**СОФИЙСКА ПРОФЕСИОНАЛНА ГИМНАЗИЯ ПО ЕЛЕКТРОНИКА**

**„ДЖОН АТАНАСОВ“**

**Д И П Л О М Е Н П Р О Е К Т**

**Тема: „**Проектиране на уеб базирана платформа за участниците на капиталовите пазари“

**Практическа част:** „Реализация на уеб базирана платформа за участниците на капиталовите пазари“

Дипломант: Александър Андреев Иванов 12В клас

*/име, презиме, фамилия, клас/*

Професия: 481030 „ Приложен програмист“

Специалност: 4810301 „Приложно програмиране“

Ръководител на дипломен проект: инж. Любица Димитрова

Дипломант: .....................

*/подпис/*

Ръководител на Дипломен проект: .................

*/подпис/*

София

2023

**СЪДЪРЖАНИЕ**

Съдържание............................................................................................................2

**Теоретична част**...................................................................................................4

Глава I. Увод...........................................................................................................4

Глава II. Изложение................................................................................................6

1. Съществуващи решения.........................................................................6
2. Предпроектно проучване.......................................................................7
3. Архитектури, база данни, парадигми и дизайн шаблони................9
   1. „Клиент – сървър“ мрежова архитектура............................8
   2. Трислойна архитектура........................................................10
   3. Уеб API..................................................................................10
   4. Едностранични приложения...............................................10
   5. Обектно-ориентирано програмиране.................................10
   6. SOLID принципи..................................................................10
   7. Релационни бази данни........................................................10
   8. Обектно-релационно картографиране................................10
   9. Дизайн шаблони...................................................................10
   10. Асинхронно програмиране..................................................10
   11. Автентикация с JWT токени...............................................10
4. Платформи, езици, библиотеки и протоколи.....................................10
   1. C#, .NET и ASP.NET CORE.................................................10
   2. SQL и Entity Framework Core..............................................10
   3. HTTP и WebSocket протоколи............................................10
   4. Уеб функционалности в реално време и SignalR..............17
   5. Допълнителни библиотеки на С#.......................................17
   6. HTML, CSS и JS...................................................................18
   7. React - JavaScript библиотека..............................................18
5. Структура на имплементацията.........................................................18
   1. Структура на базата данни..................................................18
   2. Структура на сървърното приложение..............................18
   3. Структура на клиентското приложение.............................18

Глава III. Заключение..........................................................................................28

**Практическа част**...............................................................................................29

Тема, реализация, изводи.....................................................................................30

Използвана литература.........................................................................................31

Приложения...........................................................................................................32

***Теоретична част***

**Глава I.   
Увод**

Преди дефинирането на съществуващите всекидневни технически проблеми във финансовия свят трябва да бъдат дадени отговори на следните фундаментални въпроси: „Какво са капиталовите пазари?“; „Защо са важни?“; „Кои са участниците в тях?“; „Какво сa финансови инструменти?“. Капиталовите пазари са интернационална система от борси, която позволява свободното движение на капитал в целия свят. Тази система улеснява компаниите и правителствата в процеса им на привличане на финансиране от обществото. Участниците на капиталовите пазари могат да бъдат инвестиционни банки, брокери, пенсионни фондове, взаимни фондове, застрахователни компании и хедж фондове. Могат индивидуално да участват в пазара и физически лица. Участниците на капиталовите пазари се наричат търговци или борсови спекуланти. Финансовите инструменти са индивидуални активи, които се търгуват на капиталовите пазари, като се купуват и продават на борсата от участниците в нея.

Темата за капиталовите пазари и тяхната дигитализация е актуална още от 80-те години на двадесети век и до днес. Тя е важна, защото дигитализацията осигурява достъп на повече необразовани в конкретната материя хора до пазарите, с което се увеличава рискът от манипулации чрез фалшиви новини, грешна образователна информация и изопачени икономически данни. Участниците на капиталовите пазари днес срещат проблеми и с огромния обем от дистрибутирана на различни платформи информация, ограничено време за работа и липса на специализирана платформа, отговаряща на техните нужди. Затова основните цели на платформата, са:

1. спестяване на време чрез автоматизиране на повтарящите се всекидневни задачи на търговците;
2. централизиране и филтриране на висококачествена информация.

Тези цели ще бъдат постигнати, ако платформата, включва следните функционалности:

1. достъпване на източник на достоверни новини, на календар с важните актуални събития и на икономически и финансови данни;
2. осигуряване на затворена и конфиденциална социална мрежа от спекуланти, базирана на групи, чрез която търговците да обменят знания, прозрения и мнения помежду си;
3. водене на счетоводна книга за търговските поръчки и на записки, свързани с психологическото състояние на спекуланта по време на търговия;
4. достъпване на актуални котировки на финансови инструменти;
5. калкулатор за изчисляване на възвръщаемостта на търговска позиция.

Благодарение на това, че платформата, е тясно ориентирана, потребителското мнение под формата на предложения и оценки за функционалностите на софтуера ще бъдат разглеждани и обсъждани с финансисти и софтуерни инженери. На база финансовото и инженерното експертно мнение ще се правят решения дали потребителските желания, изразени като мнения, да се имплементират в платформата или не. Това ще даде възможност на спекулантите потребители на платформата да имат частичен контрол върху функционалностите ѝ. Този процес е невъзможен при повечето съществуващи платформи, които търговците използват днес, тъй като голяма част тях не са предназначени само за работата на борсовите спекуланти.

Задачите за създаване на гореописаните функционалности са:

1. да се проучат източници и начини за събирането на данни като новини, календарни събития, икономически показатели котировки на финансови инструменти, за да се създаде и тяхна визуализация;
2. да се дигитализира счетоводен дневник за търговски позиции;
3. да се дигитализира дневник за емоционалното състояние по време на работа на борсовия спекулант;
4. да се имплементират чат стаи, в които търговците ще обменят писмено информация в реално време;
5. да се разработи калкулатор за изчисляване на възвръщаемостта на търговски позиции чрез математически формули;
6. да се разработят функционалности за управление на потребители като регистрация, оторизация и опции за редактиране на потребителски профил.

**Глава II.  
Изложение**

1. **Съществуващи решения**

В днешно време съществуват платформи за новини като Newsquawk и Financial Times, калкулатори за изчисляване на възвръщаемостта на търговска сделка като тези на StoneX Financial, уебсайтове като Trading Economics, които да визуализират икономически и финансови показатели, и социални мрежи като Microsoft Teams и Facebook, чрез които хората могат да комуникират. Много източници на образователно съдържание за това как се спекулира и инвестира на капиталовите пазари могат да бъдат открити в интернет. Създадени са отдавна и софтуерни разработки като Ексел на Майкрософт, които се използват за счетоводна дейност. Но всички тези приложения се поддържат от различни компании с различни цели, което води до невъзможността те да работят заедно върху желанията на общ за всички тях тесен кръг от потребители (спекулантите на капиталовите пазари). Най-често предназначението на тези платформи не е фокусирано върху улесняването на работната дейност на търговците. Тези две причини водят до реалността, в която борсовите спекуланти трябва да се адаптират със софтуерните приложения, които съществуват, без да имат възможността да поискат промени по функционалностите или визуализацията. Тази нежелана адаптация е една от причините за създаването на платформа, която обединява всички, необходими на търговците, инструменти за работа и източници на информация.

Платформата ще бъде предназначена за анализиране на капиталовите пазари, за информиране на техните участници и за следене на производителността и психологическото състояние на търговците по време на работа. Именно заради тясната ниша на платформата потребителите могат да имат частичен контрол върху нея чрез заявки за функционалности или визуализация. Това е невъзможен процес при повечето съществуващи софтуерни решения.

1. **Предпроектно проучване**

В този раздел ще бъде представено накратко предпроектното проучване и изводите от него, които бяха направени преди да бъде започнато създаването на софтуерния дизайн и имплементацията на платформата за участниците на капиталовите пазари. Резултатите от проучването ще послужат за решаването на първата задача, представена в уводната част на страница 4.

Преди началото на изграждането на софтуерен дизайн и реализацията на платформата за участниците на капиталовите пазари, трябва да се направи анализ на функционалните изисквания към платформата. Те са декларирани в уводната част на този научен труд на страница 3 и 4, където са изброени какви функционалности са необходими, за да се постигне целта на платформата. Анализът върху функционалните изисквания е необходим, за ​​да се провери осъществимостта на разработването на дадения софтуерен продукт. След като се определи дали проектът е технически и финансово осъществим, се създава документ за спецификация на софтуерните изисквания (Software Requirement Specification - SRS). В него се описват функционални и нефункционални изисквания към софтуера, както и неговия обхват. Документът е необходим, за да се вземат правилни решения при създаването на софтуерния дизайн и за да бъдат дефинирани задачи, които да се изпълнят по време на имплементацията на платформата. Този научен труд - дипломният проект - може да бъде частично разглеждан като документ за спецификация на софтуерните изисквания (Software Requirement Specification – SRS).

Най-важната стъпка от цялото предпроектно проучване е да се подберат качествените източници на икономически и финансови данни, новини, актуални събития, параметри на финансови инструменти и котировки на търгувани активи. Без качествени данни, платформата е лишена от причина да съществува, тъй като никой участник на капиталовите пазари няма да изложи средствата си на риск, ако знае, че той работи с некачествена информация. Затова въпросът за данните е първи, защото отговорът ще определи дали реализацията на платформата е осъществима.

Най-популярните източници на новини са: Yahoo Finance, Investing.com, Newsquawk и Benzinga PRO. Това са и едни от най-легитимните официални платформи за новини. От изброените доставчици на новини Newsquawk и Benzinga PRO са най-бързи и най-качествени. Ползването на техните услуги обаче изисква средства, които надхвърлят бюджета на проекта. Поради ограничените финансови условия, изборът на доставчик на данни е между Yahoo Finance, Investing.com. Според наблюденията на група търговци Investing.com са по-бързия доставчик на новини. Следователно Investing.com ще бъдe източникът на новини при реализацията на платформата.

Източникът на икономически и финансови данни ще бъде базата данни на системата от централни банки на Съединените американски щати. Името на базата данни е „Federal Reserve Economic Data(FRED)“. Изборът за този източник се базира на легитимността на тази институция.

Източникът на декларирани отворени позиции на пазарите на деривати, които са необходими за направата на качествени анализи от професионалните търговци, ще бъде Комисията по търговски деривати на суровини - Commodity Futures Trading Commission(CFTC). Изборът за този източник се базира на легитимността на тази институция.

Източникът на параметри за деривативните финансови инструменти ще бъде Чикагската стокова борса - Chicago Mercantile Exchange (CME)“. Изборът за този източник се базира на легитимността на тази институция.

Източникът на актуални събития и на котировките на финансови инструменти ще бъде платформата TradingView, защото предлага надеждна и рентабилна информация. Има и платени доставчици на данни, които са по-достоверни, но тяхното ползване се заплаща, а първоначалният бюджет не покрива разходите.

**Таблица 1. Използвани външни източници на данни**

|  |  |
| --- | --- |
| Източник | Какви данни предоставя? |
| TradingView | Актуални котировки на множество от активи; Календарни събития |
| Federal Reserve Economic Data | Икономически и финансови показатели |
| Commodity Futures Trading Commission | Доклади за декларираните отворени позиции на деривативните пазари |
| Investing.com | Актуални новини |
| Chicago Mercantile Exchange | Спецификации по деривативните договори; математически формули за изчисляване на възвръщаемостта на търговска позиция |

След като източниците на качествени данни са подбрани, то следва да се проучи дали технически е възможно да се разработи уеб платформа, която, освен да събира и да визуализира тези данни от външни източници, може да предлага услуги като частни чат стаи, калкулатори, персонални акаунти и дигитализирани опростени счетоводни книги. Чрез създаването на база от данни и използването на обектно-ориентиран език, скриптов език, уеб протоколи за комуникация, допълнителни библиотеки и технологична рамка за разработка на уеб приложения може да бъде създадена платформа, улесняваща ежедневната дейност на професионалните участници на капиталовите пазари.

Тъй като успешно са подбрани достъпни източници на качествена информация и е възможно да се имплементира платформа, достъпвана по интернет, то софтуерът може да се определи за технически и финансово осъществима.

1. **Архитектури, база данни, парадигми и дизайн шаблони**

Жизненият цикъл за софтуерната разработка гласи, че след предпроектното проучване трябва да бъде направен дизайн на софтуерния продукт. В този раздел ще бъдат разгледани концепции, използвани при реализацията на платформата за участниците на капиталовите пазари.

Създаването на дизайн на софтуерен продукт включва решения за това каква мрежова архитектура да бъде използвана, какви парадигми и дизайн шаблони да бъдат приложени, какви принципи и протоколи да бъдат следвани, на какви модули от високо и от ниско да бъде разделен софтуерът и какви типове бази данни ще бъдат необходими.

* 1. **„Клиент – сървър“ мрежова архитектура**

**М**режова архитектура, върху която ще се базира платформата, е „клиент-сървър“. Тя дефинира две основни роли: клиент и сървър. Тяхната комуникация се изпълнява посредством заявки и отговори. Тази архитектура е отлична за целта на платформата, защото сървърът ще бъде ангажиран с изпълнение на конкретна услуга само след получаване на заявка от клиент, а не постоянно. Редът на обмен на информация между „клиент-сървър“ е:

1. Клиентът подава заявка за определена услуга към съответния порт на сървъра
2. Сървърът изпълнява конкретната заявена услуга
3. Сървърът връща евентуален отговор (възможно е да не изпрати такъв в следствие на прекъсване на връзката)
4. Клиентът получава отговора

Този модел на работа позволява използването на различни протоколи за комуникация като HTTP и WebSocket, които ще бъдат разгледани в раздел „HTTP и WebSocket протоколи“.

* 1. **Трислойна архитектура**

В софтуерното инженерство многослойната архитектура (позната още като N-слойна архитектура) е архитектура от типа клиент-сървър, в която потребителският интерфейс, бизнес логиката на приложението и съхранението на данните са логически разделени на модули. Най-разпространената форма на многослойна архитектура е трислойната архитектура. На нея ще се базира платформата.

Причината за използването на многослойната архитектура е, че тя позволява създаване на гъвкави приложения. При разделянето на едно приложение на слоеве добавянето или променянето на отделен слой е възможно, без необходимостта да се преработва цялото приложение. Начинът на работа на отделните модули е независим.

Трислойната архитектура е изградена от презентационен слой (наричан още потребителски интерфейс), който служи за прякото взаимодействие с потребителя и изпращането на заявки към бизнес слоя. Не трябва да има директна връзка между презентационния слой и слоят за бази данни. Бизнес слоят контролира функционалността на приложението като извършва различни процеси по обработката на данните. Той комуникира и с презентационния слой, и с базите данни. Слоят за данни е третият слой. Той служи за съхранение данни и комуникира само с бизнес логиката.

Платформата за участниците на капиталовите пазари, ще спазва принципите на традиционния трислоен модел. Използвайки този подход, потребителите правят заявки през графичен интерфейс(презентационен слой), който взаимодейства само със слоя с бизнес логиката. Слоят с бизнес логиката може да достъпва слоя за данни. Един недостатък на този традиционен подход е, че презентационният слой зависи от слоя с бизнес логиката, който пък зависи от слоя за данни. Това означава, че слоят с бизнес логиката, който обикновено съдържа най-важната програмна логика в приложението, зависи от подробностите за изпълнение на достъпа до данни.

Diagram, timeline

Description automatically generated

**Фиг. 1 Традиционен трислоен модел**

Алтернатива на традиционния трислоен модел е Изчистената архитектура (Clean architecture). Тя поставя бизнес логиката в центъра на приложението. Вместо бизнес логиката да зависи от достъпа до данни или други инфраструктурни проблеми, тази зависимост е обърната: детайлите на инфраструктурата и изпълнението зависят от ядрото на приложението (бизнес логиката). Тази функционалност се постига чрез дефиниране на абстракции или интерфейси в ядрото на приложението, които след това се имплементират от типове, дефинирани в инфраструктурния слой. Слоят с бизнес логиката е толкова независим, че ще продължава да функционира правилно дори при драстични промени като смяна на двигателя на база данни, на технологична рамка или на библиотека.

Diagram

Description automatically generated

**Фиг. 2** **Изчистената архитектура (Clean architecture)**

Причините за избора на традиционния трислоен модел пред изчистената архитектура са две:

1. Изчистената архитектура изисква много предварително програмиране преди действително да се приложи софтуерно решение. Липсата на достатъчно време за имплементация подтиква към по-бързата за реализация архитектура (традиционния трислоен модел).
2. Не се предполага в бъдеще промяна на системата за управление на базата данни (промяна на системата за управление на базата данни е удобна при изчистената архитектура).

Diagram

Description automatically generated

**Фиг. 3 Примерна диаграма на етапите на заявка в трислойната архитектура**

* 1. **Уеб API**

Преди разяснението „Какво е уеб API и защо е необходим?“ е важно да се определи как на практика платформата ще бъде реализирана спрямо принципите на трислойната архитектура. Тя ще бъде разделена на две отделни приложения. Едното ще е сървърно, другото – клиентско. Разликата между тях е, че сървърното приложение ще приема заявки без значение дали източникът е уеб клиент, мобилно приложение или дори друг уеб сървър. Сървърното приложение поема отговорността на слоя с бизнес логиката и отговорността на слоя за данни. Клиентското приложение ще бъде зареждано и изпълнявано в рамките на уеб клиента на потребителя (ще се стартира в браузъра). Сървърното приложение приема ролята на уеб API. Но какво е уеб API?

Общото определение за API: Това е приложно програмен интерфейс (application programming interface - API), който представлява набор от дефиниции на подпрограми, протоколи и инструменти за изграждане на софтуерни приложения. Казано с прости думи, API е интерфейс, който има множество от функции, които позволяват на програмистите да имат достъп до специфични функционалности или данни на приложение. Уеб API, както подсказва името, е API в мрежата, който може да бъде достъпен по HTTP или WebSocket протокол. Важно е да се отбележи, че това е концепция, а не технология. Можем да изградим уеб API, използвайки различни програмни технологии като Java, .NET и др.

Причината за разделянето на платформата на две отделни приложения и съответно имплементацията на сървърното приложение като уеб API е независимостта на сървърното приложение (сървъра от „клиент-сървър“ мрежовата архитектура) от типа клиент (браузър, мобилно приложение или друг уеб сървър). Това е необходим подход на имплементация, защото в бъдеще потребителите могат да поискат не само достъп до платформата чрез уебсайт, но и чрез мобилно приложение или ще заявят желание просто да достъпват данните, които се доставят от сървърното приложение (без визуализация). Директна комуникация със сървърното приложение за достъп до данни и функционалности, реализираща се без графичен интерфейс, е позната като „Машина към машина“ („machine-to-machine“). Тази комуникация е силно разпространена в днешно време и това дава стимул на решението за имплементация на уеб API.

Независимостта на сървърното приложение ще бъде постигната, като то е имплементирано да генерира отговори с ресурси във формат JSON, които може да бъде получен, запазен, зареден и прочетен от всяка популярна операционна система. Сървърното приложение не трябва да отговаря за графичния интерфейс, следователно не трябва да връща статични файлове като HTML, CSS, JS, XAML, снимки и др. Това е отговорност вече на клиентското приложение. Сървърното приложение трябва да доставя нужните данни и услуги за създаване, актуализиране и изтриване на данни (CRUD операции) и не трябва да отговаря за презентационната логика. Това е основната причина за разделянето на платформата на сървърно и клиентско приложение.

Този подход ще позволи и създаването на модерни едностранични приложения с добро потребителско преживяване, но този аспект ще бъде разяснен подробно в подраздел „Едностранични приложения“ на страница 13.

Diagram

Description automatically generated  
**Фиг. 4 Уеб API, „Клиент-сървър“ мрежова архитектура и трислойна архитектура, обединени в една диаграма**

* 1. **Едностранични приложения**

Едностраничното приложение (Single-page application - SPA) е уеб приложение, което работи изцяло в уеб браузъра и зарежда само един документ. То не изисква опресняване на страницата от страна на браузъра по време на употреба, а по-голямата част от DOM(Document Object Model) дървото остава непроменено, тъй като само малка част от него трябва да бъде актуализирана при определено събитие. Когато съдържанието на страницата трябва да бъде променено, едностраничното приложение използва API на JavaScript. Потребителите могат да получат достъп до уебсайта, без да се налага постоянно да теглят цяла нова страница при промяна на URL адреса. В резултат на това производителността и потребителското преживяване се подобряват.

Едностраничните приложения работят по следния начин: Когато потребител въведе URL адрес на уеб страница в браузъра, за да поискаме достъп, браузърът прави заявката до определения в URL адреса сървър, който отговаря с HTML документ и други статични файлове като Cascading Style Sheets (CSS) и JavaScript. Сървърът доставя HTML съдържание и други статични файлове само с първата заявка. Всяка следваща заявка до сървъра обаче връща данни в JSON или XML формат, но не и HTML или други статични файлове, свързани с презентационната логика на уебсайта. Тези последващи заявки се приемат от уеб API, който представих в подраздел „Уеб API”. Отговорите на уеб API пораждат промени по страницата на едностраничното приложение. Тези промени не изискват опресняване на страницата от уеб браузъра, тъй като се изпълнява JavaScript код (който е получен при първата HTTP заявка).

Тяхната алтернатива – многостаничните приложения – получават от уеб сървъра HTML файл, който те трябва да изобразят в браузъра. Този нов HTML файл налага опресняване на страницата от уеб браузъра.

Клиентското приложение за уеб браузъри на платформата за участниците на капиталовите пазари ще бъде едностранично. Поради това, ще бъдат представени ползите и недостатъците на едностраничните приложения (SPA).

**Таблица 2. Ползите и недостатъците на едностраничните приложения**

|  |  |
| --- | --- |
| **Ползи** | **Недостатъци** |
| По-добро потребителско преживяване | Потенциално по-дълго време за първоначално зареждане |
| Подобрена скорост | Неефективно SEO |
| Използване на по-малко интернет ресурс |  |
| Ефективно кеширане |  |

Недостатъкът „Потенциално по-дълго време за първоначално зареждане“ е пренебрежимо малък при правилното използване на технологичните рамки за създаване на едностранични приложения. А недостатъка „Неефективно SEO“ е решим с SEO техники за оптимизиране.

Поради съществуването на ползи и на решения на недостатъците на едностраничните приложения по-доброто решение за клиентската част на платформата е тя да се базира на концепцията за SPA.

* 1. **Обектно-ориентирано програмиране**

След представянето на използваните архитектури и абстрактните концепции е редно да се разгледат програмни парадигми и принципи. В този подраздел ще бъдат представени накратко трите най-популярни парадигми в програмирането, избраната парадигма за имплементация и обосновка защо тя е предпочетена. Тези три най-популярни парадигми са: процедурно програмиране, обектно-ориентирано програмиране и функционално програмиране.

Процедурното програмиране директно инструктира хардуерното устройство какви логически стъпки последователно трябва да се извършат. Тази парадигма използва линеен подход, като третира данните и процедурите като две различни свои структурни единици. Процедурното програмиране разделя програмата на процедури (подпрограми или функции), съдържащи поредица от стъпки, които трябва да бъдат извършени. Казано синтезирано, процедурното програмиране е създаване на списък с инструкции, които се изпълняват стъпка по стъпка, за да се свърши дадена логическа задачата.

Обектно-ориентираното програмиране (ООП) е наследник на процедурното програмиране. То разглежда компютърната програма като реалния живот, изпълнен с разнообразни обекти, които имат различни характеристики. ООП се базира на множество от обекти и тяхното взаимодействие помежду им. Всеки обект е самоустойчив, защото съхранява в себе си всички нужни променливи, които го характеризират и методи, представляващи действия, които той може да извършва. Най-важните принципи в ООП са наследяването, капсулирането, полиморфизмът и абстракцията.

Функционалното програмиране се различава от процедурното и обектно-ориентираното програмиране по това, че използва концепцията за чисти математически функции, чрез които резултатите от операциите се базират само на въведените входни данни на функцията без да извикват външни променливи (познати като състояние на програмата).

Diagram

Description automatically generated

**Фиг. 5 Типове програмни парадигми**

Обектно-ориентираното програмиране е най-подходящата парадигма за реализацията на дипломния проект, защото:

1. Капсулирането на класовете предоставя модулиране на приложението, което води до по-лесно управление и дебъгване на кода
2. Бъдещи добавяния към проекта не оказват влияние върху вече съществуващия код (стига да се спазват SOLID принципите, представени в подраздел „SOLID принципи“ на страница 17)
3. Повторяемостта на кода се намалява чрез преизползване и наследяване на обекти
4. Разделянето на отговорности (като основно правило в програмирането) между обектите е най-лесно осъществимо
5. Имплементацията на решения, които ще се използват в реалния свят, се реализира по-лесно чрез ООП, защото то се базира на реалния живот

Чрез обекти могат да се представят различните финансови активи, акаунти на потребители, новини, актуални събития, книги (или друг вид обучително съдържание за секция образование в платформата), съобщения по чат стаите, самите чат стаи и записите в счетоводните книги. Разглеждайки по-абстрактно обектите, всички клиентски заявки, услуги, които предоставя уеб API-а на платформата, и техните резултати могат да бъдат разглеждани като обекти.   
 Това са причините за използването на парадигмата „Обектно-ориентирано програмиране“ за имплементацията на платформата.

* 1. **SOLID принципи**

Чисто философски погледнато, принципите съществуват, за да помагат, а не за да създават неудобство. SOLID принципите не странят от този философски светоглед.

SOLID принципите са пет принципа за проектиране на обектно-ориентиран клас. Те са набор от правила и най-добри практики, които програмистите трябва да следват, докато проектират структурата на даден клас.

Причината да бъдат спазвани тези принципи е намаляването на зависимости между класовете, така че промяната в една класова структура от софтуера да не засяга друга. Освен това SOLID принципите са предназначени да направят дизайна на софтуерния продукт по-лесен за разбиране, поддържане и разширяване.

Естествено е всяко нещо да има предимства и недостатъци. Макар и SOLID принципите да имат много предимства, тяхното спазване обикновено води до писане на по-дълъг и по-сложен код. Това означава, че може да се увеличи времето за проектиране и имплементиране, което прави разработката малко по-трудна. Това допълнително време и усилия обаче си заслужават, защото SOLID принципите допринасят за много по-лесното поддържане, тестване и разширяване на софтуера в бъдеще.

SOLID означава:

1. S – Single Responsibility Principle
2. O – Open-Closed Principle
3. L – Liskov Substitution Principle
4. I – Interface Segregation
5. D – Dependency Inversion

Single Responsibility принципът гласи, че при проектирането на клас, трябва да се гарантира, че той отговаря само за 1 задача или функционалност и когато има промяна в тази задача/функционалност, само тогава този клас трябва да бъде променян. Приносът на този принцип се изразява в това, че кодът по-лесно се поддържа, разбира, тества и дебъгва. Това води съответно до по-малко дефекти.

Open-Closed принципът гласи, че класът трябва да може да разшири поведението на друг клас, без да го променя. Този принцип разделя съществуващия код от модифицирания, за да осигури по-добра поддръжка, като минимизира промените в кода. Казано с други думи, добавянето на нов код не трябва да изисква драстична промяна на съществуващ такъв.

Liskov Substitution принципът гласи, че всеки клас, който е дъщерен на бащин клас, трябва да е способен да замества своя родител без неочаквано поведение. Този принцип помага да се избегне неочаквано поведение след имплементация на нови промени по кода и премахва необходимостта от редакция на много класове, за да се реализират тези нови промени.

Interface Segregation принципът гласи, че интерфейсите, с които се постига абстракция в ООП, трябва да бъдат разделени на малки части, всяка от които да има само една специфична отговорност. Трябва да се предпочитат малки конкретни интерфейси вместо големи, обобщени такива. Този принцип улеснява взимането на решение кои интерфейси да бъдат имплементирани в даден клас, без да се имплементират излишни в него функционалности.

Dependency Inversion принципът гласи, че модулите(класовете) от високо архитектурно ниво не трябва да зависят от модулите (класовете) от ниско архитектурно ниво. И двете групи модули трябва да зависят от абстракции. Това позволява промяна на класове от по-високо или по-ниско архитектурно ниво, без да бъдат засегнати други класове. Тези други класове ще се променят само ако се промени тяхното абстрактно представяне.

* 1. **Релационни бази данни**

Без база от данни, в която да се запазва информация за потребителите, детайли за финансовите инструменти, съобщения, групи и обучителни материали, платформата няма да е надеждна и не може да съществува пълноценно. Съществуват 2 типа бази данни: релационни и нерелационни. За разработката на дипломния проект е необходима релационна база данни, защото съществуват връзки между моделите (потребители, счетоводни книги, групи, тикер символи, борси и др.). Например: потребители – счетоводни книги, счетоводни книги – търговски позиции, потребители – група, група – съобщения и др.

В бъдеще може да бъде добавена и нерелационна база данни по необходимост.

Софтуерът, чрез който се създава и поддържа релационна база данни, се нарича система за управление на релационни бази данни.

Основна структурна единица на релационните бази данни са релациите. Това са групи от записи, които имат едни и същи атрибути. Всяка релация формира таблица, съставена от атрибути (познати още като полета или колони) и записи (познати още като кортежи или редове).

Друга важна концепция в релационните бази данни са ключовете. Те представляват един или повече атрибути от една таблица, които носят специфично предназначение. Най-важните видове ключове спрямо нуждите на дипломния проект са първичните(primary) и външните(foreign) ключове.

Първичният ключ служи като уникален идентификатор на всеки запис в дадена таблица.

Външният ключ служи за референция от една таблица към уникален идентификатор (първичен ключ) в друга таблица.

Чрез двата вида ключове се създават отношения между две релации. Отношенията между таблиците се наричат зависимости, чието установяване е необходимо за реализацията на платформата за участниците на капиталовите пазари.

Последната фундаментална концепция, без която базите данни са неприложими, са операциите, които могат да бъдат извършвани върху една база от данни. Те са добавяне, четене, актуализиране и изтриване (още познати като Create Read Update Delete – CRUD операции).

Полезен начин за представяне на структурата на релационна база данни е чрез ER(Entity Relationship) диаграми.

Diagram

Description automatically generated

**Фиг. 6 Примерна ER диаграма**

Базата данни и съответно ER диаграмата са обект на промяна в процеса на надграждане на платформата.

* 1. **Обектно-релационно картографиране**

След като бяха представени парадигмата за обектно-ориентирано програмиране и релационната база данни, е време да бъде представен начинът, по който ще бъдат свързани тези две концепции. Той се нарича обектно-релационното картографиране.

Това е техника, при която чрез обектно-ориентиран код се работи с релационна база данни. Това е възможно благодарение на дескриптор, който чрез метаданни за обектите сформира структурни връзки с релации (т.нар. mapping).

Oбектно-ориентираният код трябва да бъде написан на обектно-ориентиран език, каквито са например C# и Java.

Oбектно-релационното картографиране е познато още като (Object Relational Mapping - ORM).

Най-общо казано чрез ORM се генерират SQL заявки, базиращи се на обектно-ориентиран код.

Причината за използването на обектно-релационното картографиране са многото предимства и малкото решими недостатъци, които тази техника предоставя.

**Таблица 3. Ползите и недостатъците на ORM**

|  |  |
| --- | --- |
| Ползи | Недостатъци |
| Автоматизиран начин на преобразуването от обект в таблица и таблица в обект | По-сложни заявки могат да доведат до проблеми с производителността |
| По-малко програмен код (заради липсата на SQL заявки) | Някои заявки могат да бъдат много бавно генерирани |
| Оптимизиране SQL заявките |  |
| Предпазване от SQL Injection атаки |  |
| Кешира резултати |  |

* 1. **Дизайн шаблони**

В софтуерното инженерство с дизайн шаблон се назовават обобщени и повтарящи се решения на често срещани проблеми в софтуерния дизайн. Един шаблон не е завършено решение на проблема, а е план, по който да се имплементира решението.

Ползите от дизайн шаблоните са две:

1. Те са проверени и доказани (легитимни) решения на често изникващи проблеми в обектно-ориентирания дизайн.
2. Научават програмиста как да решава всякакъв вид проблеми в процеса на създаване на софтуерен дизайн.
3. Служат като основен език при комуникация между колеги програмисти. Един софтуерен инженер може да каже „А, тук може да се използва Декоратор (дизайн шаблон)!“ и неговите да го разберат, без да има нужда инженерът да обяснява какво е „Декоратор“.

За имплементацията на платформата ще бъде необходим дизайн шаблона „Dependency Injection“ (Инжектиране на зависимости). Зависимостите са необходими параметри за работата на един обект или функция.

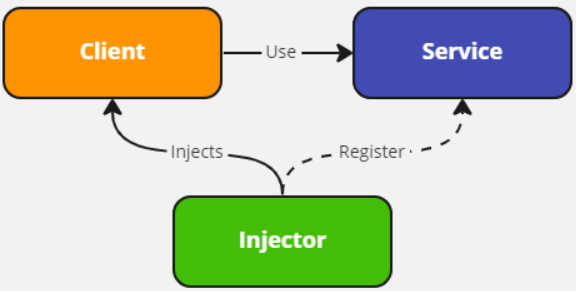
Този шаблон подсигурява това, че конструкторът на обект или самата функция няма да инстанцират зависимостите си. Тяхната отговорност ще бъде само да извършват конкретна услуга. Фундаментално погледнато, Dependency Injection дизайн шаблонът се състои в това да подава параметри на конструктор или функция (наречени още клиенти) чрез външен за класа код (наречен още injector). Този процес е възможен благодарение на използването на интерфейси при дефинирането на параметрите на конструктор или функция.

Дизайн шаблонът „Dependency Injection“ се състои от четири роли:

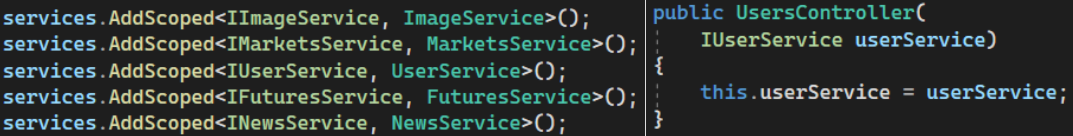
1. Service – зависимост (клас), която ще бъде използвана
2. Client – клас, които ще използва Service (зависимостта)
3. Interface – използван като посредник между Client и Service, като е имплементиран от Service
4. Injector – създава инстанция на Service по съответен Interface и я „инжектира“ (подава като параметър) на Client

Дизайн шаблонът „Dependency Injection“ работи по следния начин:

1. Създават се зависимостите под формата на интерфейс (Interface) и клас (Service), който имплементира интерфейса.
2. Създава се контейнер с регистрирани зависимости (всеки „Interface” се свързва със „Service“), от който „Injector”-ът ще „взима“ (инстанцира) обекти.
3. Създава се друг клас (Client), който ще използва „Service” чрез дефиниране на неговия „Interface“ в себе си.
4. При инстанциране на обект (Client), „Injector“-ът ще достави обект зависимост (Service) като параметър на конструктора на инстанцирания се обект (Client).



**Фиг. 7 Диаграма на Dependency Injection дизайн шаблона**



**Фиг. 8 Имплементация на Dependency Injection в платформата;  
регистрация на зависимости (вляво),  
 инжектиране на зависимости (вдясно)**

* 1. **Асинхронно програмиране**

За да бъде представена асинхронността в платформата за участниците на капиталовите пазари, трябва първо да се представи разликата между пет понятия в сферата на компютърните науки. Те са: нишки, синхронност, асинхронност, конкурентност и паралелизъм.

Нишката е последователност от изпълнение на код, който може да се изпълнява независимо от други нишки. Това е най-малката единица от задачи, която може да бъде изпълнена от операционна система. Една програма може да бъде еднонишкова или многонишкова.

Конкурентност означава, че няколко нишки са изпълнявани в един и същ времеви период, но не е задължително те да се изпълняват едновременно. Нарича се конкурентност, защото изпълняваните нишки се конкурират за хардуерни ресурси, които са им необходими. Конкурентността може да се постигне чрез процеса „Смяна на контекст“ или чрез паралелизъм.

Паралелизъм означава, че няколко нишки са стартирани в един и същ времеви период и се изпълняват едновременно. Необходими са поне две процесорни ядра за паралелизъм.

Тоест на компютърна машина с едноядрен процесор може да се постигне конкурентност само чрез процеса „Смяна на контекст“, докато на компютър с многоядрен процесор конкурентността се реализира чрез паралелизъм.

Синхронният модел на програмиране гласи, че задачите на една програма се изпълняват една след друга. Всяка задача изчаква всяка предишна задача да завърши и след това тя започва да се изпълнява. Асинхронният модел на програмиране гласи, че докато една задача се изпълнява, може да се превключи към изпълнение на друга задача, без да се изчаква предишната да бъде завършена.

Накратко казано, синхронното и асинхронното програмиране са модели на програмиране, докато конкурентността и паралелизмът реферират компютърната архитектура по отношение на това как се изпълнява дадено множество нишки.

Chart, bar chart

Description automatically generated

**Фиг. 9 Нишки, синхронност, асинхронност, конкурентност и паралелизъм**

Сървърното приложение на платформата за участниците на капиталовите пазари ще бъде имплементирано по модела на многонишковата асинхронност и ще бъде стартирано на многозадачна операционна система върху компютърна машина с многоядрен процесор, за да бъде възможен паралелизмът.

Ползите от асинхронността на уеб API, стартиран на многозадачна операционна система върху компютърна машина с многоядрен процесор, са:

1. По-бързо генериране на отговор по конкретна заявка.
2. Повече едновременно приети заявки за обслужване.
3. По-добро потребителско преживяване.
4. Използва максимално хардуерния потенциал на процесора.

Ползите от асинхронността на сървърното приложение на платформата за участниците на капиталовите пазари и липсата на нейни недостатъци са причините то да бъде имплементирано по модела на многонишковата асинхронност.

* 1. **Автентикация с JWT токени**

JWT (JSON Web Token) е стандарт за създаване на токени, в които да се съдържат твърдения („claims“) като, например, „автентикиран като george@emailexample.com“. Тези токени се генерират от сървъра и се пращат на клиента, който да ги съхранява. Те имат валидност, която предварително се задава. Тези токени са криптирани, като са подписани чрез частен ключ на сървъра при генерирането. (използван източник - 6)

JWT токените ще съдържат информация за потребителя като име, имейл и идентификационен номер. Те ще бъдат изпращани, когато потребител отваря платформата или изпраща заявки, които изискват авторизация. Чрез тях клиентът ще бъде идентифициран, когато това е необходимо при обработването на заявката.

Благодарение на JWT токените потребителят няма да бъде задължен да се регистрира всеки път, когато отвори платформата. Неговата автентикация ще бъде изисквана през определен период от време (след изтичане на токена).

Важно е да се уточни, че въведената парола в полетата за регистрация и автентикация ще се хешира в клиентското приложение преди тя да бъде изпратена от клиента към сървъра. Така ще се гарантира нейната сигурност при пътуването по мрежата. След това на сървъра ще се „посоли“(промени) и хешира пак. Хеширането на клиента ще се извърши чрез хеширащия алгоритъм SHA512. Хеширането и „посоляването“ като концепции и в частност на сървъра ще бъдат представени в подраздел „Допълнителни библиотеки на С#“ в раздел „Платформи, езици, библиотеки и протоколи“.

1. **Платформи, езици, библиотеки и протоколи**

Изборът какви инструменти за разработка да бъдат използвани, се базира на отговорите на следните въпроси: „Език от високо или от ниско ниво ще бъде необходим?“, „Би ли било полезно този език да бъде обектно-ориентиран?“, „Ще бъде ли необходима база данни и какъв тип трябва да бъде тя?“, „Ако приложението е уеб базирано, какви протоколи за комуникация трябва да се спазват?“ и „Какви допълнителни библиотеки и технологички рамки(„Фреймуъркс“, от англ. “Frameworks”) ще бъдат необходими?“. Частичен отговор на тези въпроси е предоставен в раздел „Архитектури, парадигми и дизайн шаблони“.

* 1. **C#, .NET CORE и ASP.NET CORE**

Езикът С# е избраният език за имплементация на сървърното приложение на платформа за участниците на капиталовите пазари. Той е компилиращ се, обектно-ориентиран и строго типизиран. Базира се на поддържаната от Microsoft платформа .NET CORE, която може да бъде стартирана на всякаква операционна система и получава подобрения всяка година. На платформата .NET CORE е базирана ASP.NET CORE уеб рамката за разработка на уеб приложения, която ще бъде използвана за разработването на уеб API като сървърно приложение.

Причините за използването на .NET и съответно C# са:

1. Поддържана и актуализирана ежегодно
2. Безплатна
3. Поддържа асинхронни многонишкови операции
4. Бърза компилация
5. Лесно дебъгване
6. C# e интуитивен обектно-ориентиран език
7. На .NET се базира най-бързата уеб рамка ASP.NET CORE

Причините платформата за участниците на капиталовите пазари да бъде имплементирана върху ASP.NET CORE са:

1. Най-бързата уеб рамка до този момент
2. Бърза при обработване на заявки и интуитивна при разработка
3. Лесна за дебъгване
4. Кросплатформeна (работи на различни операционни системи)
5. Мащабируема
6. Висока сигурност от XSS и CSRF атаки

A picture containing diagram

Description automatically generated

**Фиг. 10 Брой обработени заявки за секунда спрямо средата за изпълнение** (използван източник - 1)

Сравнявайки ASP.NET CORE 7.0 и неговата най-популярната алтернатива SpringBoot Java 18, Slavius констатира, че най-новата версия на ASP.NET рамката до момента на написване на дипломния проект е почти два пъти по-бърза в обработването на заявки при едни и същи условия. „Изглежда сървърът Kestrel с .Net 7 върши доста прилична работа с прост API в сравнение със сървъра Tomcat, базиран на Java 18 приложение SpringBoot.” – Slavius (използван източник - 3). Друг източник на данни за производителността е „techempower.com“, които се занимават с сравняване на уеб технологични рамки за разработване на уеб приложения. Според техният анализ от деветнадесети юли 2022 година, базиран на емпиричен метод на тестване, който включва сериализация в JSON формат, достъпване на база данни, композиране на шаблони от сървъра и др, като се изпълнява с реалистична продуктционна конфигурация, показва, че „asp.net core“ е на седмо място, докато spring - на шейсет и седмо място (използван източник - 4).

Многонишковата асинхронност в C# работи с нишки, колекция от нишки, наречена „Thread pool“, задачи, наречени „Tasks” и програма за управление на задачи, наречена „Task scheduler”.(използван източник - 2)

Нишките са последователност от изпълнение на код и най-малката единица за задача в операционната система. Те са описани в раздел „Архитектури, парадигми и дизайн шаблони“ в подраздел „Асинхронно програмиране“. Те приемат задачи „Tasks”, когато са „свободни“.

Колекция от нишки или Thread pool е множество от създадени нишки, които са „свободни“ за изпълнение на задачи. Когато една нишка приема задача за изпълнение, тя излиза от Thread pool-а, а когато приключи, се връща обратно в колекцията. Thread pool гарантира преизползването на нишки, без да се създават нови и да се трият стари. Това води до намаляване на използваната оперативна памет. (използван източник - 2)

Задачата („Task“) е обект, представляващ операция, която най-често се изпълнява асинхронно спрямо други операции. Тази операция се изпълнява на нишка от Thread pool-a. Операцията се нарича задача в контекста на приложението, но не и в контекста на операционната система, защото нишката е най-малката единица за задача в операционната система, не „Task”. Причината за използването на „Task” е, че позволява една нишка да приема нова операция за изпълнение, след като приключи предишната задача. Също, задачите (операциите) връщат стойност, което е невъзможно за нишките. Това се нарича многонишкова асинхронност базирана на задачи „Tasks“.

„Task scheduler” е програма, която управлява нишките и тяхното създаване, управлява Thread pool-a и задачите, които дадена нишка да изпълнява. (използван източник - 2)

Ползата от „Task“ базираната асинхронност е по-малко използвана оперативна памет и по-бързото изпълнение на операции.

* 1. **SQL и Entity Framework Core**

Платформата за участниците на капиталовите пазари не може да бъде полезна, без да се запазват данни в база данни. Както е спомената в раздел „Архитектури, база данни, парадигми и дизайн шаблони“ в подраздел „Релационни бази данни“, трябва да бъде създадена релационна база данни. Езикът за достъпване и манипулиране на такава база от данни е SQL(Structured Query Language). SQL e декларативен език, но не и програмен. Той не може да бъде разбран от компилатора на C#, което означава, че всякакви SQL заявки трябва да бъдат пазени като символен низ. Това е неудобен начин на робота и създава възможности за грешки. Този казус може да бъде решен с концепцията за обектно-релационно картографиране. Тя се интегрира в платформата за участниците на капиталовите пазари чрез технологичната рамка Entity Framework Core. Тази рамка предоставя и достъп до базата данни. Благодарение на нея, за реализирането на CRUD заявки към базата няма да се пишат на езика SQL, а на C#.

Като недостатъци Entity Framework Core е по-бавен от стандартния метод за изпълнение на SQL заявки, но с години оптимизация и с последната до този момент версия на технологичната рамка този недостатък е пренебрежимо малък. Друг рядко срещан недостатък е високата сложност за създаване на комплексни SQL заявки.

* 1. **HTTP и WebSocket протоколи**

HTTP и WebSocket са протоколи за комуникация между клиент и сървър. Върху тях ще се базира комуникацията между сървърното приложение – уеб API – и клиентското приложение. И двата протокола се намират на приложния слой от OSI модела.

OSI модела е концептуален модел, който описва начина на комуникация в компютърните мрежи. Той стандартизира проектирането на хардуер, операционни системи и протоколи.

Diagram

Description automatically generated

**Фиг. 11 OSI Модел (основният слой е „Физическия слой“)**

HTTP е протокол, работещ със заявки и отговори по модела клиент-сървър. Уеб браузърът е „клиентът“, докато уеб сървърът е „сървърът“. Клиентът изпраща съобщение по HTTP чрез заявка до сървъра. Сървърът, който предоставя ресурси като HTML файлове или друго съдържание, връща съобщение като отговор на заявката на клиента. Отговорът съдържа информация за статуса на завършване на заявката и може да съдържа желан ресурс в тялото на съобщението. Всяка HTTP заявка си има метод, който декларира желаното действие, което да бъде извършено върху идентифицирания ресурс. По някое време комуникацията между клиента и сървъра е прекъсната. По този протокол сървърът е статичен и не може да изпрати съобщение без заявка.

За повечето заявки на платформата за участниците на капиталовите пазари се използва HTTP протокола. Методите, които ще бъдат използвани са GET(взимане на съществуващ ресурс), POST(създаване на нов ресурс), PUT(подмяна на съществуващ ресурс с нов), DELETE(изтриване на съществуващ ресурс) и PATCH(частична промяна на съществуващ ресурс).

WebSocket протоколът осигурява двупосочна непрекъсната комуникация по TCP връзка между клиента и сървъра. Той ще бъде необходим за създаване на функционалности в реално време, каквато е чат стаята на платформата и съобщенията в нея. Създаването на този тип комуникация се случва по HTTP протокол с т.нар. handshake(ръкостискане). Когато настъпи промяна в данните на сървъра, клиентите ще получават съобщение с новите данни без да правят заявки.

Table

Description automatically generated

**Фиг. 12 HTTP и WebSocket протоколи**

* 1. **Уеб функционалности в реално време и SignalR**

Уеб функционалност в реално време е възможността сървъра да изпраща актуално съдържание на свързани с него клиенти.

Това е възможно по три различни начина:

1. Short polling – Реализира се по HTTP протокол. Клиентите изпращат постоянно заявки през фиксиран период от време, докато не получат отговор. Изисква много хардуерен ресурс от страна на сървъра. Не е ефикасен начин на имплементация.
2. Long polling – Реализира се по HTTP протокол. Клиентите изпращат заявка и чакат, докато не получат отговор или връзката прекъсне по HTTP протокол. Когато връзката се прекрати, се изпраща новия заявка, която също се изчаква до получаване на отговор или до прекъсване на връзката по протокол.
3. Server-Sent Events(SSE) – техника, позволяваща на клиент да получава автоматично актуални данни от сървър по HTTP протокол. Комуникацията е еднопосочна, от сървър към клиент.
4. Web sockets – двупосочна, непрекъсвана автоматично, комуникация. Този подход използва протокола WebSockets, описан в подраздел „HTTP и WebSocket протоколи“ в раздел „Платформи, езици, библиотеки и протоколи“.

Благодарение на тези подходи за комуникация между клиент и сървър, е възможна имплементацията на приложения като игри, GPS карти, работни табла за управление и наблюдение с актуални данни (в това число са и графики с актуални котировки), онлайн срещи и чат стаи.

SignalR е библиотека, която улеснява създаването на приложения с функционалности в реално време. Тя добавя слой абстрактност над начините за комуникация в реално време. SignalR поддържа и работи със следните подходи на комуникация:

1. WebSockets
2. Server-Sent Events
3. Long Polling

Редът на изброяване е реда, по който SignalR се опитва да създаде комуникацията. В някои случай например WebSockets протоколът няма да бъде поддържан от клиента и ще трябва да се премине към Server-Sent Events подхода. Ако и SSE не се поддържа, се преминава на стандартен Long Polling подход в комуникацията.

След като начина на комуникация е определен, трябва да се определи как клиентът и сървърът ще работят помежду си посредством нея.

SignalR работи по метода на „отдалечено извикване на процедури“ или „remote procedure calls (RPC)”. RPC работи по следния постъпков модел:

1. Клиент притежава процедури, които да бъдат извиквани от сървъра, като изчаква „нареждане“ от него.
2. Сървърът притежава процедури, които да бъдат извиквани от клиента, като изчаква „нареждане“ от него.
3. При определено събитие като промяна на данни на сървъра, сървърът извиква метод, притежаван от клиента. Или при определено събитие като клиентско действие, клиентът извиква метод, притежаван от сървъра.
4. Методът клиента/сървъра се изпълнява

Пример:

1. Потребител отваря уеб страница, която представлява чат стая.
2. Отваря се връзка за комуникация (по WebSockets протокола например) между браузъра (клиент) на потребителя и сървъра
3. Потребител въвежда съобщение в чата и го изпраща
4. Клиентът извиква „отдалечено“ процедура (метод) на сървъра за запазване и изпращане на съобщението до другите участници в чата
5. Сървърът изпълнява метода
6. Сървърът извиква „отдалечено“ метод на клиента за зареждане на новото/новите съобщения
7. Клиентът изпълнява метода
8. Всички виждат новото съобщение

SignalR използва концепцията за хъбове, които абстрактно представляват канала за комуникация между клиента и сървъра. Създаването, поддържането и затварянето на тези канали се поема от библиотеката. Така тя улеснява имплементацията, като я свежда до създаването само на процедурите (методите) на сървъра и на клиента.

За създаването на чат стаята ще бъде използван SignalR. За показване на актуалните котировки на активите като за начало ще бъде използван готов модел на TradingView. В бъдеще с достатъчно капитал, ще бъдат използвани данни от платени доставчици в текстови формат, като актуализацията на данните на уеб страницата с котировките на платформата за участниците на капиталовите пазари ще бъде имплементирана отново чрез SignalR.

* 1. **Допълнителни библиотеки на С#**

Събирането на данни за разпределението на текущите отворени позиции от докладите за декларираните отворени позиции на деривативните пазари може да бъде постигнато по два начина. Първият е чрез получаване на файл в текстови формат на данните за всяка седмица и съответно тяхното четене (интерпретиране) и запазване в базата. Този начин изисква работа с файлове, които са в трудно четим и склонен към промяна формат, и източник на тези файлове. Вторият по-достъпен и по-невремеотнемащ начин е да се намери сайт, който публикува тези данни и да се „scrape“-не (изтръгне) информацията от него. Това е възможно чрез автоматизирано зареждане на уеб страница и четене на конкретни HTML елементи, които съхраняват тази информация.

Този подход на събиране на данните е временен и ще бъде променен в по-нататъшни версии на платформата за участниците на капиталовите пазари, интегрирайки правителствен или частен API, чрез който да се събират, периодично и автоматизирано, данните без необходимостта от файлове или чужди сайтове.

Библиотеката, която ще бъде необходима за „изтръгването“ (или „scraping“-га), е AngleSharp. Чрез нея HTML елементите на заредената уеб страница източник могат да бъдат запазвани под формата на обекти по време на изпълнение на конкретна заявка. Така данните от сайта източник ще бъдат достъпвани и запазвани в базата данни и съответно визуализирани на уеб страницата на платформата за участниците на капиталовите пазари.

Друга полезна библиотека, чието използване e наложително, е Argon2id. Тя е необходима за хеширането и „посоляването“(salting) на пароли при регистрация преди те да бъдат запазени в базата данни. Причината за манипулацията на паролите преди запазването им в базата данни е тяхното предпазване от компрометиране. Действието е необходимо с цел увеличаване на сигурността на поверителността и на съществуването на профила на потребителя.

В чист вид („plaint text”) запазване на пароли не е добра практика. Причината е в това, че ако някой получи достъп до базата данни (дали авторизиран или не, не е от значение), той/тя ще може да прочете паролите на потребителите. Ако този човек с лоши намерения достъпва базата, цялата сигурност на приложението е компрометирана, а потребителите - застрашени.

Решението е да се хешират предварително „посолени”(променени) паролите на потребителите. Хеширането е алгоритъм, който се извършва чрез хеш функции. Те получават като параметър символен низ и връщат шифрован на него нов символен низ. Основната им характеристика е, че те са еднопосочни. Тоест от хешираната стойност на символен низ не може да се получи обратно нехеширания символен низ. Не съществува концепция като „дехеширане“.

Text

Description automatically generated

**Фиг. 13 „Посоляване“ и хеширане на текст (парола)**

Популярни хеш алгоритми са MD5 (message-digest algorithm 5) и SHA (secure hash algorithm). Тяхната популярност обаче ги прави опасни и несигурни за използване. За популярните хеширащи алгоритми се създават „rainbow” таблици (таблица дъга). Тя представлява предварително подготвена таблица „пищов“ с двойки „текст“-„хешираната стойност на текста“, чрез която може да се разбиват пароли по метода „brute force”. Речник от потенциални пароли, който може да включва често използваните такива, се превръща, автоматизирано чрез програма, в речник с хеш стойностите на тези потенциални пароли. Това обезсмисля хеширането на паролите на потребителите, ако се използват популярни и бързи хеш алгоритми. Трябва да се има предвид съществуването на тези речници с пароли, защото тяхната големина може да достигне до 1 терабайт информация. Това застрашава киберсигурността на системи, ползващи популярни и бързи хеширащи алгоритми.

Един непопулярен и бавно генериращ хеширащ алгоритъм е Argon2. Изборът за използването на този алгоритъм не се базира само на неговата непопулярност. Неговите разновидности като Argon2d, Argon2i и Argon2id предоставят допълнителна защита. (използван източник - 5)

1. Argon2d – увеличава издръжливостта към разбиването на пароли чрез графична карта(GPU), но създава риск от side-channel атаки
2. Argon2i – предпазва от side-channel атаки
3. Argin2id – е хибридна разновидност, която ползва частично Argon2d и Argon2i

Argon2id предоставя възможност и паролата да бъде асоциирана с други данни. Това прави атаката над паролите почти невъзможна, защото атакуващият системата трябва да има достъп освен до базата данни и до кода на платформата, за да разбере кои данни се използват за асоциация с паролите. (използван източник - 5)

Преди хеширането обаче трябва да се „посоли“(промени) паролата. „Посоляването“ представлява добавяне на допълнителни символи към паролата на конкретен потребител. За всяка парола се добавя различен символен низ, като той може да бъде конкатениран преди и/или след паролата. Полученият символен низ след конкатенация се хешира с хеширащ алгоритъм. Argon2id има вградена функционалност за „посoляване“ на парола.

* 1. **HTML, CSS и JS**

В този подраздел ще бъдат разгледани езиците, които ще бъдат използвани за създаването на визуализацията на платформата за участниците на капиталовите пазари.

HTML е система от анотации на документ, чрез която се дефинира съдържанието на уеб страници и тяхното оформление. Създаването на уеб съдържание се базира на HTML елементи. Чрез тях могат да бъдат визуализирани текст, снимкови и видео материали, таблици, формуляри и листове с данни.

Всеки HTML елемент се състои от таг с име, затворено между ъглови скоби („<име на таг>“) . Един елемент може да съдържа и атрибути, които представляват двойка ключ-стойност и се записват след името на тага в рамките на ъгловите скоби („<име на таг „име на атрибут“=“стойност на атрибута“>“). Те позволяват добавяне на допълнителна информация или функционалност към HTML елемента. Атрибутите не са задължителни за създаването на всеки HTML елемент. Някои елементи обаче трябва да имат дефинирани определени атрибути, за да бъдат визуализирани. HTML елементите съдържат готови вградени атрибути, но могат да бъдат създавани и персонализирани нови от уеб разработчик. Създаването на персонализирани атрибути е възможно чрез добавяне на „data-“ пред името на атрибута, който трябва да бъде създаден.

Примерен елемент би изглеждал по следните два начина:

1. <p>Текстов параграф, който ще се визуализира</p>
2. <img src=”source.png” \>

С „<име на таг>“ се задава начало на нов HTML елемент, а с „</p>“ се индикира неговия край. Между тях се съдържа друго уеб съдържание като текст или други HTML елементи, които са „вложени“ в зададения чрез таговете HTML елемент. Ако тагът е във формат „<img src=”source.png” \>“, то той е нарича самозатварящ се таг и той не може да съдържа вложени в себе си елементи.

Концепцията за влагане на елементи в един елемент създава дърво, което в компютърните науки се нарича „Document Object Model (DOM)“, или „документен обектен модел“. То представлява структура на един документ, каквато е HTML страницата.

Чрез HTML елементи могат да бъдат създадени компонентите на различните страници на платформата за участниците на капиталовите пазари като таблици, информационен текст, плочки, отделящи една информация от друга и визуализиране на графики и снимки.

HTML езикът обаче не позволява манипулирането на стила на страницата. Стилът на една страница включва размера, формата, цвета, дебелината, шрифтa и анимацията на всеки HTML елемент, както и подредбата на всички елементи.

За манипулацията на стила на една уеб страница се използва езикът CSS (Cascading Style Sheets). Чрез него с помощта на селектори, който използват атрибутите на даден HTML елемент, се променя стилизацията на този или тези елементи, които отговарят на съответния селектор. Например ако една уеб страница притежава HTML елемент: „<p class=”text”>Текст за визуализация</p>“ и CSS файл, в който е написан кода: „.text { font-size: 12pt; }”, то „Текст за визуализация“ ще се изобрази с дванадесет пункта големина на уеб страницата.

Една модерна уеб страница обаче трябва да бъде динамична, тоест променяща се при настъпването на определено събитие. Това става възможно чрез скриптовия програмен език JavaScript (JS). Това е програмен език, който може да бъде интерпретиран (а не компилиран) от уеб браузъра. Той се използва за манипулация на DOM дървото, което позволява промяна на структурата, съдържанието и стила на една уеб страница. Това се случва без да бъде зареждан нов HTML документ и съответно не е необходимо презареждане на уеб страницата. Това допринася позитивно към потребителското преживяване. Чрез JavaScript един уебсайт от една или няколко уеб страници се превръща в уеб приложение.

С езикът HTML се създава статичната структура на една уеб страница, със CSS – стила на тази страница, а с JS – динамиката ѝ.

* 1. **React - JavaScript библиотека**

Създаването на големи уеб приложения често води до създаване на еднотипни компоненти на различни места. За тяхната имплементация се използва почти идентичен код. Това поражда повтаряемост на HTML, CSS и JavaScript кода, което затруднява поддържането на части от уеб приложението, защото при бъдеща промяна във визуализацията на два или повече еднотипни на външен вид и функционалност компоненти трябва да се промени код на повече от едно място. От обектно-ориентираното програмиране може да бъда заимствана концепцията за класове и обекти, но перифразирана като модели на компоненти и компоненти.

Една уеб страница може да бъде моделирана чрез компоненти, които енкапсулират всички HTML елементи от дадено поддърво от DOM дървото. Тази енкапсулация се нарича компонент, който ще бъде структурната единица на едно уеб приложение. Когато един компонент бъде дефиниран, то той може да бъде използван на различни места в уеб приложението. Аналогия за компонентно базиране на уеб приложение може да бъде направено с дефинирането на класове, чрез които да се създават на различни места в програмния код обекти. Ползите от тази енкапсулация са:

1. преизползване на един и същ HTML, CSS и JavaScript код на няколко места в уеб приложението, по-бърза за имплементация;
2. задача, която изисква промяна по визуализацията или по функционалността на компонент, ще наложи промени само в рамките на дефиниционния блок на компонента, по-бързо свършена работа;
3. обединява блокове код на различни езици, които са концептуално свързани помежду си, по-лесно управление на файловата система на програмния код;
4. по-ниска вероятност от грешки при писането на JavaScript код.

За създаването на компонентно-базирано уеб приложение е наложително да се използва библиотека или технологична рамка, която на управлява компонентите така, че те да се изобразяват на уеб страниците и динамично да се променят. За създаването на платформата за участниците на капиталовите пазари ще бъде използвана библиотеката „ReactJS”. React работи на базата на концепцията за виртуално дърво, което е подобно на Document Object Model дървото на HTML документа в браузъра. Структурната единица виртуалното дърво обаче не е HTML елемент, а JavaScript обект. React манипулира тези обекти при определено събитие като отваряне на страница или натискане върху бутон. След като приключи с манипулацията по виртуалното дърво, React синхронизира DOM дървото чрез DOM API на браузъра по промените, направено върху виртуалното дърво. С този подход на работа React създава възможно най-малко промени по DOM дървото на уеб страницата, като целия този процес не изисква опресняване на браузъра или зареждане на нов HTML документ.

Редом със използването на React Библиотеката, е необходимо използването на един, надграден на JavaScript, език – JavaScript XML (JSX). Това е разширение на JavaScript езика, което позволява писането на HTML код в рамките на JavaScript файл. Това улеснява създаването на шаблони, които са смесица между HTML и JS синтаксис. JSX значително ще улесни създаването на компоненти. Важно е да се уточни, че тези шаблони представляват визуализацията на компонента, който трябва да бъде изобразен на уеб страницата.

Text

Description automatically generated

**Фиг. 14 Примерен JSX код**

Тъй като React работи с JavaScript, то JSX кодът трябва да бъде компилиран към JavaScript. Отговорността за този процес поемат компилатори като Babel или Typescript. JSX кодът се компилира до JavaScript код, състоящ се от множество извиквани функции, които са имплементирани в React библиотеката.

Diagram

Description automatically generated

**Фиг. 15 Последователност на работата на клиентското приложение**

1. **Структура на имплементацията**

В този раздел ще бъдат разгледани в детайли структурата на базата данни и на файловете както на сървърното, така и на клиентското приложение. Ще бъдат дадени отговори на въпросите как и защо по съответния начин ще бъдат разделени отделните компоненти на приложението.

* 1. **Структура на базата данни**

Правилното структуриране на релационна база данни гарантира нежеланото повтаряне на информация в нея. В някои случаи добрият дизайн на базата намалява времето за добавяне и търсене на информация в нея. Структурирането на база данни включва проектиране и създаване на схеми, таблици, атрибути за всяка една релация и връзките между отделните таблици. Принципът, по който ще бъде структурирана базата данни за платформата за участниците на капиталовите пазари, е „обект 🡪 таблица(релация)“. Принципът на създаване на връзки между таблиците се определя от това дали обектът съдържа поле (член-данни) от референтен тип, за който също има създадена релация.

За текущите цели на платформата за участниците на капиталовите пазари ще бъде необходима само една схема, в която ще бъдат дефинирани следните релации и техните атрибути:

1. Books:
   1. Guid – uniqueidentifier
   2. Title – nvarchar
   3. Author – nvarchar
   4. Is Deleted – bit
   5. Deleted On – datetime2
2. COT ReportedInstruments:
   1. Guid – uniqueidentifier
   2. Instrument Name – nvarchar
   3. Is Deleted – bit
   4. Deleted On – datetime2
3. COT Reports:
   1. Guid – uniqueidentifier
   2. Date Published– datetime2
   3. Asset Managers Long – bigint
   4. Asset Managers Short – bigint
   5. Leveraged Funds Long – bigint
   6. Leveraged Funds Short – bigint
   7. Asset Managers Long Change – bigint
   8. Asset Managers Long Change – bigint
   9. Leveraged Funds Long Change – bigint
   10. Leveraged Funds Short Change – bigint
   11. COT Reported Instrument Id - uniqueindentifier
   12. Is Deleted – bit
   13. Deleted On – datetime2
4. Exchanges:
   1. Guid – uniqueidentifier
   2. Name – nvarchar
   3. Is Deleted – bit
   4. Deleted On – datetime2
5. Financial Instruments:
   1. Guid – uniqueidentifier
   2. Ticker Symbol – nvarchar
   3. Exchange Id – uniqueidentifier
   4. Market TypeId – uniqueidentifier
   5. Is Deleted – bit
   6. Deleted On – datetime2
6. Futures Contracts
   1. Guid – uniqueidentifier
   2. Name – nvarchar
   3. Tick Size – float
   4. Tick Value – float
   5. Is Deleted – bit
   6. Deleted On – datetime2
7. Futures Positions:
   1. Guid – uniqueidentifier
   2. Direction – int
   3. Trade Position Id– uniqueidentifier
   4. Futures Contract Id – uniqueidentifier
   5. Is Deleted – bit
   6. Deleted On – datetime2
8. Journals:
   1. Guid – uniqueidentifier
   2. User Id – uniqueidentifier
   3. Title – nvarchar
   4. Content – nvarchar
   5. Posted On – datetime2
   6. Is Deleted – bit
   7. Deleted On – datetime2
9. Market Types:
   1. Guid – uniqueidentifier
   2. Name – nvarchar
   3. Is Deleted – bit
   4. Deleted On – datetime2
10. Organizations:
    1. Guid – uniqueidentifier
    2. Name – nvarchar
    3. Is Deleted – bit
    4. Deleted On – datetime2
11. Organizations Messages:
    1. Guid – uniqueidentifier
    2. Sender Id– uniqueidentifier
    3. Organization Id – uniqueidentifier
    4. Message – nvarchar
    5. Sent On - datetime2
    6. Is Deleted – bit
    7. Deleted On – datetime2
12. Positions Record Lists:
    1. Guid – uniqueidentifier
    2. Name – nvarchar
    3. Last Updated - datetime2
    4. Instrument Group – int
    5. User Id - uniqueidentifier
    6. Is Deleted – bit
    7. Deleted On – datetime2
13. Stocks Positions
    1. Guid – uniqueidentifier
    2. Name – nvarchar
    3. Trade Position Id – uniqueidentifier
    4. Buy Commission – int
    5. Sell Commission – int
    6. Is Deleted – bit
    7. Deleted On – datetime2
14. TradePositions:
    1. Guid – uniqueidentifier
    2. Position Added On – datetime2
    3. Quantity Size – float
    4. Positions Record List Id – uniqueidentifier
    5. EntryPrice – float
    6. Exit Price – float
    7. Realized Profit And Loss – float
    8. Is Deleted – bit
    9. Deleted On – datetime2
15. Users:
    1. Guid – uniqueidentifier
    2. Email – nvarchar
    3. Password Hash – nvarchar
    4. Salt – nvarchar
    5. First Name – nvarchar
    6. Last Name – nvarchar
    7. Description – nvarchar
    8. Profile Picture URL – nvarchar
    9. Organization Id – uniqueidentifier
    10. Organization Role – int
    11. Is Deleted – bit
    12. Deleted On – datetime2

Лесно може да бъде забелязан фактът, че всяка релация съдържа атрибутите: „Guid“, „Is Deleted“ и „Deleted On“. Причината е в това, че е добра практика всяка релация да има уникален идентификатор, върху който да бъде дефиниран частния ключ за таблицата. Атрибутите за „изтрит“ запис – „Is Deleted“ и „Deleted On“ – са необходими, за да послужат катo флаг дали записът не е изтрит от потребителска гледна точка. Записите не се изтриват физически от базата данни, защото могат да послужат за създаването на доклади и анализи от софтуерните разработчици за използваемостта на отделни части от платформата.

Базата данни на платформата за участниците на капиталовите пазари е релационна. Това означава, че тя може да поддържа връзки между релациите. Следната ER(Entity Relationship) диаграма представя връзките между релациите в базата данни на платформата:

A picture containing text, screenshot, indoor

Description automatically generated

**Фиг. 16 Цялостна текуща версия на ER диаграмата на базата данни на платформата**

* 1. **Структура на сървърното приложение**

Пиши за сървърното приложение.

**Използвана литература**

**Електронни източници**

1. Microsoft Corporation, [електр. ресурс], 2023, ASP.NET Framework, <https://dotnet.microsoft.com/en-us/apps/aspnet>
2. Dot Net Tutorials, [електр. ресурс], 2023, Task-Based Asynchronous Programming in C#, <https://dotnettutorials.net/lesson/asynchronous-programming-in-csharp/>
3. Slavius, [електр. ресурс], 2022, Java SpringBoot vs. Asp.Net REST API performance, <https://dev.to/slavius/til-java-springboot-vs-aspnet-rest-api-performance-jm2>
4. Tech empower, [електр. ресурс], 2022, Web Framework Benchmarks, <https://www.techempower.com/benchmarks/#section=data-r21&test=composite>
5. Wikipedia, [електр. ресурс], 2023, Argon2, <https://en.wikipedia.org/wiki/Argon2>
6. Wikipedia, [електр. ресурс], 2023, JSON Web Token <https://en.wikipedia.org/wiki/JSON_Web_Token>