

# Cryptocurrency Trading Simulator

Eamon Goonan, 19765759 & Mudiaga Jerry Dortie, 19339753

20/03/2023

<b>1. Overview</b>	<b>2</b>
<b>2. Motivation</b>	<b>3</b>
<b>3. Research</b>	<b>3</b>
3.1 Web Application - Backend	3
3.2 Web Application - Frontend	4
3.3 Binance API - Real-time Cryptocurrency Data	5
<b>4. Design</b>	<b>6</b>
4.1 System Architecture	6
4.2 User Accounts and Authentication	8
<b>5. Implementation</b>	<b>16</b>
5.1 User Accounts and Authentication	16
5.2 Charting Live data	18
5.3 Tracking Traders	21
5.4 Trading Simulator	23
<b>6. Problems and Solutions</b>	<b>27</b>
<b>7. Testing</b>	<b>28</b>
7.0 Git + Ci/CD usage	28
7.2 User Testing	32
<b>8. Bibliography</b>	<b>33</b>

# 1. Overview

Our project entailed the development of a “Cryptocurrency Trading Simulator”. Our goal was to create a platform where beginner cryptocurrency traders could experiment with cryptocurrency trading strategies in a risk-free simulated environment.

We decided to create a web application as the platform for our Cryptocurrency Trading Simulator. While we initially planned to use Django as a backend framework and React as a frontend framework, we eventually decided to forego the Django frontend and make the web application purely Django.

The web application makes calls to the Binance API to obtain real-time data such as live cryptocurrency exchange rates. The user is also provided with live price charts, which are obtained from third party sources.

We succeeded in delivering a functional cryptocurrency trading simulator. The user may register an account, which will be given a default balance of \$7000. Users can then go to the “Trade” page, where they have the option to invest in several crypto currencies. The user is provided with a live price chart, to more accurately assess the value of their chosen currency.

Once the user has purchased a crypto currency with their given balance, they can go to their “Portfolio” page, where their open and closed positions are displayed, alongside relevant data pertaining to these positions.

The “Home” page also provides a dashboard, where upon logging in, the user is provided with useful statistics such as the amount of open trades they hold, their all-time PNL and all-time ROI.

We also developed an additional “Following” page, where users are provided with a selection of top Binance.com traders. Users may find it useful to track the trades of these users, with it being possible to mirror their trading strategies using the trading simulator.

## 2. Motivation

We both had limited cryptocurrency trading experience prior to this project. We were aware of the risks beginners face as they begin trading. Cryptocurrencies are a volatile investment, and with little knowledge or experience of the markets, beginner traders often lose a lot of money.

For this reason, we decided to create a risk-free environment for beginners to experiment with new trading strategies. Users of our web app are provided with an initial sum of mock USD, which they may then use to trade in crypto currencies. Users are shown clearly the progress of their trades, with statistics such as PNL and ROI.

We hope that users of our application can gain effective experience in cryptocurrency trading, which can then be applied to real-world trading.

## 3. Research

### 3.1 Web Application - Backend

*Describe the requirements of this component and how this influenced your choice of languages/frameworks etc.. for the component*

There are several reasons we chose to build a **web application** for our project:

1. Accessibility: Web applications are accessible from anywhere with an internet connection, making them easy to use and convenient for users.
2. Scalability: Web applications can handle a large number of users and can be scaled up or down as needed.
3. Compatibility: Building a web application can be more effective than building a native mobile application or desktop application because it can be developed once and run on multiple platforms.
4. Easy maintenance: Web applications can be updated and maintained easily as they are centrally hosted and can be updated without requiring users to download new versions.
5. Integration with other systems: Web applications can easily integrate with other systems and APIs, making it easier to incorporate additional functionality into the application.
6. Analytics and tracking: Web applications can be easily tracked and analysed to gather valuable data on user behaviour, preferences, and other metrics.

We felt that building a web application was the right choice, as we wished to create an accessible application that can be easily updated and maintained.

We chose **Django** as our web application framework, as our CA298 module familiarised us with the process of full-stack Django development.

The Django framework boasts many advantages:

1. **Rapid development:** Django is designed to make development fast and efficient. It provides a lot of built-in features and functionality that can be used to quickly create web applications.
2. **Scalability:** Django can handle a large amount of traffic and data, making it a good choice for building scalable applications.
3. **Security:** Django has built-in security features, such as protection against SQL injection attacks, cross-site scripting attacks, and clickjacking, making it a secure platform for building web applications.
4. **Reusability:** Django's code is designed to be reusable, which means that developers can write code once and use it in multiple parts of the application, making development faster and more efficient.
5. **Community support:** Django has a large and active community of developers who are constantly contributing new features, libraries, and tools to improve the platform.
6. **Compatibility:** Django is compatible with a variety of databases, web servers, and operating systems, making it a flexible platform that can be used in many different environments.

Overall, we find Django to be a powerful and versatile platform that offers many benefits for building web applications.

## 3.2 Web Application - Frontend

We initially chose to use **React** as a front-end framework, to work in conjunction with the Django backend. Although we had no prior experience with React, we were aware of its popularity in industry, and believed the project would be a suitable time to gain experience with it.

Django and React are compatible and can be used together to build full-stack web applications. Django is a server-side web framework that is used to handle requests and responses, manage data, and provide API endpoints. React, on the other hand, is a client-side JavaScript library that is used to create dynamic user interfaces.

As is common when building web applications with Django and React, We used Django as the backend API and React as the frontend framework. This means that Django handles requests from the client-side application and returns JSON data, which can be consumed by the React application to render the UI.

There are several libraries and tools available to facilitate the integration of Django and React, including Django REST framework, which provides a simple and powerful toolkit for building APIs with Django, and create-react-app, which provides a set of tools and configurations for building React applications.

Despite the high compatibility between Django and React, our lack of experience with the framework became a roadblock in the project. For this reason we decided to make the project frontend purely Django based.

We were aware that Django comes with some built-in tools for generating HTML pages, such as template rendering and form handling, and we were familiar with these tools from our CA298 module.

Django can still be used to serve HTML pages and static assets such as images, CSS, and JavaScript files. This can be useful for building simple web applications or for rendering server-side components of more complex applications. We are aware that, for more complex frontend development, it is common to use a separate frontend framework or library, such as React. Although, for the purposes of this project, which entails a more simple web application, Django alone would suffice.

### **3.3 Binance API - Real-time Cryptocurrency Data**

Binance provides a comprehensive API that allows developers to access a wide range of market data and trading information in real-time. This includes real-time price data, order book data, trading history, and more.

To access real-time market data through the Binance API, we created an API key on the Binance website, and used that key to authenticate API requests. We then used the API to make requests for real-time market data, such as the current price of a particular cryptocurrency or the order book data for a particular trading pair.

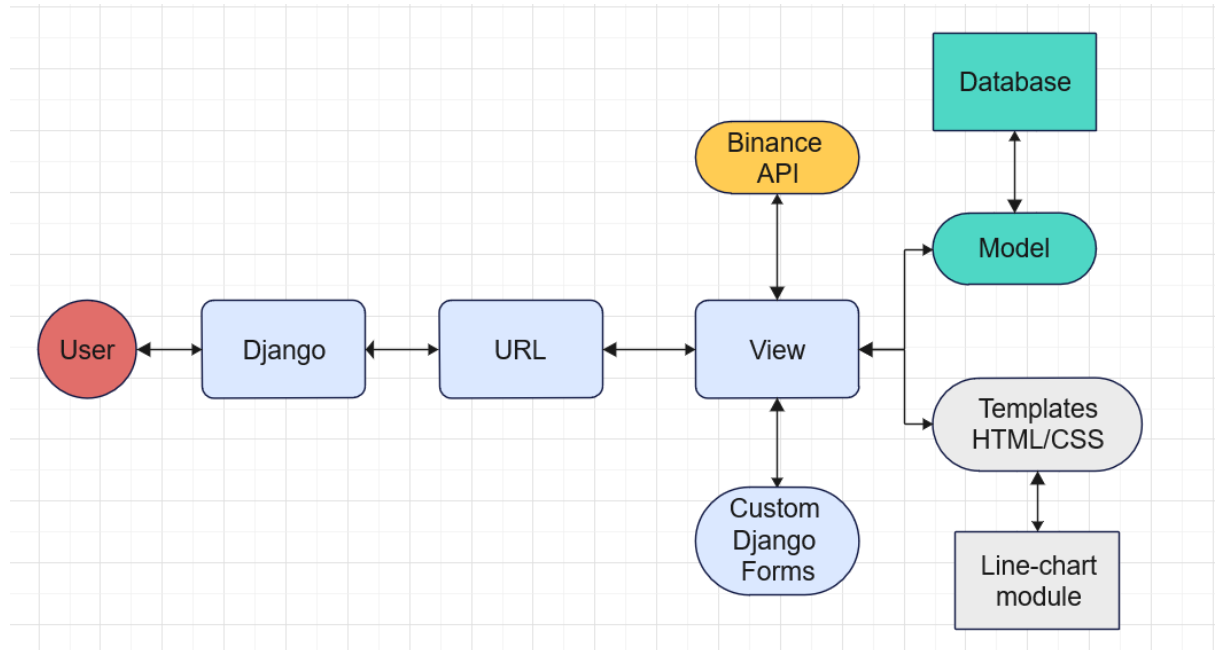
The API provides the web application with necessary data, such as real-time cryptocurrency prices, as well as trading information relating to Binance.com traders.

## 4. Design

In this section we will detail the project system architecture

### 4.1 System Architecture

Django System Architecture Diagram:



The system architecture for our cryptocurrency trading simulator involves multiple layers of the application stack, including the web server, application server, database, and front-end client.

Here's a brief overview of each layer:

1. Web server: The web server layer is responsible for handling incoming requests from clients and routing them to the appropriate application server.
2. Application server: The application server layer is where the Django framework runs and processes incoming requests. This layer includes the Django middleware and views, which handle authentication, user input validation, and data retrieval from the database.
3. Database: The database layer is responsible for storing and retrieving data related to the cryptocurrency trading simulator project. Our project database system is SQLite, which is included in Django.
4. Front-end client: The front-end client layer is responsible for displaying the user interface to clients and handling user input. We used Django templates

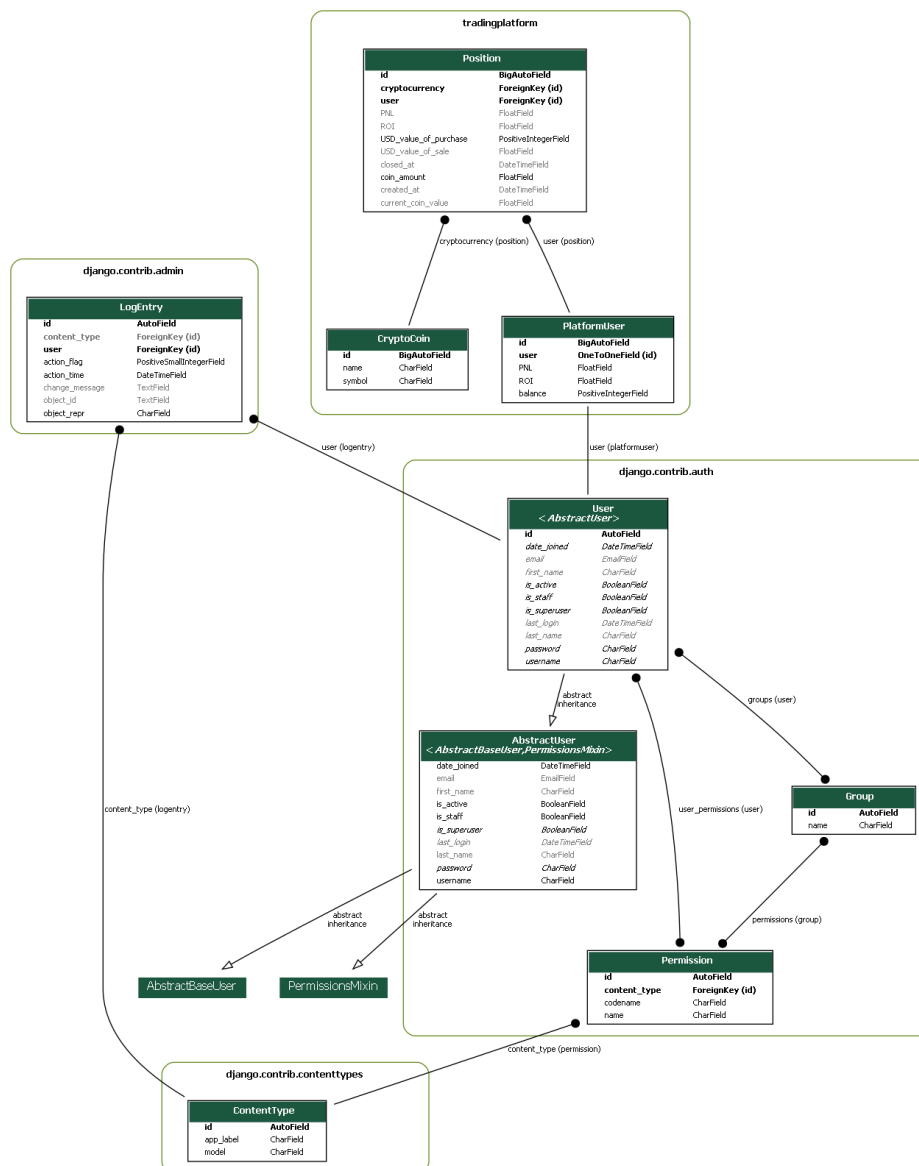
The project architecture includes several key components:

1. Django models: Models are used to define the database schema and handle data storage and retrieval. We defined models for cryptocurrencies, users and user trade positions.
2. Django views: Views are used to process incoming requests and generate responses. Views can also handle authentication and data validation.
3. Django templates: Templates are used to generate HTML pages based on data retrieved from the database.
4. Django middleware: Middleware is used to intercept incoming requests and perform additional processing, such as logging or authentication.
5. Custom forms: We created custom Model Forms for the Position model, which allow the user to create positions or sell them. These forms are in the forms.py file in the "tradingplatform" app. We instantiated these forms in our views.py file and passed them into our HTML templates via context dictionary.
6. Binance API: The "tradingplatform" app contains a python file that calls to a Binance endpoint to obtain the live prices of crypto currencies.
7. Back end threading: the get\_binance\_user\_datav2 file contains the threading module so the two functions can run concurrently with each other and with the front end.

## 4.2 User Accounts and Authentication

*Details specific to component X , how it works, Use case diagrams, system sequence diagrams, data flow diagrams – each picture must be accompanied with explanatory text*

Below is a UML class diagram for the Django models, showing the relation between the app and the User Authentication System.



Well-designed databases are pivotal in the development of a Django application. We took care in designing the normalised database you see above. “tradingplatform” is the primary app in the Django project, while the other models you see here are derived from the Django User Authentication system



To handle user authentication, we created the “members” app within our Django project. This app contains the login and registration forms, and the views which pass these forms into Django HTML templates.

The default Django User Model is a built-in authentication system in Django that provides the functionality for managing users and their authentication. It is a model that defines the fields and methods necessary for user authentication, authorization, and management.

The User model is defined in the Django's built-in auth application and provides fields for storing user information such as username, password, email, first name, and last name. The model also includes methods for authentication and authorization, such as checking if a user is active or staff.

By default, the User model includes the following fields:

- username: a unique identifier for the user
- password: a hashed version of the user's password
- email: the email address associated with the user
- first\_name: the user's first name
- last\_name: the user's last name
- is\_active: a boolean flag indicating if the user is active or not
- is\_staff: a boolean flag indicating if the user has staff-level access

Django provides a built-in authentication system that allows developers to easily handle user authentication and authorization in their web applications. Here's how Django user authentication works:

1. User logs in: The user submits their username and password through a login form.
2. Django verifies the credentials: Django checks the submitted credentials against the username and password stored in the database.
3. User is authenticated: If the credentials are valid, Django creates a session for the user and logs them in.
4. User is authorised: Once the user is logged in, Django checks their permissions to determine if they have access to the requested resources or views. If the user is authorised, they can proceed with their requested action.
5. User logs out: When the user logs out, Django destroys their session and logs them out.

The Django authentication system uses cookies and sessions to keep track of the user's authentication status. Once a user is authenticated, a session ID is created and stored in a cookie in the user's browser. This session ID is used to identify the user and their authentication status on subsequent requests.

As shown in the UML diagram, we have created the model “PlatformUser” which extends the default Django User model. This model was created for the purpose of adding extra fields to the default Django User model (i.e. Balance, PNL, ROI). It has a “user” field, with a one-to-one relationship to the Django User Model. Below is a snippet showing the custom user model. See also the Django signal, which instantiates the custom user model any time a User is registered.

```
17 class PlatformUser(models.Model):
18     user = models.OneToOneField(User, on_delete=models.CASCADE)
19     balance = models.PositiveIntegerField(default=7000)
20     ROI = models.FloatField(default=0)
21     PNL = models.FloatField(default=0)
22
23     def __str__(self):
24         return str(self.user)
25
26     # Triggers instance of custom user model whenever a Django User is registered.
27     @receiver(post_save, sender=User)
28     def create_platform_user(sender, instance, created, **kwargs):
29         if created:
30             PlatformUser.objects.create(user=instance)
```

The user’s balance is always displayed in the Navbar, and user statistics such as PNL, ROI and a count of the user’s trades, are displayed on the homepage.

## 4.3 Simulated Trading System

We created a “Cryptoin” model, with instances containing the name and symbol of the cryptocurrencies we make available to trade.

The “Positions” model is at the heart of the trading system. It contains 2 foreign key fields: “user” and “cryptocurrency”, which take from the PlatformUser and CryptoCoin models, respectively.

We created a Django ModelForm based on the positions model, which is then passed into the trading page HTML via Django view.

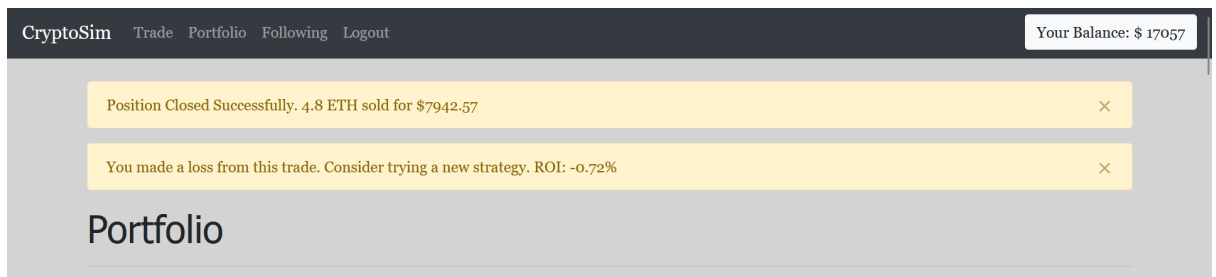
By submitting this form, the user can create a new Position instance that is tied to their account.

## 4.4 User Interface

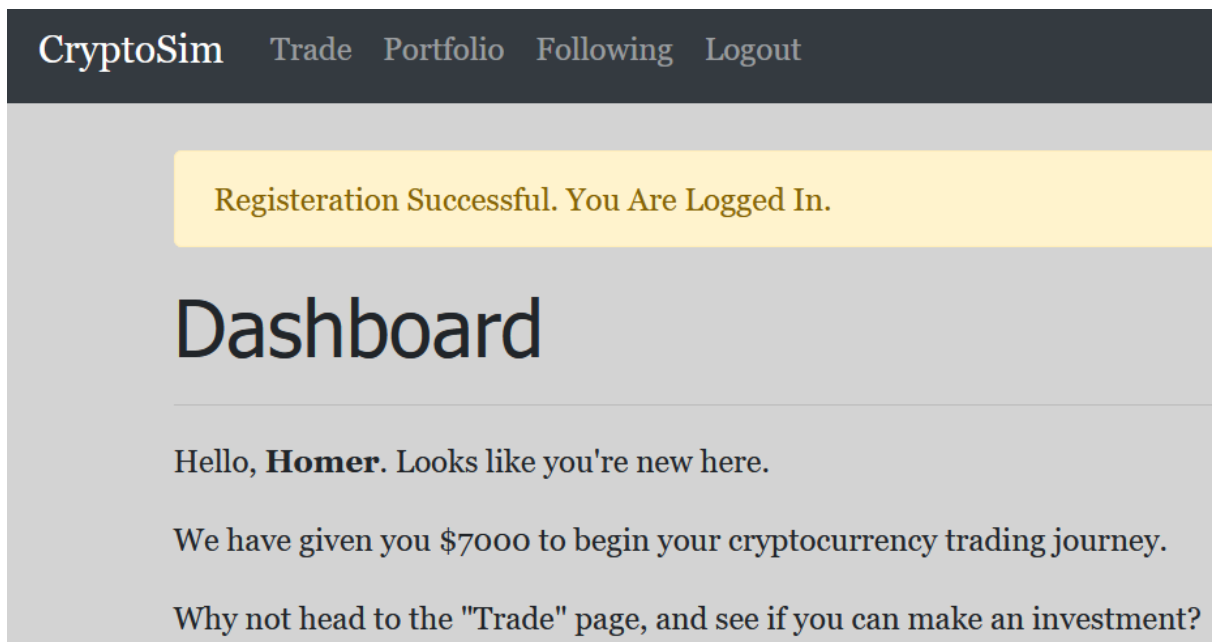
Our interface was designed with Schneiderman's "Eight Golden Rules of Interface Design" in mind.

We kept the website consistent with the use of the same fonts, colours and bootstrap styling. We ensured the Navbar was visible on every page of the website.

Users are offered informative feedback based on their trading. See below for example.



Dialogs are designed to usher new users towards making their first trades. See below for example.



We also consulted Jakob Nielsen's "Ten Usability Heuristics" in designing our interface.

Errors are prevented in forms through the use of Model constraints. See below for example form error.

# Trade

Select a Cryptocurrency, and how much of your money you would like to invest.

Cryptocurrency:  Purchase Value:

Please select a valid value. The two nearest valid values are 100 and 101.

**BTC/USD**

We strived for a minimalist design in our website. For example, the “Login” and “Register” navbar buttons disappear once the user is logged in, as a means of decluttering the interface. See navbar below.

CryptoSim Trade Portfolio Following Logout Your Balance: \$ 9115

We will now walk you through the front-end UI page-by-page.

## 1. Homepage (not logged in)

CryptoSim Login Register

## About Us...

Hello, beginner trader! Welcome to CryptoSim.  
We are a simulated Cryptocurrency Trading Platform.  
We offer a risk-free environment for you to experiment with cryptocurrency investment strategies.


Please register or login to start your crypto-trading journey.

CryptoSim: "The Cryptocurrency Trading Simulator"

127.0.0.1:8000

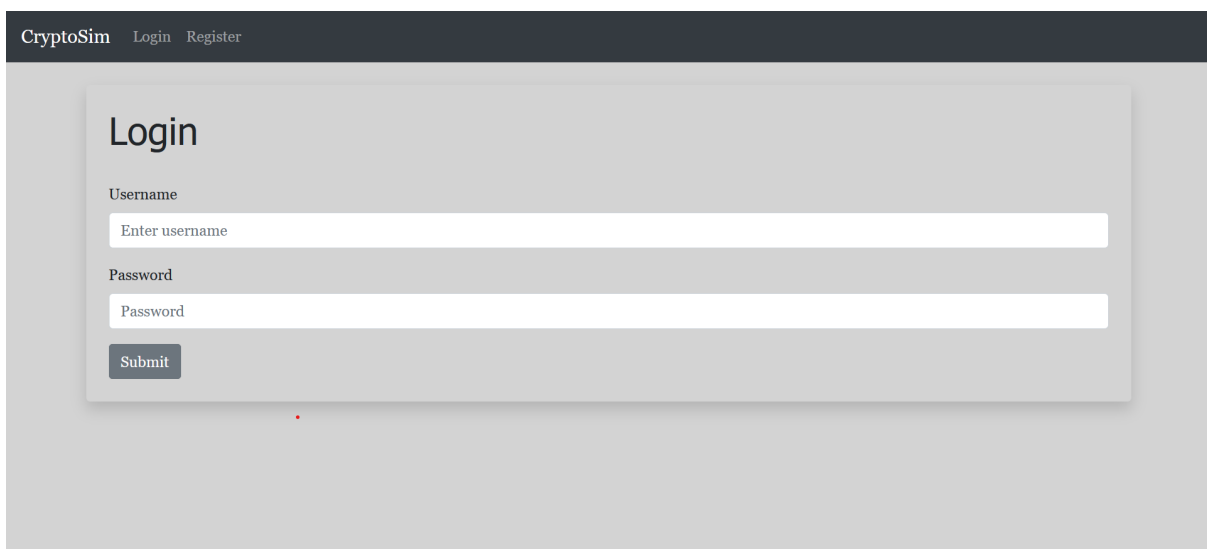
Here the user is presented with the option to login or register.

## 2. Login/Register Pages



The screenshot shows the 'Register' page of the 'CryptoSim' application. The page has a dark header with the application name and links to 'Login' and 'Register'. The main content area is light gray and contains a white registration form. The form has the title 'Register' and includes input fields for 'Username', 'First name', 'Last name', and 'Email'. Below the 'Email' field is a 'Password' field. A small red dot is visible next to the 'Username' field. At the bottom left of the page, there is a URL: 127.0.0.1:8000/members/login\_user

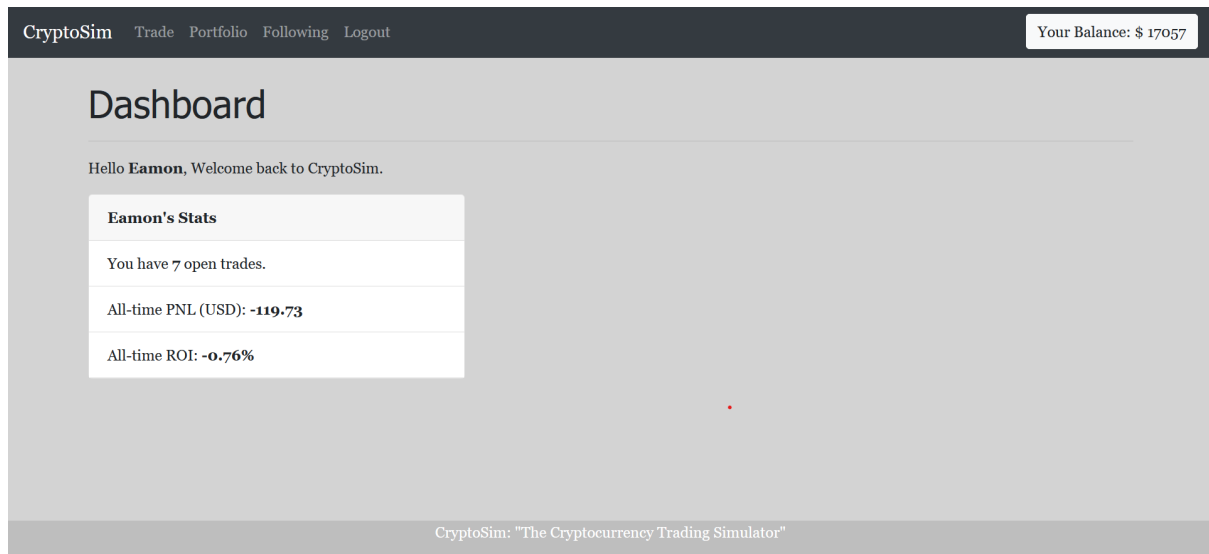
The default Django user registration form is passed in to the register page template through views.py. We added optional fields such as first name, so we could address the user by their name and not their username in the homepage.



The screenshot shows the 'Login' page of the 'CryptoSim' application. The page has a dark header with the application name and links to 'Login' and 'Register'. The main content area is light gray and contains a white login form. The form has the title 'Login' and includes input fields for 'Username' and 'Password'. Below the 'Password' field is a 'Submit' button. A small red dot is visible below the form.

The default Django user login form is passed in to the register page template through views.py.

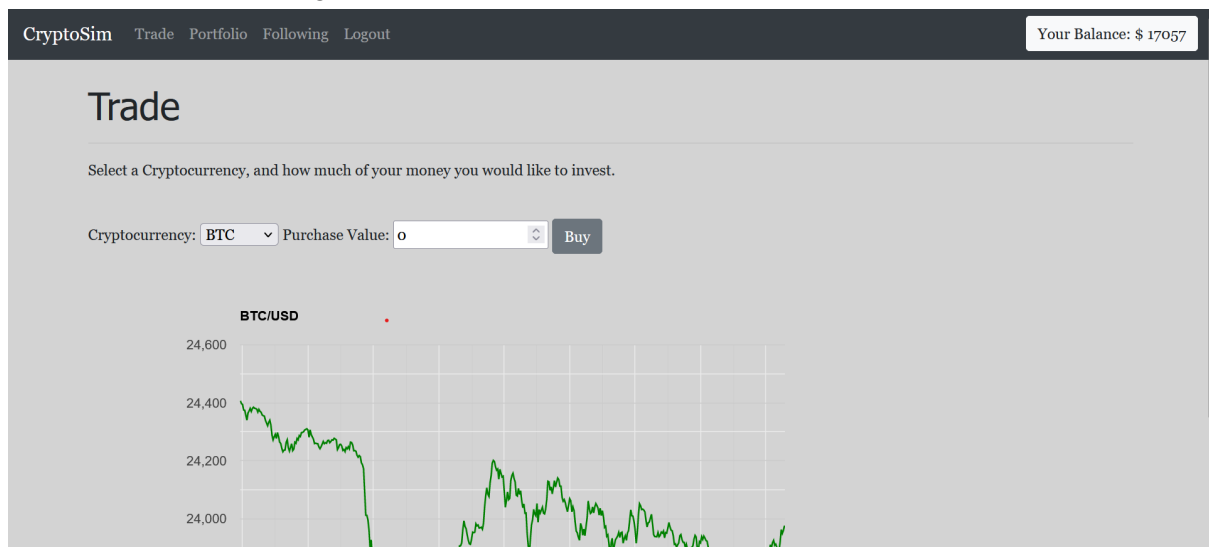
### 3. Homepage (logged in)



Once logged in, the user is redirected to the homepage, where they can now view simple statistics regarding their trading history.

The nav bar now displays links for the Trade, Portfolio, Following pages. The links for logging in and registration have been replaced with the logout link.

### 4. Trade page



Here, the user may select from several cryptocurrencies as well as the amount they would like to purchase. The user is also provided with a chart, which displays live prices for the currency they have selected from the dropdown.

## 5. Portfolio page

CryptoSimTradePortfolioFollowingLogout

Your Balance: \$ 17057

### Portfolio

#### Open Positions

Analyse or sell your open positions.

Coin: ETH

Coin Amount: 4.798119

Purchased for: \$8000

Current Value: \$7963.87

ROI: -0.45% ↓

Purchase Date: Feb. 21, 2023, 8:38 p.m.

Sell

Coin: ETH

Coin Amount: 4.799213

Purchased for: \$8000

Current Value: \$7965.69

ROI: -0.43% ↓

Purchase Date: Feb. 21, 2023, 8:38 p.m.

Sell

#### Closed Positions

Analyse your closed positions.

Coin: ETH

Coin Amount: 4.798148

Purchased for: \$8000

Sold for: \$7942.57

PNL: \$-57.43

ROI: -0.72% ↓

Purchase Date: Feb. 21, 2023, 8:38 p.m.

Sell Date: Feb. 23, 2023, 3:31 p.m.

Coin: LTC

Coin Amount: 5.340739

Purchased for: \$500

Sold for: \$494.07

PNL: \$-5.93

ROI: -1.19% ↓

Purchase Date: Feb. 21, 2023, 1:02 p.m.

Sell Date: Feb. 21, 2023, 9:34 p.m.

Coin: ETH

Coin: ETH

On the portfolio page, users can view their open and closed positions. We display important statistics regarding each trade. Users have the option to sell any of their open trades.

## 5. Implementation

In this section we will detail how the components of the application interact.

### 5.1 User Accounts and Authentication

The below image shows our 'forms.py' file in the members app, which contains all code for user authentication for the project.

User authentication in our project is handled by the default Django Authentication System. This includes the default Django User model and Registration form, which you can see in the import statements.

You can see that we added the fields email, first name and last name to the registration form.

```
1 from django.contrib.auth.forms import UserCreationForm
2 from django.contrib.auth.models import User
3 from django import forms
4
5
6 class RegisterUserForm(UserCreationForm):
7     email = forms.EmailField(widget=forms.EmailInput(attrs={'class': 'form-control'}))
8     first_name = forms.CharField(max_length=20, widget=forms.TextInput(attrs={'class': 'form-control'}))
9     last_name = forms.CharField(max_length=20, widget=forms.TextInput(attrs={'class': 'form-control'}))
10
11     class Meta:
12         model = User
13         fields = ('username', 'first_name', 'last_name', 'email', 'password1', 'password2')
14
15     def __init__(self, *args, **kwargs):
16         super(RegisterUserForm, self).__init__(*args, **kwargs)
17
18         self.fields['username'].widget.attrs['class'] = 'form-control'
19         self.fields['password1'].widget.attrs['class'] = 'form-control'
20         self.fields['password2'].widget.attrs['class'] = 'form-control'
```



We then instantiated this form in members/views.py. The form is then passed into the register.html template.

```
30 def register_user(request):
31     if request.method == "POST":
32         form = RegisterUserForm(request.POST)
33         if form.is_valid():
34             form.save()
35             username = form.cleaned_data['username']
36             password = form.cleaned_data['password1']
37             user = authenticate(username=username, password=password)
38             balance = 10000
39             login(request, user)
40             messages.success(request, "Registration Successful. You Are Logged In.")
41             return redirect('tradingplatform:index')
42         else:
43             form = RegisterUserForm()
44
45     return render(request, 'authentication/register.html', {
46         'form': form,
47     })
```

When the user submits the form, an instance of the User model is created. They are then logged in and redirected to the homepage.

Here you can see the register.html template, where the form is passed in.

```
4     {% if form.errors %}
5     <div class="alert alert-warning alert-dismissible fade show" role="alert">
6         There was an error in your form.
7         <button type="button" class="close" data-dismiss="alert" aria-label="Close">
8             <span aria-hidden="true"></span></button>
9     </div>
10    {% endif %}
11
12    <div class="shadow p-4 mb-5 bg-body rounded">
13        <h1>Register</h1>
14        <br/>
15
16        <form action="{% url 'members:register' %}" method=POST>
17            {% csrf_token %}
18
19            {{ form.as_p }}
20
21            <input type="submit" value="Submit" class="btn btn-secondary">
22        </form>
23    </div>
```

We created another view for logging in.

```
7 def login_user(request):
8     if request.method == "POST":
9         username = request.POST['username']
10        password = request.POST['password']
11        user = authenticate(request, username=username, password=password)
12        if user is not None:
13            login(request, user)
14            return redirect('tradingplatform:index')
15
16        else:
17            messages.success(request, "There Was An Error Logging In, Try Again...")
18            return redirect('members:login')
```

Here is the Post method in the template which handles the users input:

```
4 <div class="shadow p-4 mb-5 bg-body rounded">
5     <h1>Login</h1>
6     <br/>
7
8     <form action="" method="POST">
9         {% csrf_token %}
10
11
12     <div class="form-group">
13         <label for="exampleInputEmail1">Username</label>
14         <input type="text" class="form-control" id="exampleInputEmail1" name="username" aria-describedby="emailHelp" placeholder="Enter username">
15     </div>
16
17     <div class="form-group">
18         <label for="exampleInputPassword1">Password</label>
19         <input type="password" class="form-control" id="exampleInputPassword1" name="password" placeholder="Password">
20     </div>
21
22     <input type="submit" value="Submit" class="btn btn-secondary">
23 </form>
```

The user's input is passed back to the views.py file, where the login and password is then authenticated. If the credentials are valid, the user is redirected to the homepage. Otherwise, they receive an error message.

## 5.2 Charting Live data

Firstly, we initialised the chartData variable as an empty array to hold the historical price data. We also initialised the curr variable with the default cryptocurrency symbol 'BTC'. We created the drawChart() function to take an optional curr parameter and specified the cryptocurrency symbol to be charted. The ccxt.binance() object was created to interact with the Binance exchange's API. We used the fetchOHLCV() method to call on this object to retrieve the historical price data for the specified cryptocurrency symbol (\$curr)/USDT with a time interval of 1 minute ('1m'). The console.log() statement was also used to print the extracted data to the console so we could see if it was functioning correctly. We then created a DataTable object using the google.visualization.DataTable() constructor, and added historical price data to the table using the addRows() method. The data.addColumn() method was used to add the column headers for the timestamp and price data. We set the

chart options in the options object, so the chart could have a specific title, legend, size, colour, line width, and background colour.

```
var chartData = [];
window.curr = 'BTC';
function drawChart(curr='BTC') {
  // Initialize the Binance exchange object
  var exchange = new ccxt.binance();

  // Get the historical data
  exchange.fetchOHLCV(`${curr}/USDT`, '1m').then((ohlcvs) => {
    // Extract the timestamp and close price
    for (var i = 0; i < ohlcvs.length; i++) {
      chartData.push([new Date(ohlcvs[i][0]), ohlcvs[i][4]]);
      console.log([new Date(ohlcvs[i][0]), ohlcvs[i][4]]);
    }

    // Create the data table
    var data = new google.visualization.DataTable();
    data.addColumn('datetime', 'Time');
    data.addColumn('number', 'Price');
    data.addRows(chartData);

    // Set the chart options
    var options = {
      title: `${curr}/USD`,
      legend: 'none',
      width: 900,
      height: 500,
      colors: ['green'],
      lineWidth: 1.7,
      backgroundColor: 'transparent'
    };
  });
}
```

We defined the `updateChart()` function and used the `ccxt.binance()` exchange object to fetch the latest OHLCV (open, high, low, close, volume) data for the currency pair (`window.curr`), with a time interval of 1 minute. We extracted the timestamp and the close price of the latest data points and added it to the `chartData` array. Next, we updated the data object with the new data point using the `addRow()` method, and updated the options object with the new title (including the updated currency pair). We then created a new chart object and we updated the data and options passed the `draw()` method to update the chart with the new data.

```
// Create the chart object
var chart = new google.visualization.LineChart(document.getElementById('chart_div'));

// Draw the chart
chart.draw(data, options);

// Define the function to update the chart
function updateChart() {
  exchange.fetchOHLCV(`${window.curr}/USDT`, '1m').then((ohlcv) => {
    // Extract the timestamp and close price
    var timestamp = new Date(ohlcv[ohlcv.length - 1][0]);
    var close = ohlcv[ohlcv.length - 1][4];

    // Add the new data point to the chart data
    chartData.push([timestamp, close]);

    // Update the data table
    data.addRow([timestamp, close]);
    console.log([timestamp, close]);

    // Set the chart options
    var options = {
      title: `${window.curr}/USD`,
      legend: 'none',
      width: 900,
      height: 500,
      colors: ['green'],
      lineWidth: 1.7,
      backgroundColor: 'transparent'
    };

    // Create the chart object
    //var chart = new google.visualization.LineChart(document.getElementById('chart_div'));

    // Draw the chart
    chart.draw(data, options);
  });
}
```

## 5.3 Tracking Traders

We established a connection to the API using the `http.client.HTTPSConnection` method with the API endpoint URL as the argument. The function sends a GET request to the API with the user ID passed as a parameter to the function. The headers are included in the request as well. The response from the API is obtained using the `getresponse()` method on the connection object. The response data is then read and stored in the `performance_data` variable. The `performance_data` was then converted from a byte string to a JSON object using the `json.loads()` method and assigned to the `performance` variable to make the data readable. The function then prints the performance data.

```
def get_binance_user_data(user_id):
    conn = http.client.HTTPSConnection("binance-futures-leaderboard1.p.rapidapi.com")

    headers = {
        'X-RapidAPI-Key': "467157cd05mshb970c2bf96194acp1f1317jsn55233c04915d",
        'X-RapidAPI-Host': "binance-futures-leaderboard1.p.rapidapi.com"
    }

    conn.request("GET", "/v1/get0therPerformance?encryptedUid=" + user_id, headers=headers)

    response = conn.getresponse()
    performance_data = response.read()

    performance = json.loads(performance_data.decode("utf-8"))

    print(performance)
```

We designed this function to run in an infinite loop in a thread. We update the trader's position data every minute until the program is terminated. Similar to the previous code snippet, the function establishes a connection to the Binance Futures API using the `http.client.HTTPSConnection` method and sets up the authentication headers. We added a while loop to check if the trader still has an open position and it executes indefinitely, updating the positions data while waiting for 60 seconds before checking again.

```

def get_user_position_data(user_id):
    conn = http.client.HTTPSConnection("binance-futures-leaderboard1.p.rapidapi.com")

    headers = {
        'X-RapidAPI-Key': '467157cd05mshb970c2bf96194acp1f1317jsn55233c04915d',
        'X-RapidAPI-Host': "binance-futures-leaderboard1.p.rapidapi.com"
    }

    conn.request("GET", "/v2/getTraderPositions?encryptedUid=" + user_id, headers=headers)

    response = conn.getresponse()
    data = response.read()

    positions = json.loads(data.decode("utf-8"))

    perpetual = positions['data'][0]['positions']['perpetual']

    while True:
        if perpetual is not None:
            print(positions)

            time.sleep(60) # wait for 60 seconds before checking again

```

We added this thread code so that it runs when the Python script is executed. We created two threads, thread1 and thread2, which run the two functions. The target argument specifies the function to be run in the thread, and the args argument specifies the arguments to be passed to the function. We passed the user ID as an argument to both functions. The start() method is called on both threads to begin their execution simultaneously and The join() method is then called on both threads, which waits for each thread to finish its execution before the program exits. We added both of these methods to ensure that both threads complete their tasks before the program terminates.

## 5.4 Trading Simulator

Here is the view for the homepage:

```
15 def index(request):
16     new_user = True
17     if request.user.is_authenticated:
18         platform_user = PlatformUser.objects.filter(user=request.user.id)
19         logged_in_user = platform_user[0]
20         user_positions = Position.objects.filter(user=logged_in_user)
21         open_positions = user_positions.filter(closed_at=None)
22         closed_positions = user_positions.filter(closed_at__isnull=False)
23         if closed_positions.count() != 0:
24             logged_in_user.PNL = get_user_all_time_pnl(logged_in_user)
25             logged_in_user.ROI = get_user_all_time_roi(logged_in_user)
26             logged_in_user.save()
27         if open_positions.count() != 0 or closed_positions.count() != 0:
28             new_user = False
29     else:
30         platform_user = None
31         open_positions = None
32
33     context = {'platform_user': platform_user, 'open_positions': open_positions, 'new_user': new_user}
34     return render(request, 'index.html', context)
```

PlatformUser is our extension of the Django user model. It contains useful fields such as balance, ROI and PNL. Firstly, we filter through the PlatformUser model with the user ID of the logged in user. This PlatformUser instance is stored in the logged\_in\_user variable.

We then filter through the user's positions, storing open and closed positions in different variables. These variables are passed into the html template through the context dictionary. The homepage template, shown below, shows the user statistics being passed in via context dictionary.

```
23 <h1>Dashboard</h1>
24 <hr/>
25
26 {% if new_user %}
27 <p>Hello, <b>{{ user.first_name }}</b>. Looks like you're new here.</p>
28 <p>We have given you $7000 to begin your cryptocurrency trading journey.</p>
29 <p>Why not head to the "Trade" page, and see if you can make an investment?</p>
30 {% else %}
31 <p>Hello <b>{{ user.first_name }}</b>, Welcome back to CryptoSim.</p>
32 {% endif %}
33
34 <div class="card" style="...">
35     <div class="card-header">
36         <b>{{ user.first_name }}'s Stats</b> {{ position.cryptocurrency }}
37     </div>
38     {% for user in platform_user %}
39     <ul class="list-group list-group-flush">
40         <li class="list-group-item">You have <b>{{ open_positions.count }}</b> open trades.</li>
41         <li class="list-group-item">All-time PNL (USD): <b>{{ user.PNL|floatformat:2 }}</b></li>
42         <li class="list-group-item">All-time ROI: <b>{{ user.ROI|floatformat:2 }}%</b></li>
43     </ul>
44     {% endfor %}
```

Below is the view for the Trade page. Similar to the homepage, we first obtain the PlatformUser instance of the logged in user by filtering through the model with the current user's id.

```
37 def trade(request):
38     platform_user = PlatformUser.objects.filter(user=request.user.id)
39     logged_in_user = platform_user[0]
40     if request.method == "POST":
41         form = TradeForm(request.POST)
42         if form.is_valid():
43             new_position = form.save(commit=False)
44             new_position.user = logged_in_user
45             new_position.coin_amount = usd_coin_exchange(new_position.USD_value_of_purchase,
46                                                         str(new_position.cryptocurrency))
47             new_balance = logged_in_user.balance - new_position.USD_value_of_purchase
48             logged_in_user.balance = new_balance
49             new_position.save()
50             logged_in_user.save()
51             messages.success(request, "Trade Executed Successfully. " + str(round(new_position.coin_amount, 2)) +
52                             str(new_position.cryptocurrency) + " purchased for $" +
53                             str(round(new_position.USD_value_of_purchase, 2)))
54             messages.success(request, "Check Your Portfolio for Updates on this Trade.")
55             return HttpResponseRedirect("/trade")
56     else:
57         form = TradeForm
58     context = {'platform_user': platform_user, 'form': form}
59     return render(request, 'trade.html', context)
```

The user balance is set to 7000 by default in models.py. Whenever a trade is made, as seen above, the user balance is updated as the value of their purchase is subtracted from their balance.



The trade form we defined in forms.py is then instantiated and passed into the HTML via the context dictionary. See the trade form below:

```
7 class TradeForm(ModelForm):
8
9     Eamon Goonan
10    class Meta:
11        model = Position
12        fields = ('cryptocurrency', 'USD_value_of_purchase')
13
14    Eamon Goonan
15
16    class SellForm(ModelForm):
17
18        Eamon Goonan
19        class Meta:
20            model = Position
21            fields = ()
```

When the user selects a cryptocurrency to invest in, and the dollar value of the amount they would like to purchase, these values are passed in to the `usd_coin_exchange` function:

```
5 def get_coin_price(coin_symbol):
6     key = "https://api.binance.com/api/v3/ticker/price?symbol=" + coin_symbol + "USDT"
7     # requesting data from url
8     data = requests.get(key)
9     data = data.json()
10    # print(f"{data['symbol']} price is {data['price']}")
11    return data['price']
12
13
14    get_coin_price("BTC")
15
16    Eamon Goonan
17
18    def usd_coin_exchange(dollar_amount, coin_symbol):
19        coin_amount = dollar_amount / float(get_coin_price(coin_symbol))
20        return coin_amount
```

This function calls a Binance API endpoint that retrieves the current exchange rate between the given cryptocurrency and USDT. USDT is a cryptocurrency stablecoin, the value of which

is pegged to the US dollar. This exchange rate is then used to calculate the amount of the coin the user has purchased. This same function is also used to track the current value of user positions, against the value at the time of purchase.

Below is the portfolio page view:

```
62 def portfolio(request):
63     platform_user = PlatformUser.objects.filter(user=request.user.id)
64     logged_in_user = platform_user[0]
65     user_positions = Position.objects.filter(user=logged_in_user)
66     open_positions = user_positions.filter(closed_at=None).order_by('-created_at')
67     closed_positions = user_positions.filter(closed_at__isnull=False).order_by('-closed_at')
68     for position in open_positions:
69         position.current_coin_value = coin_usd_exchange(position.coin_amount, str(position.cryptocurrency))
70         position.ROI = (position.current_coin_value -
71                        position.USD_value_of_purchase) / position.USD_value_of_purchase * 100
72         position.save()
73
74     if request.method == 'POST':
75         sold_position = Position.objects.get(id=request.POST['position_id'])
76         form = SellForm(instance=sold_position)
77         sold_position = form.save(commit=False)
78         sold_position.USD_value_of_sale = sold_position.current_coin_value
79         sold_position.closed_at = datetime.datetime.now()
80         sold_position.PNL = sold_position.USD_value_of_sale - sold_position.USD_value_of_purchase
81         sold_position.ROI = sold_position.PNL / sold_position.USD_value_of_purchase * 100
82
83         new_balance = logged_in_user.balance + sold_position.USD_value_of_sale
84         logged_in_user.balance = new_balance
85
86         sold_position.save()
87         logged_in_user.save()
88         if sold_position.PNL > 0:
89             profit_loss_message = "Well done! You made a profit from this trade. "
90         else:
91             profit_loss_message = "You made a loss from this trade. Consider trying a new strategy. "
92         messages.success(request, "Position Closed Successfully. " + str(round(sold_position.coin_amount, 2)) +
93                          " " + str(sold_position.cryptocurrency) +
94                          " sold for $" + str(round(sold_position.USD_value_of_sale, 2)))
95         messages.success(request, profit_loss_message + "ROI: " + str(round(sold_position.ROI, 2)) + "%")
96         return HttpResponseRedirect("/portfolio")
97     else:
98         form = SellForm
99
100     context = {'platform_user': platform_user, 'open_positions': open_positions,
101               'closed_positions': closed_positions, 'form': form}
102     return render(request, 'portfolio.html', context)
```

User balance is updated when a position is sold, with the value of the sale being added to the balance field of the PlatformUser instance.

This view filters through a user's positions and passes a list of the user's closed positions and open positions into the portfolio HTML template.

Everytime the user accesses the portfolio page, the current value of their positions is refreshed.

The homepage displays the aggregate ROI and PNL of a user. These statistics are calculated with the following functions:

```
4 def get_user_all_time_pnl(user):
5     user_closed_positions = Position.objects.filter(user=user, closed_at__isnull=False)
6     all_time_pnl = 0
7     for position in user_closed_positions:
8         all_time_pnl += position.PNL
9
10    return all_time_pnl
11
12    Eamon Goonan
13 def get_user_all_time_roi(user):
14     user_closed_positions = Position.objects.filter(user=user, closed_at__isnull=False)
15     # COI = Cost of Investment
16     all_time_coi = 0
17     for position in user_closed_positions:
18         all_time_coi += position.USD_value_of_purchase
19     all_time_roi = user.PNL / all_time_coi * 100
20
21    return all_time_roi
```

These functions simply take the current user instance as an argument, and loop through their closed positions, counting and aggregating their PNL and ROI. This data is then returned and sent to the homepage HTML.

## 6. Problems and Solutions

### 6.1 Tracking the traders

Tracking the traders was one of the biggest challenges we faced while creating our crypto trading simulator. Initially, we thought scraping the data directly from the binance website would be the easiest solution. However, the website's complex structure made the scraping process extremely difficult. We spent hours browsing the internet and experimenting with various scraping tools before finally finding a reliable API that gave us access to all the data we needed. This allowed us to track the traders' activity accurately and efficiently, which was crucial for creating a realistic simulation.

### 6.2 Finding accurate coin data

Finding accurate and reliable coin data was another significant hurdle we had to overcome. We needed to have access to up-to-date data on a wide range of coins to create a realistic

trading experience. Initially, we tried using various APIs and scraping tools, but we found that most of them provided incomplete or inaccurate data. After several weeks of searching, we finally found a database that was easy to manipulate and had accurate data on all the coins we needed. This database was a game-changer for us, as it made it possible to create a realistic and engaging crypto trading experience.

## **6.3 Charting the data**

Charting the coin data was a vital aspect of our trading simulator. Initially, we used Matplotlib to create the charts, but we quickly realised that it wasn't the best option for creating live charts. We wanted our users to see real-time changes in the coin prices, which required a charting tool that could update quickly and accurately. After some research, we discovered that Google Charts was an excellent option for live charts. It was easy to use, had a wide range of customization options, and updated quickly, making it perfect for our needs. Switching to Google Charts allowed us to create engaging and informative charts that helped our users make informed trading decisions.

## 7. Testing

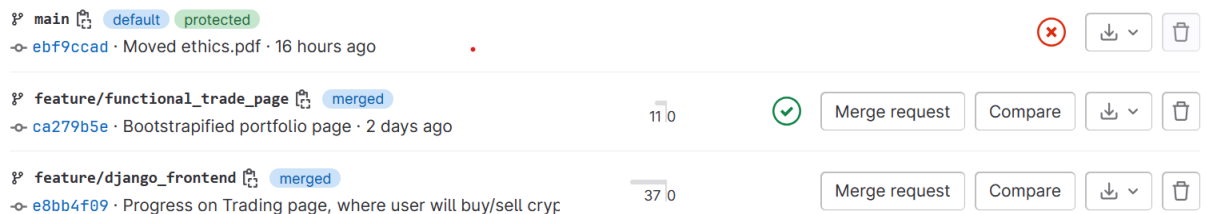
For all kinds of tests include a text paragraph explaining the tests run (how were they run ) + any screenshots / results

### 7.0 Git + Ci/CD usage

Talk about your use of Git/branches + ci/CD , give screenshots of branches

We created feature branches when working on project components. We found Git branches are incredibly useful because they allowed us to work on multiple features or fixes simultaneously without disrupting the main codebase or each other's work.

Below is a list of project branches.























Talk about your workflow for CI/Cd , diagram of this workflow + screenshots of results

We added a Django-tailored .yaml file to our project from the GitLab repository settings. We then added a build stage with scripts that run migrations. Additionally, we wrote a test stage with scripts that run the tests contained in each app.

See below for .yaml content.

```
59
60
61 migrations:
62   stage: build
63   script:
64     - python3 ./code/criptotradingsimulator/manage.py makemigrations
65     - python3 ./code/criptotradingsimulator/manage.py migrate
66
67
68
69 django-tests:
70   stage: test
71   script:
72     # The MYSQL user only gets permissions for MYSQL_DB, so Django can't create a test database.
73     # - echo "GRANT ALL on *.* to '${MYSQL_USER}';" | mysql -u root --password="${MYSQL_ROOT_PASSWORD}"
74     -h mysql
75     # use python3 explicitly. see https://wiki.ubuntu.com/Python/3
76     # - python3 ./code/criptotradingsimulator/manage.py
77     - python3 ./code/criptotradingsimulator/manage.py test members
78     - python3 ./code/criptotradingsimulator/manage.py test tradingplatform
```

The pipeline runs everytime commits are pushed, as you can see below.

<div>passed</div> <div>00:02:47 21 hours ago</div>	Added unit/integration tests #38092 main d1fa3908	  	
<div>passed</div> <div>00:02:52 1 day ago</div>	Updates to UI for more intuitive user exp... #38051 main 9ed641b0	  	
<div>passed</div> <div>00:02:51 1 day ago</div>	Added ethics.pdf to repo #38014 main f3be3056	  	
<div>passed</div> <div>00:02:47 1 day ago</div>	Added ROI to sold positions in portfolio #38004 main 39bdf916	  	
<div>passed</div> <div>00:02:37 2 days ago</div>	btc line chart added #37987 main f9098edb	  	

The below snippet shows the pipeline running test scripts.

Search job log

937 Ran 4 tests in 0.530s

938 OK

939 Destroying test database for alias 'default'...

940 `$ python3 ./code/cryptotradingsimulator/manage.py test tradingplatform`

941 Creating test database for alias 'default'...

942 Found 5 test(s).

943 System check identified no issues (0 silenced).

944 .....

945 -----

946 Ran 5 tests in 0.007s

947 OK

948 Destroying test database for alias 'default'...

950 Saving cache for successful job 00:00

951 Creating cache default-protected...

952 WARNING: ~/.cache/pip/: no matching files. Ensure that the artifact path is relative to the working directory

953 Archive is up to date!

954 Created cache

956 Cleaning up project directory and file based variables 00:01

958 Job succeeded

django-tests

Duration: 1 minute 26 seconds

Finished: 21 hours ago

Queued: 3 seconds

Timeout: 20m (from runner)

Runner: #23 (96c96812) @gitlab-runner

Commit d1fa3908

















Added unit/integration tests

Pipeline #38092 for main

test

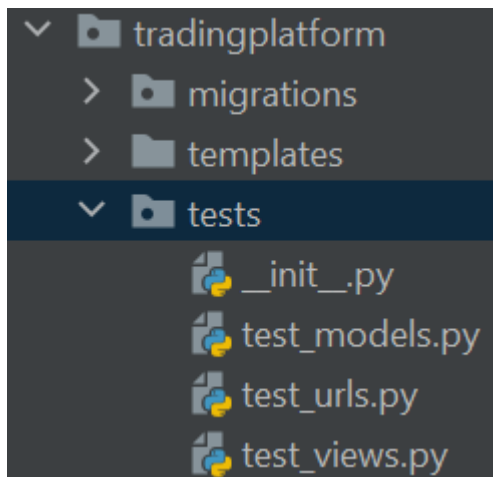
→ django-tests

The below commit history exemplifies our CI/CD workflow:

07 Feb, 2023 2 commits		
	<b>Merge branch 'main' of https://gitlab.computing.dcu.ie/dortiem2/2023-ca326-mdortie-tradingbot</b> Eamon Goonan authored Feb 07, 2023	67b0da63  
	<b>Created serializers for user login and registration models</b> Eamon Goonan authored Feb 07, 2023	9084bcad  
02 Feb, 2023 1 commit		
	<b>Replace plot.py with working btc live chart</b> Mudiaga Jerry Dortie authored Feb 02, 2023	78da9e83  
30 Jan, 2023 2 commits		
	<b>Created gitlab pipeline</b> Eamon Goonan authored Jan 30, 2023	0e7e86fb  
	<b>Update .gitlab-ci.yml file</b> Eamon Goonan authored Jan 30, 2023	 0fab56ce  
29 Jan, 2023 2 commits		
	<b>Merge branch 'main' of https://gitlab.computing.dcu.ie/dortiem2/2023-ca326-mdortie-tradingbot</b> Eamon Goonan authored Jan 29, 2023	e0261341  

## 7.1 Testing

We wrote unit tests for each app in the Django project. As seen below, unit testing was split between views, URLs and models.



Example of Model test:

```
class TestModels(TestCase):  
  
    @unittest.skipIf(not django.test.TestCase.skipIfNotDBAvailable, 'DB not available')  
    def create_crypto_coin(self):  
        return CryptoCoin.objects.create(  
            name="TestCoin",  
            symbol="TST"  
        )  
  
    def test_crypto_coin_model(self):  
        coin = self.create_crypto_coin()  
        self.assertTrue(isinstance(coin, CryptoCoin))
```

Example of URL test:

```
class TestURLs(SimpleTestCase):  
  
    def test_index_url_resolves(self):  
        url = reverse('tradingplatform:index')  
        self.assertEqual(resolve(url).func, index)  
  
    def test_trade_url_resolves(self):  
        url = reverse('tradingplatform:trade')  
        self.assertEqual(resolve(url).func, trade)
```



Example of View test:

```
class TestViews(TestCase):  
  
    @unittest.skipIf(sys.version_info < (3, 6), "requires python3.6 or later")  
    def setUp(self):  
        user = User.objects.create(username='homer')  
        user.set_password('simpson')  
        user.save()  
  
    def test_login(self):  
        login = self.client.login(username='homer', password='simpson')  
        self.assertTrue(login)
```

## 7.2 User Testing

After obtaining ethics approval, we conducted user testing on the DCU campus. We recruited participants (two pairs of novice traders, and two pairs of experienced traders) and observed their usage of our web application. We then posed questions to the participants, asking them to describe their experience of the web application.

The results of our user testing were largely positive. The testing involved two pairs of novice traders and two pairs of experienced traders who were recruited to use our web application on the DCU campus.

Both novice and experienced traders found the platform easy to use, and they were particularly pleased with the selection of popular coins and the easy registration and login process. The experienced traders were impressed with the charts and the ability to track other traders, which allowed them to compare their trades with profitable traders.

The experience provided feedback that the platform should include more coins and that we should implement candle charts for each coin. On the other hand, novice traders suggested adding more traders to follow so that they can have a wider selection of traders to follow. The feedback provided by the experienced and novice traders was valuable for improving the web application and enhancing the user experience.

## 8. Bibliography

- Binance API: <https://binance-docs.github.io/apidocs/spot/en/#change-log>
- Shneiderman, B., Plaisant, C., Cohen, M., Jacobs, S., and Elmqvist, N., Designing the User Interface: Strategies for Effective Human-Computer Interaction: Sixth Edition, Pearson (May 2016) <http://www.cs.umd.edu/hcil/DTUI6>
- Nielsen, J. (1994a). Enhancing the explanatory power of usability heuristics. Proc. ACM CHI'94 Conf. (Boston, MA, April 24-28), 152-158.
- CryptoCurrency eXchange Trading Library: <https://pypi.org/project/ccxt/>
- Binance Futures Leaderboard API:  
<https://rapidapi.com/DevNullZero/api/binance-futures-leaderboard1>