# Web プログラミング期末レポート

202210016 髙橋伊織

# 金銭管理 Web アプリケーション Manay

## 1 概要

Manay は、普段の金銭管理を簡単に行うことができる Web アプリである。

このアプリケーションでは、 ユーザが自分の収入と支出を追跡・管理することができる。簡単な操作を通して、入出金の登録・履歴の閲覧・残高の推移の確認が行える。

このアプリケーションは、以下の URL からアクセスできる。http://20.243.83.92:3000/

## 2 特徴と工夫した点

### 2.1 フレームワーク

今回のアプリケーションでは、開発をより簡単・効率化するため、Ruby on Rails というフレームワークを使用した。MVC モデルを利用することで、頭の中を整理しながらコーディングをすることができた。

### 2.2 モデル

このアプリケーションでは、以下のようにモデルを定義した。

- User
  - email:string
  - encrypted_password:string
  - current_balance:decimal

- Transaction
  - user_id:integer
  - amount:decimal
  - transaction_type:string
  - created_at:datetime
  - after_balance:decimal

ここで、balance（残高）属性が User/Transaction のどちらにもあることに疑問を持つかもしれない。

これは、以下の機能を実装するためである。

- 現在の残高を表示する
- これまでの残高の推移をグラフで表示する

current_balance のみ存在する場合は、1 ユーザにつき 1 つの残高しか持てないため、残高の推移をグラフで表示することができない。 また、after_balance のみ存在する場合は、残高を表示するために全ての Transaction を走査し、計算する必要がある。

そのため、どちらのモデルにも現在の残高を保持する実装をすることにした。以下は、`app/models/transaction.rb` のコードの一部である。

```
1  class Transaction < ApplicationRecord
2    # 省略
3    after_save :update_user_balance
4
5    private
6
7    def update_user_balance
8
9      #省略
10
11     new_balance = transaction_type == 'payment' ? user.curr↵
        ↳ent_balance - amount : user.current_balance + amount
12     user.update(current_balance: new_balance)
13     update_columns(after_balance: new_balance)
14   end
```

```
15
16  end
```

`after_save`コールバックを使用し、transaction によって増減した balance を、`current_balance`・`after_balance`の両方に格納している。これにより、上記要件が実装できた。

## 2.3 アカウント作成 / ログイン

このアプリケーションは、ユーザごとにサービスを提供する必要があった。今回は、devise という gem を使用した。devise を用いることで、ユーザ登録・認証系の機能を簡単に実装することができた。

特に、`app/conrollers/application_controller.rb`に以下のコードを追加することで、アプリケーションにおける全てのアクションに対して、ログイン状態を要求することができた。

```
1  class ApplicationController < ActionController::Base
2    before_action :authenticate_user!
3  end
```

また、アカウント作成時に現在所持している金額を入力する機能を実装した。これは、以下の 2 点を狙ったものである。

- アカウント作成時に`after_balance`にデータを入れることで、トップページの折れ線グラフでレンダリングエラーを発生させない
- 現在の所持金を、入出金取引と同様に登録するのは直観的ではない

`current_balance`の初期登録は`app/views/devise/registrations/new.html.erb`によって実装されている。

```
1  # 省略
2  <div class="field">
3    <%= f.label :current_balance, "残高登録" %>
4    <%= f.number_field :current_balance, min: 0 %>
5  </div>
6  # 省略
```

また、`after_balance`の初期登録は`app/controllers/transactions_controller.rb`によって実装されている。

```
# 省略
  def create
    super do |user|
    if user.persisted?
      initial_balance = params[:user][:current_balance].to_f
      user.transactions.create(amount: initial_balance, description: "初期登録", transaction_date: Time.current, after_balance: initial_balance)
    end
    end
  end
# 省略
```

この時、`after_balance`の登録より前に`current_balance`の登録が行われるため、例外処理を行う必要があった。以下は`app/models/tranaction.rb`での例外処理の内容である。

```
# 省略
  def update_user_balance

    return if description == '初期登録'

    new_balance = transaction_type == 'payment' ? user.current_balance - amount : user.current_balance + amount
# 省略
```

## 2.4 入出金処理

このアプリケーションでは、入出金処理を行うために、Transaction モデルを用いた。以下は、`app/controllers/transactions_controller.rb`のコードの一部である。

```ruby
# 省略
  def new
    @transaction = Transaction.new
  end

  def create
    @transaction = current_user.transactions.new(transactio
    ↳ n_params)
    if @transaction.save
      redirect_to root_path
    else
      # 失敗時の処理
      Rails.logger.debug @transaction.errors.full_messages
      render :new
    end
  end

  private

  def transaction_params
    params.require(:transaction).permit(:amount, :descripti
    ↳ on, "transaction_date(1i)", "transaction_date(2i)",
    ↳ "transaction_date(3i)", "transaction_date(4i)", "tra
    ↳ nsaction_date(5i)", :transaction_type)
  end
# 省略
```

この時、入金と支払いを正負の数で入力するシステムは、ユーザーフレンドリーではないと考えた。そのため、入金と支払いを区別するために、`transaction_type`属性を用いた。この属性では、`payment`と`deposit`の 2 つの値を取る。erb ファイルでは、チェックボックスで選択してもらい、`after_balance`の計算は以下のように場合わけを行なった。

```ruby
# 省略
new_balance = transaction_type == 'payment' ? user.current_b
```

```
  ↪alance - amount : user.current_balance + amount
# 省略
```

`app/views/transactions/new.html.erb`で実装した画面が以下である。

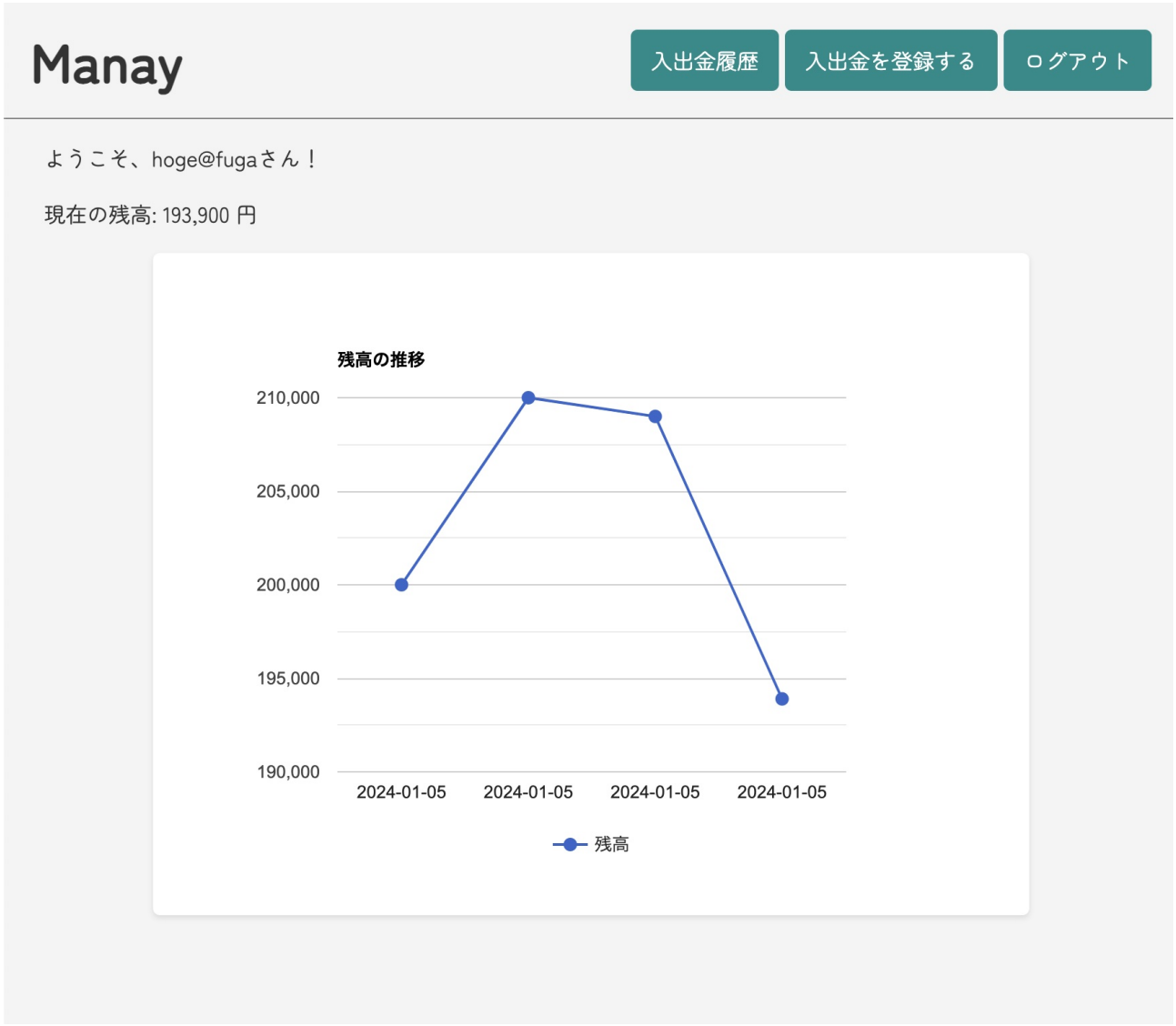

## 2.5 残高推移グラフ

このアプリケーションでは、残高の推移をグラフで表示する機能を実装した。以下は、`app/controllers/home_controller.rb`のコードの一部である。

```
# 省略
  <div id="balance_chart" style="width: 70dvw; height: 50dv
   ↪ h;"></div>
  <script type="text/javascript" src="https://www.gstatic.c
   ↪ om/charts/loader.js"></script>
```

```
4     <script type="text/javascript">
5       google.charts.load('current', {'packages':['corechart'] ↵
        ↳ ]});
6       google.charts.setOnLoadCallback(drawChart);
7
8       function drawChart() {
9         var data = google.visualization.arrayToDataTable([
10          ['日付', '残高'],
11          <% current_user.transactions.each do |transaction| ↵
            ↳ %>
12            ['<%= transaction.created_at.strftime("%Y-%m-%d") ↵
              ↳ %>', <%= transaction.after_balance %>],
13          <% end %>
14        ]);
15
16        var options = {
17          title: '残高の推移',
18          pointSize:8,
19          legend: { position: 'bottom' }
20        };
21
22        var chart = new google.visualization.LineChart(docume ↵
          ↳ nt.getElementById('balance_chart'));
23
24        chart.draw(data, options)
25      }
26    </script>
27  # 省略
```

これによって、グラフを表示することができた。実際の例は以下の通りである。




## 2.6 2 つのアルゴリズムの違い

アルゴリズムの手順は以下の通りである。

- 全ての辺を重みの昇順にソートする
- 重みの小さい辺から順に、その辺で接続されている 2 つの頂点が同じ木に属していない場合（閉路を形成しない場合）、その辺を最小全域木に加える
- 全ての頂点が最小全域木に含まれるまで、2 番目の操作を繰り返す

2 つのアルゴリズムの主な違いは、最小全域木を選ぶ基準にある。Prim のアルゴリズムは頂点をベースに木を構築していくのに対し、Kruskal のアルゴリズムは辺をベースに木を構築していく。