**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
4. Платформи, езици, библиотеки и протоколи.....................................10
   1. C#, .NET и ASP.NET CORE.................................................10
   2. SQL........................................................................................10
   3. Entity Framework Core..........................................................10
   4. HTTP и WebSocket протоколи............................................10
   5. SignalR и уеб функционалности в реално време...............17
   6. Допълнителни библиотеки на С#.......................................17
   7. HTML, CSS и JS...................................................................18
   8. 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 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 источниици 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.

**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 платофрмата 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. Седователно 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 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 е платформата TradingView, защото предлага надеждна и рентабилна информация.

Тъй като проектът изисква използването на няколко, вече съществуващи и достъпни, източници на информация, то събирането на икономически и финансови данни, новини, актуални събития, параметри по финансови инструменти и текущите котировки на водещи активи е възможно.

**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 мрежата, 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 концепци, 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 в това, 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.

Diagram

Description automatically generated

**Фиг. 1 Примерна постъпкова диаграма на трислойната архитектура**

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 слоя с бизнес логиката, който обикновено съдържа най-важната логика в приложението, зависи от подробностите за изпълнение на достъпа до данни.

Diagram, timeline

Description automatically generated

**Фиг. 2 Традиционен трислоен модел**

Алтернатива на традиционния трислоен модел е Изчистената архитектура (Clean architecture). Изчистената архитектура 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. Казано синтезирано, слоя с бизнес логиката е толкова независима, че при драстични промени като промяна на база данни или цялостна промяна на технологичната рамка на приложението тя ще продължава да функционира правилно.

Diagram

Description automatically generated

**Фиг. 3** **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).
   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.

Незаивсимостта 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 зараден при първата заявка).

Тяхната алтернатива – многостаничните приложения – получават от уеб сървъра HTML файл, който те трябва да рендерират 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  (Индексация, уникални връзки и метаданни липсват) |
| Използване на по-малко интернет ресурс |  |
| Ефективно кеширане |  |

Недостатъка „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 (Индексация, уникални връзки и метаданни липсват)“ е решим с SEO техники за оптимизиране.

Поради съществуването на ползи и на решения на недостатъците на едностраничните приложения по-доброто решение за клиенската 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. Обектно-ориентираното програмиране 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 характериират и методи, представляващи действия, които той може да извършва. Най-важните принципи в ООП са наследяването, капсулирането, полиморфизмът и абстракцията. Накратко казано, програмата се разделя на обекти, които взаимодействат помежду си.

Функционалното програмиране се различава от процедурното програмиране и обектно-ориентираното програмиране, тъй като използва чисти математически функции. Чрез него резултатите от операциите се базират само на въведените входни данни на математическата функция(разглеждата като чиста функция). Операциите (чистите функции) не разчитат на 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. Повтаряемостта 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 може да се представят различните финансови активи, акаунти на потребители, новини, актуални събития, книги (или друг вид обучително съдържание за секция образование в пларформата), 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 същестуват, 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 принципа 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 принципа 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 принципа 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 принципа 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 принципа 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 атрибута от една таблица, като 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 таплица към уникален индетификатор (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.

A picture containing text, screenshot, indoor

Description automatically generated

**Фиг. 6 ER диаграма на временното състояние на базата данни на платформата**

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 обекно-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).

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

Oбекно-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 инженество 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 необохидими параметри за работата на един обект или функция.

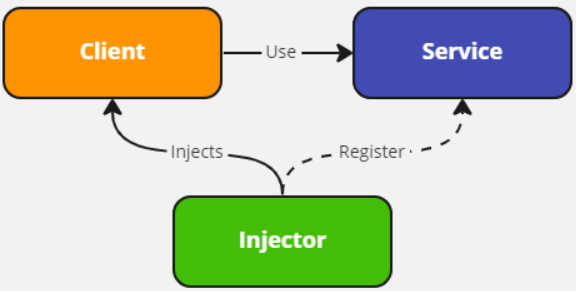
Този шаблон подсигурява това, че конструкторът на обект или самата функция няма да инстанцират зависимостите си. Тяхната отговорност ще бъде само да извършват конкретна услуга. Фундаментално погледнато, Dependency Injection дизайн шаблонът се състои в това да подава параметри на конструктор или функция (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:

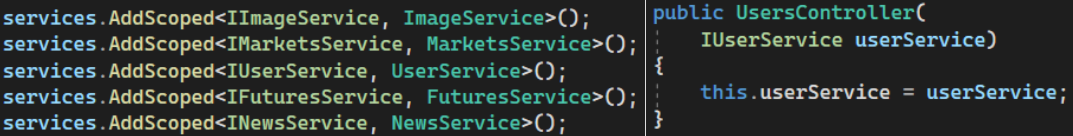
1. Service – зависимост (клас), която ще бъде използвана
2. Client – клас, които ще използва Service (зависимостта)
3. Interface – използван като посредник между Client и Service, като е имплементиран от Service
4. Injector – създава инстанция на Service по съответен Interface и я „инжектира“ (подава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

**3.10 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.
5. **Платформи, езици, библиотеки и протоколи**

**---------**Реферирай раздел „Архитектури, парадигми и дизайн шаблони“ **---------**

Изборът какви инструменти за разработка да използвам, 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(„Фреймуърс“, от англ. “Frameworks”) ще бъдат необходими?“.

* 1. **C#, .NET и ASP.NET CORE**

Платформата, която ще разработвам, ще бъде уеб базирана, достъпвана през уеб клиент – браузър, ще се базира на парадигмата в компютърното програмиране „Обектно-ориентираното програмиране“ и няма проблем да бъде реализирана на език от високо ниво.

Езикът С# е избраният от мен език, чрез който да имплементирам платформа за участниците на капиталовите пазари. Той е обектно-ориентиран, строго типизиран

**---------**Асинхронността в c# **---------**