Паттерны проектирования

Задание:
Обсуждение применительно к нашему проекту: какие паттерны будут необходимы и какие - нет, и по каким причинам.

Проанализировав разрабатываемый проект «CryptoTracker» и рассмотрев возможные проблемы в течение проектирования продукта, в качестве подходящих популярных паттернов (шаблонов) проектирования можно выделить:

1. Паттерн Factory «Фабрика»
   Фабричный паттерн предоставляет интерфейс для создания объектов, но позволяет подклассам решать, какой класс инстанцировать. Он упрощает создание объектов и позволяет легко добавлять новые типы объектов без изменения существующего кода.

Django предоставляет базовый класс `django.db.models.Model`, который может быть наследован для создания конкретных моделей. Для создания объектов моделей можно использовать фабричный метод `create()` класса модели, который создает новый объект модели и сохраняет его в базе данных.

2. Паттерн MVC «Model-View-Controller»
   Архитектурный паттерн, который разделяет приложение на три основных компонента: Модель (хранит данные и бизнес-логику), Представление (отвечает за отображение данных пользователю) и Контроллер (управляет взаимодействием между Моделью и Представлением). Этот паттерн повышает модульность, повторное использование кода и облегчает сопровождение приложений.

В Django паттерн MVC реализован с помощью шаблона проектирования MTV (Model-Template-View), который является модификацией классического MVC.

В MTV:

- Модель (Model) представляет данные приложения и обеспечивает доступ к ним. В Django модели описываются с помощью классов, которые наследуются от базового класса models.Model.

- Шаблон (Template) отвечает за отображение данных на странице и представляет собой HTML-шаблон с вставками переменных и тегами шаблонизатора. В Django используется встроенный шаблонизатор, который позволяет создавать шаблоны с помощью языка шаблонов Django.

- Представление (View) является связующим звеном между моделью и шаблоном. Оно обрабатывает запросы пользователя, извлекает необходимые данные из модели и передает их в шаблон для отображения. В Django представления описываются с помощью функций или классов, которые наследуются от базовых классов View или TemplateView.

Таким образом, в Django модель, шаблон и представление выполняют те же функции, что и в классическом MVC, но с некоторыми отличиями в реализации. В целом, MTV является более гибким и удобным для разработки веб-приложений, чем классический MVC.

3. Паттерн Observer «Наблюдатель»
   Паттерн определяет зависимость "один-ко-многим" между объектами, таким образом, что при изменении состояния одного объекта все зависимые от него объекты автоматически оповещаются и обновляются. Он позволяет реализовать слабую связь между объектами и облегчает поддержку согласованности между ними.

Важный паттерн для реализации интерактивного интерфейса, позволяющий в случае изменения информации самим пользователем, обновить все связанные элементы интерфейса путем передачи информации о изменении одного из них. Например в случае изменения(добавлении) портфеля активов, происходит оповещение класса с контейнера со списком всех активов о необходимости обновится или

добавление актива оповещает объект графика о необходимости обновить данные и перерисовать график стоимости портфеля.

4. Паттерн Decorator «Декоратор»
   Позволяет добавлять новые функции к существующему классу без изменения его структуры. Он предлагает гибкую альтернативу наследованию для расширения функциональности классов.

```python
class AssetViewSet(viewsets.ModelViewSet):
    queryset = Asset.objects.all()
    serializer_class = AssetSerializer

    @action(url_path="by_coin_id", methods=["POST"], detail=False,
            permission_classes=[permissions.AllowAny])
    def get_asset_by_coin_id(self, request):
        coin_id = request.data.get("coin_id")
        # print(coin_id)
        assets = get_object_or_404(Asset, coin_id=coin_id)
        # print(assets)
        asset_serializer = AssetSerializer(assets)
        return Response(data=asset_serializer.data, status=status.HTTP_200_OK)


    @action(url_path="fill_assets", methods=["POST"], detail=False,
            permission_classes=[permissions.isAdmin])
    def fill_assets(self, request):
        """CoinGecko api"""

        asset_data = cg.get_coins_list(include_platform=True)
        json_data = pd.json_normalize(asset_data, max_level=0)

        asset_instances = [Asset(
            coin_id=asset[0],
            symbol=asset[1],
            name=asset[2],
            platforms=asset[3],
        ) for asset in json_data.values]

        self.queryset.bulk_create(asset_instances)
        return Response(data={"message": "База данных успешно обновлена"}, status=status.HTTP_200_OK)

    @action(url_path="update_assets", methods=["POST"], detail=False,
            permission_classes=[permissions.AllowAny])  # isAdmin
    def update_assets(self, request):
        local_data = pd.read_json(os.path.join(settings.BASE_DIR, 'coins_markets.json'))
        local_data = local_data.replace({np.nan: None})
        # print(local_data)
```

Среди неподходящих к разрабатываемому проекту вариантов явно выделяются:

1. Паттерн Prototype «Прототип»
   Прототипный паттерн предлагает создание объекта на основе уже существующего объекта-прототипа. Это позволяет создавать новые объекты, избегая сложных иерархий классов или дорогостоящих операций инициализации.

Не используется, поскольку отсутствует потребность в постоянном создании прототипов основных объектов программной структуры проекта.

2. Паттерн Strategy «Стратегия»
   Стратегический паттерн позволяет определить семейство алгоритмов, инкапсулировать их в отдельные классы и сделать их взаимозаменяемыми. Он позволяет менять поведение объекта во время выполнения программы, не изменяя его структуры.

В виду максимизации производительности, при стандартизированных задачах решаемых программным продуктом, вариативность исполнения функционала не требуется, как следствие есть четко определенный наиболее продуктивный алгоритм действий, что убирает необходимость наличия нескольких сценариев.

3. Паттерн Singleton «Одиночка»
   Этот паттерн гарантирует, что класс имеет только один экземпляр, и предоставляет глобальную точку доступа к этому экземпляру. Он полезен, когда требуется общий ресурс, доступный из разных частей приложения.

В виду решения использовать для взаимодействия между глобальными элементами систем (frontend и backend) протокола сетевых запросов и “инкапсуляции” этих элементов, необходимость в глобальной точке доступа к общим ресурсам отсутствует, также каждый элемент frontend системы которому нужно получить доступ к глобальной информации об активах пользователя запрашивает ее самостоятельно и распространяет полученную информацию согласно паттерну наблюдателей по другим элементам системы если в этом есть необходимость.

4. Паттерн Builder «Строитель»
   Строительный паттерн используется для создания сложных объектов шаг за шагом. Он позволяет разделить процесс конструирования объекта от его представления, что упрощает создание различных вариаций объекта.

Не используется, так как нету потребности в генерации новых сложных объектов и классов в больших количествах.