



ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ  
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

## ДИПЛОМНА РАБОТА

по професия код 481020 „Системен програмист“  
специалност код 4810201 „Системно програмиране“

Тема: Децентрализирана Web3.0 платформа за продажба на продукти с  
offchain транзакции

Дипломант: Йосиф Хамед

Дипломен ръководител: инж. Огнян Чиков

СОФИЯ

2023





ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ  
СИСТЕМИ  
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

**Становище на  
дипломния ръководител**

В процеса на работа, дипломантът Йосиф Хамед изпълни задълженията си отговорно и последователно като завърши целият процес по създаване на дипломната си работа. Напредъка беше бърз, качествен и безкомпромисен. Успя да отстрани всички проблеми и предизвикателства, като взе адекватни и навременни технически решения. Надявам се трудът да бъде уважен и дипломантът да бъде допуснат до защита. Препоръчвам за рецензент г-н Мартин Добрев, технически ръководител в LimeChain. За мен проектът заслужава вниманието Ви и съответно отлична оценка.

Дата: 06.03.2023

Ръководител: .....

инж. Огнян Чиков

## Използвани термини и съкращения

- Блокчейн/Blockchain - Структура от данни, която наподобява свързан списък и се съхранява върху възли в peer to peer мрежа.
- Умни договори/Smart contracts - програми, които са качени върху блокчейн мрежа, която поддържа изпълнението им.
- Web3.0 - общо наименование за екосистема от умни договори и приложенията, които предоставят достъп до тях.
- ECDSA - Elliptic curve digital signature algorithm е алгоритъм за дигитално подписване на информация чрез частен ключ.
- Частен ключ - случайно генерирана поредица от битове с определена дължина, с която можем да подписваме, криптираме и декриптираме съобщения и да генерираме публичен ключ
- Публичен ключ - определен брой битове, генериирани от частен ключ, с които можем да криптираме и декриптираме данни.
- Framework - Цялостна рамка или инструмент, предоставящ абстракция, целяща да улесни разработка на софтуер.
- EVM - Виртуалната машина на Етериум е децентрализиран софтуер за поддържане на състоянието на Етериум мрежата.
- Faucet - Безвъзмезден източник на монети за тестова мрежа
- CLI - Command Line Interface е интерфейс за комуникация между потребител и софтуер през текстов терминал
- MVC/Model View Controller - архитектура, която обединява описанията на таблиците в базата данни, програмната логиката и структурата на потребителския интерфейс в един модул.
- ABI/Application Bytecode Interface - описание на входните и изходните параметри на функциите и атрибутите в контекста на умни договори.
- Mnemonic phrase/мнемомонична фраза - последователност от 12 или 24 думи, която се получава от случайно генериирани битове. Тези битове се използват за генериране на поредица от публични и частни ключове с ECDSA. Част е от BIP39

- BIP - Bitcoin improvement proposal - предложения за добавяне на функционалности в блокчейн мрежи и съществуващи технологии.
- dApp - приложение, използвано за комуникация и работа с умни договори.
- UTC - координирано универсално време.
- Provider - софтуерна услуга, която ни дава достъп до състоянието на блокчейн мрежата. Използва се в контекста на Web3.0.
- Токен/token - виртуален актив и единица за разплащане, поддържана от умен договор по стандарта ERC-20.
- IPFS - InterPlanetary File System е протокол за децентрализирано запазване на информация в peer to peer мрежа.
- Emit - изльчване на събитие при викане на метод от договор.

## УВОД

В началото си интернетът е представлявал информационна връзка с мрежите на различни институции. Първите уеб сайтове се появяват през 90-те години на 20. век заедно със софтуера, разработен от ЦЕРН, за глобална комуникационна мрежа - World Wide Web. [1]

Първите уеб страници са можели само да бъдат зареждани върху браузър без опцията за изпращане на потребителски входни данни до сървъра. Това е представлявал Web1.0 и е бил използван главно за показване на предварително подгответа информация.

С развитието на интернета и увеличаването на броя на потребителите му се развиват и различни Web2.0 страници, които позволяват на посетителя на сайта да въвежда информация, която да се съхранява в бази данни. Това създава промяна в начина, по който хората комуникират.

Сред проблемите на този тип продукти е склонността на корпорациите да събират и продават личните данни на потребителите. Централния авторитет разполага с правото да отказва достъп на определени лица и да забранява анонимната употреба. Въпреки факта, че много от тези продукти са безплатни, те използват потребителя като генератор на стойност, която се приема вместо плащане. Тази стойност стига до корпорацията под формата на лична информация и внимание за реклами.

През 2009 анонимната личност Сатоши Накамото публикува статия за разработката на децентрализирана блокчейн мрежа за разплащане, основаваща се на криптографски функции вместо на доверие и централен авторитет. Целта е да предостави софтуер за разплащане, който позволява анонимно и свободно ползване за всеки с достъп до интернет. През 2015 Виталик Бутер публикува своята мрежа Етериум, която позволява на потребителите да качват програмен код върху блокчен мрежата. Това позволява създаването на децентрализиран код, които да е със свободен достъп и неизменяемо съдържание. Така поставя началото на Web3.0.

Дипломната работа цели да осъществи начин за продажба на продукти посредством децентрализирано приложение, към което е изграден потребителски интерфейс и сървърен софтуер, който да улеснява работата на потребителя. Това се случва със създаването на

“умни договори”, които да отговарят за запазването на основната информация и осъществяване на токенизация на валутата етър.

След публикуването на умните договори не можем да правим промени по тях, затова е от изключително значение е да бъдат написани пълни тестове за всичките им методи.

Сред основните изисквания за сървърния софтуер е запазването на offchain транзакции, които представляват подписано съгласие от потребителя за осъществяване на транзакция от негово име. В конкретния случай това са наддаванията за аукционите.

Една от основните задачи на уеб интерфейса е да предоставя интуитивен начин за комуникация с умните договори и базата данни, имайки предвид особеностите на Web3.0 приложенията.

Последната стъпка е свързана с отчетността пред потребителите. Използваме механизъм за верификация на умните договори, който ни позволява да публикуваме и оригиналния некомпилиран четим код.

# Глава 1

## 1.1 Други подобни системи и продукти

Фундаментално ролята на пазара или т.нар. Marketplace, е да бъде посредник между източника на продукт или услуга и крайния клиент. Една от целите на всички изброени проекти е да разработят посреднически софтуер, който да бъде децентрализиран, прозрачен и в ръцете на потребителите му.

### 1.1.1 BitBay

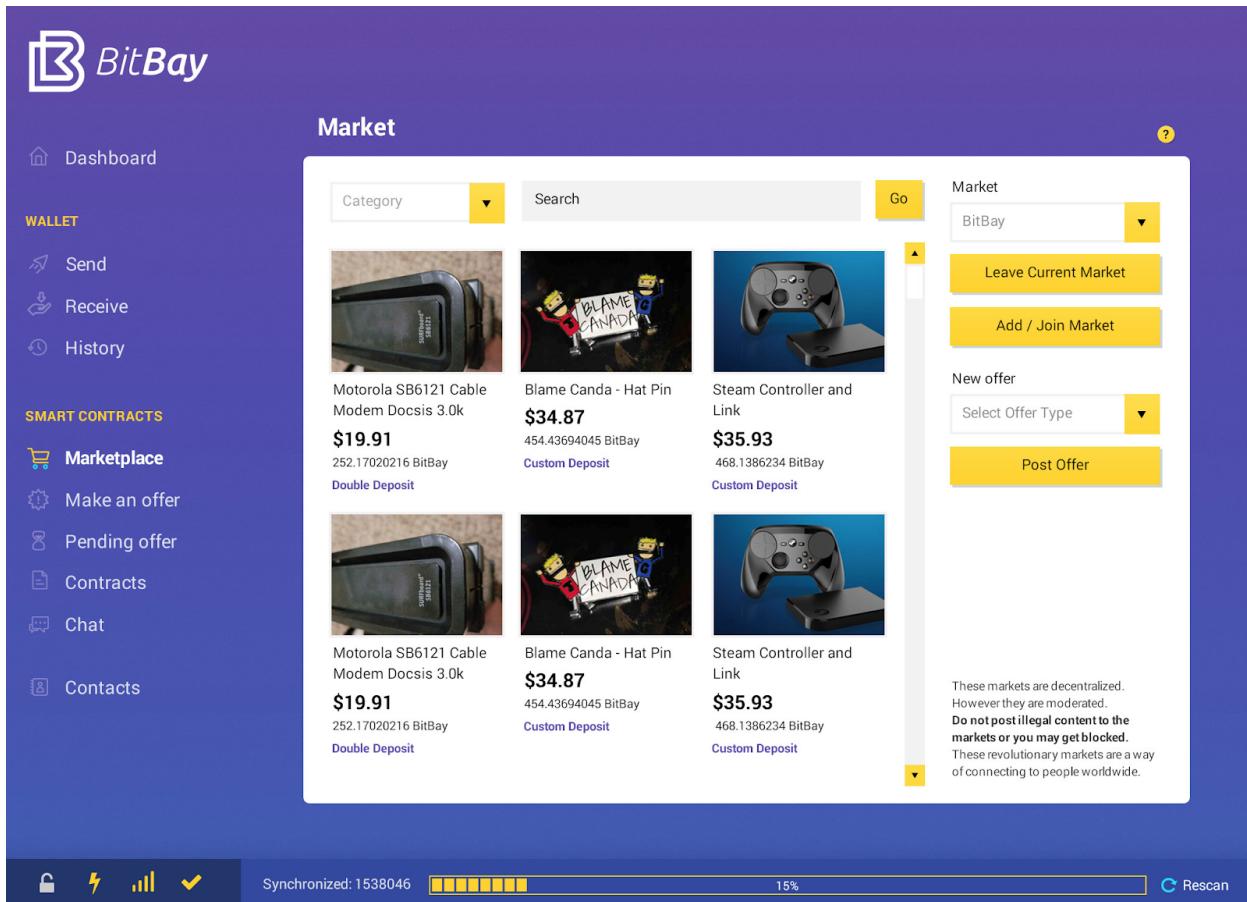


Изображение 1.1 Логото на BitBay

Децентрализирана блокчейн мрежа, целяща да осигури сигурен начин за изпълняване на транзакции между продавачи и купувачи на продукти. BitBay има разработена собствена мрежа със собствени механизми, които позволяват оптимизация в операциите. Разполага със собствена криптовалута - BAY.

Мрежата поддържа създаването на умни договори, както и специален вграден механизъм за изпълнение на транзакции, наречен ескроу с двоен депозит - DDE. Компании като eBay обично ощетяват продавачите, когато получателят на продукта не е доволен. BitBay решава този проблем с механизма си double escrow. При създаване на поръчка и двете страни подават депозит, който е неизползваем до приключването на сделката. Обично този депозит е равен на цената на продукта. Когато продуктът пристигне и двете страни маркират продажбата като валидна и си получават вкарани депозити. Продавачът ще изгуби този депозит ако се опита да измами чрез неизпълнение на поръчката. Същото се отнася и за купувача при непризнаване на успешната доставка. Тези депозити са отделни от самото плащане, но се случват заедно с него. Дефакто това вкарва трета страна в сделката, но тази трета страна е децентрализиран софтуер, който не може да бъде манипулиран в нечия полза.

Проблемът в тази система е, че продавачът трябва да притежава толкова BAY, колкото е говот да продаде в периода от създаването на първата поръчка до пристигането ѝ. Продажбите могат да спрат в резултат на много едновременно направени поръчки и да се възстановят чак когато първата вълна пристигнат. Това притиска продавачите да закупуват по-големи количества BAY. [2]



Изображение 1.2 - Интерфейсът на BitBay

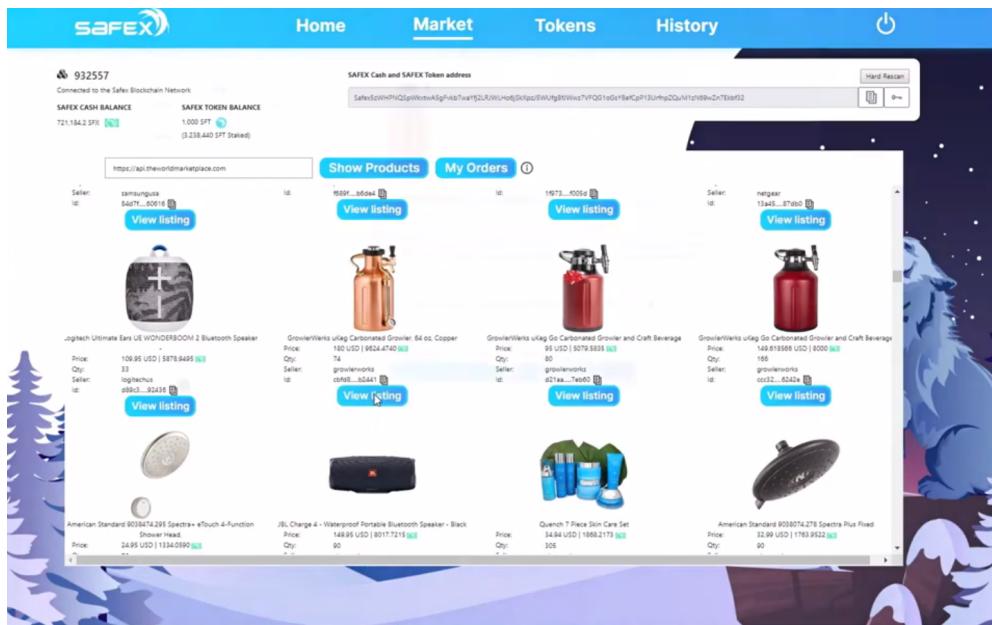
Интерфейсът е семпъл и ясен (изображение 1.2). Предоставя възможност за сдобиване с BAY с приети фиатни валути. Вграденият обмен на фиатни валути към крипто активи е плюс.

Един от основните минуси на платформата е нуждата от специален крипто акаунт, създаден в контекста на тази една мрежа. Този профил е неизползваем в други крипто мрежи. Втория проблем е нуждата от сваляне на BitBay - крипто портфейл за използване на BitBay Marketplace. Това значително забавя процеса на регистриране дори и за редовни потребители на различни крипто услуги.

## 1.1.2 Safex

Децентрализиран протокол с отворен код, който установява мрежа, която позволява електронната търговия да се извършва с помощта на криптовалути.

Safex също разполага със собствена блокчейн мрежа, което създава ограничения от гледна точка на удобство за използване. Вторият проблем с този продукт е нуждата от трета страна за сдобиване със Safex Cash (SFX), който е отделен от основната монета на мрежата Safex Token (SFT). Трети проблем е нуждата от портфел, създаден само за тази мрежа, което може да се случи само през приложението. Също както в BitBay, тук не можем да използваме продукта без десктоп приложение (изображение 1.3).



Изображение 1.3 - Интерфейсът на Safex

## 1.2 Блокчейн. Имплементация в Биткойн

### 1.2.1 История

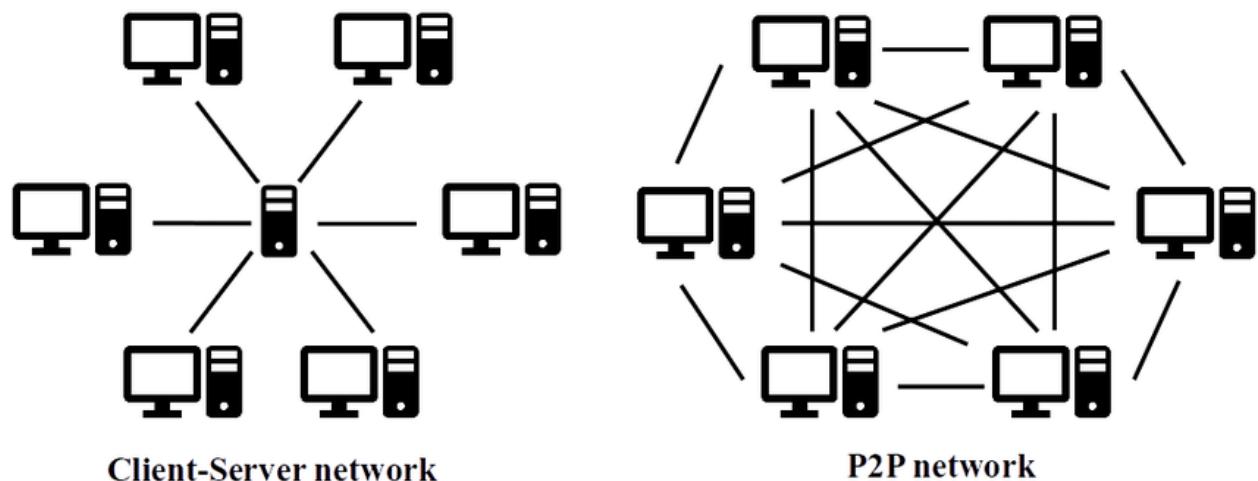
На 31 октомври 2008 г. Сатоши Накамото публикува своя труд “Bitcoin: A Peer-to-Peer Electronic Cash System”, описващ разработката и действието на децентрализирана система, поддържаща обмена на виртуални активи, наречени биткойни. Вместо системата да се основава на доверие към централизирана организация, както наблюдаваме в Web2.0, Биткойн използва криптографски функции, P2P архитектурата и

специален консенсус алгоритъм за да се осигури интегритет на информацията. Идентичността на човек е заменена с неговия публичен адрес, който представлява уникален идентификатор и виртуална локация, до която да бъдат изпратени активите при транзакция. Това прави всеки един участник анонимен и отговорен за финансите си. Самия Сатоши Накамото и до днес отсава анонимен. [3]

През 2013 биткойн ентузиаста Виталик Бутерин публикува статия, описваща бъдеща блокчейн система, която, освен осъществяването на транзакции може да изпълнява код, чрез т. нар. “Smart contracts” или “умни договори”. Върху тази система всеки може да напише свой собствен умен договор и да го внедри в мрежата, като така осигурява отворен достъп. Това поставя началото на Web3.0 екосистемата и отваря света от възможности за създаването на децентрализирани приложения.

### 1.2.2 Какво е P2P мрежа

P2P архитектурата представлява мрежа от 2 или повече устройства, които обменят информация без нуждата от централизиран сървър. (Изображение 1.4)



Изображение 1.4 - Топология на Client-Server и P2P мрежи

P2P мрежите са децентрализирани, което означава, че няма централен авторитет, който да регулира истинността на информацията. Най-често се използва в организации или общности от хора, които имат обща цел и доверие в добрите намерения на останалите

участници. Клиент-сървър моделът цели да предаде отговорността на единна организация, която да обслужва останалите участници в мрежата.

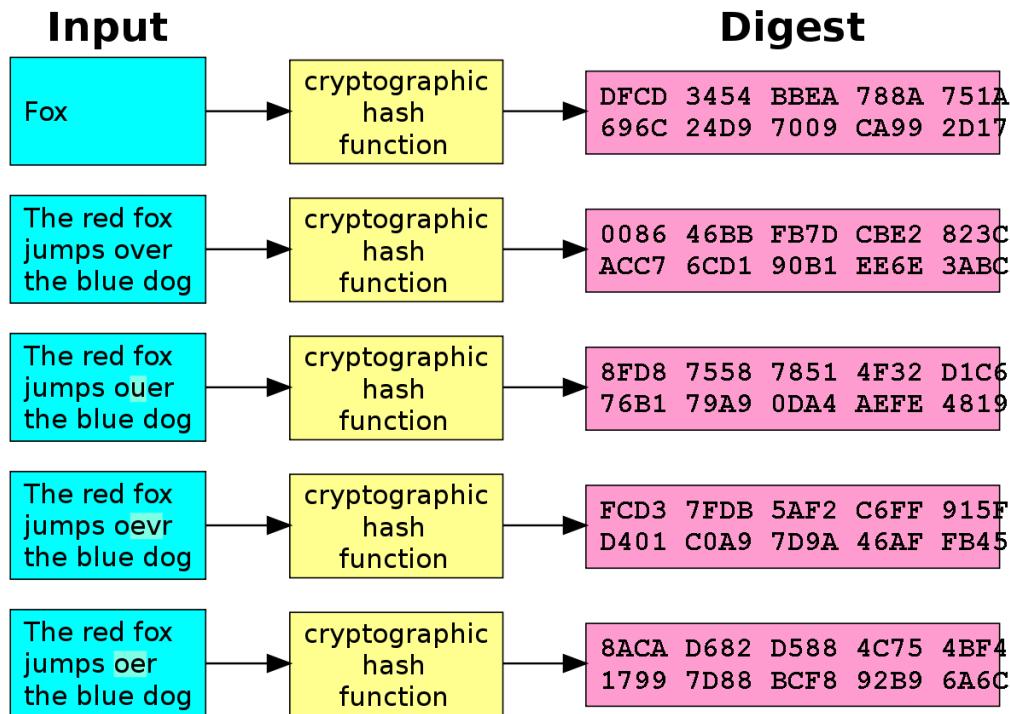
Централизираният модел също се възползва от бързо обработване на заявки, капацитет за запазване на по-големи количества синхронизирани данни, начин за упражняване на контрол от страна на отговорната организация по отношение на достъпа на потребителите, налагане на правила и политики за използване на ресурсите и правене на бързи промени по софтуерната база на услугите, които се предоставят.

P2P мрежите осигуряват равноправие между участниците, което води до липса на авторитет, който да поставя определени рамки за начина на комуникацията. Това поставя фокуса върху изграждането на надеждни, бързи и скалируеми протоколи. С развитието на блокчейн мрежите се появява т. нар. блокчейн трилема, която гласи, че един протокол може да е само две от три неща - скалируем, бърз или сигурен.

### 1.2.3 Какво е CHF функция

Криптографска хеш функция, или Cryptographic Hash Function, е всяка функция, която:

- получава входни данни с произволен размер и има изход с фиксиран размер.
- връща регулярен изход за всеки вход. Всеки път, когато получи аргумент A да връща изход B.
- е оптимизирана за колизии. Не се очаква да върне един и същ изход за два различни входа. Това е невъзможно, имайки предвид, че дължината на изхода е фиксирана. Нека P = броя на битове в изхода. Това означава, че една оптимизирана криптографска хеш функция има  $1/(2^P)$  шанс да покаже един изхода за два различни входа.
- да е непредвидима. Да не може да се определи изхода на произволен вход преди да се прекара през функцията.
- е необратима. От изхода ѝ не можем да намерим подадения аргумент в началото. [4]



Изображение 1.5 - Примерни входни и изходни данни на CHF функция

На изображение 1.5 е показано действието на криптографска функция. Въпреки малките промени във входа, резултатът може да изглежда напълно случаен. Сравнявайки два изхода не можем да кажем, че подадените данни са близки по вид. Също така от получените данни не можем да намерим първоначалните, но можем да ги проверим.

Сред най-използваниите функции са:

- MD5 - 128 бита
- SHA1 - 160 бита
- SHA256 -256 бита

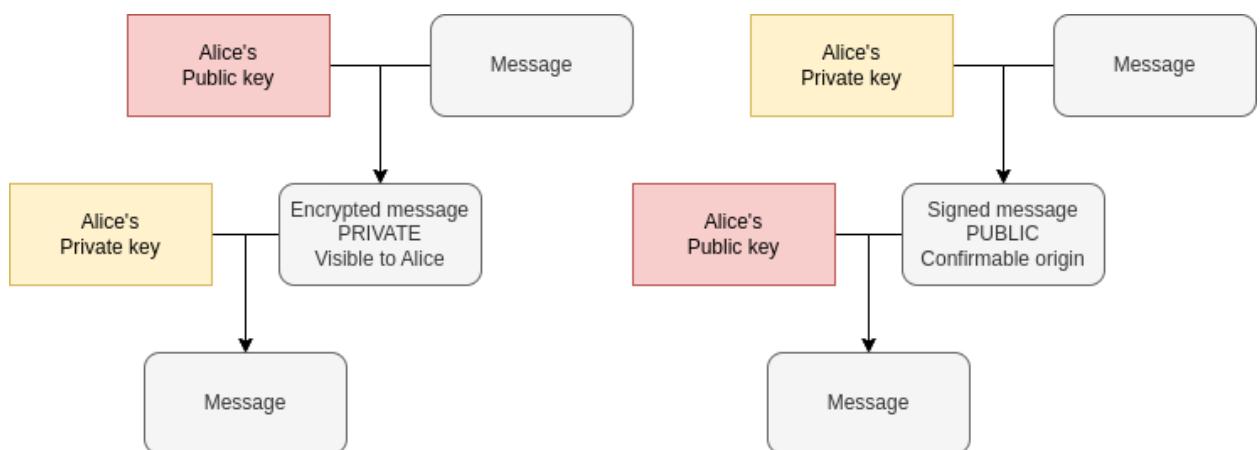
Можем да ги сравним в изображение 1.6.

Input (Message)	Hash Function	Output (Hash Value)
CFI	MD5 (128-bit, 16-byte) 32 characters	3A10 0B15 B943 0B17 11F2 E38F 0593 9A9A
CFI	SHA-1 (160-bit, 20-byte) 40 characters	569D C9F0 7B48 7F58 9241 AD4C 5C28 7DA0 A448 8D08
CFI	SHA-256 (256-bit, 32-byte) 64 characters	F3ED 0867 48FF 3641 3091 0BB6 6293 7080 2958 B5A2 52AF F364 1FC5 07FD E80D 9929

Изображение 1.6 - Сравнение в параметрите на 3 CHF функции

#### 1.2.4 Асиметрично криптиране

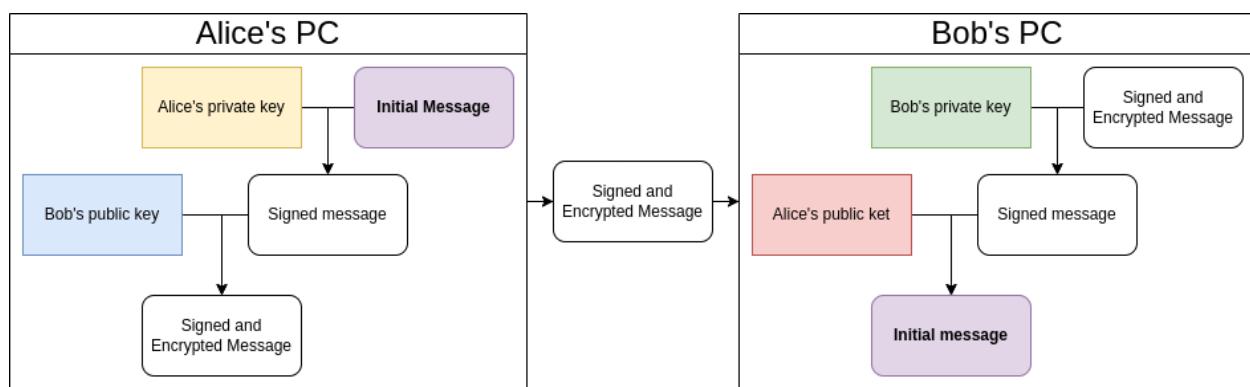
Използва се за доказване на източника на дадена информация, както и за сигурно изпращане на лични данни, без да бъдат достъпни за посредници в комуникацията. За този тип криптиране са ни нужни публичен и частен ключ. Частния е случаен генериран и се използва за генериране на частен. Когато използваме публичния ключ за да заключим информация, тя може да бъде отключена само от частния. Когато искаме да изпратим тайно съобщение до някого ще направим точно това с техния публичен ключ. При заключване с частния ключ можем да предоставим съобщения, чийто източник да е потвърдим, защото съобщението може да се отключи с публичния ключ.



Диаграма 1.1 - Криптиране и декриптиране с частен и публичен ключ

На диаграма 1.1 е показано криптиране и декриптиране. Ако Алиса иска да изпрати съобщение до Боб, то тя ще го криптира с неговия публичен ключ (за да гарантира, че е скрито за всички останали) и със собствения си частен ключ (за да може да се потвърди, че тя е източника).

На диаграма 1.2 е показан пример.



Диаграма 1.2 - Пример за комуникация посредством асиметрично криптиране

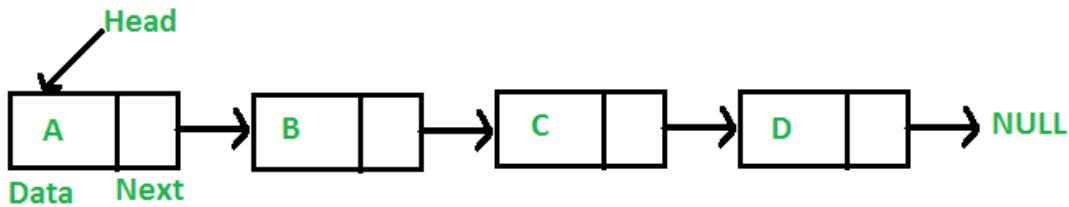
### 1.2.5 ECDSA

Elliptic curve digital signature algorithm е алгоритъм за “подписване” и потвърждаване на информация посредством частен и публичен ключ. При подаване на частен ключ и произволно съобщение можем да създадем подpis, който отговаря на нашия акаунт, но не запазва съдържанието на информацията. За да потвърдим сигнатурата трябва да имаме оригиналното съобщение и подписаното съобщение. При прекарване през потвърждаваща функция ще получим публичния ключ на подписващия, което потвърждава източника. По този начин източникът на транзакция в блокчейн мрежа потвърждава пред валидатора, че информацията е достоверна и има съгласие за извършване на операцията. Разликата между подписването и криптирането с частен ключ е в размера на изхода на функцията. Подписването не запазва информацията и има нужда от оригиналната за да потвърди източника.

### 1.2.6 Linked list

Linked list, или свързан списък, е съвкупност от 2 или повече инстанции на структура “възел”. Възел е структура, съдържаща ценна информация и показател към

друга такава инстанция. Полученият резултат е последователност от информационни възли, които съдържат собствено съдържание и референция към друг възел.



Диаграма 1.3 - свързан списък

На графика 1.3 е показана последователност от 4 инстанции на структура възел. Първата има собствено съдържание А и показател към възел със съдържание В. След последната инстанция няма повече данни, затова показателят има стойност NULL.

Тази структура от данни се използва, когато не искаме цялата информация да е записана последователно в паметта. Също може да е полезна ако съдържанието на възлите не е от един и същ тип и дължина. Сред минусите е неконстантното време за намиране на елементи, което става линейно по-бавно с нарастване на списъка. Също нуждата да запазваме адреса на всеки възел, което е допълнително място в паметта.

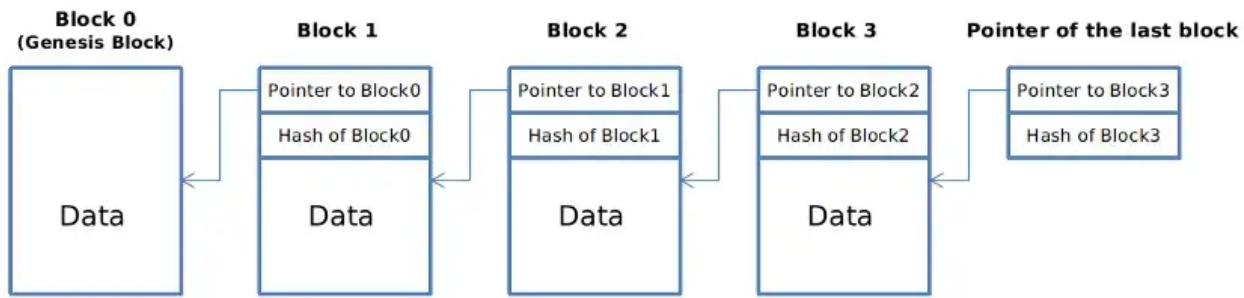
## 1.2.7 Block structure

Bitcoin поставя началото на децентрализираните финансови системи, основани на криптографски функции и консенсус алгоритми. За да сме добре запознати с блокчайн технологията, трябва да имаме добра представа за структурите от данни, които изграждат системата. Затова ще разгледаме най-важните атрибути на един биткойн блок.

### 1.2.7.1 Prev block hash

Системата разпределя транзакциите си в блокове, които служат за възли в свързан списък. Всеки един блок трябва да сочи към предходния за да се знае последователността на транзакциите. Този показател представлява изхода от криптографска хеш функция, която е получила предишния блок като аргумент. Това гарантира, че след изчисляването на

този хеш, всеки следващ блок е зависим от предишния. Хеша на текущия блок ще бъде показателя на следващия, но в същото време в него влиза хеша на предишния. Това означава, че не можем да променим съдържанието на предишен блок без да променим указателя на текущия, съответно и на следващия и тн (показано в диаграма 1.4). [5]



Диаграма 1.4 - Свързване на блокове в Биткойн

#### 1.2.7.2 Nonce

Това е атрибут в структурата на блока, в който се поставя число. По време на “изкопаването” на един блок в блокчейн система, всъщност се изчислява хеша на блока с nonce=0, след това хеша с nonce=1 и тн. докато nonce-а не получи стойност, за която хеша на целия блок отговаря на определени изисквания. Това е полето, чиято стойност последна се намира в блока. В Биткойн се търси nonce, който прави хеша да е по-малък от определена стойност. Тази стойност се избира на база времето, което е било необходимо за последното такова изчисляване.

#### 1.2.7.3 Merkle tree transaction hash

Това е колективния хеш на всички транзакции в блока. Разполагайки с всички транзакции в блока и този хеш, можем да определим дали са валидни наистина. За имплементацията е използвана структурата Merkle tree, която ни позволява да не запазваме всички транзакции за да потвърдим валидността на тези, които ни трябват. В края на главата има бяснение на структурата и имплементацията ѝ.

#### 1.2.7.4 Transactions

Основната цел на всеки блок е да запомня изпълнените транзакции. Това поле отговаря за запазването им.

#### 1.2.7.5 Block hash

Това е хеша на текущия блок след изчисляването на nonce-а.

### 1.2.8 PoW vs PoS

Proof of Work (PoW) и Proof of Stake (PoS) са най-често срещаните консенсус механизми, използвани в блокчейн системите. Трябват ни консенсус алгоритми, за да потвърждаваме валидността на информацията, която се подава от участниците в мрежата. И двата алгоритъма залагат на идеята за използване на определен тип ресурси за придобиване на право за участие и валидиране на транзакции. Всеки валидатор изпълнява определени операции в зависимост от консенсус алгоритъма. Така се валидират чужди транзакции, генерират се нови активи и се събират такси от транзакциите.

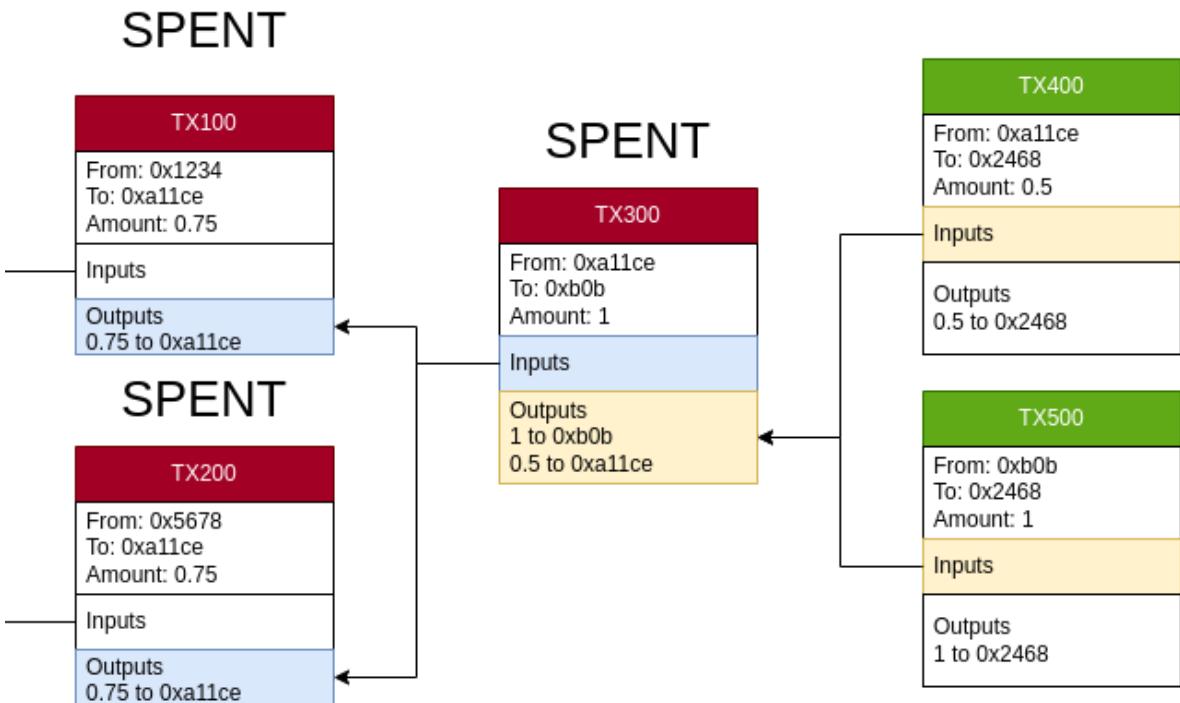
Валидирането на транзакции чрез Proof of Work, или Доказателство за Работа, се нарича “копаене” и се използва от Bitcoin. В този процес средството, което се влага от участниците е изчислителната им мощност. Всеки един участник в мрежата слуша за broadcast-нати транзакции и ги записва в собствен спийк. След събирането на определено количество транзакции се изчислява хеша на целия блок отново и отново, като с всяко следващо изчисление се променя полето nonce докато хеша на блока не започва с N на брой нули. Алгоритъма на биткойн определя N чрез колективната изчислителна мощност на мрежата, така че правилния nonce да се намери за приблизително 10 минути. Това затруднява изключително много процеса за “изкопаването” на валиден блок, което гарантира трудността на задачата и случайността за избиране на победител на надпреварата за намиране на този nonce. Вероятността определен участник да намери решение за текущия блок е директно пропорционално на процента от изчислителната мощност от мрежата, която притежава. Ако Алиса има 10% от изчислителната мощност, то тя ще валидира 10% от блоковете, което е приблизително 1 блок на всеки 100 минути. След изкопаването на блока той бива препратен до всички участници в мрежата, за да може да бъде много лесно проверен и всяка една транзакция да бъде валидирана.

Основната цел е да се намали вероятността от злонамерен участник да изкопава блокове по-бързо от останалата част от мрежата. Това може да се случи само ако притежава повече от 50% от изчислителната мощност.

Валидирането чрез Proof of Stake, или Доказателство за Залог, може да бъде имплементирано по много различни начини, но винаги използваните средства са крипто активи. Докато в PoW валидаторът залага процесорното си време и електроенергията, която е използвал, в PoS се залагат крипто активи. По този начин, ако мрежата засече, че се случва измама, може да “изгори” или преразпредели депозита. Избирането на валидатори също може да бъде случайно и правопропорционално на депозита. За Етериум мрежата минималния залог е 32 етъра (приблизително 50 хил. долара за цена на ETH 1.600\$). От 15 септември 2022 година Ethereum е сред мрежите, които имплементират този механизъм, което намали разхода за електроенергия с 99.95%. [6]

### 1.2.9 Структура на транзакцията

Транзакцията е трансфер на единици, който е публичен за всеки участник в мрежата, затова и системата е напълно прозрачна. Представлява структура с няколко основни атрибути, освен задължителните: От, До и Количество активи. Имплементацията на Биткойн използва и тн. нар. Input и Output полета. Всеки инпут е референция към предишна транзакция, чрез която изпращача на текущата е получил средствата си. Ако Алиса иска да изпрати на Боб 1 биткойн, трябва да посочи транзакция или транзакции, чиито сбор е по-голям или равен на 1 биткойн, чрез които тя е получила активите си. Ако сбора на инпутите е равен на изпратеното количество, то транзакцията има един аутпут - количеството, което се изпраща до Боб. Така той може да използва тази транзакция за инпут при бъдещо търгуване. Ако сборът е по-голям, аутпутите са 2, един за Боб, който е равен на изпратеното количество, и един за Алиса, който е равен на останалите активи. Ако Алиса разполага с два аутпути по 0.75 и иска да изпрати 1 биткойн, то тя може да ги използва като инпути за текущата транзакция.  $0.75 + 0.75 > 1$ , затова в транзакцията ще има два аутпути. Един за Боб със стойност 1 и един за Алиса със стойност 0.5. Благодарения на Merkle Tree структурата, транзакциите, които са използвани като инпути няма нужда да се запазват в паметта на участниците в мрежата. [7] Посочен е пример в диаграма 1.5.



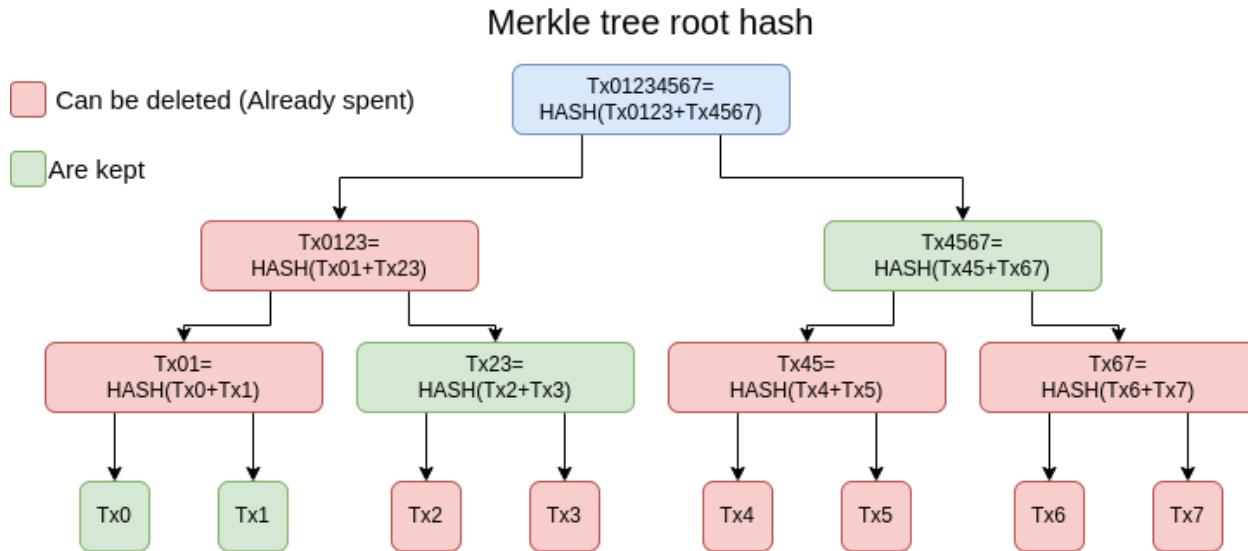
Диаграма 1.5 - примери за транзакции с входове и изходи

Валидаторът включва транзакция в блок само ако е била подписана с частния ключ, който съответства на адреса в полето From. Самото валидиране включва проверка на инпутите.

### 1.2.10 Merkle tree

Всеки блок трябва да съдържа в себе си някакъв метод за проверяване на историята на транзакциите. В биткойн това се постига чрез полето Merkle Tree Transactions Hash. В него се съдържа хеша на всички транзакции в блока. Така ако ги имаме, можем много лесно да проверим дали целия списък е валиден. Проблемът идва от факта, че след “използването” на транзакция като инпут за друга, тя вече става ненужна и заема излишно място в паметта на валидаторите. Ако изчисляваме хеша на списъка с всички транзакции, щяхме да имаме нужда от всяка една за да сме сигурни, затова използваме структурата Merkle tree, която ни позволява да изтриваме транзакции, докато запазваме потвърждаемостта на останалите. Това представлява бинарно дърво, в което всеки възел е

результат от хеш функцията на долните два възела. На дъното са транзакциите, а коренът е крайния резултат, който записваме в хедъра на блока. [8]



Диаграма 1.6 - Пример за структурата Merkle Tree

В диаграма 1.6 транзакциите са листата на бинарното дърво. Родителите на тези транзакции са техните хешове. Tx01 е резултата от колективния хеш на Tx0 и Tx1. Tx0123 - на Tx01 и Tx23. Червените възли от дървото могат да бъдат изтривани, защото са ненужни за оперирането на системата или могат да се изчислят от по-нисшите елементи в дървото. Във фигурата транзакции от 2 до 7 са използвани.

## 1.3 Развойни средства и среди

### 1.3.1 Блокчейн мрежи

За реализацията на дипломната работа се изискава блокчейн система, която да поддържа изпълнението на инструкции под формата на код върху възли в мрежата. Сред мрежите, които имплементират тази функционалност са Ethereum, Polkadot, Solana и други.

Етериум е първата разработена и оперираща блокчейн мрежа, която поддържа качването и изпълнението на умни договори.



Изображение 1.7 - Логото на Етериум

Полкадот е мултичийн решение за блокчейн системи, която цели да увеличи скоростта и броя на осъществимите транзакции чрез поддържане на взаимносвързани мрежи. Т. Нар. Блокчейн трилема описва, че една мрежа не може да е едновременно бърза, сигурна и скалируема, затова Полкадот свързва отделни бързи и сигурни парамрежи, които могат да бъдат създавани и взаимно свързани чрез централни мрежи.



Изображение 1.8 - Логото на Полкадот

Солана е блокчейн мрежа, която е създадена с цел да бъде бърза и използвана за ежедневни плащания. Това е постигнато чрез 10 мегабайтови блокове, които биват приети на всеки 400 милисекунди и съдържат транзакции, които не надвишават 1232 байта. Това ни дава по 20 хиляди транзакции на секунда, проемайки, че всяка транзакция заема максималния предефиниран размер. [9]

До момента мрежата е показала, че може да поддържа до 65 хиляди транзакции на секунда, докато Visa е показала до макс от 65 хиляди на секунда. [10]



Изображение 1.9 - Логото на Солана

### 1.3.2 Инструменти за разработка на умни договори

За разработката на умните договори ни трябва framework, който да работи с програмен език с голяма общност и материали за разработчици. Това изключва Remix - online IDE, който е първия избор за начинаещи разработчици в сферата. Няма нужда от допълнителна конфигурация, но не предоставя всички нужни функционалности.

1. Hardhat - позволява ни да компилираме, внедряваме, тестваме умни договори, както и да намираме покритието на тестовете. Написан е на JavaScript, а езикът за договорите е Solidity по подразбиране. Разполага с голяма общност и много допълнителни библиотеки за добавяне на функционалности. Един от основните плюсове е добавения израз console.log(), който можем да ползваме в кода на smart contract-ите, което много улеснява процеса за разработка. За тестове използва Mocha и Chai.



Изображение 1.10 - Логото на Hardhat

2. С Truffle можем да компилираме, внедряваме и тестваме готовите договори. По подразбиране използва Ganache локална мрежа, която да симулира Етериум по време на разработката. За тестове използва библиотеката Mocha. Тестове и скриптове се пишат на JavaScript.



Изображение 1.11 - Логото на Truffle

3. Ape - инструмент, който ни позволява да пишем скриптове и тестове за договорите, както и да ги внедряваме в EVM мрежи. За целта се използва Python, а езикът за smart contract-ите по подразбиране е Vyper (много подобен на Python), но може чрез плъгини да се смени на Solidity, който е много по-често срещан.



Изображение 1.12 - Логото на екипа зад Аре

4. Web3j - библиотека, използваща Java, с високо ниво на абстракция. Можем да изпълняваме тестове, както и да пишем смарт контактите си на Java, която се превежда вътрешно на Solidity.



Изображение 1.13 - Логото на Web3j

### 1.3.3 Език за разработка на умни договори

През 2013 Виталик Бутерин публикува статия, относно децентрализирана мрежа, подобна на Биткойн, която да поддържа изпълнението програмни функции, повикани от потребителите. Това поставя началото на идеята за “умните” договори, Smart Contracts, които се изпълняват върху EVM, Ethereum Virtual Machine. Езици като Python, C++ или JavaScript не могат да се използват за написването на такава програма, защото не са създадени за работа върху EVM. Затова имаме езици като Solidity, Vyper, Yul и Yul+, Huff, LLL, Fe и други. Ще разгледаме четирите най-използвани за избор в дипломната работа.

1. Solidity - представлява статичен обектно-ориентиран език от високо ниво, който наподобява C++ и JavaScript чрез синтаксиса си. Разполага с много функционалности, сред които са наследяване на контракти, използване на библиотеки, имплементиране на интерфейси, модификатори за функции, рекурсия и др. Стандартно се компилира със SOLC. Има изградени инструменти за работа и голяма общност и едно от най-големите му предимства е, че има опцията да се пише inline асемблер



Изображение 1.14 - Логото на Solidity

2. Vyper - език за писане на умни договори, наподобяващ Python, като всеки Vyper код е валиден и в Python, но обратното не е задължително вярно. Разполага с много по-малко функционалности от Solidity, но е много по-лесен за четене и одитиране. Байткодът му след компилация е силно оптимизиран. Не поддържа наследяване, модификатори за функции, писане на асемблер и др., но затова се използва за правене на прототипи и разписване на идеи.



Изображение 1.15 - Логото на Vyper

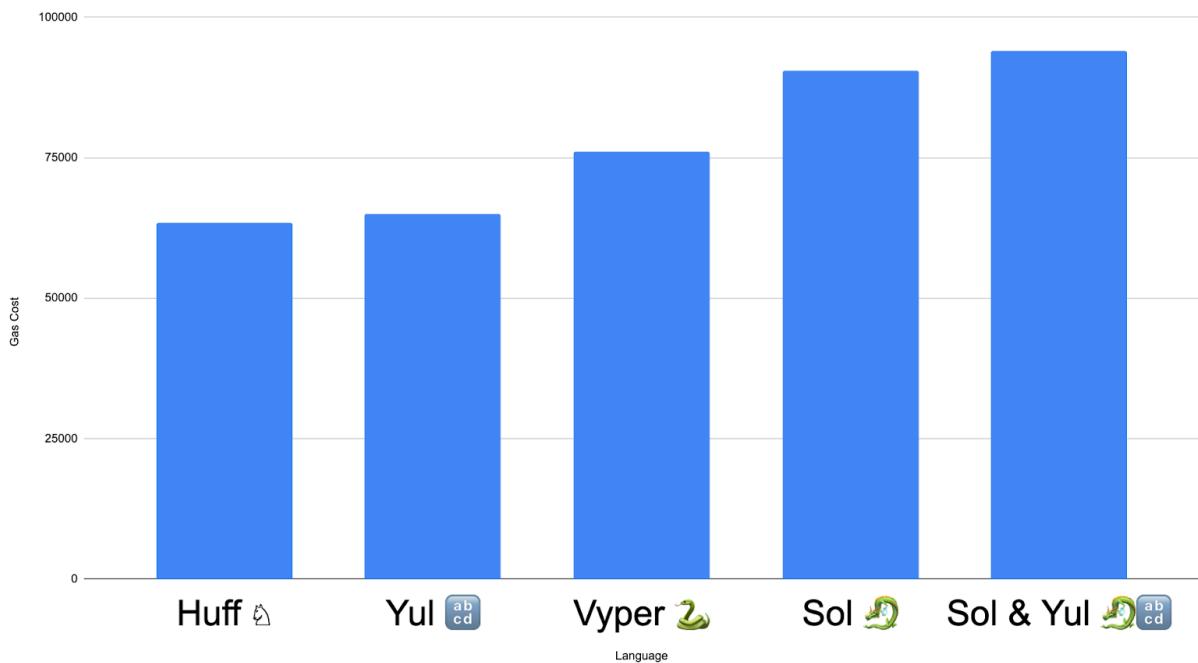
3. Yul, Yul+ (JULIA) - език от ниско ниво, използван за оптимизация. Използва се като инструмент за Solidity разработчици с цел да се предостави начин да се оптимизира кода, без да трябва да се използва асемблер. В една Solidity програма може да се използва Yul за оптимизация. Този език може да се компилира със SOLC. Разполага с for, if и switch, за разлика от асемблер, което много улеснява ръчните одити.

- Huff - език за програмиране от ниско ниво, предназначен за разработване на силно оптимизирани умни договори, които работят на EVM. Huff не крие вътрешната работа на EVM и вместо това излага програмния стек на разработчика за ръчна манипулация.



Изображение 1.16 - Логото на Huff

Gas Cost vs. Language - SimpleStorage Contract Creation

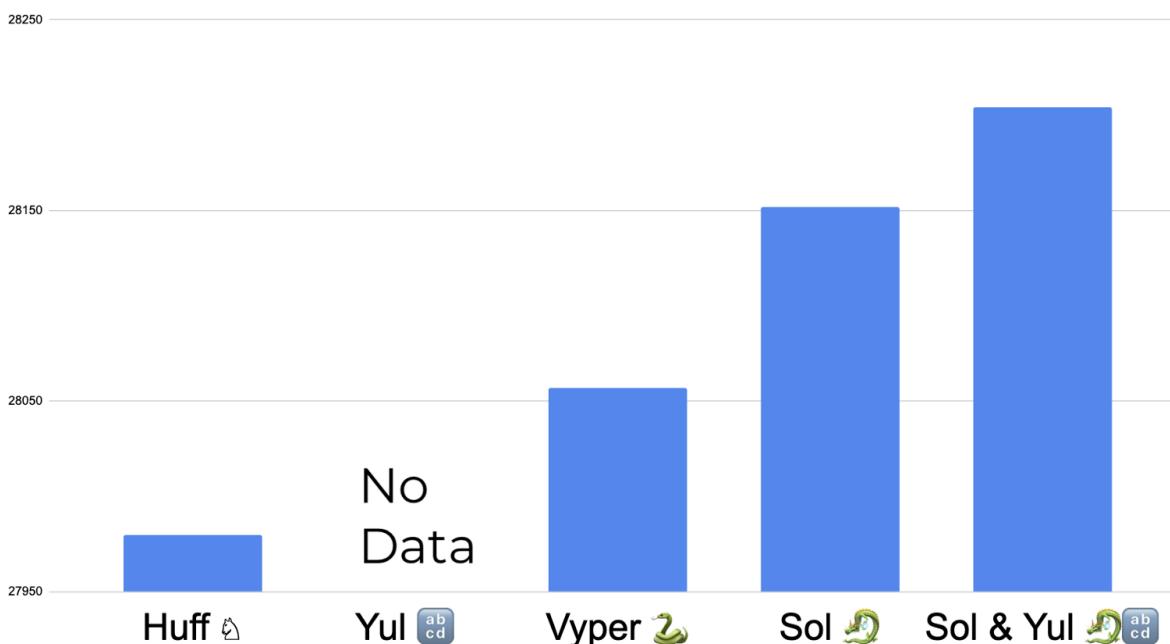


Диаграма 1.7 - Сравнение между таксите за качване на умен договор с различни езици

На диаграма 1.7 е показано срвнение между цените за транзакции в Етериум за различните езици при създаване на договор. Цените са относително близки, защото

операциите са едни и същи. Няма големи разлики в размера на договора след компилиране.

### Gas Cost vs Language - SimpleStorage Read & Write



Диаграма 1.8 - Сравнение при таксите при извикване на методи от договори с различни езици

На диаграма 1.8 са показани цените на транзакциите, които четат и пишат в умния договор. Чрез Huff се постигат значително по-ефективни транзакции заради оптимизираността на кода. [11]

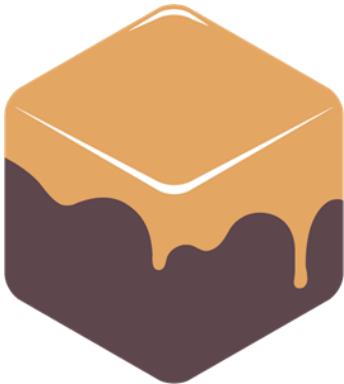
#### 1.3.4 Тестова мрежа за разработка

По време на разработката трябва смарт контрактите да се внедрят в мрежа, осъществяваща повикванията им, за да можем да тестваме функционалности, както и да проверим правилното свързване на front-end и back-end частта. Имаме три основни избора за изграждането на нашите смарт контракти.

1. Официалната Етериум мрежа - публична мрежа, която е в настояща експлоатация от хиляди потребители по света. Използването ѝ ни гарантира максимално реалистични условия за изпълнение на всички функции на умните договори, защото внедряването в нея е от финалните цели на проекта. Сред предимствата ѝ са

интегритета и реалистичните условия. Среди минусите са бавното изпълняване на транзакции (около 12 секунди за изкопаване на блок) и факта, че трябва да разполагаме с активи с реална стойност, което означава, че трябва да се закупи някакво количество етъри (локалната монета) за да се осъществяват транзакции.

2. Goerli (Görlī) - публична тестова мрежа, която е копие на Етериум и следва същите правила за публикуване на умни договори. Основното ѝ предимство пред официалната мрежа е нулевата стойност на локалната монета, което означава, че можем да тестваме безплатно. Мрежата може да е нестабилна и да има нерегуляри цени за транзакции, което означава, че за едно повикване на функция в dApp в един момент може да се иска определена такса, но след време да се иска много по-висока. Самото добиване на етъри в тази мрежа може да се случи по два начина. Единия е да се поиска малко количество от т. нар. faucet, където безвъзмездно се раздават тестови етъри за мрежата. Другия начин е чрез копаене за мрежата, което може да се случи и чрез сайтове, които управляват изчислителната мощност на потребителите. Непредвидимостта в мрежата може да затрудни процеса, а високите такси да забавят набавянето на нужните тестови монети.
3. Ganache - инструмент за създаване на локална тестова мрежа с помощта на CLI. С пускането на мрежата разполагаме с тестови акаунти, предварително заредени с етъри. Част от плюсовете са скоростта на мрежата, предварително заредените профили, активите в тях и опциите при пускане на мрежата. Инструментът предоставя възможността изпълнените действия да се запазват в база данни, която да се използва при бъдещо пускане на локалната мрежа. Един от минусите е факта, че ангажира нашата машина с пускането на допълнителни процеси.



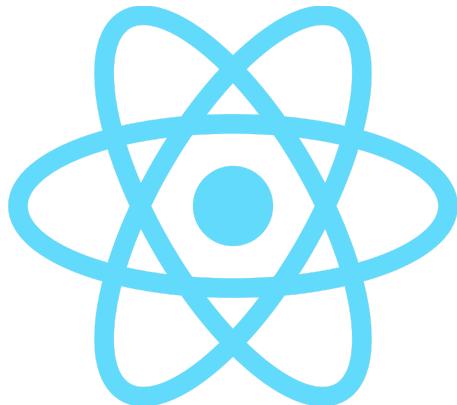
Изображение 1.17 - Логото на Ganche

### 1.3.5 Инструменти за разработка на уеб потребителски интерфейси

След създаването, тестването и интегрирането на нашите смарт контракти потребителя трябва да има лесен достъп до тях. Това се постига чрез разработване на уеб графичен интерфейс. Първата част от дипломната работа използва рамката Hardhat с JavaScript, затова за създаване на преизползваем код ще използваме и инструменти за разработка на уеб приложения с JavaScript.

1. React - инструмент, разработен от Facebook, използван за изграждане на интерактивни потребителски интерфейси и уеб приложения бързо и ефективно със значително по-малко код, отколкото би се използвал с обикновен JavaScript. Разработката се осъществява чрез изграждане на преизползваеми и независими компоненти, които се включват като градивни части иза цялостния интерфейс.

Като най-използвана библиотека за подобен тип приложения има и много голяма общност, съответно и учебни материали. Всеки написан компонент може да бъде преизползван многократно в приложението.



Изображение 1.18 - Логото на React.js

2. Vue - лека JavaScript рамка за изграждане уеб потребителски интерфейси. Vue разширява стандартния HTML и CSS, за да създаде набор от мощни инструменти за изграждане на интерактивни уеб приложения.

Намира се в топ 5 най-използвани инструменти за създаване на front-end приложения, което означава, че разполага с голяма общност и много учебни материали. Няма голяма корпорация, която да стои зад проекта.



Изображение 1.19 - Логото на Vue.js

3. Angular - рамка за разработка на уеб приложения, изградена на TypeScript. Сред плюсовете е изграждането на компоненти, колекция от добре интегрирани библиотеки, които покриват голямо разнообразие от функции, пакет от инструменти за разработчици. Поддържа се от Google.



Изображение 1.20 - Логото на Angular.js

### 1.3.6 Инструменти за разработка на сървърен софтуер

С подаването на транзакция към Етериум мрежата се плащат и такси, които са изцяло зависими от заетостта на възлите ѝ, но в някои случаи бизнес логиката ни позволява да отложим изпращането на определени транзакции. Това ни позволява да си спестим сумарните такси, които потребителите трябва да плащат. В дипломната работа се имплементира договор, който да осъществява наддавания за продукти под формата на аукцион, което означава, че всяко наддаване е транзакция с такса, но само едно от тези наддавания ще даде истински резултат за съответния потребител. Това е една от причините да имаме нужда от централизиран сървър, който да изпълнява определени функции. Ще разгледаме най-често използваните библиотеки и рамки за създаване на сървърен софтуер.

1. Node.js - платформа с отворен код за създаване на среда, върху която да се осъществява изпълнението на JavaScript програми. Сред предимствата ѝ са скростта, поносимостта на голям брой заявки, асинхронните операции, които идват с JavaScript и гъвкавостта, която се предлага от липсата на предефинирана файлова структура.



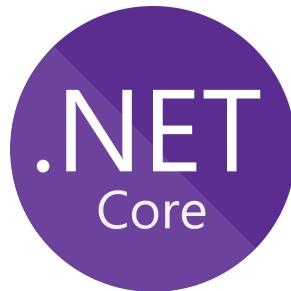
Изображение 1.21 - Логото на Node.js

2. Django - рамка за създаване на сървърен софтуер с Python, която по подразбиране следва MTV модела (MVC) с установена файлове структура. Сред плюсовете са предварително изградените инструменти, високото ниво на абстракция, сигурност и ясни, насочващи разработчика, структури.



Изображение 1.22 - Логото на Django

3. .NET – инструмент с отворен код, разработен от Microsoft, използван за писане на бизнес логиката на сървъра. Работи с C# и се избира за уеб приложения, използващи HTML, CSS и JavaScript. Може да е по-ефективен при по-натоварващи процеси.



Изображение 1.23 - Лого на .NET Core

### 1.3.7 Библиотеки за комуникация с умни договори

Всички инструменти, които ползваме работят с JavaScript и нямат директен API към функционалностите на умните договори или блокчейн мрежата, която ползваме. Затова ни трябва библиотека, която да осъществява връзката между децентрализираните приложения и скриптовете, които пишем за тестове, потребителски интерфейс и комуникацията с база данни в сървърната част. За това ни трябва JavaScript библиотека, която да осъществява абстракция за нашия код. Избраните решения предлагат класове с методи, които да представляват нашите смарт контракти, като ни разрешават да извършваме повиквания към техните функции.

1. Web3.js - колекция от библиотеки, които позволяват да взаимодействаме с локален или отдалечен етериум възел, използвайки HTTP, IPC или WebSocket.



Изображение 1.24 - Логото на Web3.js

```

const web3 = new Web3(/* your provider */);

async function main() {
    let accounts = await web3.eth.getAccounts();
    let contract = new web3.eth.Contract(abi)
        .deploy({ data: bytecode.object })
        .send({ from: accounts[0], gas: "1000000" });
    console.log(contract.address);
}

main();

```

Изображение 1.25 - пример за код за внедряване на договори чрез Web3.js

2. Ethers.js - пълна и компактна библиотека за взаимодействие с Ethereum Blockchain и нейната екосистема. Първоначално е проектиран за използване с ethers.io, но по-късно се разширява в библиотека с по-общо предназначение.



Изображение 1.26 - Логото на ethers.js

```

async function main() {
    const signer = new ethers.providers.Web3Provider(/* your provider */).getSigner();
    const contractFactory = new ethers.ContractFactory(abi, bytecode.object, signer);
    const contract = await contractFactory.deploy()
    console.log(contract.address);
}

main();

```

Изображение 1.27 - Примерен код за внедряване на договори чрез ethers.js

Библиотеката EthersJS е създадена да бъде проста, лека и ефективна. Това може да доведе до по-компактен код, когато използваме Ethers вместо Web3. Ethers постигат това ниво на простота, като ограничава разработчиците и дава само това, което е необходимо за взаимодействие с мрежата.

Библиотеката Ethers е забележително по-малка от библиотеката на Web3. Поради това уеб интерфейсните приложения са по-малки, когато използват Ethers, а не Web3. Това позволява да се зареждат по-бързо.

Едно от предимствата на Web3.js е факта, че съществува от повече време, съответно има и по-голяма общност. Затова е стандартната практика за по-големите компании да се използва Web3.js.

### 1.3.8 Развойни среди

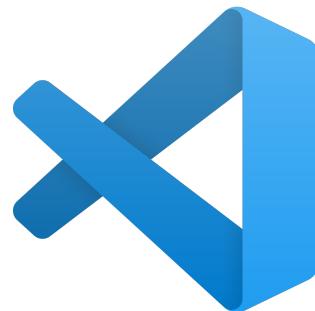
Всеки програмист си избира възможно най-подходяща среда за програмиране в зависимост от изискванията на проекта, нужните функционалности на средата, устройството, върху което се програмира, възможността за добавяне на инструменти за улесняване, както и лични преференции спрямо изгледа.

1. Atom - бесплатен редактор на текст и код, разработен от GitHub. Позволява на потребителите да инсталират пакети и теми от трети страни за да персонализират функциите и външния вид на редактора.



Изображение 1.28 - Логото на Atom

- VS Code - редактор на код, разработван от Microsoft. Функциите включват поддръжка за отстраняване на грешки, оцветяване на синтаксис, интелигентно допълване на код, фрагменти, рефакторинг на код и вграден Git. Потребителите могат да променят темата, клавишните комбинации, предпочтенията и да инсталират разширения, които да добавят допълнителна функционалност. Едно от предимствата в случая е ниското натоварване, което оказва върху машината, на която се използва.



Изображение 1.29 - Логото на VS Code

По време на разработката на дипломната работа трябва да бъдат стартирани три основни процеса, които могат да натоварят процесора. Това са интерфейса, сървърния софтуер и Ganache мрежата. Тук предимството на VS Code като по-лек текстов редактор е причината да го изберем.

## Глава 2

### 2.1 Функционални изисквания към дипломната работа

Проекта цели да реализира децентрализирана платформа и интерфейс за продажба и препродажба на продукти върху установена блокчайн мрежа. Цялостното решение се разделя на няколко основни части.

Първата и най-важна се състои от умните договори, които съдържат в себе си основната логика за функционирането на платформата. Те трябва да са максимално оплекотени и бързи, затова отговарят само за създаването и плащането на продукти, както и абсолютния минимум други необходими логистически операции.

Втората част е интерфейс за потребителя. Смарт контрактите са публични и общодостъпни, но единствения начин да се комуникира с тях е чрез програмни интерфейси, предоставяни от различни доставчици като Infura, Alchemy и другу. Затова потребителя има нужда от оптимизиран и удобен графичен интерфейс, чрез който да изпълнява заявките си.

В решението включваме и back-end за да се възползваме от всички плюсове на Web2.0, сред които са ниските цени за запазване на данни и бързодействие при изпълняване на заявки. Основната цел на базата данни е индексираща.

### 2.2 Избор на езика за програмиране и софтуерните средства

Използваме Solidity за децентрализираните приложения, защото е широко разпространен и има голяма общност и наличие на учебни материали. Езикът е от високо и ниво и променливите се дефинират с конкретни типове, което намалява грешките в кода по време на изпълнението му.

За тестовете, скриптовете, учеб интерфейса и сървърния софтуер използваме JavaScript заради удобните библиотеки за комуникация с умни договори, както и изградените инструменти за работа в сферата. Други плюсове са голямата общност и

наличието на добра документация за използваните библиотеки. Предоставя вграденото изпъкнение на асинхронни операции чрез `async/await` API-я.

## 2.3 Описание на структурата на базата данни

Всеки един продукт трябва да бъде записан в умния договор, който отговаря за неговата продажба. Затова в двата пазарни договора имаме две структури за продукти. В приложението за обикновено продаване са включени името, цената, продавача, купувача, датата на добавяне, тайна информация за админа, инструкции за доставка, линк за снимка, статус за плащане, потвърденост и доставка. В приложението за аукциони са включени същите полета, но купувачът е заменен с източник на наддаване, вместо статус за плащане има дата за край на аукциона и променлива за най-голямо наддаване.

Използваме и сървър, чрез който да се възползваме от ползите на Web2.0, сред които са бързото и евтино изпълнение на код, както и възможността да правим промени по кода в последствие. Повтаряме полетата от умните договори и в MySQL базата данни. Единствената нова таблица е тази за `offchain` наддаванията. Там полетата са за индекс на продукта, наддаващ, стойност, инструкции за доставка и подписана транзакция.

## 2.4 Избрани технологии и спомагателни инструменти

### 2.4.1 Етериум

Етериум е първата блокчейн мрежа, имплементираща изпълнението на децентрализиран код. Разработката ѝ започва през 2014, но не е публична преди 30 юли 2015. Виталик Бутерин е сред основните създадети и лица на Фондацията Етериум (EF), която отговаря за разработката на протоколите и промените в системата. Днес самата мрежа е на второ място по пазарна капитализация сред блокчейн технологиите. Сумарната стойност на етърите в циркулация е \$204 милиарда за февруари 2023.

Както Биткойн, така и Етериум разполага с естествена монета, която служи като награда за валидаторите, но докато един биткойн се разделя на  $10^{8}$  сатошита, така един етър се разделя на  $10^{9}$  гигауей, което са  $10^{18}$  уей.

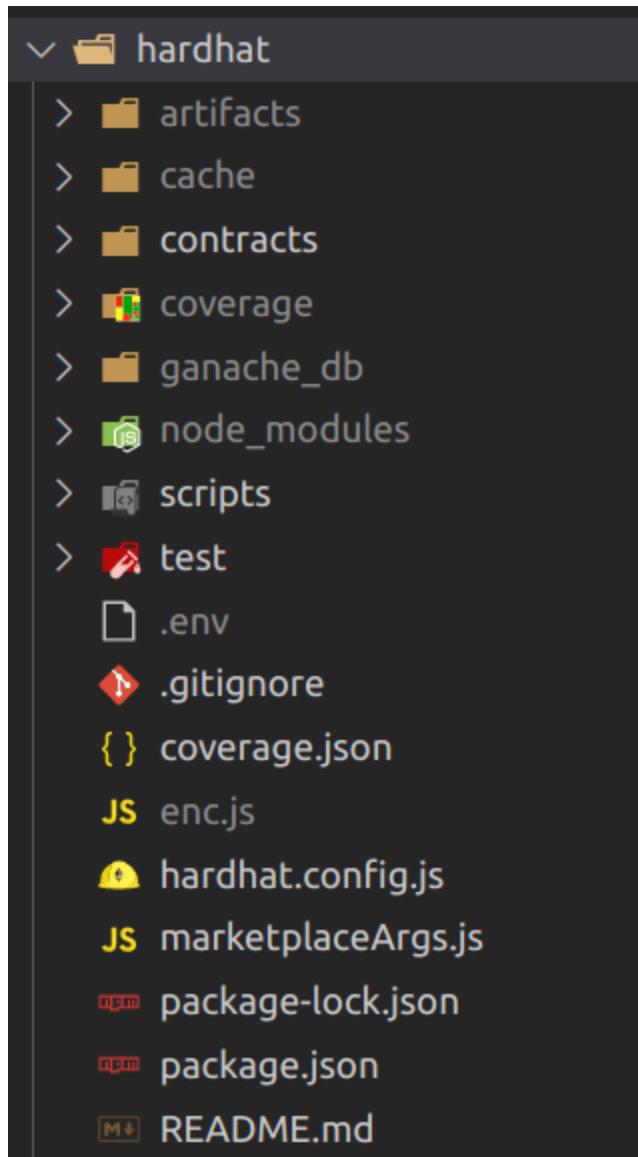
Едни от най-разпространените приложения на мрежата са създаването на

апликации за децентрализирани финанси (DeFi), създаване на договори за токенизация по стандарта ERC-20 и създаване на незаменими и уникални виртуални артикули по стандарта ERC-721 (NFT). Всеки умен договор наподобява клас с атрибути и методи, който може да наследява други класове. При внедряването му ние правим инстанция с определен адрес в мрежата.

За изпълнението на програмния код се използва Виртуалната машина на Етериум (Ethereum Virtual Machine), която представлява единно образование, поддържано от всички възли. Всички протоколи в мрежата са създадени за да обслужват непрекъснатата комуникация с EVM. Функцията на EVM е да поддържа и променя дистрибуираното състояние на мрежата и нейните компоненти. [12]

## 2.4.2 Hardhat

Hardhat е инструмент, създаден за разработването и тестването на умни договори посредством JavaScript и Node.js. Сред използваните функционалности са писането и автоматичното компилиране на договори, изпълняване на тестове, преглед на покритието на тестовете, писане на скриптове за внедряване на договорите в избрани мрежи и верифицирането на договорите в сайтове като <https://etherscan.io/>.



Изображение 2.1 - Файлова структура на Hardhat

Във файловата структура на този модул файловете са разделени според операциите, за които отговарят. Показани са в изображение 2.1. Папката contracts съдържа договорите, чиято компилация генерира файловете в папката artifacts. Най-важните файлове в artifacts са ABI инструкциите. Application Bytecode Interface файловете са във формат JSON и служат за описание на аргументите и връщаните стойности на нашите контракти. В стандартния случай не очакваме приложенията, които използват нашите договори да имат техния некомпилиран код, затова трябва да им предоставим данни за входните данни преди да ги подадат до блокчейна. В Coverage се пазят резултатите от проверката за

прокритие на тестовете, а в папката tests са самите автоматизирани тестове. В папката scripts е скрипта за внедряване на договорите в избраната мрежа. Папката ganache\_db не е част от структурата на Hardhat. Използваме локална мрежа ganache и тази папка служи за запис на изпълнените събития до момента. В hardhat.config.js се пазят всички специфични настройки, като API ключове за доставчици на услуги като Infura (provider, който прави връзка с goerli мрежата) и etherscan (предоставя начин за верификация на договорите).

За компилация на договорите използваме ‘npx hardhat compile’, а за тестовете ‘npx hardhat test’ с опция да се подаде конкретен файл. При приключване на тестовете използваме ‘npx hardhat coverage’, който ни показва покритието им.

93 passing (22s)					
File	%Stmts	%Branch	%Funcs	%Lines	Uncovered Lines
<b>contracts/</b>					
<b>AgoraToken.sol</b>	<b>100</b>	<b>100</b>	<b>100</b>	<b>100</b>	
<b>Marketplace.sol</b>	<b>100</b>	<b>100</b>	<b>100</b>	<b>100</b>	
<b>SimpleAuction.sol</b>	<b>100</b>	<b>100</b>	<b>100</b>	<b>100</b>	
<b>SimpleSeller.sol</b>	<b>100</b>	<b>100</b>	<b>100</b>	<b>100</b>	
<b>All files</b>	<b>100</b>	<b>100</b>	<b>100</b>	<b>100</b>	

Изображение 2.2 - Покритие на автоматизирани тестове

Командата ‘npx hardhat verify {contract address} –network goerli’ качва умния договор, който отговаря на подадения адрес в <https://goerli.etherscan.io/> за отчетност пред потребителите. Това позволява да видимия Solidit код на договора.

#### 2.4.3 Solidity

Обектно ориентиран език от високо ниво, създаден за изграждане на умни договори върху EVM. Използва фигурни скоби, подобно на езиците, по чиито модел е неговия дизайн - C++ и JavaScript. Пише се статично, което означава, че всяка променлива си има определен тип още от дефинирането. Това намалява грешките по време на изпълнение на компилирания код. [13]

Разполага с много функционалности, сред които са наследяване на контракти, използване на библиотеки, имплементиране на интерфейси, модификатори за функции, рекурсия, използване на асемблер и др.

Разполага с изключително голяма общности и учебни материали.

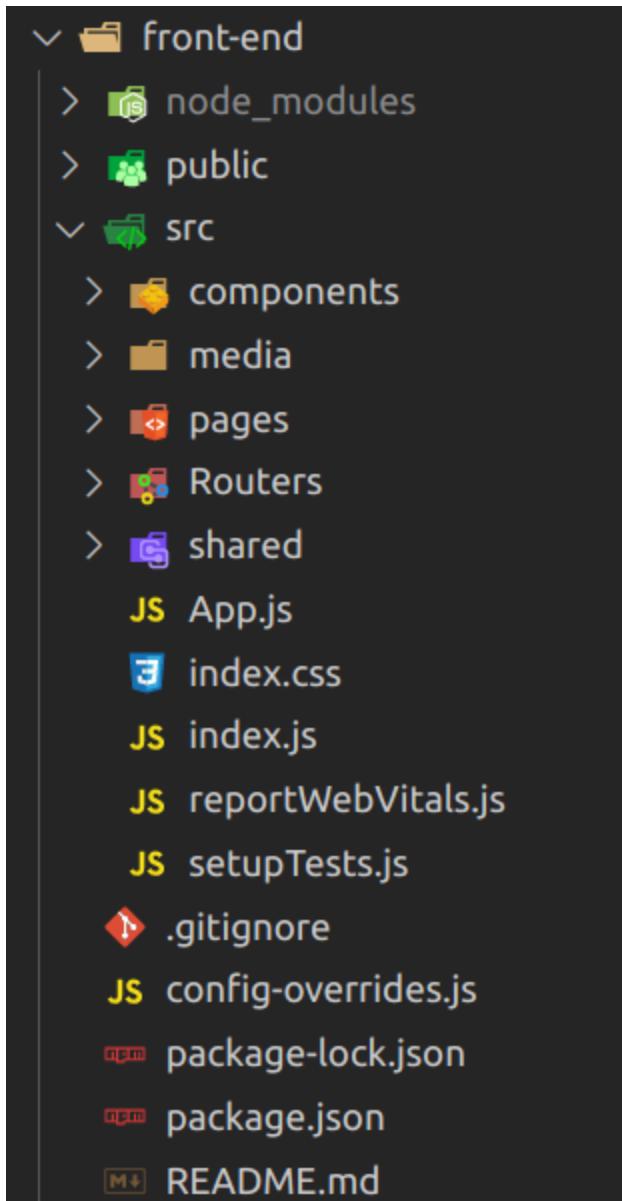
#### 2.4.4 Ganache

Ganache е симулатор на локална етериум мрежа, в която можем да тестваме умните си договори. Разполага със CLI, който да ни предостави лесен достъп до минималните функционалности. Тази мрежа е за предпочтение пред Goerli по време на разработка, защото със стартирането разполагаме с акаунти с етъри, много по-бързо приемане на транзакциите и лесно нулиране на мрежата. За стартиране на мрежата използваме командата ‘ganache-cli -d -m "myself armed safe reveal tissue bag milk coil call sweet adult clever" --db ./ganache\_db’, която стартира локалната мрежа. Аргумента ‘-d’ сигнализира, че искаме мрежата да е детерминистична, ‘-m’ сигнализира, че искаме да използваме мнемонична фраза за генериране на акаунтите, а ‘--db’ показва път до база данни, в която да бъдат запазени действията, които сме извършили. С тази команда можем да стартираме и спираме мрежата, продължавайки откъдето сме стигнали последни път.

#### 2.4.5 React.js

Инструмент, разработен от Facebook, използван за разработка за графични уеб потребителски интерфейси. Работи с JavaScript и TypeScript. Разработката се осъществява чрез изграждане на преизползваеми и независими компоненти, които се включват като градивни части за цялостния интерфейс.

За изпълнението на дипломната работа се използва React.js заради голямата общност, наличието на много учебни материали, както и модуларността на кода във финалния продукт.



Изображение 2.3 - Файлова структура на React апликация

Основния код на програмата се съдържа в папката src. Components съдържа компонентите, които се очаква да се преизползват на много места, например навигационното табло. Pages е за компоненти, които ще заемат цялата страница и ще се показват само в един контекст, както се случва със страниците за изброяване и създаване на артикули. Media е за изображения, а Routes за дефиниране на пътища към страниците. Папката shared съдържа ABI файлове за умните договори и адресите им в JSON формат.

#### 2.4.6 Node.js

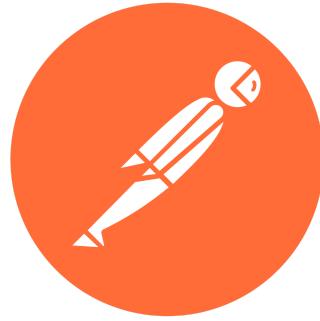
Node.js е среда за асинхронно изпълнение на JavaScript за скалируеми сървърни софтуери. Изграден е върху средата V8, разработена от Chrome. Едно от предимствата на Node.js пред другите сървърни технологии е способността му бързо да изпълнява голям брой заявки. Всички допълнителни библиотеки в един Node.js проект могат да бъдат свалиени чрез npm, yarn или други ресурсни мениджъри, което ни дава голям избор от допълнителни софтуерни пакети.

#### 2.4.7 Ethers.js

Представлява JavaScript библиотека за комуникация с децентрализирани приложения. Предоставя класове, които да служат за абстракция на договорите, методи за работа с данни, специфични за етериум, и интерфейси за избираеми мрежи. За разработката на проекта се използва Ethers.js заради добавените функционалности - методи за конвертиране на единици и константи. Предоставя методи, които изпълняват цяла операция наведнъж чрез допълнителна абстракция. Особено полезен е в потребителския интерфейс, където допринася с малкия си размер. Въпреки, че за сървърната част на проекта може да ни трябват повече функционалности, там също се използва Ethers.js за да поддържаме консистентност във всички модули.

#### 2.4.8 Postman

Десктоп приложение, използвано за ръчно тестване на REST API. Комуникацията между сървъра и потребителския интерфейс се осъществява посредством API, който трябва да бъде тестван преди да се интегрира в уеб частта на проекта. За целта използваме функционалностите на Postman, сред които са създаването, приемането и запазването на шаблони за заявки.



Изображение 2.4 - Логото на Postman

The screenshot shows the Postman application interface. On the left, there's a sidebar with 'My Workspace' containing collections like 'web3marketplace' with various API endpoints such as 'GET Seller Products', 'POST UPDATE Seller product', etc. The main workspace shows a 'GET' request for 'localhost:5000/a/p/0'. The 'Body' tab displays a JSON response with a single item:

```
1  {
2     "id": 1,
3     "instanceId": 0,
4     "name": "new p",
5     "minimalPrice": "0x15ec1616231deb",
6     "seller": "0x5077c057c943b24080d3faf55d405029540cc0C",
7     "currentBidder": "0x5077c057c943b24080d3faf55d405029540cc0C",
8     "bidAmount": "0x1ae0e75d571b",
9     "finishDate": "2023-10-10T08:10:00.000Z",
10    "addDate": "2023-02-28T10:07:11.000Z",
11    "linkForMedia": "http://localhost:5000/i/1677578822996-images.jpeg",
12    "marketHashOfData": {}},
13    },
14    "approved": false,
15    "delivered": false,
16    "deliveryInstructions": {},
17    },
18    "description": "new description 2.2",
19    "createdAt": "2023-02-28T10:07:11.000Z",
20    "updatedAt": "2023-03-01T15:28:51.000Z"
```

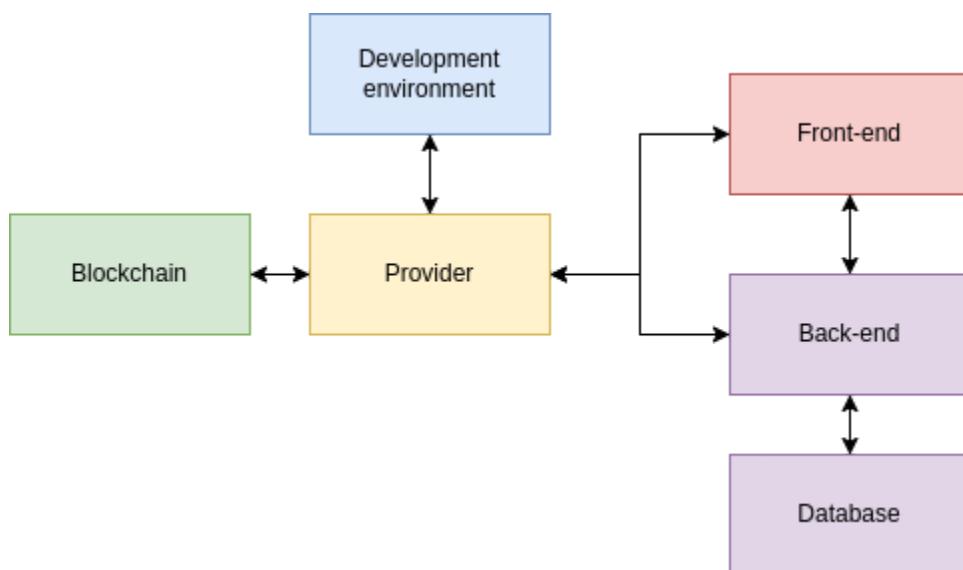
Изображение 2.5 - Интерфейсът на Postman

На изображение 2.5 е показан интерфейса на Postman, през който правим GET заявка до “localhost:5000/a/p/0” за да получим първия аукцион. Информацията се връща в JSON формат.

# Глава 3 Реализация на проекта

## 3.1 Архитектура

С разработването на децентрализирани приложение идва и проблемът с неизменимостта (immutability) на кода, който внедряваме в блокчейн мрежата. Затова трябва да имаме ясна представа за цялостната архитектура и комуникация между компонентите преди да започнем писането на смарт контрактите.

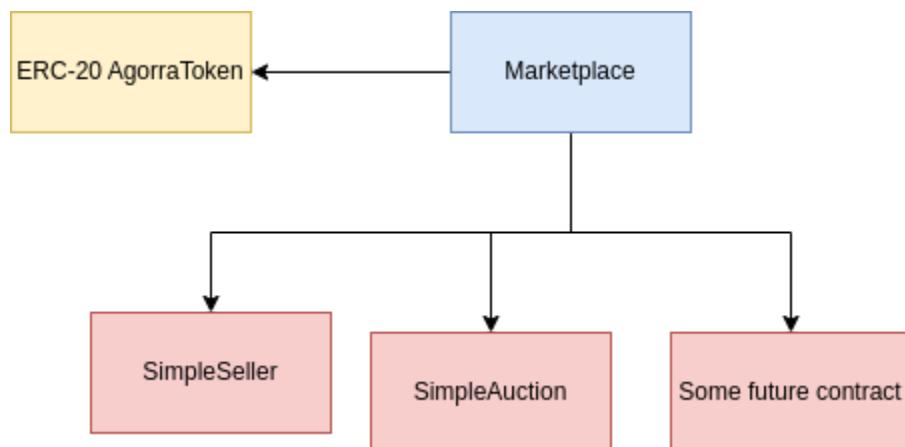


Когато искаме да комуникираме с истинска тестова или продуктова блокчейн мрежа ще използваме provider-ите на Infura или Metamask. По време на тестването на договорите ще използваме вградения provider за локалната мрежа на Ganache.

Уеб интерфейсът ни трябва да може да достъпва информация от блокчейна и от базата данни за оптимално потребителско преживяване. Сървърният софтуер трябва да има достъп до излъчените събитията в мрежата, както и да изпълнява определени предварително подписани транзакции.

### 3.2 Контракти

За реализация на дипломната работа смарт контрактите трябва да са модуларни, което означава, че всеки трябва да си има отделна функция, която да не се препокрива с останалите. Трябва да имаме обединяващо звено между всички типове пазари (продавач, аукцион, холандски аукцион, лотария и тн.), както и да поддържа добавяне на нови след внедряване на първите. Към цялостното изпълнение се включва и функционалност за осъществяването на предварително подписани транзакции за преместване на средства от потребител към умен договор. Това се постига с помощта на специализиран умен договор, който да отговаря за създаването и преместването на токени по стандарта ERC-20. Marketplace не е родителски клас, от който другите договори да наследяват структурата си, той е стартовата точка, която съдържа адресите на другите части от блокчайн архитектурата на децентрализираното приложение.



Диаграма 3.2 - Архитектура на умните договори

### 3.2.1 Marketplace

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

import "@openzeppelin/contracts/access/Ownable.sol";
import "./AgoraToken.sol";
contract Marketplace is Ownable {
    event addedCourier(address adder, address courier);
    event removedCourier(address remover, address courier);
    event addedAdmin(address adder, address admin);
    event removedAdmin(address remover, address admin);

    ...
}
```

Създаваме Marketplace договора като обединяващо звено между всички останали договори. Самия той наследява Ownable, който е разработен от OpenZeppelin като стандарт за други договори, които трябва да имат собственик. Ownable имплементира силно одитирани методи, свързани с правата за викане на функции. Стандартна практика е да се използват заготовки на силно одитирани договори от утвърдени компании.

В началото определяме версията на Solidity и наследяваме OpenZeppelin Ownable заготовката. AgoraToken.sol е нашия ERC-20 договор, който отговаря за токенизацията.

Посочваме event-и, които имат индексираща функция. Много по-бързо и лесно можем да индексираме в една блокчейн мрежа, защото можем да направим listener-и за тях в нашия JavaScript код. Част от бъдещата разработка е имплементирането на технологията The Graph, която ни помага в правенето на заявки. Има нужда от event-и, които да слуша.

```

constructor(bytes memory _publicKey) Ownable() {
    require(_publicKey.length==65,"Bad public key");
    publicKey = _publicKey;
}

struct Market{
    address contractAddress;
    address addedBy;
    string name;
    uint addDate;
}

bytes public publicKey;
address public myToken;
address[] public marketAddresses;
uint public marketCount;
mapping(address => Market) public markets;
mapping(address => bool) public couriers;
mapping(address => bool) public admins;
...

```

В показания код декларираме променливите на договора. Bytes е тип за променлива с динамична дължина, който използваме за publicKey. На това място ще се съхранява публични ключ на админа на умния договор, с който продавачите ще могат да качват скрита информация, с която да потвърждават легитимността на продуктите си пред собственика. Това дава възможност на админа да определя някои продукти като валидни за да може крайния купувач да има доверие на продавача. MyToken е адреса на договора за ERC-20 токените. Списъкът от адреси marketAdresses запазва адресите на умните договори, отговарящи за създаването и продаването на продукти. За начало това са SimpleSeller и SimpleAuction. MarketCount е броя на тези пазари, а mapping-a markets отговаря за връзката между адрес и структурата Market, създадена по-горе. В този контекст трябва да свеждаме използването на for и while цикли до минимум, затова използваме mapping за информацията на пазарите, но в същото време пазим адресите им и в списък

при нужда от достъпване на всички. Имаме и променливи за куриери и админи, които да пазят ролите на определени адреси. Адрес, който има записан true в мапинга admins има правото да добавя куриери. Куриерите имат право да извикват функция deliverProduct в търговските договори, така пренасочват активи от умния договор към продавача.

Конструкторът се изпълнява по време на deploy за да зададе начална стойност на publicKey и owner. Използва конструкторът на Ownable, който всъщност е parent contract.

```
modifier onlyAdmin(){  
    require(admins[msg.sender]==true,"Sender is not an admin!");  
    _;  
}
```

Модификаторът onlyAdmin се използва за ограничаване на достъпа до определени функции. Require изразът проверява дали адресът на източникът на транзакцията (msg.sender) е админ чрез admins[] мапинга. В противен случай спира извикването на функцията и връща грешка “Sender is not an admin!”. Синтаксисът \_; отбелязва началото на изпълнението на извиканата функция, чрез която сме стигнали до модификаторът.

```

function addContract(address _contractAddress, string calldata _name) public onlyOwner {
    require(_contractAddress!=address(0),"Address shouldn't be 0");
    require(bytes(_name).length != 0, "Name length shouldn't be 0");
    require(
        markets[_contractAddress].contractAddress == address(0),
        "Market already added");

    markets[_contractAddress] = (Market(
        _contractAddress,msg.sender,_name,block.timestamp));
    marketAddresses.push(_contractAddress);
    marketCount+=1;
}

```

AddContract функцията добавя търговски договор в списъка на Marketplace. Използва модifikаторът onlyOwner, който е имплементиран от Ownable заготовката на OpenZeppelin така, че само собственикът да може да вика тази функция. Първите 3 реда валидират подадената информация. Проверяват дали адресът не е 0, дали подаденото име не е празно и дали не се опитваме да добавим един договор 2 пъти. Ако всичко е наред се създава нова структура, която се записва в мапинга за договори. Адресът се записва в списъка с адреси, а броя се увеличава.

```

function addCourier(address _courier) public onlyAdmin{
    require(_courier!=address(0),"Address shouldn't be 0");
    couriers[_courier]=true;
    emit addedCourier(msg.sender,_courier);
}

function removeCourier(address _courier) public onlyAdmin{
    couriers[_courier]=false;
    emit removedCourier(msg.sender,_courier);
}

function addAdmin(address _admin) public onlyOwner{
    require(_admin!=address(0),"Address shouldn't be 0");
    admins[_admin]=true;
    emit addedAdmin(msg.sender,admin);
}

function removeAdmin(address _admin) public onlyOwner{
    admins[_admin]=false;
    emit removedAdmin(msg.sender,admin);
}

```

Функциите за добавяне и премахване на куриер използват модификатора `onlyAdmin` и излъчват event за съответното действие, след като са валидирали информация и са добавили адреса.

Функциите за добавяне и махане на админ използват модификатора `onlyOwner` за да може само собственикът да извършва такива действия.

### 3.2.2 AgoraToken

AgoraToken наследява ERC-20 стандарта, разработен от OpenZeppelin и отговарящ за създаване и обменянето на token-и.

Това е умния договор, който отговаря за разплащанията между купувач и умен договор, след това между договор и продавач. Основната цел е да се създават предварително подписани транзакции за активи, които да бъдат валидни само в контекста на тази екосистема. По този начин избягваме подписането на транзакции в native монетата етър и избягваме други рискове, свързани със сигурността.

```
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "hardhat/console.sol";

contract AgoraToken is ERC20 {
    mapping(bytes32 => bool) nonces;
    constructor() ERC20("AgoraToken", "AGR") {}

    function buyTokens() external payable{
        require(msg.value>0,"Should send some eth");
        _mint(msg.sender, msg.value);
    }

    function sellTokens(uint amount) external {
        require(amount>0,"Amount must be >0");
        require(
            this.balanceOf(msg.sender)>=amount,
            "Your balance is < amount you want to sell");
        _burn(msg.sender, amount);
        payable(msg.sender).transfer(amount);
    }
}
```

В началото добавяме стандартната заготовка от OpenZeppelin за ERC-20, която AgoraToken наследява. След това добавяме библиотека, разработена от Hardhat, която се използва за дебъгване на кода. Езика Solidity не разполага с `console.log()` или друга подобна функция, която директно да изписва информация в конзолата на Hardhat. Опциите са четене на стойности или на изльчени event-и, но това усложнява процеса. В последната версия на договорите няма да бъдат кченти с този `import`.

Nonce играе ролята на ID за всяка предварително подписана транзакция. Целта на тази променлива е да отбелязва кои ID-та съответстват на осъществени транзакции. Целта е веднъж подписана транзакция да не бъде осъществена втори път. Тази проверка се прави във функцията `transactWithSignature`.

Дефинираме функции за купуване и продаване на AgoraToken-и в съотношение 1:1 с етъри. `Msg.value` е стойността на транзакцията т.е. етърите, които изпращаме. `_mint` е метод на ERC-20 родителския договор и създава новите токени. `SellTokens` “изгаря” определен брой токени и връща съответното количество етъри на потребителя чрез `.transfer` метода. Методът проверява дали притежаваме активите, които искаме да продадем. `_burn` изгаря токените, които продаваме.

```

function prefixed(bytes32 messageHash) internal pure returns (bytes32) {
    bytes memory hashPrefix = "\x19Ethereum Signed Message:\n32";
    return keccak256(abi.encodePacked(hashPrefix, messageHash));
}

function splitSignature(bytes memory sig) internal pure returns
    (uint8 v, bytes32 r, bytes32 s){
    require(sig.length == 65,"Signature has bad length");
    assembly {
        r := mload(add(sig, 32))
        s := mload(add(sig, 64))
        v := byte(0, mload(add(sig, 96)))
    }
}

function getMsgSigner(bytes32 _hashedMessage, bytes memory signature)
    internal pure returns (address) {
    bytes32 prefixedHashMessage = prefixed(_hashedMessage);
    (uint8 v, bytes32 r, bytes32 s) = splitSignature(signature);
    address signer = ecrecover(prefixedHashMessage, v, r, s);
    return signer;
}

```

Тези три функции са pure и internal, което означава, че работят само с данни от аргументите си, а не с променливи от договора. Също, че могат да се викат само вътрешно. Те отговарят за валидирането на подписано съобщение.

Prefixed взима сировата информация, която потребителят подписва на front-end-а и ѝ добавя стандартния префикс за Етериум. Започва с “\x19”, след това текст и накрая размера на информацията. С цел опростяване на тази функция винаги работим с 32-байтови съобщения. Накрая изпълнява хеш на резултата и го връща. Abi.encodePacked извършва проста конкатенация. Keccak256 е CHF.

SplitSignature разделя подписаното съобщение на основните му три компонента и ги връща като резултат. В случая използваме inline асемблер за да намалим таксата за транзакция.

GetMsgSigner получава сурвото съобщение преди подписване и го подава на prefixed. Също го получава и подписано и го подава на splitSignature. Резултатите от тези две функции подава на есгесовер, който връща адреса на източника на подписа. По-късно проверяваме дали адресът съвпада с очаквания.

```
function transactWithSignature (
    uint expiration, bytes32 nonce, uint amount,
    address from, address to, bytes memory signature ) public{
    require(expiration>block.timestamp,"Signature expired");
    require(nonces[nonce]==false,"Nonce used");
    require(amount>0,"Amount should be >0");
    require(from!=address(0),"From can't be address(0)");
    require(to!=address(0),"To can't be address(0)");
    require(amount<=balanceOf(from),"Not enough tokens");

    bytes32 message = keccak256(
        abi.encodePacked(expiration,nonce,amount,from,to));
    address signer = getMsgSigner(message,signature);
    require(signer==from,"Wrong arguments (recoveredAddress!=from)");
    nonces[nonce] = true;
    _transfer(from, to, amount );
}
```

TransactWithSignature приема като аргументи цялата информация, която се съдържа в сурвото съобщение, както и самата подписана транзакция. Потребителят подписва съгласие за изпращане на определено количество активи, до определена дата, до определен адрес. Това е цялата транзакция, която има ID nonce, който може да се използва само веднъж. Функцията проверява всеки един аргумент чрез require функции. Block.timestamp е datetime променлива, която се сравнява с expiration. Прави се проверка за това дали nonce е използван, както и дали изпращача притежава съответните активи. Променливата

message представлява реконструираното съобщение, което потребителят е подписан на front-end-а. Signer е резултата от internal функцията, която написахме за проверка на подписа и връща адреса на подписващия. Ако подписа е невалиден, то signer няма да е равен на from. Ако всичко е наред отбелязваме ID-то като използвано и преместваме активите чрез вътрешната функция \_transfer.

### 3.2.3 Simple Seller

Това е приложението, отговарящо за простите продажби на продукти. Продавачът създава продукт, купувачът плаща за него на договора, а след доставката токените пристигат в продавача. С всяка продажба администрация на сайта получава 1% от стойността.

```
contract SimpleSeller is Ownable{
    event sellerProductAdded(
        string name, uint price, address seller, uint index);
    event sellerProductSold(
        uint index, address buyer);
    event sellerProductDelivered(
        uint index, address seller, address courier);
    event sellerProductApproved(uint index);

    struct Product{
        string name;
        uint price;
        address seller;
        address buyer;
        uint addDate;
        string linkForMedia;
        bytes marketHashOfData;
        bool approved;
        bool paid;
        bool delivered;
        bytes deliveryInstructions;
    }
}
```

Дефинирани са 4 типа събития, които ще следим по-късно в сървъра - за създаване, продаване, доставяне и потвърждаване на продукт. Сървърът ще слуша за тях за да може да обнови информацията в базата данни за да можем по-късно да индексираме по-добре тези продукти. Структурата Product запазва в себе си основната информация, нужна за операцията на договора. Сред задължителните са име, цена и продавач. Buyer, addDate, paid, delivered и approved са поставени за да улеснят по-късното индексиране. LinkForMedia е линк към изображението на продукта, което ще се показва на сайта. Това може да е изображение, което е запазено на нашия или чужд сървър, както и в IPFS. MarketHashOfData е криптирана информация, която продавача поставя за да може админа на сайта да потвърди дали продуктът е валиден. Може да е посочен линк към сертификат за качество, видео клип за повече информация или друг тип секретни данни. Очаква се да бъде криптирана с публичния ключ, записан в Marketplace. При потвърждаване на продукта approved става true. DeliveryInstructions са инструкциите за доставка, които купувачат поставя.

```
uint public belongsToContract=0;
```

```
Marketplace public ownerMarketplace;
mapping(uint => Product) public products;
mapping(address => uint[]) public sellerToProductIndexes;
//seller to (product to money)
mapping(address => mapping(uint => uint)) public owedMoneyToSellers;
mapping(address => mapping(uint => uint)) public owedMoneyToBuyers;
```

```
uint public productCount=0;
```

BelongsToContract е количеството токени, които договора има право да изпрати до Marketplace след взимане на комисионна. Баланса на договора не винаги е равен на belongsToContract, понеже ако един предмет е купен, но не е доставен този договор ще държи в себе си съответната сума преди да я изпрати на продавача. OwnerMarketplace е договора, към който SimpleSeller може да изпраща активите си. От него получава информация за адреса на AgoraToken. Mapping-a sellerToProductIndexes държи списък с

индексите на продуктите на всеки продавач. OwedMoneyToSeller запазват количеството токени, които трябва да получи продавача след успешна доставка на продукт. Щом продукт 1 от продавач 0x01 е заплатен, ще бъде записано, че owedMoneyToSellers[0x01][1] трябва да е 99% от цената на продукта. Когато продукта е доставен тази стойност става 0 и изпращат парите до продавача. OwedMoneyToBuyers работи по почти същия начин, с разликата, че се записва цялата заплатена цена и се нулира, когато продукта е доставен. Купувачът не си получава парите обратно при доставка, те само се помнят преди нея.

```
modifier correctlyInstantiated{
    require(
        address(ownerMarketplace)!=address(0),
        "Doesn't have owner marketplace");
    require(
        address(ownerMarketplace.myToken())!=address(0),
        "No token specified");
    ...
}
```

Модификаторът correctlyInstantiated проверява дали в търговския договор е посочен главния Marketplace и ERC-20 договор към него. Използваме го в методи, в които ще достъпваме информация от ownerMarketplace.

```

function payProduct(
    uint index, bytes calldata deliveryInstructions,
    uint expiration, address from, bytes memory sig
) public correctlyInstantiated{
    Product storage p = products[index];
    require(p.seller!=address(0), "No such product");
    require(p.paid==false,"Product already bought");
    require(deliveryInstructions.length!=0, "No delivery instructions");
    bytes32 nonce=keccak256(abi.encodePacked(address(this),index));
    p.deliveryInstructions = deliveryInstructions;
    p.paid=true;
    owedMoneyToSellers[p.seller][index] = p.price*99/100;
    owedMoneyToBuyers[from][index] = p.price;
    products[index].buyer = from;

    AgoraToken token = AgoraToken(ownerMarketplace.myToken());
    token.transactWithSignature(
        expiration,nonce,p.price,from,address(this),sig);

    emit sellerProductSold(index, from);
}

```

Във функцията payProduct се случват няколко основни неща, свързани с плащането и записване на стойности. Първо се правят проверките от модификатора correctlyInstantiated, който проверява за ownerMarketplace и дали има посочен ERC-20 договор към него. След това функцията получава индекса на продукта, заедно с инструкции за доставка и подписаното плащане. В началото на функцията се проверяват входните данни и локално се създава nonce от адреса на договора и индекса на продукта. Потребителят е подписал плащането със същия nonce още в клиентската апликация. Задават се стойностите на owedMoneyToSeller и owedMoneyToBuyers. Накрая се изпълнява плащането с данните, които трябва да бъдат получени от transactWithSignature. Тази функция приема expiration, генерирания nonce, цената, източник, получател и целия подпис. Нарочно много от тези аргументи не се подават на payProduct, защото можем да ги

намерим в паметта на контракта. Например изпращаните средства са цената на продукта, nonce се изчислява от адрес и индекс, а получателя е SimpleSeller. След това се изльчва събитието.

```
function deliverProduct(uint index) public correctlyInstantiated {
    Product memory p = products[index];
    require(p.seller!=address(0), "No such product");
    require(p.paid==true,"Product not paid");
    require(p.delivered==false,"Product already delivered");
    require(
        ownerMarketplace.couriers(msg.sender)==true,
        "Not an authorized courier");
    uint pay = owedMoneyToSellers[p.seller][index];
    belongsToContract+=p.price-pay;
    owedMoneyToSellers[p.seller][index] = 0;
    owedMoneyToBuyers[p.buyer][index] = 0;
    products[index].delivered=true;

    AgoraToken(ownerMarketplace.myToken()).transfer(p.seller,pay);
    emit sellerProductDelivered(index, p.seller, msg.sender);
}
```

DeliverProduct е функция, която само куриери трябва да могат да викат. Понеже е единствената такава функция в договора няма нужда от създаването на модификатор за проверка на куриер. Получава индекс за продукта и в началото проверява статуса на променливите. Можем да извършим проверка за статус на куриер в ownerMarketplace без притеснения, че ownerMarketplace е нулев адрес, защото е поставен модификатор correctlyInstantiated, който проверява това.

Ако всичко е наред се зануяват owednMoneyToSeller и owedMoneyToBuyer, след което токените се изпращат до продавача и се изльчва event.

```
function approveProduct(uint index) public productExists(index) onlyOwner{
    products[index].approved=true;
    emit sellerProductApproved(index);
}
```

Функцията approveProduct посочва продукта като потвърден и emit-ва събитието.

```
function transferFunds() public onlyOwner correctlyInstantiated {
    AgoraToken token = AgoraToken(ownerMarketplace.myToken());
    token.transfer(address(ownerMarketplace),belongsToContract);
    belongsToContract=0;
}
```

След успешното изпълнение на продажба и доставка, в AgoraToken трябва да е записано, че SimpleSeller вече притежава определени токени. Също променливата belongsToContract трябва да отразява печалбата. TransferFunds разрешава на собственика да преведе тези средства на ownerMarketplace-а. Там могат да се складират, докато обственика не ги изкара и продаде.

### 3.2.4 Simple Auction

```
contract SimpleAuction is Ownable{
    event auctionProductAdded(
        string name, uint minimalPrice, address seller, uint index);
    event auctionProductBid(uint index,address bidder, uint amount);
    event auctionProductDelivered(
        uint index, address buyer, address courier);
    event auctionProductApproved(uint index);

    struct Product{
        string name;
        uint minimalPrice;
        address seller;
        uint addDate;
        string linkForMedia;
        bytes marketHashOfData;
        bool approved;
        bool delivered;
        bytes deliveryInstructions;

        address currentBidder;
        uint bidAmount;
        uint finishDate;
    }
}
```

Договорът отговаря за създаването на аукциони и наддаването. Имплементира event-и, които да служат за обновяване на информацията в базите данни.

Структурата на продукта тук е подобна на тази в SimpleSeller, с разлика, че съдържа currentBidder, bidAmount, minimalPrice и finishDate.

```

...
bytes32 nonce = keccak256(
    abi.encodePacked(address(this),index,amount));
p.deliveryInstructions = deliveryInstructions;
AgoraToken token = AgoraToken(ownerMarketplace.myToken());
token.transactWithSignature(p.finishDate,nonce,amount,from,address(this),sig);
//RETURN PREV BID MONEY
if(p.currentBidder!=address(0)){
    owedMoneyToBidders[p.currentBidder][index] = 0;
    token.transfer(p.currentBidder,p.bidAmount);
}
owedMoneyToBidders[from][index] = amount;
p.bidAmount=amount;
p.currentBidder=from;

emit auctionProductBid(index, from, amount);
}

```

Това е част от функцията `bidForProduct`, която, също като в `SimpleSeller buyProduct`, валидира входните данни. `Nonce` тук е адрес, индекс и количеството токени, които се изпращат. Включваме и количеството, защото в противен случай `nonce`-овете за всяко наддаване ще са еднакви, следователно няма да работят след първия. Това решава и проблема с наддаването с една и съща стойност от няколко човека. Така ако Алиса иска да наддаде 1 етър, тя ще използва съответния `nonce`, но ако Боб иска след това да наддаде пак с 1 етър, то той няма да може, защото е изчислил `nonce`, който вече е използван.

При извършването на наддаването договорът всъщност прави депозит от името на наддаващия. Правим проверка за връщане на депозита на последния наддаващ, след това прехвърляме транзакцията на ERC-20 договора.. Променливите за дължими средства се променят и накрая се излъчва event.

### 3.3 Тестове на умните договори

Децентрализираните приложения в Етериум блокчейна не могат да бъдат обновявани след внедряването им в мрежата. Това означава, че всеки бъг, допуснат по време на разработка, остава в умния договор и не може да бъде премахнат. За програми, които предстои да управляват големи количества активи това е недопустимо, затова една от най-важните части от разработката е изграждането на пълни и автоматизирани тестове. За целта използваме Hardhat и Chai, които ни дават възможност да пишем скриптовете и тестовете си на JavaScript.

```
const { expect } = require("chai");
const { ethers } = require("hardhat");
const dotenv = require("dotenv");
dotenv.config();
```

Тестовете ни започват с import на нужните библиотеки. Hardhat ни дава достъп до библиотеката ethers, която ни служи като интерфейс към умните договори. Chai е библиотеката за тестване и от нея взимаме функцията expect, която взима като аргумент първата стойност, която сравняваме, и връща обект с методи за сравнение с втора стойност. Dotenv е средата, която съдържа в себе си чувствителна информация. Файлът с нея не се качва на хранилището.

```
describe("Marketplace", function () {...});
```

За създаването на тестове се използват describe и it блокове. Describe съдържа в себе си it блокове, които са самите тестове. BeforeEach блокове се изпълнява преди всеки тест. Като аргументи взима string за описание и функция, която представлява неговото тяло.

```

beforeEach(async function () {
    accounts = await ethers.getSigners();
    Marketplace = await ethers.getContractFactory(
        "Marketplace");
    publicKey = await ethers.utils.computePublicKey(
        process.env.ACCOUNT_PRIVATE_KEY);
    publicKey = hexToArray(publicKey);
    marketplace = await Marketplace.deploy(publicKey);

    AgoraToken = await ethers.getContractFactory("AgoraToken");
    agoraToken = await AgoraToken.deploy();

    SimpleSeller = await ethers.getContractFactory(
        "SimpleSeller");

    simpleSeller1 = await SimpleSeller.deploy();
    simpleSeller2 = await SimpleSeller.deploy();
    simpleSeller3 = await SimpleSeller.deploy();
});


```

Chai ни позволява да пишем BeforeEach процедура, която да се изпълни преди всеки тест, съдържащ се в съответния describe блок. Показания beforeEach ще се изпълни преди всеки един тест, защото се намира в началото на главния describe. Ethers.getSigners() ни връща списък от класове за акаунти, представляващи всеки един потребител. Marketplace с главно “M” е обекта за договора ни, докато е marketplace, е инстанция на внедрен договор. Същото се отнася за SimpleSeller и simpleSeller, както и AgoraToken и agoraToken. PublicKey е публичния ключ на админа на marketplace. Изчисляваме го от предварително подгответен частен ключ.

```

describe("Deployment", async function () {
  it("Should have the correct owner", async function () {
    const addressOfOwnerOfMarketplace = await marketplace.owner();
    expect(addressOfOwnerOfMarketplace).to.equal(
      accounts[0].address,
      "Wrong owner");
    expect(addressOfOwnerOfMarketplace).to.not.equal(
      accounts[1].address,
      "Wrong owner");
  });
  ...
});

```

Показан е тест, сравняващ стойности. Тук проверяваме дали при deployment са се създали договори с правилно инициализирани променливи. Проверява се адреса на собственика, който очакваме да е адреса на акаунт 0 от accounts. Първо взимаме адреса на собственика и го запазваме в addressOfOwnerOfMarketplace. След това ползваме expect за сравнение с accounts[0].address. Ако не съвпадат получаваме грешка “Wrong owner” и теста не минава. Проверяваме дали е различен от адреса на accounts[1]. Ако не е различен също ще даде грешка.

```

it("Not owner add contract", async function () {
  await expect(
    marketplace.connect(accounts[1])
    .addContract(simpleSeller1.address, "SimpleSeller1"))
    .to.be.revertedWith('Ownable: caller is not the owner');
  expect( (await marketplace.getMarketAddresses()).length).equal(0);
});

```

В този тест проверяваме дали се ограничава достъпа до функции, които трябва да са достъпни само за собственика. Чрез метода .connect, извършваме транзакция от името на accounts[1], който не е собственик. Извикваме методът addContract, който е достъпен само за собственика. Очакваме функцията да не ни позволи да извършим действието. Това

се проверява с `.to.be.revertedWith()`, което гледа съобщението, получено от грешката на договора, и го сравнява със стойността, подадена в скобите.

```
it("Has correct market address", async function () {
    expect(await marketplace.addContract(
        simpleSeller1.address,"SimpleSeller1"))
        .to.not.throw;
    expect(await marketplace.addContract(
        simpleSeller2.address,"SimpleSeller2"))
        .to.not.throw;
    expect(await marketplace.addContract(
        simpleSeller3.address,"SimpleSeller3"))
        .to.not.throw;
    expect( (await marketplace.getMarketAddresses() ).length).equal(3);
    expect( await marketplace.marketCount()).equal(3);
    expect((await marketplace.markets(simpleSeller1.address)).addedBy)
        .equal(accounts[0].address);
    expect((await marketplace.markets(simpleSeller2.address)).addedBy)
        .equal(accounts[0].address);
    expect((await marketplace.markets(simpleSeller3.address)).addedBy)
        .equal(accounts[0].address);
});
```

Това е пълен тест за успешно добавяне на нови инстанции на един и същ договор. `To.not.throw` проверява дали всичко е наред с транзакцията. След добавянето на новите три договора проверяваме стойностите в `marketplace`.

```

sigData = {
    "currentTime": Math.floor(Date.now()/1000),
    "futureTime": Math.floor(Date.now()/1000)+1000,
    "nonce0": await ethers.utils.keccak256(
        await ethers.utils.solidityPack(
            ["address","uint"],
            [simpleSeller.address,0]
        )
    ),
    "nonce1": await ethers.utils.keccak256(
        await ethers.utils.solidityPack(
            ["address","uint"],
            [simpleSeller.address,1]
        )
    )
}

```

SigData запазва данни, които преизползваме в тестовете за създаване на подпис на транзакции. Целта е да се избегне повторно извършване на операциите, които ще дадат един и същ резултат. Current time е datetime променлива, която намира секундите, изминали от 1. януари 1970 до сега. Това е подобно на стандарта UTC, с разликата, че UTC е в милисекунди, а умните ни договори работят с времето в секунди. FutureTime е времето след 1000 секунди. Първото ще използваме за валидиране на грешки, а второто за правилно изпълнение на методите. Nonce0 и nonce1 са id-тата за транзакциите за продукти 0 и 1. Изчисляваме ги с конкатенация от solidityPacked и хеширане с keccak256, предоставени от ethers.utils.

```

const identity = EthCrypto.createIdentity();
testingPubliCKey = identity.publicKey;
testingPrivateKey = identity.privateKey;
hashedData = await encryptWithPublicKey(secretMessage,testingPubliCKey);
encryptedDeliveryInstructions = encryptWithPublicKey(
    deliveryInstructions,
    testingPubliCKey
);

```

Модулът EthCrypto ни помага с криптирането на съобщения с публичен и частен ключ. Първо ги генерираме от една самоличност. Променливата hashedData се използва като аргумент при създаване на продукт. Играе ролята на marketHashOfData - информация която е достъпна само за собственика на договорите, затова се криптира с публичен ключ. В бъдеще ще може да се декриптира с частния ключ. Същото важи и за encryptedDeliveryInstructions, които са инструкциите за доставка на продукт. Те трябва да са достъпни само за продавача. С цел опростяване на тестовете продавача и собственика на договорите са един човек, представляван от една двойка ключове.

```

const message = await ethers.utils.arrayify(
    await ethers.utils.keccak256(
        await ethers.utils.solidityPack(
            ['uint','bytes32','uint','address','address'],
            [sigData.futureTime,sigData.nonce0,
            oneETH,accounts[1].address,simpleSeller.address]
        )
    )
);
const signature = accounts[1].signMessage(message);

```

Самото подписване на транзакция за купуване или наддаване се извършва в няколко стъпки. Изготвя се тялото на самото съобщение, което в случая съдържа expiration (sigData.futureTime), nonce, количеството, адреса на източника и адреса на получателя.

След това се прекарва през хешираща функция, която да го превърне в 32-байтова стойност и накрая се подписва чрез signMessage метода на accounts[1]. Това подписано съобщение може да бъде изпратено на payProduct или на bidForProduct заедно с другите параметри.

```
await expect(simpleSeller.connect(accounts[3]).deliverProduct(0))
.to.emit(simpleSeller,"sellerProductDelivered")
.withArgs(0,accounts[0].address,accounts[3].address);
```

По този начин тестваме дали определен event е бил излъчен. В случая доставяме продукт 0 от името на акаунт 3. С .to.emit проверяваме дали съответното събитие е било излъчено, а чрез .withArgs проверяваме стойностите му.

```
jo@jos-laptop:~/Documents/dev/TUES/diplomna/Web3Marketplace/hardhat$ npx hardhat test test/AgoraToken.js

AgoraToken
Buying Tokens
✓ Buying tokens successfully (60ms)
✓ Buying tokens with 0 eth (51ms)
Selling Tokens
✓ Selling tokens successfully
✓ Try to sell more than available
✓ Try to sell 0
Working with pre-signed transactions
✓ Transacting tokens successfully via pre-signed transaction (41ms)
✓ Expired
✓ Used nonce (45ms)
✓ Amount should be > 0
✓ No enough tokens (57ms)
✓ From=addres(0)
✓ To=addres(0)
✓ Bad arguments for signature (47ms)
✓ Bad signature

14 passing (3s)
```

Изображение 3.1 - Тестовете на AgoraToken

За изпълнението на тестове използваме командата “npx hardhat test” ако искаме да изпълним всички. Добавяме името на конкретен файл за определени тестове. На изображение 3.1 са показани тестовете от AgoraToken.js, които са за договора, отговарящ за токените.

93 passing (22s)					
File	%Stmts	%Branch	%Funcs	%Lines	Uncovered Lines
<code>contracts/</code>					
<code>    AgoraToken.sol</code>	100	100	100	100	
<code>    Marketplace.sol</code>	100	100	100	100	
<code>    SimpleAuction.sol</code>	100	100	100	100	
<code>    SimpleSeller.sol</code>	100	100	100	100	
<b>All files</b>	<b>100</b>	<b>100</b>	<b>100</b>	<b>100</b>	

### 3.2 - Покритието на всички тестове за договорите

За намиране на покритието на тестовете използваме командата “npx hardhat coverage”, която изпълнява всички тестове в папката tests и показва какъв процент от договорите са проверени.

### 3.4 Скриптове за качване и верификация на умните договори

```
publicKey = await ethers.utils.computePublicKey(  
    process.env.ACCOUNT_PRIVATE_KEY);  
publicKey = hexToArray(publicKey);  
const marketplace = await Marketplace.deploy(publicKey);  
const agoraToken = await AgoraToken.deploy();  
const simpleSeller = await SimpleSeller.deploy();  
const simpleAuction = await SimpleAuction.deploy();  
console.log("deploying...")  
await marketplace.deployed();  
console.log("deployed marketplace")  
await agoraToken.deployed();  
console.log("deployed agoraToken")  
await simpleSeller.deployed();  
console.log("deployed simpleSeller")  
await simpleAuction.deployed();  
console.log("deployed simpleAuction")
```

Това е част от deployment скрипта, с който внедряваме договорите в избраната мрежа. Генерираме публичния ключ за акаунта, който ползваме. След това го подаваме като аргумент в конструктора на Marketplace. Така ще можем да видим тайните данни на всеки качен продукт. След това изчакваме всичките ни договори да се качат за да можем да изпълним по-нататъчни операции с тях.

```

await marketplace.setToken(agoraToken.address);
await marketplace.addContract(simpleSeller.address,"Simple Seller");
await marketplace.addContract(simpleAuction.address,"Simple Auction");
const owner = marketplace.owner();
await marketplace.addAdmin(owner);
await marketplace.addAdmin(process.env.ACCOUNT_ADDRESS);

await simpleSeller.joinMarketplace(marketplace.address);
await simpleAuction.joinMarketplace(marketplace.address);
console.log(
  `{
    "marketplace": "${marketplace.address}",
    "agoraToken": "${agoraToken.address}",
    "simpleSeller": "${simpleSeller.address}",
    "simpleAuction": "${simpleAuction.address}"
  }
`);

);

```

Във втората част от деплоимента посочваме връзките между договорите за продукти, токени и този за marketplace. Също така добавяме админ. Накрая принтираме адресите на договорите в JSON формат.

### 3.5 Интерфейс

Децентрализираните ни приложения могат да “живеят” върху блокчайн мрежата напълно отделени от всяка възможна потребителска интеракция. Могат да бъдат повиквани от други програми, други умни договори и, като цяло, от всякакви типове програмни интерфейси, разработени от програмисти. Това обаче е само част от изискванията към дипломната работа, затова се разработва потребителски графичен интерфейс, който

представлява уеб приложение, написано на React.js. Целта му е да предоставя бърза и интуитивна връзка с договорите, докато в същото време е оптимизиран за конкретната сфера. За разлика от стандартните web2.0 приложение, тук трябва да се имплементира различен UX, който да наподобява този, с който са запознати потребителите.

Една от основните разлики между изграждането на web2.0 и web3.0 UX е комуникирането и забавените заявки към блокчайна. Друга разлика е липсата на стандартни регистрации за потребителите. Те биват заместени от частните и публичните ключове, които представляват потребителската идентичност. Използваме ги чрез инструменти като Metamask. Третата основна разлика е неотменимостта на информацията, която се подава от нашите dApps.

Входната точка в кода на нашия интерфейс е файла index.js в папката src. Кода за рендериране на нашия основен елемент - App:

```
root.render(  
  <React.StrictMode>  
    <BrowserRouter>  
      <App />  
    </BrowserRouter>  
  </React.StrictMode>  
)
```

В него импортираме бъдещите компоненти. BrowserRouter е част от библиотеката react-router-dom, която отговаря за всички route-ове в уеб приложението. Това ни позволява да пренасочваме потребителя по време на неговата работа. Също разделя логически различните функционалности в сайта чрез избирамо рендериране на определени компоненти в зависимост от достъпения route.

```

{errorMessage}
{ account && provider
?
<Routes>
<Route path="/" element={<Base provider={provider} account={account} />}>

<Route index element={<HomePage update/>} />

<Route path="s/*" element={<SimpleSellerRoutes provider={provider} signer={signer}/>} />

<Route path="a/*" element={<SimpleAuctionRoutes provider={provider} signer={signer}/>} />

<Route path="t/" element={<BuyTokensPage provider={provider} signer={signer}/>} />

<Route path="admin/" element={<AdminPage provider={provider} signer={signer}/>} />

<Route path="*" element={<h1>404 not found</h1>} />

</Route>
</Routes>
:<ConnectButton onClick={connectWalletHandler}>
}

```

Основната структура на App.js е изградена от кондиционна проверка за рендериране на бутон или рутер, отговарящ за показването на конкретни компонент. За кондиционното рендерираме използваме тернарен if оператор, който проверява дали account и provider са null. Ако не са ще покажем крайния компонент, върнат от рутера. Ако 1 или 2 от тези 2 променливи е null ще покажем бутон, който вика функцията connectWalletHandler. Тази функция отваря web extension-a Metamask, който иска

потвърждение за достъп. Служи ни за подписване и изпращане на транзакции. В Routes изброяваме пътищата и слагаме основния, който разпределя към останалите. Index пътят отговаря за “/” и показва HomePage. Пътищата “s/” и “a/” ни изпращат до други рутери, които отговарят за продаването на и аукционите за продукти. Пътищата “t/” и “admin/” са за купуване на токени и администраторската страница. Пътят “\*” отговаря за всичко останало, което не е попаднало в изброените пътища. То е за ненамерени пътища - грешка 404.

С текущата структура ще имаме достъп до всички описани пътища при спазване на определено условие, в противен случай виждаме бутон за свързване с Metamask. На всяка страница рендерираме Base компонент, който съдържа в себе си NavBar и SideBar.

Компонента Base:

```
return (
  <>
  <MyNavbar provider={provider} account={account} />
  <div className='mt-4'>
    <Row className="mx-5">
      <Col md={2}>
        <MySidebar/>
      </Col>
      <Col>
        <Outlet/>
      </Col>
    </Row>
  </div>
</>
);
```

Outlet е страницата, която трябва да се зареди. Може да е който и да е елемент, подаден на Route.

При избирането на път с компонент рутер ще се изпълнява вложено извикване на пътища, което ще върне компонента на най-млатчия route. Например ако в основния Route отидем на “s/”, то ще се извика SimpleSellerRoutes. Това ще ни отведе до съответния код:

```

function SimpleSellerRoutes(props) {
  return(<>
    {props.provider
      ?
      <Routes>
        <Route index element={<SimpleSellerListPage provider={props.provider}
          signer={props.signer} />} />

        <Route path="/c" element={<SellerCreateProductPage
          provider={props.provider} signer={props.signer} />} />

        <Route path="/:id" element={<SellerDetailProduct provider={props.provider}
          signer={props.signer} />} />
      </Routes>
      :<Navigate to="/" />
    }
    </>);
}

```

Ако сме получили provider като аргумент за компонента ще преминем в route-а. При път “s/” ще влезем в index-а, който връща SimpleSellerPage - страница, която отговаря за показване на всички продукти за продажба. Ако влезем в “s/c” ще получим SimpleSellerCreateProduct за създаване на продукт. При третия route ще очакваме “s/:id”, където “:id” е аргумент, който ще се подава чрез адреса в браузъра. На това място е число, което ще играе ролята на индекс за търсене на съответния продукт. Страницата е за преглед на конкретен продукт.

Пътищата SimpleAuctionRoutes са аналогични.

```

{props.provider
?<Container className='mt-1'>
    <Button variant='secondary' onClick={updateProducts}>
        Update
    </Button>{' '}
    <Link to="/s/c">
        <Button variant='secondary'>
            Create product
        </Button>
    </Link>
<h1>Products for sale</h1>
<Row style={containerStyle}>
    {productCards}
</Row>
</Container>
:<Navigate to="/">
}

```

Това е структурата на SellerListPage - страницата, която отговаря за показването на всички продукти от контракта SimpleSeller. Първо прави проверка за наличие на provider, който играе ролята на интерфейс с блокчейна. Ако нямаме provider, ще се използва Navigate, който ще ни върне до началната страница. В противен случай ще покажем бутон за обновяването на продуктите, бутон за страницата за създаване на продукт и списък с всички продукти. Този списък е productCards и представлява array от компоненти.

```

const productCards = products.map((p) =>
    <Col key={p[1]} md={3} >
        <SellerProductCard className="mt-3" product={p[0]} id={p[1]} />
    </Col>
);

```

Products е списъка с всички продукти и индексите им, получени от смарт контракта. Изпълняваме тар за да създадем нов списък, състоящ се от компоненти SellerProductCard, като p[0] е самия елемент от products, а p[1] е индексът му. Тези променливи се подават като props на компонента.

```
function SellerProductCard(props) {  
    const p = props.product;  
    return (  
        <Card className="itemCard">  
            <Link to={`/s/${props.id}`}  
                style={{textDecoration: 'none', color: 'black'}}>  
                <Card.Img className="h-50" src={p.linkForMedia} />  
                <Card.Body>  
                    <Card.Title>{p.name}</Card.Title>  
                    <Card.Text>  
                        {Number(ethers.utils.formatUnits(  
                            p.price._hex, "ether")  
                            .toFixed(5).toString())} AGR  
                        <br/>  
                    </Card.Text>  
                    {p.approved  
                    ?<img style={{height:35}} src={checkedGIF}/>  
                     :"")}  
                </Card.Body>  
            </Link>  
        </Card>  
    );  
}
```

SellerProductCard представлява react-bootstrap карта, която описва подадения продукт. Взимаме информацията за него от props.product. В картата показваме изображение, име, цена и дали е потвърден. При кликване на картата ни препраща до страницата за купуване.

```

useEffect(()=>{
    simpleSeller.on("sellerProductAdded", updateProducts);
    updateProducts();
    return ()=>{
        simpleSeller.removeListener(
            "sellerProductAdded", updateProducts);
    }
},[]);

```

```

async function updateProducts (){
    let c =parseInt(await simpleSeller.productCount());

    let tmpProducts = [];
    for(let i=0;i<c;i++){
        let p = await simpleSeller.products(i);
        tmpProducts.push([p,i]);
    }
    setProducts(tmpProducts);
    setCount(c);
}

```

За вземане на информацията от договора използваме useEffect, за да може да се случи след render на страницата. Първо взимаме броя на продуктите, след това един по един ги извличаме и подаваме на state-a products.

В useEffect слушаме за създаване на нов продукт, при което да обновим списъка. В return спираме процеса, който слуша.

```

const simpleSeller = new ethers.Contract(
    addresses.simpleSeller,
    simpleSellerJSON.abi,
    props.provider
);
simpleSeller.on("sellerProductAdded", updateProducts);

```

Както в Hardhat, така и във front-end-а използваме ethers.js. В показания код го използваме за инициализиране на интерфейс със SimpleSeller. Правим нова инстанция на Contract от ethers, в която подаваме адреса на договора, ABI-я, който представлява json описание на методите на договора и provider, който е връзката ни с блокчайн мрежата, която сме избрали. На последния ред създаваме “слушател” за събитието sellerProductAdded, който ще извика updateProduct за обновяване на страницата при добавяне на нов продукт в договора.

```

return(<>
{product
?<Container className="mt-2">
<Row>
<Col md={4}>
<img className="w-100" src={product.linkForMedia}/>
</Col>
<Col md={8}>
<h2>{product.name}</h2>
<h3>Price:
{Number(ethers.utils.formatUnits(
product.price,"ether")).toFixed(5)} AGR
</h3>
<h4>Price in USD: ${priceInUSD.toFixed(2)}</h4>
<h6>(powered by <a href='https://www.coingecko.com/'> Coingecko </a>)</h6>
{product.approved
?<h3>Approved</h3>
:<></>
}
<h6>Seller: {product.seller}</h6>
<h6>Added on
{dateAdded.getFullYear() + "/" +
(dateAdded.getMonth() + 1) + "/" +
dateAdded.getDate()}</h6>
</Col>
</Row>
...

```

Показана е първата част от структурата на страницата SellerDetailProduct, която дава повече информация за избрания продукт. Това включва име, снимка, цената в AGR, цената в долари спрямо текущия курс на етериум според Coingecko, дали продуктът е потвърден, продавача и датата на качване.

```

<Row>
  {userAddress === product.seller
  ?<Form.Group className="mb-3" controlId="formName">
    <Form.Control
      as="textarea" defaultValue={description}
      onChange={e=>setDescription(e.target.value)} type="text"
      placeholder="Delivery instructions"/>
    <Button onClick={updateDescription}> Update description </Button>
  </Form.Group>
  :<h6>{description}</h6>
  }
</Row>

```

Показаният код отговаря за описанието на дадения продукт в страницата за подробно разглеждане. Ако текущият потребител е и продавач на продукта ще види текстово поле, попълнено с информацията, която се съдържа в базата данни. Също ще види бутона за обновяване на описанието. В противен случай ще види само описанието, но без полето за въвеждане.

```

const updateDescription = async () => {
    const message = ethers.utils.keccak256(
        ethers.utils.solidityPack(
            ['address','uint256','string'],
            [simpleSeller.address,id,description]
        ));
    try{
        const signature = await signer.signMessage(message)
        axios
            .put(`http://localhost:5000/s/d/${id}`, {description,signature})
            .catch(e){
                console.log(e)
                alert("not updated")
            }
        return
    }
}

```

При натискане на бутона за обновяване на описанието се вика функцията updateDescription, която изисква подпись от потребителя. Продавачът подписва съобщение, което съдържа адреса на договора, индекса на продукта и самото описание, което трябва да бъде поставено. Така сървърът може да потвърди източникът на заявката преди да промени полето в базата данни.

```

...
<Row>
  {!product.paid
  ?<>
    <Form.Group className="mb-3" controlId="formName">
      <Form.Label>Delivery Instructions:</Form.Label>
      <Form.Control
        onChange={e=>setDeliveryInstructions(e.target.value)}
        type="text"
        placeholder="Delivery instructions"
      />
    </Form.Group>
    <Button onClick={buyProduct}> Buy now </Button>
  </>
  :isCourier
  ?product.delivered==false
  ?<Button onClick={deliverProduct}>Deliver now </Button>
  :<h4>Already delivered</h4>
  :<></>
</Row>
</Container>
:<h1>Loading</h1>
</>
);
}

```

Втората част включва проверка за статуса на продукта. Ако не е платен, се показва форма за въвеждане на инструкции за доставка и бутон за плащане. Ако е платена, се прави проверка дали текущия потребител е куриер. Ако не е, няма да показваме повече елементи. Ако е, правим проверка дали продукта е доставен. В зависимост от това ще покажем бутон за отбелязване на продукта като доставен или ще изпишем, че вече е. Края на най-големия if, който започва с това дали продуктът е null ще завърши с Loading на страницата, ако все още не е получен отговор от контракта.

```

const weiToUsd = (wei,r) => {
    let usd = 1/r;
    const priceInEth = ethers.utils.formatEther(wei);
    return priceInEth/usd;
}

```

За намирането на цената в долари използваме функцията `weiToUsd`, която взима като аргументи цената в Wei и цената на 1 етериум. `Ethers.utils.formatEther` намира количеството етъри за определен брой wei.

```

const getRates = async () => {
    try{
        const response = await fetch(
            'https://api.coingecko.com/api/v3/simple/price?ids=ethereum&vs_currencies=usd',
            {
                method: 'GET',
            }
        );
        let r = (await response.json()).ethereum.usd;
        setRate(r);
        return r;
    }catch(e){
        console.log(e);
    }
}

```

Извличаме цената на етъра чрез API-я на Coingecko.

```

const buyProduct = async () => {
    let nonce, expiration, sig;
    try{
        const sigData = await makeSignature();
        nonce=await sigData['nonce'];
        expiration=await sigData['expiration'];
        sig=await sigData['signature'];
    }catch(e){
        console.log(e.message);
    }
}
...

```

Във функцията `buyProduct` се извършва извикването на метода `payProduct` на договора `SimpleSeller`. В първата част на функцията взимаме пригответните данни за подписа и самия подпись от `makeSignature`.

```

let deliveryData = await ethers.utils.solidityPack(
    ['string'],[deliveryInstructions]);
deliveryData = await ethers.utils.arrayify(
    await ethers.utils.keccak256(deliveryData));
try{
    await simpleSeller.payProduct(
        id,deliveryData,expiration,await signer.getAddress(),sig);
}catch(e){
    console.log(e);
}

```

Във втората част на `buyProduct` пригответваме `deliveryInstruction` за подаване в метода на договора. Накрая подаваме всички аргументи за купуването на продукт чрез `simpleSeller`.

Методите в страницата за наддаване при `auctionDetailProduct` са аналогични.

В auctionDetailProduct страницата трябва да покажем същите неща, които имахме в selletDetailProduct, но добавяме и крайната дата на аукциона, както и поле за въвеждане на сума за наддаване. Числото, което приемаме е в долари и се конвертира в Wei.

```
try{
    const response = await fetch('http://localhost:5000/a/b/', {
        method: 'POST',
        body:JSON.stringify({
            "instanceId":id,
            "bidder":await signer.getAddress(),
            "amount":myBid._hex,
            "deliveryInstructions":deliveryData,
            "signature":sig
        }),
        headers: {
            'Content-Type': 'application/json'
        },
    });
    console.log(await response.status);
}catch(e){
    console.log(e);
}
```

Подписването на самата транзакция за наддаване е аналогично, разликата е, че потребителя има избора да изпрати подписаната транзакция до базата данни вместо до договора директно. Така се избягват такси за осъществяване на транзакции, когато само едно от наддаванията е нужно на договора за да изпълни съответните операции. Пътят “a/b/” е за auction bids.

```

const usdToWei = (usd,r) => {
  let eth = usd/r;
  eth = parseFloat(eth).toFixed( 18 );
  return ethers.utils.parseEther(eth.toString());
}

```

Това е функцията за конвертиране от подадените долари към Wei. Пак използваме цената на етъра, получена от Coingecko.

```

return(
<Container>
  <h1>Create seller product</h1>
  <Modal show={show} onHide={()=>setShow(false)}>
    ...
  </Modal>
  <Form>
    ...
  </Form>
</Container>
);

```

Страницата SellerCreateProduct е изградена от два основни компонента. Полетата за информация относно продукта, който искаме да качим и Modal прозорец, който се отваря след въвеждането на информацията. Целта на този прозорец е да покаже изображението, подадено от потребителя и да обновява в реално време цената на етъра. Тъй като валутата е силно волатилна, трябва да се даде обратна връзка на потребителя за потенциална промяна в цената.

```

const response = await axios({
  method: "post",
  url: "http://localhost:5000/i",
  data: formData,
  headers: { "Content-Type": "multipart/form-data" },
});

setLinkForMedia(
  "http://localhost:5000"+response.data.pathToImage);

```

Добавяме снимката с axios заявка до backend-а и взимаме линка към нея за да я покажем в Modal прозореца.

```

async function encryptWithPublicKey(message,publicKey){
  let pk = new Uint8Array(publicKey)
  let data = await EthCrypto.encryptWithPublicKey(
    pk,message);
  data = JSON.stringify(data)
  data = stringToHex(data);
  return data;
}

```

За да криптираме тайната информация от продавача към админа на договорите при създаването на продукт използваме библиотеката EthCrypto, която ни дава достъп до функция, криптираща с публичен ключ. Това криптирано съобщение се подава на метода addProduct на SimpleSeller.

Употребата на функцията в SimpleAuction е аналогично.

```
"scripts": {  
  "start": "react-app-rewired start",  
  "build": "react-app-rewired build",  
  "test": "react-app-rewired test",  
  "eject": "react-scripts eject"  
},
```

React.js не изпраща някои библиотеки до клиента с цел да олекоти комуникацията и действието на приложението. Изключва модули като stream, fs и crypto. Проблемът е, че EthCrypto, модулът, който ползваме за криптиране с публичен ключ, трябва да бъде изпратен до и ползван от клиента, но той използва crypto и stream, които не са налични. Затова променяме конфигурацията на приложението, така че да се работи по желания начин. Първо променяме скриптовете в package.json.

```
resolve: {  
  ...config.resolve,  
  fallback: {  
    "stream": require.resolve('readable-stream'),  
    "crypto": require.resolve('crypto-browserify'),  
  },  
},
```

След това създаваме файл config-overide.js, който съдържа допълнителни конфигурации, сред които са и новите библиотеки. Така заместваме stream с олекотената му версия readable-stream и crypto с crypto-browserify. Това е т.нар. Polyfill, който ни позволява да изпълняваме определени функционалности от библиотеки върху клиента.

### 3.5 Сървърна част

С всяко качване на информация в умните договори плащаме такса за осъществяване на качването, за изпълнение на кода, който отговаря за него и за количеството информация, която качваме. Това значимо намалява данните, които потребителите могат да качват, както и броя на транзакциите, които могат да правят. Това затруднява създаването на продукти, индексирането на продукти спрямо интересите на хората и други действия, като наддаване в аукциони. Затова ще се възползваме от предимствата на Web2.0, сред които са ниската цена за складиране на данни, бързото и евтино изпълнение на код, възможността да правим промени по логиката и контрола върху операциите. Всичко това може да се постигне чрез създаването на централизиран back-end, който да изпълнява определени операции. Този сървър ще е отделен от потребителския интерфейс, но ще могат да си комуникират през API.

Сървърът ни трябва да може слуша за събития от блокчайна и да записва промените в продуктите. Да съхранява цялата информация за всеки продукт, както и допълнителни данни, използвани за бъдещо индексиране, например описания, тагове и др. Трябва да може да запазва неосъществени, но подписани наддавания за аукциони, както и да насрочва процес за изпълнението на най-голямото наддаване.

Технологията, която ще използваме е Node.js, защото работи с JavaScript, който е избраният език за проекта. Интеракциите с децентрализираните приложения се случват отново с ethers.js, както е и във front-end-а и тестовете.

Входната точка при стартиране на back-end-а е файлът index.js, в който стартираме уеб сървъра и слушателите за блокчайн събития. За създаване на API - програмен интерфейс, който да служи за подаване на информация до front-end-а, използваме библиотеката express, която ни позволява да създаваме сигурни предефинирани пътища за програмния интерфейс.

```
const app = express();
const PORT = 5000;

app.use(bodyParser.urlencoded());
app.use(bodyParser.json());
```

Показана е началната конфигурация на приложението. Портът, от който ще се достъпа, е 5000. Използваме bodyParser за обработка на уеб заявките.

```
app.get("/", (req, res) => res.send("Welcome to the API!"));
app.use("/s", sellerProductRoutes);
app.use("/a", auctionRoutes);
app.use("/i", imageRoutes);

app.all("*", (req, res) => res.send("404"));
app.listen(PORT, async () => console.log(
  `Server running on port: http://localhost:${PORT}`));
```

След дефиниране на пътищата стартираме сървъра и очакваме заявки. Кореновия път на продуктите за продажба е “/s”, докато този за аукциони е “/a” от auction. Пътят “/i” е за качване на снимки, а “\*” е за всичко, която не е било разпознато.

```

simpleSeller.on("sellerProductAdded", createSellerProduct);
simpleSeller.on("sellerProductSold", sellSellerProduct);
simpleSeller.on("sellerProductDelivered", deliverSellerProduct);
simpleSeller.on("sellerProductApproved", approveSellerProduct);

simpleAuction.on("auctionProductAdded", createAuctionProduct);
simpleAuction.on("auctionProductBid", bidAuctionProduct);
simpleAuction.on("auctionProductDelivered", deliverAuctionProduct);
simpleAuction.on("auctionProductApproved", approveAuctionProduct);

console.log("set up listeners");

```

След стартиране на уеб сървъра започваме да слушаме за събитие в договорите. Целта на това е да обновяваме информацията в базата данни при промени в състоянието на продуктите. Така избягваме нуждата от правенето на регулярни заявки до контрактите.

```

const router = express.Router();

router.get('/p', getProducts);
router.get('/p/:id', getProduct);
router.post('/p/:id', instantiateOrUpdateProduct);

router.get('/b/:id', getBidsForProduct);
router.post('/b', bidForProduct);

router.put('/d/:id', setDescription)

```

Показани са пътищата за аукционите. За продуктите ни трябва взимане на всички, взимане по id, обновяване или инстанциране по id. За наддаванията имаме създаване и взимане по индекс на продукт. В случая на продуктите нарочно не е направено създаване по подадени аргументи, защото всички записи в базата данни трябва да идват от паметта

на смарт контрактите. Пътят за поставяне на описание на продукт е “/d/:id” и взима индекс от URL-а и описание през тялото на заявката.

Вместо да пишем SQL заявки ще използваме ORM, който да служи като интерфейс между скриптовете ни и базата данни. За целта ползваме sequelize, който инсталираме глобално с команда “sudo npm install -g sequelize-cli”.

Инициализираме със “sequelize init”, след това създаваме MySQL база данни с CREATE TABLE. За успешна връзка с базата трябва да инсталираме mysql2 чрез “npm install --save mysql2”. В конфигурационния файл config.js записваме нужните данни за връзка с базата.

```
const dotenv = require('dotenv');
dotenv.config();

module.exports = {
  "development": {
    "username": "root",
    "password": process.env.SQL_PASS,
    "database": "web3marketplace_db",
    "host": "127.0.0.1",
    "dialect": "mysql"
  },
}
```

“Sequelize model:generate” е за създаване на таблица в базата данни. Основните модели са за продукт за продаване, продукт за аукцион и наддаване за продукт.

Качването на снимки и създаването на продукти са два отделни процеса за сървъра. Във front-end-а първо изпращаме снимката на продукта, след това получаваме линк към нея и чак тогава извикваме метода addProduct за създаване на продукт в децентрализираното приложение.

```

app.use((req,res,next)=>{
  res.header('Access-Control-Allow-Origin','http://localhost:3000');
  res.header(
    'Access-Control-Allow-Headers',
    'Origin, X-Requested-With, Content-Type, Accept, Authorization');
  if(req.method === "OPTIONS"){
    res.header(
      'Access-Control-Allow-Methods',
      'PUT, PATCH, DELETE, GET, POST');
    return res.status(200).json({})
  }
  next()
})

```

Браузърът изпраща предварителна заявка до сървъра за проверка дали съответния софтуер приема определен тип информация. В началото на файла index.js дефинираме функция, която се активира преди обработката на всяка заявка и цели да даде стойност 200 на статус кода. Това се прави за да може сървърът да сигнализира на браузъра за приемането на POST request-и.

```

var storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, 'uploads');
  },
  filename: function (req, file, cb) {
    cb(null, Date.now() + '-' + file.originalname);
  }
});
var upload = multer({ storage: storage });

router.post('/',
  upload.single('image'),uploadImageView);
router.use('/', express.static('uploads'))

```

За запазване на изображението ползваме библиотеката multer. Подаваме “uploads” като име на папка за запазване на изображенията. Името на всяко изображение се генерира от Date.now() и оригиналното име. Пътищата на API-то са за създаване и достъпване на изображенията.

```
const getBidsForProduct = async (req, res) => {
    id = parseInt(req.params.id);
    if(id === null || isNaN(id)){
        console.log("GET /a/b/:id: id is null or NaN");
        res.status(400);
        res.send({ "message": "InstanceId=null" });
        return;
    }
    const bids = await AuctionBid.findAll(
        { where: { instanceId: id } });
    res.send(bids);
    console.log(`getBids ${id}`);
}
```

Това е функцията, която се изпълнява при “a/b/:id” на API-я ни. Връща всички подписани и запазени наддавания в базата данни за продукт с индекс id. При невалидно id връща грешка.

Най-високото наддаване за всеки аукцион се изпълнява автоматично определено време преди края на аукциона. Общата сума от такси за изпълняване на наддавания драстично намалява.

```

async function createAuctionProduct(name,minimalPrice,seller,index){
    index = parseInt(index);
    try{
        const bci = await simpleAuction.products(index);
        let marketHashOfData=hexToBytes(bci.marketHashOfData);
        await AuctionProduct.create({
            instanceId:index,
            name,
            minimalPrice: minimalPrice._hex,
            addDate: new Date(),
            seller,
            finishDate: new Date(bci.finishDate*1000),
            linkForMedia:bci.linkForMedia,
            description:"",
            marketHashOfData: marketHashOfData
        });
    }catch(e){
        console.log(e);
        return;
    }
    scheduleBidExecution(index)
    console.log(`Created auction for ${name} index ${index}`)
}

```

Функцията, подадена при създаване на listener за събитието auctionProductAdded, приема аргументите, които самото събитие има в дефиницията си. В случая това са име, минимална цена, продавач и индекс. Променливата bci, blockchain instance, запазва продукта от умния договор. След това създаваме запис в базата данни. В края на функцията насрочваме процес чрез scheduleBidExecution с индекса на аукциона.

```

const executeTime = new Date(
  (bci.finishDate-60*60)*1000)
var j = schedule.scheduleJob(
  executeTime,
  ()=> makeBidsForProduct(index));

```

В scheduleBidExecution първо правим проверка за валидността на индекса, след това за наличието на такъв продукт в договора. Накрая изпълняваме самото насрочване на процеса за наддаване.

```

console.log(`In cron job with index ${index}`)
try{
  bids = await AuctionBid.findAll({
    order: [['amount', 'DESC']],
    where:{instanceId:index}
  });
}catch(e){
  console.log("Schedule job: err on finding bids");
  return;
}

```

Във функцията makeBidsForProduct правим проверки за входните данни и взимаме всички наддавания за съответния продукт, подредени в низходящ ред спрямо стойността им.

```

try{
    for(let i=0;i<bids.length;i++){
        if(
            (await agoraToken.balanceOf(
                bids[i].bidder)).gt(bids[i].amount) &&
            !(bci.bidAmount.gt(bids[i].amount))
        ){
            await simpleAuction.bidForProduct(
                index,
                bids[i].deliveryInstructions,
                bids[i].amount,
                bids[i].bidder,
                bids[i].signature
            );
            console.log(
                `Schedule jobs: successfully bid for product ${index}`)
            flag=1;
            break
        }
    }
}catch(e){
    console.log(e)
    console.log("Schedule job: can't make bid");
}

```

След това започваме цикъл докато не изпълним най-голямата валидна транзакция. Ако първата не е валидна или потребителя няма съответното количество токени за наддаване продължаваме към следващата. В момента, в който стигнем до валидна, отбелязваме във флаг променлива и прекъсваме цикъла. Флагът ни е нужен за да можем след това да запишем правилен лог.

```

const message = ethers.utils.keccak256(
    ethers.utils.solidityPack(
        ['address','uint','string'],
        [simpleAuction.address,id,req.body.description]
    )
)
...
const signerAddr = ethers.utils.verifyMessage(
    message, req.body.signature);
if(signerAddr !== bci.seller) {
    console.log("PUT /a/d/:id")
    console.log("Incorrect signature")
    res.status(401)
    res.send({"message":"Unauthorized"})
}

```

Показана е част от функцията за обновяване на описанието на определен продукт. След проверки за подадената информация пресъздаваме подписаното и изпратено съобщение. Използваме функцията verifyMessage, която взима предполагаемото изходно съобщение и подписа. Връща предполагаемия адрес на подписващия. Ако източникът на подписа и продавача на продукта съвпадат всичко е наред и промяната се записва в базата данни.

Последната част от изискванията засяга отчетността ни пред потребителите. Това включва качването на кода на договорите в сайт за преглед на съдържанието на блокчайн мрежата. Такъв сайт е etherscan.com, в който може да се види всяка една осъществена транзакция в Етериум мрежата. Всеки трансфер на етири, създаване на договори и извикване на методи от договори може да бъде намерен.

The screenshot shows the Goerli Testnet Explorer interface on Etherscan.io. At the top, there's a navigation bar with links for Home, Blockchain, Tokens, Misc, and a Goerli button. Below the header is a search bar with placeholder text "Search by Address / Txn Hash / Block / Token / Ens". To the right of the search bar is a button labeled "Start Today" inside a box labeled "Advertise your brand here!". The main content area is divided into two sections: "Latest Blocks" on the left and "Latest Transactions" on the right. The "Latest Blocks" section lists five blocks with details like block number, timestamp, recipient, transaction count, and gas fees. The "Latest Transactions" section lists five transactions with details like hash, timestamp, sender, recipient, value, and gas fees. At the bottom of each section are buttons to "View all blocks" or "View all transactions".

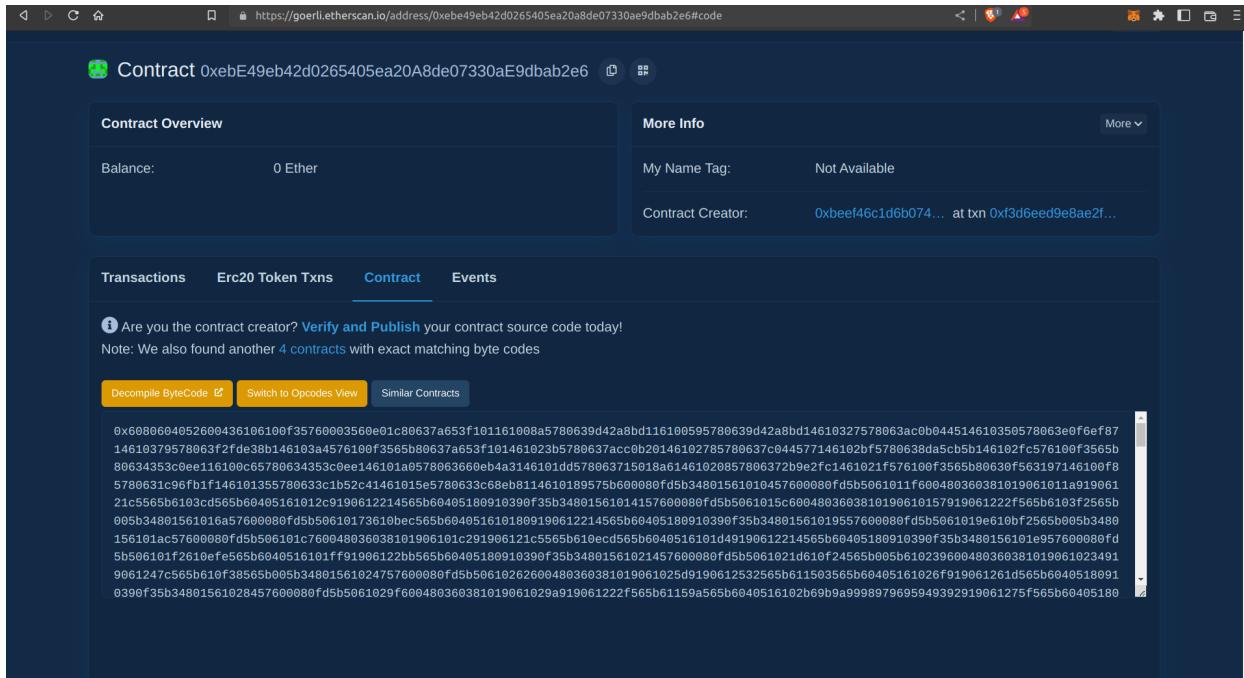
Изображение 3.3 - Началната страница на Etherscan

В searchbar-а могат да бъдат потърсени адресите на участниците в транзакцията, адресите на договорите, хеша на транзакцията, номера на блока, адрес на договор за токени и ENS (Ethereum Naming Service).

The screenshot shows a detailed transaction history for a specific Ethereum address on Etherscan. The page title is "https://goerli.etherscan.io/address/0xBEF46c1d6B074a597b436528cdfEE94ebF6EF54". The table displays 12 transactions, each with columns for Txn Hash, Method (with a dropdown arrow), Block, Age, From, To, Value, and Txn Fee. The transactions involve various Ethereum contracts and methods such as "Join Marketplace", "Add Admin", "Add Contract", and "Set Token". The "From" column shows the address 0xbeef46c1d6b074..., and the "To" column shows various contract addresses like 0xe3dbc37a4318e... and 0x5458235389799.... The "Value" column shows amounts like 0 Ether and 0.00000075 Ether, and the "Txn Fee" column shows values like 0.00000075 and 0.00005319.

Изображение 3.4 - Страница за транзакции в Etherscan

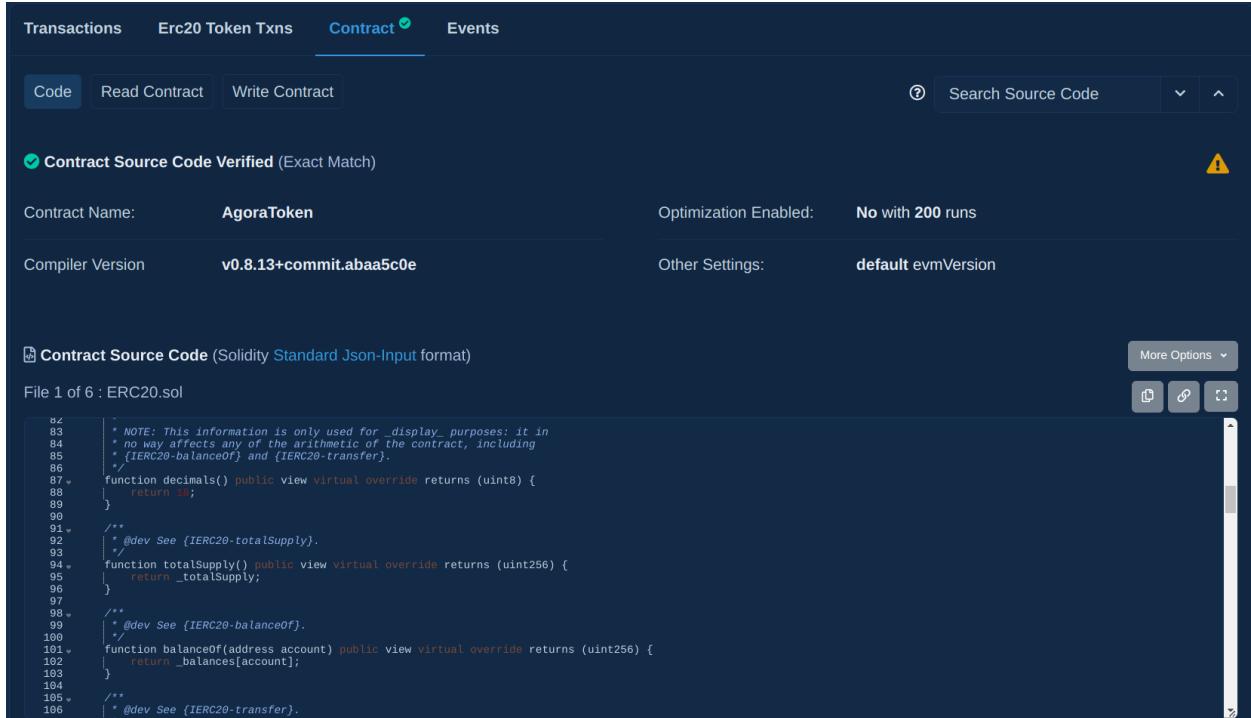
Подаваме адресът на разработческия ни акаунт и виждаме всички изпълнени операции. Сред тях са създаване на договори, отбелязани с Contract creation, и викането им.



Изображение 3.5 - Страница за разглеждане на договор в Etherscan

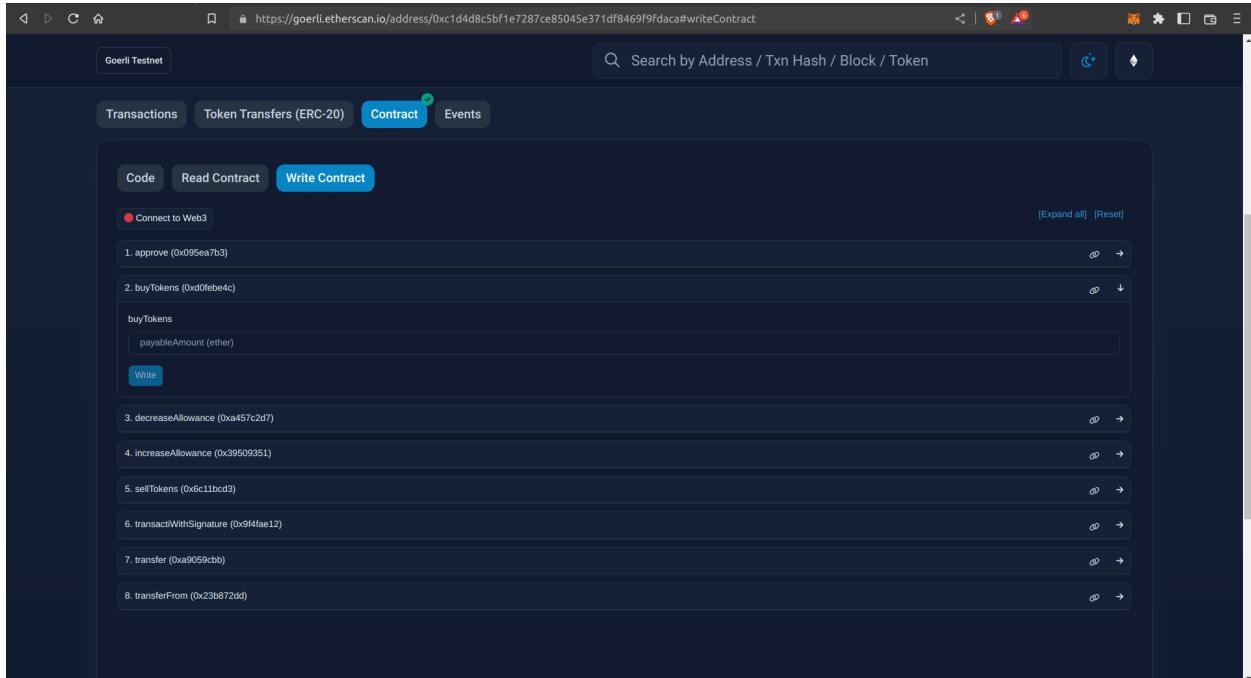
Когато отворим страницата на случаен договор ще видим неговия байткод в менюто Contract. Това е изпълнимия код, който е внедрен в мрежата. Проблемът с това е, че за обикновен човек това не дава никаква информация за операцията и функциите на договора. Затова има разработени механизми за верификация на кода, който да може да бъде достъпен за обикновения потребител.

Чрез команда “`npx hardhat verify 0xc1d4d8c5bf1e7287ce85045e371df8469f9fdaca --network goerli`” можем да верифицираме договор с посочения адрес в etherscan и мрежата goerli. При изпълнение hardhat търси наш договор, който да има същия байткод и изпраща четимия код за верификация.



Изображение 3.6 - Страница за договор, индикираща успешно верифициране в Etherscan

При успешна верификация виждаме зелена отметка и четимия код на файловете, които са включени в изпълнимия файл накрая.



Изображение 3.7 - Страница за договор в Etherscan, показваща методите

Освен Solidity кода, Etherscan ни показва и списък от методите на договора ни.

При договори, които приемат аргументи в конструктора си трябва да подадем допълнително аргументите в командата за верификация. В случая на Marketplace това е невъзможно, защото типът на аргумента е bytes, който не може да бъде въведен в терминала. Затова Създаваме нов .js файл, който да експортва аргумента.

```
publicKey = ethers.utils.computePublicKey(  
    process.env.ACCOUNT_PRIVATE_KEY);  
publicKey = hexToArray(publicKey);  
module.exports = [  
    publicKey  
];
```

И използваме следната команда “npx hardhat verify  
0xe23b3b13fc68ff5547eb358ebe84a7f523cf2cba –network goerli --constructor-args  
marketplaceArgs.js”

# Глава 4

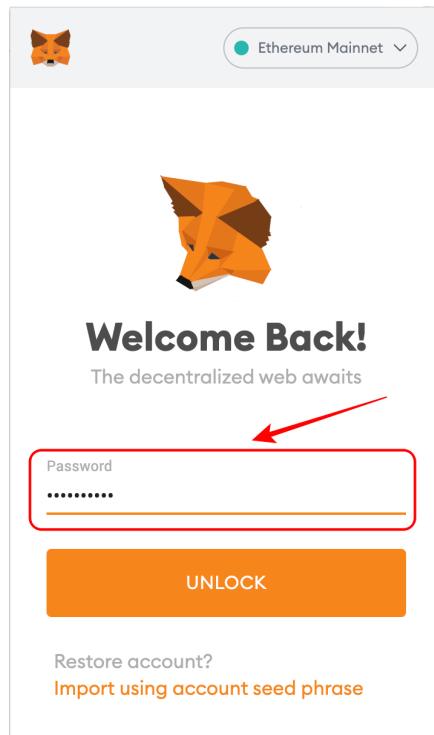
## 4.1 Ръководство за потребителя

Приемаме, че потребителят на уебсайта е запознат с Web3.0 и има свален Metamask extension за браузъра си, както и че се е сдобил с етъри. След изпълнението на тези стъпки трябва да отвори браузър с extension-а и да потърси уебсайта.



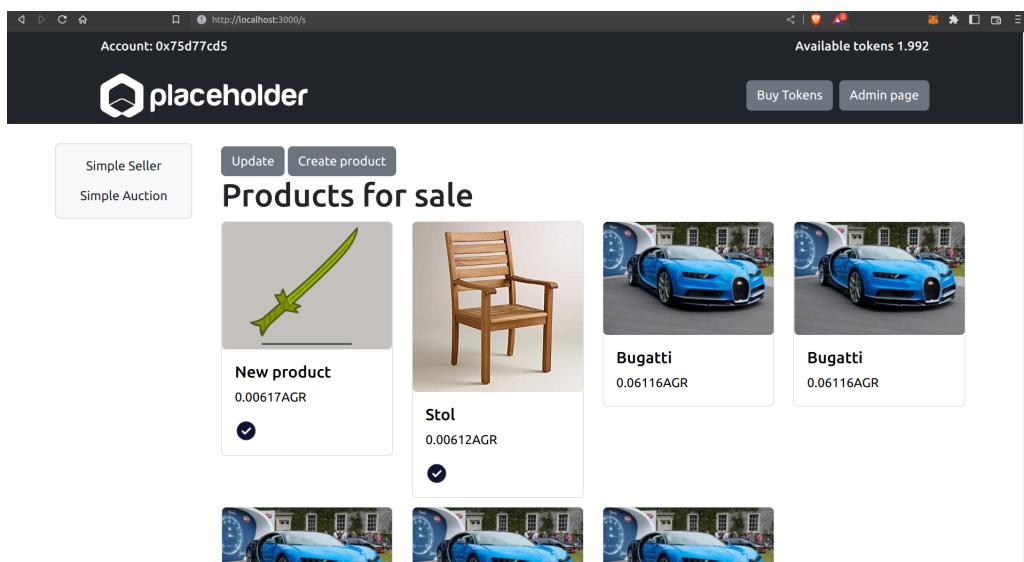
Изображение 4.1 - Първа страница на уебсайта

Първата страница се състои от един бутона за свързване на уеб портфейла Metamask, при което трябва да въведе паролата си за да се аутентицира.



Изображение 4.2 - Прозорец на Metamask за въвеждане на парола

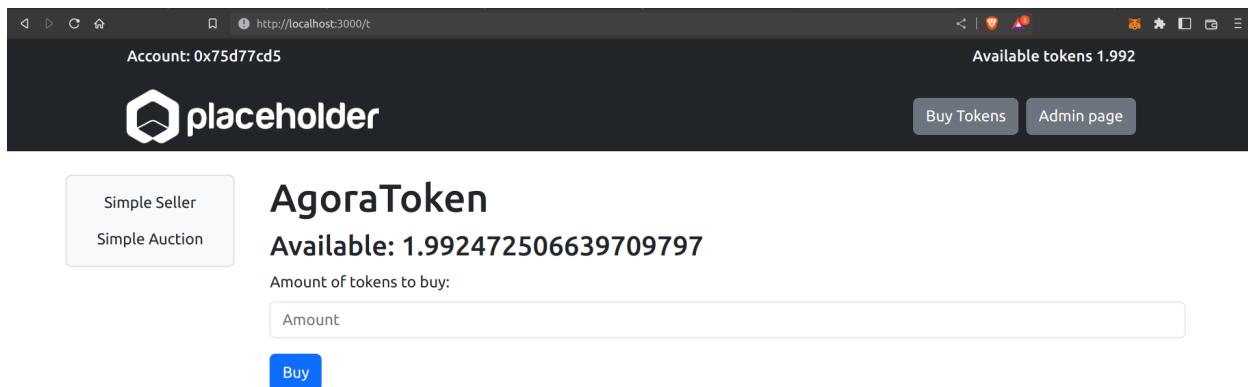
След въвеждането на паролата ще види началната страница, след която ще може да влезе в каталогите на продуктите. Навигационното табло служи за предвиждане в страници, които нямат връзка с пазарните договори. Страницното табло ни пренасочва към страници с продукти.



Изображение 4.3 - Страницата с продукти за продажба

Навигационното табло е разделено на две части - горна и долната. В горната част потребителя вижда първите букви от адреса си и наличните токени - единицата, с която ще може да закупува продукти. В долната част има бутони за насочване към страницата за закупуване на повече токени и, ако е администратор, страницата за администратори.

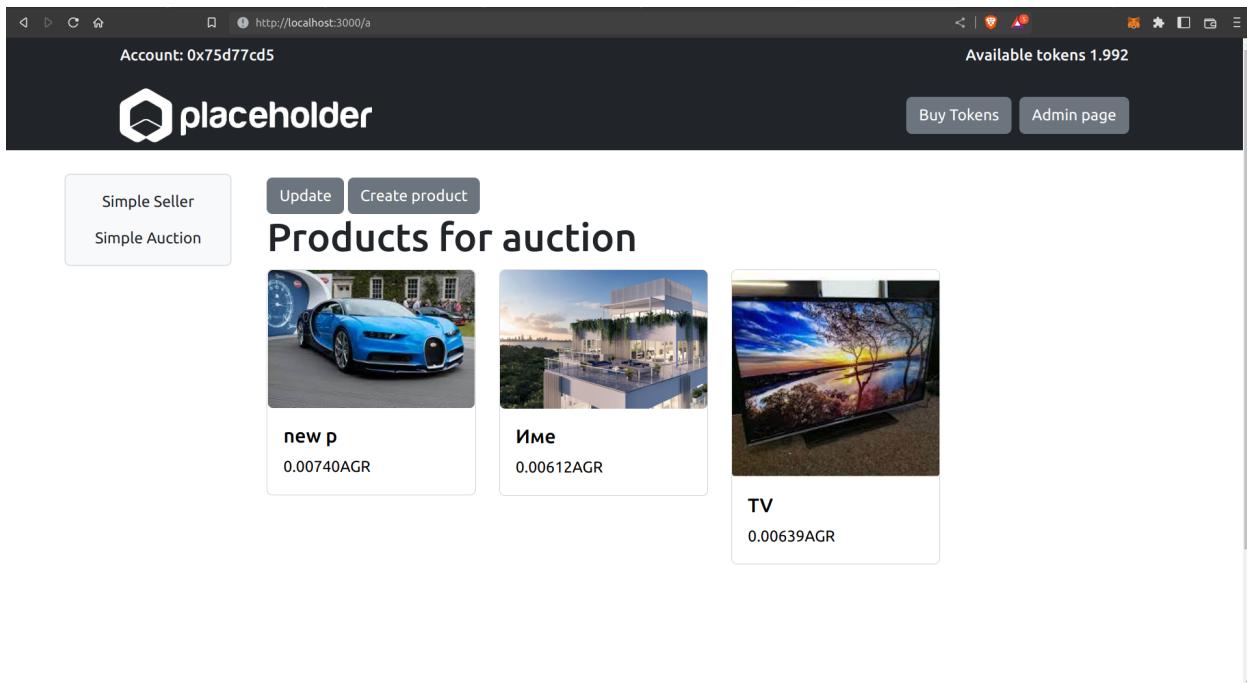
Преди да може да закупи продукт трябва да се насочи към страницата за купуване на токени и да въведе желаното количество.



Изображение 4.4 - Страница за закупуване на ERC-20 монети

След това ще види съобщение от Metamask за потвърждаване на плащането.

В каталога на продуктите за продажба, освен продуктите и обща информация за тях, има два бутона - за обновяване на информацията и за създаване на продукт.



Изображение 4.5 - страница с продукти в аукциони

Страницата за аукциони е аналогична.

Account: 0x75d77cd5 Available tokens 1.992

**placeholder**

Simple Seller Simple Auction

### Bugatti

**Price: 0.06116AGR**

**Price in USD: \$95.78**

(powered by [Coingecko](#))

Seller: 0x75D77cD57c943B24DB0d3Faf55d40502954Ccc0C

Added on 2023/2/28

*Lorem Ipsum* is simply dummy text of the printing and typesetting industry. *Lorem Ipsum* has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book.

[Update description](#)

Delivery Instructions:

Delivery instructions

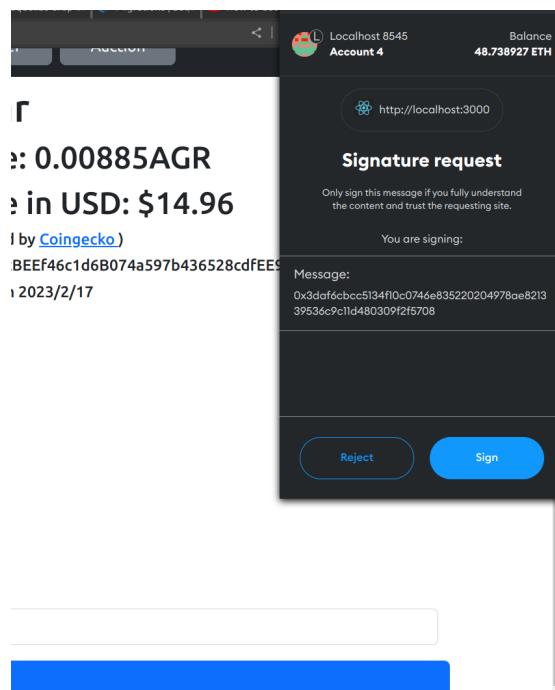
[Buy now](#)

[Approve product](#)

Изображение 4.6 - страница за купуване на продукт

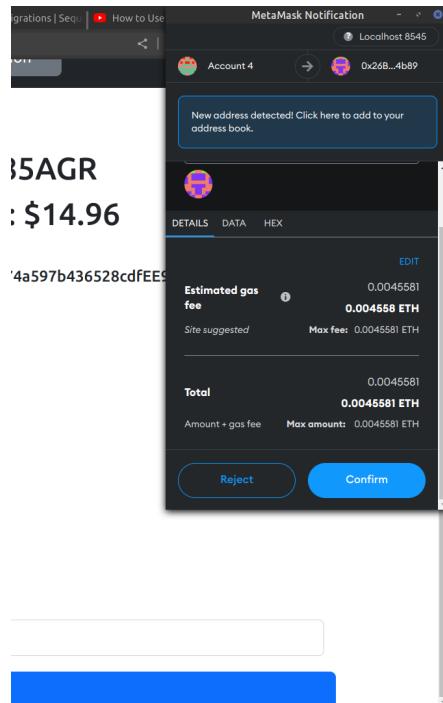
Чрез кликване на изображението на продукта можем да стигнем до страницата му. Освен снимката и информацията за артикула има и поле за въвеждане на инструкции за доставка, бутона за плащане и поле за въвеждане на описание за продавача на продукта. Ако потребителя не е продавач ще види само описанието без полето за въвеждане.

При промяна на описанието потребителя ще види заявка от Metamask за подписване на съгласие. Така сървърът, който запазва описанието валидира самоличността на източника на API извикването.



Изображение 4.7 - Metamask прозорец за подписване на транзакция

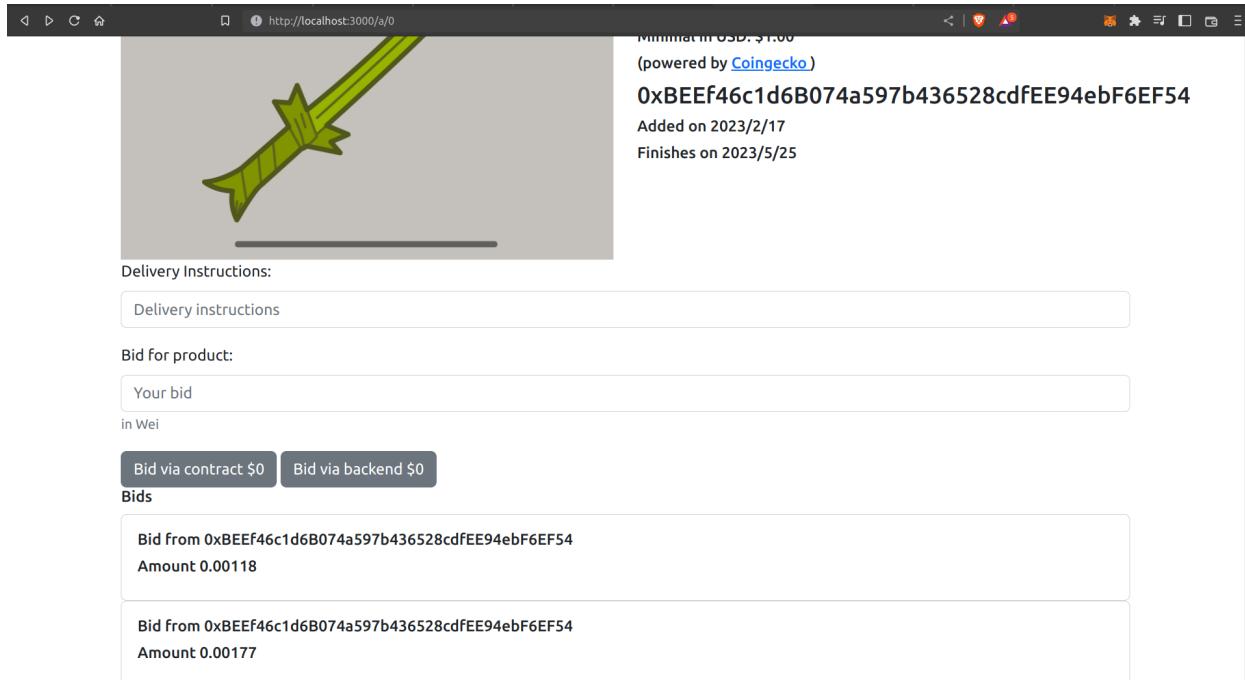
При купуване трябва първо да подпишем транзакцията за самото плащане с токени. Това се случва чрез Metamask, който използва частния ключ на потребителя за да подпише съобщението.



Изображение 4.8 - Metamask прозорец за изпращане на транзакция

След това се иска второ потвърждение за извикване на метода от умния договор. С всяко действие потребителя плаща такса, която е посочена в прозореца.

Когато извършва наддаване в аукцион стъпките са същите, но в страницата за аукциона има избора дали да наддаде чрез умния договор или чрез сървъра. Когато се осъществява наддаване чрез интерфейса директно към децентрализираното приложение потребителя плаща таксата за транзакцията. По този начин дава две потвърждения - едно за подписване на трансфер на токени и още едно за изпращане на цялостната транзакция.



Изображение 4.9 - Страница за наддаване в аукцион

Затова е добавен бутон “Bid via backend”, който очаква потребителя да подпише съгласие за изпращане на съответните токени за наддаването. След това подписаното съобщение се подава до сървъра, където се пази до 1 час преди края на аукциона. Тогава най-голямото наддаване бива изпълнено от сървъра. Така потребителя не плаща такса за транзакция, което позволява да се осъществяват повече наддавания. В края на аукциона собствениците на сървъра плащат такса за това 1 наддаване.

Под бутоните за участие има списък със запазените, но неосъществени наддавания.

## 4.2 Инструкции за стартиране

Успешното стартиране на целия проект от нулата изисква:

- Внедряване на умните договори
- Стартиране на уеб интерфейса
- Стартиране на сървърния софтуер

Всяко едно от тези три неща може да се осъществи с различни инструменти и по различни начини, но в тази глава ще разгледаме оптималните за процеса на разработка и тестване. Всички инструкции са съобразени с особеностите на операционните системи, базирани на Ubuntu.

#### 4.2.1 Предварителни изисквания

Преди свалянето на основните инструменти трябва да се изпълни “sudo apt update” и “sudo apt upgrade” за обновяване на списъка с хранилища.

Всичко освен смарт контрактите е написано на JavaScript, затова първо трябва да се свали Node.js с команда “sudo apt install nodejs”.

За да свалим инструменти като React.js и Express трябва да използваме package manager. В случая това е npm, който се сваля със “sudo apt install npm”.

Трябва да изтеглим софтуера на дипломната работа от Github, което може да се случи само ако имаме Git. Командата за сваляне на Git е “sudo apt install git”. След това сваляме проекта с “git clone https://github.com/LIOjosBG/Web3Marketplace”.

Проект разполага с три основни папки, в които трябва да влезем за да изпълним конфигурацията.

#### 4.2.2 Умни договори

В папката Hardhat се съдържат всички инструменти за писането, тестването и внедряването на договорите. Първо трябва да изпълним командата “npm i”, която ще свали всички нужни пакети. Самите договори се намират в папка contracts, а тестовете са в паката tests. За да изпълним тестовете пишем командата “npx hardhat test”, а за проверка на покритието им “npx hardhat coverage”. Преди да продължим с внедряването на договорите трябва да попълним INFURA\_API\_KEY, ACCOUNT\_PRIVATE\_KEY и ETHERSCAN\_API\_KEY в .env файла. Това са чувствителни и индивидуални данни, които не се качват в Github.

Вместо публична мрежа използваме Ganache, която представлява локална мрежа, на която можем да качваме умните договори. За да свалим инструмента ползваме командата “sudo npm i -g ganache-cli”, която ни предоставя CLI за комуникация с модула. За стартиране на мрежата използваме командата ‘ganache-cli -d -m "myself armed safe reveal tissue bag milk coil call sweet adult clever" --db ./ganache\_db’, която стартира мрежа с предвидими акаунти от мнемоничната фраза. --db показва, че искаме да запазваме действията в базата данни ganache\_db за да можем да я спирате и пускате повторно без да нулираме състоянието ѝ.

За самото внедряване в мрежата използваме командата “npx hardhat run scripts/deploy.js --network ganache”, което изпълнява deploy.js с настойките от hardhat.config.js за мрежа ganache. Изпълнението на скрипта принтира на конзолата JSON за адресите на договорите. ABI интерфейсите за договорите се намират в папката artifacts. По-късно ще трябва да ги вземем от там за да ги приложим във front и back end-a.

#### 4.2.3 Потребителски интерфейс

Изпълняваме “npm i” в папакта front-end и прилагаме получените адреси и ABI-и за умните договори в папката shared. В contractAddresses.json поставяме аутпута на скрипта за качване на договорите, а в ABIs слагаме json файловете от artifacts на hardhat.

Преди да стартираме апликацията трябва да сме стартирали локалната блокчейн мрежа чрез ganache-cli. Трябва да имаме свален Metamask, с който можем да се сдобием чрез extension manager-а на браузъра. След създаване на регистрация трябва да изберем опцията за импортиране на акаунт чрез частен ключ. Целта на това е да си осъществим достъп до създадените локални акаунти, които разполагат с етъри за ganache мрежата. Частните им ключове се генерират от мнемоничната фраза и са изписани в терминала при стартиране на ganache. От Metamask трябва да изберем определена мрежа. По подразбиране е посочен Ethereum mainnet, но за разработката трябва да я сменим с Localhost 8545.

След това сме готови да стартираме апликацията с “npm start”

#### 4.2.4 Сървърен софтуер

За начало изпълняваме “npm i” в папката back-end, след което попълваме адресите на договорите и техните ABI-и в папката contracts. В .env файла попълваме SQL\_PASS, INFURA\_KEY и ACCOUNT\_PRIVATE\_KEY, които са паролата за SQL root потребителя на нашата машина, API ключ за Infura при нужда от връзка с публична тестова мрежа и частния ключ на акаунта, чрез който искаме да оперираме.

За да работим с базата данни трябва да имаме ORM-a sequelize, който можем да си свалим със “sudo npm i -g sequelize”. В папакта изпълняваме “sequelize db:create” за създаване на база данни.

След това сме готови да стартираме софтуера с “npm start”.

## Заключение

Дипломната работа отговаря на всички посочени минимални изисквания и е в състояние да бъде използвана от потребители. Цялостната активна архитектура включва умните договори, графичния интерфейс и сървърния софтуер. Част от разработческия софтуер включва тестовете на децентрализираните приложения и скриптовете за внедряване в истинска блокчейн мрежа.

Има 3 основни типа договори. Първия отговаря за трансфера на активи и поддържането на балансите. Това е AgoraToken. Втория е обединяващото звено - Marketplace. Съдържа адресите на всички останали договори. Третия тип са всички пазари, които се присъединяват към Marketplace. Това за момента са SimpleSeller и SimpleAuction, но могат да се добавят договори за други търговски механизми, например холандски аукцион, лотария и тн.

Изградените тестове имат 100% покритие на настоящи код на нашите договори.

Разработени са автоматизирани скриптове за качване на байткода на компилираните договори в тестова и продукционна етериум мрежа. Допълнителни команди са разработени за добавяне на инициализираща информация в договорите.

При въвеждане на Etherscan API ключът в .env файла на Hardhat имаме достъп до команда за верифициране на договорите чрез Etherscan.

Потребителският уеб интерфейс ни дава достъп до най-важните методи на договорите, а сървърния софтуер запазва нужната информация за наддаванията, изображенията и описанията на продуктите.

## Бъдеща реализация

Платформата има голям потенциал, но за пълната ѝ разработка са нужни следните функционалности, чрез които може да се увеличи производителността и сигурността:

- Криптиране на инструкциите за доставка с публичния клюя на продавача на артикула.
- Добавяне на допълнителна информация за продуктите в базата данни
- Добавяне на информация за потребителите в базата данни
- Разработка на допълнителни механизми за продажби
- Развиване на администраторския панел

# Използвана Литература

[1] CERN: The birth of the web

[https://www.home.cern/science/computing/birth-web#:~:text=The%20first%20website%20at%20CERN,software%20in%20the%20public%20domain.\]](https://www.home.cern/science/computing/birth-web#:~:text=The%20first%20website%20at%20CERN,software%20in%20the%20public%20domain.)

[2] BitBay: Double Deposit Escrow

<https://bitbay.market/double-deposit-escrow>

[3] Bitcoin: A Peer-to-Peer Electronic Cash System

<https://bitcoin.org/bitcoin.pdf>

[4] Investopedia: Cryptographic Hash Functions: Definition and Examples

<https://www.investopedia.com/news/cryptographic-hash-functions/>

[5] Huabing Zhao: Hash Pointers and Data Structures

<https://zhaohuabing.medium.com/hash-pointers-and-data-structures-f85d5fe91659>

[6] Ethereum: Staking

<https://ethereum.org/en/staking/>

[7] Bitcoin.com: How do bitcoin transactions work?

<https://www.bitcoin.com/get-started/how-bitcoin-transactions-work/#2/>

[8] Suresh Jaganathan, Karthika Veeramani: A quick synopsis of Blockchain Technology

[https://www.researchgate.net/publication/333160118\\_A\\_quick\\_synopsis\\_of\\_Blockchain\\_Technology](https://www.researchgate.net/publication/333160118_A_quick_synopsis_of_Blockchain_Technology)

[9] Solana wiki: Transactions

<https://solana.wiki/docs/solidity-guide/transactions/#:~:text=The%20entire%20encoded%20size%20of%20a%20Solana%20transaction%20cannot%20exceed%201232%20bytes.>

[10] Kai Sedgwick: No, Visa Doesn't Handle 24,000 TPS and Neither Does Your Pet Blockchain

<https://news.bitcoin.com/no-visa-doesnt-handle-24000-tps-and-neither-does-your-pet-blockchain/>

[11] Chaillink: Solidity vs. Vyper: Which Smart Contract Language Is Right for Me?

<https://blog.chain.link/solidity-vs-vyper/#:~:text=Vyper%20is%20meant%20to%20be,mind%20on%20this%20subjective%20point>

[12] Ethereum: Ethereum Virtual Machine

<https://ethereum.org/en/developers/docs/evm/>

[13] Solidity: Документация

<https://docs.soliditylang.org/en/v0.8.18/>

ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ	1
<b>Използвани термини и съкращения</b>	<b>4</b>
<b>Увод</b>	<b>6</b>
<b>Глава 1</b>	<b>8</b>
1.1 Други подобни системи и продукти	8
1.1.1 BitBay	8
1.1.2 Safex	10
1.2 Блокчейн. Имплементация в Биткойн	10
1.2.1 История	10
1.2.2 Какво е P2P мрежа	11
1.2.3 Какво е CHF функция	12
1.2.4 Асиметрично криптиране	14
1.2.5 ECDSA	15
1.2.6 Linked list	15
1.2.7 Block structure	16
1.2.7.1 Prev block hash	16
1.2.7.2 Nonce	17
1.2.7.3 Merkle tree transaction hash	17
1.2.7.4 Transactions	17
1.2.7.5 Block hash	18
1.2.8 PoW vs PoS	18
1.2.9 Структура на транзакцията	19
1.2.10 Merkle tree	20
1.3 Развойни средства и среди	21
1.3.1 Блокчейн мрежи	21
1.3.2 Инструменти за разработка на умни договори	23
1.3.3 Език за разработка на умни договори	25
1.3.4 Тестова мрежа за разработка	28
1.3.5 Инструменти за разработка на уеб потребителски интерфейси	30
1.3.6 Инструменти за разработка на сървърен софтуер	32
1.3.7 Библиотеки за комуникация с умни договори	34
1.3.8 Развойни среди	36

<b>Глава 2</b>	<b>38</b>
2.1 Функционални изисквания към дипломната работа	38
2.2 Избор на езика за програмиране и софтуерните средства	38
2.3 Описание на структурата на базата данни	39
2.4 Избрани технологии и спомагателни инструменти	39
2.4.1 Етериум	39
2.4.2 Hardhat	40
2.4.3 Solidity	42
2.4.4 Ganache	43
2.4.5 React.js	43
2.4.6 Node.js	45
2.4.7 Ethers.js	45
2.4.8 Postman	45
<b>Глава 3 Реализация на проекта</b>	<b>47</b>
3.1 Архитектура	47
3.2 Контракти	48
3.2.1 Marketplace	49
3.2.2 AgoraToken	54
3.2.3 Simple Seller	58
3.2.4 Simple Auction	64
3.3 Тестове на умните договори	66
3.4 Скриптове за качване и верификация на умните договори	74
3.5 Интерфейс	75
3.5 Сървърна част	94
<b>Глава 4</b>	<b>108</b>
4.1 Ръководство за потребителя	108
4.2 Инструкции за стартиране	114
4.2.1 Предварителни изисквания	115
4.2.2 Умни договори	115
4.2.3 Потребителски интерфейс	116
4.2.4 Сървърен софтуер	116
<b>Заключение</b>	<b>117</b>
Бъдеща реализация	118
<b>Използвана Литература</b>	<b>119</b>

