
    """Edit an existing transaction entry."""

    # Prevents the user from skipping the initial configuration step.
    try:
        if Profile.objects.get(user=request.user.id):
            pass
    except ObjectDoesNotExist:
        return redirect('/configure')

    # User-specific data
    profile = Profile.objects.get(user=request.user.id)
    currency_short, currency_symbol = Profile.ProfileCurrency(profile)

    # Pulls details of the active entry
    entry = Transaction.objects.get(id=pk)

    form = TransactionForm(instance=entry)
    if request.method == "POST":
        form = TransactionForm(request.POST, instance=entry)
        if form.is_valid():
            obj = form.save(commit=False)
            obj.save()
            return redirect('/')

    context = {
        "form": form,
        "id": pk,
        "currency_short": currency_short,
        "currency_symbol": currency_symbol
```

```
        }

        return render(request, "edit.html", context)

    @login_required
    def eraser(request, pk):
        """Delete an existing transaction entry."""

        # Prevents the user from skipping the initial configuration step.
        try:
            if Profile.objects.get(user=request.user.id):
                pass
        except ObjectDoesNotExist:
            return redirect('/configure')

        entry = Transaction.objects.get(id=pk)
        if request.method == "POST":
            entry.delete()
            return redirect('/')

        context = {"id": pk}

        return render(request, "delete.html", context)

def categories_view(request):
    """
    Loads the corresponding categories for each type
    of transaction in the create entry window.
    """
    type_id = request.GET.get('type_id')
    categories = Category.objects.filter(type_id=type_id).all()

    return render(request, 'categories.html', {'categories': categories})
```

Developed & Written by **Andrei-Cristian Șandor**, 2022