

Санкт-Петербургский государственный университет

Математическое обеспечение и
администрирование информационных систем

Мамро Никита Владимирович

Разработка онлайн-платформы для трейдинга

Отчёт по учебной практике

Научный руководитель:
к. т. н., доцент Литвинов Ю. В.

Консультант:
Старший разработчик, ООО "Ланит-Терком"
Рябцев С. А.

Санкт-Петербург
2020

Оглавление

Введение и постановка задач	3
1. Обзор архитектуры приложения	5
2. Обзор аналогов	7
3. Описание решения поставленных перед автором задач	8
3.1. Используемые инструменты	8
3.2. Сервис аутентификации	9
3.3. Сервис новостей	11
3.4. Взаимодействие с биржевыми брокерами и торговые боты	12
3.4.1. Broker Service и TinkoffAPI	12
3.4.2. Торговые боты	12
3.5. Графический интерфейс	15
3.6. Обработка ошибок и логирование	17
3.6.1. Обработка ошибок	17
3.6.2. Чтение логов	18
3.7. Тестирование графического интерфейса	20
Заключение	21
Список литературы	22
Приложение	24

Введение и постановка задач

Торговля финансовыми инструментами относится к тем видам деятельности, которые люди стремятся максимально упростить или же вообще автоматизировать. Объём торгов на биржах растёт, так, например, недавно Московская биржа зафиксировала рекордный среднесуточный объём торгов, а с прошлого года объём операций с акциями вырос более чем вдвое.¹ Это мотивирует на создание специализированных приложений для проведения торговых сделок удалённо и их автоматизации.

Работа, о которой предоставлен данный отчёт, проводилась в рамках студенческого проекта, представленного компанией «Ланит-Терком». Цель проекта — разработка многопользовательской онлайн-платформы для трейдинга, предоставляющей возможности совершать сделки с помощью биржевых брокеров вручную или с помощью торговых ботов, хранить портфели пользователей, их транзакции, а также предоставлять пользователям новости об экономике.

Непосредственно перед автором стояли следующие задачи:

- реализовать логику аутентификации пользователей;
- добавить возможность получения новостей;
- разработать сервис, отвечающий за взаимодействие с биржевыми брокерами;
- реализовать логику работы торговых ботов;
- создать компоненты графического интерфейса, отвечающие описанным выше задачам (кроме аутентификации), а также компоненты, отвечающие представлению/изменению информации о пользователе и портфолио;
- принять участие в создании логики логирования внутри системы и создать приложение для чтения логов;

¹Режим доступа: <https://www.vedomosti.ru/finance/news/2020/03/19/825687-moskovskaya-birzha-soobschila-o-rekordnom-obeme-torgov> (Дата обращения: 08.06.2020)

- добавить логику обработки ошибок на уровне сервисов, к которым клиент обращается по REST;
- написать приложение для автоматического запуска тестов графического интерфейса и часть самих тестов.

1. Обзор архитектуры приложения

В ходе планирования архитектуры приложения немаловажную роль сыграл тот факт, что работа над проектом была командной, а также что в течение всего времени отдельные части приложения могли претерпевать сильные изменения в логике. С учётом обозначенных факторов выбор пал на микросервисную архитектуру по следующим причинам:

- разделение всей логики на микросервисы делает более удобной работу в команде: задачи можно разделять в соответствии с тем, к какому микросервису они относятся, что позволяет разным разработчикам зависеть друг от друга в минимальной степени, тем самым упрощая распараллеливание процесса разработки;
- для обновления какой-то части приложения достаточно провести изменения конкретного микросервиса, это позволяет избежать конфликтов с другими частями приложения и необходимости полностью останавливать работу всего сервиса при внесении изменений.

В соответствии с поставленными перед проектом задачами приложение было разбито на следующие микросервисы:

- сервис аутентификации (Authentication Service) — сервис, отвечающий за вход пользователя в систему и за управление токенами аутентификации;
- сервис операций (Operation Service) — сервис, отвечающий за проведение финансовых операций;
- сервис новостей (News Service) — сервис, отвечающий за получение новостей и курса валют;
- сервис пользователей (User Service) - сервис, отвечающий за операции с данными о пользователях;
- сервис брокеров (Broker Service) — сервис, отвечающий за взаимодействие с биржевыми брокерами;

- сервис баз данных (DataBase Service) — сервис, отвечающий за работу с базами данных.

Схема, отображающая каждый из микросервисов и связи между ними, представлена в приложении (рис. 1).

2. Обзор аналогов

Студенческий проект, о работе над которым предоставлен данный отчёт, позиционировался организаторами в первую очередь как учебный и не ставил задачи воплотить кардинально новые идеи с целью превзойти существующие аналоги. Тем не менее, при разработке во время рассмотрения существующих аналогов было обращено внимание на MetaTrader², как на самую популярную платформу для торговли³.

MetaTrader является одной из самых первых платформ для валютной торговли и пользуется большим спросом не только по той причине, что предоставляет мультичартинг, ожидающие и лимитные ордера, и многое другое, но и потому что на ранних стадиях у MetaTrader почти не было конкурентов. Тем не менее это не торговая платформа на базе веб-приложения, что означает невозможность непосредственно работать с MetaTrader в браузере. Помимо этого, MetaTrader не совместима с OSX. Проект же, работа над которым описана в данном отчёте, предназначен для работы в браузере и не накладывает никаких ограничений на устройства пользователя.

²Режим доступа: <https://www.metatrader5.com/ru> (Дата обращения: 12.06.2020)

³Режим доступа: <https://admiralmarkets.ee/ru/education/articles/trading-software/kakaia-luchshaia-torghovaia-platforma-forieks-1> (Дата обращения: 12.06.2020)

3. Описание решения поставленных перед автором задач

3.1. Используемые инструменты

Разработка приложения велась в среде ASP.NET Core [13] преимущественно на языке C#. Для клиентской части использовались HTML, CSS (в частности Bootstrap [4]) и JavaScript, однако большая часть логики на стороне клиента тоже была реализована на C# благодаря фреймворку Blazor [12], который был выбран именно с целью минимизации количества кода на других языках.

Для работы с базами данных были использованы MS SQL Server в роли СУБД и Entity Framework [10] для взаимодействия с данными из кода на C#.

Для взаимодействия между микросервисами был использован брокер сообщений RabbitMQ [14] в связке с библиотекой MassTransit [6], упрощающей работу с шиной данных.

Также для валидации данных, отправляемых пользователем, использовалась библиотека Fluent Validation [16]

Отдельно хочется отметить, что при создании логики сервисов использовался паттерн Команда [17], потому что далее при описании решения будет использоваться сочетание “команда для“. Причиной активного использования данного паттерна стало то, что он позволяет использовать единый интерфейс для классов, отвечающих обработке того или иного запроса, и Dependency Injection [8] в конвейере обработке запроса к сервису.

3.2. Сервис аутентификации

На первых этапах разработки основной целью было создание логики управления данными о пользователях, поэтому одной из первых задач стало создание сервиса для прохождения аутентификации. Для решения задачи было необходимо реализовать команды для входа пользователя в систему и проверки действительности токена пользователя, а также создать middleware [9] для того, чтобы сервис аутентификации также мог быть использован как для проверки пользователя при получении запроса на вход со стороны клиента, так и при обработке запросов другими сервисами.

Помимо сервиса аутентификации необходимо было добавить возможность хранить токен на стороне клиента, чтобы позже в запросах от клиента в заголовке можно было передавать токен, проверка которого необходима в ряде ситуаций (например, при запросе на получение новостей токен предоставлять не нужно, но при запросе на изменение данных о профиле нужно предоставить действительный токен).

Для идентификации пользователя использовался токен (JWT [5]), который выдавался в случае, если в сервис аутентификации пришёл запрос на вход в систему с корректными данными, проходящий проверку валидаторов для электронной почты и пароля, созданных с использованием Fluent Validation. В случае прохождения валидации с помощью брокера сообщений и Mass Transit формировалась цепочка запросов вплоть до сервиса, отвечающего за базы данных, с запросом на информацию о хранящемся хеше пароля, полученного с использованием алгоритма SHA-1, который соответствует данной почте. Если пользователь с указанной почтой существует, и хеш введённого пароля совпадает с хешем из базы, то специально созданный класс TokensEngine, ответственный за выдачу и проверку токенов с использованием библиотечного класса JwtSecurityTokenHandler, выдавал пользователю токен.

Для проверки токена при отправке запросов, требующих идентификации пользователя, на другие сервисы, был создан специальный класс CheckTokenMiddleware, в котором добавлен асинхронный метод

`InvokeAsync` от параметра типа `HttpContext`, где в случае, если запрос шёл на адрес, требующий проверку токена, формировался дополнительный запрос на проверку токена в сервисе аутентификации. Благодаря этому классу в остальных сервисах к конвейеру обработки запроса достаточно было подключить `CheckTokenMiddleware` для проверки токена при определённых операциях.

Для сохранения токена на стороне клиента была использована библиотека `Blazored` [3], предоставляющая удобный интерфейс для работы с сессионным хранилищем.

3.3. Сервис новостей

Сервис новостей отвечает за получение информации из новостных RSS-лент. Для вычленения нужных данных был использован класс XDocument [11], который позволяет получить удобное представление RSS-ленты. После загрузки ленты в XDocument с помощью LINQ получается необходимая информация о каждой отдельной новости, после чего формируется объект, содержащий эту информацию, для отправки на клиент, где в свою очередь новости преобразуются в отдельные компоненты интерфейса.

В проекте в качестве источника новостей используется только Рамблер, однако функциональность для их получения была реализована с учётом возможности использования и других источников. Для этого был использован паттерн Фабричный метод [18]. Создан специальный класс-фабрика NewsPublisherFactory, который в зависимости от типа сервиса, предоставляющего новости возвращал соответствующий инструмент для получения информации о новостях. Благодаря этому сервис новостей расширяем в смысле количества источников новостей.

3.4. Взаимодействие с биржевыми брокерами и торговые боты

Приложение поддерживает возможность взаимодействия с брокером Tinkoff банка, используя TinkoffAPI [2] и в частности SDK [1] для C#. В рамках проекта работа проводилась в предоставляемой брокером песочнице, операции происходили не с реальными денежными средствами и не с реальными инструментами.

3.4.1. Broker Service и TinkoffAPI

В сервисе брокеров есть команды для получения списка инструментов, подписки на цену на конкретный инструмент и на совершение транзакций. Автор отчёта отвечал за общую реализацию сервиса и первую команду из трёх.

При запросе на сервис брокеров пользователь указывает тип брокера, а также свой токен для конкретного брокера. В сервисе брокеров класс-фабрика возвращает нужного брокера, который реализует общий интерфейс IBroker в отдельной библиотеке, и вызывает у него требуемый метод. Так приложение потенциально поддерживает работу с разными брокерами, для каждого из которых будет достаточно просто создать соответствующий класс, реализующий интерфейс IBroker.

В зависимости от того, какой тип инструментов интересен пользователю, в брокере отправляется запрос в песочницу, возвращающий необходимые инструменты. После этого инструменты параллельно приводятся к типу, общему для нашего сервиса, и отправляются в виде коллекции к клиенту или же к другому сервису, который отправлял запрос.

3.4.2. Торговые боты

Для корректной работы торговых ботов было необходимо сначала реализовать логику сохранения данных о ботах, а потом реализовать непосредственно логику работы ботов.

Информация о том, какому пользователю принадлежит какой бот, хранится в отдельной таблице в базе данных. Для описания данных, хранящихся в той или иной таблице, был использован паттерн Репозиторий [7]: так, например, был создан класс `BotRepository`, который предоставлял интерфейс для обращения за теми или иными данными о ботах в базу. Соответственно были реализованы все необходимые методы на уровне сервисов операций и баз данных для получения нужной информации.

В текущей реализации боты поддерживают следующий вид правил: "Купить/продать инструмент I, если разница в цене за последние M минут изменилась на X, при этом нельзя использовать больше, чем P процентов текущего баланса пользователя". В данном случае очевидно, как хранить в базе данных информацию о каждом из правил.

Была создана таблица связей между ботами и правилами, содержащая информацию о том, какие идентификаторы правил соответствуют боту с данным идентификатором. Были реализованы методы для изменения информации в данных таблицах на уровне сервисов операций и баз данных.

Таким образом, по идентификатору пользователя можно получить список идентификаторов всех его ботов, по которому, в свою очередь, можно получить информацию о заданных правилах. Опять же реализованы все методы, отвечающие получению подобной информации на уровне сервисов операций и баз данных.

После реализации логики управления информацией о ботах и правилах оставалось создать инструменты, которые бы по этой информации запускали соответствующих ботов, применяющих правила к каждому инструменту из списка, который передаётся как параметр.

Для работы ботов были добавлены классы `BotRunner`, `BotRule` и `TimeDifferenceTrigger`.

В `BotRunner` в метод `Run` передаются данные, необходимые для запуска ботов, а в метод `Stop` передаётся идентификатор бота, и в случае, если бот в текущий момент работает, работа завершается.

В классе `TimeDifferenceTrigger` происходит в реальном времени от-

слеживание цены одного конкретного инструмента и проверка на выполнение указанного правила при обновлении информации об инструменте.

Класс `BotRule` описывает общую логику работы какого-либо правила для бота. Правило создаётся и начинает работу при вызове метода `Run` в классе `BotRunner`, каждому правилу соответствует множество триггеров для соответствующих инструментов. В триггере есть обработчик события исполнения правила, при его вызове внутри правила вызывается метод, в котором происходит попытка выполнить соответствующую транзакцию.

3.5. Графический интерфейс

Для создания графического интерфейса использовался фреймворк Blazor и Razor компоненты с использованием стилей из библиотеки Bootstrap. Засчёт этого в большинстве случаев удалось избежать необходимости прибегать к использованию JavaScript.

Для отправки http-запросов соответствующим сервисам использовался написанный другим членом команды клиент, который отвечал за формирование запросов и десериализацию полученного ответа.

Для хранения информации о данных были использованы класс-модель, который привязан к форме с информацией о пользователе. С помощью встроенных атрибутов были указаны обязательные для ввода поля и формат данных, в случае, если какое-либо условие на свойства модели не выполняется, пользователь не может подтвердить ввод данных (рис. 5).

За счёт того, что можно интегрировать код на C# в HTML разметку, изменять состояние компонентов на текущей странице можно, изменяя значения тех или иных переменных. Так, например, на странице с информацией о пользователе при нажатии на кнопку для изменения данных о себе пользователь менял состояние булевой переменной, отвечающей за видимость элементов интерфейса, отображаемых только в режиме изменения, что влекло за собой отображение/скрытие соответствующих элементов. Также благодаря возможности интегрировать код на C# в разметку сайта, на странице для новостей было достаточно один раз описать стиль отображения одной новости и обернуть его в цикл, после чего, проходя по массиву данных, полученных из сервиса новостей, подставить нужные параметры (рис. 2).

Как было упомянуто ранее, не удалось избежать написания JavaScript кода. Так, например, на главной странице новости по 3 штуки в каждой категории представляют собой карусель, которая вращается автоматически с определенным интервалом или же по нажатии на стрелки слева и справа от новостей. Для работы карусели был написан отдельный скрипт на JavaScript. Чтобы вызывать код на JavaScript на странице,

был использован JSInterop, позволяющий в коде на C# на Razor странице указать, функцию на JavaScript с каким именем нужно вызвать с данными параметрами.

Для создания страниц с портфолио, ботами и списками инструментов от биржевых брокеров были использованы таблицы, у которых столбцами были характеристики инструментов (например: идентификатор инструмента, название, валюта, количество единиц в одном лоте и возможность открыть страницу с подробной информацией), а строками непосредственно записи о каждом инструменте. На обеих страницах добавлена функциональность пагинации на уровне клиента после получения данных от соответствующего сервиса, а также в списке инструментов есть возможность выбирать размер одной страницы (рис. 3). На странице с портфолио помимо информации отдельно о каждом инструменте есть информация об общей стоимости портфеля, отслеживаемая в реальном времени (рис. 6).

На странице с ботами пользователь может создавать новых ботов, именовать их, задавать, просматривать и изменять определенные параметры правила, запускать их для выбранного списка инструментов, а также удалять (рис. 4).

3.6. Обработка ошибок и логирование

Немаловажную роль в работе приложения играют правильно настроенная обработка ошибок и возможность отслеживать состояние отдельных компонентов при обработке запросов от пользователей.

Во-первых, было необходимо настроить корректную обработку ошибок на стороне REST сервисов, чтобы все возникающие внутри системы ошибки возвращались к пользователю в виде соответствующего статус-кода. Помимо этого было необходимо настроить обработку ошибок на уровне сервисов, взаимодействующих с помощью брокера.

Во-вторых, надо было добавить возможность записывать логи о работе системы в отдельную базу данных и считывать их из неё в приемлемом виде. В данном случае перед автором отчёта стояла задача преимущественно касательно чтения логов, поскольку записью занимался другой человек. Также автором была добавлена возможность отслеживания цепочек ошибок внутри логов при записи, это было реализовано за счёт того, что при пробрасывании ошибки по сервисам сохранялся идентификатор лога, записываемого в базу данных, и при записи каждого из дочерних логов в распоряжении имелся идентификатор родительского, сохраняющийся в отдельном столбце в таблице логов.

3.6.1. Обработка ошибок

Для того, чтобы пользователь получал корректные статус-коды в случае возникновения ошибок, необходимо было создать коллекцию исключений, соответствующих разным кодам, а также добавить обработчики в конвейеры REST сервисов, а именно сервиса аутентификации, сервиса пользователей, сервиса операций и сервиса новостей.

Был реализован метод, принимающий в качестве параметра объект класса `HttpContext`, который был указан в качестве пользовательского обработчика ошибок в соответствующих конвейерах обработки запросов. Внутри этого метода формировался объект класса `ErrorResponse`, который содержал в себе такую информацию об ошибке, как время воз-

никновения и сообщение. Также созданные исключения, соответствующие статус-кодам 400, 403, 404 и 500, при обработке позволяли точно определить, какой статус-код нужно указать в ответе на запрос.

Для обработки ошибок внутри сервисов, обменивающихся информацией с помощью брокера сообщений, был создан отдельный класс-обёртка, который хранит информацию о полученном ответе и статус-код ответа. При формировании ответа вызывается вспомогательный метод, принимающий в качестве параметров запрос и функцию, вычисляющую ответ, и в случае, если во время вычисления возникла ошибка, в виде ответа посылается объект, оборачивающий определённый статус-код. За счет этого все ошибки, возникающие при получении запросов, не вызывают непредвиденного поведения программы.

3.6.2. Чтение логов

Для чтения логов было создано WPF приложение, использующее Entity для получения данных из базы. Каждый лог представлялся в виде пользовательского класса Node, содержащего объект класса с информацией о конкретном логе и коллекцией типа ObservableCollection, содержащей объекты класса Node для родительских логов. Получение цепочки логов стало возможно за счёт того, что при перекидывании ошибки с сервиса на сервис сохраняется информация об идентификаторе последнего лога. Таким образом, у каждого лога есть свойство, отвечающее за идентификатор родительского лога, значение которого также может быть и null в том случае, если записанный лог никак не связан с другими.

При запуске приложения создаётся соединение с базой данных с логами, после чего сначала отдельные логи, а потом цепочки от родительского лога к дочернему, формируются в коллекцию ObservableCollection из объектов класса Node (рис. 7).

Добавлена фильтрация по типу логов (Warning/Information/Error)

Добавлен класс, реализующий интерфейс IValueConverter, для того, чтобы можно было получать соответствующий цвет по типу записи о логе (Warning - оранжевый, Information - синий, Error - красный), ко-

торый используется как конвертер в хaml документе для того, чтобы сообщения логов разных типов окрашивались в нужный цвет.

3.7. Тестирование графического интерфейса

Для тестирования графического интерфейса приложения был использован движок Selenium [15], предоставляющий средства для автоматического запуска браузера и выполнения заранее описанных команд, соответствующих определённым действиям пользователя на сайте, а также для определения содержимого веб-страницы. Таким образом, каждый тест графического интерфейса представляет собой запуск последовательности действий, эмулирующих действия пользователя на сайте, и проверки на то, соответствует ли содержимое страницы ожиданиям. Действие пользователя на сайте описывается в 2 шага: получение нужного элемента интерфейса по id, классу или атрибуту и клик по элементу.

Пример теста графического интерфейса: после прохождения регистрации и входа на сайт, в шапке страницы кнопки входа и регистрации должны заменяться аватаром пользователя.

Для запуска тестов было создано консольное приложение, с помощью рефлексии обнаруживающее и запускающее методы, помеченные специальным атрибутом. По завершении выполнения тестов на консоль печатается сообщение о статусе тестов (рис. 8).

Заключение

В ходе работы автором были достигнуты следующие результаты:

- добавлена логика аутентификации пользователей на основе JWT;
- добавлена логика получения новостей от рамблера;
- создан сервис для взаимодействия с Tinkoff брокером и получения информации о финансовых инструментах;
- реализована логика работы торговых ботов, правил работы, их сохранения, изменения и запуска;
- создан графический интерфейс, отвечающий за управление информацией о пользователе, просмотр новостей, информации о финансовых инструментах, портфолио, а также страница с ботами;
- проведена работа над добавлением логики логирования в систему, добавлена корреляция логов об ошибках, создано приложение для чтения логов;
- добавлен обработчик исключений в сервисы, с которыми клиент связывается непосредственно через REST;
- добавлено приложение для запуска всех тестов графического интерфейса, также написана часть самих тестов с использованием движка Selenium.

Код всего проекта доступен на GitHub ⁴

⁴Режим доступа: <https://github.com/lanit-students/TradingStation> (Дата обращения: 08.06.2020)

Список литературы

- [1] API Tinkoff. OpenAPI .NET SDK // GitHub. — URL: <https://github.com/TinkoffCreditSystems/invest-openapi-csharp-sdk> (дата обращения: 05.06.2020).
- [2] API Tinkoff. Тинькофф инвестиции OpenAPI // Tinkoff. — URL: <https://tinkoffcreditsystems.github.io/invest-openapi/> (дата обращения: 05.06.2020).
- [3] Blazored. Session Storage // GitHub. — URL: <https://github.com/Blazored/SessionStorage> (дата обращения: 05.06.2020).
- [4] Bootstrap. Официальный сайт Bootstrap // Bootstrap. — URL: <https://getbootstrap.com/> (дата обращения: 05.06.2020).
- [5] М. Jones Microsoft J. Bradley N. Sakimura. О JWT // IETF Tools. — URL: <https://tools.ietf.org/html/rfc7519> (дата обращения: 05.06.2020).
- [6] MassTransit. Официальный сайт MassTransit // MassTransit. — URL: <https://masstransit-project.com/> (дата обращения: 05.06.2020).
- [7] McCool Shawn. The Repository Pattern // ShawnMc.Cool. — URL: https://shawnmc.cool/2015-01-08_the-repository-pattern (дата обращения: 05.06.2020).
- [8] Microsoft. ASP.NET Core Dependency Injection // Microsoft. — URL: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection> (дата обращения: 05.06.2020).
- [9] Microsoft. ASP.NET Core Middleware // Microsoft. — URL: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware> (дата обращения: 05.06.2020).
- [10] Microsoft. Entity documentation // Microsoft. — URL: <https://docs.microsoft.com/en-us/ef/> (дата обращения: 05.06.2020).

- [11] Microsoft. XDocument // Microsoft. — URL: <https://docs.microsoft.com/en-us/dotnet/api/system.xml.linq.xdocument> (дата обращения: 05.06.2020).
- [12] Microsoft. О Blazor // Microsoft. — URL: <https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor> (дата обращения: 05.06.2020).
- [13] Microsoft. Об ASP.NET Core // Microsoft. — URL: <https://dotnet.microsoft.com/learn/aspnet/what-is-aspnet-core> (дата обращения: 05.06.2020).
- [14] RabbitMQ. Официальный сайт RabbitMQ // RabbitMQ. — URL: <https://www.rabbitmq.com/> (дата обращения: 05.06.2020).
- [15] Selenium. Инструмент для тестирования GUI // Selenium. — URL: <https://www.selenium.dev/> (дата обращения: 05.06.2020).
- [16] Validation Fluent. Официальный сайт Fluent Validation // Fluent Validation. — URL: <https://fluentvalidation.net/> (дата обращения: 05.06.2020).
- [17] Рефакторинг.Гуру. Паттерн Команда // Рефакторинг.Гуру. — URL: <https://refactoring.guru/ru/design-patterns/command> (дата обращения: 05.06.2020).
- [18] Рефакторинг.Гуру. Паттерн Фабричный метод // Рефакторинг.Гуру. — URL: <https://refactoring.guru/ru/design-patterns/factory-method> (дата обращения: 05.06.2020).

Приложение

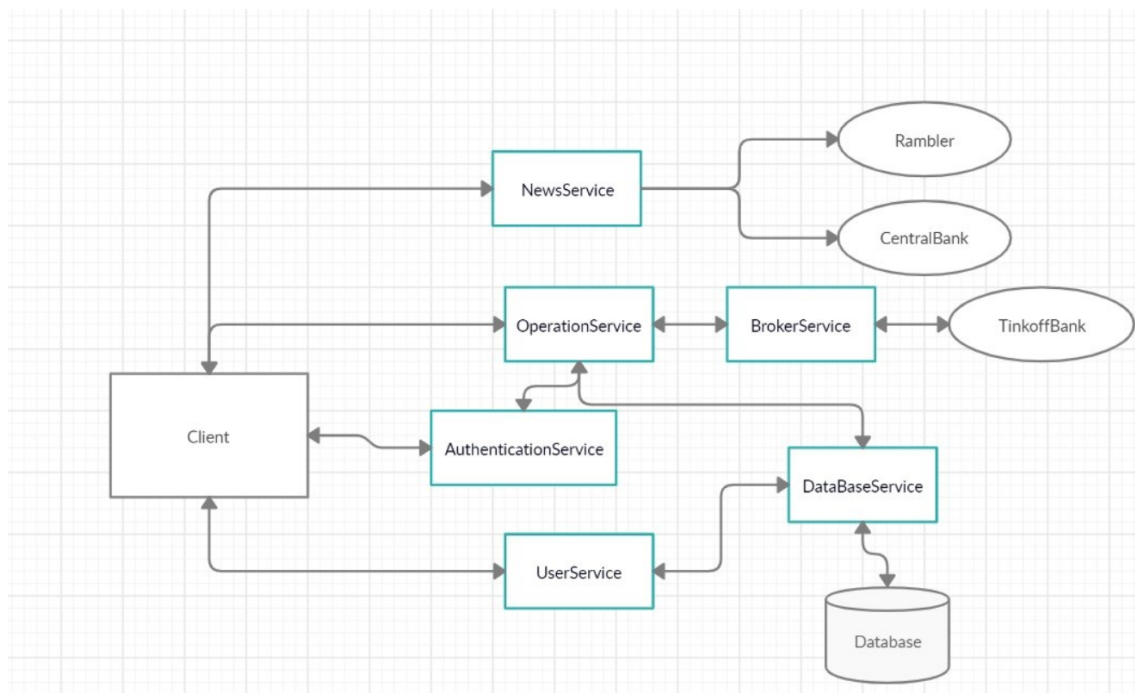


Рис. 1: Архитектура приложения

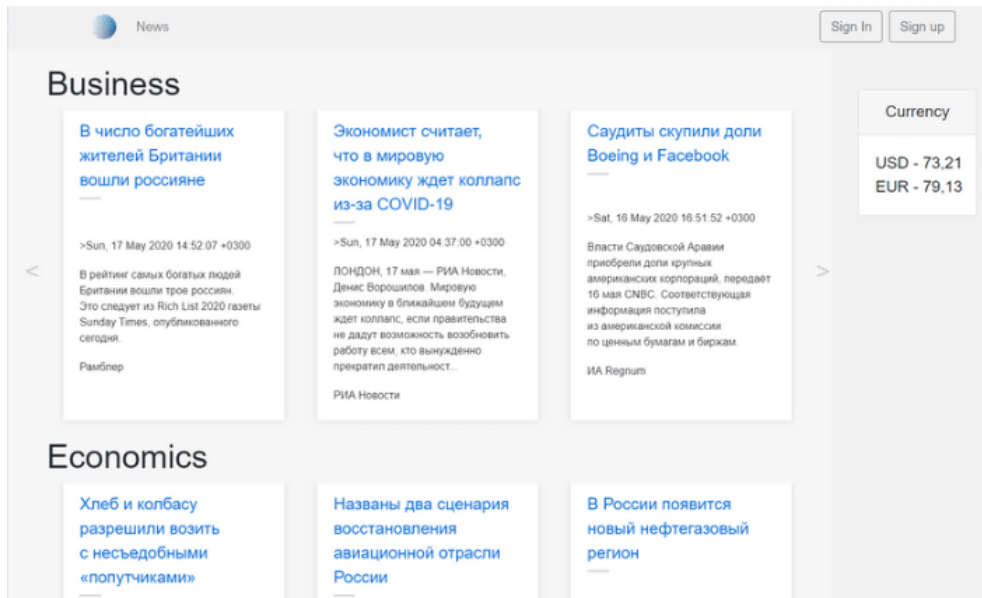


Рис. 2: Страница с новостями

Instruments

Instrument Bond Page size 10

Figi	Name	Currency	Lot	Action
BBG00FZGD8T9	ЧТПЗ выпуск 2	Rub	1	Open instrument
BBG00QYKX940	СЭЗ им. Серго Орджоникидзе	Rub	1	Open instrument
BBG00JHSCRH7	Магаданская область выпуск 1	Rub	1	Open instrument
BBG00LNNQHC8	Фольксваген Банк РУС 001P выпуск 2	Rub	1	Open instrument
BBG001QY1TQ1	Мечел выпуск 17	Rub	1	Open instrument
BBG009SZTW91	Нижегородская область выпуск 10	Rub	1	Open instrument
BBG00L92CX58	ГТЛК выпуск 11	Usd	1	Open instrument
BBG00N9S8LL0	Фридом Финанс выпуск 1	Rub	1	Open instrument
BBG00PCD8BF3	Роснано БО-002P выпуск 1	Rub	1	Open instrument
BBG00R4C0L75	ПИК-Корпорация выпуск 2	Rub	1	Open instrument

0 1 2 ... 38 Next Go to 0

Рис. 3: Компонент GUI с инструментами

Bots


Create new bot

Name	Is active	Rules	Actions		
Bot	False	Sell if price more than 5.00 in limit 10 % of balance with interval 10	Run	Edit	Delete

Рис. 4: Компонент GUI с информацией о ботах


User Info

Personal Information Transactions Balance



Name:

Surname:

Birthday: 

Email:

Рис. 5: Компонент GUI с информацией о пользователе в режиме изменения информации

Portfolio

Total portfolio price: 3858901.8000 RUB 0 EUR 295240.00 USD

Figi	Name	Provider	Count on hand	Price	Action
BBG00QYKX940	СЗЗ им. Серго Орджоникидзе	Tinkoff	12	1043.9 Rub	Open instrument
BBG0013HJJ31	Евро	Tinkoff	50000	76.9275 Rub	Open instrument
BBG000H3GDJ8	Kemper	Tinkoff	4000	73.81 Usd	Open instrument

Рис. 6: Компонент GUI с портфолио

Select log level: **Warning**

▲ 5/15/2020 4:33:46 PM | AuthenticationService | **Warning** | User with email ololo@lo.lo not found
5/15/2020 4:33:46 PM | DataBaseService | **Warning** | User with email ololo@lo.lo not found

Рис. 7: Интерфейс приложения для чтения логов

```
Testing report:
-----
Found classes to test: 3
Found methods to test (total): 4
-----
Class: GUITestsEngine.Tests.SignInTests
-----
Tested method: SignInTest()
Passed SignInTest() test
-----
Class: GUITests.Tests.FailExampleTest
-----
Tested method: ThisTestFails()
Failed ThisTestFails() test
-----
Class: GUITestsEngine.Tests.SignUpTests
-----
Tested method: EmptySignUpTest()
Passed EmptySignUpTest() test
-----
Tested method: ValidSignUpTest()
Passed ValidSignUpTest() test
-----
```

Рис. 8: Интерфейс приложения для запуска тестов на движке selenium