Кафедра цифровой экономики

Лабораторный практикум по дисциплине

Технология блокчейн и криптовалюта

Методические указания

к лабораторным работам для студентов

направлений подготовки:

38.03.05 «Бизнес-информатика» (степень - бакалавр)

Оглавление

Лабораторная работа №1 3
Лабораторная работа №2 9
Лабораторная работа №3 23
Лабораторная работа №4 32
Приложение №1 58
Приложение №2 115

Лабораторная работа №1

Установка Ethereum Wallet

Цель работы: Получить представления и начальные навыки работы в сети Ethereum.
Результат: наличие установленного кошелька Ethereum Mist Wallet.

Теоретическая справка:

Индустрия криптовалют развивается стремительно быстро: каждый месяц появляются новые варианты кошельков, торговых площадок, обменных сервисов, да и ассортимент самих цифровых активов постоянно расширяется. Это новое веяние в финансовой сфере, позволяющее умелым инвесторам неплохо заработать на волатильности рынка. А чтобы проводить с криптовалютой различные спекулятивные операции и получать свою прибыль, ее нужно где-то хранить. Вариантов огромное количество и сегодня мы рассмотрим один из них: многофункциональный десктопный кошелек Ethereum Wallet.

Особенности хранилища

Прежде всего, Ethereum Wallet - это официальный кошелек для хранения Эфира и токенов, созданных на его блокчейне. Это продукт поддерживается разработчиками криптовалюты, а значит, обеспечивает максимальную надежность и защищенность ваших активов.

Бумажник имеет еще одно имя - Ethereum Mist Wallet - он является функцией браузера Mist, который в данный момент еще дорабатывается и в скором времени станет связующим приложением платформы для взаимодействия с сетью.

Ethereum Wallet поддерживает работу на следующих операционных системах:

- Windows;
- Mac OSX;
- Linux.

Само приложение полностью бесплатно, все комиссионные сборы, которые вы будете оплачивать за переводы, переходят майнерам сети Ethereum, поддерживающих ее работоспособность.

Ethereum Wallet - один из самых безопасных вариантов хранения, так как все приватные ключи и данные о ваших сбережениях хранятся локально на вашем устройстве и не попадают в сеть. Эфириум-кошелек является шлюзом для децентрализованных приложений на blockchain Ethereum.

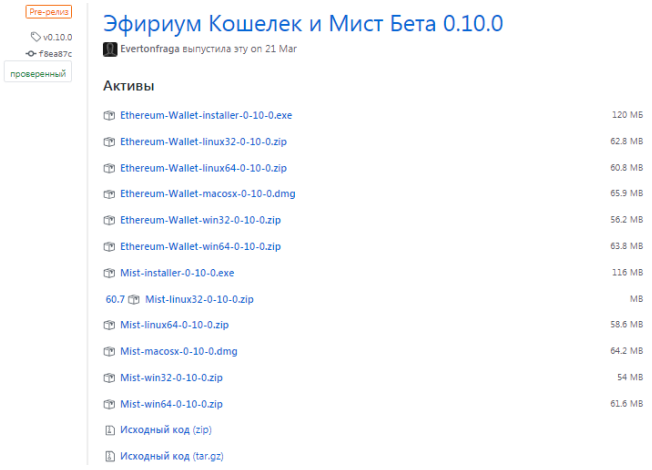
Обратите внимание на то, что приложение кошелька является довольно тяжелым, так как будет хранить полноценный блокчейн на вашем ПК, поэтому позаботьтесь о том, чтобы было достаточно свободного места (более 200 Гб).

Устанавливая Ethereum Wallet, вы имеете полную анонимность: предоставлять какие-либо личные данные или проходить процедуру верификации не придется.

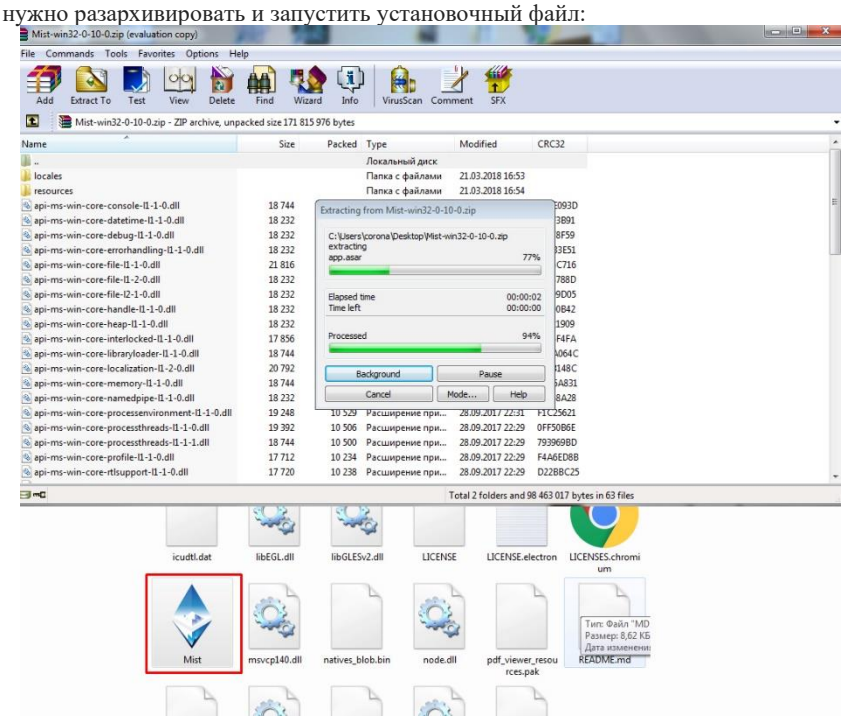
Как установить кошелек

Так как кошелек имеет официальный статус, работу начинаем непосредственно с сайта разработчика - https://ethereum.org

Все установочные файлы хранятся на Github. Выбираем свою версию и скачиваем архив на компьютер.



После этого папку нужно разархивировать и запустить установочный файл:

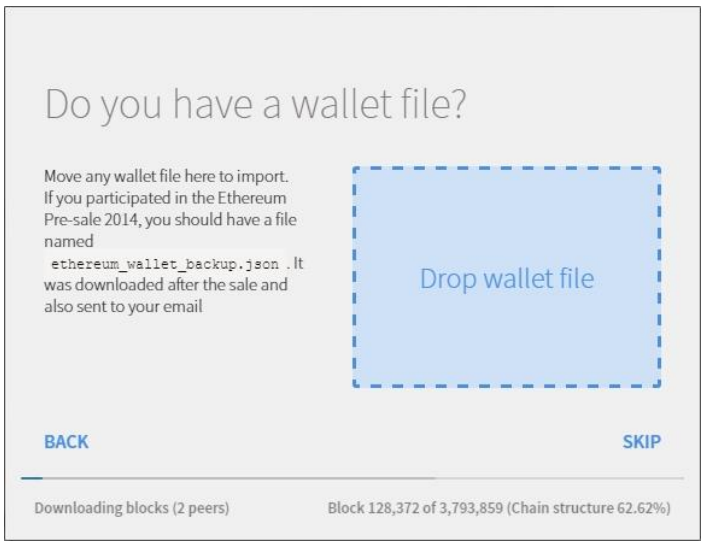


Сама программа-клиент занимает не более 150 Мб, но синхронизация блокчейна потребует около 200 Гб. После того, как установка программы пройдет успешно, перед вами откроется такое окно:

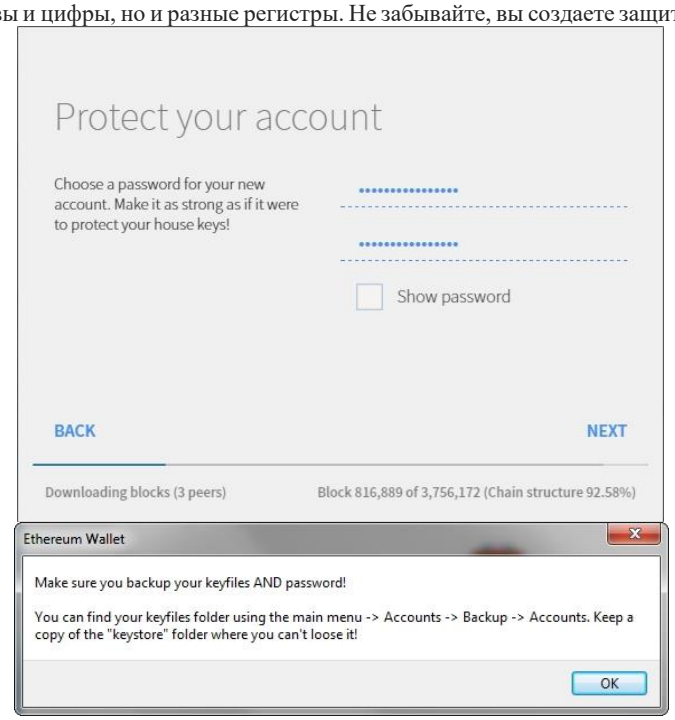
Вы должны выбрать сеть, в которой будет работать ваш кошелек:

- тестовую (для знакомства с кошельком она вполне подойдет);
- или основную (в этой версии блокчейн будет полноценно загружен).

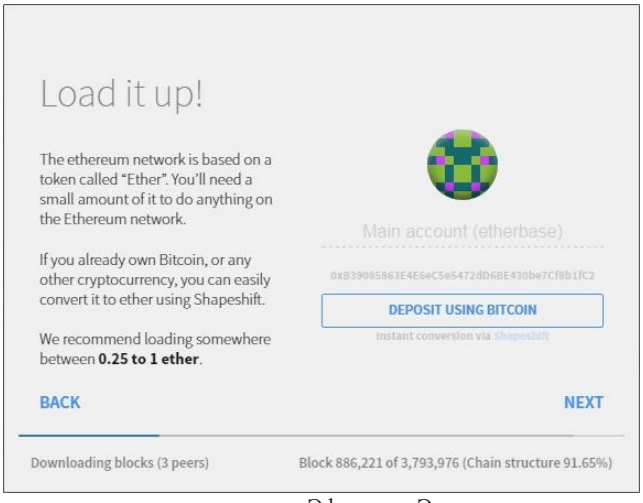
На следующем этапе можно вводить бэкап хранилища, но так как мы его не имеем, пропускаем этот шаг:



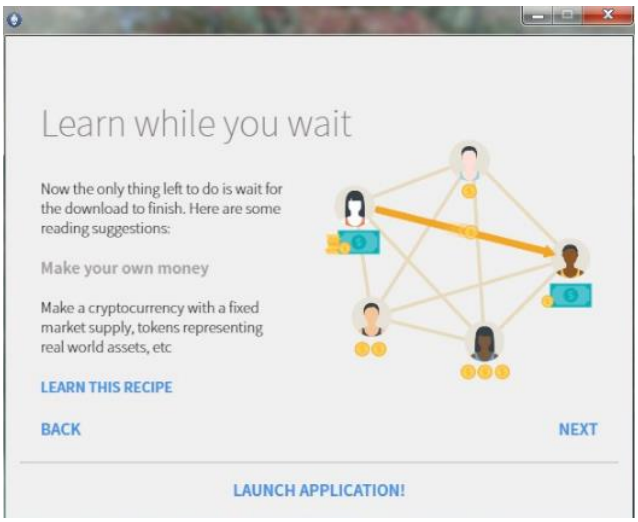
Затем открывается окно, в котором вам нужно будет создать пароль. Имейте в виду, что изменить пароль не получится, поэтому сразу отнеситесь очень серьезно к созданию сложной комбинации символов.
Используйте не только буквы и цифры, но и разные регистры. Не забывайте, вы создаете защиту своим денежным активам.



Следующее окошко напоминает нам, что использовать свой кошелек вы сможете, только имея на счету не менее 0,25 ETH. Перевести их можно прямо сейчас, используя встроенный инструмент ShapeShift. Если у вас пока нет нужных монет, вы сможете перевести их позже.

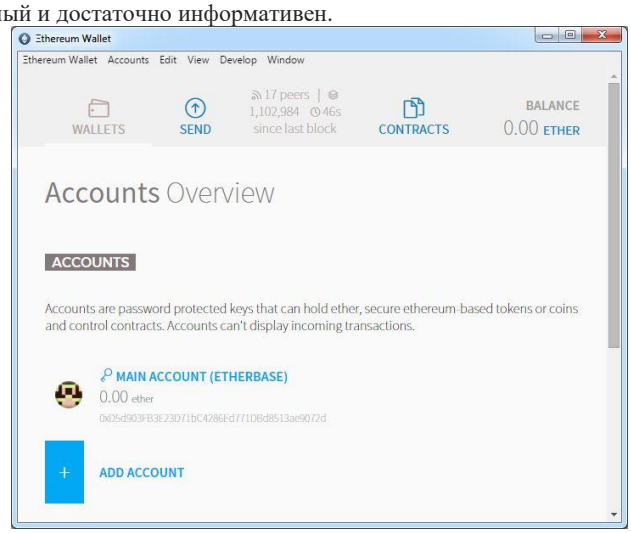


Следующее действие - синхронизация хранилища с сетью Эфириум. Этот процесс может занять несколько суток, после чего Ethereum Wallet будет полностью установлен и готов к работе.



Что внутри

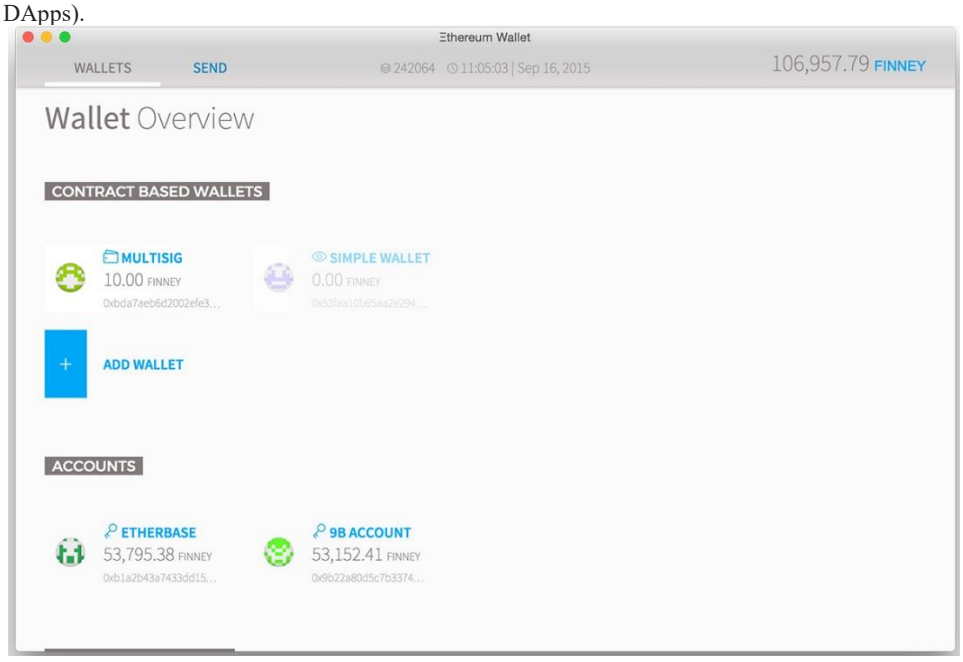
Процесс использования кошелька Ethereum Wallet будет легким не только для опытных пользователей, но и для новичков. Его интерфейс вполне понятный и достаточно информативен.



Перед тем, как пополнять баланс хранилища цифровой наличностью, лучше создать резервную копию. Для этого выбираем закладку «Аккаунты» и пункт «Резервное копирование». Сформированный файл сохраните в очень надежном месте, доступ к которому у вас будет всегда. Целесообразно иметь несколько копий.

Etereum Wallet имеет разделение на 2 основных раздела:

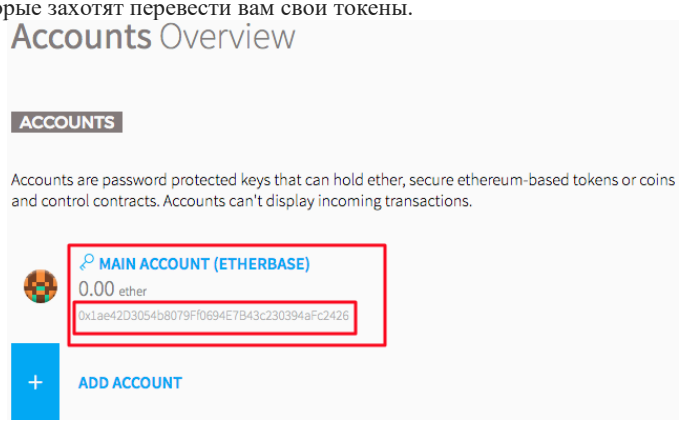
1. Accounts (в этом разделе вы сможете видеть баланс кошелька и суммы поступлений).
2. Contract Based Wallets (здесь вы сможете видеть полную информацию обо всех входящих и исходящих транзакциях и использовать DApps).



Учетный раздел (Accounts) имеет закрытый ключ с паролем и адресом, контрактный раздел (Contract Based Wallets) не имеет закрытого ключа, но имеет свой адрес, код и хранилище, с помощью которых можно не только создавать кошельки, но и всевозможные интересные децентрализованные приложения (DApps).
Можно сказать, что учетная часть кошелька отвечает за простое хранение Эфира, а контрактная часть обеспечивает управление криптовалютой.

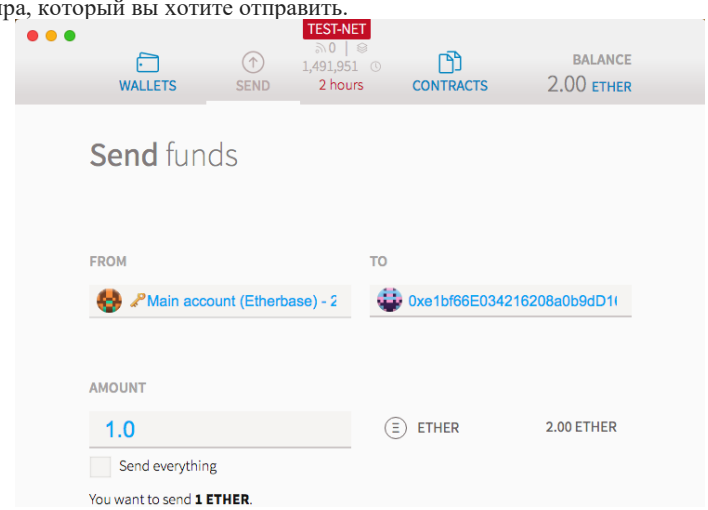
Как пользоваться кошельком

Первая задача - это пополнить баланс кошелька Эфиром. На главной странице хранилища в разделе Main account вы можете видеть адрес хранилища, на который вам нужно перечислить как минимум 0,25 ETH.
Ethereum-адреса представлены в шестнадцатеричном формате: они состоят из 40 символов. Этот адрес вы будете давать всем своим знакомым, которые захотят перевести вам свои токены.

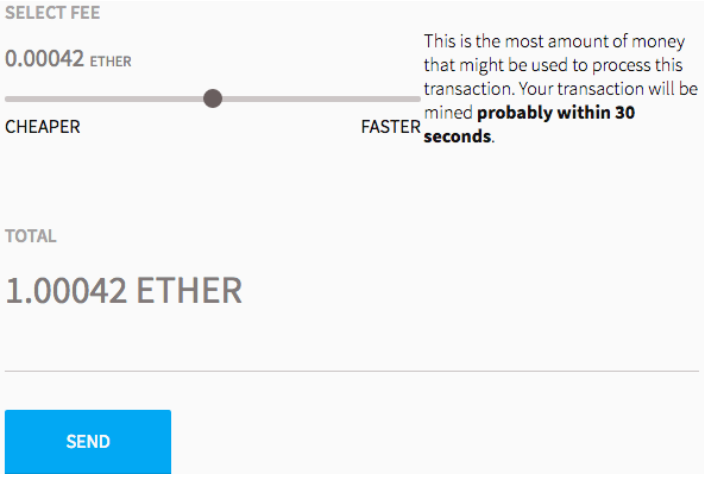


Чтобы создать исходящую транзакцию и отправить эфир из Mist Wallet, выполните следующие действия:

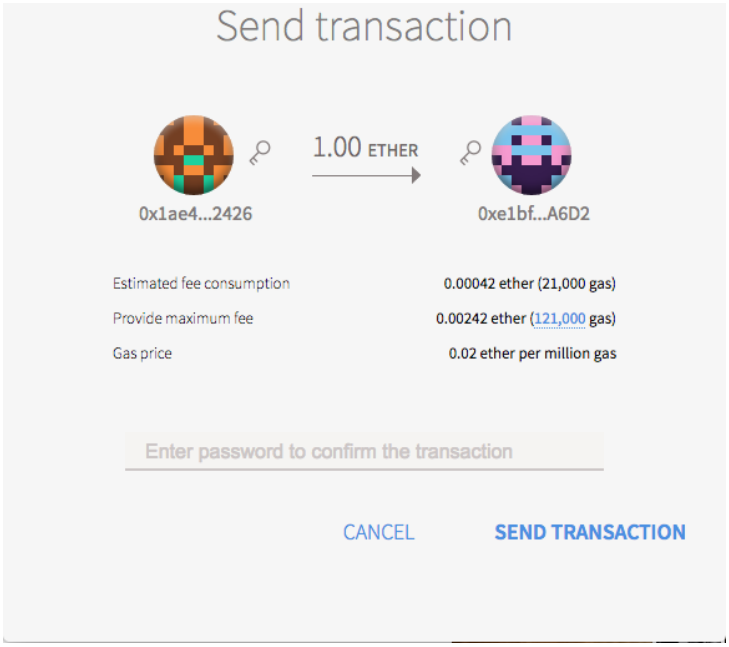
1. Перейдите к разделу SEND.
2. Укажите адрес принимающей стороны.
3. Выберите количество эфира, который вы хотите отправить.



Вы можете оставить комиссию такой, какую предложила программа по умолчанию, а можете изменять ее размер, перемещая бегунок:



Останется только ввести правильно свой пароль:



Чтобы посмотреть информацию о какой-то конкретной транзакции, перейдите в раздел «Обзор кошелька» → «Последние транзакции» и кликните на нужную операцию в списке:

Заключение

Ethereum Mist Wallet предназначен для надежного и безопасного хранения всех типов монет и токенов Эфириума, а также для смарт-контрактного взаимодействия пользователей через децентрализованные приложения браузера.

Положительные стороны Mist-кошелька - это его многофункциональность, понятный интерфейс и высокие показатели безопасности.

Из минусов можно отметить только очень долговременную синхронизацию: процесс занимает около 2 суток. Но, это плата за надежную сохранность ваших цифровых активов.

Задание

1. Установить кошелек Ethereum Mist Wallet. Подключаемся только к тестовой сети.
2. Ознакомиться с функционалом кошелька.

Контрольные вопросы

1. Почему при установке кошелька Ethereum Mist Wallet в рабочем режиме требуется большой объем свободного пространства на винчестере?
2. Какие операции доступны в кошельке Ethereum Mist Wallet?
3. Укажите недостатки кошелька Ethereum Mist Wallet?

Лабораторная работа №2

Знакомство с инструментами и средой разработки смарт-контрактов

Цель работы: изучить и закрепить на практике возможности основных инструментов разработчика смарт-контрактов.

Результат: наличие установленного кошелька MetaMask, настроенного на работу с тестовой сетью Rinkeby.

Теоретическая справка:

Смарт-контракты: что это?

Смарт-контракты, или "умные контракты", позволяют передавать некоторые ценности, например, собственность или акции, прозрачным и одновременно безопасным способом, что делает весь процесс сверхэффективным, одновременно устраняя промежуточные звенья, зачастую долгие и дорогие. Рассмотрим пример, который позволит понять, как блокчейн работает со смарт-контрактами.

Давайте представим, что есть два заинтересованных лица в сделке с недвижимостью. Один (*продавец*) желает продать жилье, а другой (*покупатель*) хочет купить это жилье. *Сделка* по продаже может быть реализована посредством блокчейна, и *покупатель* готов платить, например, биткоинами. Как только *покупатель* заплатит, то сразу получит подтверждение о транзакции, которое будет исполнено в виде виртуального смарт-контракта. *Продавец*, в свою *очередь*, передает покупателю цифровой *ключ* от *входной* двери, который будет доставлен в день, о котором заинтересованные стороны договорились. Если *продавец* вдруг передумает продавать дом, *покупатель* не получит *ключ*, блокчейн в этом случае автоматически вернет покупателю деньги в тот день, когда должен был быть получен *ключ*. А если *покупатель* получит *ключ* заранее, то блокчейн его удержит до дня, в который была договоренность осуществления передачи. Поэтому каждая из сторон получит то, что хочет, в оговоренный в контракте день: *продавец* - деньги, а *покупатель* - *ключ*. А поскольку блокчейн - это технология, основанная на пиринговой сети, договор по этой сделке будет храниться на множестве узлов, что обеспечит выполнение взятых по контракту обязательств, и ни одна из сторон не сможет изменить условия контракта после его заключения. Ну а если кто-то из сторон наберется смелости сделать это, все узлы в сети тут же об этом узнают, и проблема будет мгновенно решена.

Мы рассмотрели пример с куплей-продажей недвижимости. Но такие же соглашения могут заключаться при передаче акций, в страховании автомобилей или другого имущества и во многих других случаях. Позвольте привести несколько ключевых преимуществ смарт-контрактов.

Первое качество, за которое смарт-контракты так ценятся, это *автономность*. Смарт-контракты не могут быть изменены третьими лицами, так как только их стороны заключают соглашение. Нет необходимости обращаться к услугам юристов при заключении соглашений.

Второе преимущество, за которое люди любят - или еще полюбят - смарт-контракты, это *доверие* к ним. Смарт-контракт невозможно потерять. Они все зашифрованы и хранятся в общественном хранилище. Поэтому потеря любого из них исключена.

Это подводит к следующему плюсу - *резервированию*. Можно положиться на *надежность* смарт-контрактов, потому что они все зарезервированы. *Аннулирование* договора по причине потери его копии просто невозможно.

Следующим в списке идет *безопасность*, которая опять же связана с предыдущими двумя. Ваши смарт-контракты будут защищены современными методами шифрования данных. Это отсылает нас к вопросу доверия - вы можете полностью доверять безопасности методов шифрования. Смарт-контракт практически невозможно взломать.

Пятая причина превосходства смарт-контрактов над обычными - это *скорость* их передачи. На заключение традиционных договоров уходит уйма времени, поскольку в их эту работу вовлечено множество третьих лиц. Если речь идет о распространении кода, смарт-контракты на высоте, поскольку позволяют решать задачи в разы быстрее.

Шестая причина - это *экономия денег* на заключении договоров. Нет необходимости прибегать к услугам адвокатов. Можно просто использовать технологию смарт-контрактов.

И, наконец, огромным преимуществом является *точность*. Если все подробности контракта указаны точно, то он будет выполнен значительно точнее, чем любой другой контракт.

Инструментарий и приложения экосистемы эфириума

Прежде чем погрузиться в написание кода, стоит изучить экосистему Ethereum. Давайте разберемся, какие инструменты и подходы существуют, как они называются и взаимодействуют.

В экосистеме Ethereum широко используются такие инструменты, как Geth, *Parity*, Solidity, Remix, Truffle, Webpack, Angular и так далее. Каждый из них используется для решения конкретных задач.

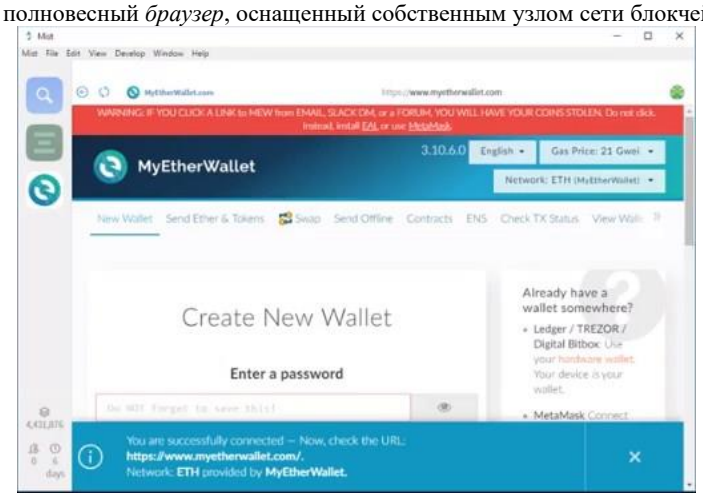
Узлы сети блокчейна: Go-Ethereum, Parity, CPP-Ethereum

Примерами узлов блокчейна выступают такие программы, как Geth, *Parity* или CPP-Ethereum. Все они работают на клиентской стороне, то есть их можно загрузить и запустить на вашем компьютере, как и для всех других пользователей сети Ethereum. Они все выполняют одну и ту же задачу: реализуют протокол Ethereum. Несмотря на то, что разные инструменты выполняют одну и ту же роль, они написаны на разных языках программирования. Развитием инструментов занимаются различные команды, которые обязательно следят за тем, чтобы даже на разных языках программирования протокол Ethereum был реализован корректно. Если проводить аналогию, то эта схема похожа на использование среды MySQL в режиме "мульти-мастер", когда все узлы выполняют одну и ту же задачу по репликации *базы данных*. Это отлично описывает то, что делают все узлы в сети блокчейна - они копируют все блоки на своих компьютерах. Поэтому при загрузке Geth, *Parity*, или CPP-Ethereum и запуске клиента после установки подключения к другим узлам будет загружено все содержимое блокчейна. *Исключение* составляет только режим "легкого клиента", когда загружаются только *заголовки* блоков.

Взаимодействие веб-сайтов и блокчейна

Рассмотрим популярные браузеры MetaMask и Mist. Оба они представляют собой связующее звено между обычным браузером для просмотра интернет-страниц и блокчейном С помощью корректно настроенного веб-сайта можно выполнять программы и отправлять команды в блокчейн. *Пользователь* сможет запустить любой *браузер*: например, *Chrome*, Firefox, *Internet* Explorer или другой *браузер*, зайти на такой *веб-сайт* и взаимодействовать с блокчейном. Для этого к блокчейну необходимо подключиться. MetaMask представляет собой надстройку для *Chrome* и Firefox, облегчающую подключение к блокчейну.

Mist, в свою *очередь* - это полновесный *браузер*, оснащенный собственным узлом сети блокчейна.



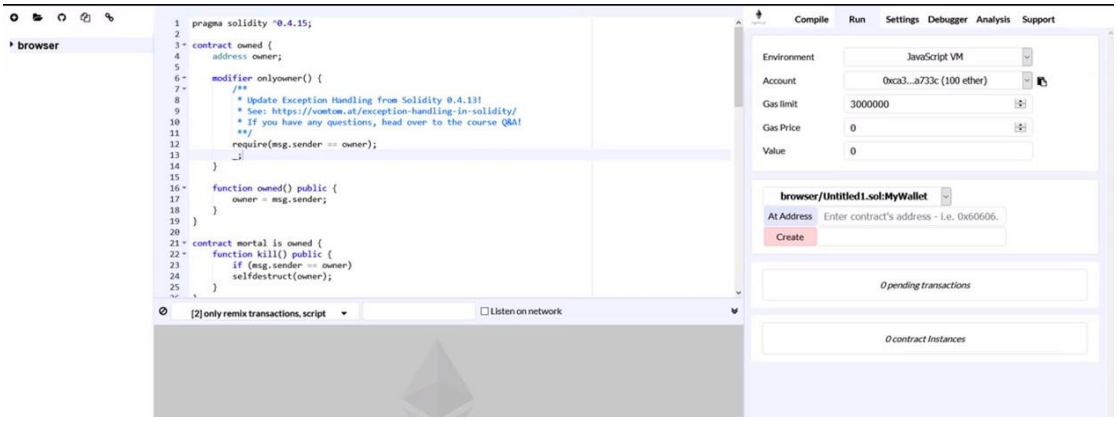
В случае с Mist, узлом блокчейна является Geth, или Go-Ethereum, непосредственно встроенный в *браузер*. MetaMask для работы использует сервис под названием Infura. В среде Infura используются узлы Geth и *Parity*, которые запущены на стороне сервера, а не на компьютере клиента, а Infura реализует подключение к ним. Чтобы познакомиться с надстройкой MetaMask, можно открыть *браузер*, например, *Chrome*, найти раздел с иконками надстроек, далее открыть MetaMask и можно начинать работать с блокчейном. Кошелек Mist выглядит так: слева *доступ* к различным разделам, есть *отображение* статуса подключения и синхронизации данных, в центре располагается собственно *браузер* Это позволяет работать с блокчейном и просматривать веб-страницы можно одновременно.

Что такое Solidity

Solidity представляет собой *язык программирования* высокого уровня. Для ее работы требуется *компилятор* solc, который формирует байткод для виртуальных машин Ethereum.Встречаются мнения, что Solidity похож на JavaScript. В первых версиях так и было, однако сейчас эти два языка значительно расходятся. Тем не менее, Solidity похож на JavaScript больше, чем любой другой *язык программирования*.

Remix, веб-среда разработки для Solidity

Remix - это облачная *среда разработки*, поддерживающая много полезных функций. *Доступ* к Remix можно получить по адресу http://remix.ethereum.org. Среда Remix позволяет создавать и запускать код на языке Solidity прямо в окне браузера. Remix оснащена встроенным отладчиком и статическим анализатором кода, а также многими другими инструментами.



На текущий момент Remix выглядит так. Слева расположен *браузер*, с помощью которого можно управлять файлами. В центре располагается окно для создания кода, а справа - *управляющие* элементы - вкладки для компиляции (*Compile*), запуска (Run), изменения настроек (Settings), отладки (*Debugger*), анализа (*Analysis*) и получения поддержки (*Support*).На вкладке Run можно выбрать среду запуска кода, например, виртуальную машину *Java*. Remix предоставляет *доступ* к нескольким счетам в эмулированной среде Ethereum для апробирования создаваемого кода. С их помощью можно размещать и обсчитывать контракты, а потом анализировать результаты благодаря наличию журнала исполнения кода.

Использование библиотек Web3.js и Eth.js

Библиотеки Web3.js и Eth.js облегчают взаимодействие между браузером и блокчейном и позволяют работать узлами сети Ethereum по протоколу *RPC* посредством *HTTP* и кода JavaScript. Если запустить локальный узел блокчейна, он откроет *интерфейс HTTP-RPC*, что позволит браузеру отправлять узлу команды, чтобы узел, в свою *очередь*, переправлял данные в блокчейн.

Библиотека Truffle и ее отличие от Web3.js

Truffle и Embark являются инструментариями для среды Solidity и разработки распределенных приложений для работы с блокчейном. Оба они поддерживают управление контрактами, их *размещение* в блокчейне, или миграцию, оснащены встроенной системой тестирования приложений, а Truffle еще и предлагает решение Truffle Boxes - предварительно настроенные среды разработки распределенных приложений, значительно облегчающие работу, такие как Truffle-React, Truffle-Webpack и так далее. При серьезном подходе к разработке приложений для блокчейна стоит уделить внимание Truffle и Embark и постепенно отходить от использования только библиотеки Web3.js.

Использование Angular, Vue.js, React и Redux в разработке приложений для блокчейна

Такие наборы инструментов, как Angular, Vue.js, React, Redux предназначены для разработки веб-страниц и непосредственно не работают с блокчейном, Truffle, Solidity и другими подобными средами. Для работы с Angular, Vue.js, React, Redux или другими инструментариями для создания веб-страниц обычно достаточно загрузить библиотеку Web3, подключиться к узлу блокчейна и настроить взаимодействие с блокчейном с помощью Web3.

Применение инструментов Browserify и Webpack

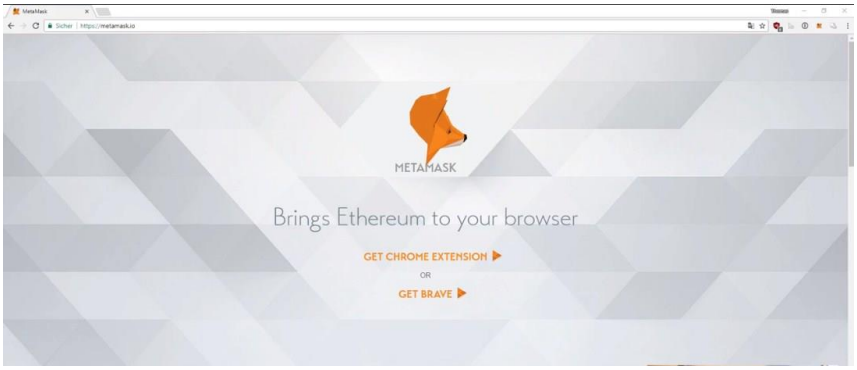
Webpack - это упаковщик файлов JavaScript, необходимо использовать тогда, когда *программа* использует большое количество файлов: Webpack собирает их воедино, разрешает все взаимозависимости между файлами, позволяя коду обращаться только к паре мастер-файлов, что значительно ускоряет загрузку веб-приложения, тк веб-серверу больше не приходится отправлять несколько сотен файлов.

Browserify делает примерно то же самое, но на базовом уровне - ведь Webpack сразу решает спектр задач по упаковке файлов для веб-разработки. Browserify представляет собой только упаковщик, разрешающий файловые взаимозависимости и объединяющий много файлов в один. *Node Package Manager*, или NPM, загружает и управляет пакетами для узлов сети Ethereum, что облегчает разработку веб-проектов.

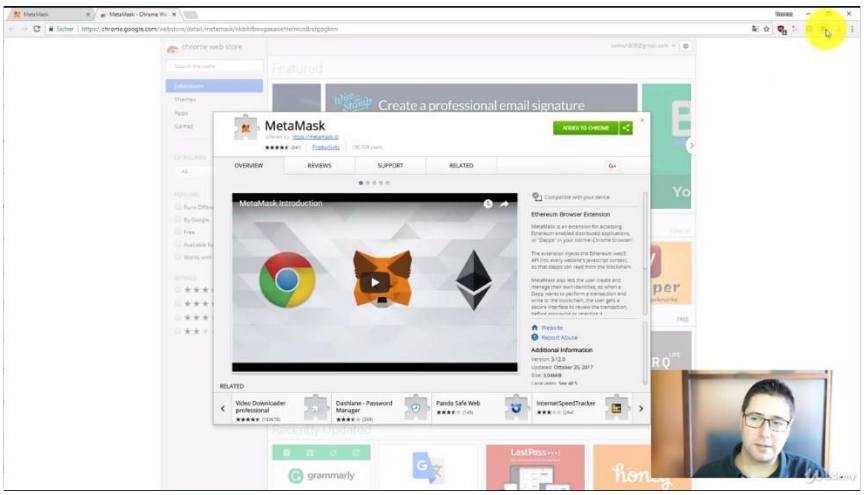
Обзор и возможности MetaMask

Поговорим о надстройке для браузера *Chrome* - MetaMask. В этом разделе установим ее, разберем функционал, а также узнаем, как получить немного эфира для тестирования MetaMask и размещения контрактор в тестовой сети Rinkeby.

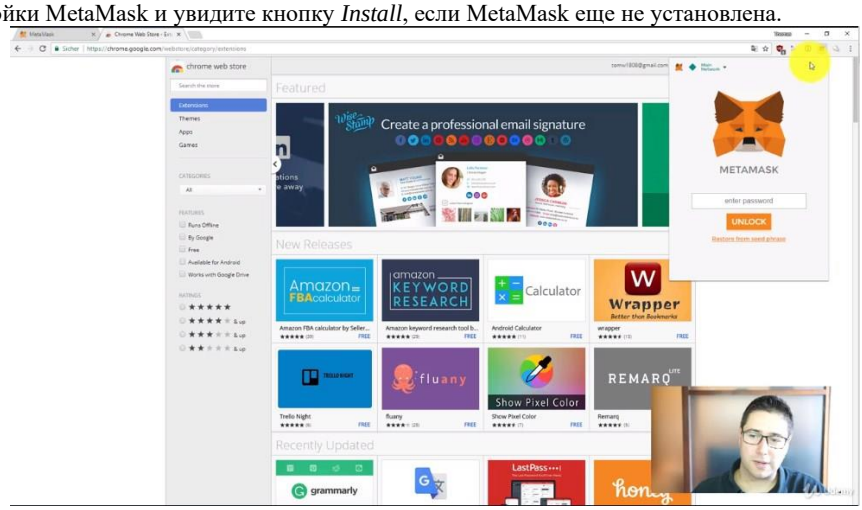
Установка MetaMask



Откроем веб-*сайт* MetaMask, маскотом которой является лисичка.



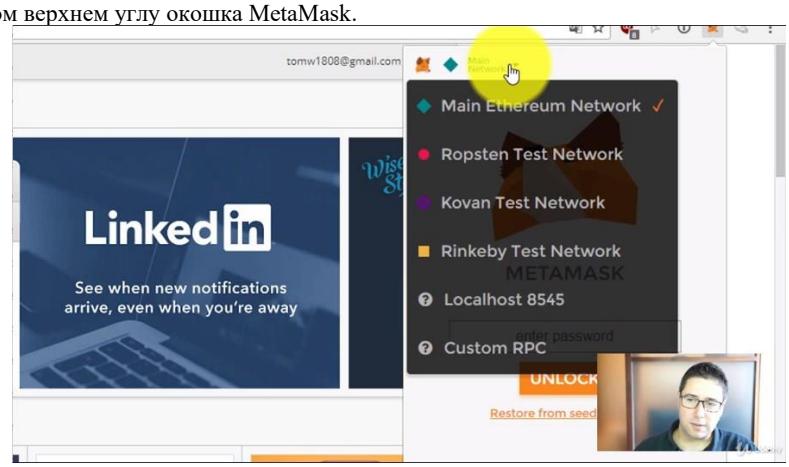
При щелчке на *Get the Chrome Extension* (*браузер Chrome* существует для всех платформ) вы перейдете на страницу установки надстройки MetaMask и увидите кнопку *Install*, если MetaMask еще не установлена.



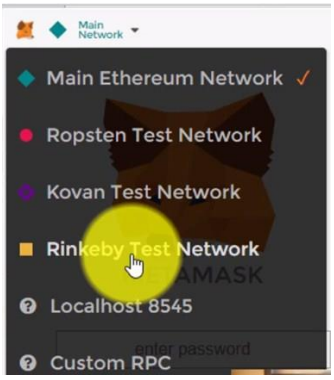
После установки MetaMask можно будет запустить с помощью ее иконки в верхнем правом углу браузера *Chrome*, а несколько простых шагов установки сопровождаются подробными инструкциями. Потребуется задать *пароль* для защиты ваших счетов, после этого можно работать с MetaMask.

Элементы MetaMask

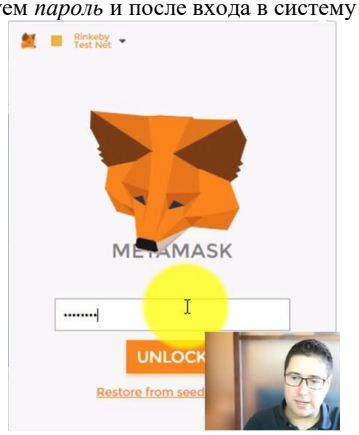
Продолжим разговор о надстройке MetaMask. Для начала выбраем *сеть*, с которой вы будете взаимодействовать - ее можно выбрать в левом верхнем углу окошка MetaMask.



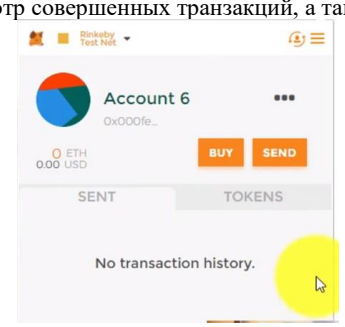
После этого необходимо опубликовать свое *приложение* в главной сети Ethereum, которое будем использовать для взаимодействия с другими смарт-контрактами, опубликованными там.



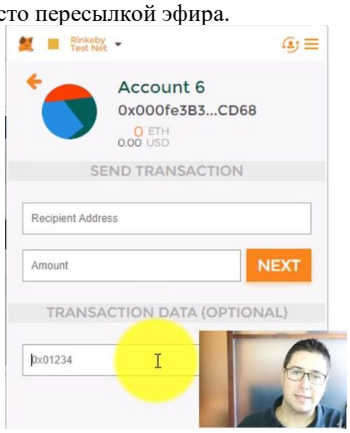
В рамках данного курса будет достаточно тестовой сети Rinkeby. Мы узнаем, как получить немного эфира для использования в этой тестовой сети, разобраться с тем, как *сеть* реагирует на запросы, посмотрим на майнинг, задержки и проблемы одновременных вычислений, характерные для блокчейна.
Теперь авторизуемся в MetaMask. Используем *пароль* и после входа в систему будет создан счет.



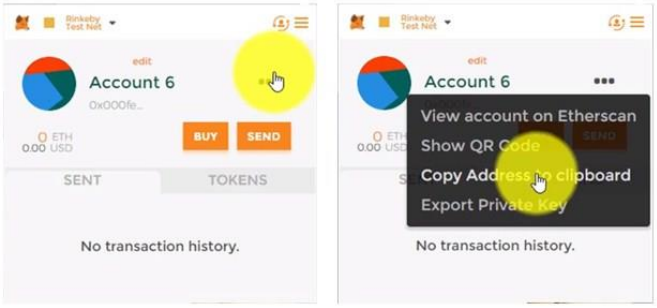
Доступен обзор имеющихся счетов, просмотр совершенных транзакций, а также жетонов среды Ethereum.



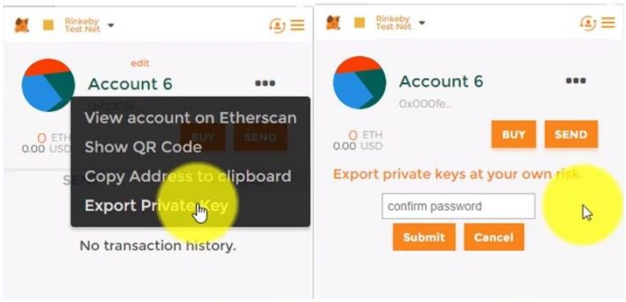
Если на счету есть эфир, его можно отправить на другой *адрес* и добавить к транзакции данные. Об этом мы поговорим позднее. На данном этапе ограничимся просто пересылкой эфира.



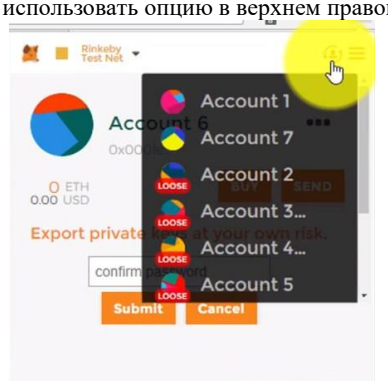
В *меню* обзора счета можно скопировать *адрес* в *буфер* обмена и экспортировать частный *ключ* вашего счета для использования его в другом приложении.



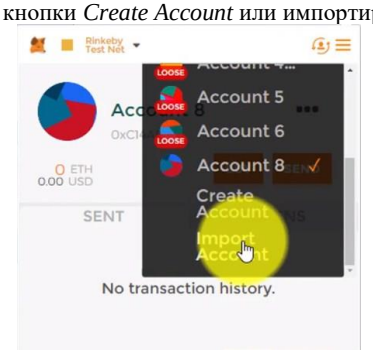
Обратите внимание, что *доступ* к счетам обеспечивается с помощью вашего частного ключа.



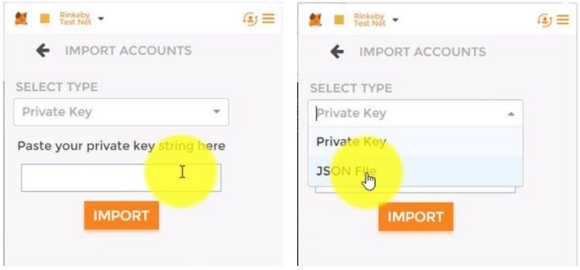
Для переключения между счетами можно использовать опцию в верхнем правом углу.



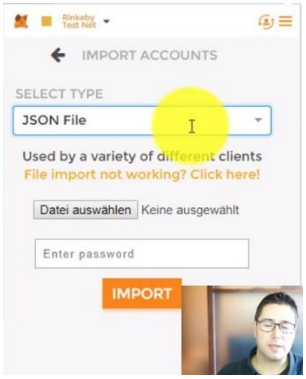
Можно создавать новые счета с помощью кнопки *Create Account* или импортировать счета с помощью *Import Account*.



Для этого потребуется частный *ключ* к счетам, предварительно созданным в Geth или другой системе, создающей ключи *JSON*.

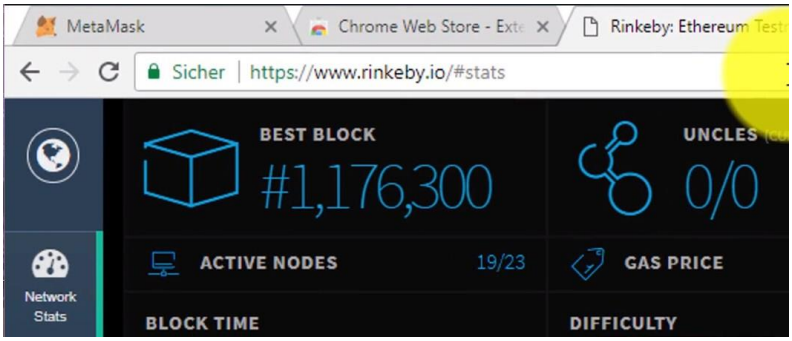


Импорт осуществляется посредством этих файлов-ключей *JSON*.

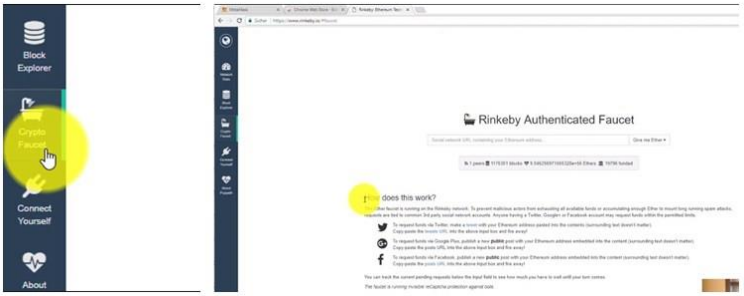


Как получить эфир для тестовой сети Rinkeby

Чтобы получить некоторое количество эфира для использования в тестовой сети Rinkeby, потребуется открыть веб-*сайт* http://rinkeby.io.

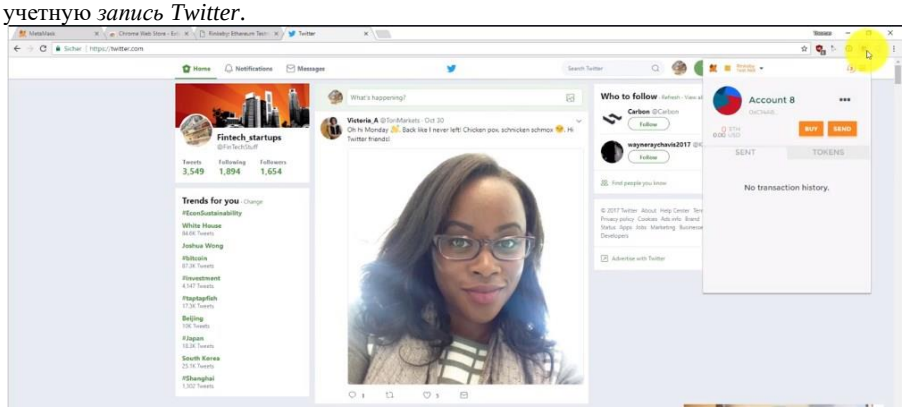


В его нижнем левом углу расположена иконка Crypto Faucet, при щелчке по которой разъясняется, как можно получить эфир на счет Rinkeby.

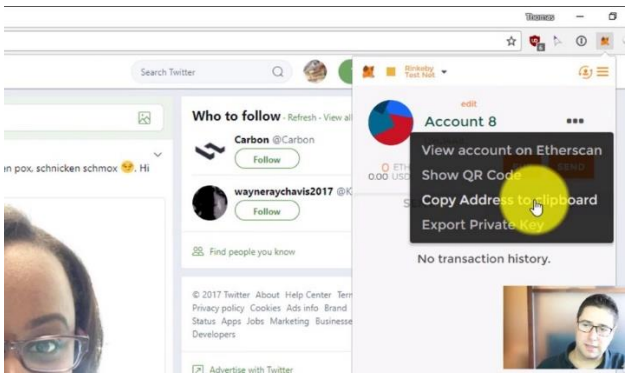


Для получения эфира достаточно опубликовать номер своего счета в *Twitter*, Google Plus или Facebook, а затем скопировать *адрес* веб-страницы с публикацией в форму на сайте rinkeby.io. Эти меры предосторожности необходимы для защиты от автоматического массового получения эфира.

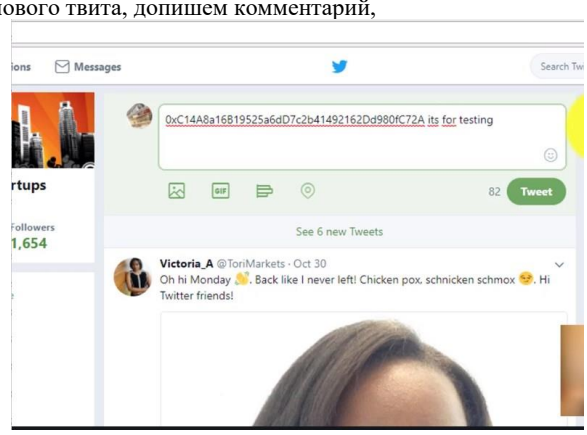
Теперь зайдем в учетную *запись Twitter*.



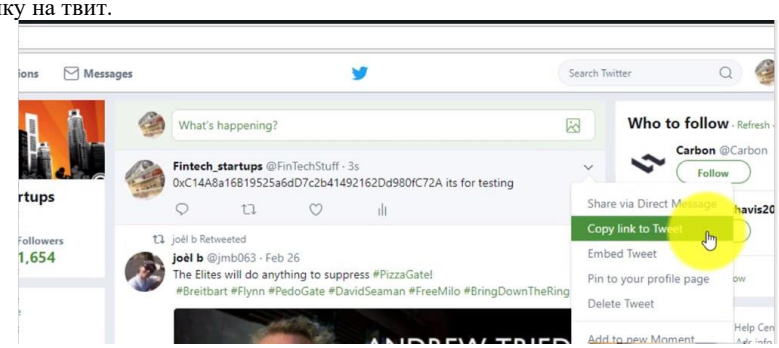
скопируем *адрес* счета из MetaMask,

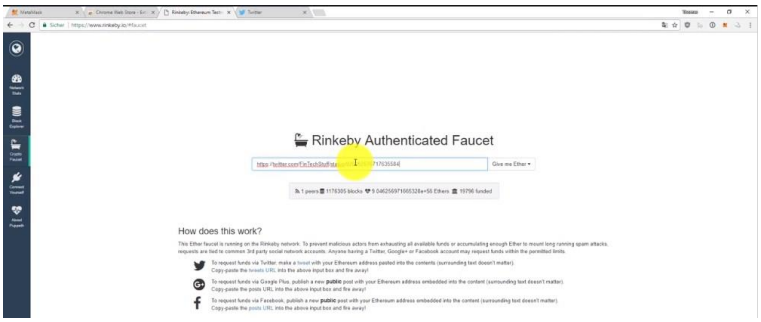


введем в *поле* для публикации нового твита, допишем комментарий,



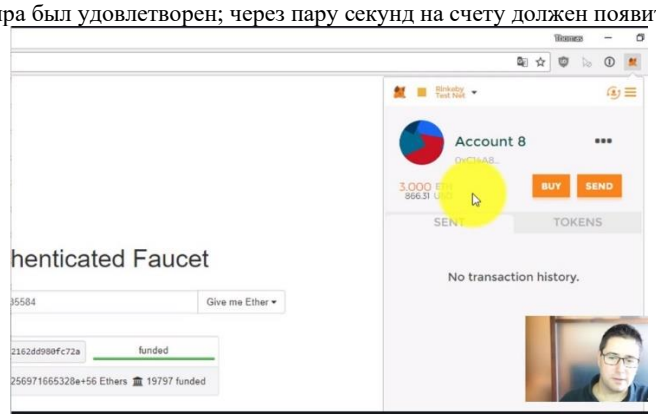
затем скопируем ссылку на твит.





То же самое можно сделать и с помощью Facebook, единственное, в чем нужно убедиться - это должен быть публичный пост. Так, *запрос* на выделение эфира был размещен.

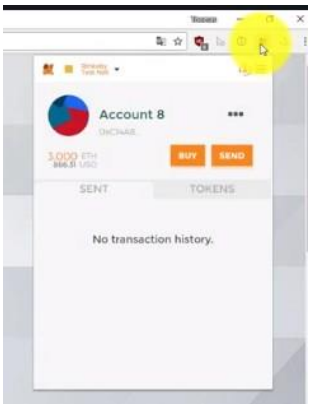
Для корректной работы системы начисления эфира потребуется отключение блокировщика рекламы. Теперь убедимся, что *запрос* на выделение эфира был удовлетворен; через пару секунд на счету должен появиться эфир.



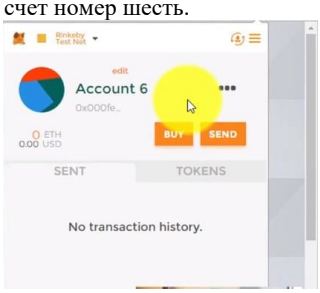
Наличие на счету эфира можно проверить с помощью сервиса Etherscan.

Пересылка эфира с помощью MetaMask

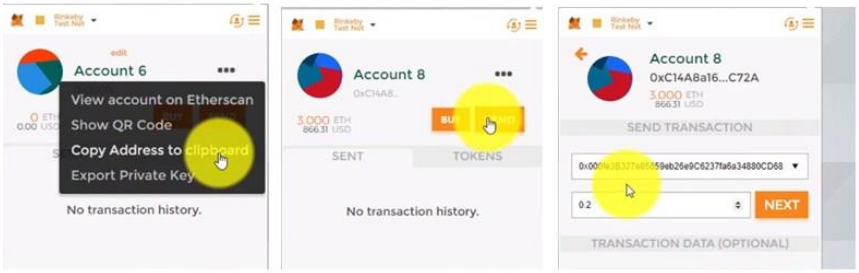
Теперь попробуем выполнить пересылку эфира между счетами с помощью MetaMask. Допустим, что на счете номер восемь есть три единицы эфира.



Необходимо переслать часть этих средств на счет номер шесть.

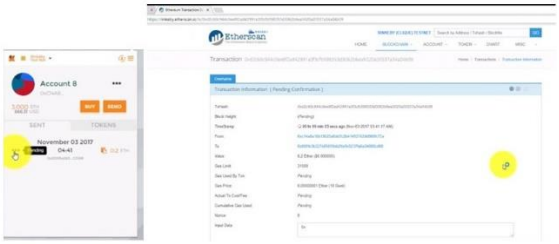


Скопируем *адрес* счета номер шесть, переключимся на счет номер восемь и отправим 0,2 единицы эфира.

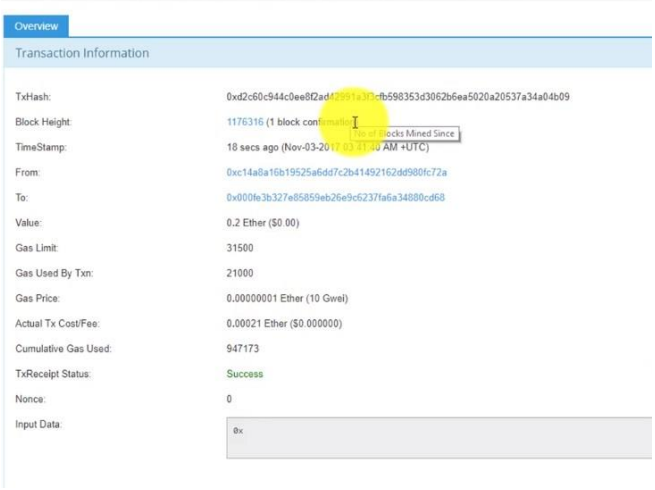


Щелкнем по *Next*, посмотрим на транзакцию и отправим ее в блокчейн.

Статус транзакции можно проверить с помощью сервиса Etherscan.

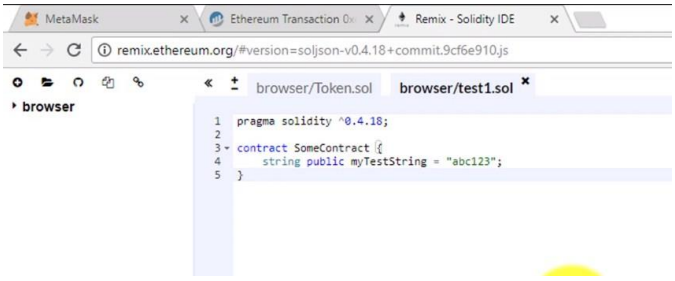


После успешно завершенного обсчета транзакции, средства будут отправлены, и будет отображаться *адрес* блока, в котором эта *транзакция* была впервые подтверждена.

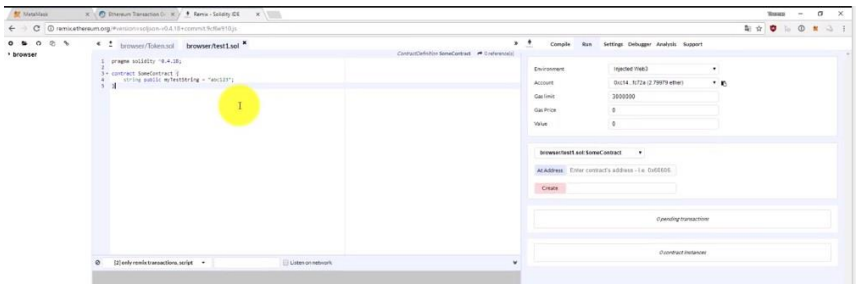


Взаимодействие MetaMask и браузера

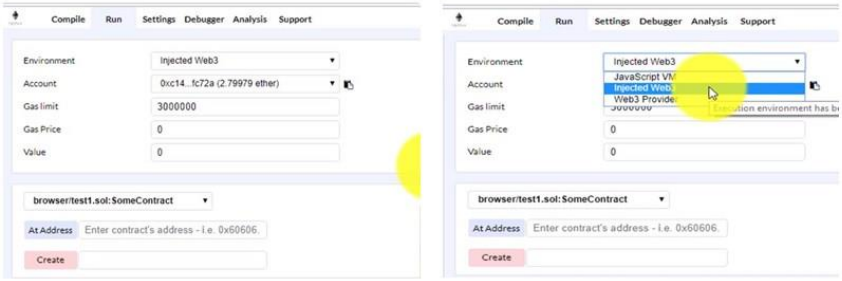
Следующей возможностью MetaMask является взаимодействие с блокчейном посредством веб-сайта. В этом случае веб-*сайт* подключается к MetaMask, MetaMask - к блокчейну с помощью сервиса Infura, а Infura, в свою *очередь*, содержит в себе запущенный клиент Geth. Позднее, когда будем разбирать тему подключения к блокчейну из браузера, данная схема будет рассмотрена подробнее - для нее возможны несколько реализаций. На текущем этапе достаточно посмотреть, что происходит в надстройке MetaMask, когда вы пытаетесь взаимодействовать с блокчейном. Откроем среду Remix, здесь есть простой смарт-контракт.

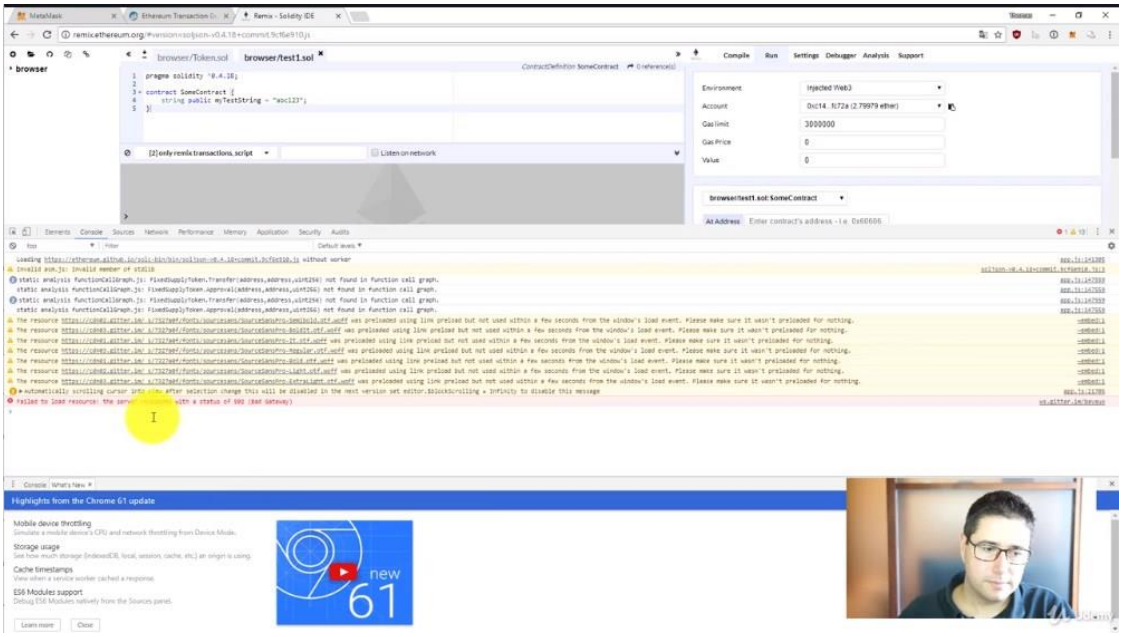


В правой стороне окна Remix видно, что в качестве опорной библиотеки выбрана Web3.

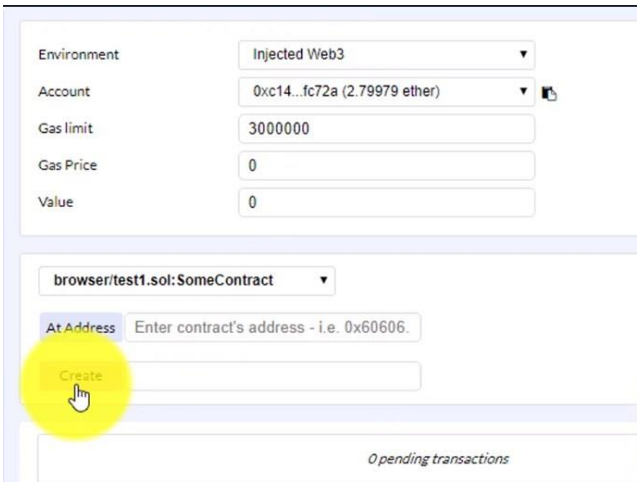


Надстройка MetaMask подключается непосредственно к окну браузера и таким образом обеспечивает *связь* с блокчейном. В раскрывающемся списке выбрана Injected Web3, а в окне разработки, доступном для любой веб-страницы, видно, что с помощью объекта *web3.currentProvider* можно работать с надстройкой MetaMask посредством обычного кода JavaScript.

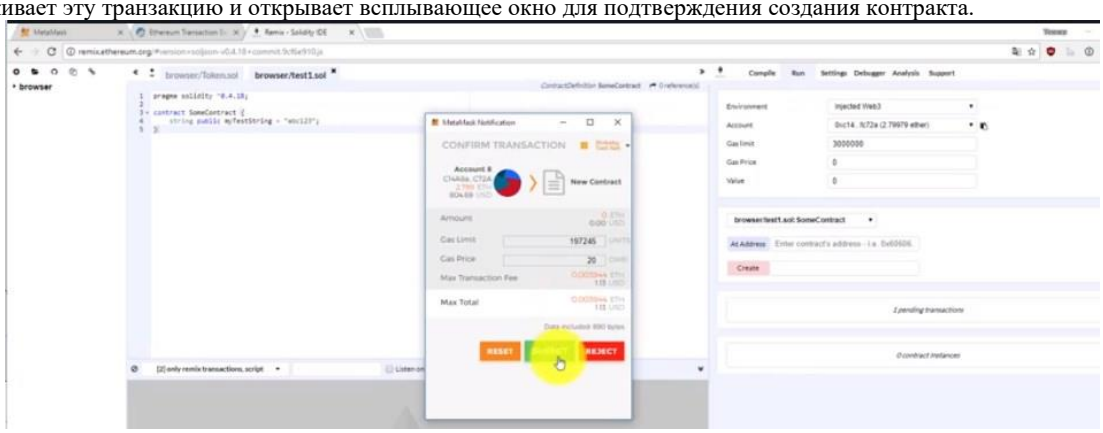




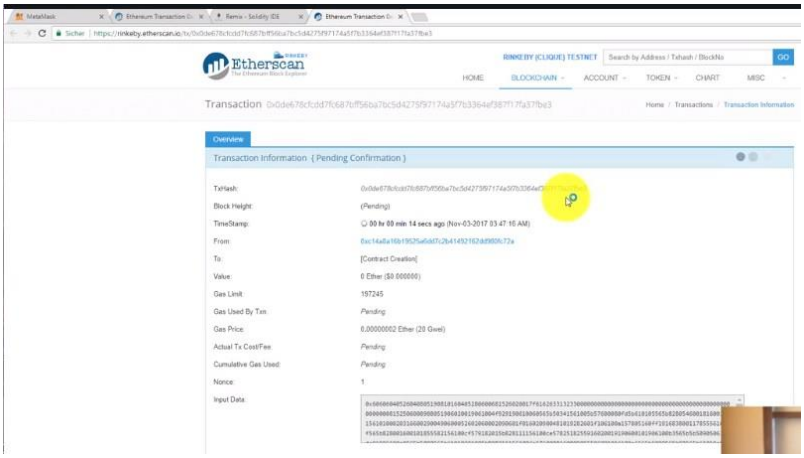
Выберем Injected Web3, затем счет - тот же самый, который открыт в надстройке MetaMask. Теперь можно перейти на вкладку Run и создать контракт.



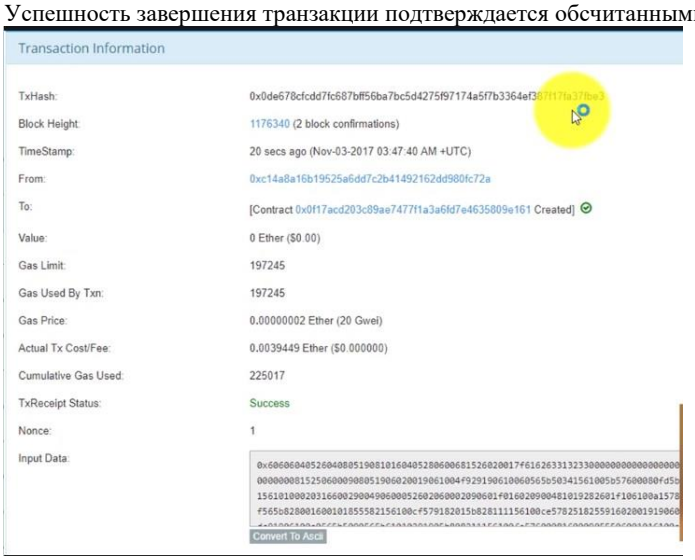
Отследим всю цепочку: контракт, написанный в среде Remix на языке JavaScript, исполняется, надстройка MetaMask отслеживает эту транзакцию и открывает всплывающее окно для подтверждения создания контракта.



Для этой процедуры потребуется израсходовать немного газа. После подтверждения MetaMask отправляет транзакцию в блокчейн.



MetaMask располагает некоторым числом узлов сети, расположенных на серверах разработчиков, которые играют роль посредника между вашим браузером и блокчейном. Если вы хотите создать и использовать свой собственный блокчейн, потребуется один из клиентов: Geth, *Parity*, Mist. Mist обеспечивает интегрированный в *браузер доступ* к блокчейну, а Geth - это работающий на вашем компьютере клиент, загружающий все блоки и предоставляющий к ним непосредственный *доступ*. Успешность завершения транзакции подтверждается обсчитанными блоками.



Основные понятия среды Ethereum

Блокчейн в среде Ethereum очень напоминает блокчейн для Bitcoin. Есть транзакции, есть эфир, подобный биткоину, есть майнеры и так далее. Однако есть и различия. Во-первых, майнеры все еще работают по механизму Proof-of-Work, то есть решают математические задачи по шифрованию данных. Это требует большого количества энергии и вычислительной мощности, а после успешного завершения к блокчейну добавляется новый блок. В скором будущем будет внедрен механизм Proof-of-Stake, который потребует приобретения определенного количества эфира, который будет использоваться для обсчета новых блоков.

Самым большим отличием от блокчейна Bitcoin является возможность размещать приложения непосредственно в блокчейне. Именно этой теме будет посвящен данный раздел.

Приложения с максимальной доступностью

Технология блокчейн подразумевает, что все приложения обладают невероятной устойчивостью и надежностью, поскольку исполняются непосредственно в блокчейне - единый *сервер* отсутствует, код выполняется на всех узлах блокчейна одновременно. Это открывает большие возможности для решения сложных вычислительных задач, а также построения высокораспределенных сред.

Основы работы среды Solidity

Среда Solidity представляет из себя высокоуровневый *язык программирования*. Существуют и другие способы разработки приложений для блокчейна, но общим правилом является необходимость компиляции текста на языке программирования в байткод, который затем размещается в блокчейн посредством транзакции. Такие транзакции очень похожи на используемые в блокчейне Bitcoin, когда вы отправляете биткойны с одного адреса на другой; в среде Ethereum для тех же целей используются единицы эфира. Если есть потребность отправить в блокчейн байткод, то он прикрепляется к транзакции как данные, а в самой транзакции при этом должно быть пустым *поле* получателя *To* - в этом случае блокчейн создаст новый *адрес* для размещения этого байткода.

Практический пример

Например, предположим, что у нас есть *функция ABC*, требующая *параметр* a. Если a меньше 50, *функция* возвращает 10, в противном случае a. *Компилятор* обрабатывает функцию, и в блокчейн отправляется новая транзация со следующими значениями полей: *from* содержит *адрес* отправителя - ваш *адрес*, *поле value* пусто, как и *поле to* (это самый важный момент), а *поле data* содержит байткод из функции, созданной в Solidity. В процессе обсчета транзакции этот код будет добавлен в очередной блок, получит собственный *адрес*, например, *0xabcdef001*, или какой-нибудь другой, и у каждого пользователя сети появится возможность взаимодействовать с этим кодом по заданному адресу.

Важность обсчета кода и учета валюты в одном блокчейне

Блокчейн представляет собой значительно распределенную базу данных. Это означает, что при сохранении в блокчейне какой-либо величины или участка кода их больше нельзя удалить, их доступность крайне высока. Ни какие-либо величины, ни участки кода не доступны из централизованного источника, поэтому никакому правительству не под силу ограничить *доступ* к ним или удалить данные из блокчейна, если только они не выключат все узлы сети по всему миру. Эта концепция напоминает сохраненные процедуры в MySQL, только применимо по отношению к коду. В частности, в среде MySQL вы можете запускать некоторые программы, изменяющие запросы *SELECT* или *RETURN* - аналогично в блокчейне, особенно в Ethereum, с помощью смарт-контрактов можно изменять значения переменных или данные, отправлять валюту и другим образом взаимодействовать с другими смарт-контрактами.

В среде Ethereum и обработка кода, и валютные *операции* проводятся в едином *поле*. Это открывает широкие возможности для различных приложений из областей условного депонирования, краудфандинга, страхового дела, операций с недвижимостью, сферы услуг, юриспруденции и прочих. В настоящее время наблюдается множество проектов, использующих возможности запуска кода в блокчейне и работы с криптовалютой с последующим проведением краудфандинговых кампаний.

Классические примеры распределенных приложений

Приведем несколько примеров распределенных приложений. Начнем с ДАО - демократических автономных организаций. Эта система представляет собой платформу для краудфандинга. Одно время широко обсуждалась в прессе, поскольку разработчики смогли привлечь шестьдесят миллионов долларов США в виде инвестиций. К большому сожалению, она впоследствии была взломана, но оставила значительный след в сознании людей, благодаря ясной логике и новому подходу к краудфандингу, при котором не представлялось возможным собрать средства и сбежать (в отличие, например, от Kickstarter, который тоже принимает средства для разработки новых продуктов, но нет гарантии, что он не обанкротится). В случае с ДАО отсутствует центральное передаточное звено, способное скрыться с деньгами, намеренно, поскольку все договоренности обеспечиваются смарт-контрактами.

Вторым распространенным типом приложений являются решения по обмену валюты. В настоящее время наблюдается всплеск числа ICO, - первичных размещений криптовалюты - реализуемых с помощью токенов стандарта *ERC*, которые можно напрямую обменивать в среде блокчейна, то есть менять токены на эфир или токены на токены, и вся эта логика хранится непосредственно в блокчейне.

Токены - это участки кода, связанные с пользовательскими учетными записями и базами данных и используемые как валюта, баллы в программах лояльности, индикаторы доли в компании, жетоны в играх виртуальной реальности и так далее. Все эти платежные средства учитываются и хранятся в среде блокчейна.

И, наконец, *базы данных*. Снабдив их некоторой логикой, можно вести в блокчейне учет владельцев земельных участков, дипломов об образовании (например, Массачусетский технологический институт выпускал свои сертификаты в

блокчейне), даже законов. В случае использования блокчейна нет необходимости прописывать положения закона в каком-то документе, вместо этого можно задать условия работы смарт-контрактов, сразу разрешая или запрещая какие-то действия в блокчейне. В перспективе можно даже прийти к автоматическому списанию средств со счета, например, в качестве штрафа за неправильную парковку.

Как работает доступ к блокчейну

Для обеспечения доступа к блокчейну используются узлы сети Ethereum, взаимодействующие друг с другом посредством протокола Ethereum. Каждый *узел сети* может обращаться к любому другому. Одним из узлов, доступных для свободной загрузки, является Go-Ethereum. Он, как и остальные реализации, подключается и взаимодействует с сетью посредством протокола Ethereum.

С другой стороны, для выполнения операций в блокчейне можно применять *удаленный вызов процедур* (*Remote Procedure Call, RPC*), запуская файлы *JSON*, созданные на JavaScript.

Удаленный вызов процедур можно реализовать через протокол *HTTP*, что позволяет взаимодействовать пользователям с узлами сети, а самим узлам - друг с другом посредством протокола Ethereum. Важно понять схему: *пользователь* работает с файлами *JSON* для удаленного вызова процедур, а узлы передают эту информацию между собой по протоколу Ethereum. Это напоминает работу в консоли MySQL, когда *пользователь* задает запросы MySQL (в случае с Ethereum отправляются команды для удаленного вызова процедур в формате *JSON*), а узлы сети MySQL обмениваются информацией по протоколу MySQL (в блокчейне узлы взаимодействуют по протоколу Ethereum).

Задание

1. Установить кошелек MetaMask. Подключаемся только к тестовой сети Rinkeby.
2. Откройте несколько счетов (минимум 3).
3. Получите эфир для дальнейшей работы.
4. Распределите полученный эфир между тремя счетами.
5. Ознакомьтесь с возможностями кошелька MetaMask.

Контрольные вопросы

1. Что такое Go-Ethereum?
2. Что такое Web3?
3. Что такое Remix?
4. Что такое Solidity?
5. Что такое MetaMask?
6. Как MetaMask взаимодействует с браузером?
7. Укажите несколько различий сети Ethereum и блокчейна Bitcoin?

Лабораторная работа №3

Знакомство с Remix - web-средой Solidity IDE.

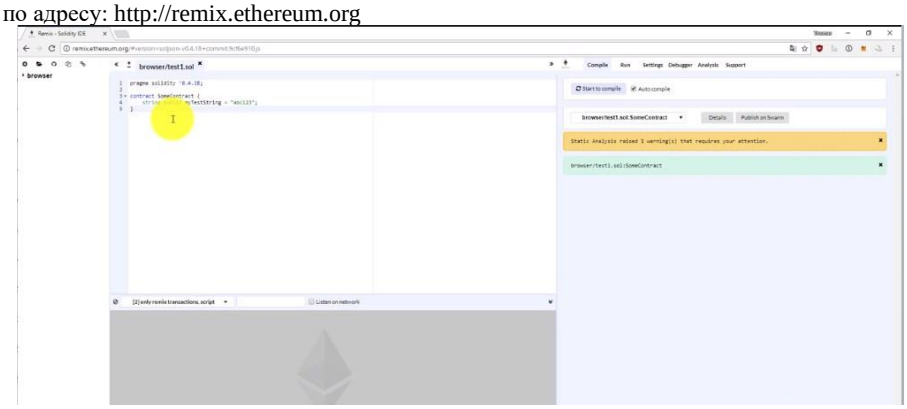
Цель работы: изучить и закрепить на практике возможности среды разработчика смарт-контрактов Remix.

Результат: практические навыки работы с инструментами среды разработчика смарт-контрактов Remix.

Теоретическая справка:

Обзор среды Remix

Откроем Remix по адресу: http://remix.ethereum.org

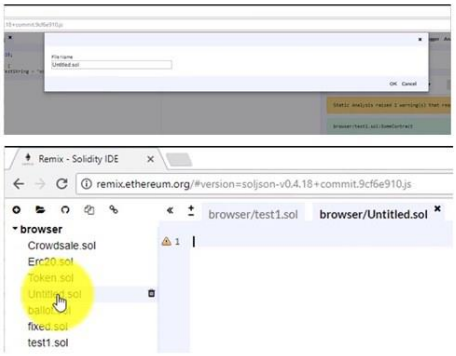


Remix - это интегрированная среда разработки, запускаемая непосредственно в браузере. Она поддерживает весь функционал, обычно ожидаемый от сред разработки, отличается тем, что работает в браузере, а также снабжена механизмом эмуляции блокчейна, в котором можно запускать распределенные приложения или контракты Solidity (смарт-контракты), а также проводить их отладку и тестирование.

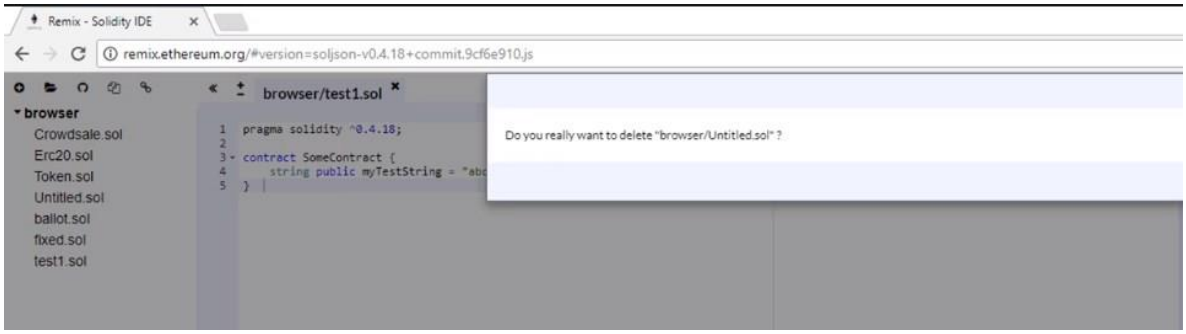
Как добавлять новые и просматривать существующие файлы в среде Remix

Обратимся к левой стороне Remix.

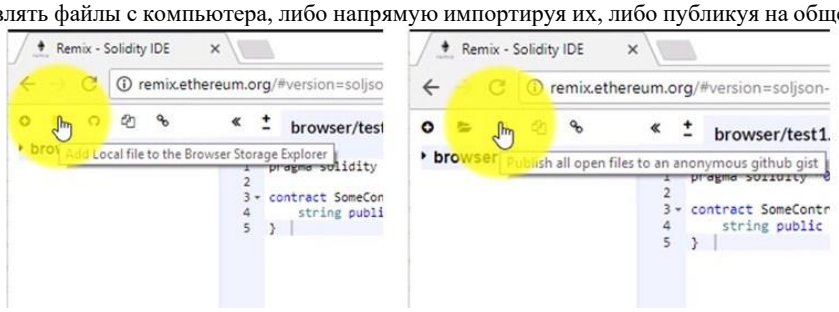
Здесь расположен файловый менеджер, с помощью которого можно создавать новые файлы, переименовывать их, после чего они становятся доступными.



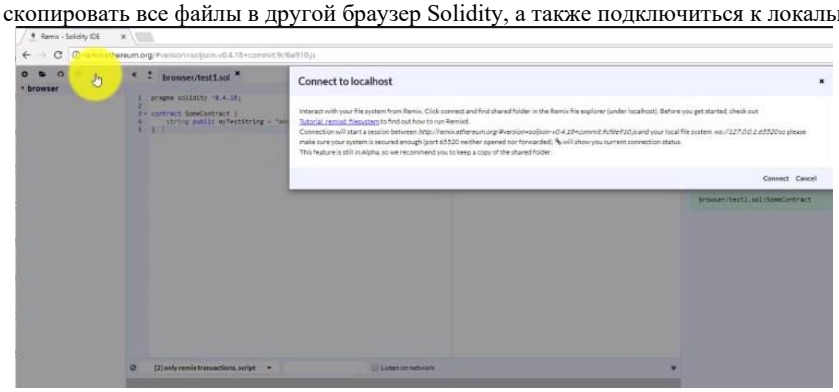
В главном окне после этого можно редактировать файлы. Закроем это окно и удалим файл.



Можно также добавлять файлы с компьютера, либо напрямую импортируя их, либо публикуя на общедоступном ресурсе.

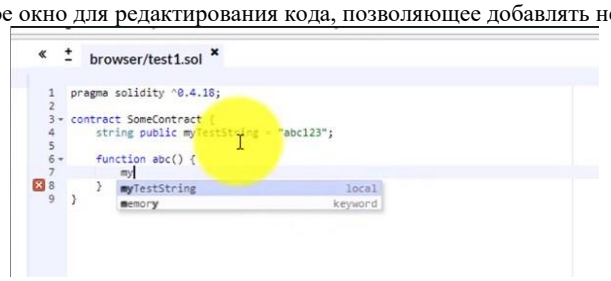


Кроме того, можно скопировать все файлы в другой браузер Solidity, а также подключиться к локальному хосту.



Главное окно редактирования

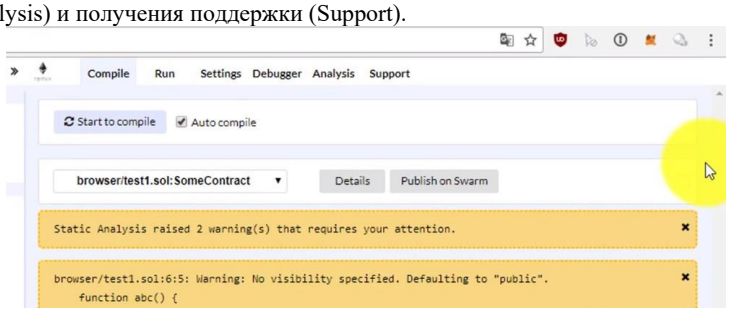
В центре расположено основное окно для редактирования кода, позволяющее добавлять необходимые функции.



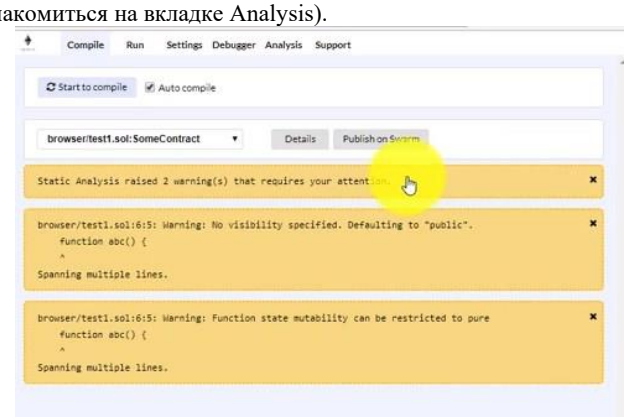
Кроме того, поддерживается функция автоматического дополнения - после набора нескольких символов система предлагает подсказки, например, что *myTestString* - это переменная.

Информация о контрактах на вкладке Compilation

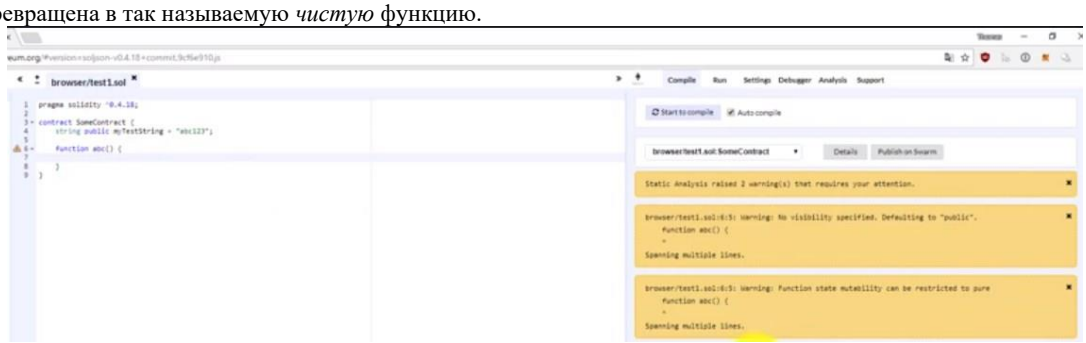
В правой части находятся несколько опций для взаимодействия со смарт-контрактом и получения информации о нем. Сверху расположены вкладки для компиляции (Compile), запуска (Run), изменения настроек (Settings), отладки (Debugger), анализа (Analysis) и получения поддержки (Support).



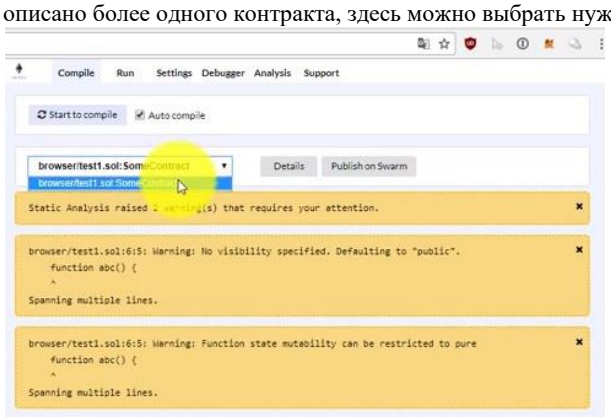
Вкладка Compile. В большинстве случаев при работе в среде Remix действует настройка автоматической компиляции Auto-Compile - код компилируется по мере ввода текста. Чуть ниже приводятся результаты статического анализа кода (подробнее с ними можно ознакомиться на вкладке Analysis).



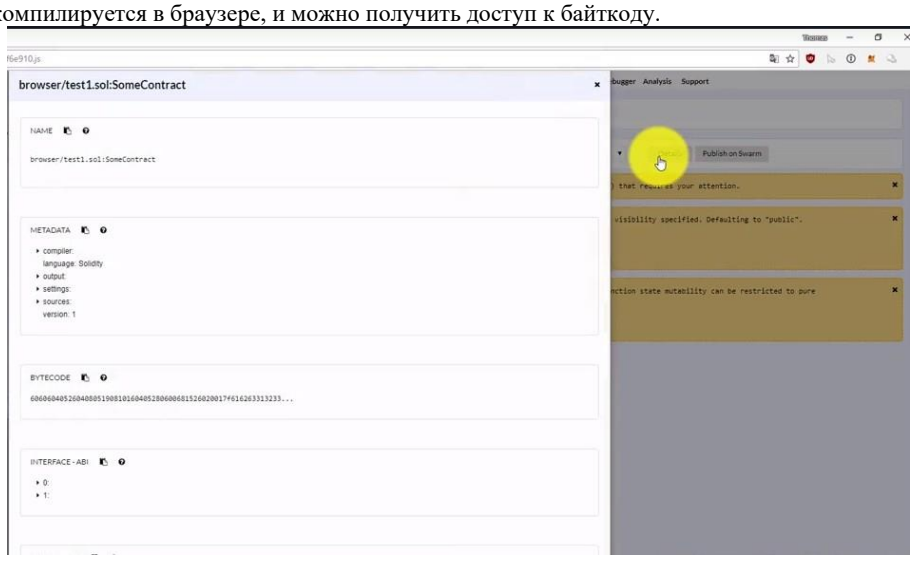
Например, для этой функции не объявлена область видимости, кроме того, она вообще не содержит инструкций и может быть превращена в так называемую *чистую* функцию.



В случае, если в файле Solidity описано более одного контракта, здесь можно выбрать нужный контракт



и получить подробную информацию о нем: название контракта, метаданные, байткод - так что видно, что контракт действительно компилируется в браузере, и можно получить доступ к байткоду.

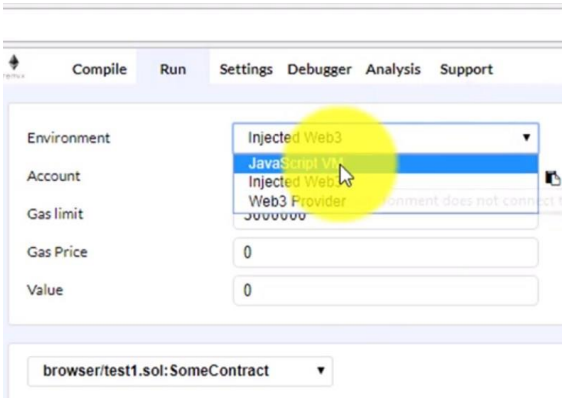


Кроме того, доступен двоичный интерфейс для приложений (API), можно скопировать участок кода в буфер обмена и использовать его для взаимодействия с контрактом посредством Mist или другого веб-кошелька для среды Ethereum.

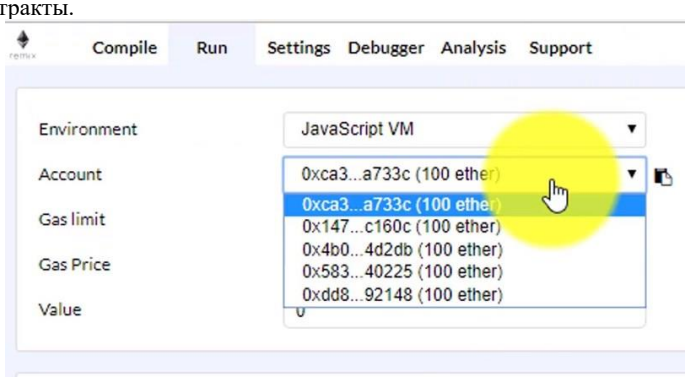
Также можно использовать этот код Web3 для непосредственной отправки контракта в Geth, и просматривать множество других сведений, например, коды операций или исполняемый байткод.

Все, что нужно знать о запуске файлов Solidity в среде Remix

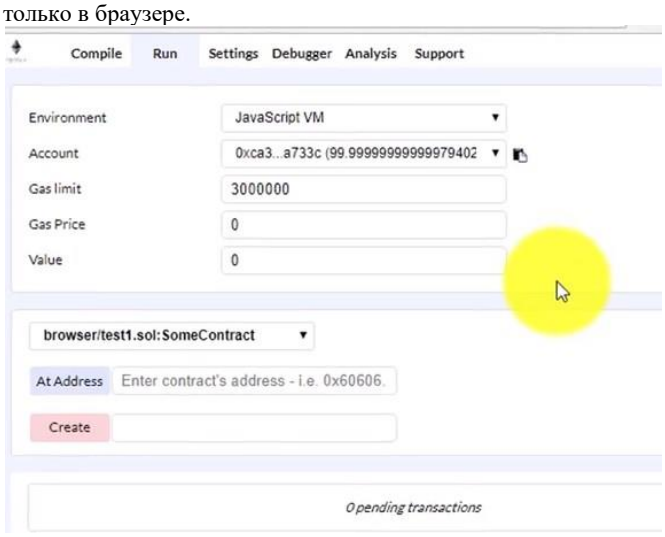
Вкладка Run. Это опция выбора среды запуска; виртуальная машина Java будет обеспечивать эмуляцию среды блокчейна прямо в браузере.



В эмулированной среде можно использовать несколько счетов, с сотней единиц эфира на каждом, и, конечно, создавать, размещать и запускать контракты.

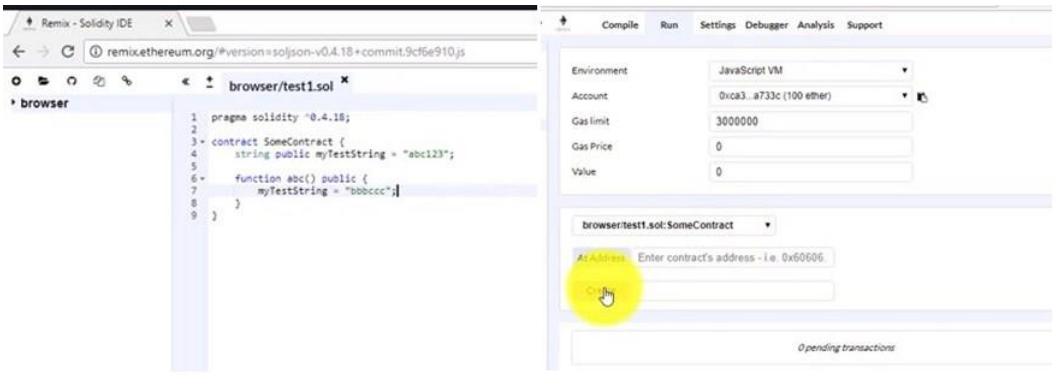


Все они будут работать в эмулированной среде блокчейна, то есть не будет подключения к MetaMask или реальному блокчейну - все происходит только в браузере.

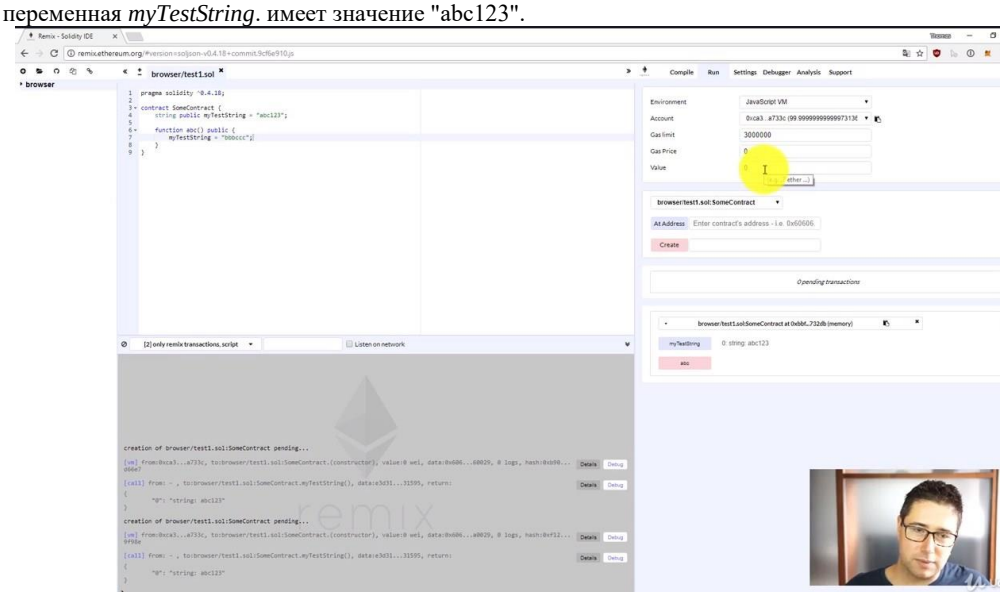


Если закрыть и открыть браузер, вся информация будет потеряна.

Для каждой транзакции, предусмотренной контрактом, можно установить свой предел расхода газа. Например, в функции *abc* (сделаем ее публичной) переменная *myTestString* меняет свое значение с "abc123" на "bbbccc". Теперь опубликуем контракт.



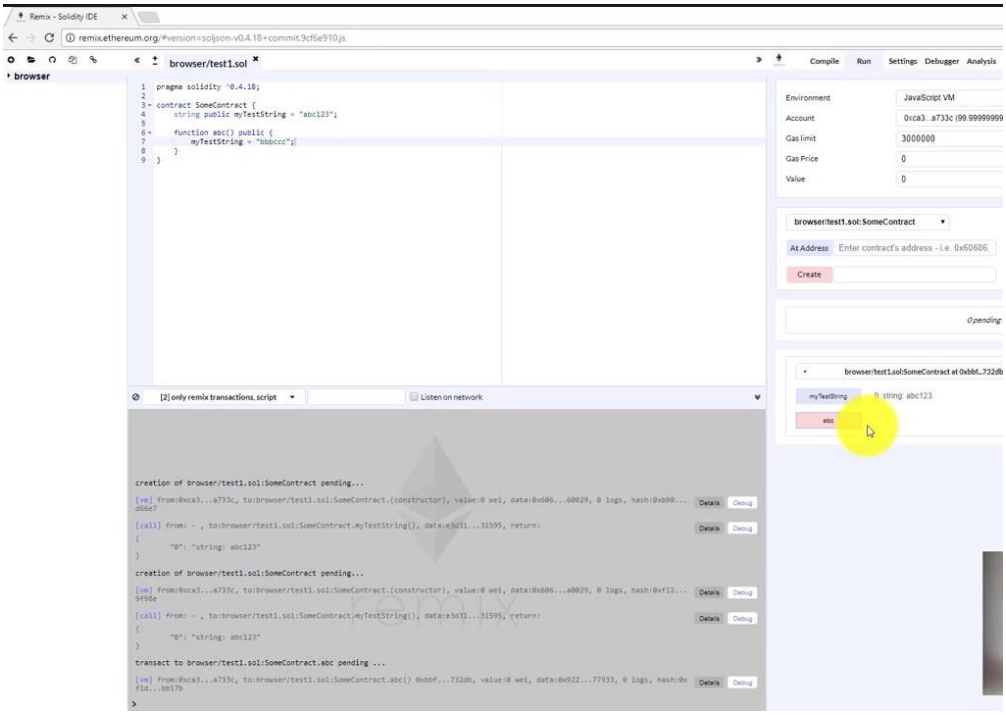
Видно, что переменная *myTestString*. имеет значение "abc123".



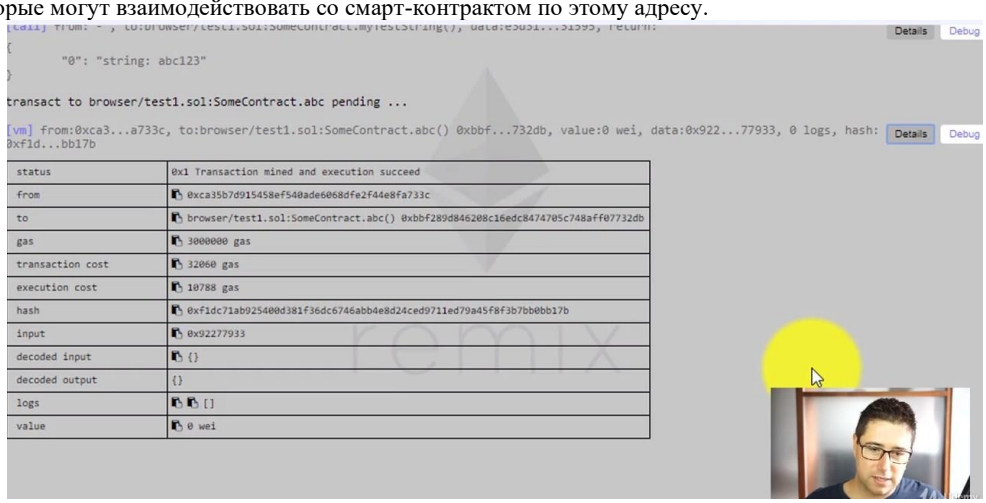
Затем можно вызвать функцию *abc*, щелкнув по ее названию. Для конкретной транзакции можно установить предел расхода газа, цену за единицу газа, а также отправить вместе с транзакцией некоторое количество валюты, например, единицу эфира.

Журнал событий в Remix

Выполним вызов функции. При вызове функции в нижней части окна отображаются все задействованные транзакции.



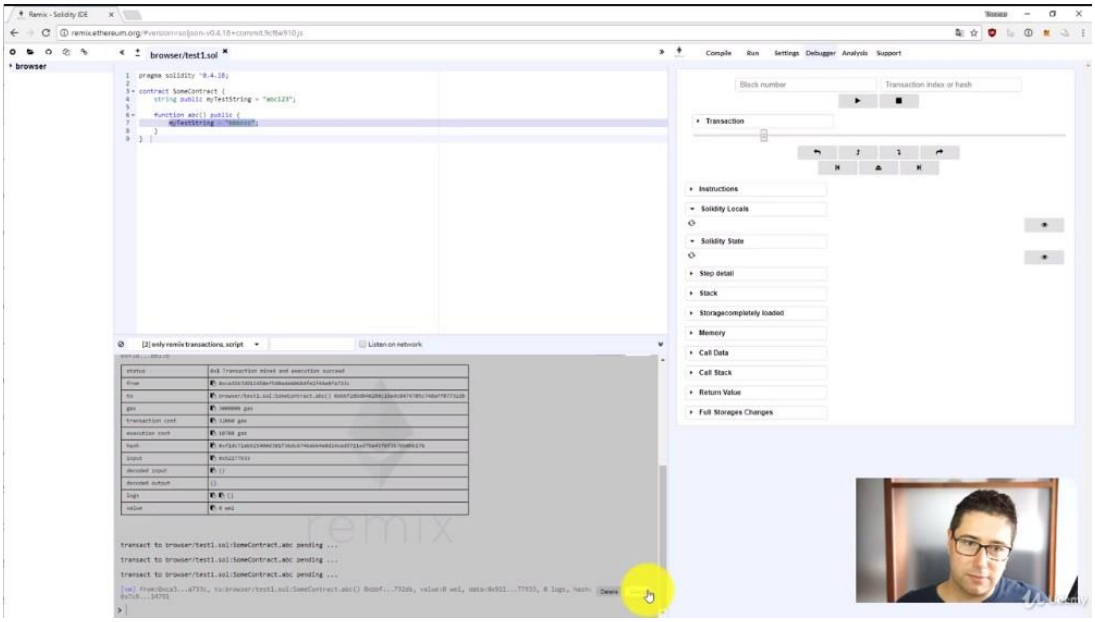
Сейчас транзакция ожидает обсчета, а затем она будет обработана сетью блокчейна. Также можно получить больше информации о транзакции. Для каждой конкретной транзакции можно увидеть, с какого адреса и на какой она была отправлена. Напомним, что сейчас все транзакции выполняются в эмулированной среде блокчейна, и существуют только внутри этого браузера. Можно увидеть установленный предел расхода газа. Также видны отправляемые в транзакции данные, которые могут взаимодействовать со смарт-контрактом по этому адресу.



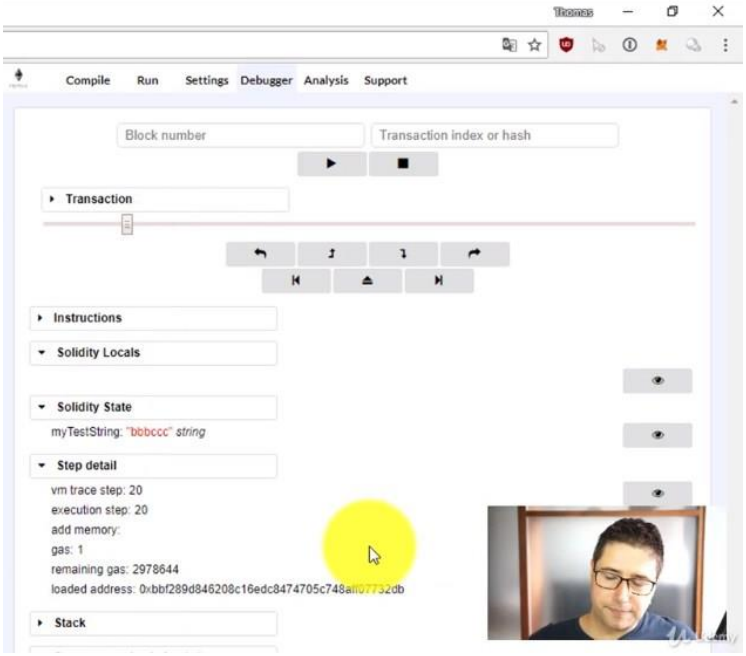
Инструментарий для отладки

В среде Remix отладчик (**вкладка Debugger**) поддерживает последовательную работу по шагам.

Очень удобно иметь отладчик, включенный непосредственно в код. Можно устанавливать точки прерывания или воспользоваться кнопкой отладки, чтобы увидеть, как проходит транзакция после отправки в сеть блокчейна.



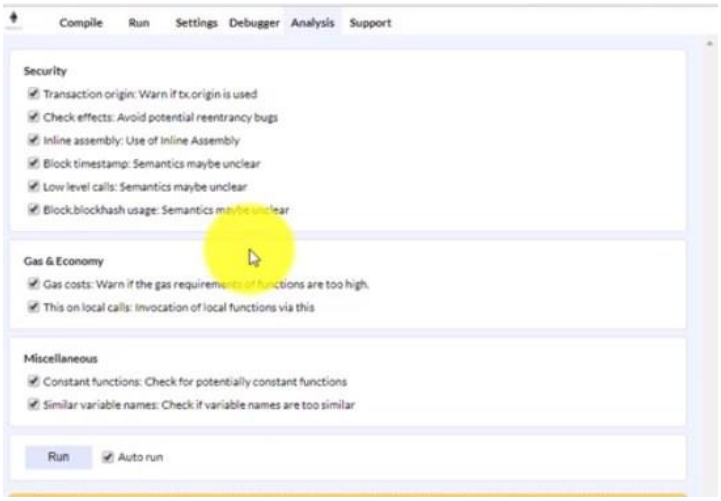
Этот функционал особенно удобен при работе со сложными функциями из большого числа шагов или циклов.



На этапе отладки можно узнать, что последняя операция кода не была выполнена или что кое-где необходимо заменить знак "меньше или равно" знаком "больше или равно" до отправки транзакции в реальный блокчейн.
Таковы типичные случаи применения отладчика. Иногда приходится иметь дело с обработкой исключений, в этих случаях отладчик позволяет быстрее установить участок возникновения исключения.

Краткий обзор инструментов для статического анализа

Вкладка Analysis содержит инструменты статического анализа.

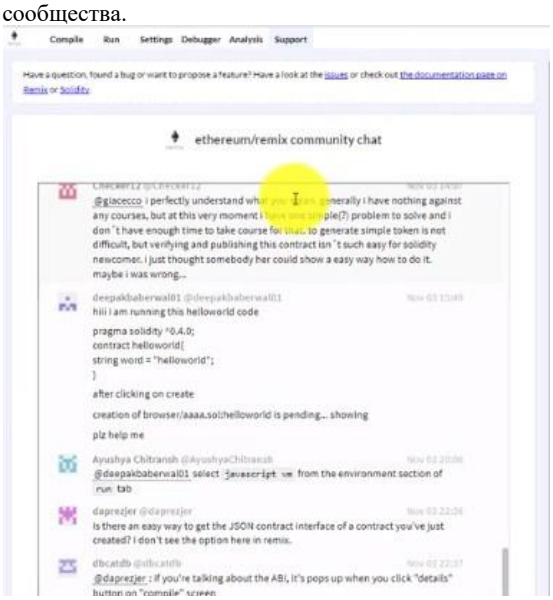


При статическом анализе код просматривается на предмет ошибок описания рабочей среды, и, в случае их нахождения, выводятся предупреждения. Например, для каких-то функций не описаны требования по расходу газа, или объявлены неиспользуемые переменные. В качестве предупреждений могут выводиться сообщения о том, что код использует временную метку блока, например, now, основанную на временной метке блока майнера, вместо временной метки блока из блокчейна.

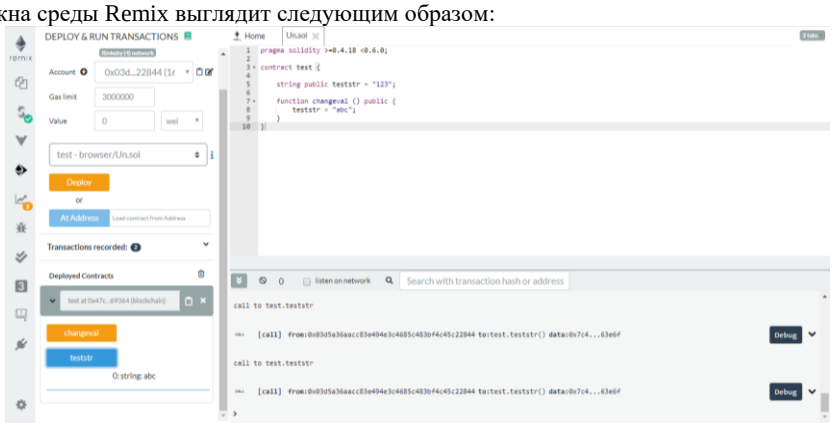
Таковы типичные предупреждения, выдаваемые инструментарием для статического анализа. Если вы уверены в своих действиях, их можно игнорировать, но зачастую они полезны при указании на опечатки или использование циклов, когда без них можно обойтись, поэтому в общем случае статическим анализом не стоит пренебрегать.

Помощь и поддержка

На **вкладке Support** можно найти прямые ссылки на чат сообщества Remix, где можно пообщаться на тему Remix или задать вопросы участникам этого сообщества.



Новый интерфейс окна среды Remix выглядит следующим образом:



Задание

1. Ознакомьтесь с возможностями, режимами работы и инструментами IDE Remix.
2. Для эксперимента используйте следующий простой смарт контракт:

```
pragma solidity >=0.4.18 <0.6.0;

contract test {

   string public teststr = "123";

   function changeval () public {
      teststr = "abc";
   }
}
```

Контрольные вопросы

1. Какие инструменты IDE Remix Вы знаете?
2. Что такое газ в Ethereum и зачем он нужен?
3. Что представляет собой смарт-контракт?

Лабораторная работа №4

Язык программирования смарт-контрактов Solidity

Цель работы: изучить и закрепить на практике основные синтаксические конструкции языка программирования Solidity.

Результат: файлы с примерами простых контрактов согласно задания.

Теоретическая справка:

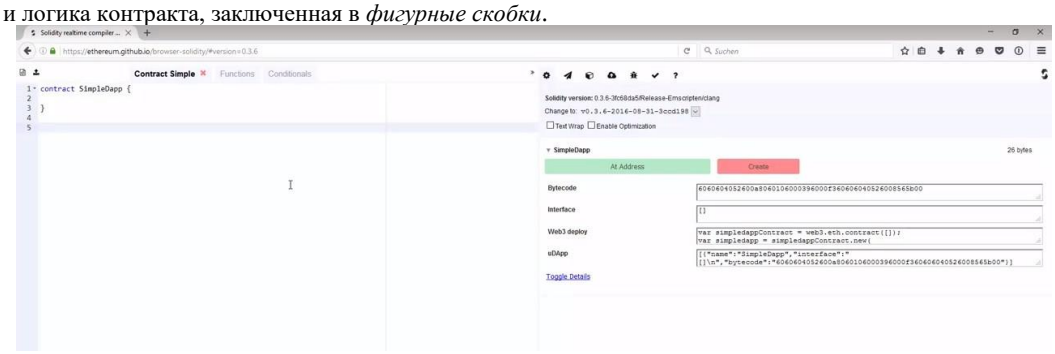
Эта работа полностью посвящена изучению среды программирования Solidity. Однако перед тем, как погрузимся в ее изучение, давайте поговорим о том, какого рода распределенное *приложение* нам предстоит написать. В ходе этого курса будем создавать распределенный веб-кошелек с высокой степенью надежности. Его *надежность* обеспечена самой структурой блокчейна. Распределенность же означает, что к кошельку возможен *доступ* нескольких пользователей, однако распоряжаться хранящимися в кошельке средствами можно только при наличии соответствующего доступа.

Как же работает распределенный кошелек? У такого кошелька есть некоторый баланс, и несколько пользователей могут подключаться к нему и отправлять средства на другие счета. Эти *операции* отслеживаются через транзакции в блокчейне. Рассмотрим подробнее на примере. В нашем кошельке пять тысяч единиц эфира, и *адрес 0xdeadbeef1* отправляет *запрос* на перевод одной тысячи единиц эфира на другой *адрес 0xdeadbeef2*. Кошелек должен будет проверить, что отправитель уполномочен распоряжаться хранящимся в кошельке эфиром. Если такие *права* есть, то на *адрес 0xdeadbeef2* отправится перевод тысячи единиц эфира. Кроме того, это событие будет зарегистрировано, о транзакции будет составлен отчет.

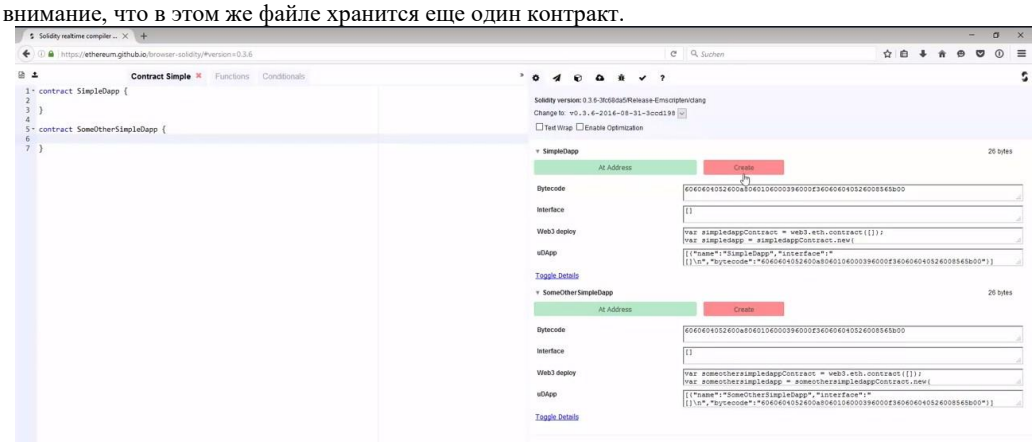
Для программирования данного примера в блокчейне будем использовать *язык программирования* высокого уровня Solidity. Solidity очень похож на JavaScript, но есть и различия. То, что в других языках называется классами, в среде Ethereum называется контрактами. Контракт может состоять из нескольких функций. Еще есть события, описывающие историю работы распределенного приложения, а также другие структуры, такие как библиотеки. Наш сравнительно простой распределенный веб-кошелек будет состоять из нескольких функций. Две функции будут реализовывать *управление учетными записями* пользователей - одна позволит добавлять адреса, уполномоченные распоряжаться эфиром в кошельке, а другая - отзывать *доступ* к кошельку. Специальная *функция* будет позволять отправлять эфир из кошелька другим пользователям сети; кроме того, должен быть и способ получать эфир, это будет реализовано так называемой функцией запасания эфира. Еще кошелек будет регистрировать и отправлять события трех видов. Одно событие будет срабатывать при отправке эфира, второе - при редактировании авторизованного на распоряжение эфиром адреса, а третье - при отзыве у такого адреса прав.

Классы контрактов, функций и условий

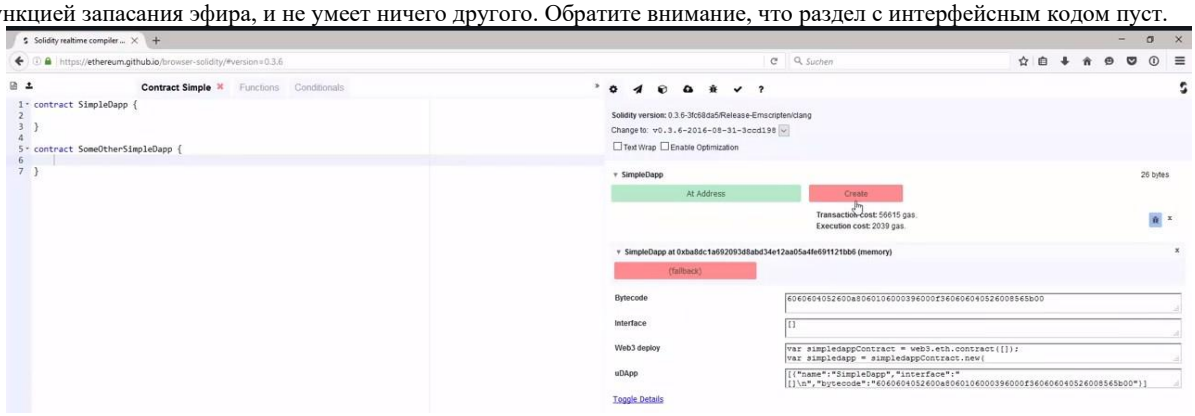
В этой работе будут раскрыты следующие вопросы: как создавать простые контракты, как выглядят структура и функции контракта, а также как использовать условия в среде Solidity. Начнем с простого контракта. Понятие контракта в среде Solidity похоже на понятие класса в других языках программирования. Контракт задается словом *contract*, а затем ему необходимо дать имя. Пусть это будет *SimpleDapp*. Итак, структура простого контракта - это имя, *служебное слово contract*, и логика контракта, заключенная в *фигурные скобки*.



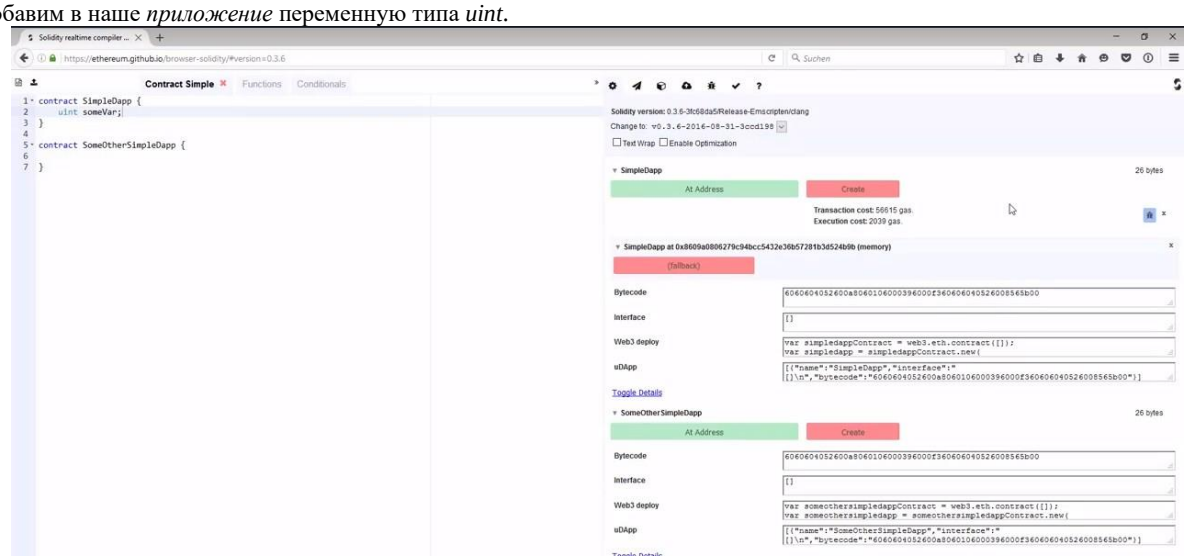
Обратите внимание, что в этом же файле хранится еще один контракт.



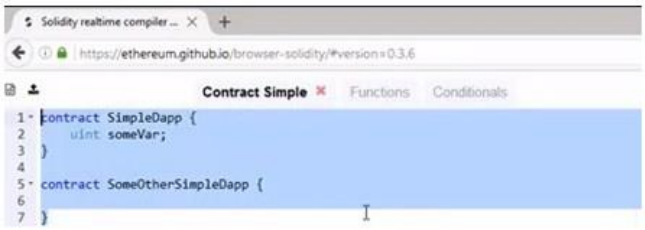
В правой части браузера Solidity видно, что *компиляция* и *отладка* контрактов происходят *по* мере их написания, что очень удобно. Можно создавать контракт. Ему сразу задается определенный *адрес*. Наш контракт снабжен только функцией запасания эфира, и не умеет ничего другого. Обратите внимание, что раздел с интерфейсным кодом пуст.

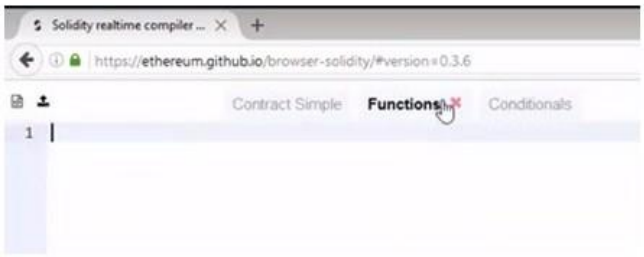


Добавим в наше *приложение* переменную типа *uint*.

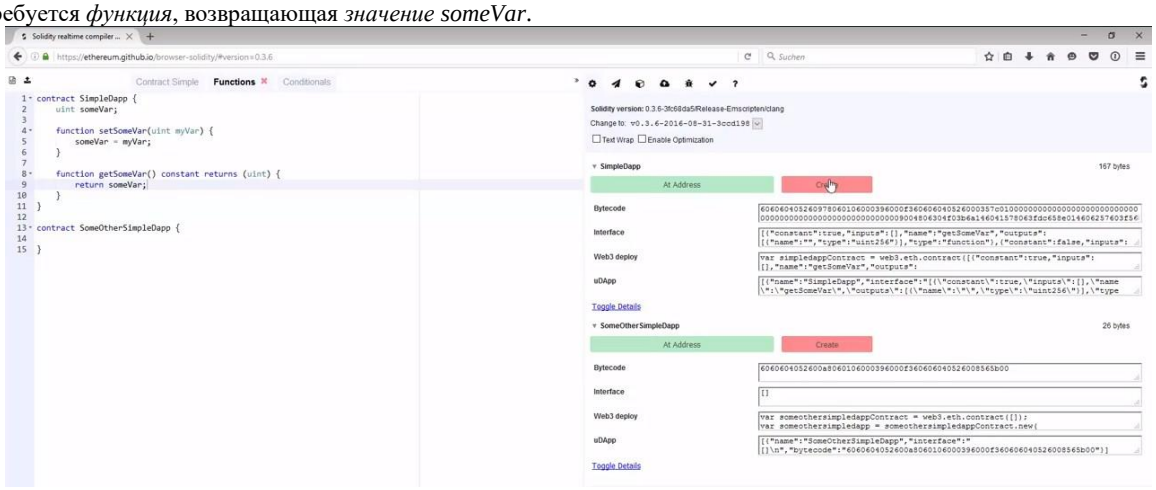


Если обратить внимание на раздел создания контракта в правой части браузера, можно заметить, что *доступ* к некоторым переменным *по* умолчанию ограничен. Чтобы взаимодействовать с этими переменными из других контрактов, необходимо задать ряд функций. Попробуем скомпилировать наш простой контракт и вставить его на вкладке *Functions*.

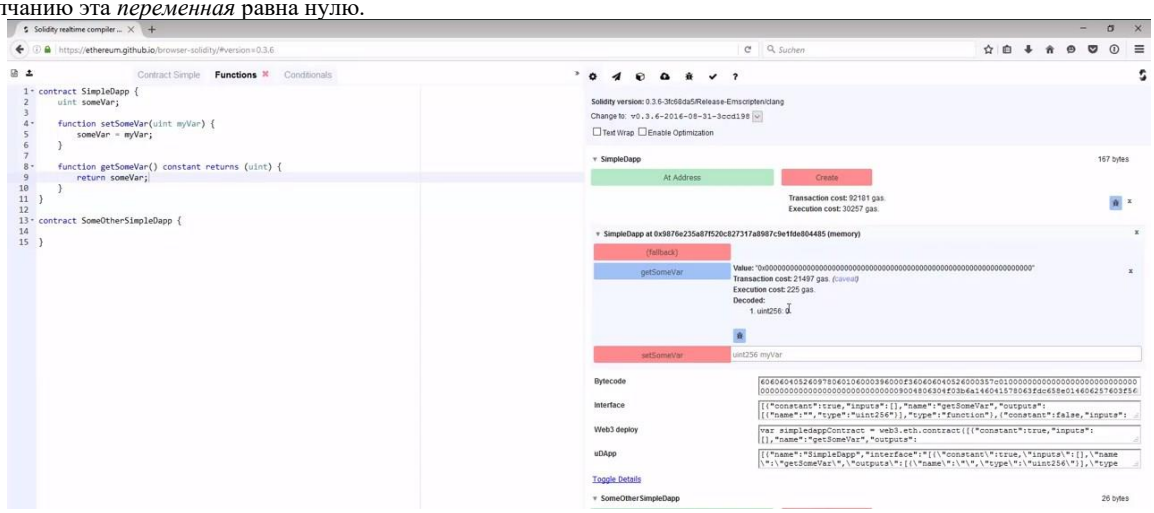




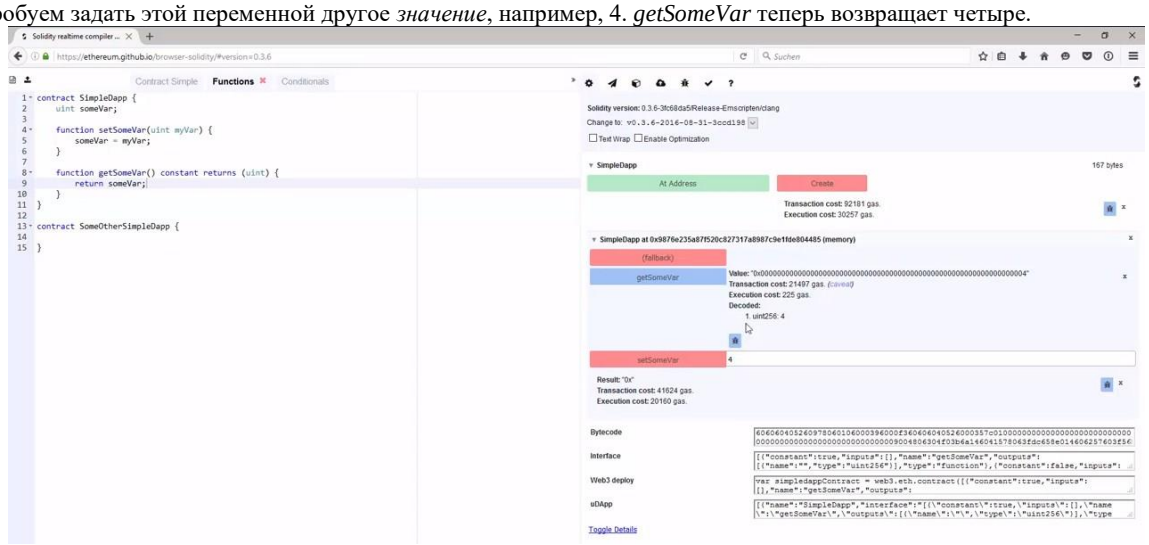
Создадим две функции для работы с этой переменной. В среде Solidity одни функции могут возвращать некоторые значения, а другие - нет. Зададим переменную *someVar*. Она будет брать *значение* другой переменной *myVar*. Еще потребуется *функция*, возвращающая *значение someVar*.



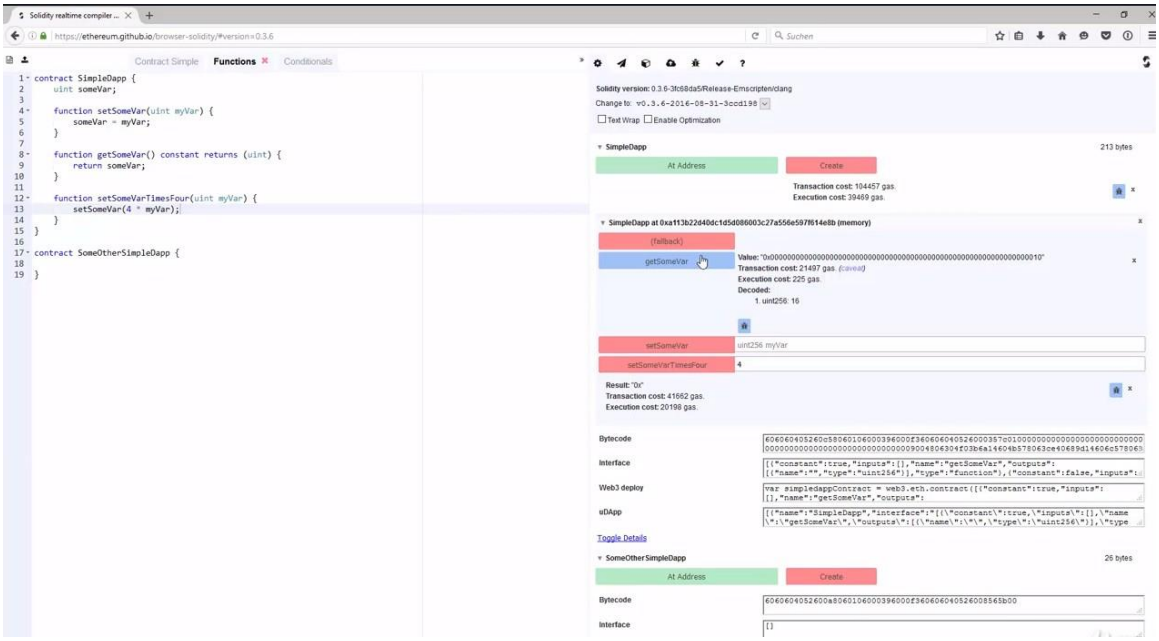
Все эти функции относятся к постоянным функциям, которые могут только возвращать значения. Они не могут изменять данные в блокчейне и поэтому для их обсчета не требуется тратить газ.
Посмотрим внимательнее, как работает *приложение SimpleDapp*. Сейчас *функция getSomeVar* возвращает наше число. *По* умолчанию эта *переменная* равна нулю.



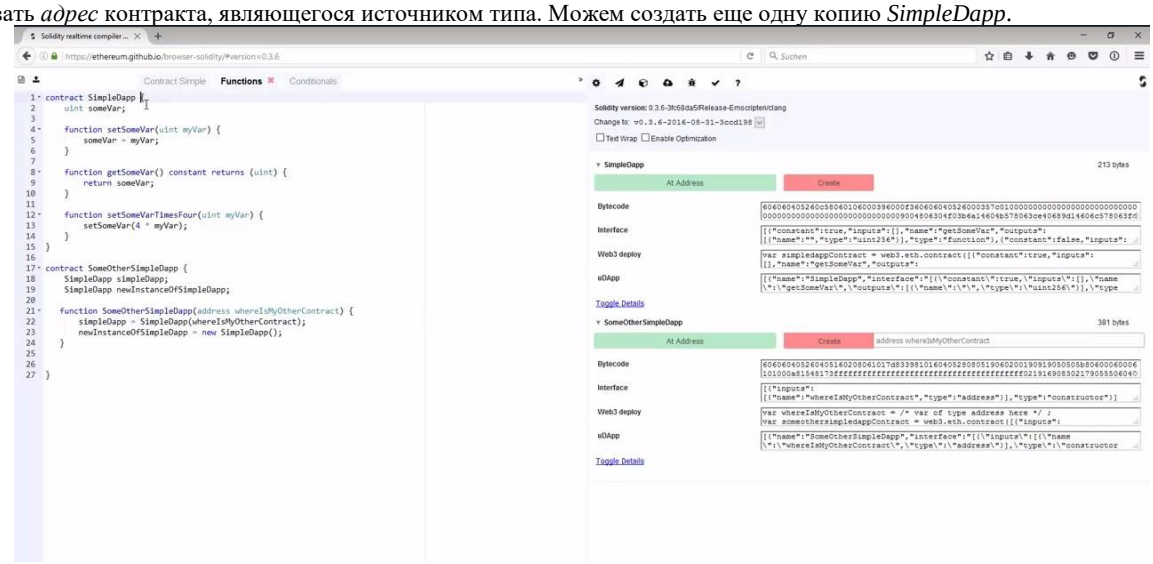
В среде Solidity переменные инициализировать не нужно, некоторые значения им заданы *по* умолчанию. Теперь попробуем задать этой переменной другое *значение*, например, 4. *getSomeVar* теперь возвращает четыре.



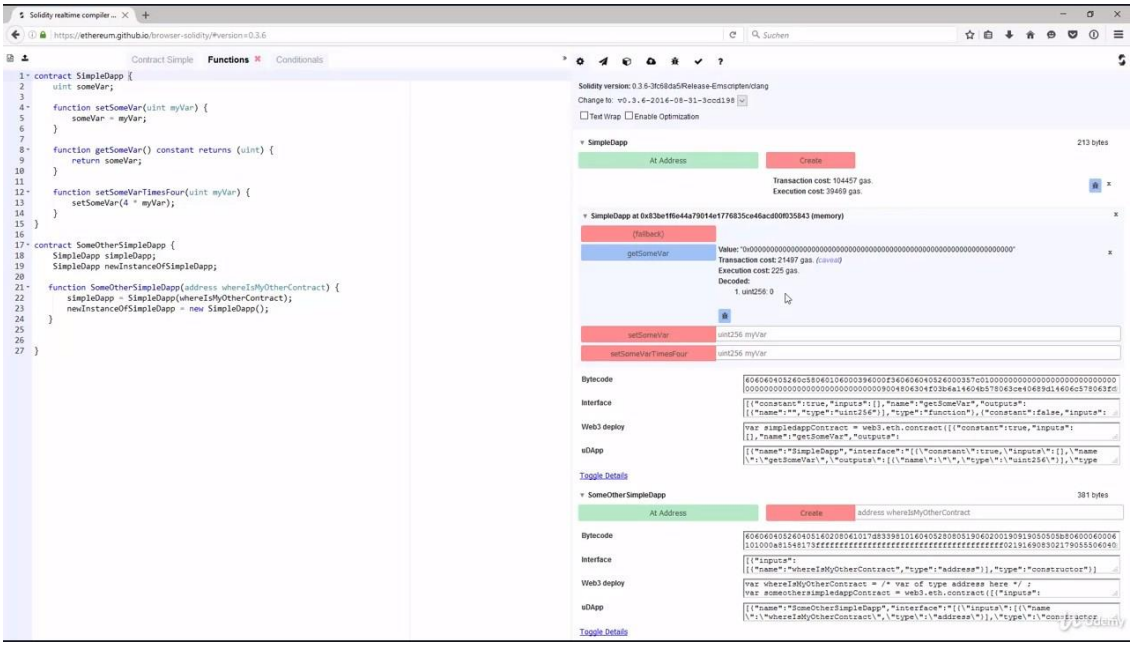
Работа с функциями в Solidity не отличается от других языков программирования. Например, можно задать другую функцию, которая будет умножать нашу переменную на четыре и задавать какое-нибудь *значение* другой переменной. Теперь *функция getSomeVar* возвращает шестнадцать.



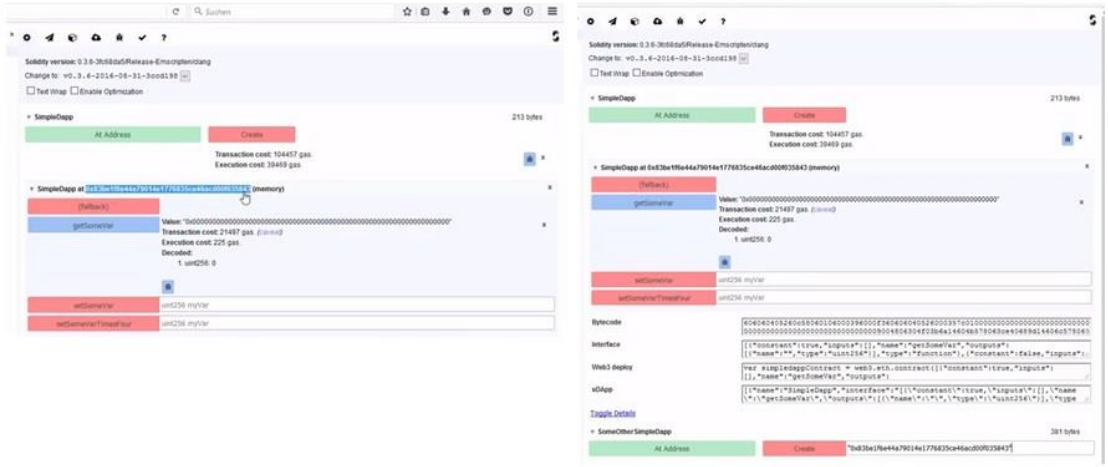
В среде Solidity содержимое контракта может обращаться к другому контракту. Давайте создадим контракт *SomeOtherSimpleDapp*, который будет возвращать *значение someVar* из контракта *SimpleDapp*.
Конструктор в среде Solidity работает примерно так же, как и любой другой *конструктор*. Он будет вызываться, когда контракт создается, а его *функция* называется так же, как и сам контракт. Вот простой *конструктор* для контракта *SomeOtherSimpleDapp*. Поработаем с контрактом *SimpleDapp*. Создадим переменную типа *SimpleDapp*, чтобы при ее инициализации во время создания контракта она сразу получила тип *SimpleDapp*. Это будет работать, если напрямую указать *адрес* контракта, являющегося источником типа. Можем создать еще одну копию *SimpleDapp*.



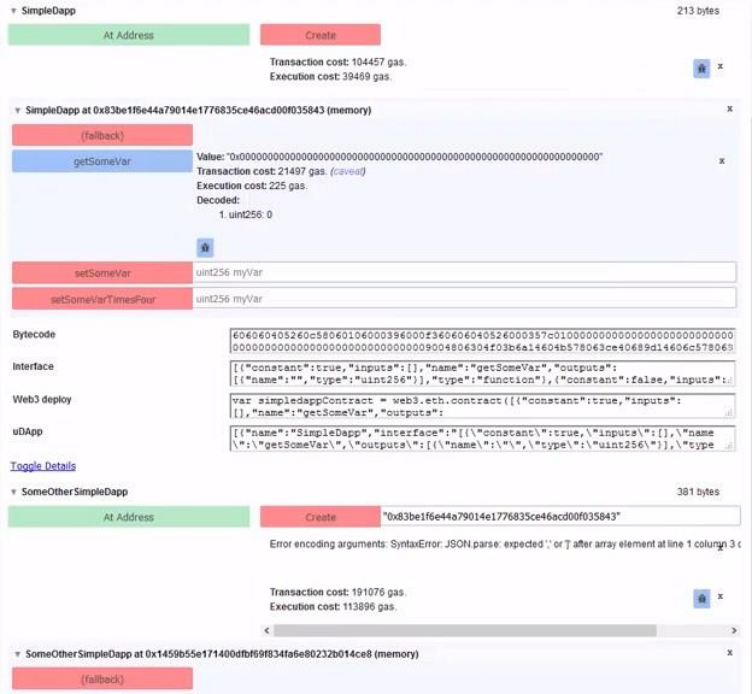
Посмотрим, как можно взаимодействовать с контрактом. В случае с первым контрактом *SimpleDapp* имеем дело со ссылкой на контракт, созданный выше. Как же это работает? Мы создали контракт *SimpleDapp*. У него есть некоторый *адрес*. Сейчас *функция getSomeVar* возвращает ноль, поскольку мы не указали конкретный *адрес*.



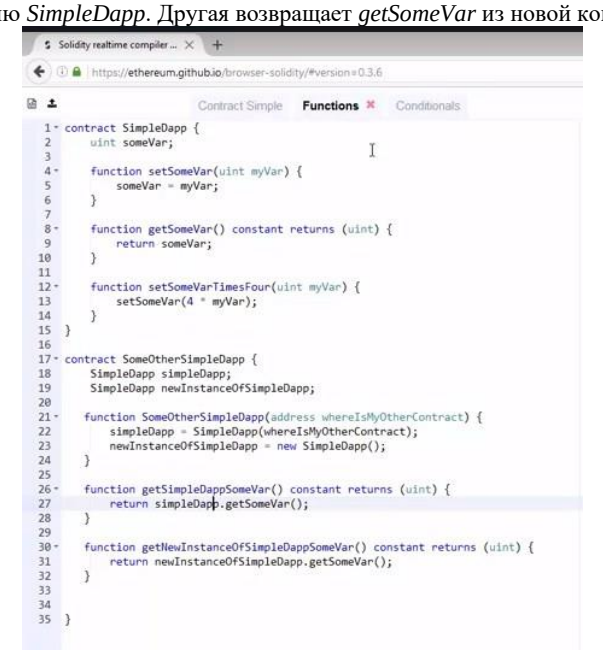
Скопируем *адрес* и создадим *OtherSimpleDapp*.



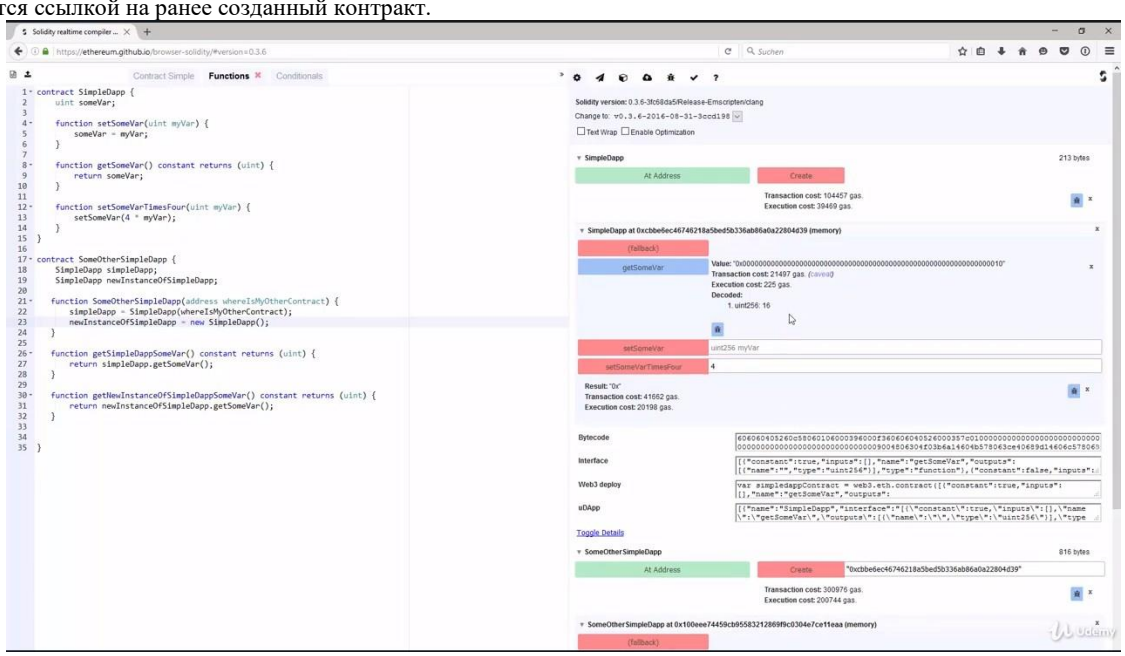
В контракте *OtherSimpleDapp* пока нет настоящих функций.



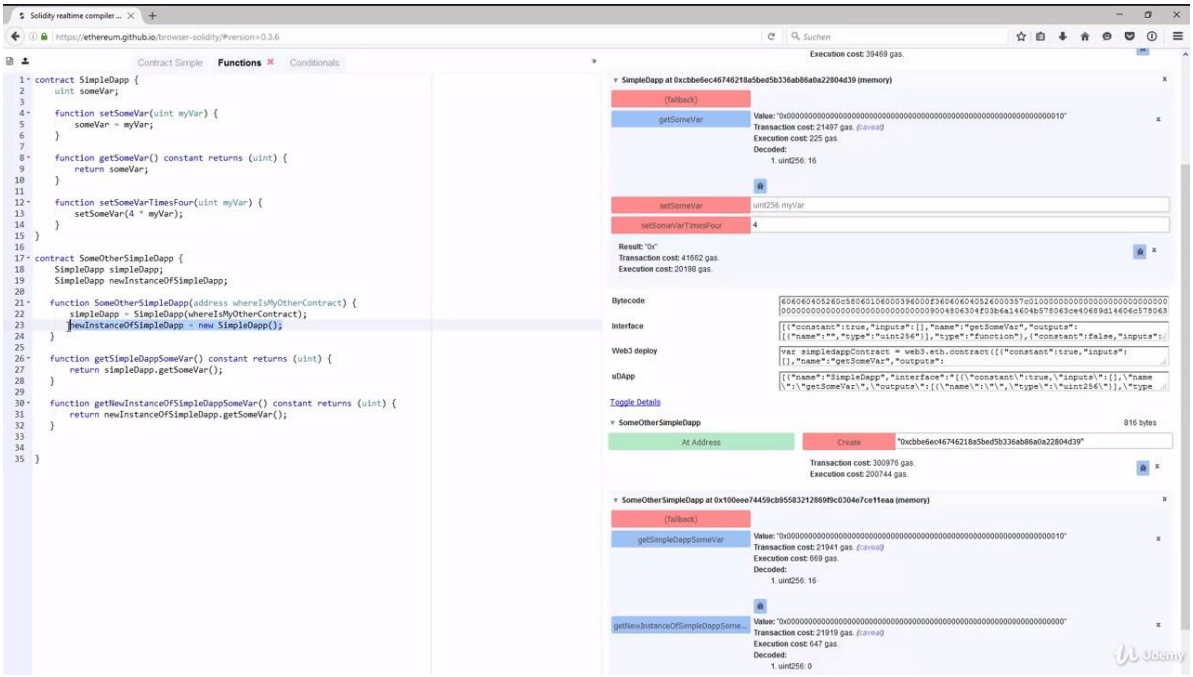
Давайте напишем пару функций для демонстрации работы ссылок. Одна возвращает *getSomeVar* из контракта *SimpleDapp*, что является ссылкой на у копию *SimpleDapp*. Другая возвращает *getSomeVar* из новой копии *SimpleDapp*.



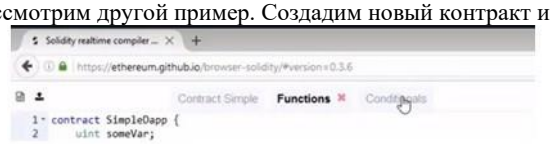
Создадим *SimpleDapp* еще раз. Теперь изменим *значение someVar* в первом контракте *SimpleDapp*, пусть это будет шестнадцать. Видно, что *функция getSomeVar* из первой копии контракта возвращает шестнадцать. *Функция GetSimpleDappSomeVar* из другого контракта, ссылающегося на контракт выше, тоже возвращает шестнадцать, поскольку является ссылкой на ранее созданный контракт.

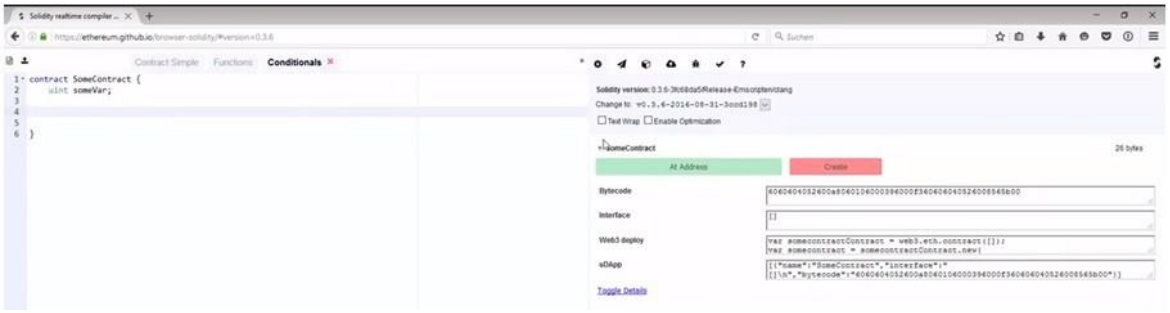


А другая *функция по*-прежнему возвращает ноль, поскольку она ссылается на новую копию контракта.

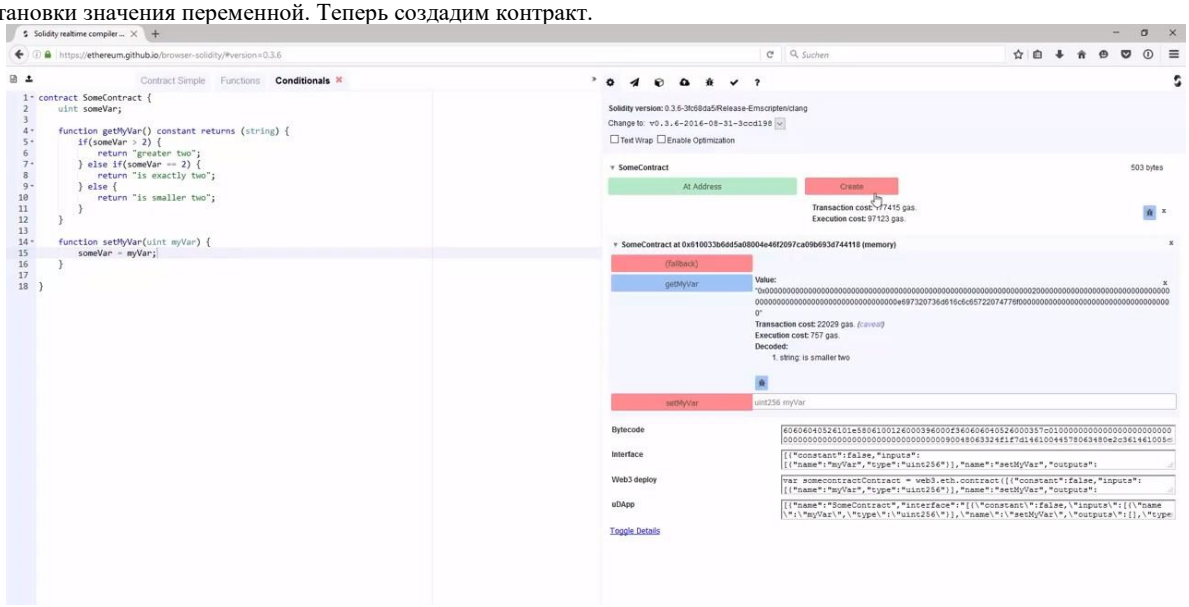


Рассмотрим другой пример. Создадим новый контракт и новую переменную.

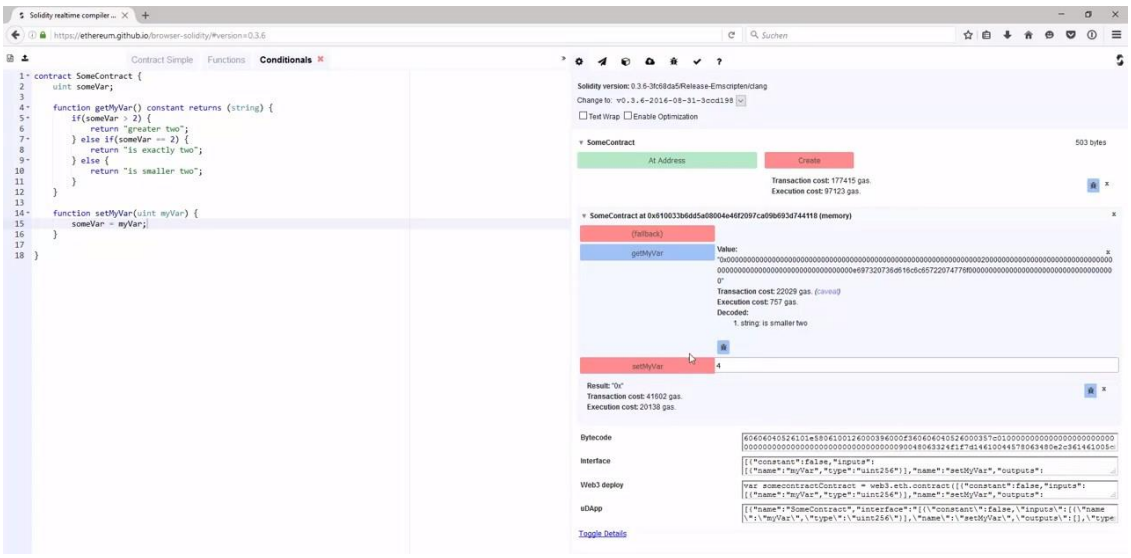




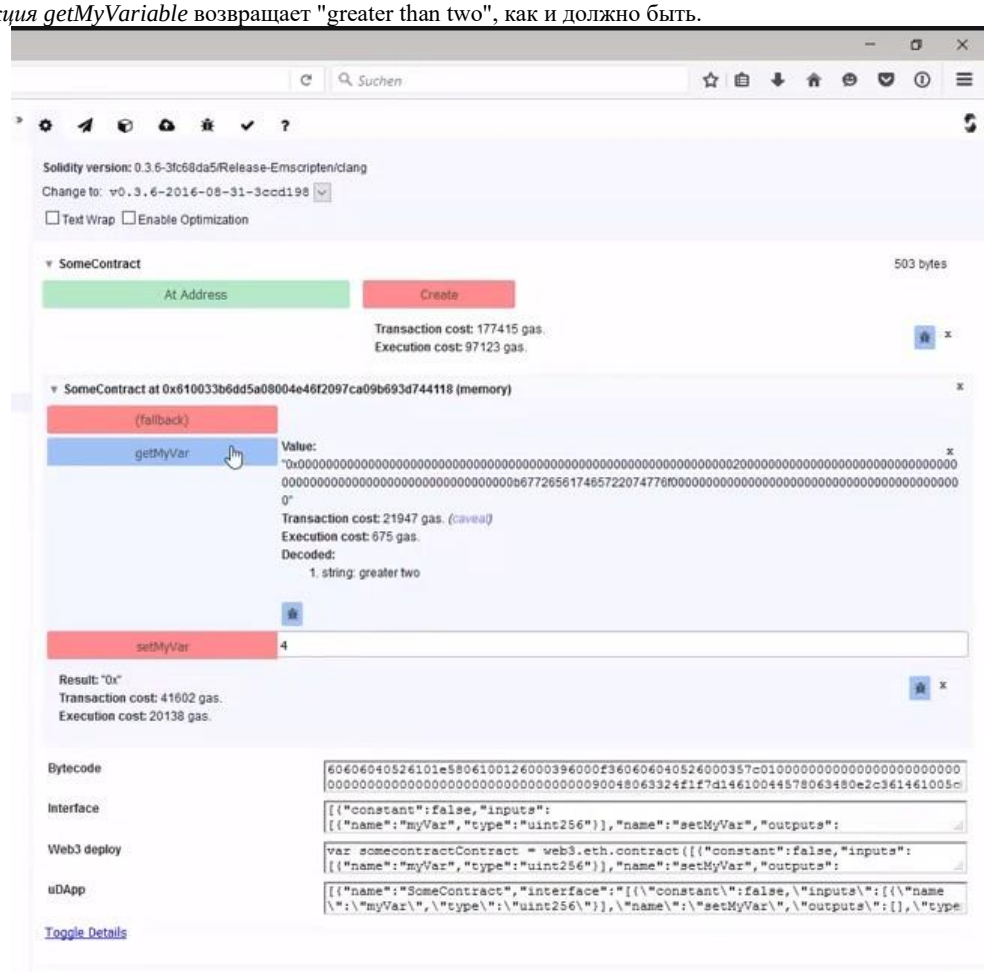
А теперь применим условие и вернем какое-нибудь *значение* для случаев, когда *значение* переменной равно двум, меньше двух или больше двух. Для ввода этого условия пригодится *служебное слово if*. Создадим еще одну функцию для установки значения переменной. Теперь создадим контракт.



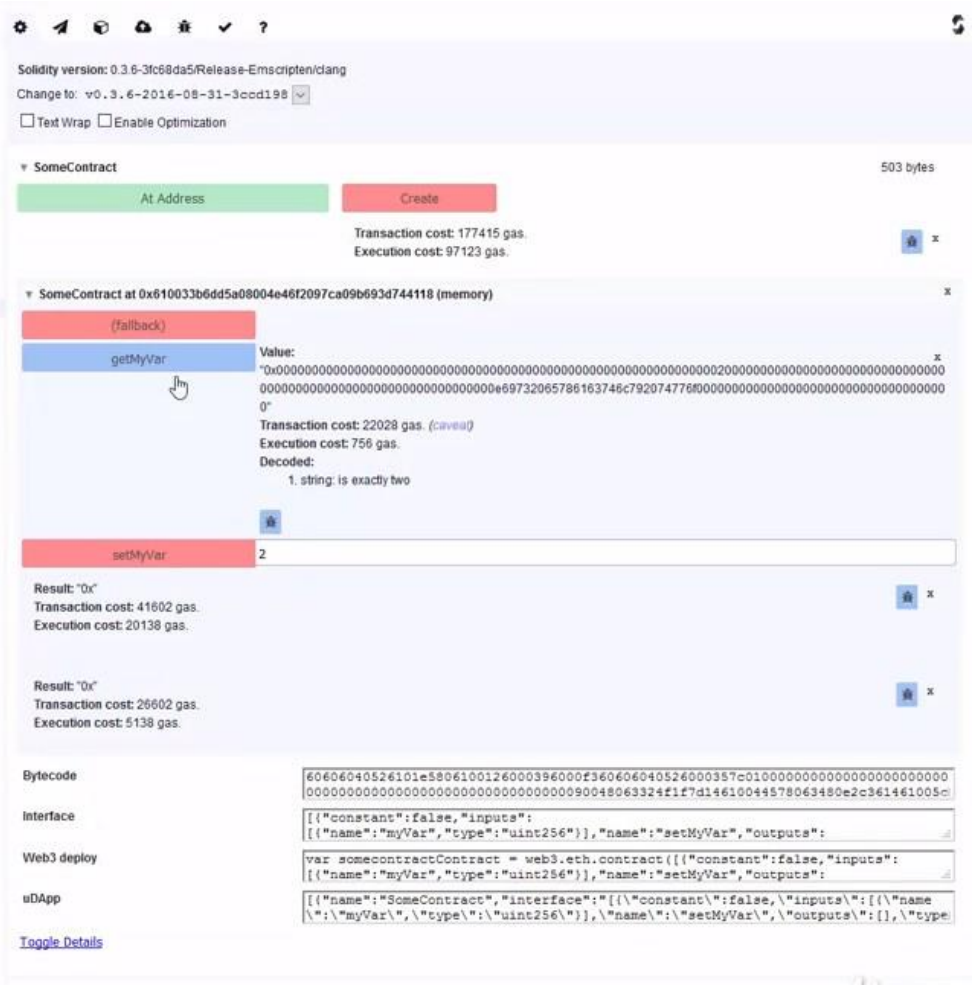
Изначально *переменная someVar* имеет *значение* ноль, что, конечно, меньше двух. Изменим *значение* на четыре.



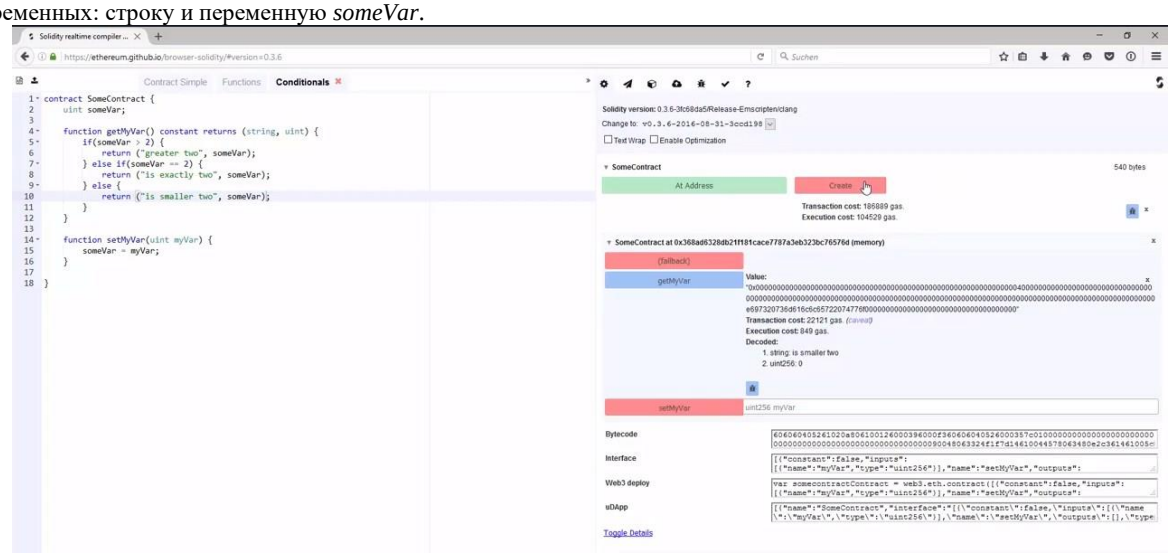
Теперь *функция getMyVariable* возвращает "greater than two", как и должно быть.



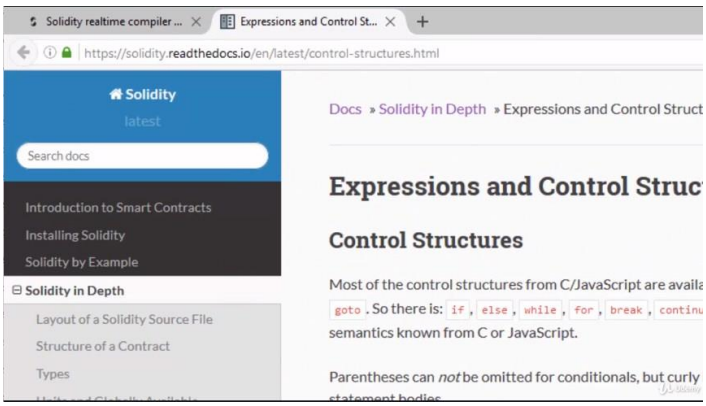
Изменим *значение someVar* на два. Теперь *getMyVariable* возвращает "is exactly two".



Можно задавать функции, которые будут возвращать больше одной переменной. *Функция getMyVar* возвращает две переменных: строку и переменную *someVar*.

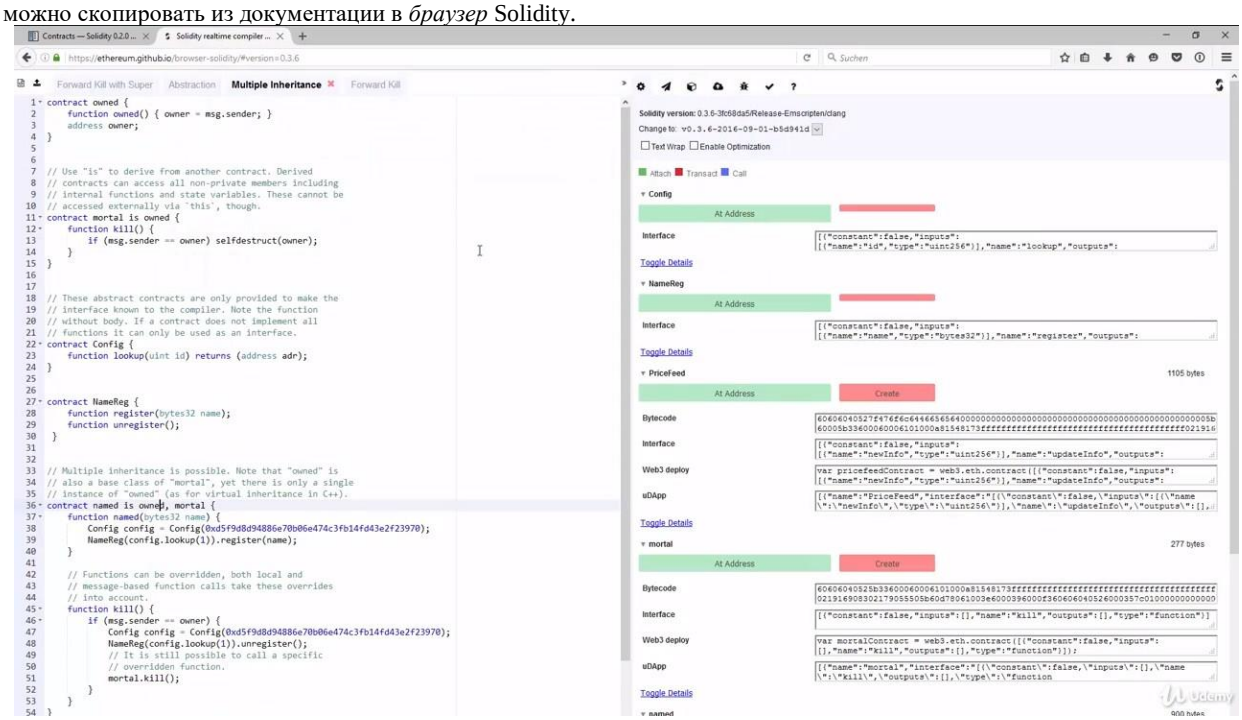


Помимо условной функции *if*, в среде Solidity есть и другие составные структуры, например, *else, while, for, break, continue, return* и *short if*. Чтобы узнать больше о структурах контроля, рекомендуется ознакомиться с документацией Solidity на http://solidity.readthedocs.io. Там хранится огромное число примеров и вариантов применения среды Solidity.

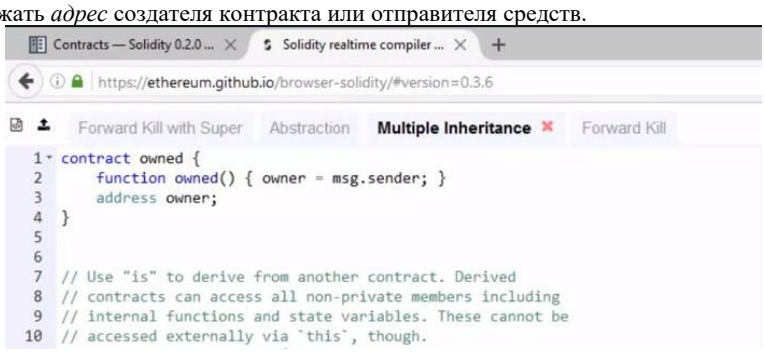


Наследование и абстрактные контракты

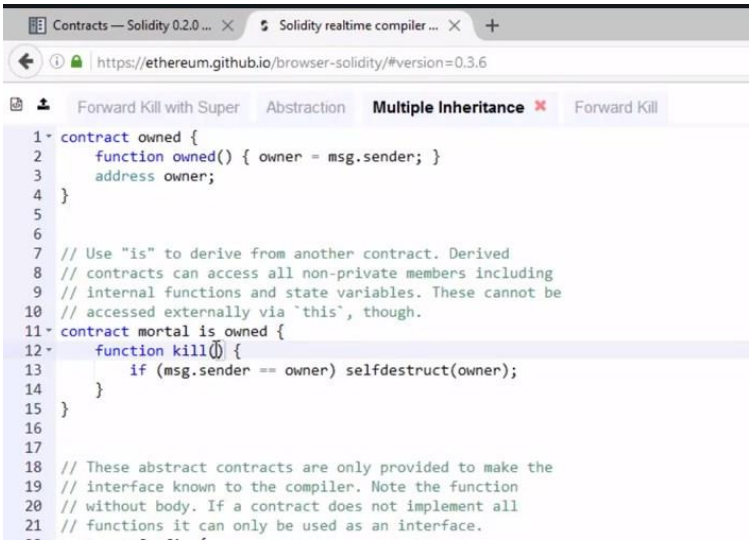
В рамках этой работы поговорим о наследовании в среде Solidity. Solidity поддерживает *множественное наследование*. Подробно рассмотрим примеры из документации Solidity: три на *наследование* и один на *конструирование*. Примеры можно скопировать из документации в *браузер* Solidity.



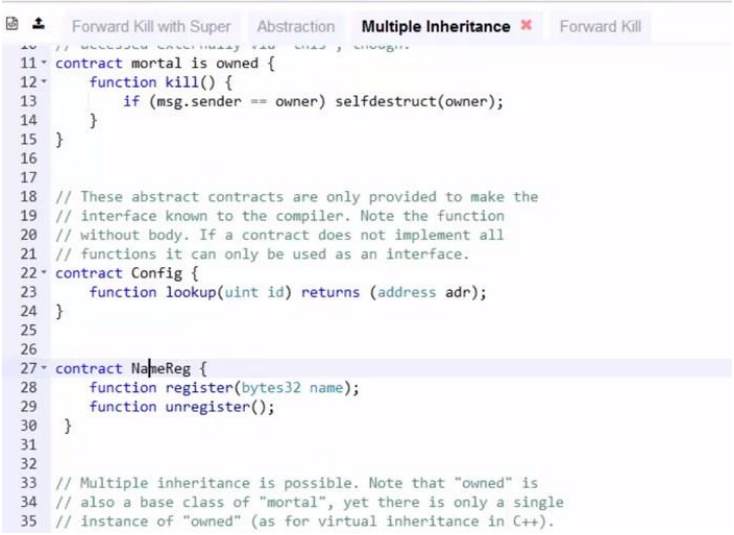
Простой контракт *Owned*. В нем есть *конструктор*, у конструктора есть владелец, а *адрес* владельца устанавливается равным адресу отправителя сообщения. *Объект* сообщения играет особую роль в среде Solidity и имеет несколько значений. Одно из них - это отправитель. Если кто-то отправляет в контракт средства или создает новый контракт, *объект* msg.sender будет содержать *адрес* создателя контракта или отправителя средств.



В контракте *Owned конструктор* устанавливает *адрес* владельца равным адресу отправителя сообщения. Посмотрим на контракт *mortal*. Этот контракт является дочерним *по* отношению к контракту *owned*. Он обладает дополнительной функцией *kill*. Когда контракт *mortal* выполняется, он автоматически обращается к контракту *owned* и переменной *owner*, хранящей *адрес* пользователя, выполняющего код. При вызове функции *kill* из контракта *mortal* будет произведена проверка, является ли *пользователь*, а именно *отправитель сообщения*, msg.sender, владельцем, и в случае успешной проверки контракт самоуничтожится и вернет все оставшиеся средства владельцу.



В этом примере два интерфейса, или два абстрактных контракта. В зависимости от ситуации их можно рассматривать *по-*разному. В данном случае будем считать их интерфейсами. Интерфейсы выглядят так же, как абстрактные контракты. У них есть имя контракта, а также задана *функция*, но тела функции нет. Это относится и к контракту *NameReg*.



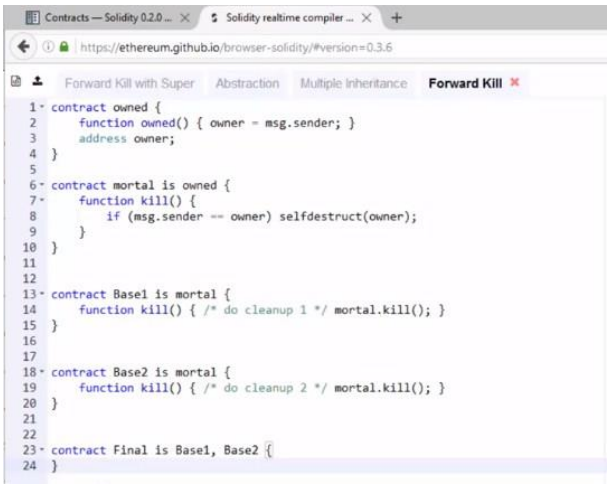
Рассмотрим контракт named. Он является дочерним сразу для двух контрактов, что демонстрирует *множественное наследование* в среде Solidity. Сначала наследуются свойства основного контракта, а затем - контракта *mortal*. Контракт *mortal*, в свою *очередь*, является дочерним *по* отношению к контракту *owned*. Контракт *named* - это очередной *конструктор*, который требует имя в качестве входных данных и выполняет некоторые *операции*. Он также содержит функцию *kill*. Эта *функция* переписывает функцию *kill*, унаследованную от контракта mortal. Но *функция kill* выполняет несколько другие *операции*, чем *функция kill*, унаследованная от контракта *mortal*. В итоге она носит название mortal.kill.

```
25
26
27  contract NameReg {
28      function register(bytes32 name);
29      function unregister();
30  }
31
32
33  // Multiple inheritance is possible. Note that "owned" is
34  // also a base class of "mortal", yet there is only a single
35  // instance of "owned" (as for virtual inheritance in C++).
36  contract named is owned, mortal {
37      function named(bytes32 name) {
38          Config config = Config(0xd5f9d8d94886e70b06e474c3fb14fd43e2f23970);
39          NameReg(config.lookup(1)).register(name);
40      }
41
42      // Functions can be overridden, both local and
43      // message-based function calls take these overrides
44      // into account.
45      function kill() {
46          if (msg.sender == owner) {
47              Config config = Config(0xd5f9d8d94886e70b06e474c3fb14fd43e2f23970);
48              NameReg(config.lookup(1)).unregister();
49              // It is still possible to call a specific
50              // overridden function.
51              mortal.kill();
52          }
53      }
54  }
55
56
57  // If a constructor takes an argument, it needs to be
58  // provided in the header (or modifier-invocation-style at
59  // the constructor of the derived contract (see below)).
```

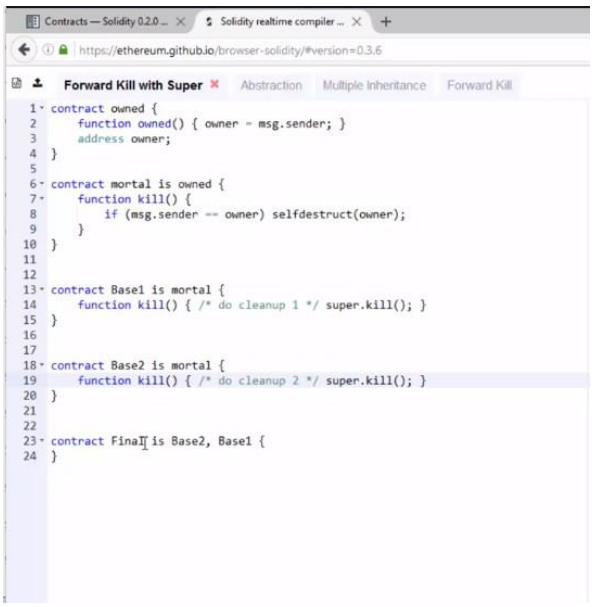
Рассмотрим следующий контракт *PriceFeed*. Он является дочерним сразу для трех контрактов. Один из них - это *owned*, основной контракт. Второй - это *mortal*, а третий - *named*, напрямую задающий имя. Если *конструктор* требует *аргумент*, например, контракт *named*: имя контракта, имя конструктора - это требование должно быть упомянуто в заголовке.

```
55
56
57  // If a constructor takes an argument, it needs to be
58  // provided in the header (or modifier-invocation-style at
59  // the constructor of the derived contract (see below)).
60  contract PriceFeed is owned, mortal, named("GoldFeed") {
61      function updateInfo(uint newInfo) {
62          if (msg.sender == owner) info = newInfo;
63      }
```

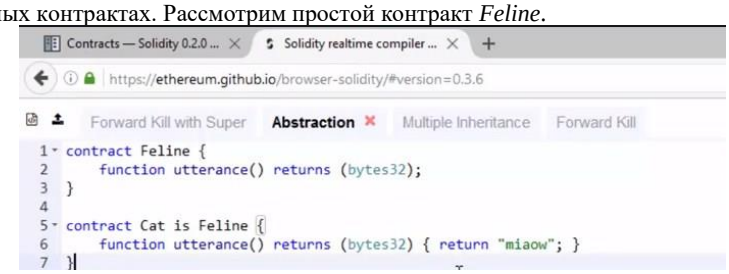
В контракте *PriceFeed* есть и другая логика. Посмотрим, что произойдет, если другой контракт вызовет *PriceFeed* и его функцию *kill*. Вернемся к контракту owned с конструктором. Есть контракт *mortal*, являющийся дочерним для *owned*, а также *функция kill*. Кроме того, есть контракты *Base1* и *Base2*. Оба они являются дочерними для *mortal*, оба наследуют функцию *kill* и вызывают в конце унаследованную процедуру mortal.kill. Контракт *Final* является дочерним для *Base1* и *Base2*.



Что произойдет, если другой контракт вызовет функцию Final.kill Она вызовет функцию Base2.kill, а до контракта *Base1* цепочка вызовов так и не дойдет. Эту проблему можно обойти с помощью служебного слова *Super*. Мы имеем дело с двумя классами, *Base1* и *Base2*, и вместо вызова функции mortal.kill будем вызывать super.kill. Этот способ вызова гарантирует, что если какой-либо другой контракт будет вызывать final.kill, то вначале он вызовет функцию *kill* из контракта *Base1*, Base1.kill, а затем - следующий контракт в графе наследования, то есть *Base2*. Поэтому будет вызвана *функция* Base2.kill, то есть *вызов функции* super.kill обеспечивает *вызов функции* mortal.kill.



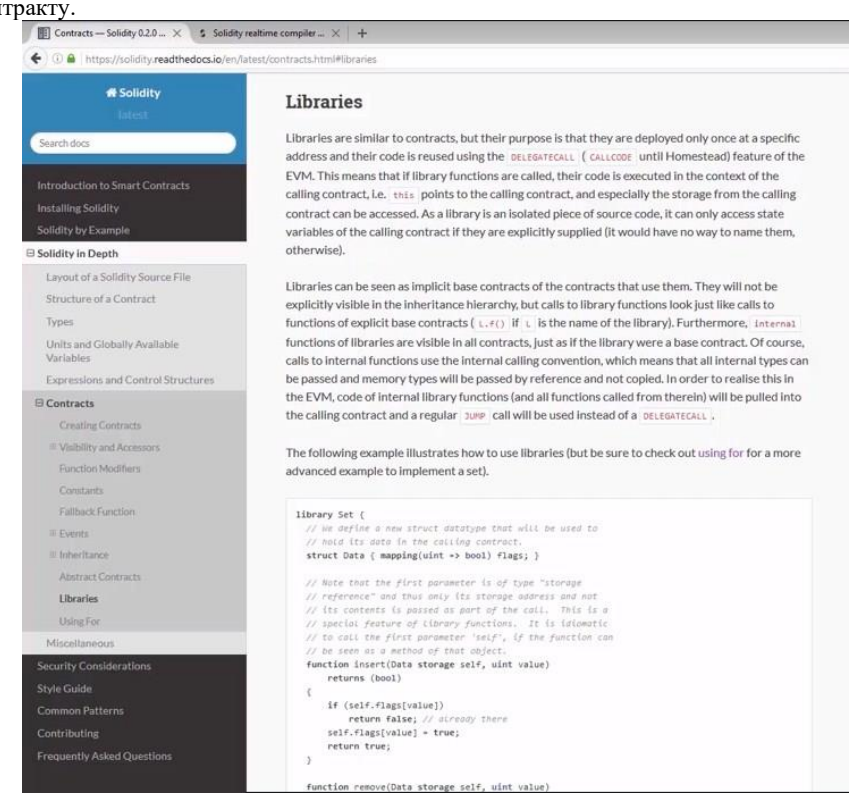
Поговорим об абстрактных контрактах. Рассмотрим простой контракт *Feline*.



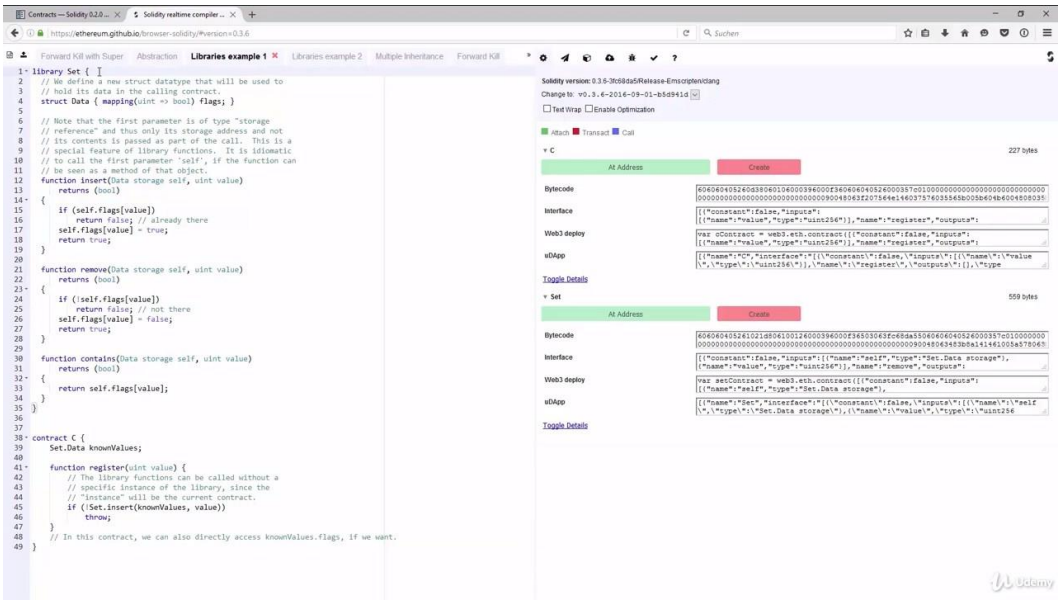
Это абстрактный контракт, поскольку в нем есть *функция* с пустым телом функции. Другой контракт, *Cat*, является дочерним *по* отношению к *Feline*. Он содержит ту же функцию, но в отличие от *Feline* она содержит *тело функции* и возвращает некоторое *значение*. Контракт *Feline* скомпилировать не удастся, а контракт *Cat* - наоборот, удастся.

Библиотеки и работа с ними

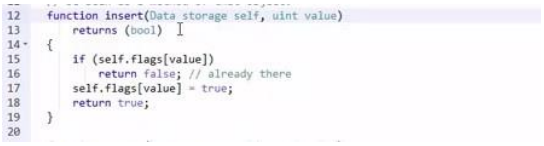
Библиотеки очень похожи на контракты, но их основная особенность и одновременно задача в том, что они размещаются в сети один раз, имеют фиксированный *адрес*, а другие контракты используют их код. Обращаться к библиотеке можно так же, как и к контракту.



Как уже было отмечено, библиотеки очень похожи на контракты, но есть и некоторые отличия. В библиотеках используются разные переменные и структуры данных, и, конечно, функции. Рассмотрим на примере из документации Solidity.

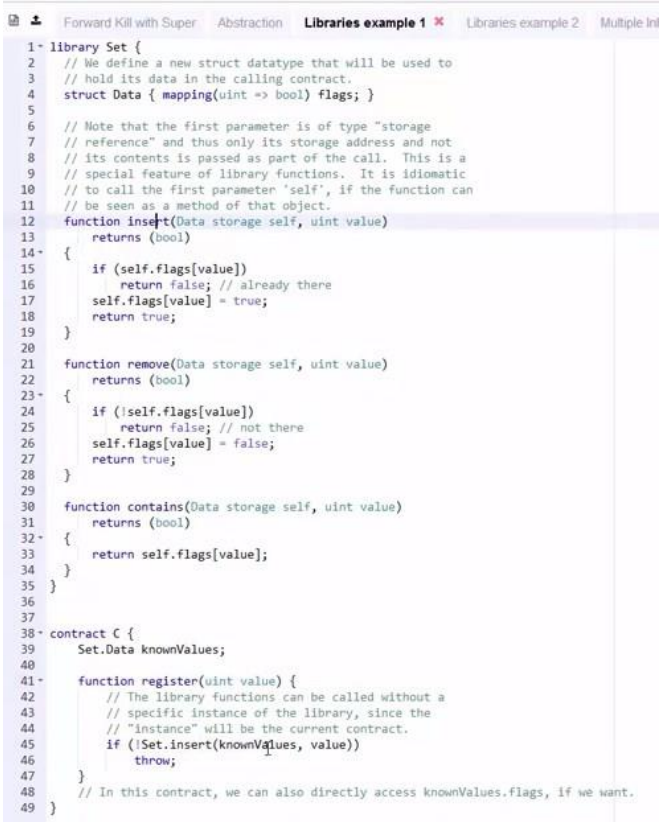


Посмотрим на первую функцию *insert*.



```
function insert(Data storage self, uint value)
    returns (bool)
{
    if (self.flags[value])
        return false; // already there
    self.flags[value] = true;
    return true;
}
```

Она требует указания двух параметров. Первый называется *self* и имеет тип *data storage*. Он обеспечивает получение адреса текущего контракта. Второй *параметр*, *value*, представляет собой некоторую величину. Вызовем функцию *insert* из нашего контракта *C*. Здесь есть *переменная knownValues* типа *Set.Data*. В функции *register* из контракта *C* мы вызываем функцию *insert* из типа *Set*, используя в качестве параметров переменную *knownValues*, которая берется из собственного хранилища данных, что задано служебными словами *Data storage self*, и некоторую величину, обозначенную как *value*.

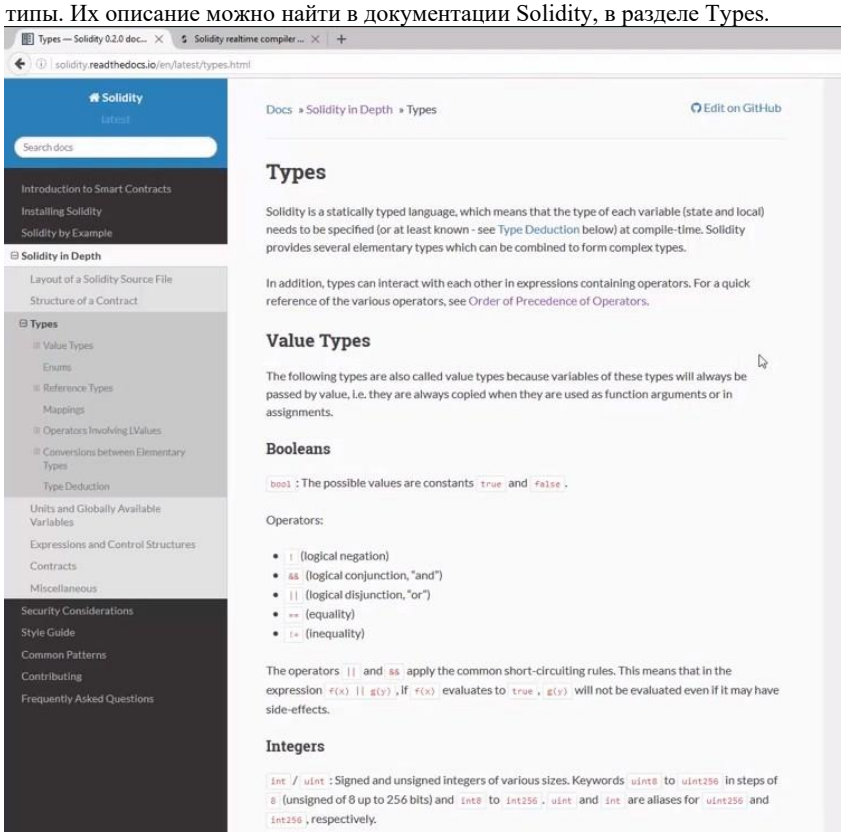


```
library Set {
  // We define a new struct datatype that will be used to
  // hold its data in the calling contract.
  struct Data { mapping(uint => bool) flags; }

  // Note that the first parameter is of type "storage
  // reference" and thus only its storage address and not
  // its contents is passed as part of the call.  This is a
  // special feature of library functions.  It is idiomatic
  // to call the first parameter 'self', if the function can
  // be seen as a method of that object.
  function insert(Data storage self, uint value)
      returns (bool)
  {
      if (self.flags[value])
          return false; // already there
      self.flags[value] = true;
      return true;
  }

  function remove(Data storage self, uint value)
      returns (bool)
  {
      if (!self.flags[value])
          return false; // not there
      self.flags[value] = false;
      return true;
  }

  function contains(Data storage self, uint value)
      returns (bool)
  {
      return self.flags[value];
  }
}


contract C {
    Set.Data knownValues;

    function register(uint value) {
        // The library functions can be called without a
        // specific instance of the library, since the
        // "instance" will be the current contract.
        if (!Set.insert(knownValues, value))
            throw;
    }
    // In this contract, we can also directly access knownValues.flags, if we want.
}
```

Стоит отметить, что *компилятор* не знает, *по* какому адресу хранится библиотека. Поэтому эти адреса должны быть указаны, чтобы связующая процедура могла к ним обратиться. Можно также указывать адреса вручную. Чтобы узнать больше о библиотеках, ознакомьтесь с документацией Solidity - в разделе *Contract and* Libraries.

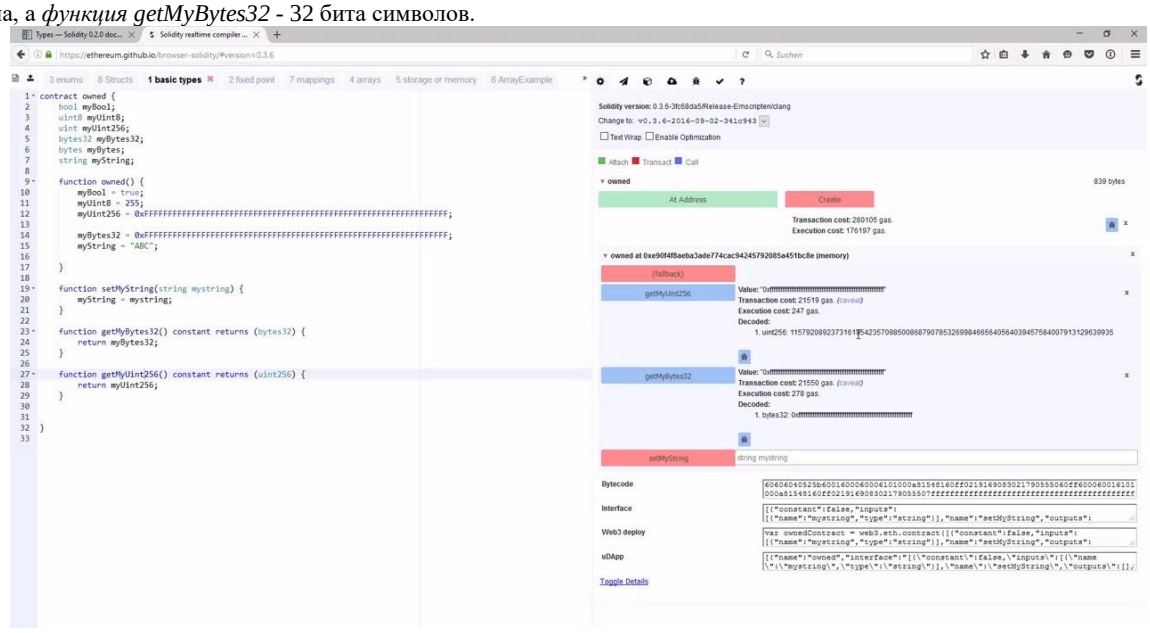
Типы переменных, массивы, структуры и ассоциации

Эта работа посвящена основным типам переменных, структурам, ассоциациям и способам их применения в среде Solidity. В среде Solidity есть булев тип, тип целых чисел и тип неотрицательных целых чисел, байты, строки, массивы, структуры, ассоциации и *пользовательский тип*. Величины с фиксированной запятой еще не внедрены. Посмотрим в деталях, как работают основные типы. Их описание можно найти в документации Solidity, в разделе Types.

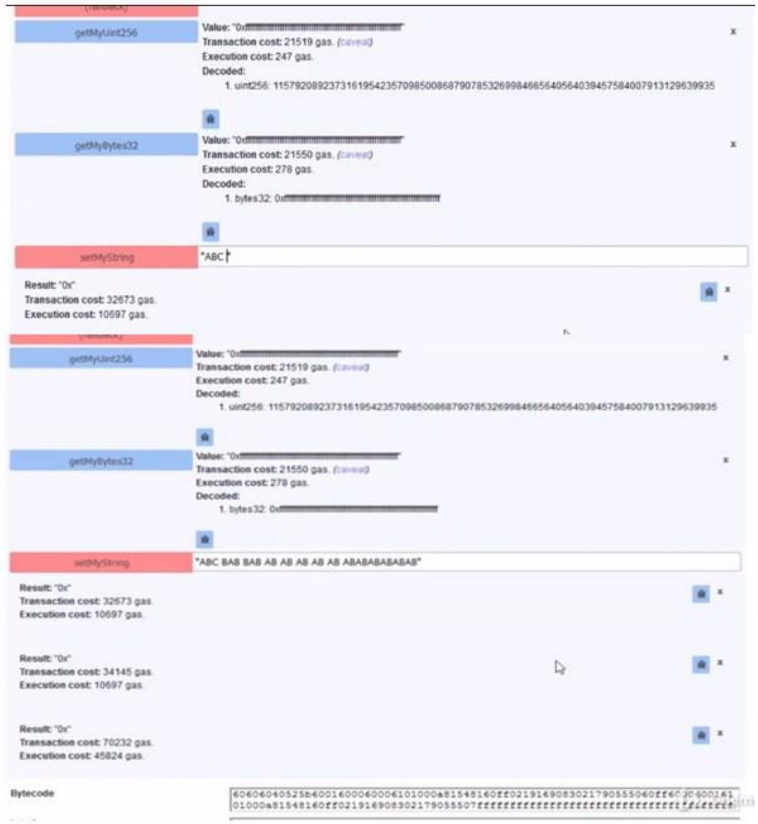


Переменные булевого типа могут принимать два значения: правда или *ложь*, как и в других языках программирования. Численные величины бывают типа *int* - с поддержкой знака минус, или *uint* - только для неотрицательных значений. Для их определения есть *целый* ряд ключевых слов, начиная от uint8 и до uint256 с шагом в 8. Это верно и для переменных целочисленного типа.

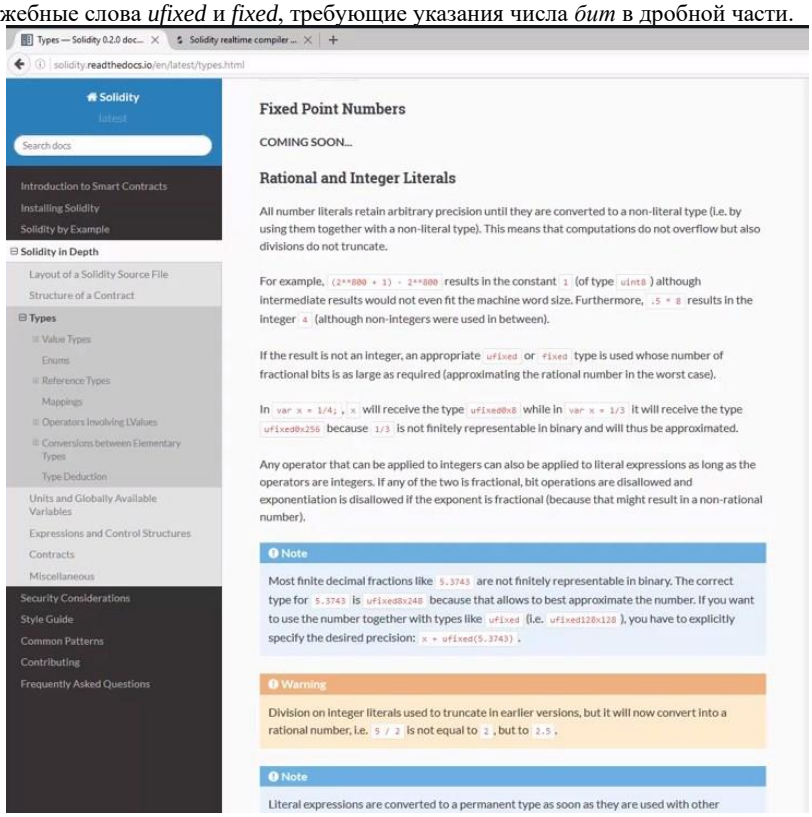
Рассмотрим пример. В конструкторе, записанном в нашем контракте, назначим несколько величин. Переменной *myBool* будет присвоено *значение* "правда", переменной *myUint8* - максимальное *значение* для типа *uint8, значение* "X" - символьным переменным. Есть также типы массивов символов и строки. В данном случаем две функции, возвращающие значения для символьных и численных переменных. *Переменная myUint256* возвращает 256 *бит* неотрицательного целого числа, а *функция getMyBytes32* - 32 бита символов.



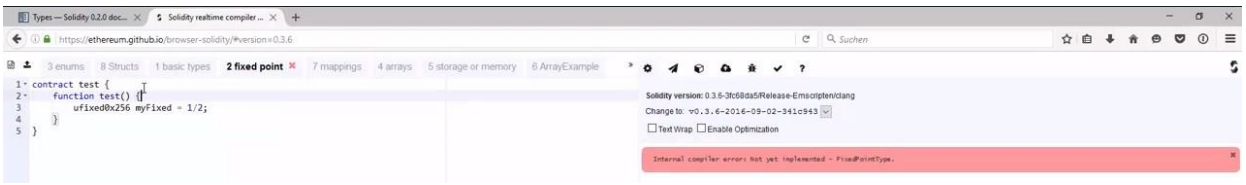
С помощью функции *setMyString* подчеркнем, что в зависимости от длины используемых строк на их обсчет будет уходить разное количество газа. Обратите внимание, что к массивам всегда присоединяется 256 *бит*.



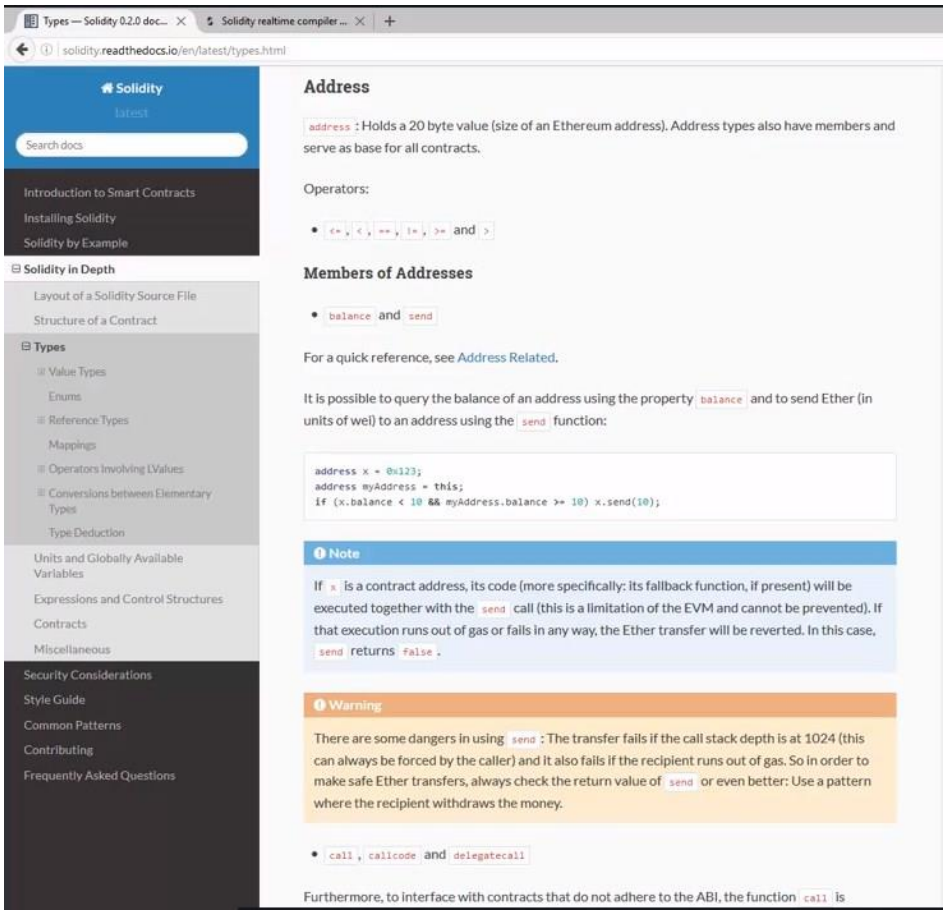
Поговорим о числах с фиксированной запятой. Они еще не представлены в среде Solidity, но скоро будут включены. Для их описания есть служебные слова *ufixed* и *fixed*, требующие указания числа *бит* в дробной части.



Если попытаться использовать тип *ufix* в браузере Solidity, будет выведено сообщение о внутренней ошибке компилятора, т.к. *поддержка* этого типа еще не включена.

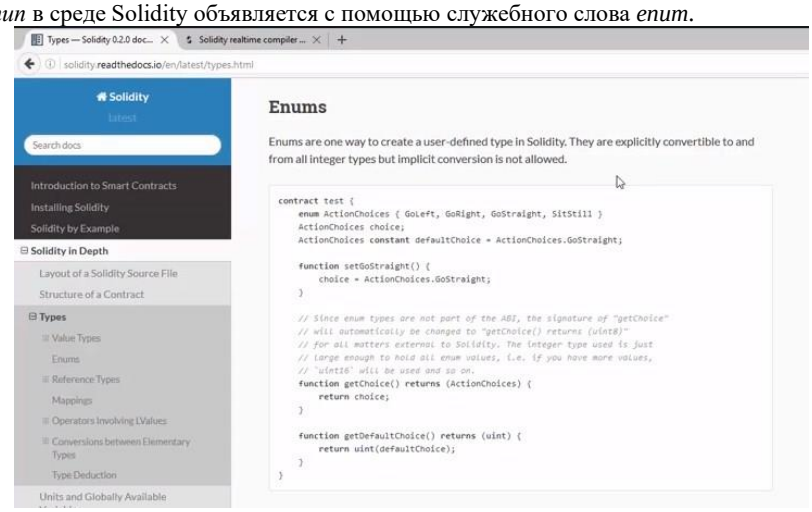


Еще один специальный тип в среде Solidity - это адресный тип. Он хранит двадцатибайтное *значение* адреса в сети Ethereum. Для этого типа есть несколько членов, например, balance, при вызове которого можно узнать баланс на конкретном адресе. Эта *функция* работает *по* умолчанию. Еще можно отправлять эфир на *адрес* с помощью команды address.send.

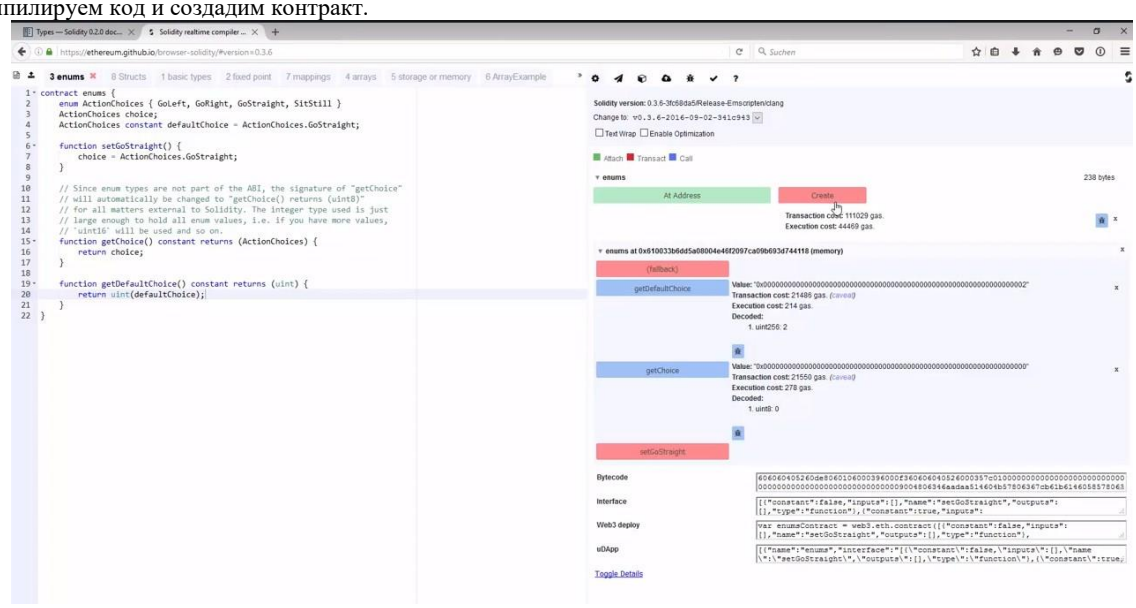


Рассмотрим на примере. Предположим, что есть *адрес* 0x123. Можно вызвать процедуры address.balance и address.send, первая вернет величину баланса на этом адресе, а вторая вернет *значение* "правда" в случае успешного перевода средств. Отдельно стоит отметить, что при отправке эфира необходимо быть внимательным. Если на обсчет транзакции *по* отправке эфира потратится весь предназначенный для этого газ, она завершится неуспешно и вернет *значение* "*ложь*". Поэтому всегда, когда пересылаете эфир, проверяйте *код возврата*.

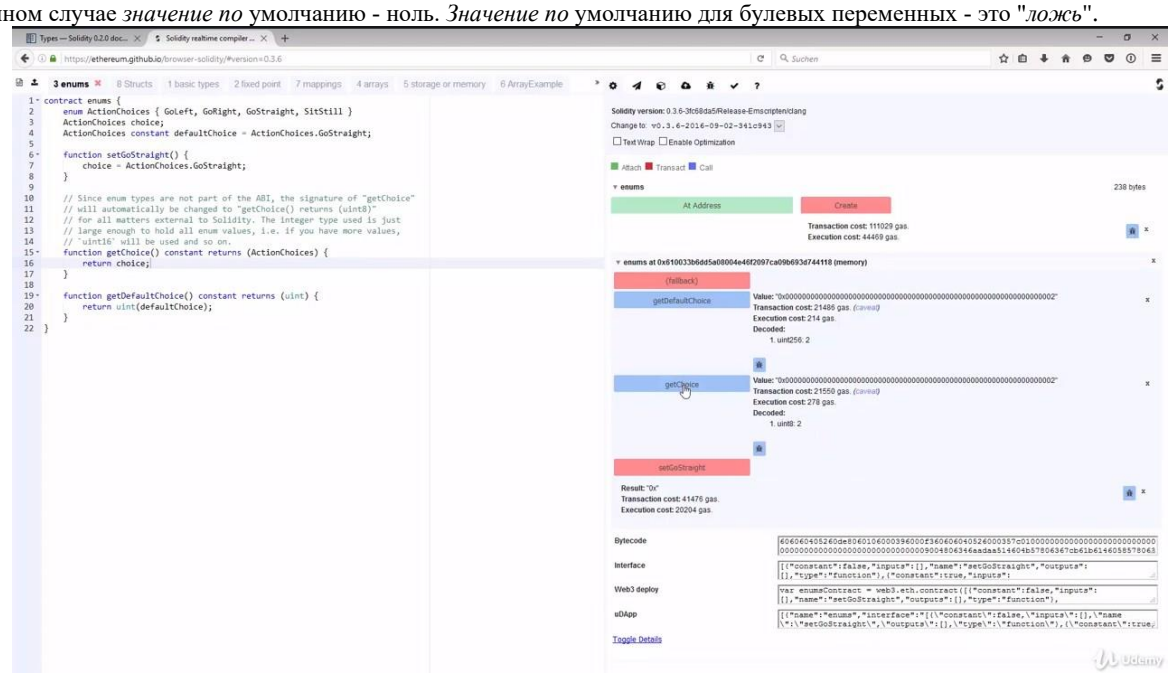
Пользовательский тип в среде Solidity объявляется с помощью служебного слова *enum*.



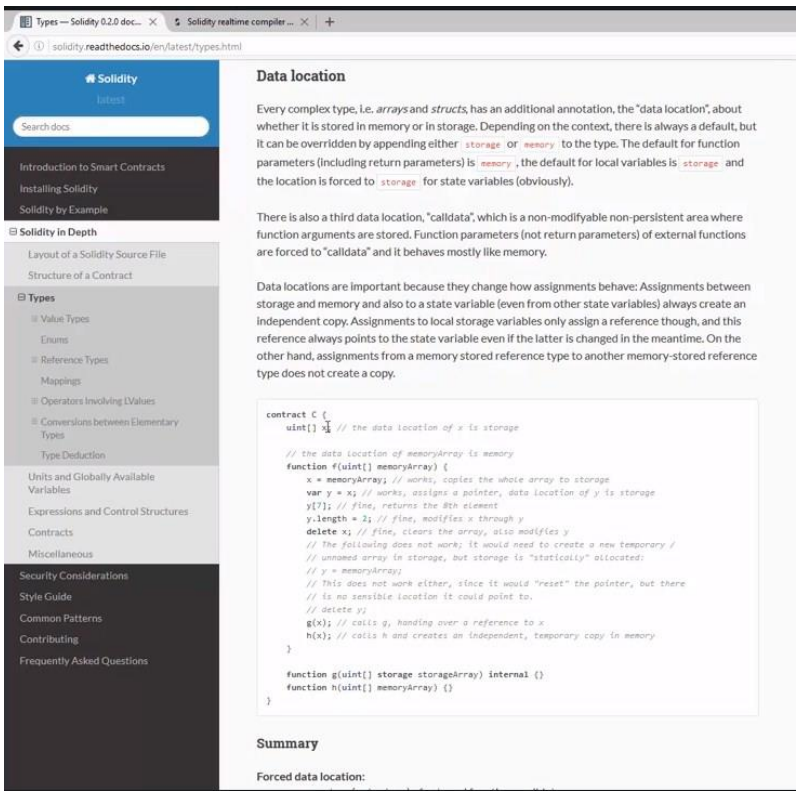
Скопируем пример с раздела в *браузер* Solidity. В контракте этот тип назван *ActionChoices* и включает в себя значения переменных *GoLeft, GoRight, GoStraight* и *SitStill*. Для внутренней обработки Solidity переводит подобные массивы в тип *uint*, сначала применяя тип *uint8*, и расширяя его до *uint16* и так далее *по* мере роста числа вариантов значения переменных. Скомпилируем код и создадим контракт.



В нем две функции - *getChoice* и *getDefaultChoice*. Обратим внимание на возвращаемую величину. *Функция getChoice* возвращает выбор действий; для целей внутренней обработки *список* действий, из которых можно выбирать, будет храниться в типе *uint8*, поскольку *список* состоит всего из четырех позиций. Одновременно с этим *функция getDefaultChoices* будет возвращать результат типа *uint256*, поэтому если потребуется работать с типом полученного результата, его нужно будет переопределить в тип *uint256*. Посмотрим на полученный результат в правой части браузера Solidity. *Функция getDefaultChoice* возвращает результат типа *uint256*, и этот результат - два. (Обратите внимание на порядок перечисления вариантов выбора в определении типа.) *Функция getChoice* возвращает результат типа *uint8*, в данном случае - ноль. Напомним, что все вводимые переменные и функции имеют некоторые значения *по* умолчанию, в данном случае *значение по* умолчанию - ноль. *Значение по* умолчанию для булевых переменных - это "*ложь*".



Давайте сделаем выбор *goStraight* с помощью функции *setGoStraight* и проверим, каков будет результат работы функции *getChoice*. Теперь он тоже равен двум.
Массивы в среде Solidity бывают фиксированные и динамические.

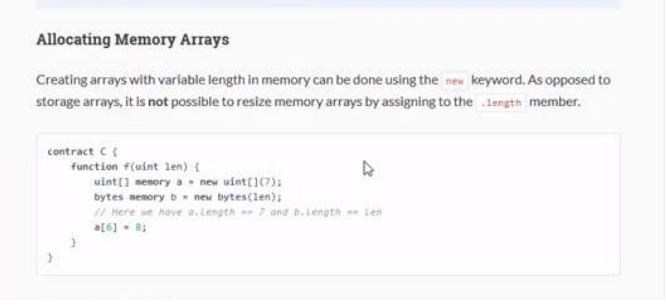


Однако перед тем, как перейти к массивам, давайте поговорим немного о способах хранения данных. Solidity предусматривает два варианта: *хранение данных* в накопителе или непосредственно в памяти. *По* умолчанию все данные для параметров функций, в том числе результаты, которые функции возвращают, хранят в памяти, а локальные переменные - в накопителе. Важно понимать разницу между этими способами хранения данных. Если вы выполняете переназначение между двумя переменными, одна из которых хранится в памяти, а другая - в накопителе, то эта процедура всегда будет создавать независимую копию одной из переменных. Если обе переменные хранятся одинаковым способом, то создается только *ссылка* на *адрес* хранения данных. Посмотрим подробнее в следующем примере.

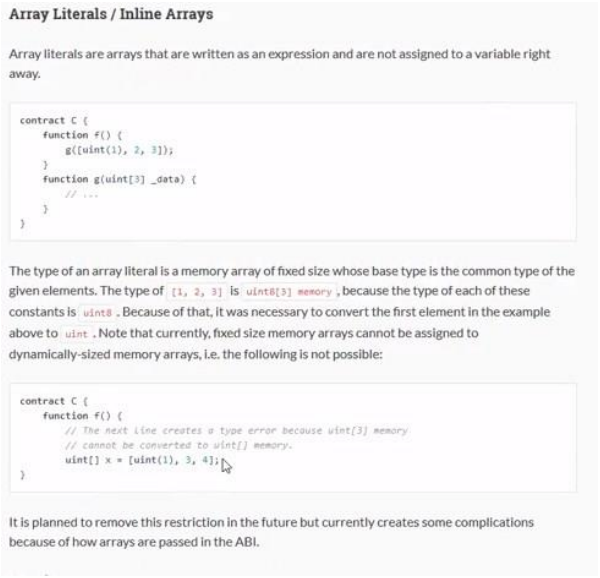
```
contract C {
    uint[] x; // the data location of x is storage

    // the data location of memoryArray is memory
    function f(uint[] memoryArray) {
        x = memoryArray; // works, copies the whole array to storage
        var y = x; // works, assigns a pointer, data location of y is storage
        y[7]; // fine, returns the 8th element
        y.length = 2; // fine, modifies x through y
        delete x; // fine, clears the array, also modifies y
        // The following does not work; it would need to create a new temporary /
        // unnamed array in storage, but storage is "statically" allocated:
        // y = memoryArray;
        // This does not work either, since it would "reset" the pointer, but there
        // is no sensible location it could point to.
        // delete y;
        g(x); // calls g, handing over a reference to x
        h(x); // calls h and creates an independent, temporary copy in memory
    }

    function g(uint[] storage storageArray) internal {}
    function h(uint[] memoryArray) {}
}
```

Массив X типа *uint* - это динамический *массив*, хранящийся в накопителе. *Функция* F - это *массив* типа *uint*, который хранится в памяти, и называется *memoryArray*. Теперь присвоим X *значение memoryArray*, что скопирует весь *массив* в накопитель. Затем введем новую переменную Y, которой присвоим *значение* X. Y хранит только ссылку на *адрес* расположения данных из X. *Команда* Y[7] вернет нам восьмой элемент X. Поскольку все величины имеют некоторые значения *по* умолчанию, если ничего не было присвоено этому элементу, то будет возвращен ноль. Теперь изменим что-нибудь в Y, тем самым мы вносим изменения и в X. Удаляем всё, кроме первых двух элементов. А теперь удаляем весь *массив*.

Вернемся к примеру с массивами.



В контракте C есть *функция* F с параметром *Len* типа *uint*, а именно *uint256*. *Массив* a хранится в памяти. Массивы в памяти можно создавать с помощью служебного слова *new*. Стоит обратить внимание на то, что размер массивов, хранящихся в накопителе, можно легко изменять с помощью процедуры *length*, просто назначая новое число элементов массива. Такую операцию нельзя выполнять с массивами, хранящимися в памяти. *Массив* a насчитывает семь элементов. А хранящийся в памяти *массив bytes* имеет длину *len*. Все должно работать без ошибок, мы даже назначили седьмому элементу массива a *значение* восемь.
В Solidity можно работать со вложенными массивами.



Array Literals / Inline Arrays

Array literals are arrays that are written as an expression and are not assigned to a variable right away.

```
contract C {
    function f() {
        g([uint(1), 2, 3]);
    }
    function g(uint[3] _data) {
        // ...
    }
}
```

The type of an array literal is a memory array of fixed size whose base type is the common type of the given elements. The type of `[1, 2, 3]` is `uint8[3] memory`, because the type of each of these constants is `uint8`. Because of that, it was necessary to convert the first element in the example above to `uint`. Note that currently, fixed size memory arrays cannot be assigned to dynamically-sized memory arrays, i.e. the following is not possible:

```
contract C {
    function f() {
        // The next line creates a type error because uint[3] memory
        // cannot be converted to uint[] memory.
        uint[] x = [uint(1), 3, 4];
    }
```

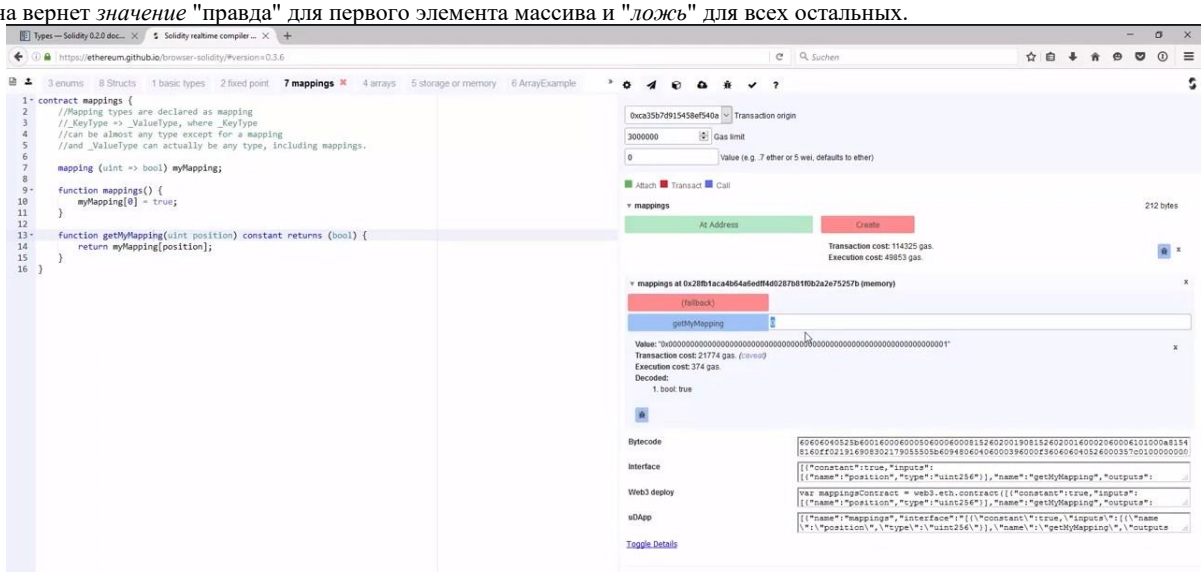
It is planned to remove this restriction in the future but currently creates some complications because of how arrays are passed in the ABI.

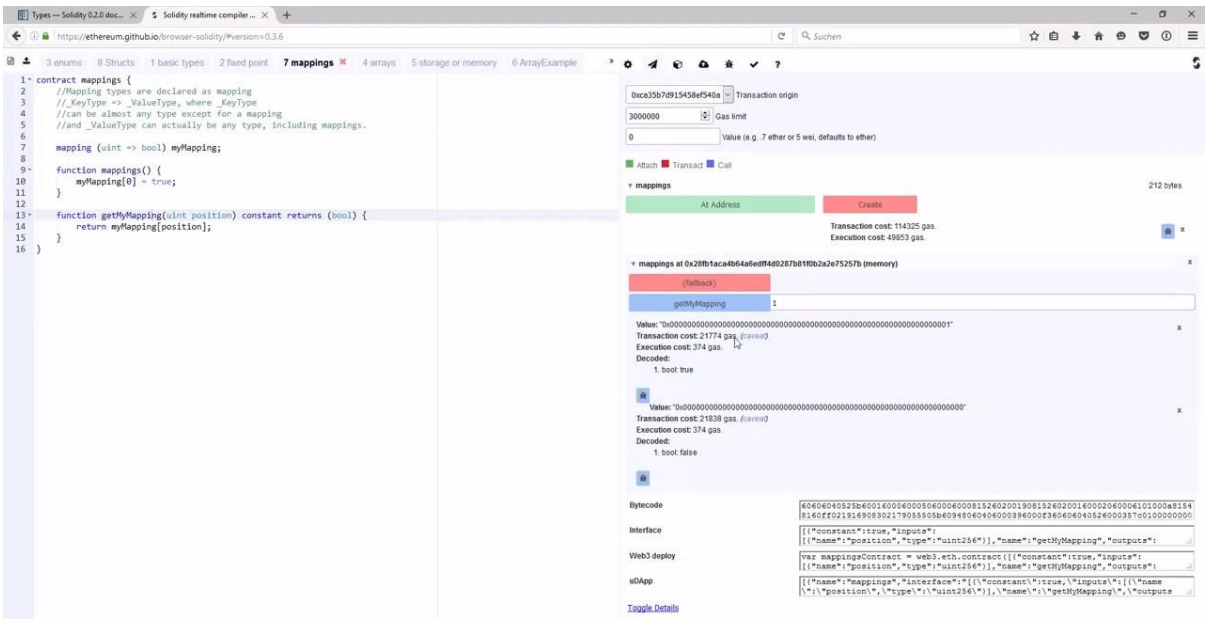
В демонстрируемом примере *функция* f вызывает функцию g, у которой есть один *параметр* - *массив* из трех элементов. Обратим внимание, что при вызове функции g, когда будет задан этот *массив*, назначив величины, скажем, один, два и три, получится *массив* типа *uint8*. Придется изменять *тип массива* на *uint256*. Основным типом массива считается общий тип всех его членов. Также важно отметить, что хранящиеся в памяти массивы фиксированной длины не могут стать динамическими. Поэтому в этом участке кода *компилятор* выдаст ошибку - нельзя присвоить динамическому массиву *массив* фиксированной длины.
У типа массива есть два члена. Один, обозначающий длину, вызывается служебным словом *length*, и возвращает число элементов массива. Размер динамических массивов, хранящихся в накопителе, может быть изменен редактированием члена *length*. Это не происходит автоматически: перед записью в *массив* величин необходимо вручную изменить его длину. Размер массивов, хранящихся в памяти, как уже упоминалось ранее, фиксирован. Вторым членом для всех типов массивов, кроме строковых, является *push*, применяемый для добавления элементов к концу массива. Вызов процедуры *push* возвращает новую длину массива.
Обратимся к примеру из документации.

```
contract ArrayContract {
    uint[2**20] m_aLotOfIntegers;
    // Note that the following is not a pair of arrays but an array of pairs.
    bool[2][] m_pairsOfFlags;
    // newPairs is stored in memory - the default for function arguments

    function setAllFlagPairs(bool[2][] newPairs) {
        // assignment to a storage array replaces the complete array
        m_pairsOfFlags = newPairs;
    }

    function setFlagPair(uint index, bool flagA, bool flagB) {
        // access to a non-existing index will throw an exception
        m_pairsOfFlags[index][0] = flagA;
        m_pairsOfFlags[index][1] = flagB;
    }

    function changeFlagArraySize(uint newSize) {
        // if the new size is smaller, removed array elements will be cleared
        m_pairsOfFlags.length = newSize;
    }

    function clear() {
        // these clear the arrays completely
        delete m_pairsOfFlags;
        delete m_aLotOfIntegers;
        // identical effect here
        m_pairsOfFlags.length = 0;
    }

    bytes m_byteData;

    function byteArrays(bytes data) {
        // byte arrays ("bytes") are different as they are stored without padding,
        // but can be treated identical to "uint8[]"
        m_byteData = data;
        m_byteData.length += 7;
        m_byteData[3] = 8;
        delete m_byteData[2];
    }

    function addFlag(bool[2] flag) returns (uint) {
        return m_pairsOfFlags.push(flag);
    }

    function createMemoryArray(uint size) returns (bytes) {
        // Dynamic memory arrays are created using `new`:
        uint[2][] memory arrayOfPairs = new uint[2][](size);
        // Create a dynamic byte array:
        bytes memory b = new bytes(200);
        for (uint i = 0; i < b.length; i++)
            b[i] = byte(i);
        return b;
    }
```

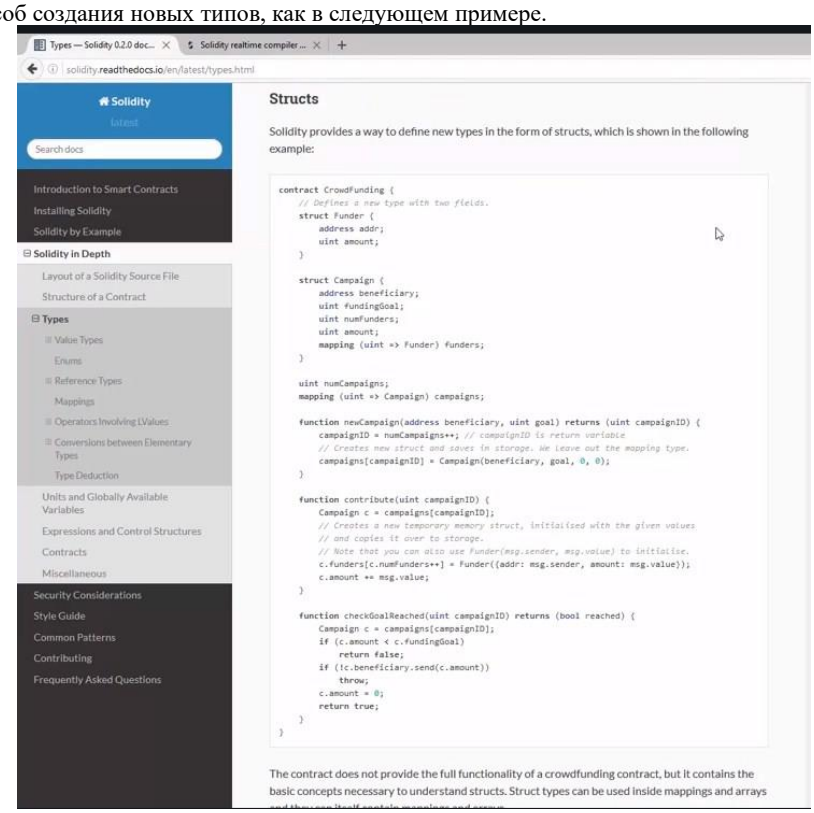
Создаем очень крупный *массив*, хранящийся в накопителе. В нем два в двадцатой степени элементов. Затем создаем еще пару массивов. *Функция setAllFlagPairs* имеет один *параметр - newPairs*, который *по* умолчанию хранится в памяти, как и все параметры функций. Если его *значение* присвоить *m_PairsOfFlags*, он будет скопирован в накопитель. *Функция setFlagPair* имеет три параметра: один - *index* типа *uint*, и два булевых параметрах. При попытке записать что-либо в *index компилятор* выдаст ошибку. Всегда необходимо изменять длину массива перед записью в него данных. Будьте внимательны при редактировании размеров массивов. Если величина *length* будет больше, чем текущий размер массива, он будет расширен. С другой стороны, если величина *length* будет меньше, чем размер массива, все элементы после *length*, будут удалены. Чтобы полностью удалить *массив*, можно воспользоваться служебным словом *delete*. Тот же эффект можно получить, выставив размер массива, *length*, на ноль.

С помощью ассоциаций можно наделить почти все типы свойствами других. Ассоциации можно рассматривать как хэш-таблицы всех возможных комбинаций величин. Работа с ассоциациями похожа на работу с массивами. Давайте рассмотрим простую ассоциацию. Я добавил *числовой тип* булевым величинам, и в конструкторе назначил первому элементу *значение* два. *Функция getMyMapping* возвращает булеву величину в зависимости от номера массива, поэтому она вернет *значение* "правда" для первого элемента массива и "*ложь*" для всех остальных.





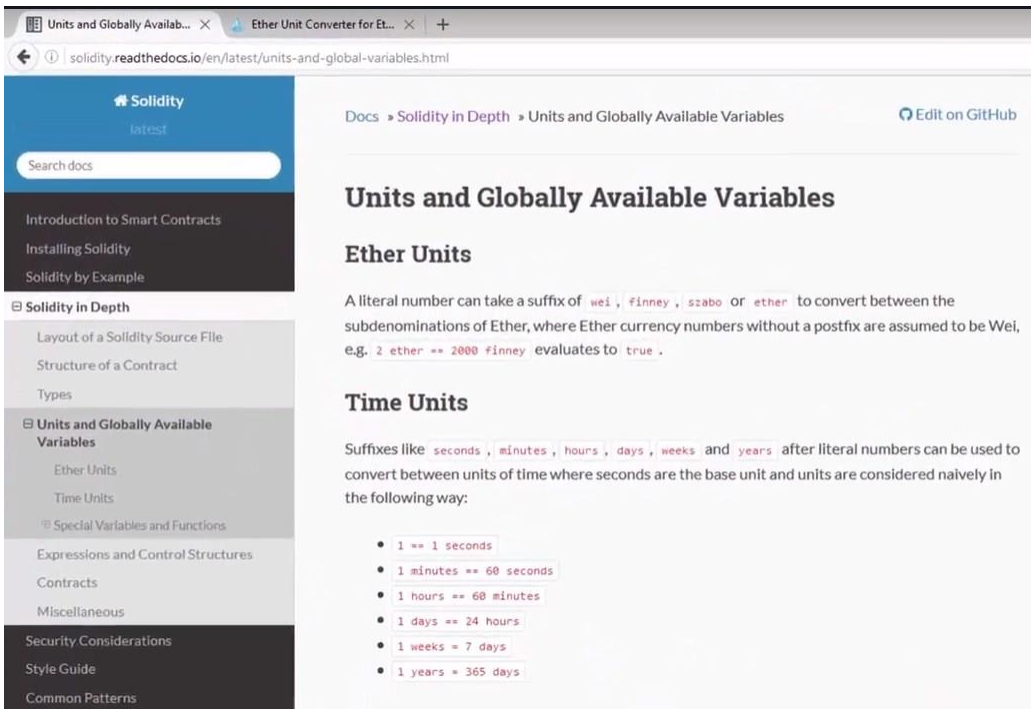
Структура - это способ создания новых типов, как в следующем примере.



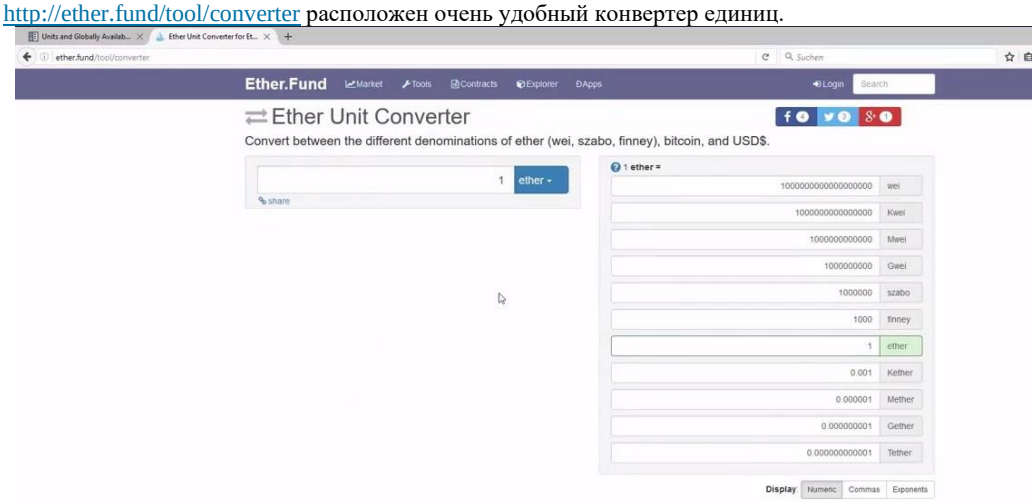
Вот две структуры: одна называется Funder, а другая - *Campaign*. (Стоит отметить, что этот пример в целом не слишком работоспособен, он приведен в иллюстративных целях.) Число кампаний - это величина типа *uint*. *Ассоциация* производится между величинами типа *uint* и кампаниями. При добавлении новой кампании нужно увеличить число кампаний (величины этого массива названы *campaignID*), затем назначить новую кампанию. Если работа над этим кодом ведется группой программистов, то можно узнавать названия кампаний из их массива и добавлять нужную логику в программу.

Глобальные переменные

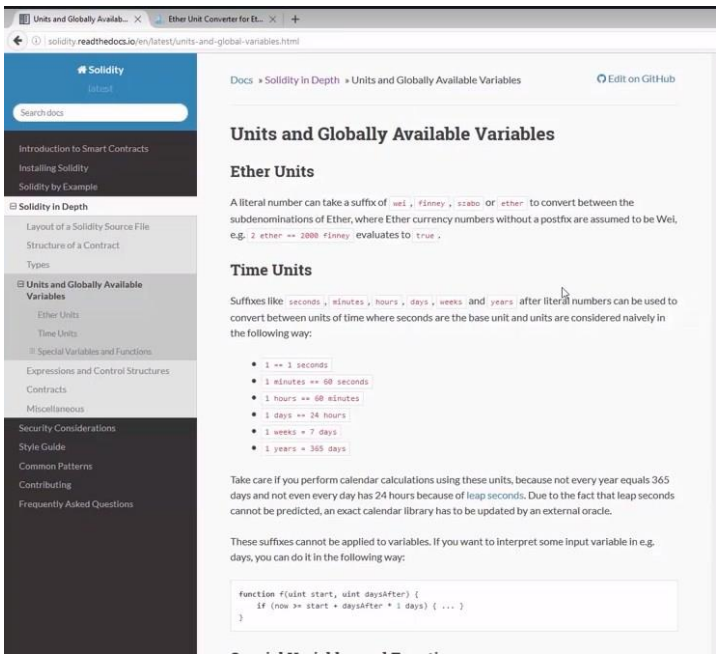
Эта часть посвящена единицам измерений и глобальным переменным.



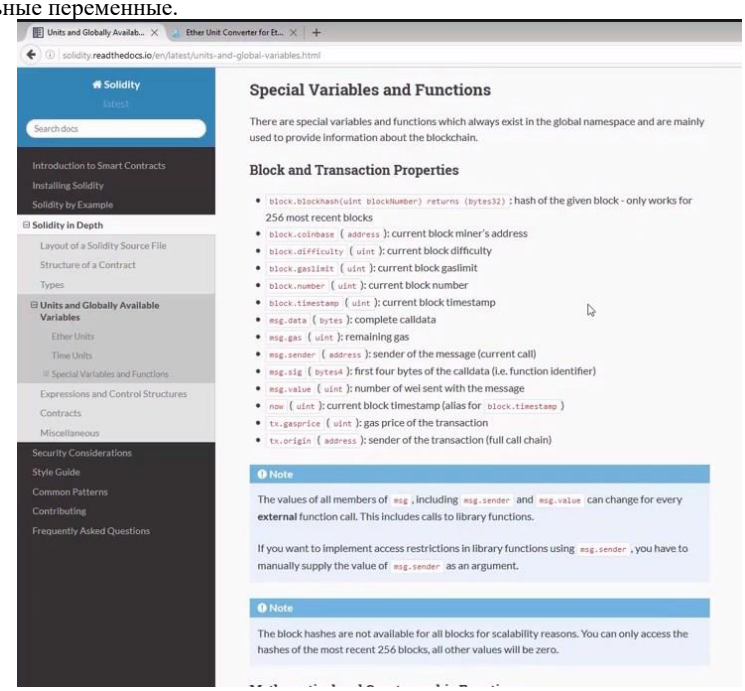
Прежде чем начать программировать, стоит ознакомиться с используемыми в среде Ethereum единицами измерений. На веб-сайте http://ether.fund/tool/converter расположен очень удобный конвертер единиц.



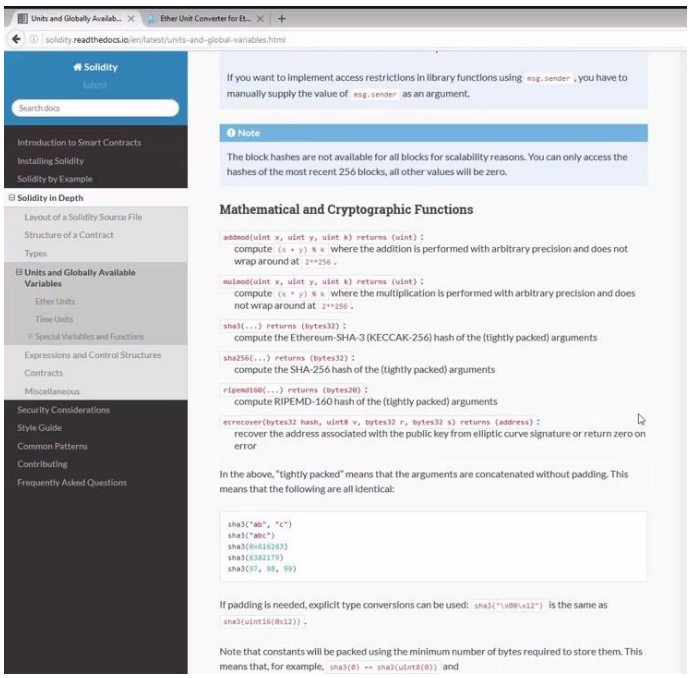
Самой маленькой единицей измерения эфира является вей (wei). Один квинтиллион веев составляют эфир (ether). Финни (finney) меньше эфира, а сабо (szabo) меньше финни. Один эфир - это тысяча финни, а один финни - это тысяча сабо, и так далее. В коде Solidity можно использовать эти названия как служебные слова для определения единиц измерений, например, можно написать "2 ether == 2000 finney", и система вернет *значение* "*истина*".
Что касается единиц измерения времени, то минимальной величиной в Ethereum является секунда (seconds). Как и в случае с валютой, работают служебные слова, например, можно написать "1 minutes is 60 seconds".



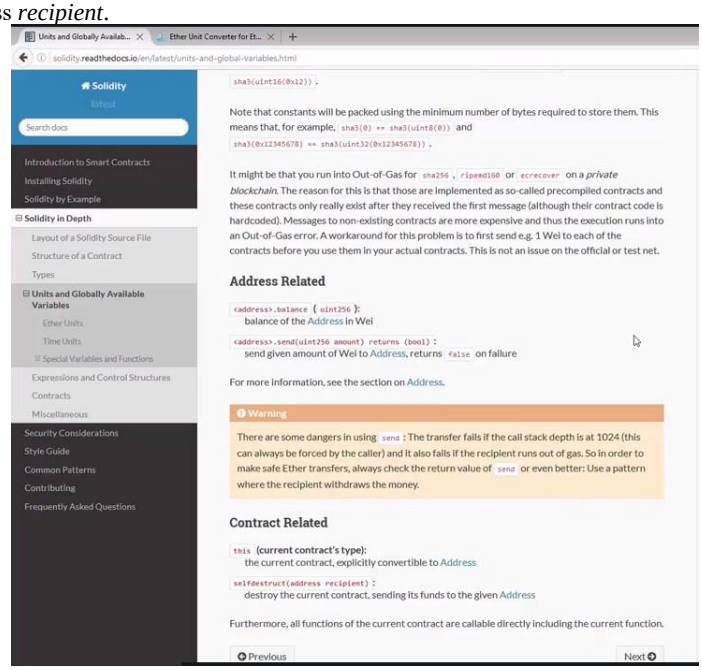
Посмотрим на специальные переменные.



Они являются составной частью среды Solidity и доступны без объявления. Одной из самых важных переменных является *msg*, в частности, в переменной msg.sender всегда хранится *адрес* вызывающей процедуры. Если же при проведении транзакции пересылался эфир, то *переменная* msg.value будет содержать количество отправленного эфира, выраженное в вей. *Служебное слово* now возвращает временную метку текущего блока. Если ваш код обращается к внешним процедурам или библиотекам, *адрес* вызывающего контракта будет храниться в переменной msg.sender. Обратите внимание, что *адрес* лица, запустившего транзакцию, не хранится в этой переменной.
Solidity также поддерживает некоторые математические функции и функции шифрования.



У типа *address* есть некоторые особые члены. Один из них - это *balance*, возвращающий количество эфира на балансе у этого адреса, выраженное в веях. Другой - это *send*. С помощью функции *send* можно отправить этому адресу эфир. *Функция send* вернет *значение* "*истина*" в случае успешного перевода средств.
Наконец, у типа *contract* тоже есть два члена. Первый - это *this*, *указатель* на *адрес* текущего контракта. Второй - это *функция* self-destruct. Ее вызов уничтожит контракт и отправит оставшийся эфир получателю, указанному в качестве параметра функции address *recipient*.



Задание

1. Создайте следующие смарт-контракты на языке Solidity:

– Контракт с тремя функциями, одна задает значение переменной (используя механизм передачи параметра), вторая - удваивает значение переменной, а третья - его возвращает.
– В одном файле два контракта: первый - из предыдущего задания, второй - содержит функцию, возвращающую значение переменной из первого контракта.
– Контракт, в котором используется условный оператор.
– Контракт с массивами.

Контрольные вопросы

1. Структура контракта?
2. Единицы и глобальные переменные в языке Solidity
3. Создание контракта
4. Область видимости
5. Геттер-функции
6. Модификаторы функций
7. Константы
8. Функции
9. События
10. Наследование
11. Абстрактные контракты
12. Интерфейсы
13. Библиотеки?
14. Типы данных в языке Solidity?

Приложение №1

Размещение контрактов в сети

Экосистема платформы Ethereum, распределенные приложения и ДАО

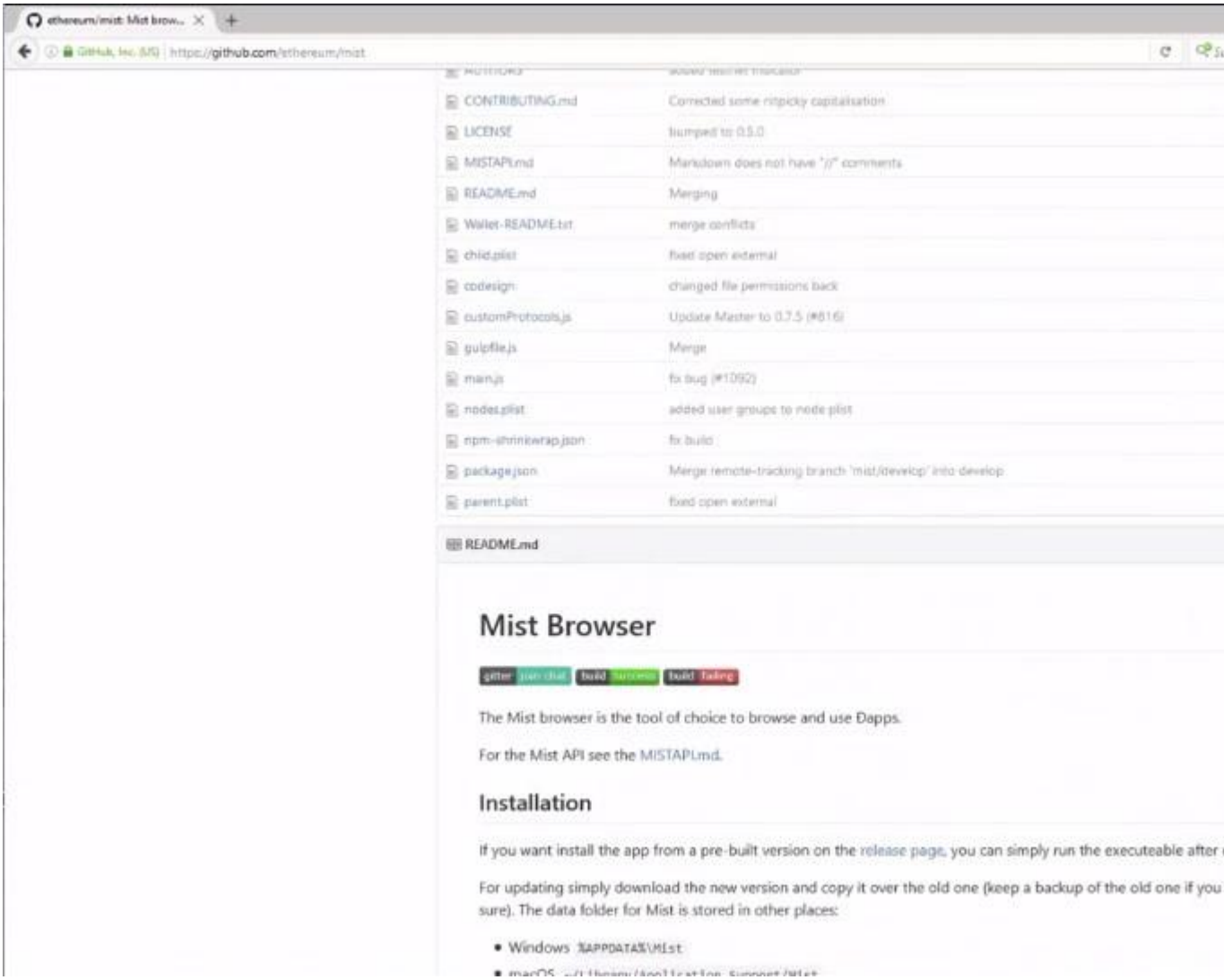
Эта приложение посвящено платформе Ethereum. Узнаем, что такое распределенные приложения и демократические автономные организации, с описанием которых можно ознакомиться на веб-сайте http://ethereum.org. Начнем с распределенных приложений. В отличие от традиционной архитектуры, в которой одна программа запускается на сервере и ждет, пока другая программа к ней подключится, в среде распределенных приложений они работают на всех серверах и всех узлах сети одновременно. В следующих разделах расскажем, как программировать такие распределенные приложения и размещать их в блокчейне. Распределенные приложения работают на так называемых виртуальных машинах Ethereum, а для их создания используются несколько высокоуровневых языков программирования, например, Solidity. Эти языки чем-то напоминают ассемблер, поскольку программы на них состоят из нескольких кодов операций, которые и запускаются на виртуальной машине Ethereum в блокчейне.

Платформа Ethereum также поддерживает свой официальный бумажник - Mist. В этом приложении можно создавать счета, заключать контракты или добавлять уже размещенные контракты и взаимодействовать с ними. Эти контракты, как и другие распределенные приложения, пишутся на высокоуровневых языках программирования, например, на Solidity. На текущий момент это самый продвинутый и наиболее широко используемый язык программирования для этой среды, хотя есть и другие реализации. Запущенные распределенные приложения работают непосредственно в блокчейне, и поэтому не существует единой силы или организации, способной вывести работающее приложение из строя. Одной из самых значительных сфер применения распределенных приложений является финансовая сфера. Если ваше приложение запущено в блокчейне, можно быть уверенным, что никакое правительство или другая сила не смогут отключить его или отредактировать код. Когда приложение размещено в блокчейне, его можно изменить только с помощью внутренней инструкции в его собственном коде. Например, вы можете развернуть в блокчейне краудфандинговую кампанию, в которой и само приложение, и собираемые денежные средства будут храниться в блокчейне, а между ними не будет какой-либо прослойки.

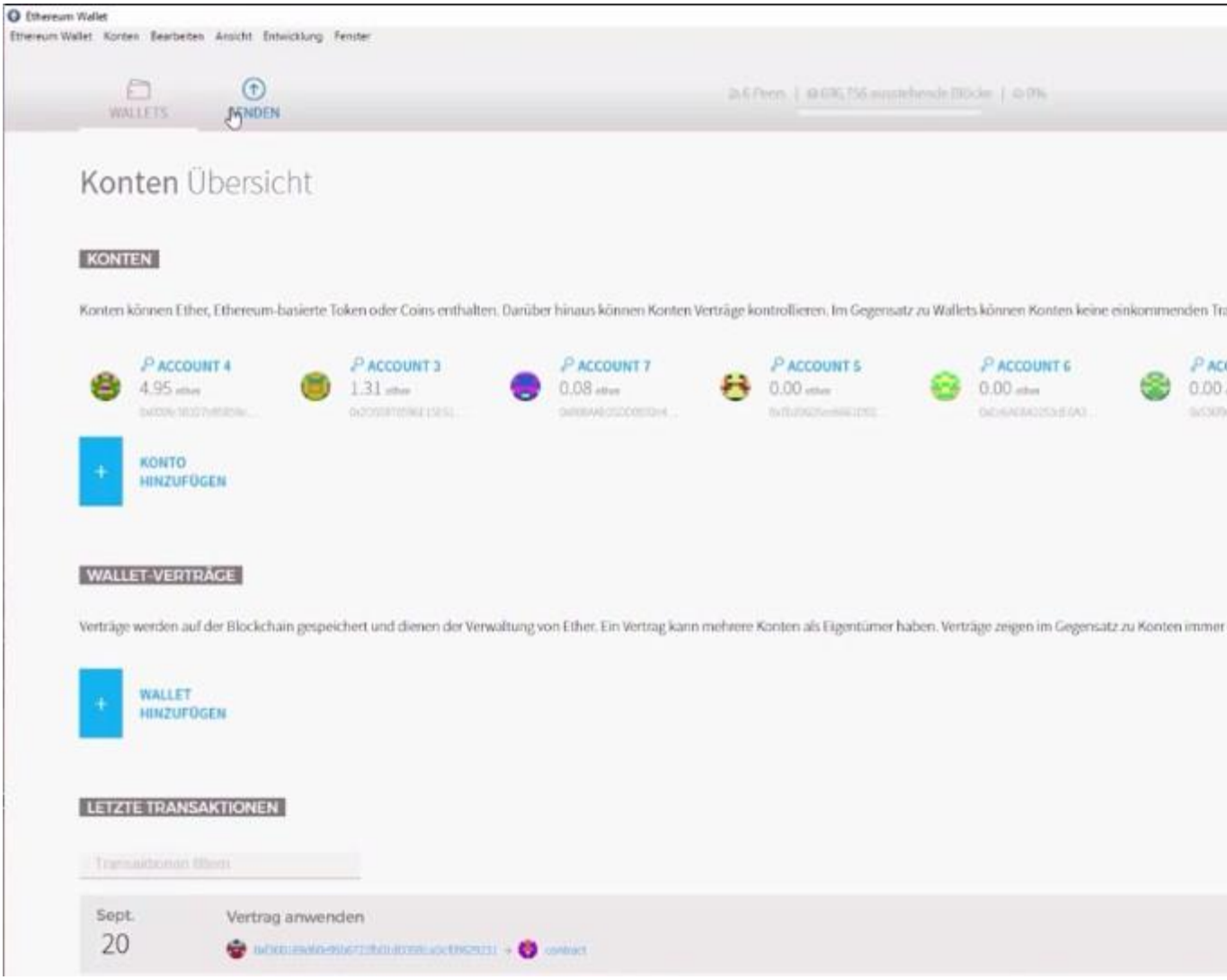
Эта же идея лежит в основе демократических автономных организаций, или ДАО. ДАО - это виртуальная организация, члены которой могут голосовать по определенным вопросам. Деятельность таких организаций прозрачна и основана целиком на голосах участников. После того, как голос был отдан и размещен в блокчейне, его уже нельзя изменить. Это напоминает государство с неизменяемой конституцией, и всё это описано в программном коде. В этой области уже есть ряд выдающихся проектов, с некоторыми из них можно ознакомиться на веб-сайте EtherCasts. Ссылку на него можно найти в разделе распределенных приложений DApps, а далее перейти на ethercasts.com. Есть и целый ряд инструментов для командной строки. Один из самых известных клиентов написан на языке Go и называется Go Ethereum, или просто Geth. Для платформы Ethereum есть еще много различных полезных ресурсов, а также мощная поддержка сообщества. Кроме собственно сайта и блога на нем, можно пойти на форум Reddit и задать интересующие вопросы. Можно принимать участие в обсуждениях на Stack Exchange. Можно также посетить Meetups или отправиться на конференции вроде Devcon, проходящей в Шанхае.

Что такое кошелек Mist и как он работает

Для платформы Ethereum существует несколько веб-кошельков. Mist, приложение с открытым исходным кодом, имеет статус официального. Его можно загрузить через веб-сайт http://github.com/Ethereum/mist.



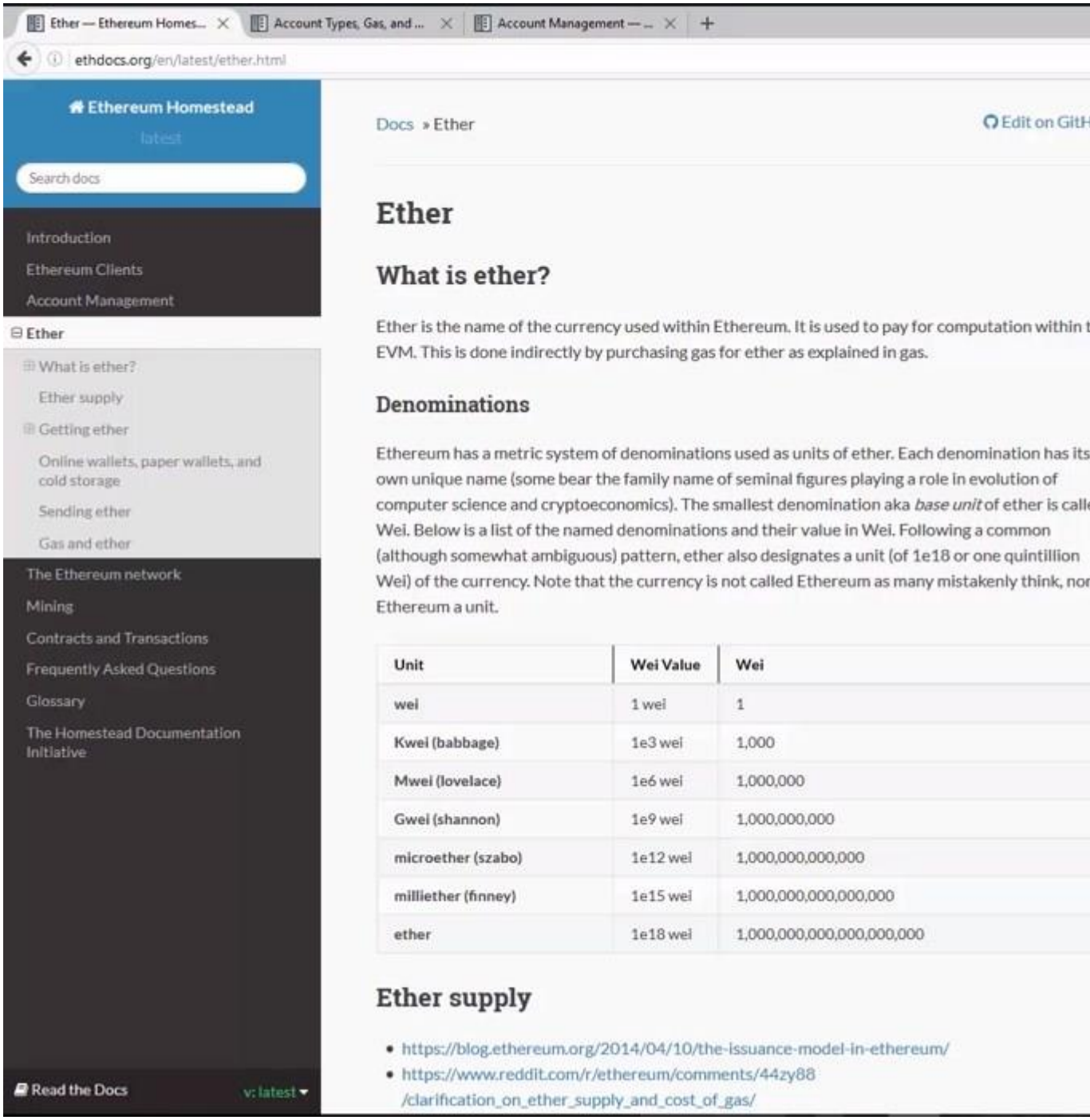
По этой ссылке можно найти инструкции по установке для всех типов операционных систем. Mist работает на Windows, на Mac OS и на Linux. Mist позволяет управлять своими счетами, отправлять валюту, запускать и добавлять новые контракты, просматривать списки последних транзакций.



В следующих разделах будем работать с Mist и разберем, как другие инструменты для командной строки взаимодействуют с Mist и блокчейном в целом.

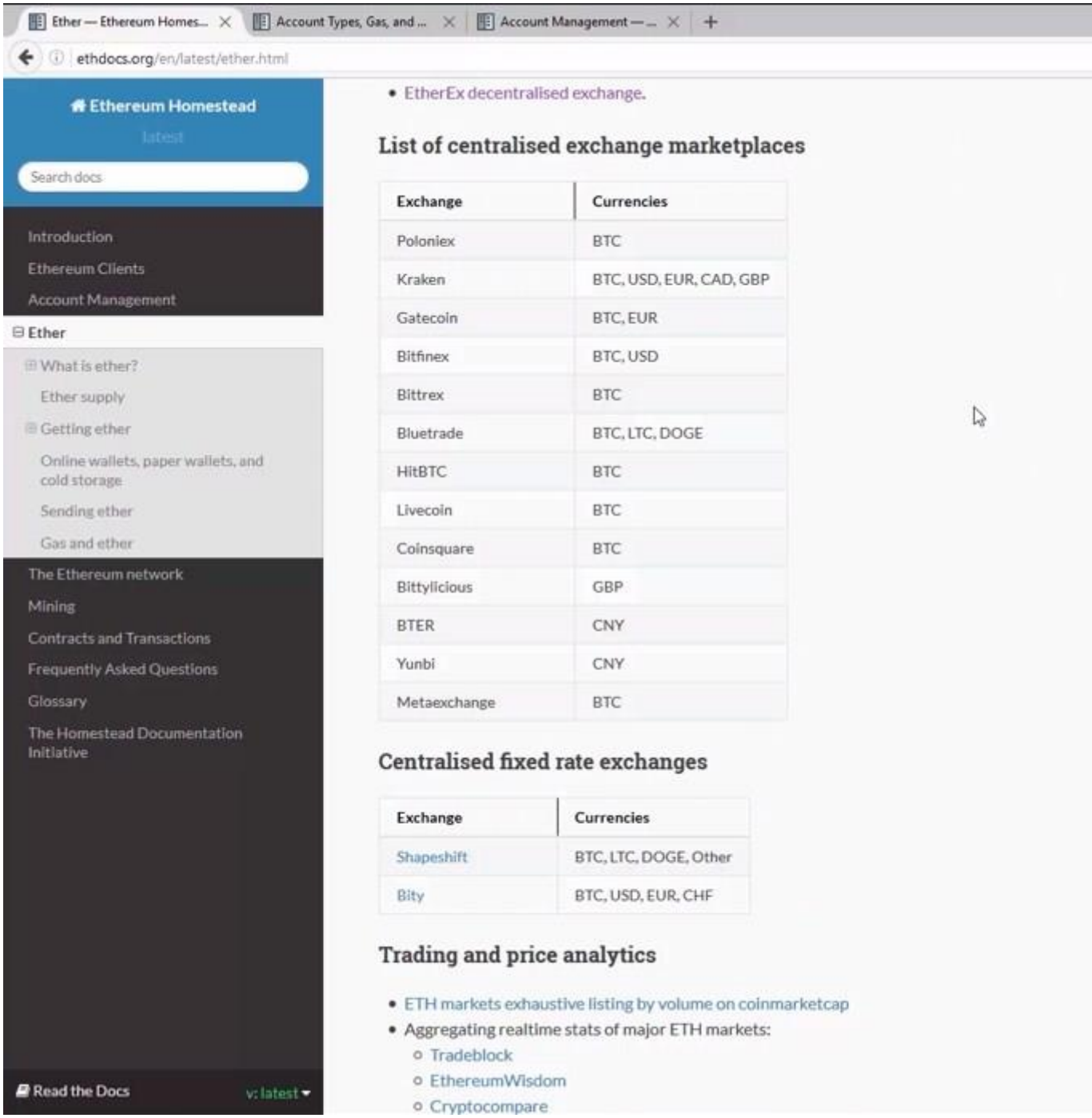
Разбираемся, что такое эфир и какие есть типы счетов

В среде Ethereum есть своя валюта, называемая эфиром.

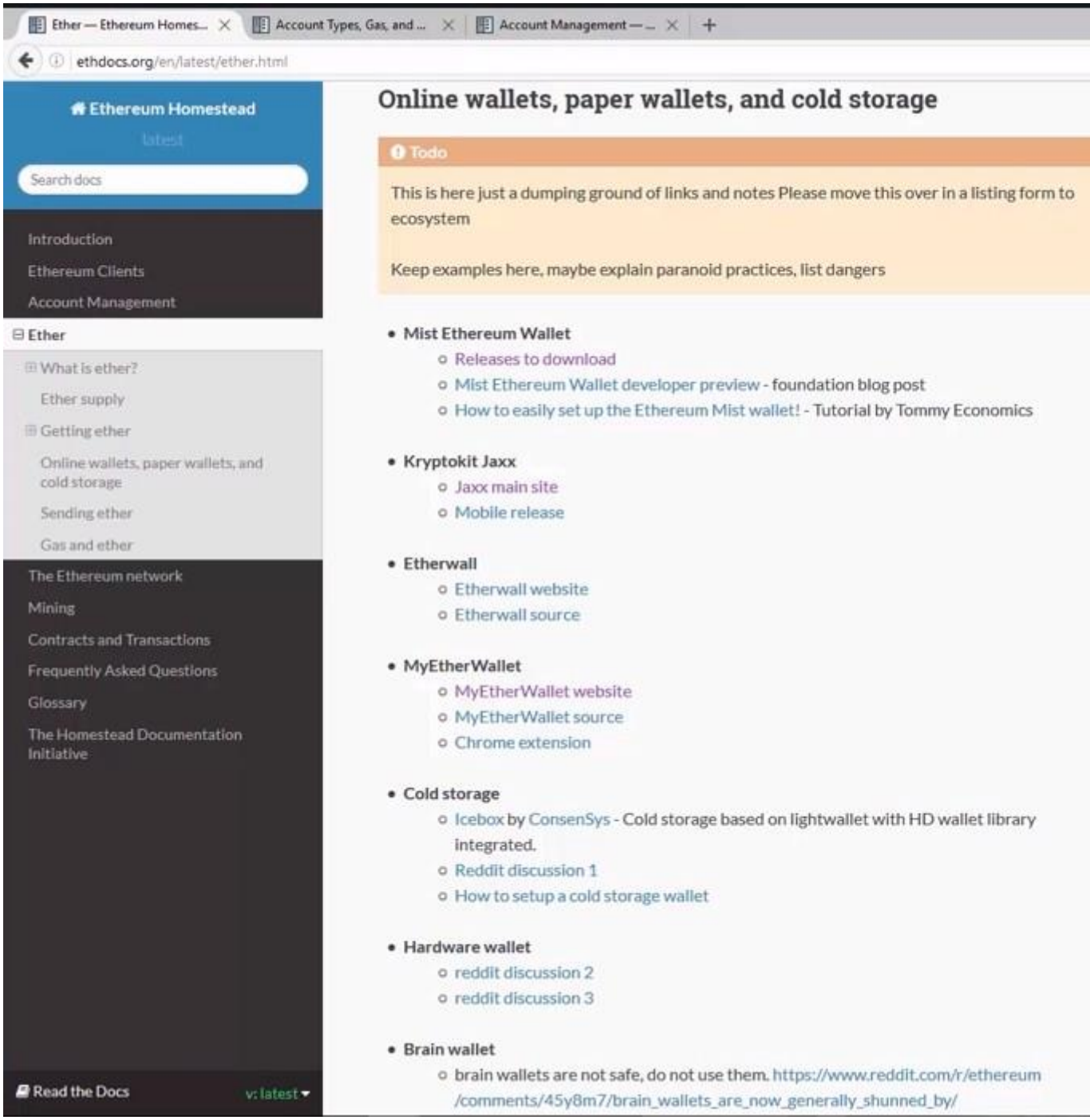


Она используется для взаиморасчетов за проведение вычислений. Эфир может дробиться на части, самая маленькая часть - основа валюты - называется вей. Помимо вея, есть другие названия для дробных частей эфира. Один эфир - это квинтиллион, или единица с восемнадцатью нулями, веев.

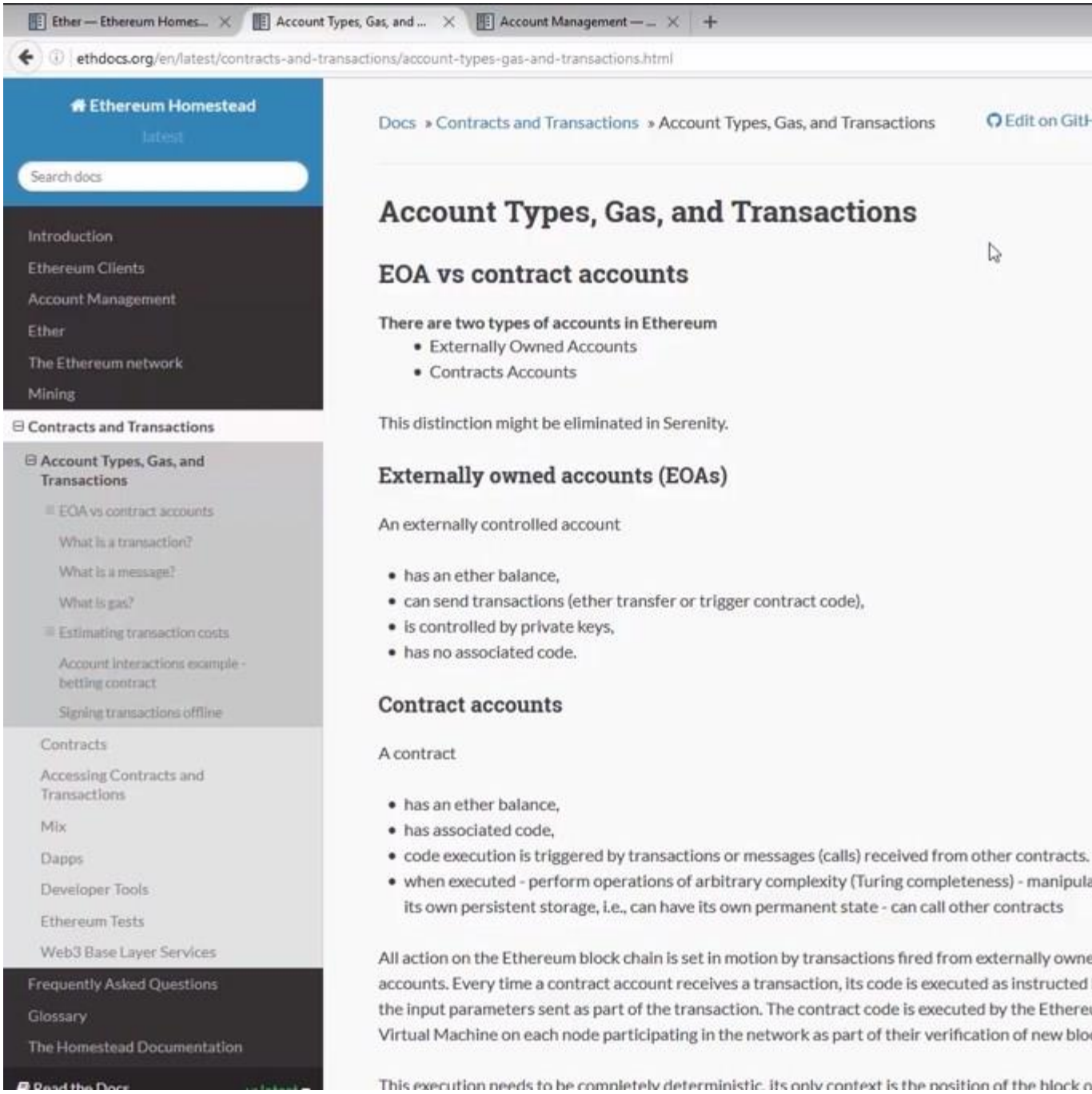
Эфир можно получить разными способами. Можно стать майнером эфира, или обменять его на другие валюты на сайтах обменных платформ. Третьим способом является загрузка кошелька Mist, получение адреса в блокчейне и использование сервиса , например, ShapeShift для покупки эфира.



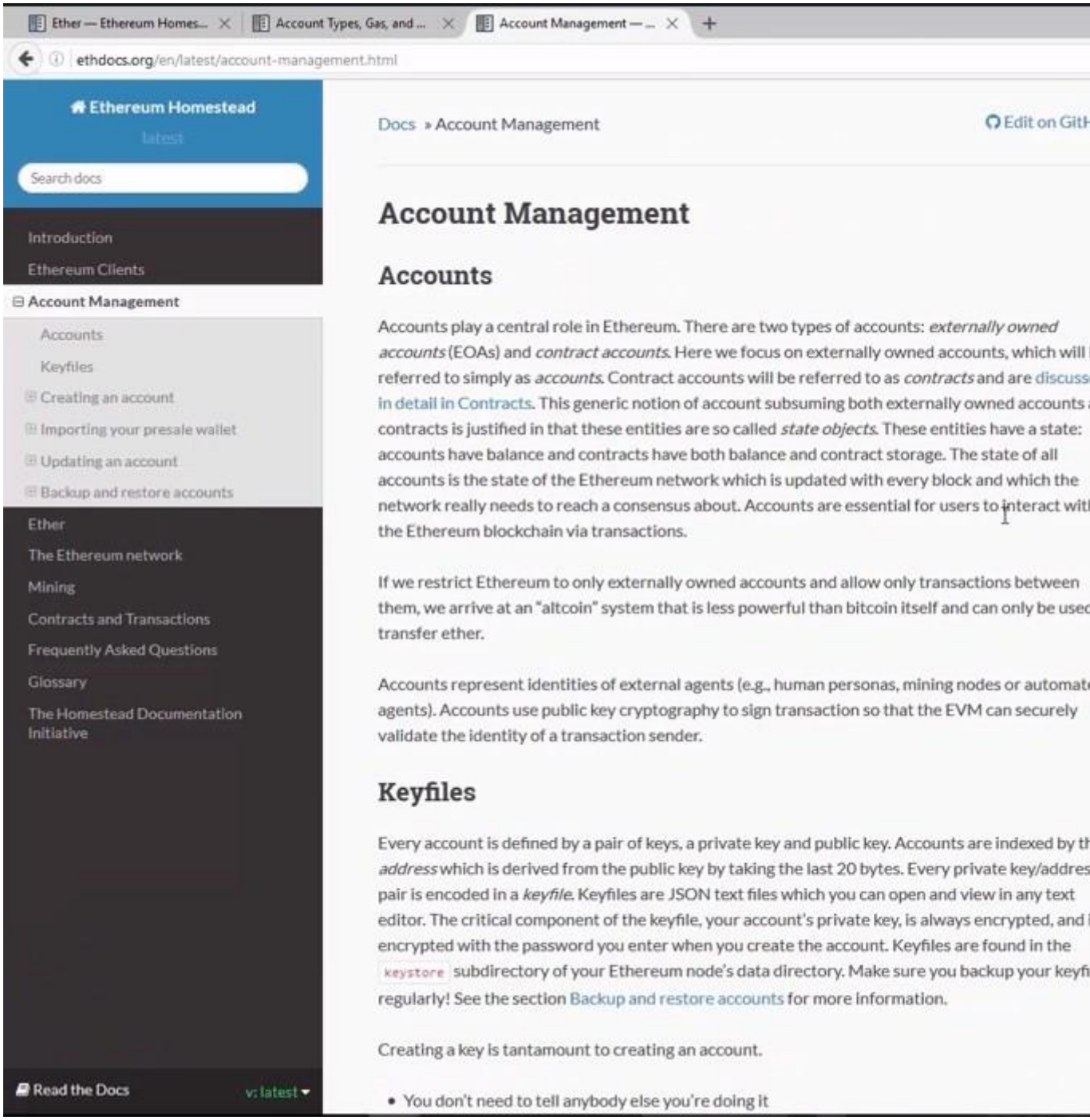
Список площадок для обмена традиционных валют на эфир.



Не обязательно использовать Mist, для создания и управление счетами с эфиром можно также использовать чеки или кошельки Etherwall, MyEtherWallet, ColdStorage, Hardware или другие.
В среде Ethereum есть несколько типов счетов.



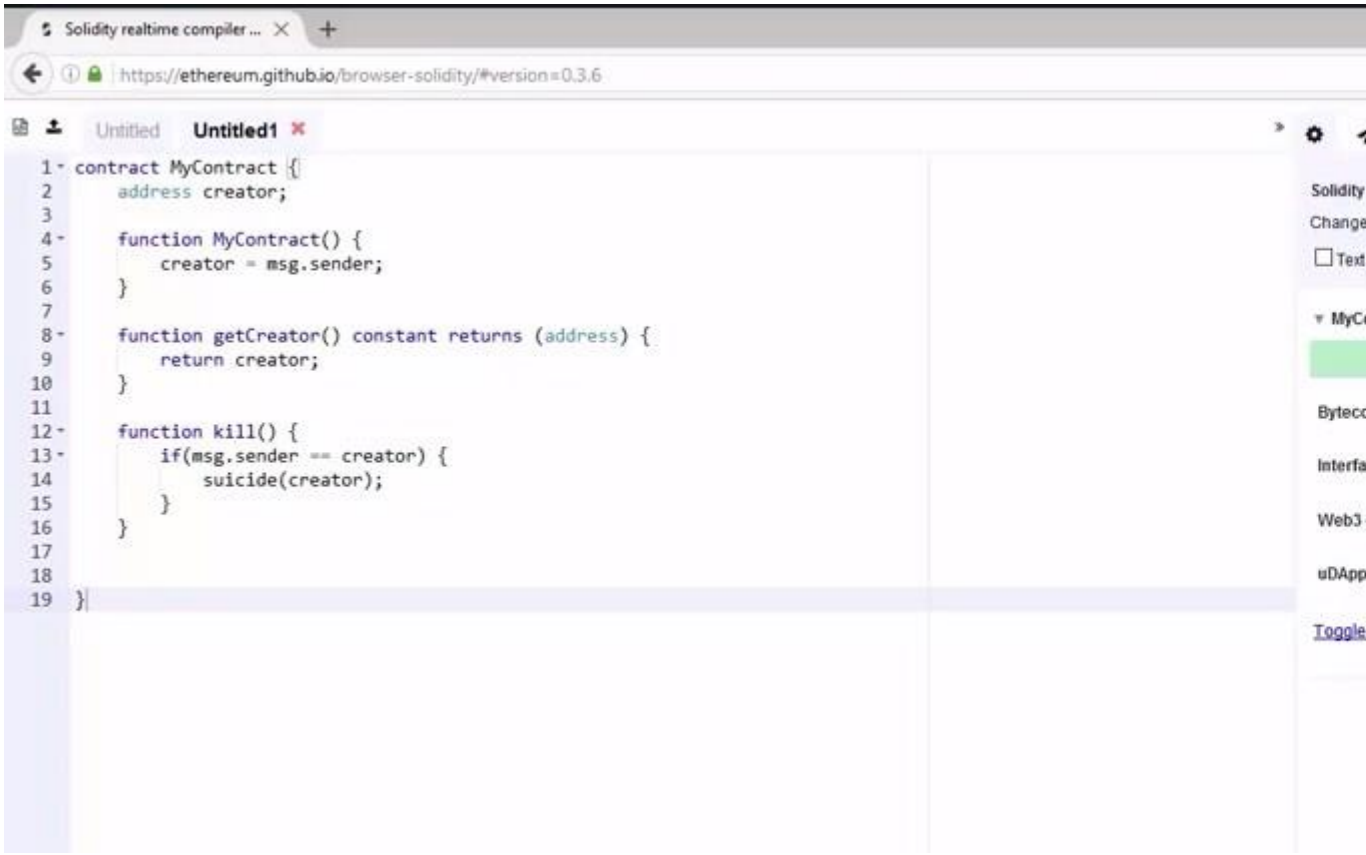
Есть внешние счета, которые можно создать в своем кошельке, назначить им пары частных и публичных ключей и пароли. Вторым типом счета является контрактный счет. Такие счета размещаются в блокчейне и содержат программный код. Внешние счета, или *счета*, могут иметь некоторое количество эфира на счету. С их помощью можно проводить операции с использованием частных ключей, и с ними не связано какого-либо программного кода. Контрактные счета, или *контракты*, тоже содержат некоторое количество эфира и несут некоторый программный код. Запуск кода осуществляется при помощи транзакций - специальных сообщений, а когда этот код работает, он может выполнять в блокчейне операции различной сложности. Он может управлять своим хранилищем данных, иметь некоторое стабильное состояние и, наконец, может вызывать другие контракты.



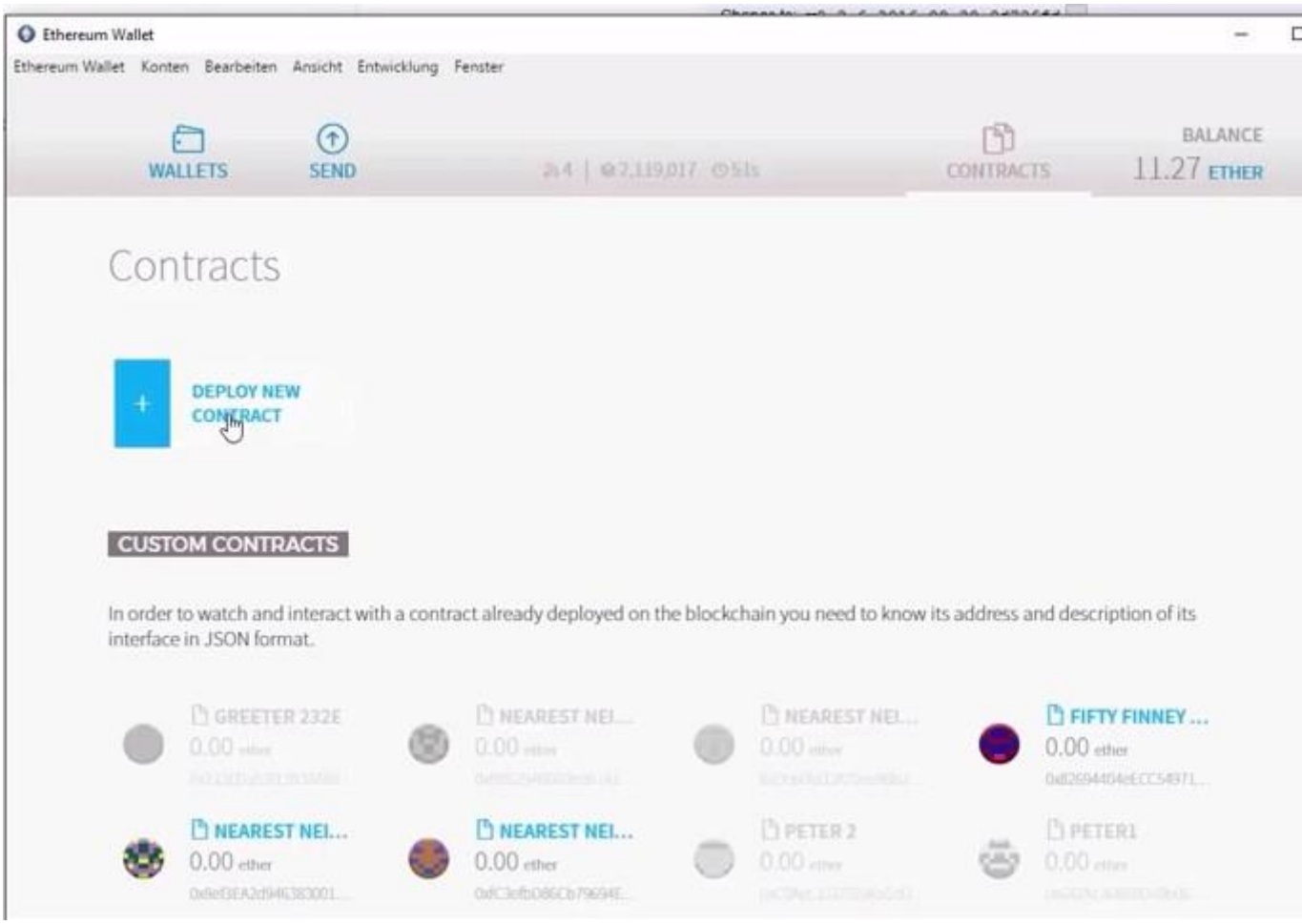
Счета в среде Ethereum играют ключевую роль, особенно счета, которые есть в вашем кошельке и управляются файлами-ключами. Если вы создаете счет в своем кошельке, автоматически создается пара публичного и частного ключей, а также задается пароль для доступа к счету. Создать счет в Ethereum легко, для этого не требуется быть подключенным к блокчейну или Интернету. Вы просто создаете новую пару ключей, и ваш частный ключ будет ассоциирован со счетом.

Как создавать, обрабатывать и размещать контракты

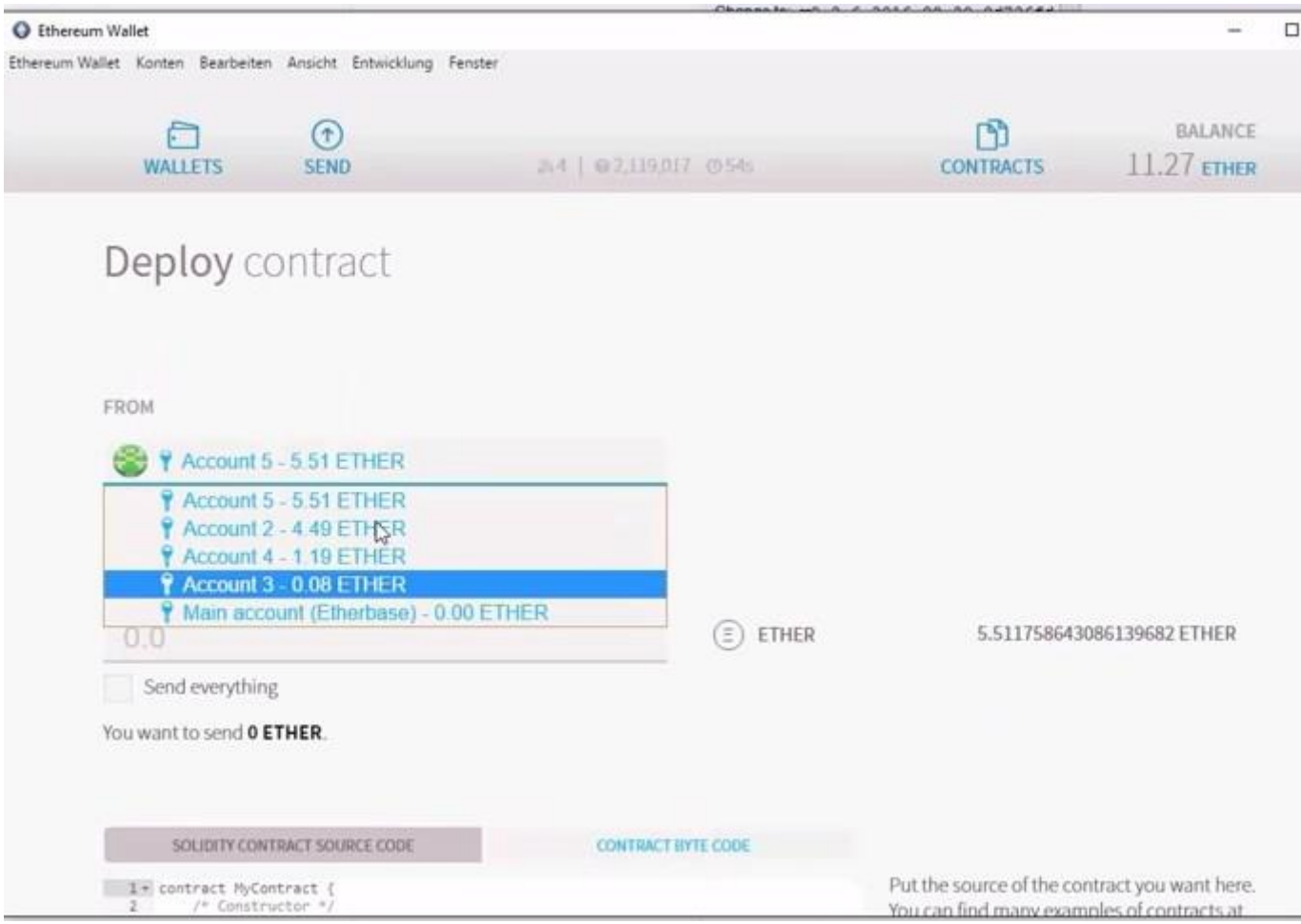
Откроем браузер Solidity, в котором можно эмулировать блокчейн. Загрузить Solidity можно по ссылке http://ethereum.github.io/browser-solidity.



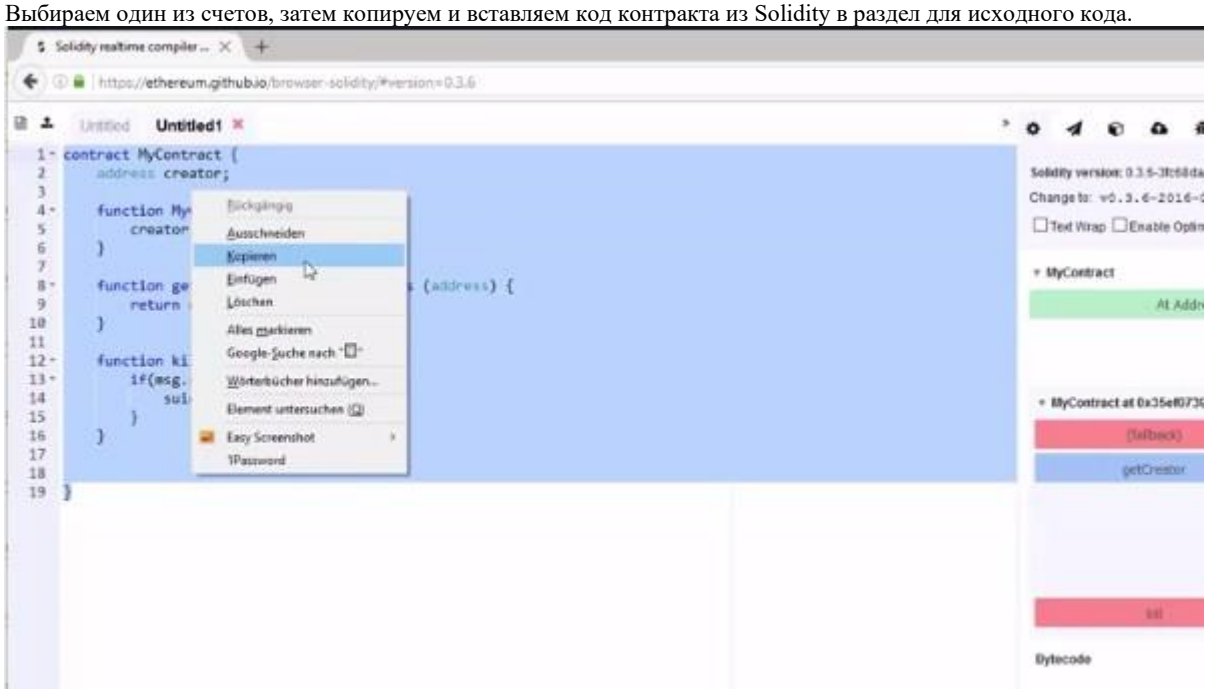
Слева отображается ваш код. Справа показаны его интерпретации в байткоде, интерфейсном коде и коде Web3. Браузер Solidity удобен для работы и тестирования контрактов. Можно быстро создавать контракты и сразу выполнять с ними нужные операции. Стоит помнить, что это просто симуляция и контракты не обрабатываются в настоящем блокчейне. Для размещения контракта в блокчейне и предоставления доступа к нему другим воспользуемся кошельком Mist.

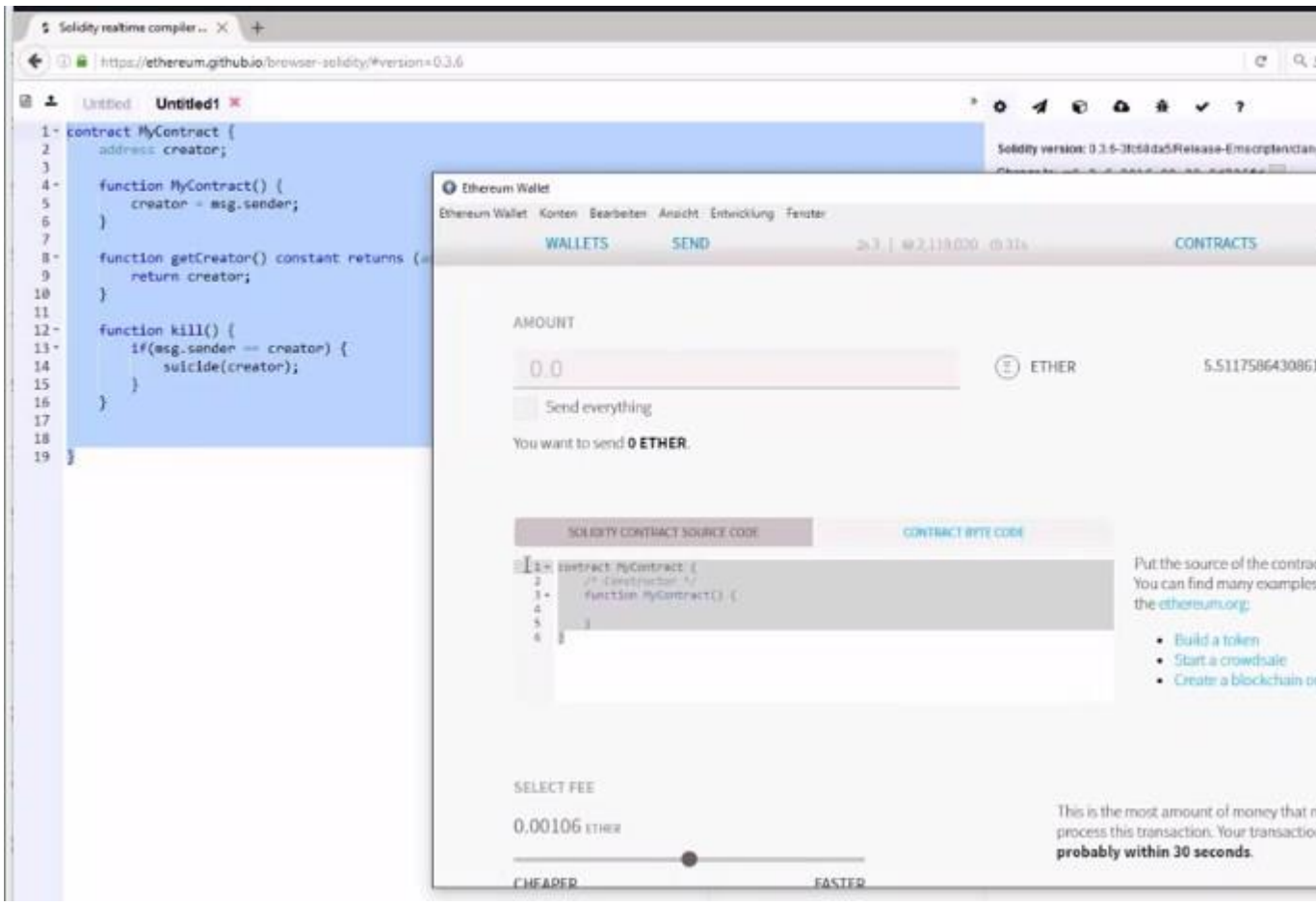


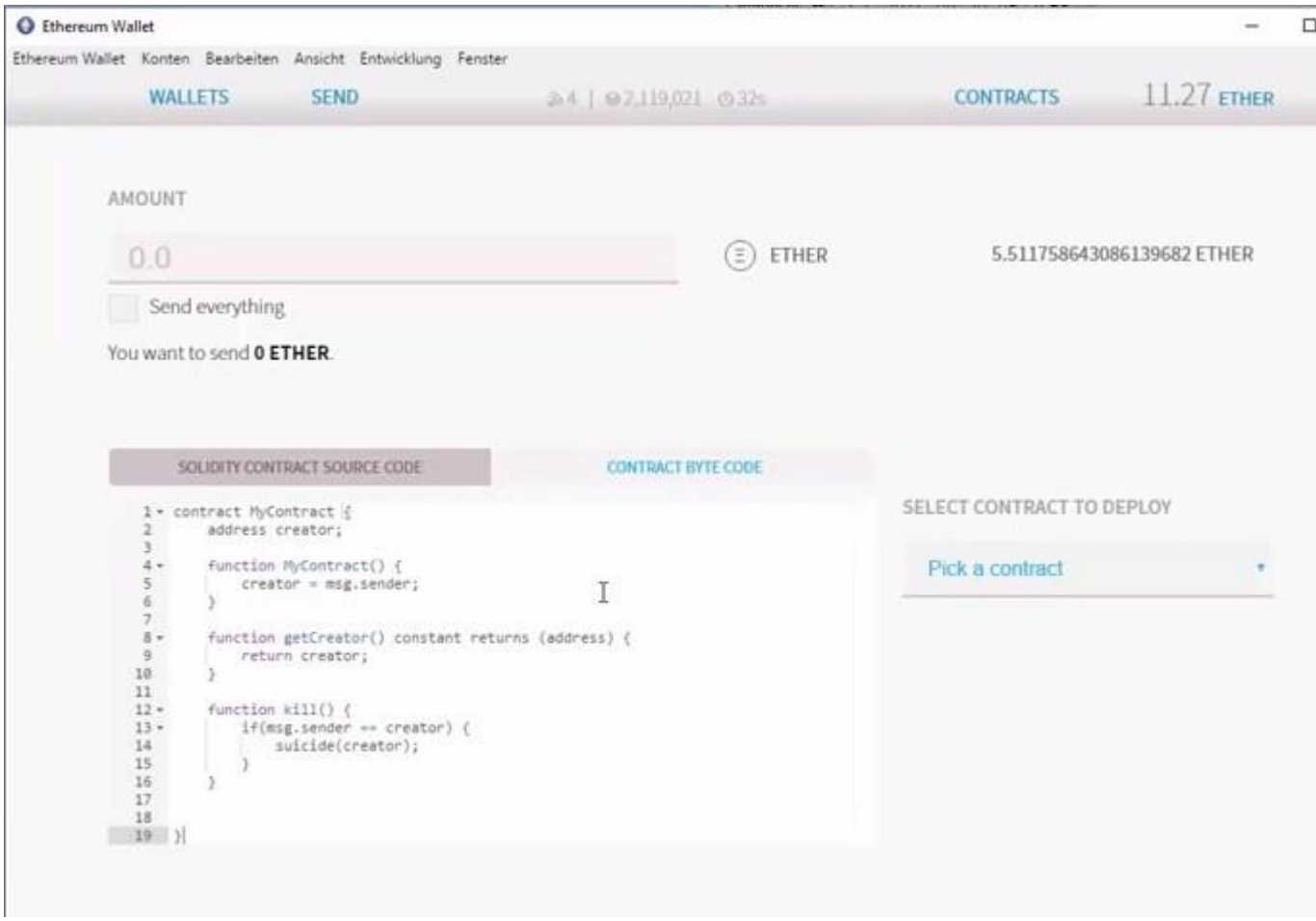
Для создания нового контракта откройте раздел Contracts и щелкните по кнопке Deploy a New Contract, затем выберите счет с достаточным количеством средств. Для создания контракта не потребуется слишком много эфира: от 0,5 до 1 единицы.



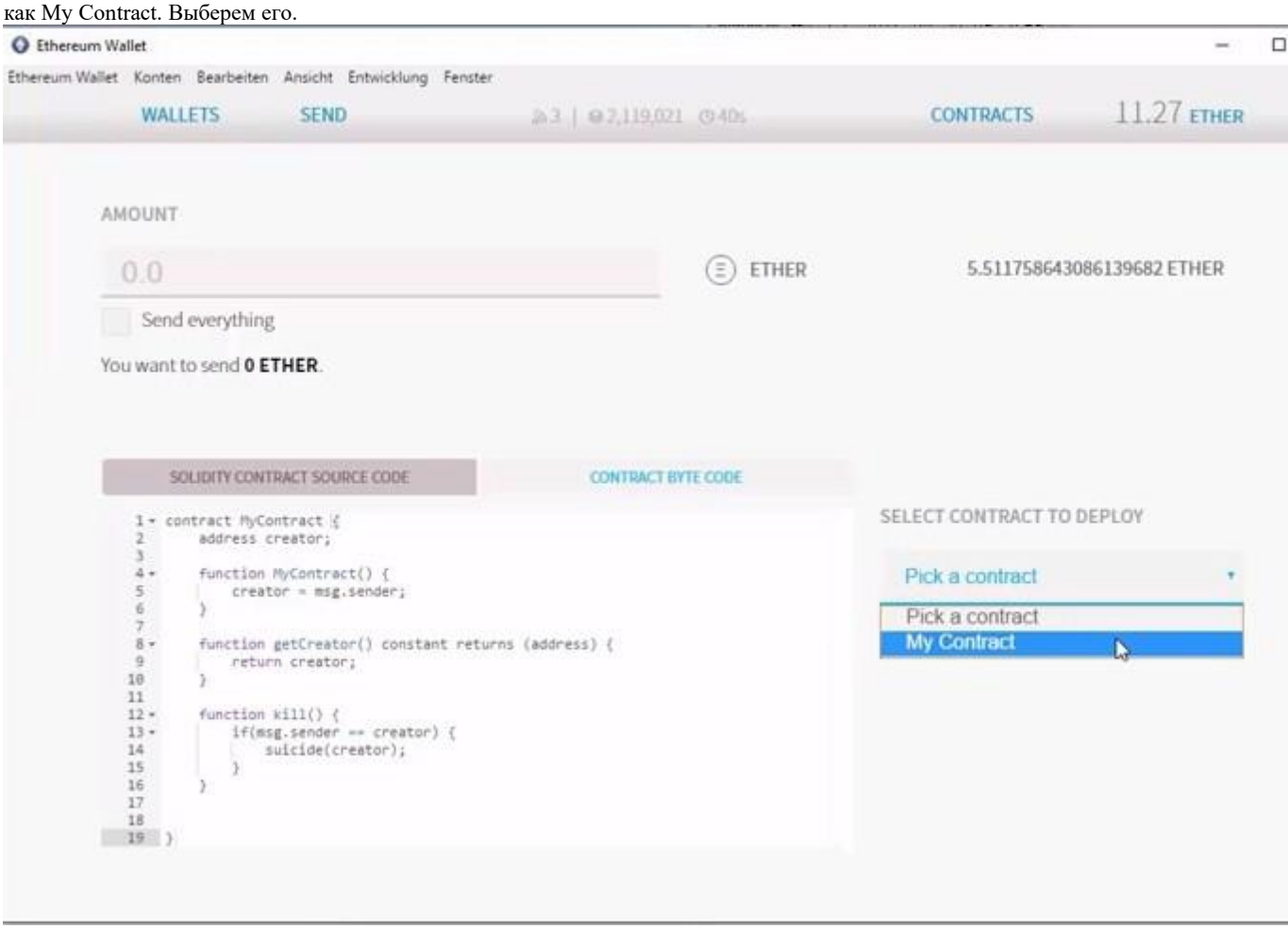
Выбираем один из счетов, затем копируем и вставляем код контракта из Solidity в раздел для исходного кода.



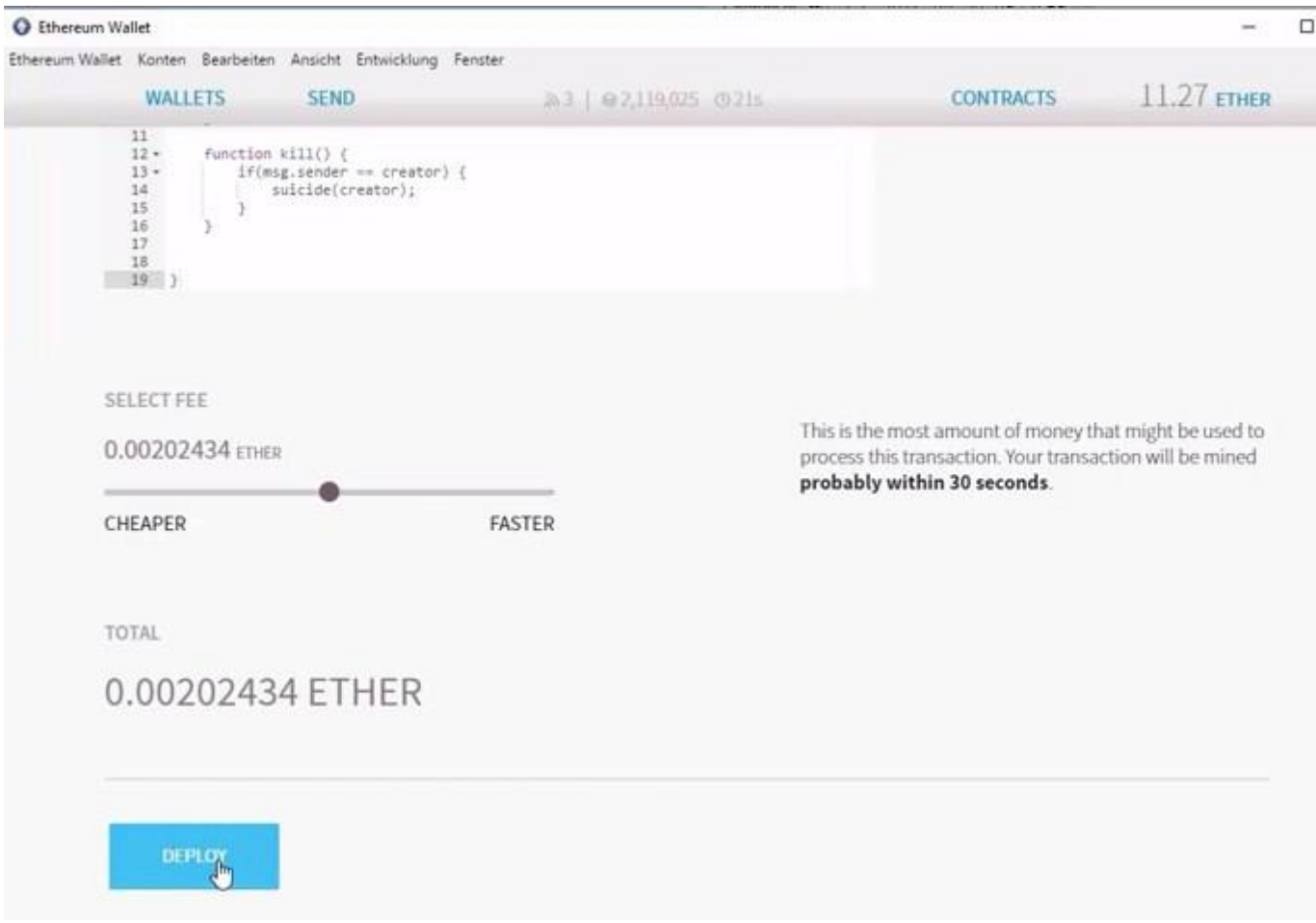

Solidity realtime compiler ...
https://ethereum.github.io/browser-solidity/#version=0.3.6
Untitled
Untitled1
contract MyContract {
address creator;
function MyContract() {
creator = msg.sender;
}
function getCreator() constant returns (
return creator;
}
function kill() {
if(msg.sender == creator) {
suicide(creator);
}
}
}
Ethereum Wallet
Ethereum Wallet Konten Bearbeiten Ansicht Entwicklung Fenster
WALLETS
SEND
CONTRACTS
AMOUNT
0.0
ETHER
Send everything
You want to send 0 ETHER.
SOLIDITY CONTRACT SOURCE CODE
CONTRACT BYTE CODE
SELECT FEE
0.00106 ETHER
CHEAPER
FASTER
probably within 30 seconds.


Ethereum Wallet
Ethereum Wallet Konten Bearbeiten Ansicht Entwicklung Fenster
WALLETS
SEND
4 | 2,119,021 32s
CONTRACTS
11.27 ETHER
AMOUNT
0.0
ETHER
5.511758643086139682 ETHER
Send everything
You want to send 0 ETHER.
SOLIDITY CONTRACT SOURCE CODE
CONTRACT BYTE CODE
contract MyContract {
address creator;
function MyContract() {
creator = msg.sender;
}
function getCreator() constant returns (address) {
return creator;
}
function kill() {
if(msg.sender == creator) {
suicide(creator);
}
}
}
SELECT CONTRACT TO DEPLOY
Pick a contract

Контракт начнёт компилироваться в фоновом режиме. Он отображается в списке контрактов на размещение в блокчейне как My Contract. Выберем его.

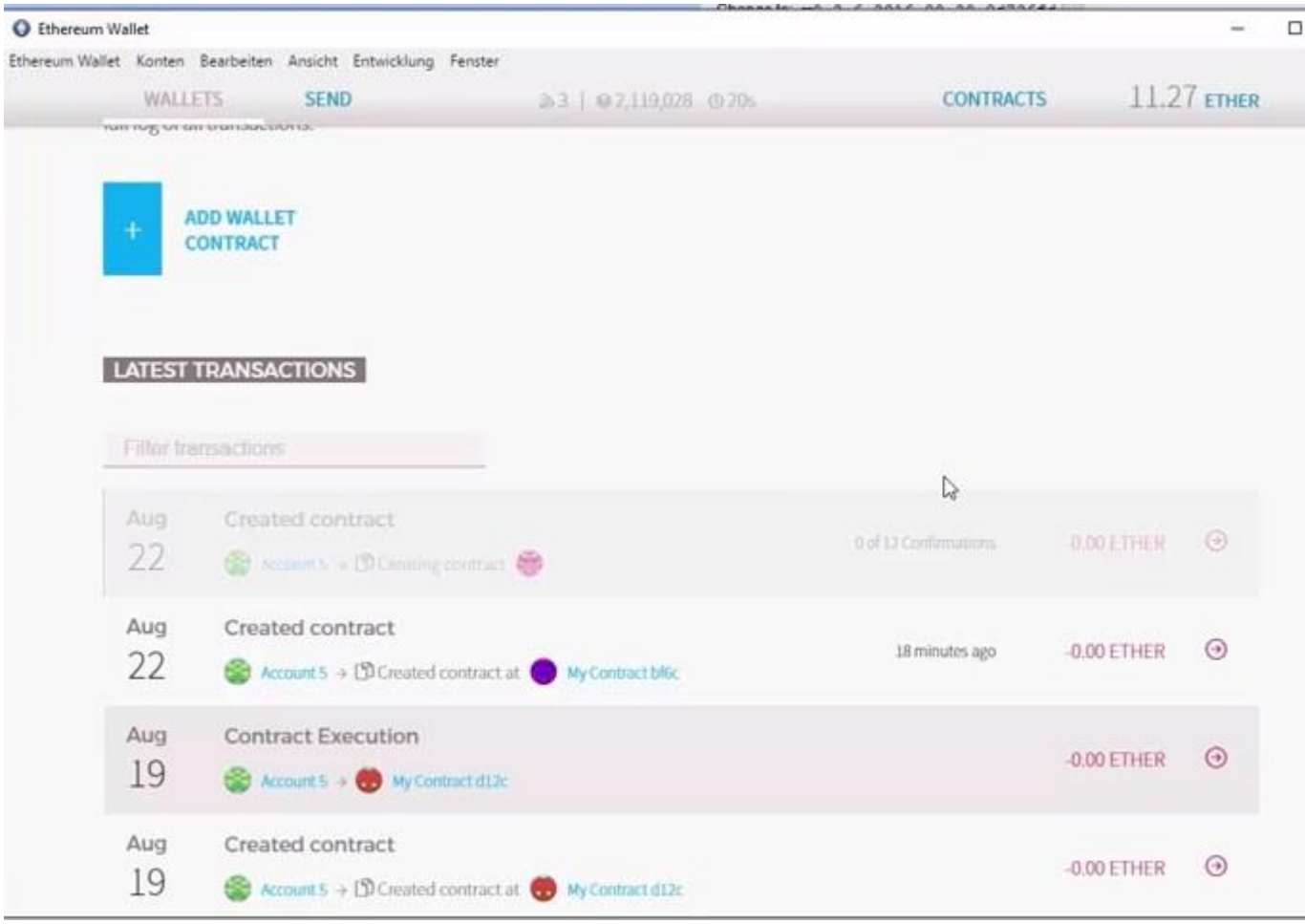


Теперь перейдем к размещению контракта в блокчейне. Для совершения этой операции нужно заплатить майнерам - пользователям сети, которые будут обсчитывать контракт - любое количество эфира. Чем выше будет плата, тем быстрее контракт будет обсчитан. Обратите внимание, как скорость обработки контракта изменяется в зависимости от количества эфира, которое вы готовы заплатить. Выбрав нужные параметры, необходимо щелкнуть по кнопке Deploy для отправки контракта в блокчейн.

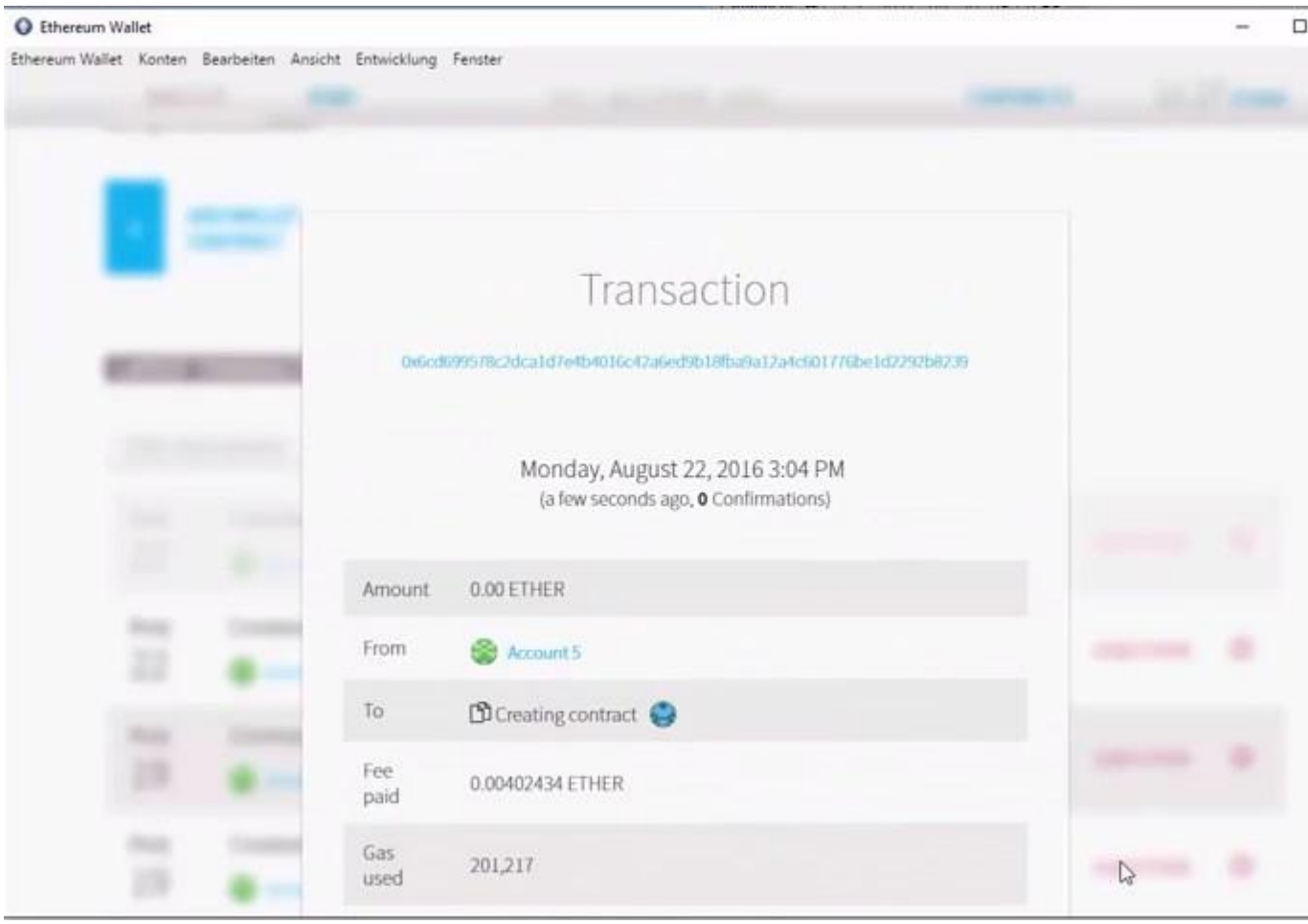


Откроется всплывающее окно для ввода пароля, заданного при создании счета. После ввода пароля щелкните по кнопке Send Transaction. Через некоторое время размещение контракта в блокчейне будет подтверждено.

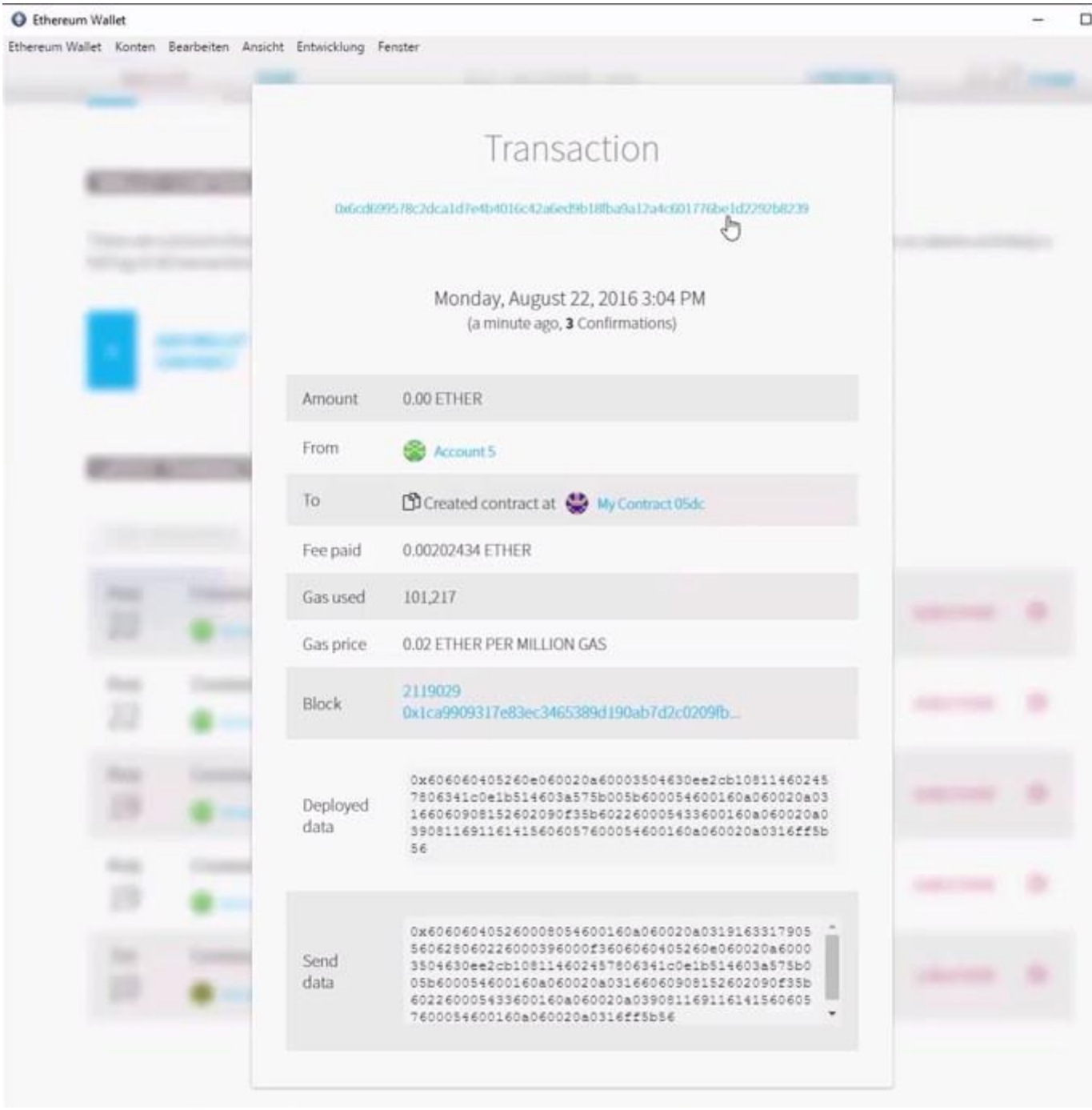
Размещенный контракт можно найти в кошельке Mist, в разделе с последними транзакциями Latest Transactions.



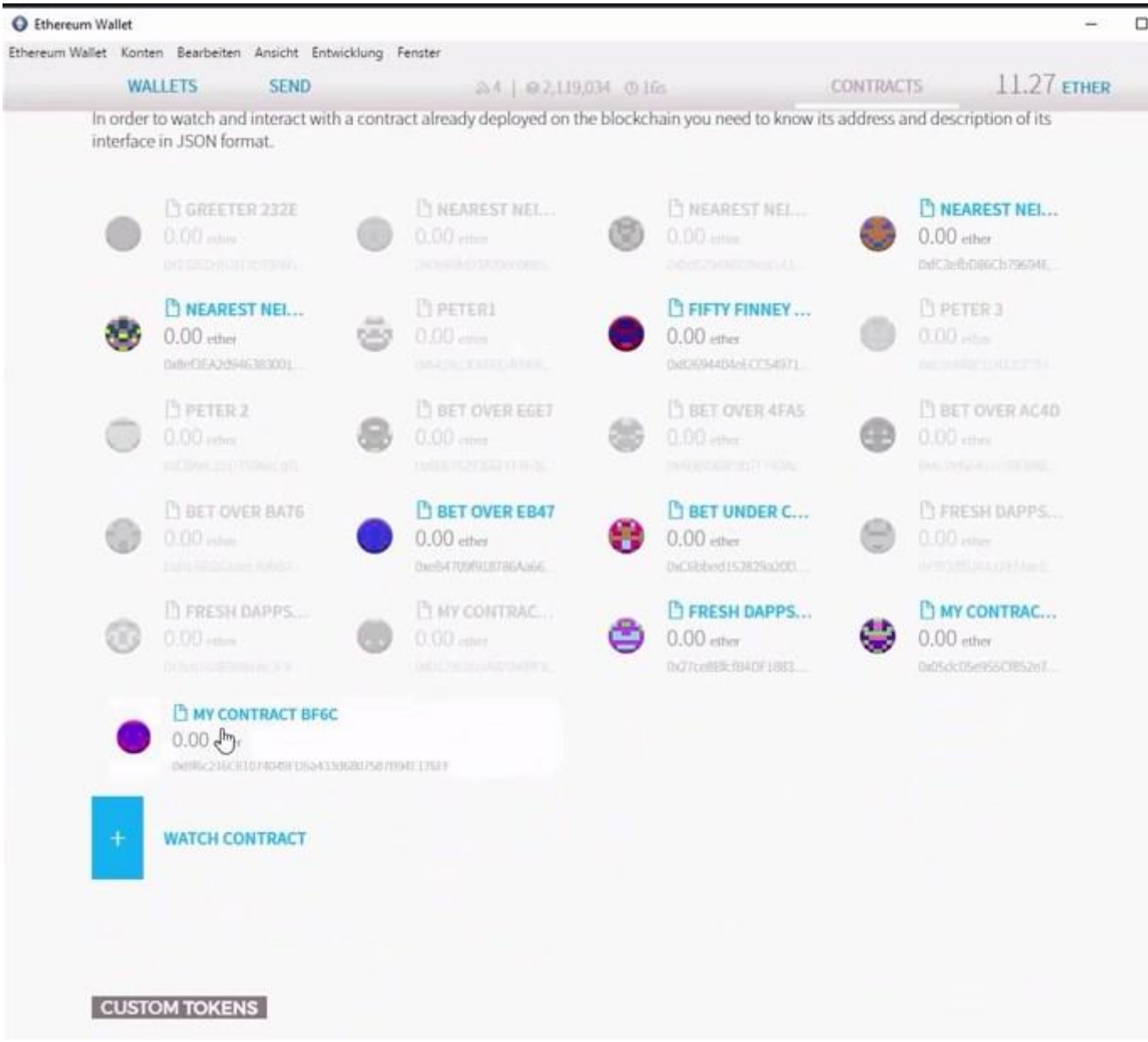
Щелкнув по контракту, можно увидеть дополнительную информацию.



Для того, чтобы контракт был взят в обработку и получил собственный адрес в блокчейне, может потребоваться некоторое время - это произойдет, как только кто-нибудь из пользователей блокчейна завершит обсчет контракта. Еще больше информации можно получить, щелкнув по идентификатору транзакции. Она отобразится в веб-приложении Etherscan.

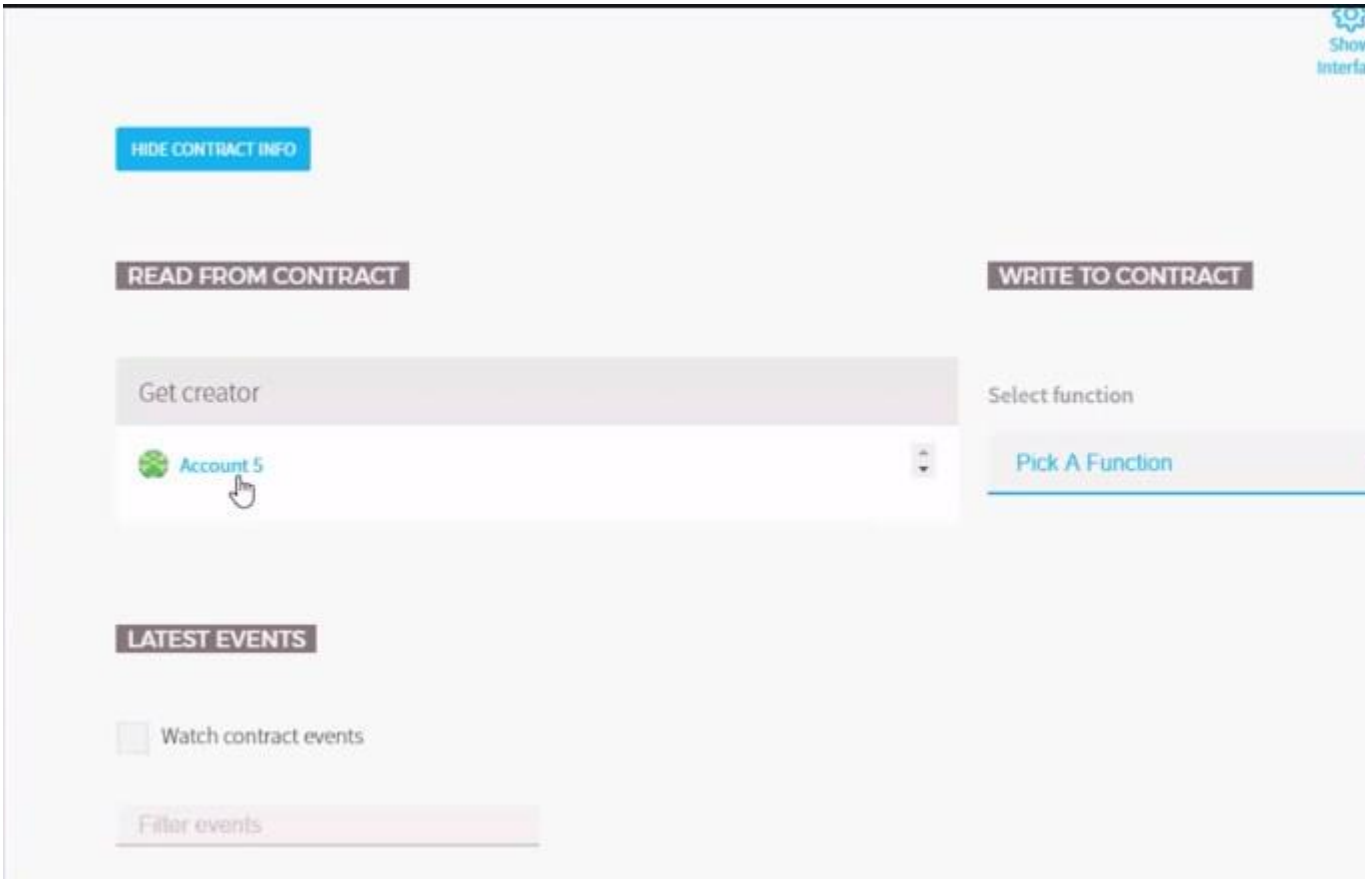


Теперь вернемся в кошелек Mist и посмотрим, как можно взаимодействовать с контрактом. Откроем раздел Contracts. Выберем контракт, который только что разместили в блокчейне (адрес контракта расположен под его названием).

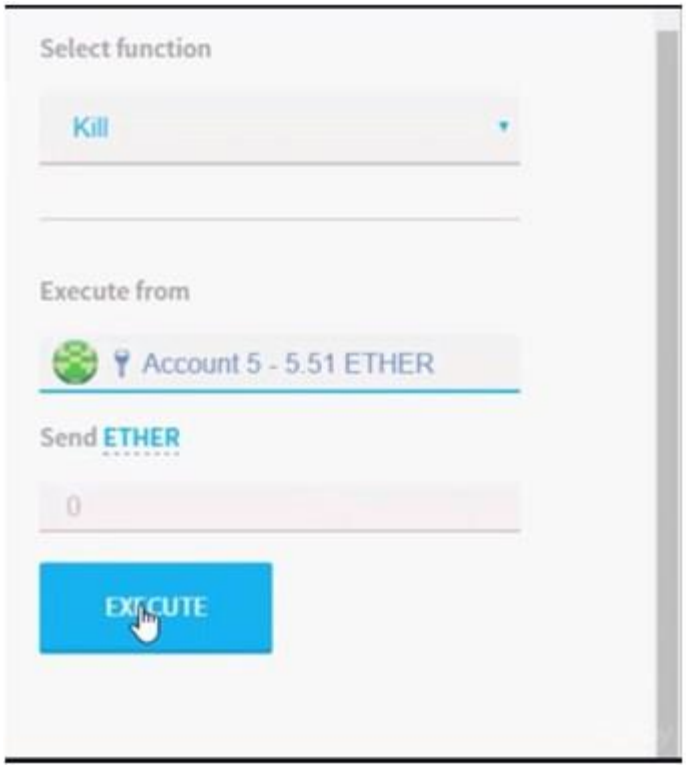


При щелчке по кнопке Show Interface отображается интерфейсный код контракта.

Эти два компонента необходимы, если возникнет желание поделиться своим контрактом с другими пользователями. Если прокрутить экран вниз, можно найти дополнительные возможности взаимодействия с контрактом.

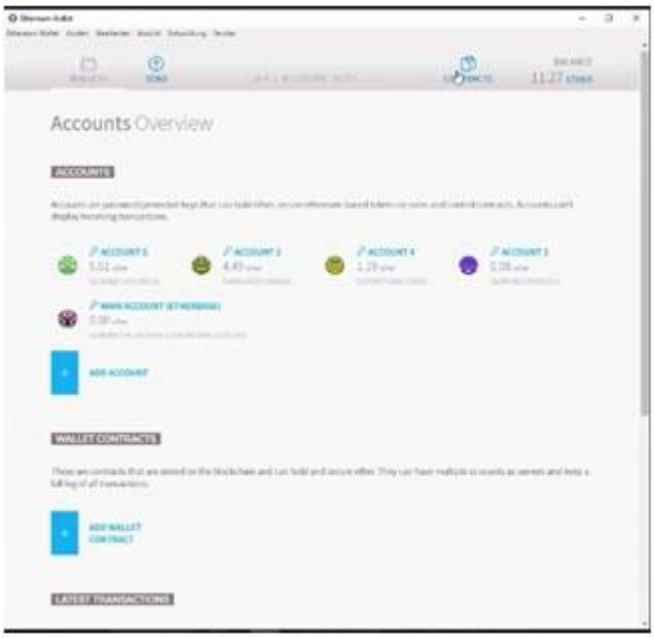


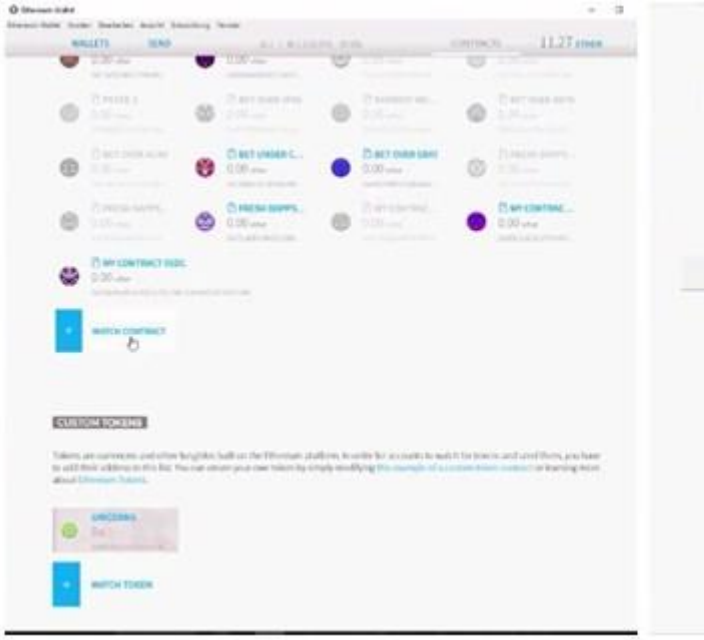
Например, в число постоянных функций входит чтение содержимого контракта. Это можно сделать без дополнительной оплаты эфира. Однако для внесения изменений в блокчейн потребуется создавать новую транзакцию. Давайте выполним процедуру по удалению контракта из блокчейна, выбрав действие Kill. Выберем тот же счет, что мы использовали для создания контракта, и выполним транзакцию. Снова потребуется ввести пароль, после чего можно щелкнуть по кнопке Send Transaction.



Вернемся в обзорное меню кошелька. Как и раньше, потребуется некоторое время для размещения транзакции в блокчейне.

Теперь рассмотрим, как поделиться своим контрактом с другими. Для этого нужно передать другому пользователю адрес контракта в блокчейне и интерфейсный код JSON.

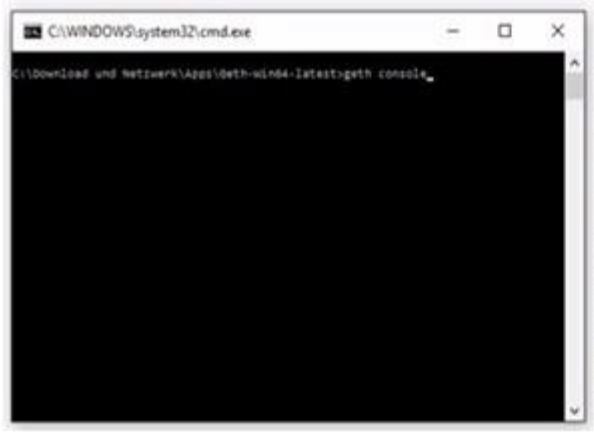


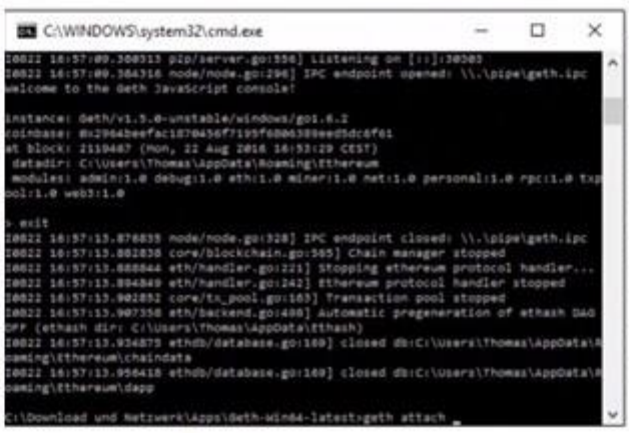


Если кто-то делится контрактом с нами, необходимо пойти в раздел Contracts и щелкнуть по кнопке Watch a contract для просмотра контракта. Присвоив контракту имя, вставляем адрес контракта в блокчейне и интерфейсный код. После этого новый контракт появится в списке рядом с существующими.

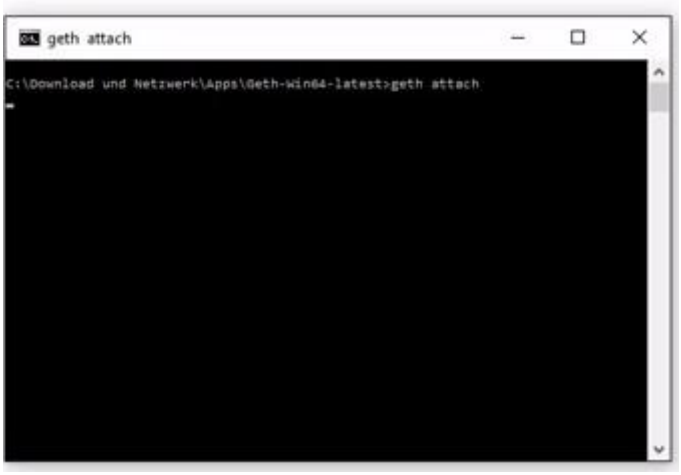
Введение в настройку и работу с клиентом Go Ethereum

Geth - это интерфейс командной строки Go Ethereum, применяется для управления узлами в сети Ethereum. Он реализован на языке Go. С помощью Geth можно добывать, или майнить, эфир, пересылать криптовалюту между адресами, создавать контракты и отправлять. По своей сути Geth представляет собой интерфейс командной строки для всех транзакций, которые можно совершать в кошельке Mist. Чтобы установить Geth, достаточно его просто загрузить на свой компьютер, например, со страницы http://github.com/Ethereum/go-ethereum. Для запуска Gethнеобходимо открыть командную строку и просто написать geth. Однако в таком виде Geth не позволит вам как-либо взаимодействовать с работающими в блокчейне транзакциями. Для взаимодействия с консолью JavaScript, нужно либо набрать команду geth console, которая запустит процесс geth и подключится к нему, либо команду geth attach, которая осуществляет подключение к уже запущенному процессу geth.

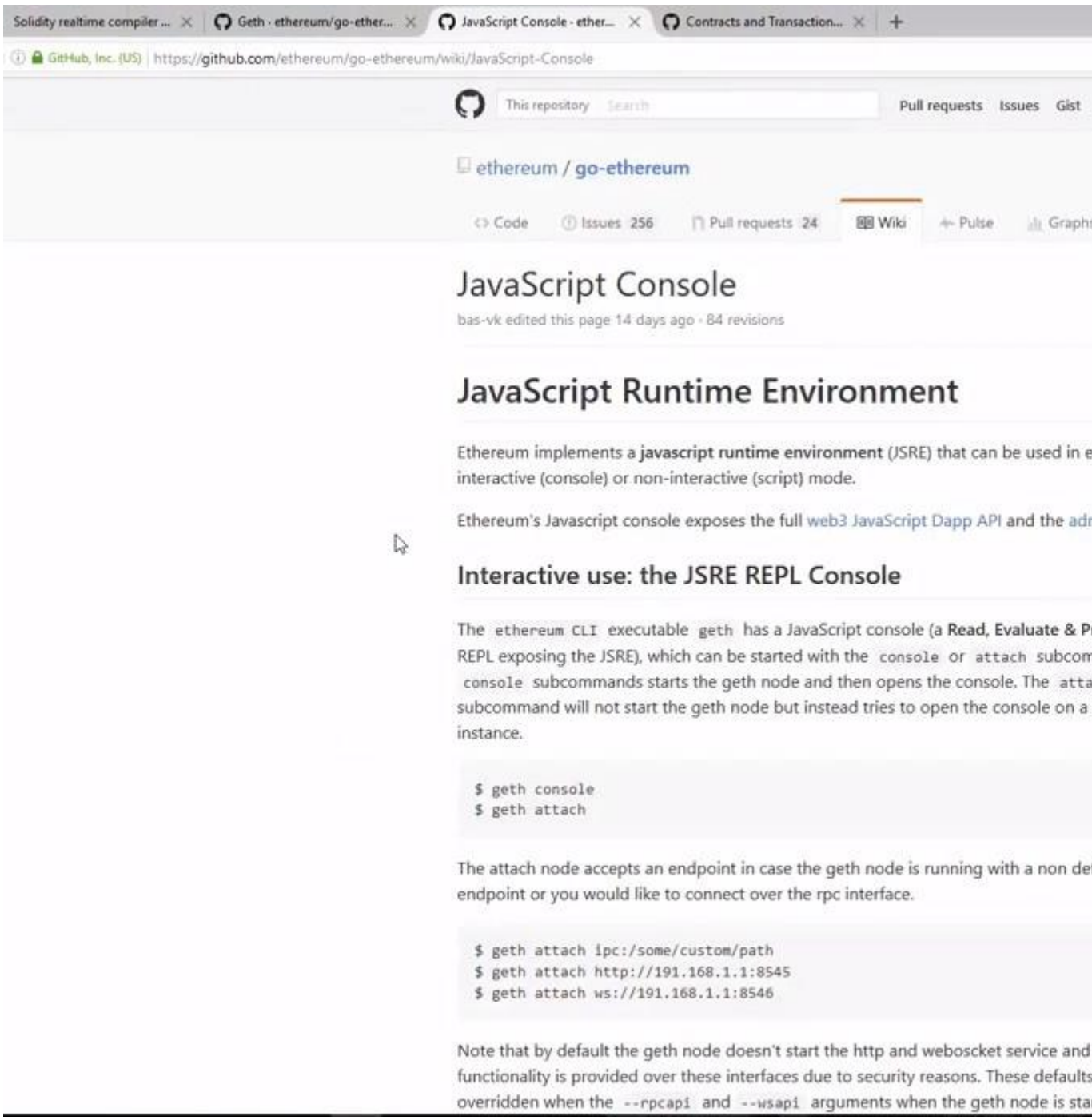




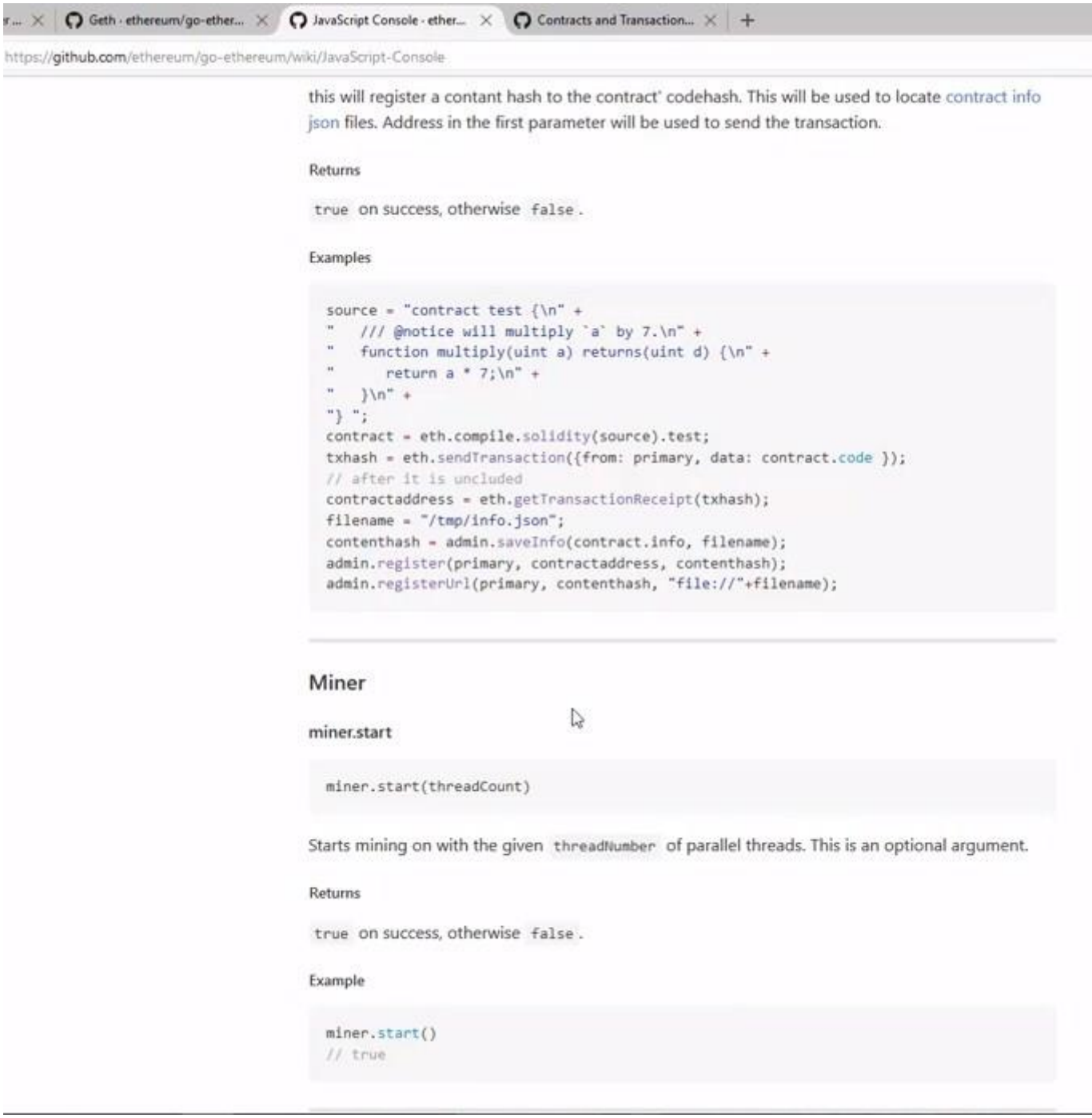
Во втором окне с командной строкой выполним команду geth attach для подключения к запущенному процессу Geth.



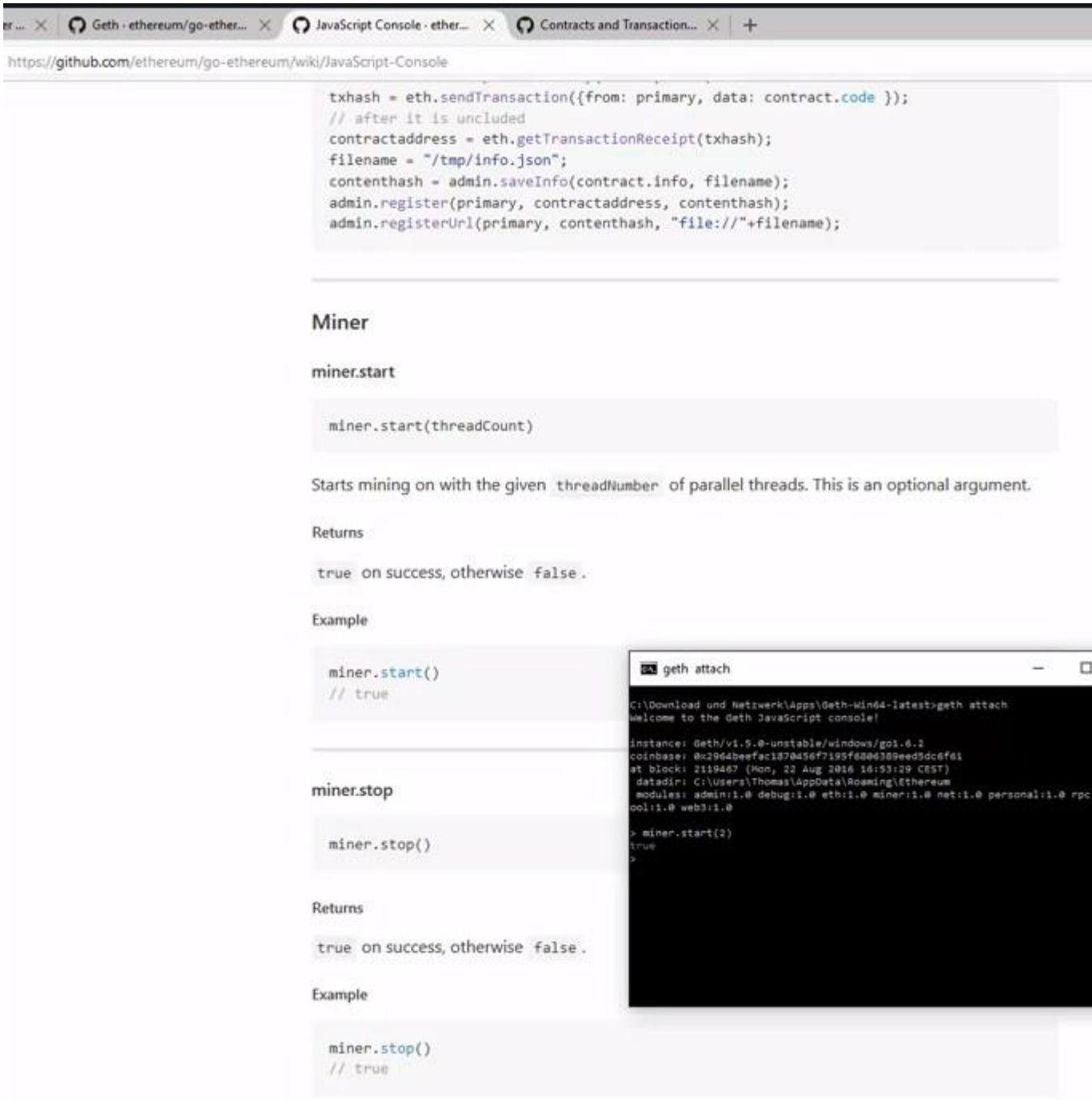
Geth работает с интерфейсом JavaScript. По JavaScript существует очень объемная и масштабная статья: начиная с получения доступа к консоли JavaScript и заканчивая изменениями уровня наполнения логов.



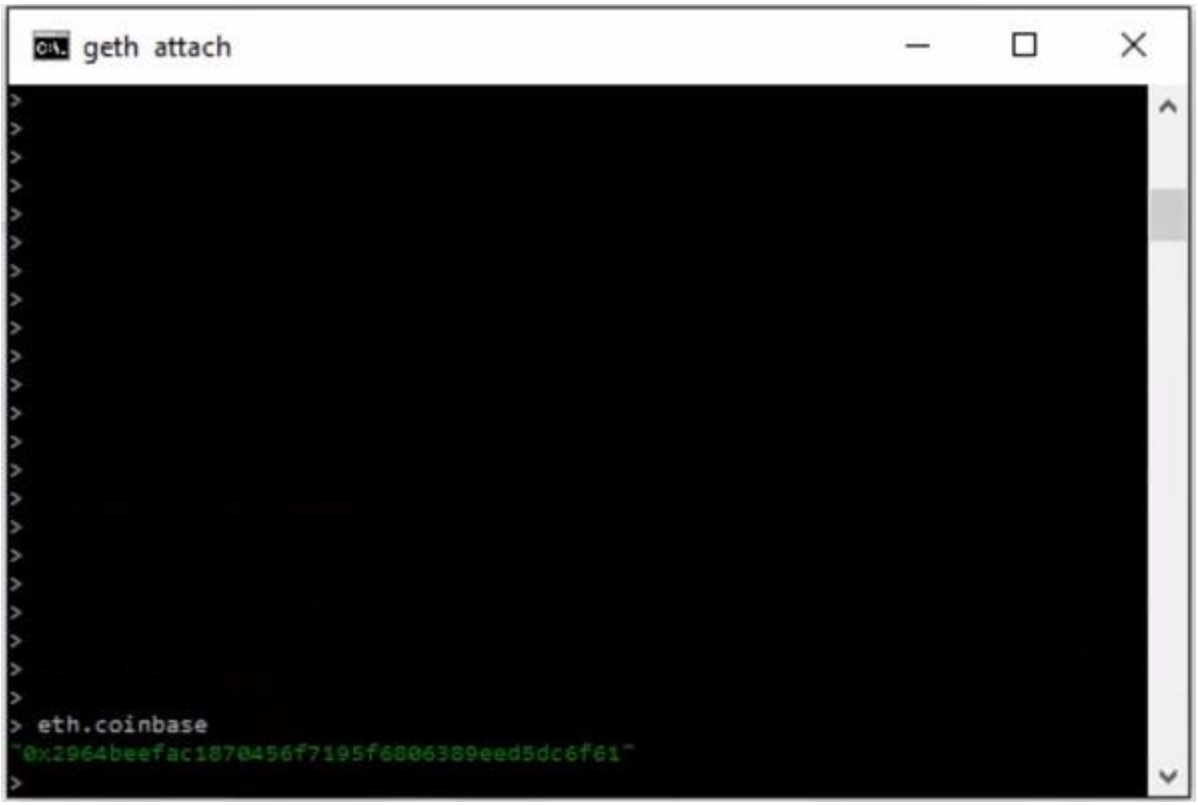
Поговори о том, как начать добывать эфир и создавать новые счета. Откроем wiki-пространство. В нижней части вики есть целый раздел про добычу, или майнинг, эфира.



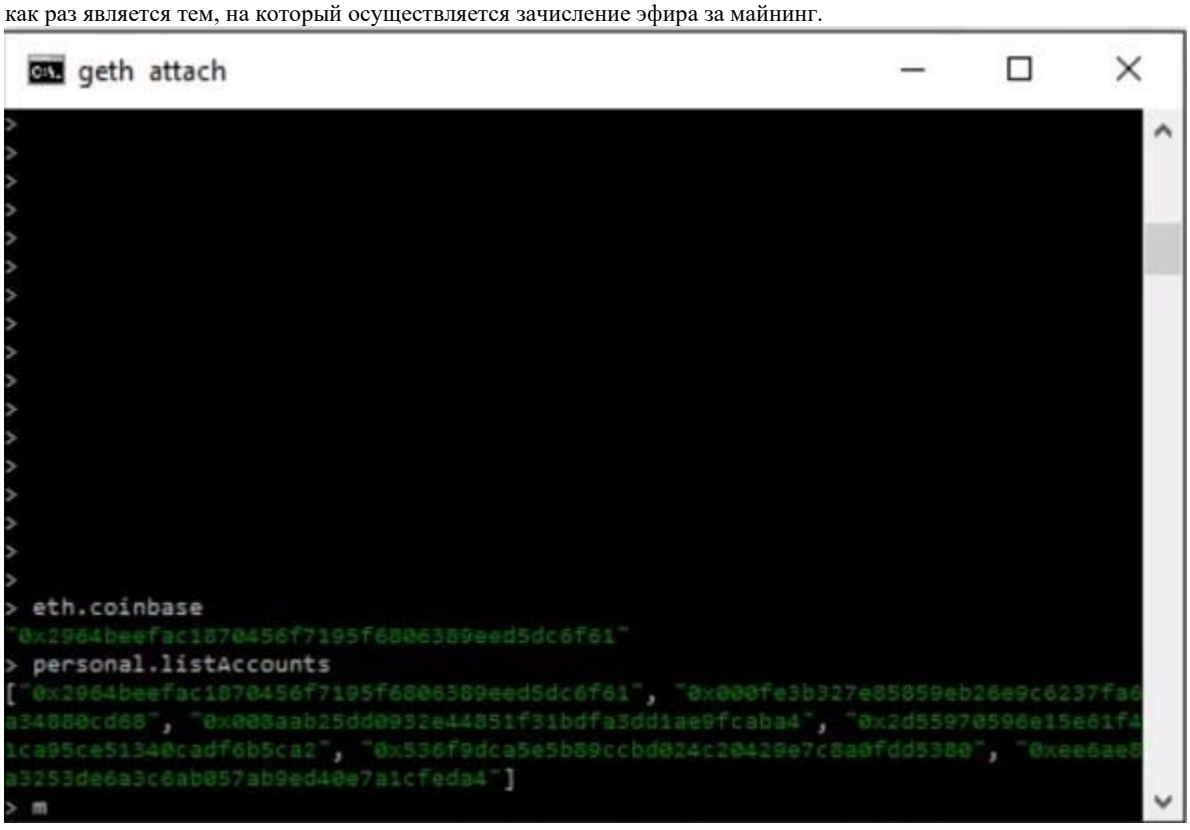
Эфир добывается путем резервирования части вычислительной мощности вашего компьютера под обсчет транзакций других пользователей сети. В консоли JavaScript обсчет данных запускается с помощью команды miner.start, в которой можно задать число потоков, например, 2.



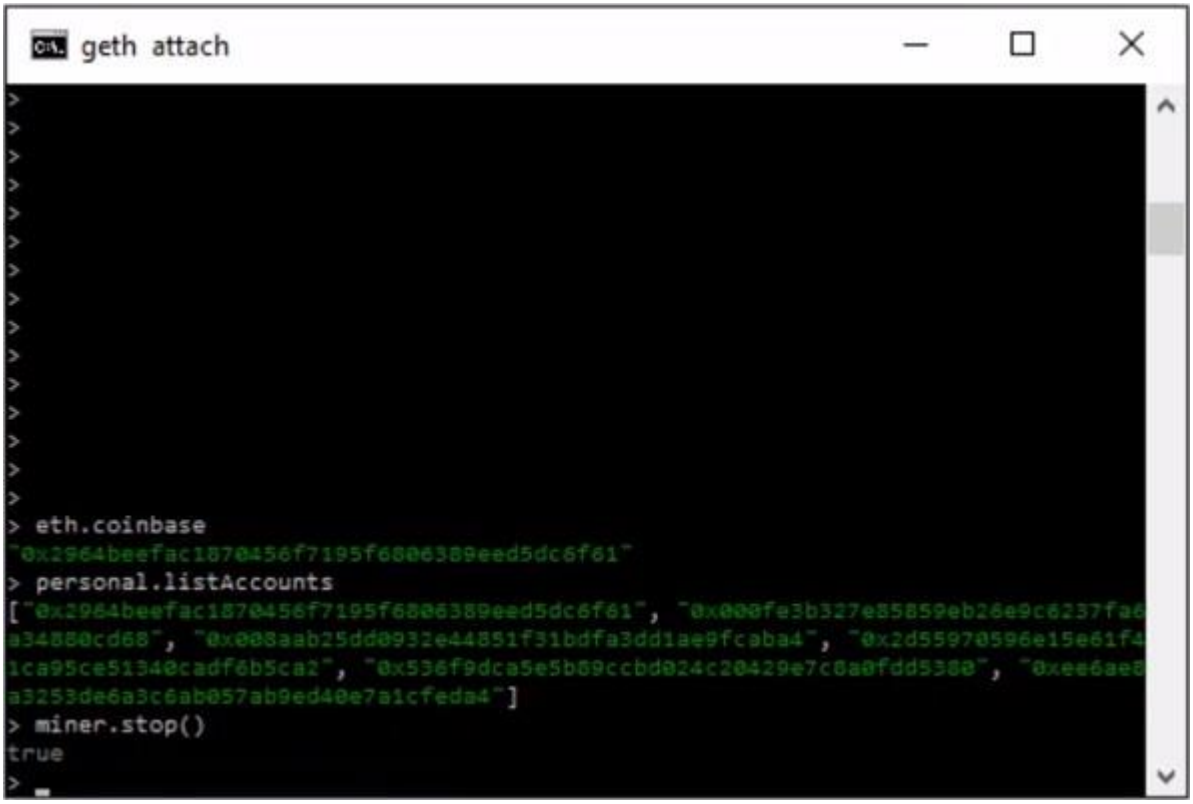
После этого Geth автоматически начинает обрабатывать транзакции, а вырученный эфир зачисляет на ваш счет. Обычно зачисление производится на тот счет, который был создан первым. Адрес счета, куда производится зачисление эфира за майнинг, отображается с помощью команды eth.coinbase.



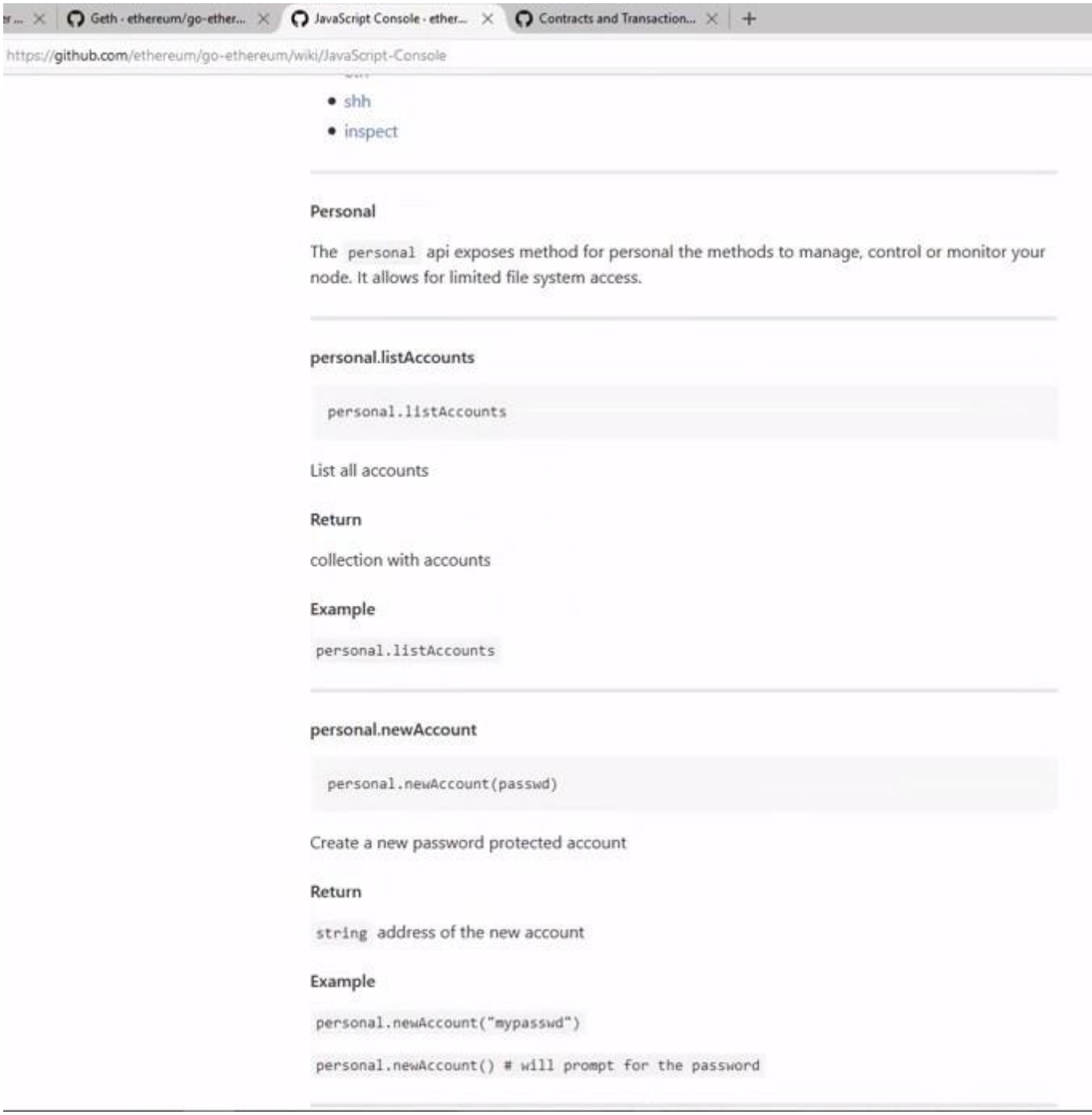
Чтобы отобразить список всех ваших счетов, можно ввести команду personal.listAccounts. Видно, что первый из счетов как раз является тем, на который осуществляется зачисление эфира за майнинг.



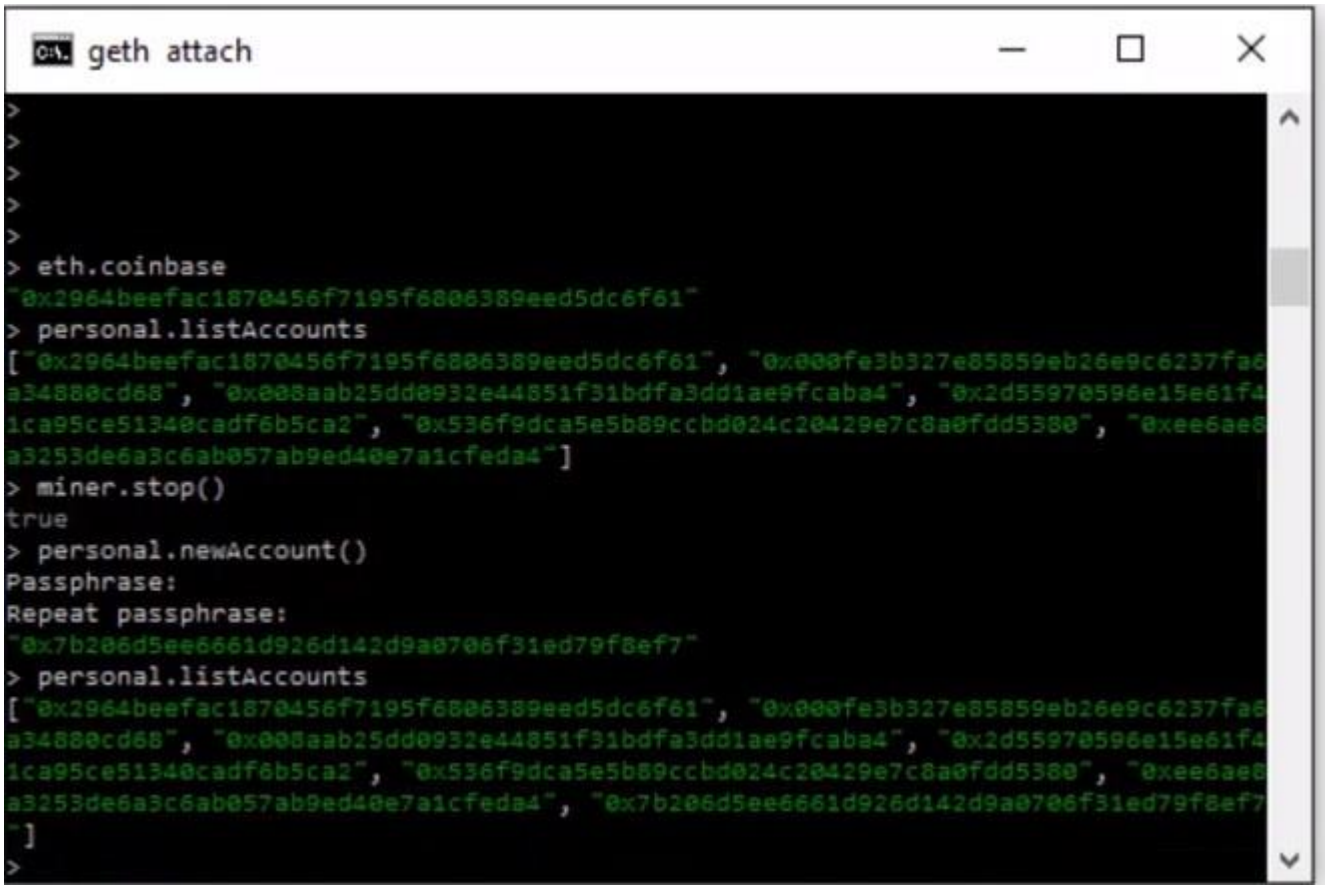
Для остановки процесса добычи эфира нужно ввести команду miner.stop().



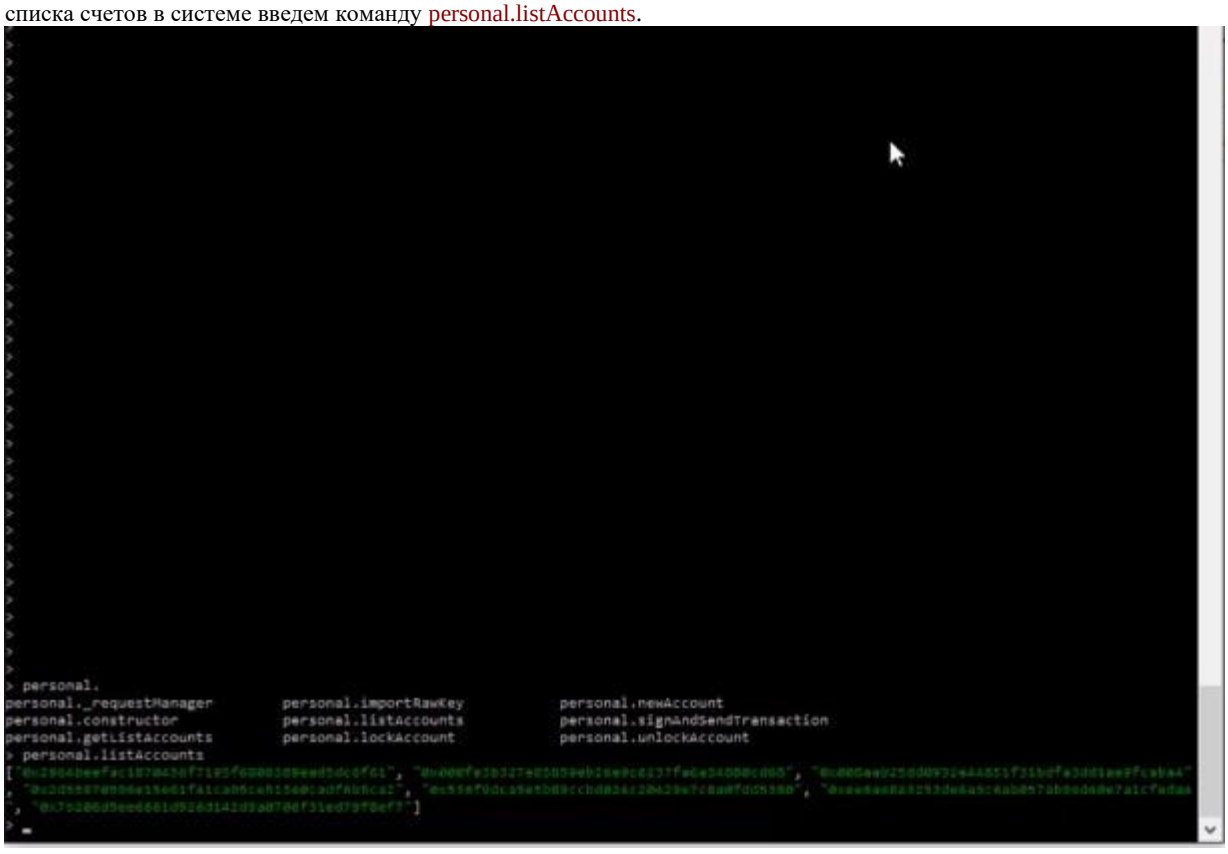
Теперь разберемся, как создать новый счет. В wiki можно найти раздел Personal, где перечислены команды для работы со своим кошельком.



Команда personal.listAccounts отображает все ваши счета, а команда personal.newAccount() создает новый счет. Для создания нового счета зададим команду personal.newAccount(),Geth попросит ввести и повторить пароль. После этого новый счет создастся и будет виден в списке счетов при вводе команды personal.listAccounts.

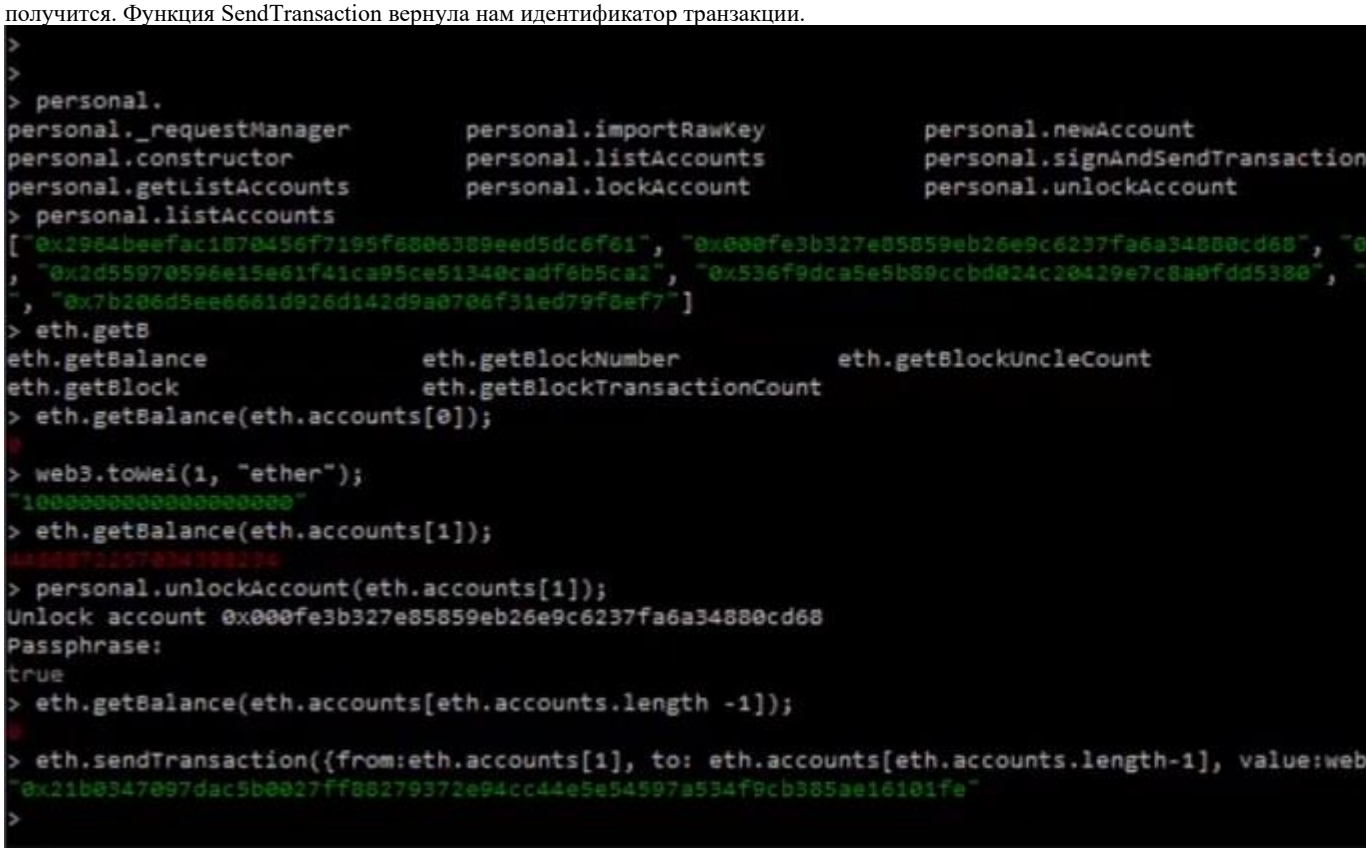


Теперь попробуем переслать немного эфира через блокчейн. В качестве счета для получения эфира я буду использовать тот счет, который был только что создан, а источником эфира выберем один из счетов, которые уже были. Для просмотра списка счетов в системе введем команду personal.listAccounts.

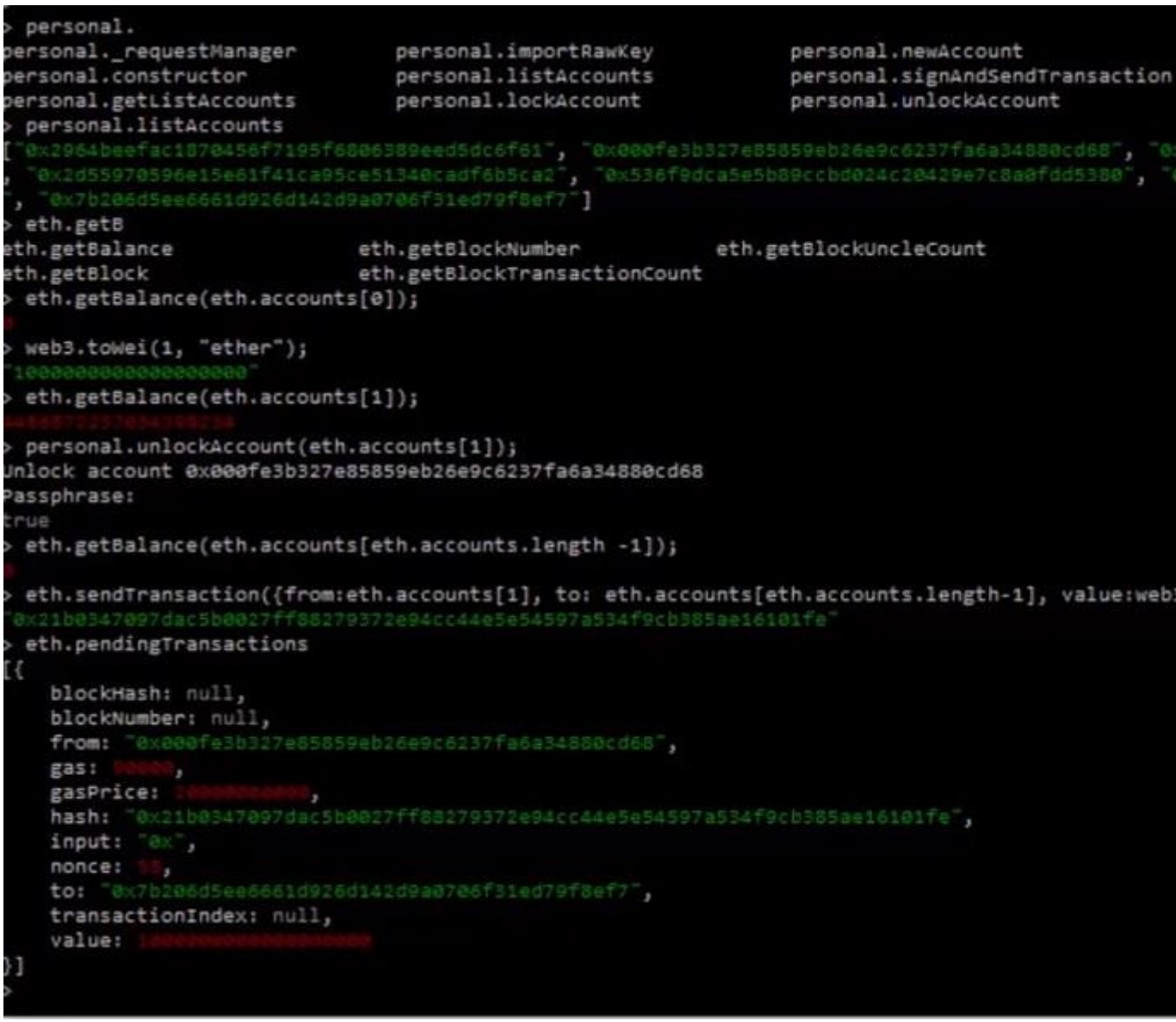


Необходимо проверить, сколько на каждом счете из списка эфира. Для этого используем отдельную команду eth.getBalance. Параметром для функции eth.getBalance является адрес счета в формате eth.account[номер счета] . Посмотрим состояние первого счета. Видно, что на этом счете ничего нет.

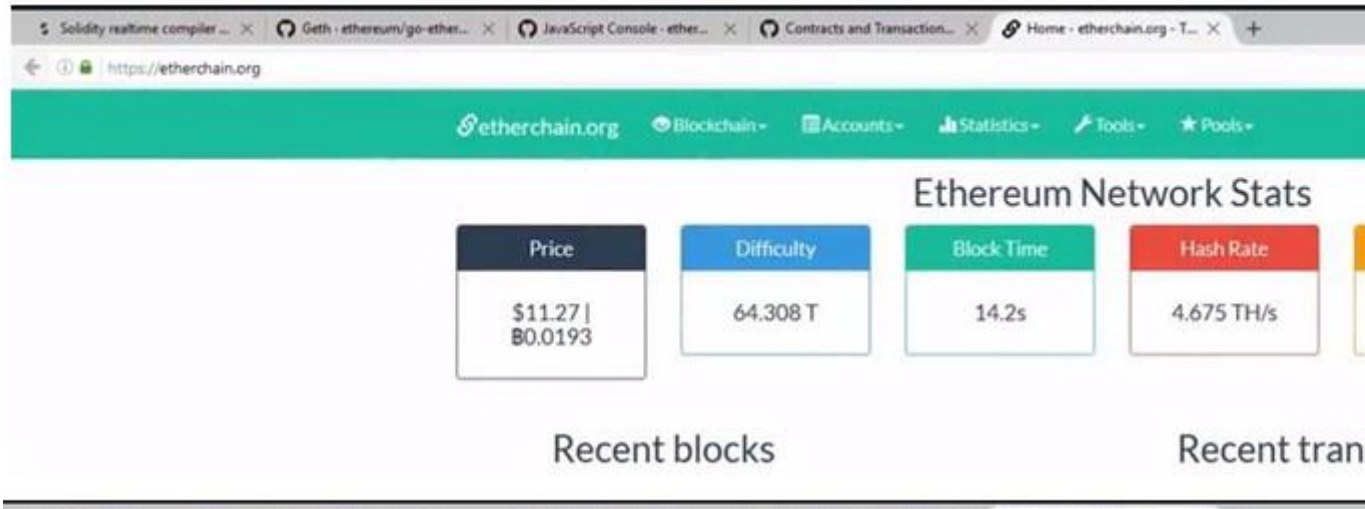
Величины балансов и размеры средств для перевода выражаются в веях. Чтобы выполнить конверсию веев в другую валюту, или эфир в веи, можно использовать команду web3.toWei, задав в ней количество валюты и ее название, например, web3.toWei(1, "ether");. Теперь посмотрим на второй счет с помощью команды eth.getBalance(eth.accounts[1]);. На нем достаточно средств для перевода эфира на только что созданный счет. Для осуществления перевода счет надо сначала разблокировать. Для этого используется команда personal.unlockAccount(eth.accounts[1]); . Geth запросит пароль. После обработки и подтверждения введенный пароль, счет разблокирован. Теперь проверим баланс последнего счета в системе с помощью команды eth.getBalance(eth.accounts[eth.accounts.length -1]);. Видно, что последний счет имеет на своем балансе ноль. Для перевода валюты используем функцию SendTransaction с соответствующей командой eth.sendTransaction. Эта функция требует использования объекта JavaScript в качестве аргумента, который состоит из трех параметров: From - источник переводимых средств, To - то, куда средства будут переведены, и Value - объем переводимых средств, выраженный в веях. Полностью команда выглядит как eth.sendTransaction({from:eth.accounts[1], to: eth.accounts[eth.accounts.length-1], value: web3.toWei(1, "ether")});. Закроем все нужные скобки и посмотрим, что получится. Функция SendTransaction вернула нам идентификатор транзакции.

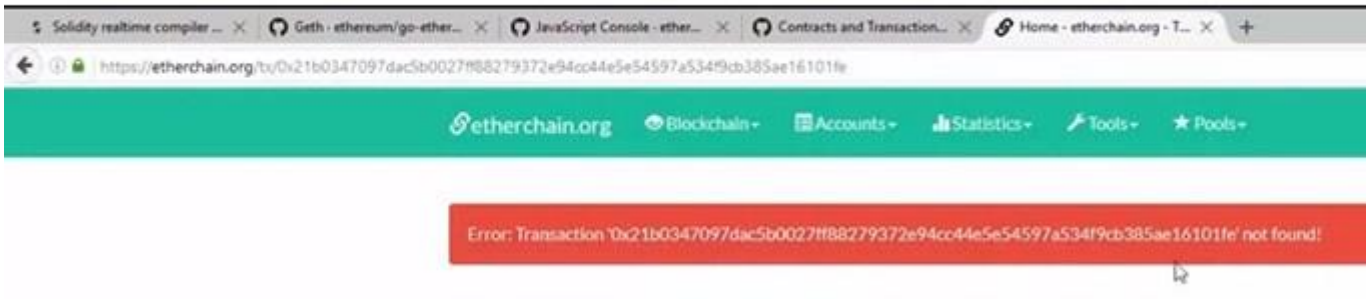


Обратим внимание на то, что результат выполнения транзакции - успех или неудача при ее выполнении - на данном этапе недоступен, это станет понятно позже.
Для просмотра списка транзакций, ожидающих выполнения, используется команда eth.pendingTransactions. Транзакция по переводу эфира между счетами все еще находится в этом списке.

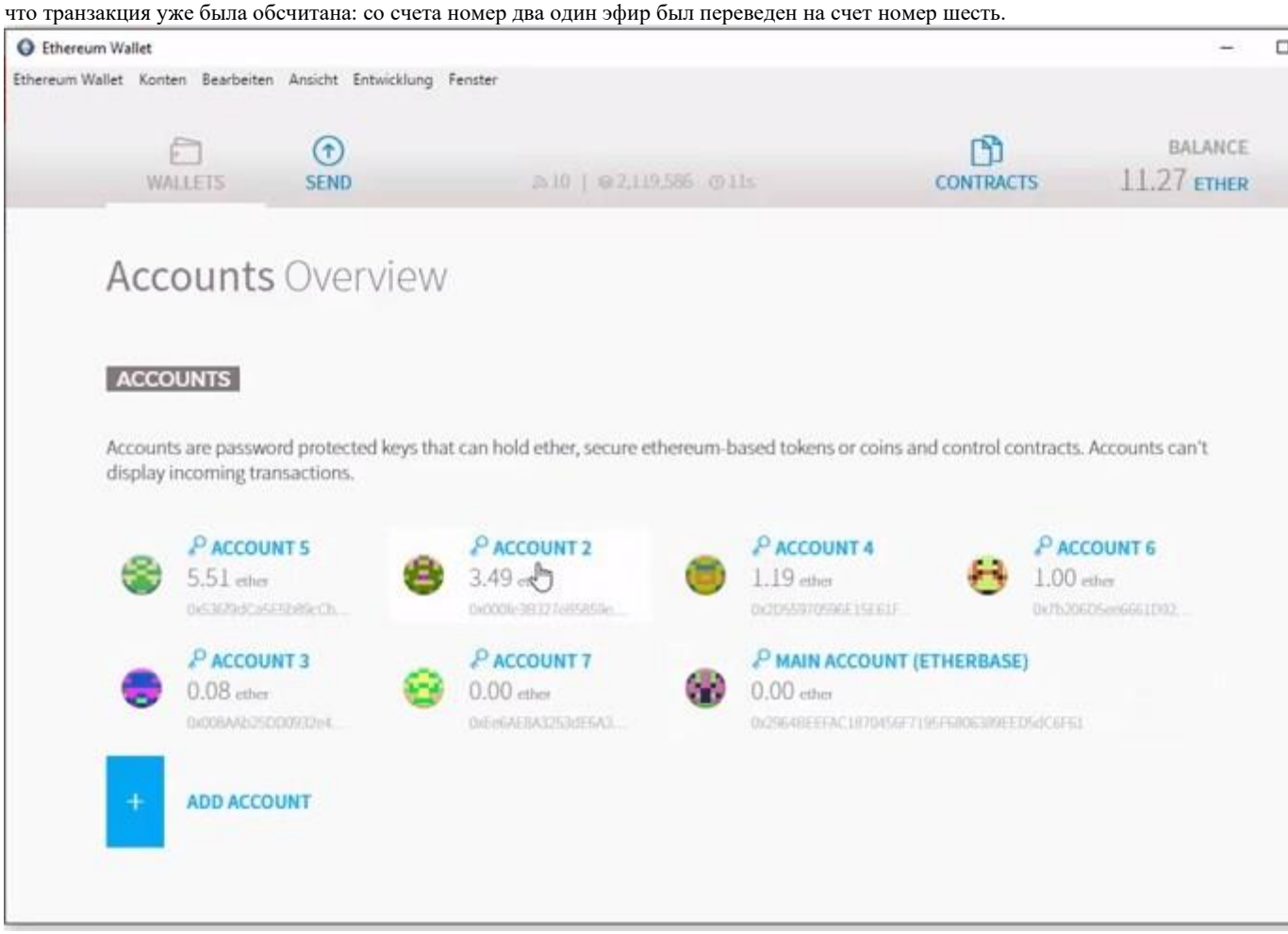


Чтобы посмотреть, как идет выполнение транзакции, можно скопировать идентификатор транзакции и найти его в веб-приложениях http://EtherScan.io или http://EtherChain.org. Транзакция пока еще не найдена, так как еще никто не согласился ее обсчитать, или замайнить.

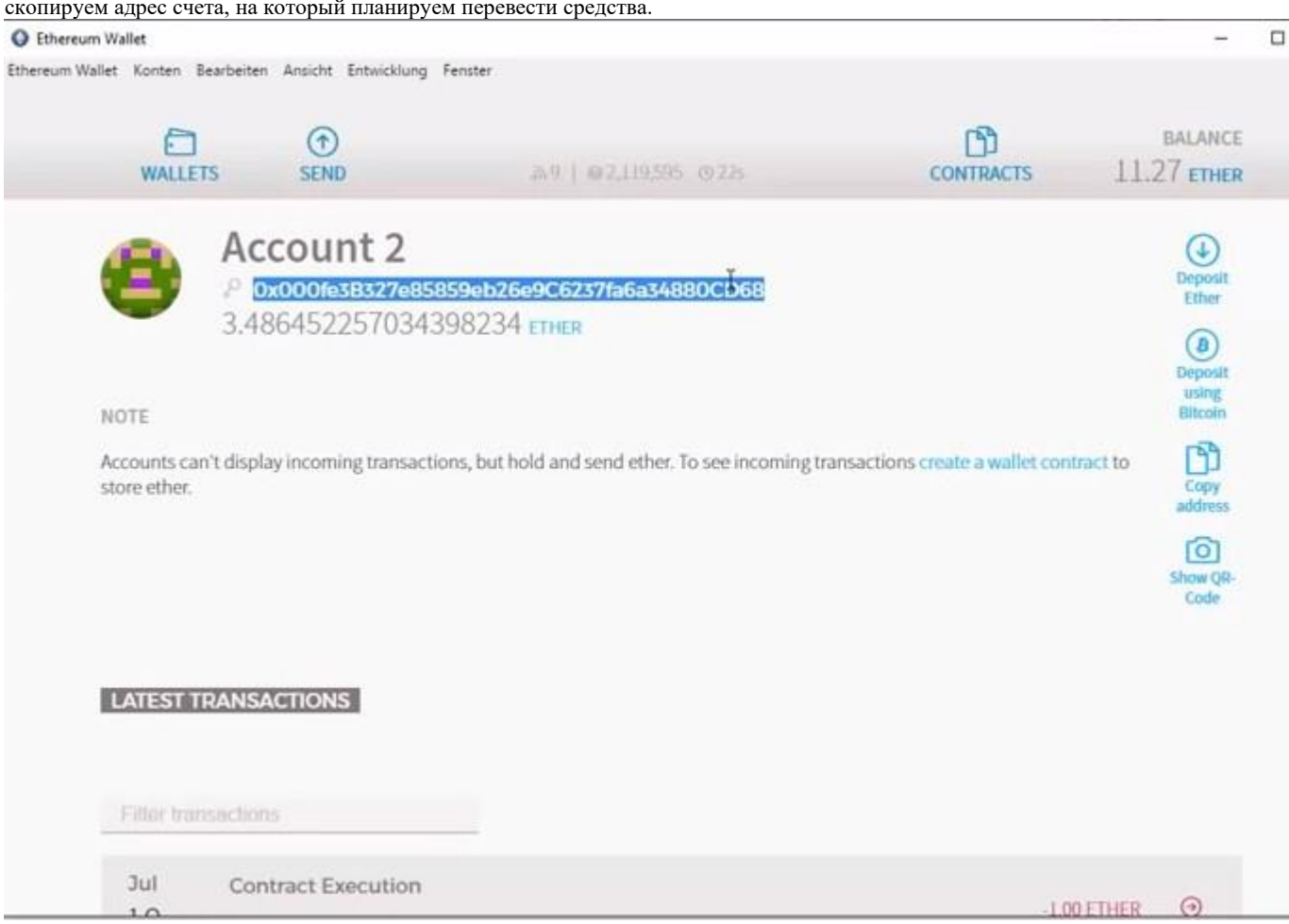




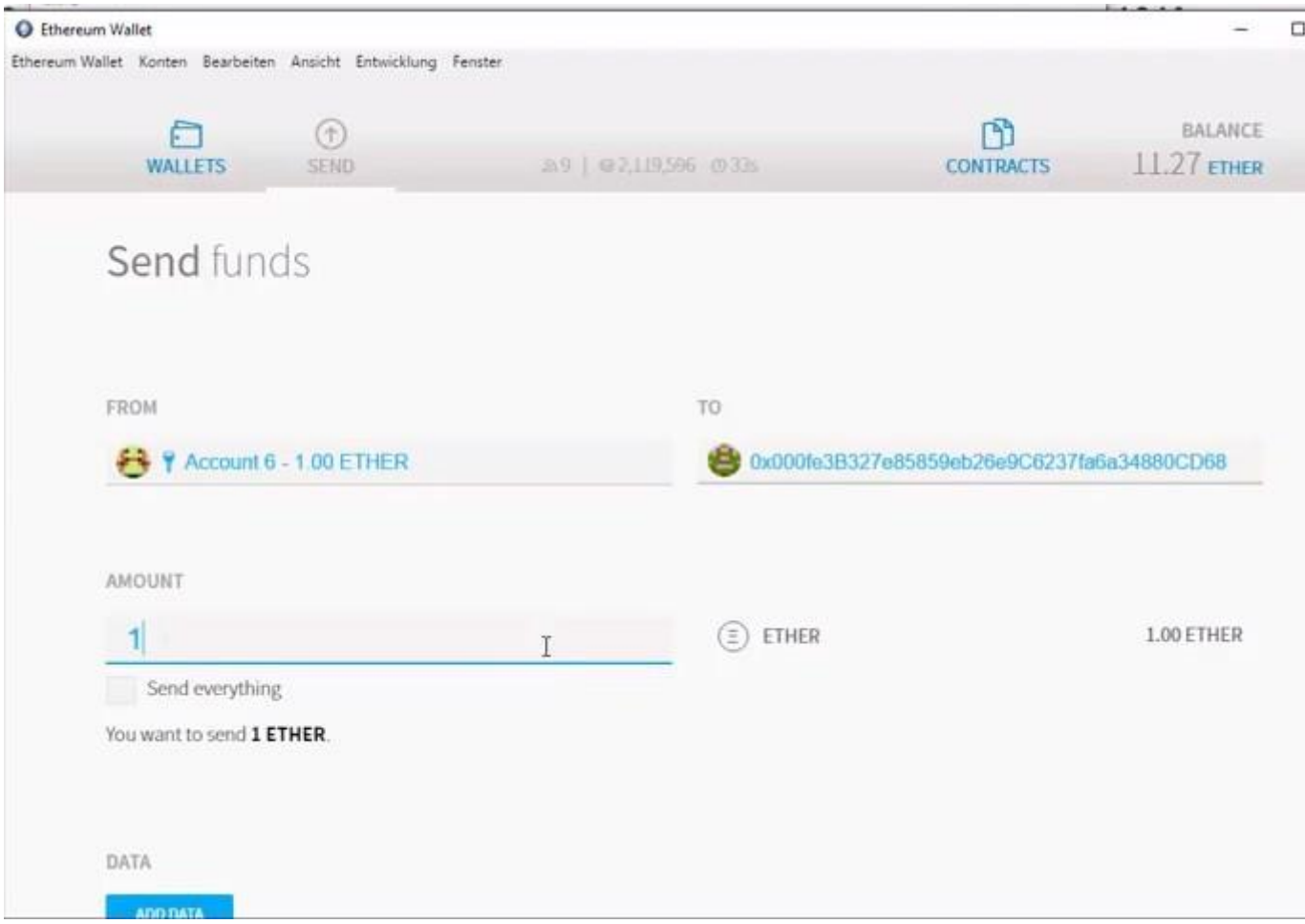
Откроем пока кошелек Mist. Он автоматически подключится к уже работающему процессу GoEthereum. Mist показывает, что транзакция уже была обсчитана: со счета номер два один эфир был переведен на счет номер шесть.



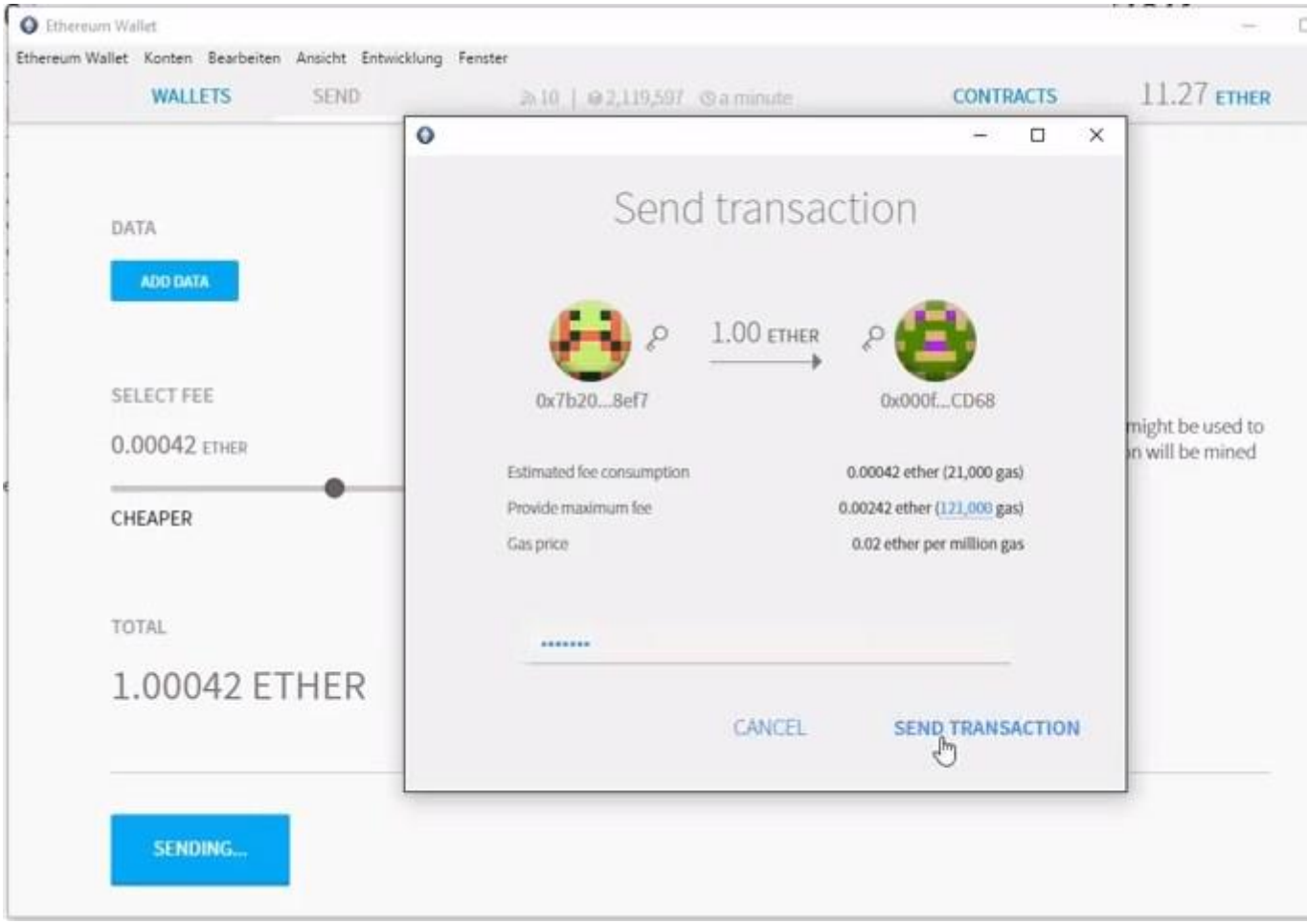
Теперь переведем средства обратно. Видно, что список транзакций, ожидающих обработки, снова пуст. Для осуществления обратного перевода будем использовать кошелек Mist, чтобы показать, что, в принципе, не так важно, каким интерфейсом или программой вы пользуетесь для работы со своими счетами в среде Ethereum. Для начала скопируем адрес счета, на который планируем перевести средства.



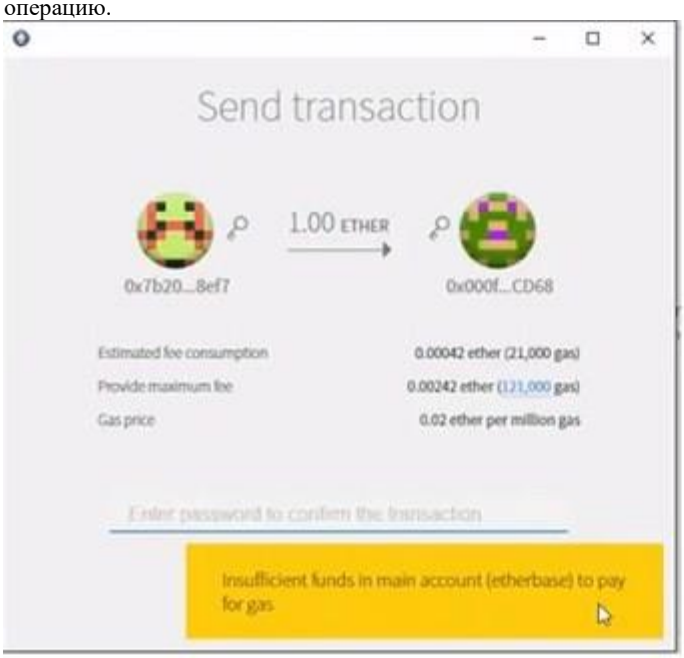
Затем щелкнем по кнопке Send и выберем счет номер шесть, на который только что переводили эфир. Теперь вставим предварительно скопированный номер счета, и укажем объем в один эфир.

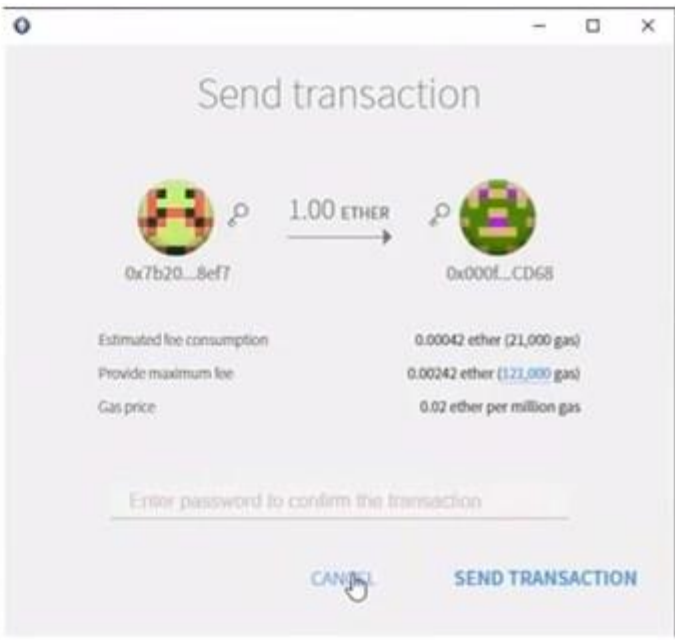


Еще раз обратим внимание на то, что для всех операций существует некоторая комиссия в виде эфира, выплачиваемая тем, кто обсчитывает транзакцию, или майнерам. Снова вводим пароль, созданный при создании счета, и отправляем транзакцию на обработку.

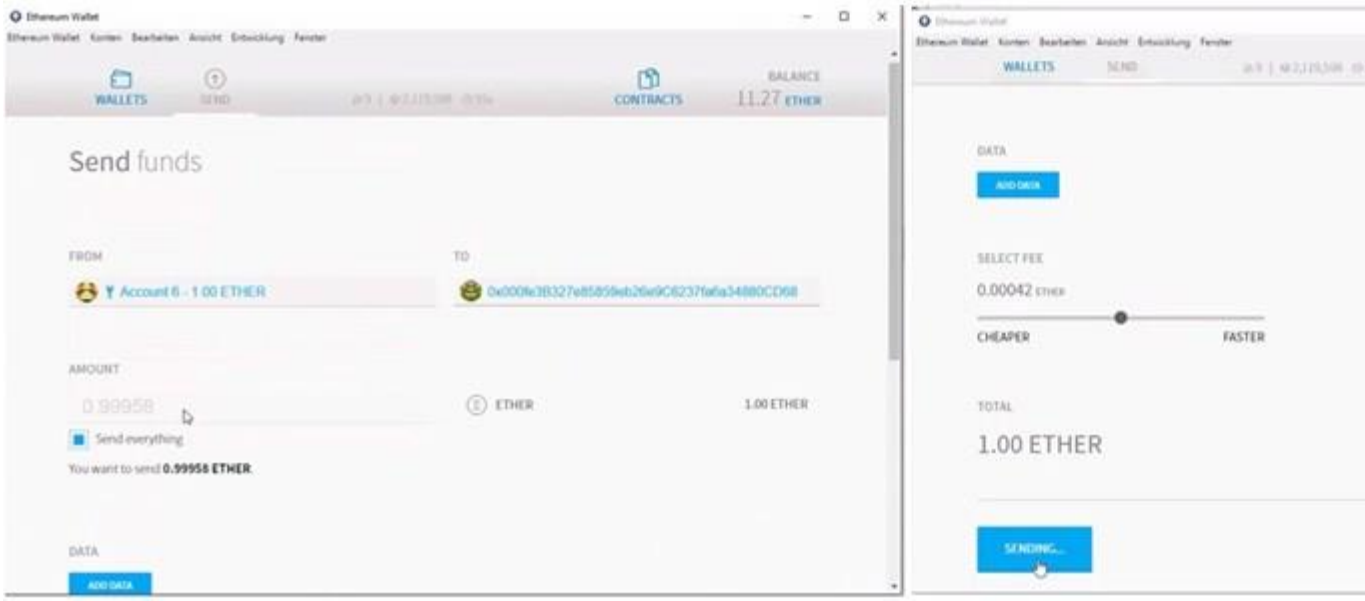


Видно, что данная транзакция не прошла по причине недостатка средств для оплаты комиссии майнеров. Отменим эту операцию.

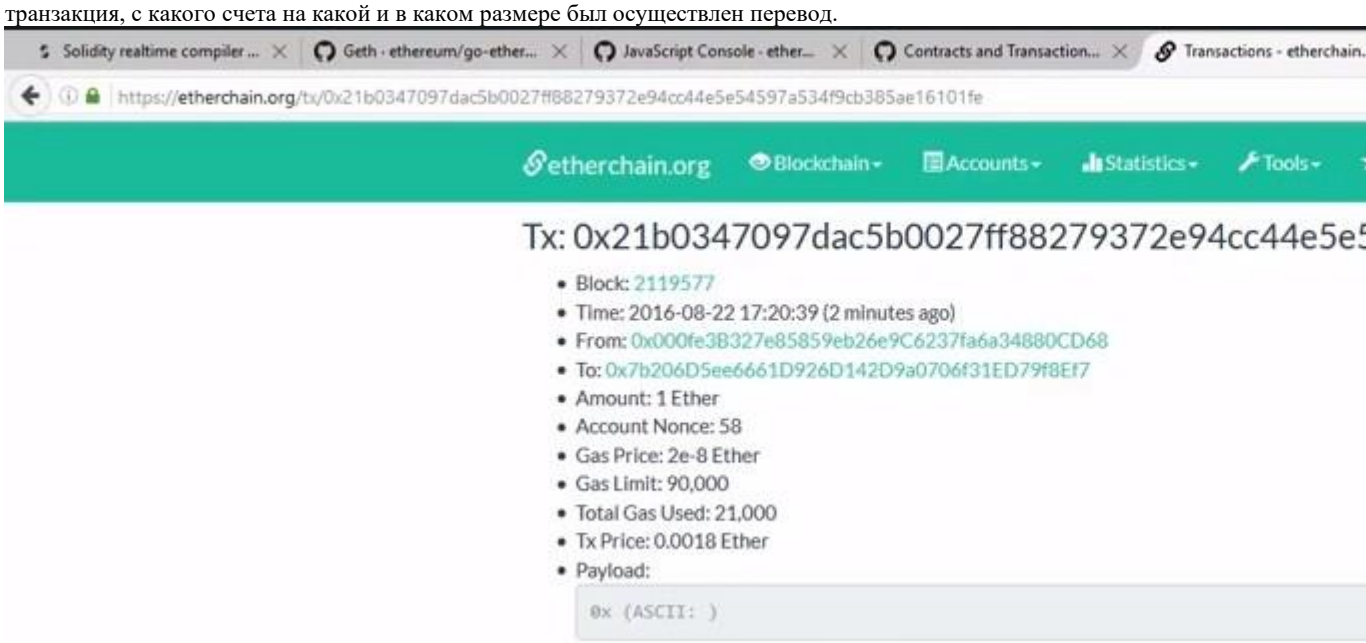




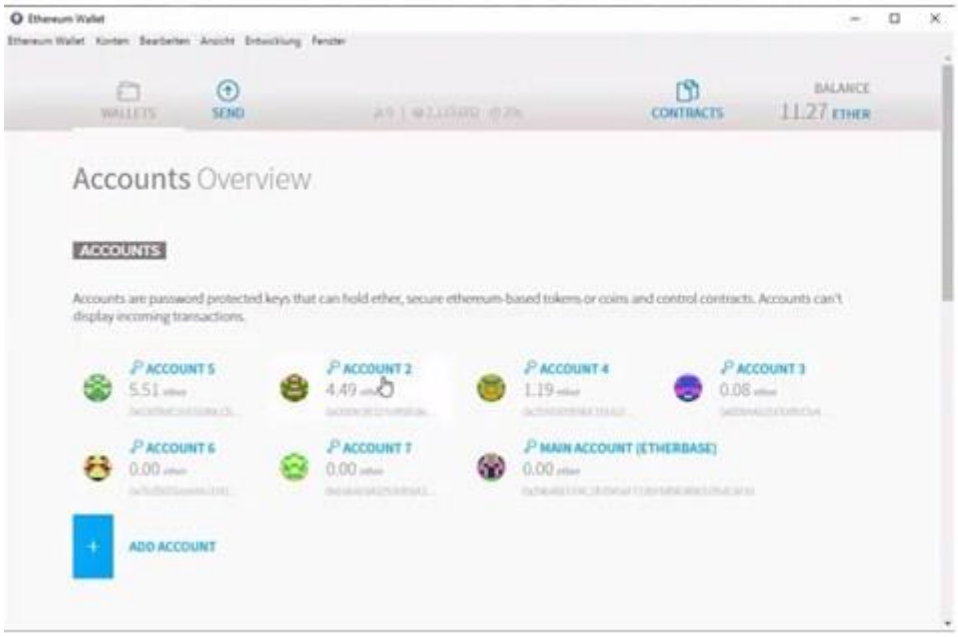
Теперь вместо выбора конкретного объема валюты для перевода выберем опцию перевода всех средств. Все необходимые расчеты Mist сделает сам. Снова вводим пароль и ждем.

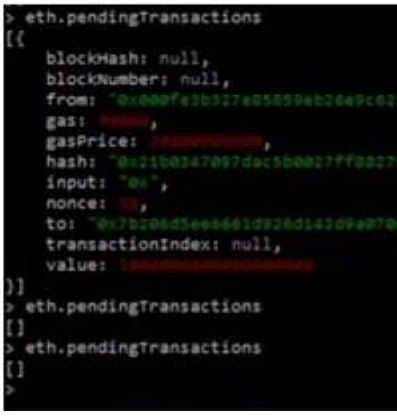


Тем временем давайте посмотрим на данную транзакцию на сайте EtherChain.org. Здесь видно, в какой блок попала наша транзакция, с какого счета на какой и в каком размере был осуществлен перевод.



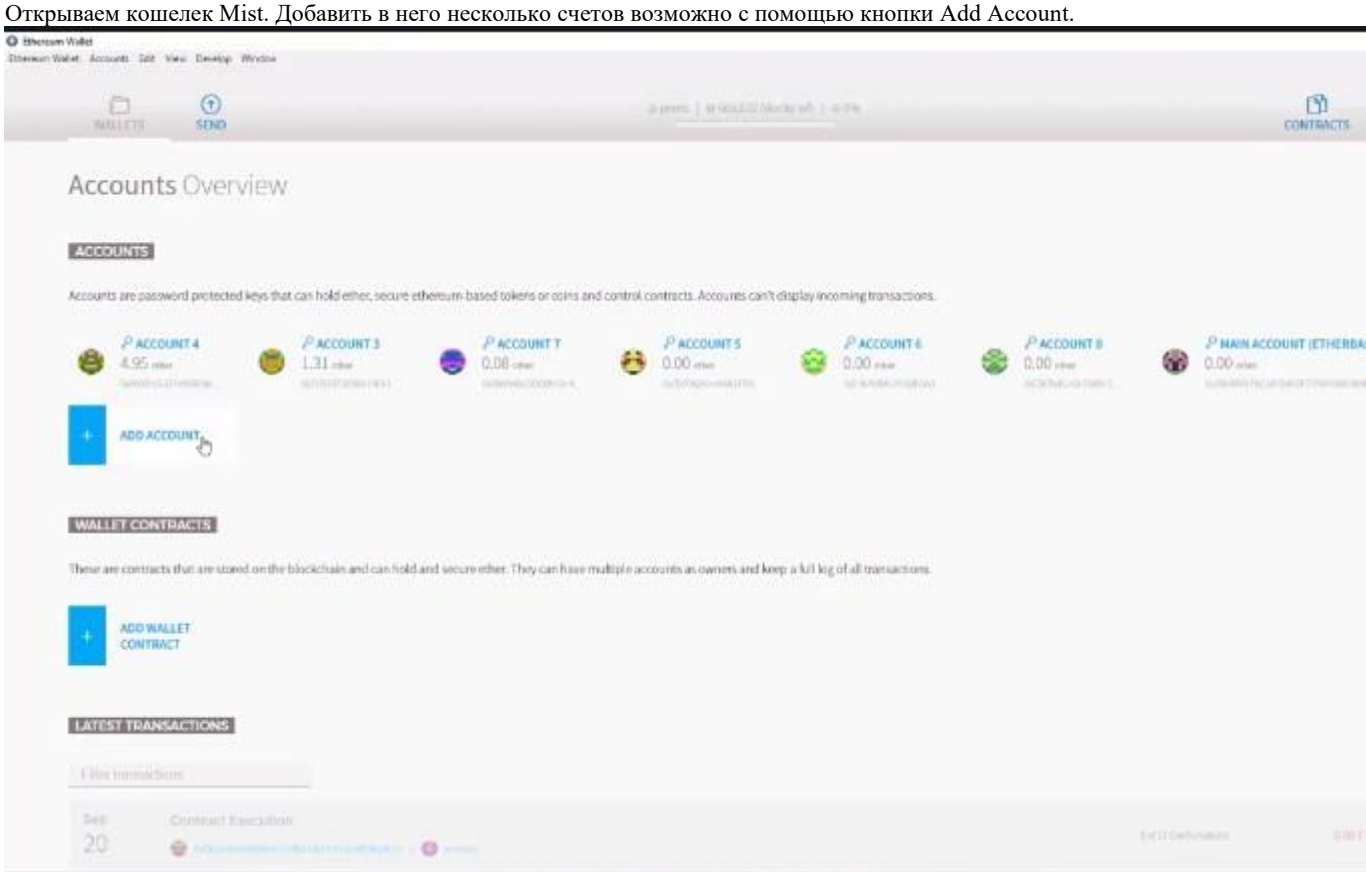
Видно, что счет номер шесть снова пуст, а счет номер два пополнился переведенными средствами, и, кроме того, не осталось незавершенных транзакций.



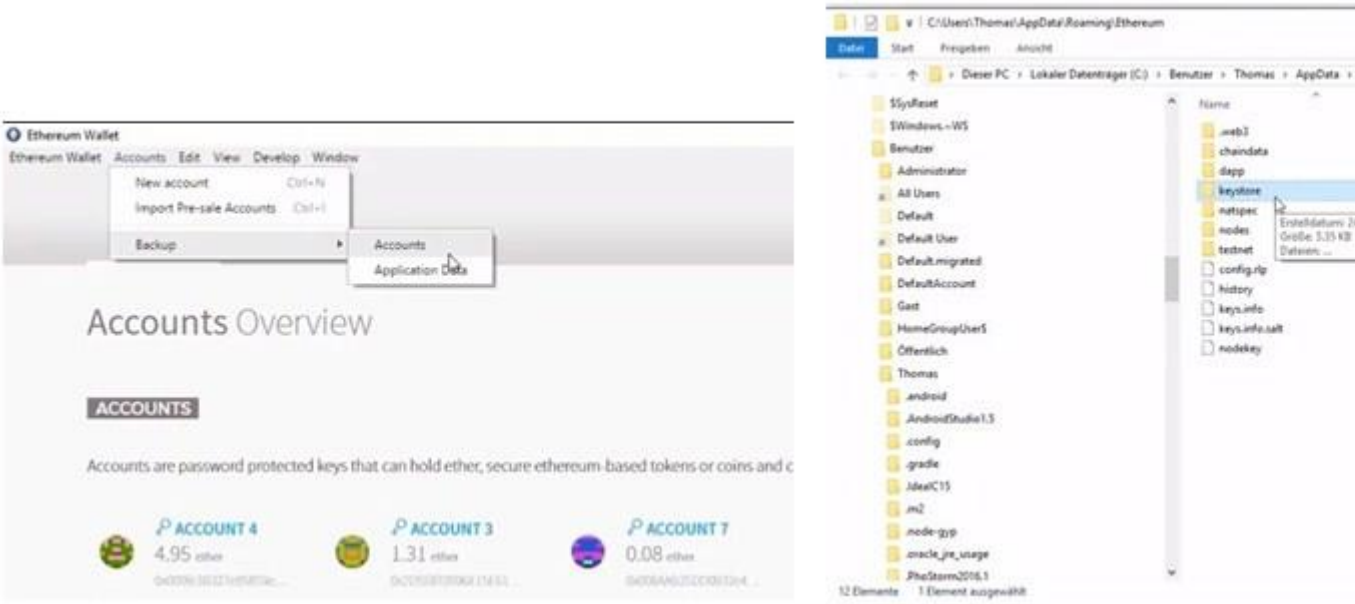


Управление счетами и добыча эфира (майнинг)

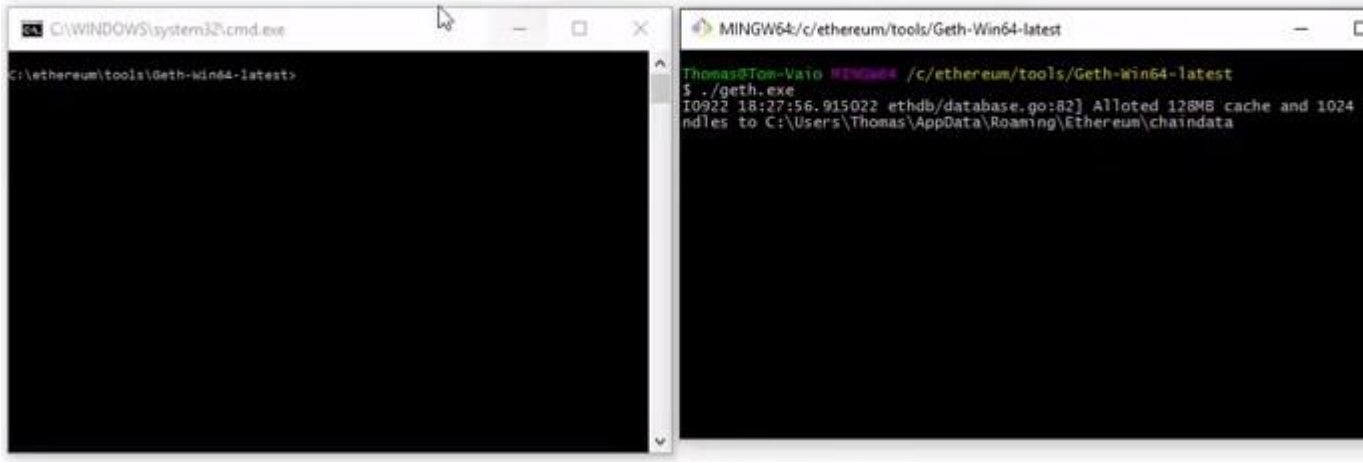
Открываем кошелек Mist. Добавить в него несколько счетов возможно с помощью кнопки Add Account.



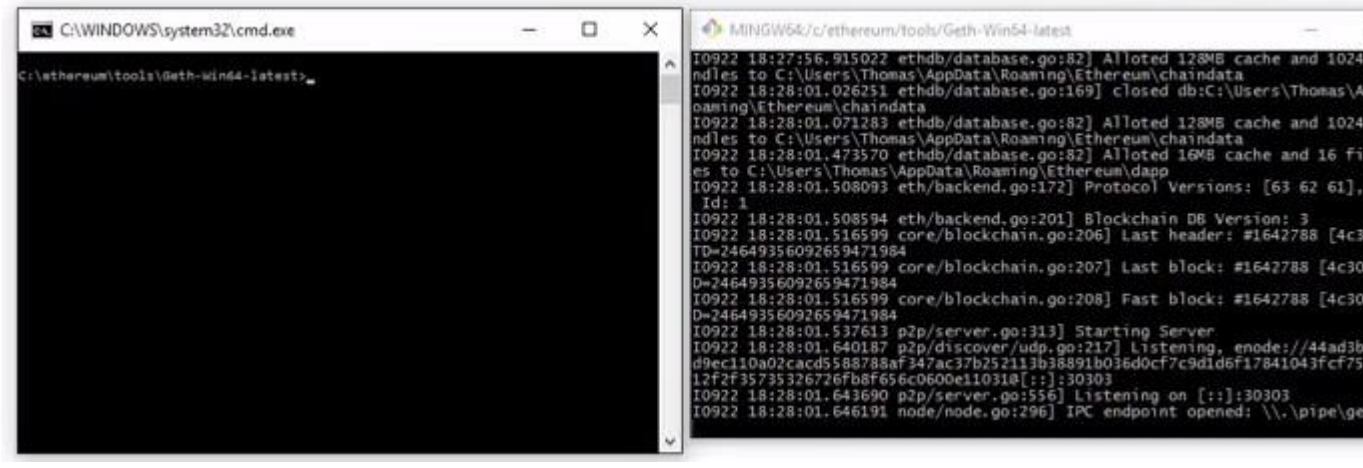
Как сделать резервную копию счетов? В зависимости от того, с какой операционной системой вы работаете, Mist и другие клиенты для платформы Ethereum будут сохранять файлы со счетами в различных каталогах. В меню Mist нужно последовательно щелкнуть по пунктам Accounts, затем Backup и затем Accounts еще раз. Откроется каталог Keystore, в котором хранятся ключи к счетам Ethereum.



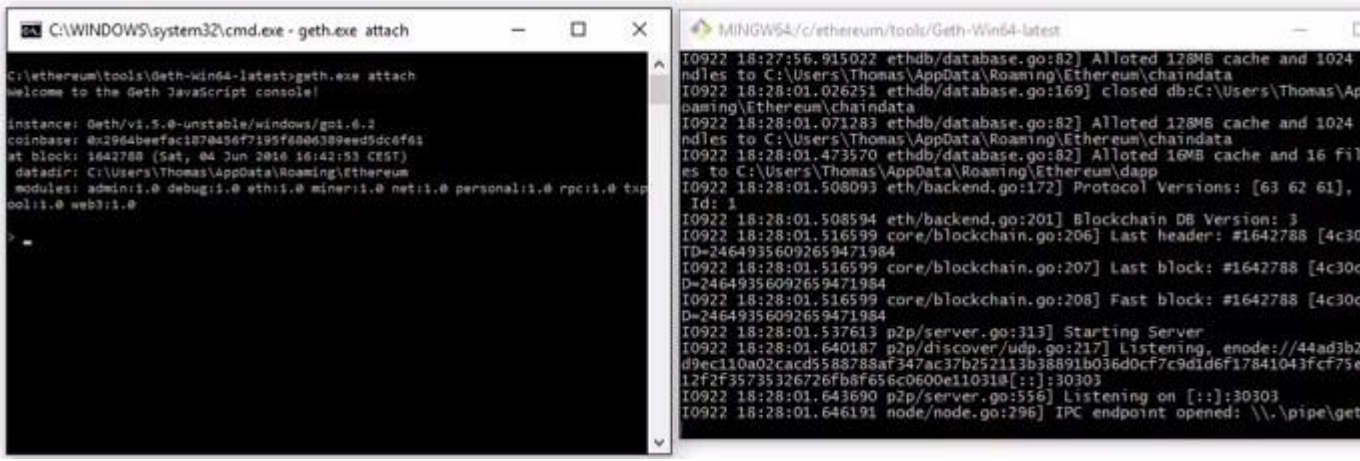
Эти ключи можно скопировать на USB-носитель или отправить в облачное хранилище. Крайне важно убедиться, что они хранятся в надежном месте. Их потеря по любой причине приводит к абсолютно безвозвратной недоступности счетов - без этих ключей ни вы, ни кто-либо другой не сможет получить доступ к счетам.
Загружаем Geth, с помощью которого продемонстрируем, как можно начать майнинг эфира. В двух разных окнах открываем две консоли для работы с Geth. В правом окне - командная строка для Geth.



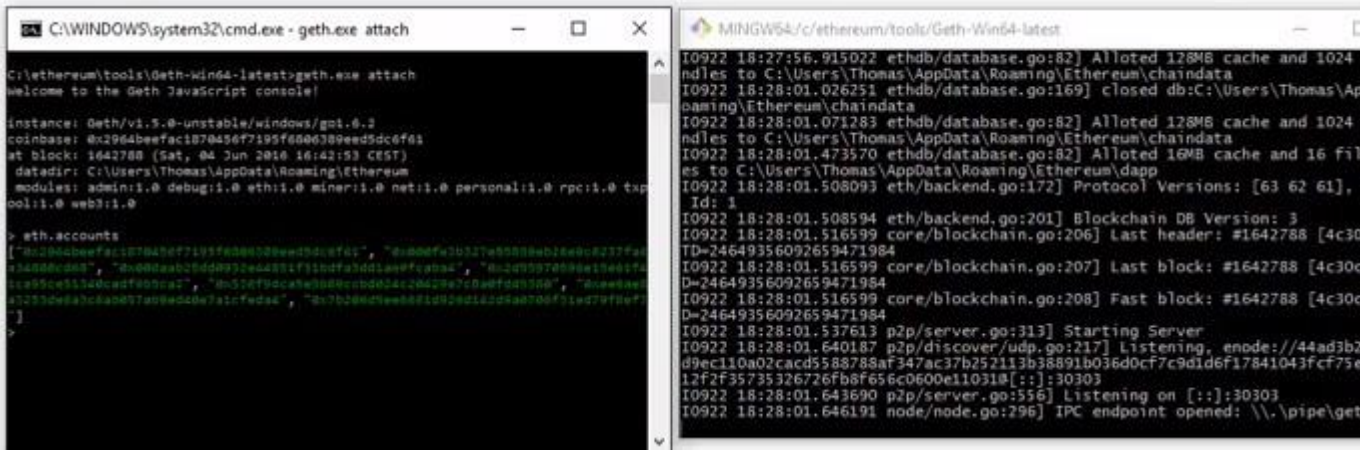
Если вы работаете с Linux, можно использовать привычную вам оболочку. При работе с Windows подходит стандартная командная строка. Веб-кошелек закрываем и запускаем сервис Geth.



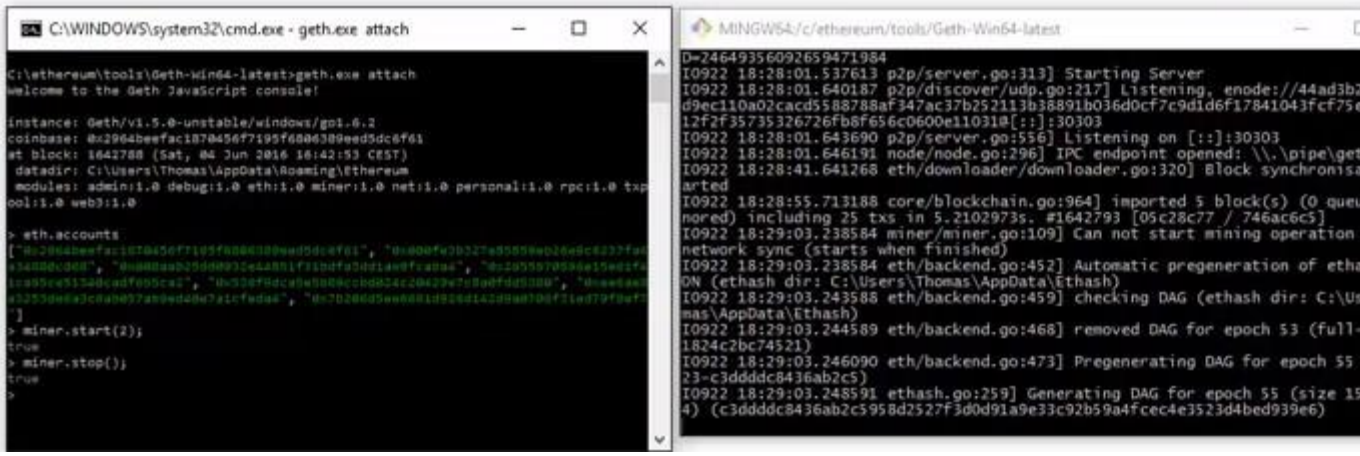
Во втором окне подключаемся к запущенному процессу Geth.



Geth открывает доступ ко всем счетам и операциям с ними так же, как и Mist, но посредством командной строки.



Все счета хранятся в едином центральном каталоге, и все клиенты в среде Ethereum - будь то Mist, Geth или CPP-Ethereum - подключаются к нему. На текущий момент Geth поддерживает майнинг только с помощью центрального процессора, что, возможно, и не очень быстро, но вполне достаточно для работы в тестовой среде или частной сети. Чтобы запустить процесс майнинга из консоли, достаточно задать команду miner.start. В качестве параметра можно задать число потоков, в которых будет осуществляться обсчет операций в блокчейне, например, два. В любой момент майнинг можно остановить.

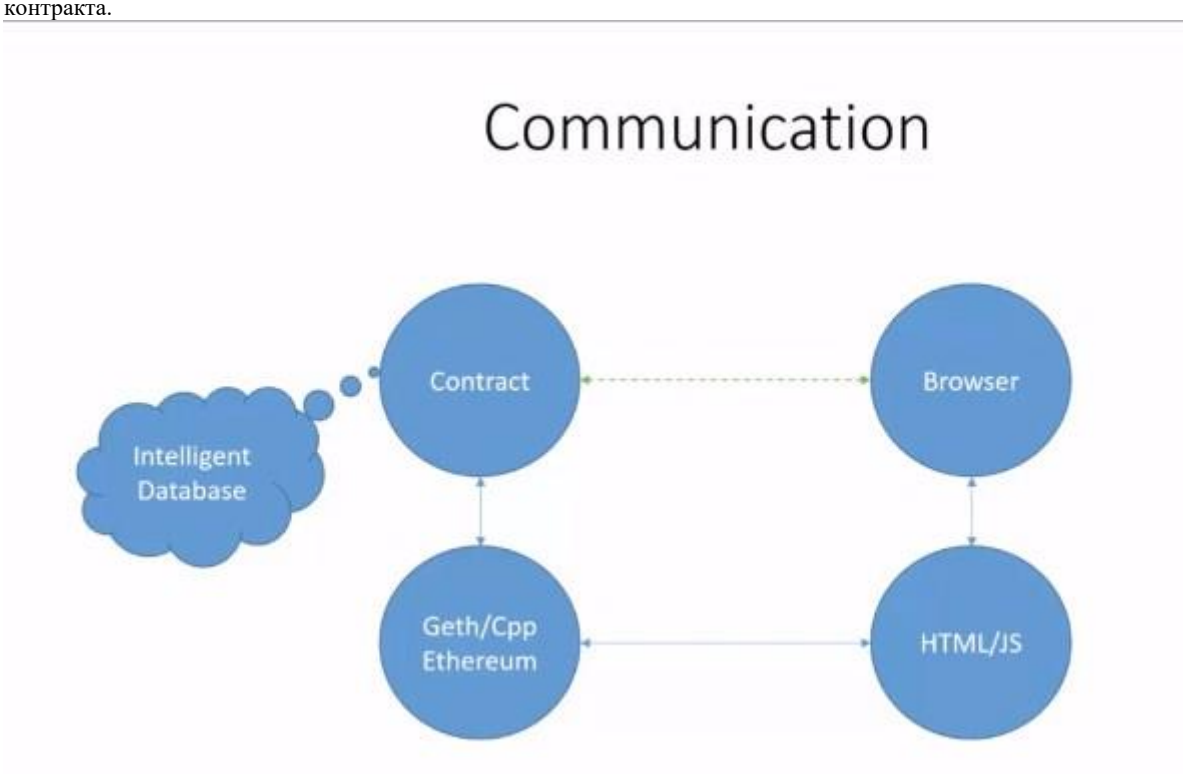


Отличное вводное руководство в майнинг можно найти на wiki-пространстве Github Ethereum по ссылке http://github.com/ethereum/wiki/wiki/miningintroduction. Помимо общей информации, там приведены разъяснения используемых алгоритмов майнинга и даны расшифровки терминологии. Чтобы узнать больше о майнинге с помощью графических процессоров, можно посетить репозиторий GoEthereum в разделе wiki/Mining. В нижней части страницы можно найти целый раздел на эту тему, где перечислены инструменты, настройки и инструкции. В рамках этого курса сконцентрируемся на написании контрактов и размещении их в блокчейне.

Фазы разработки контрактов

В этой части разберем фазы процесса разработки распределенных приложений. Первая фаза - написание кода. Это можно делать в привычной среде разработки, например, в Ethereum Studio или Mix, или даже в блокноте. После разработки

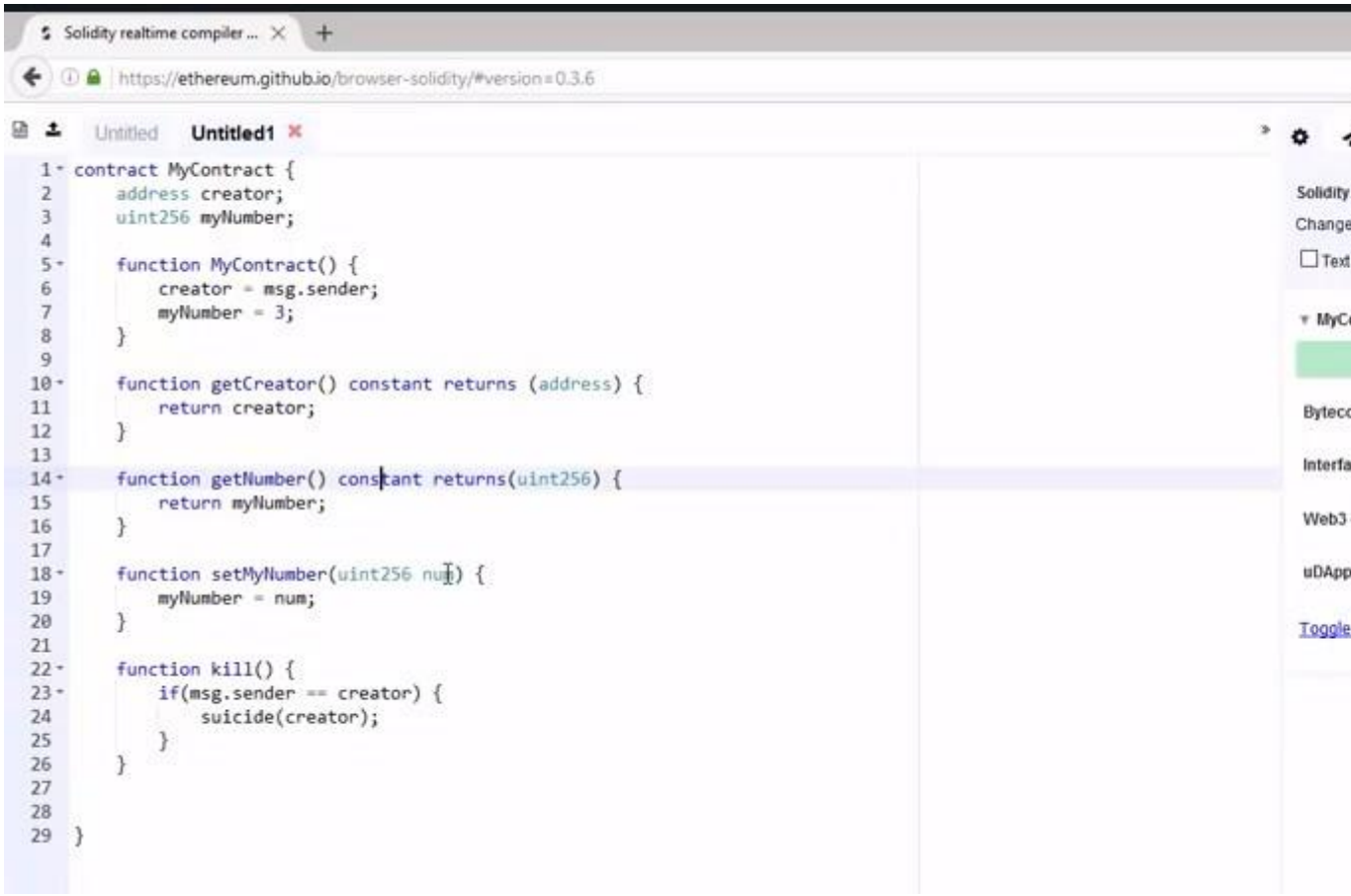
целесообразно провести ряд тестов. Для этого хорошо подходит Mocha ("Мокка" Блочное тестирование удобно проводить в среде Truffle, да и в Ethereum Studio есть стандартный проект с уже встроенными тестами. Конечно, не обойтись и без ручных тестов, которые можно проводить, к примеру, в браузере Solidity, или в частном блокчейне. Затем проводятся окончательные проверки. Необходимо убедиться, что ваше распределенное приложение работает в соответствии со спецификацией. Это можно сделать в тестовой среде без расходования эфира.
После завершения этой фазы целесообразно разместить контракт в реальном блокчейне. Потребуется также подключить ваш код HTML или JavaScript к адресу контракта или, если их несколько, к адресам всех контрактов. Затем можно отправлять HTML-код на сервер.
Разберемся, как работает коммуникация. Необходимо, чтобы браузер мог взаимодействовать с контрактом. Однако напрямую они общаться не могут. Обычно браузер обрабатывает код HTML, который вызывает код JavaScript, который, в свою очередь, связывается с контрактом посредством узла сети Ethereum, например, запущенной службой Geth или CPP-Ethereum. Узел сети Ethereum обеспечивает доступ в блокчейн, и, таким образом, обеспечивает связь блокчейна и вашего контракта.



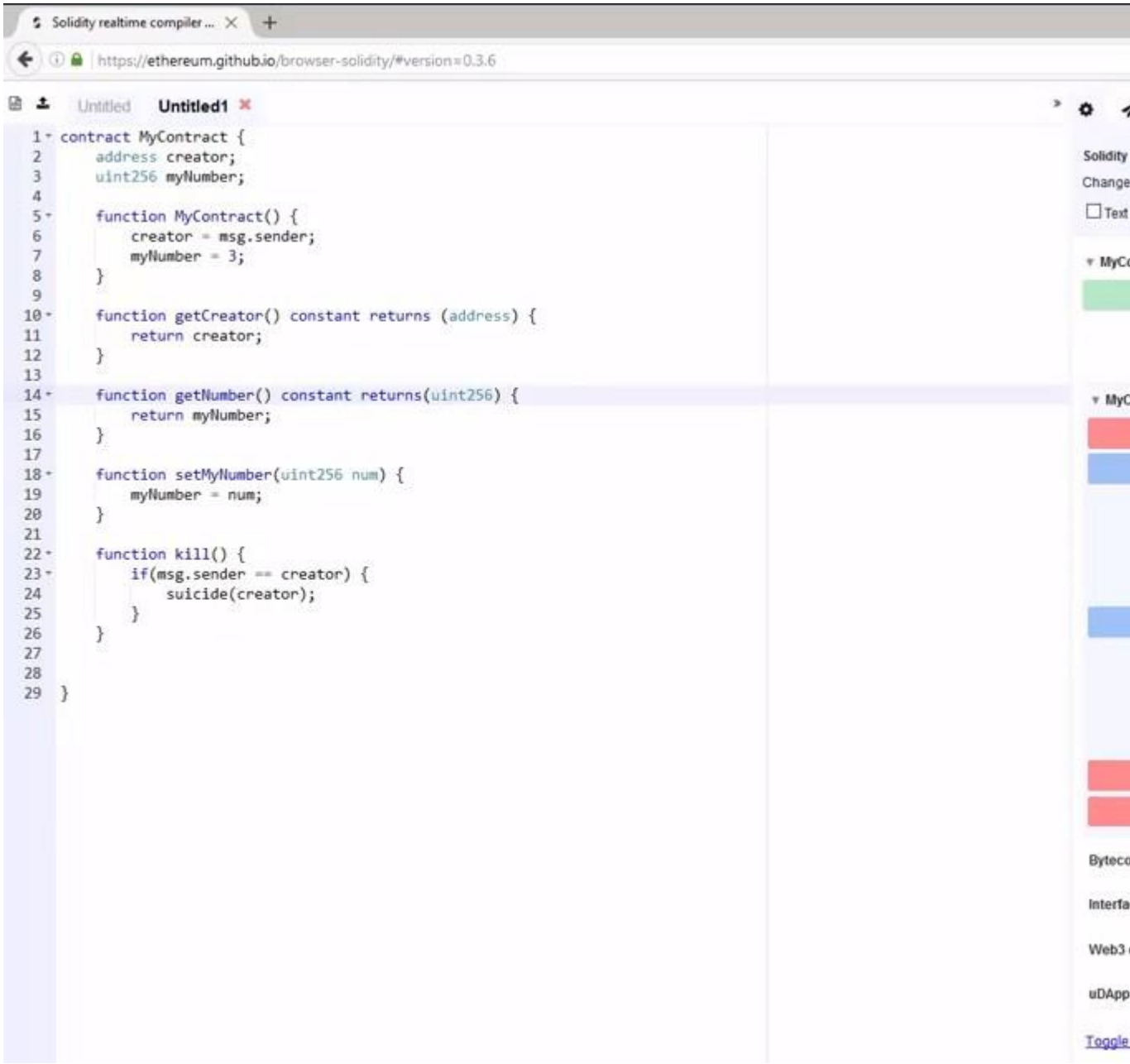
Контракт представляет собой некоторое подобие базы данных со встроенной логикой. Итак, для работы распределенного приложения потребуются браузер, традиционный сервер для обработки HTML-кода и кода JavaScript и узел Ethereum с интерфейсом HTTP RPC для доступа к блокчейну.

Работа с контрактом после его размещения в блокчейне

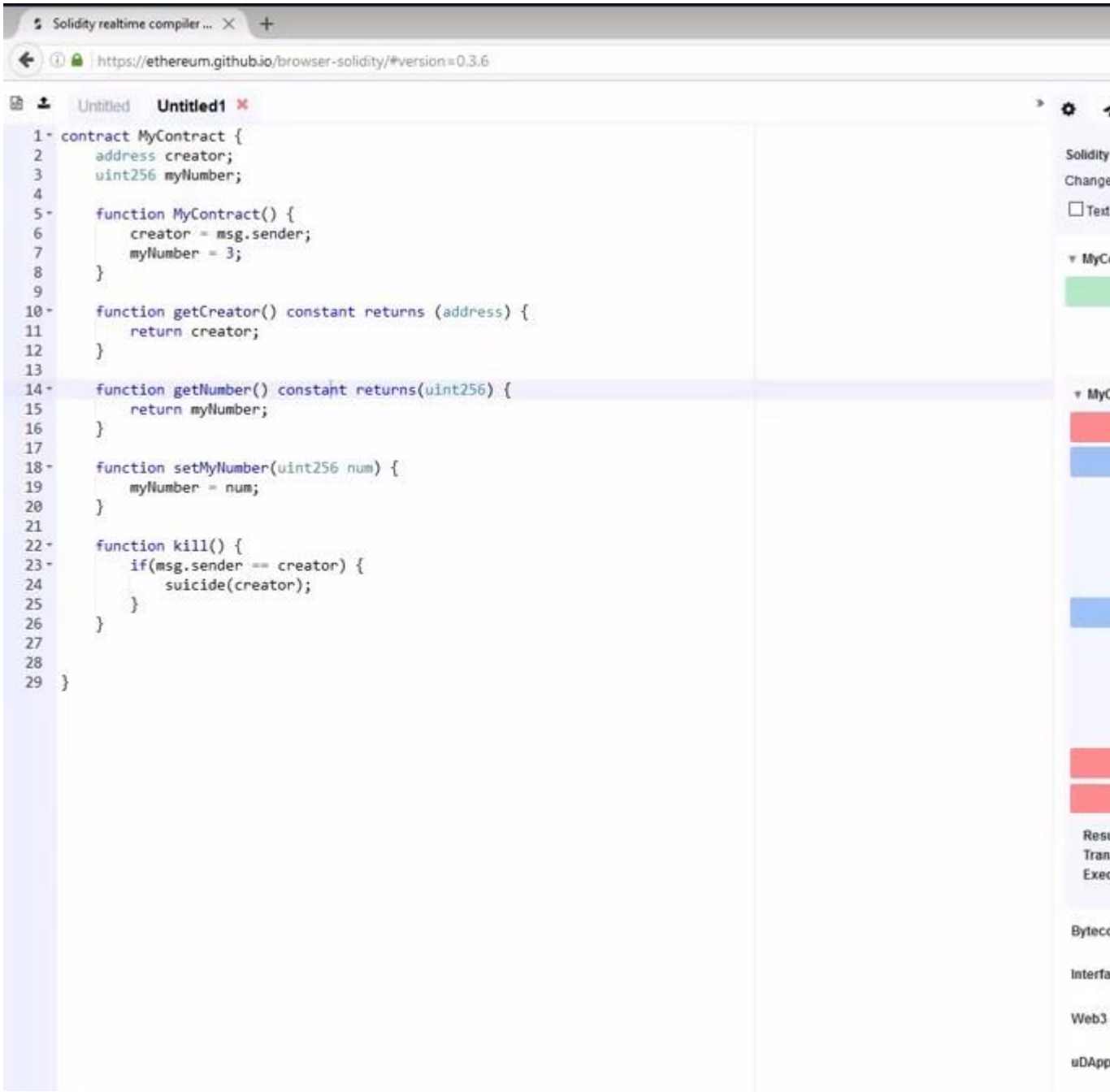
В этой части поговорим о том, как взаимодействовать с контрактом на базе браузера Solidity. В контракт были внесены изменения.



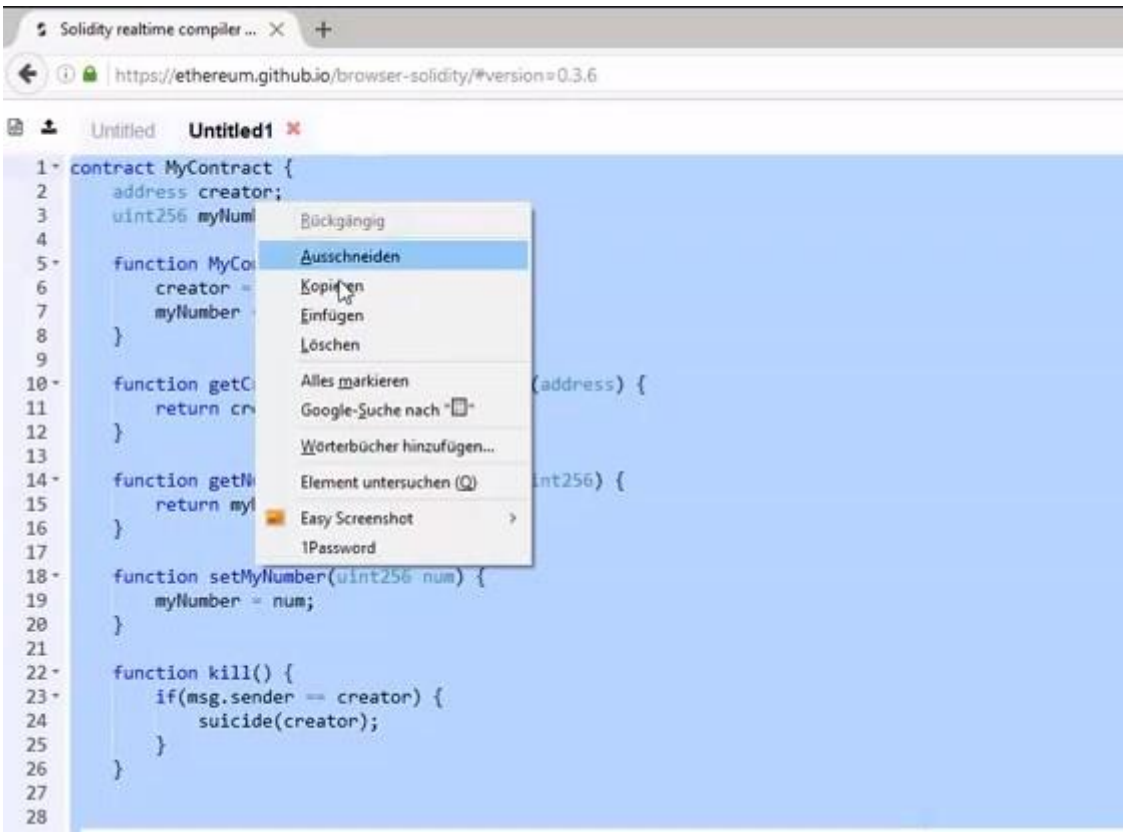
В контракт добавлена числовая переменная *myNumber*, и назначил ей в конструкторе значение "три". Поэтому при создании контракта число *myNumber* будет равно трем. Кроме этого, добавлены еще две функции. Первая - это *getNumber*, возвращающая значение *myNumber* и заданная как постоянная функция. Вторая - это *setNumber*, позволяющая задавать значение *myNumber* в диапазоне uint256, то есть до примерно 1,16 ? 10^{77}.
Щелчок по кнопке Create запускает эмуляцию контракта прямо в браузере Solidity. Сразу видно, что заданные функции, *getNumber* и *getCreator*, возвращают свои начальные значения, и на эти вычисления эфир не расходуется. Эти функции заданы как постоянные, а за загрузку постоянной информации не нужно ничего платить. Эфир нужно тратить только в тех случаях, когда вносят изменения в данные, хранящиеся в блокчейне. Вы запускаете нужный код с помощью транзакции и уплачиваете определенную сумму в эфире за обработку и реализацию транзакции.

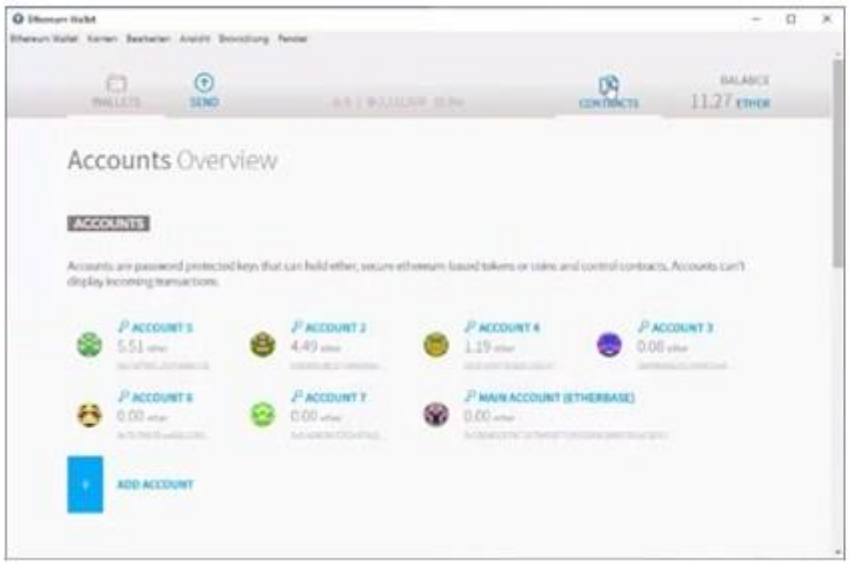


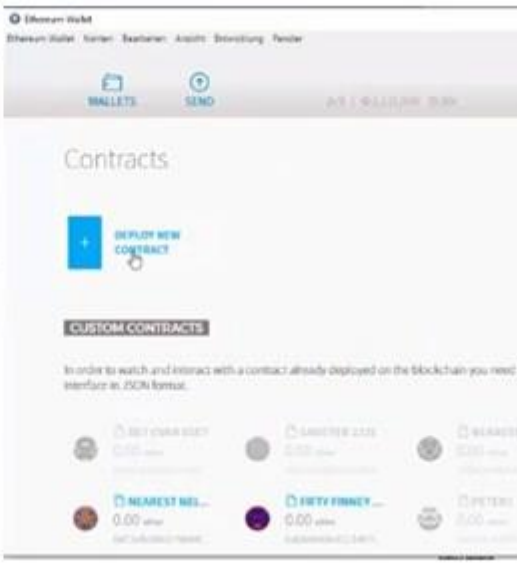
Посмотрим, что произойдет, если изменить число на пять. Видно, что при этом создалась транзакция на изменение данных в блокчейне, за что пришлось заплатить немного газа. А новое значение переменной - пять. Блокчейн работает как автомат состояний: когда вы меняете какую-либо величину, за это изменение нужно заплатить некоторое количество газа.



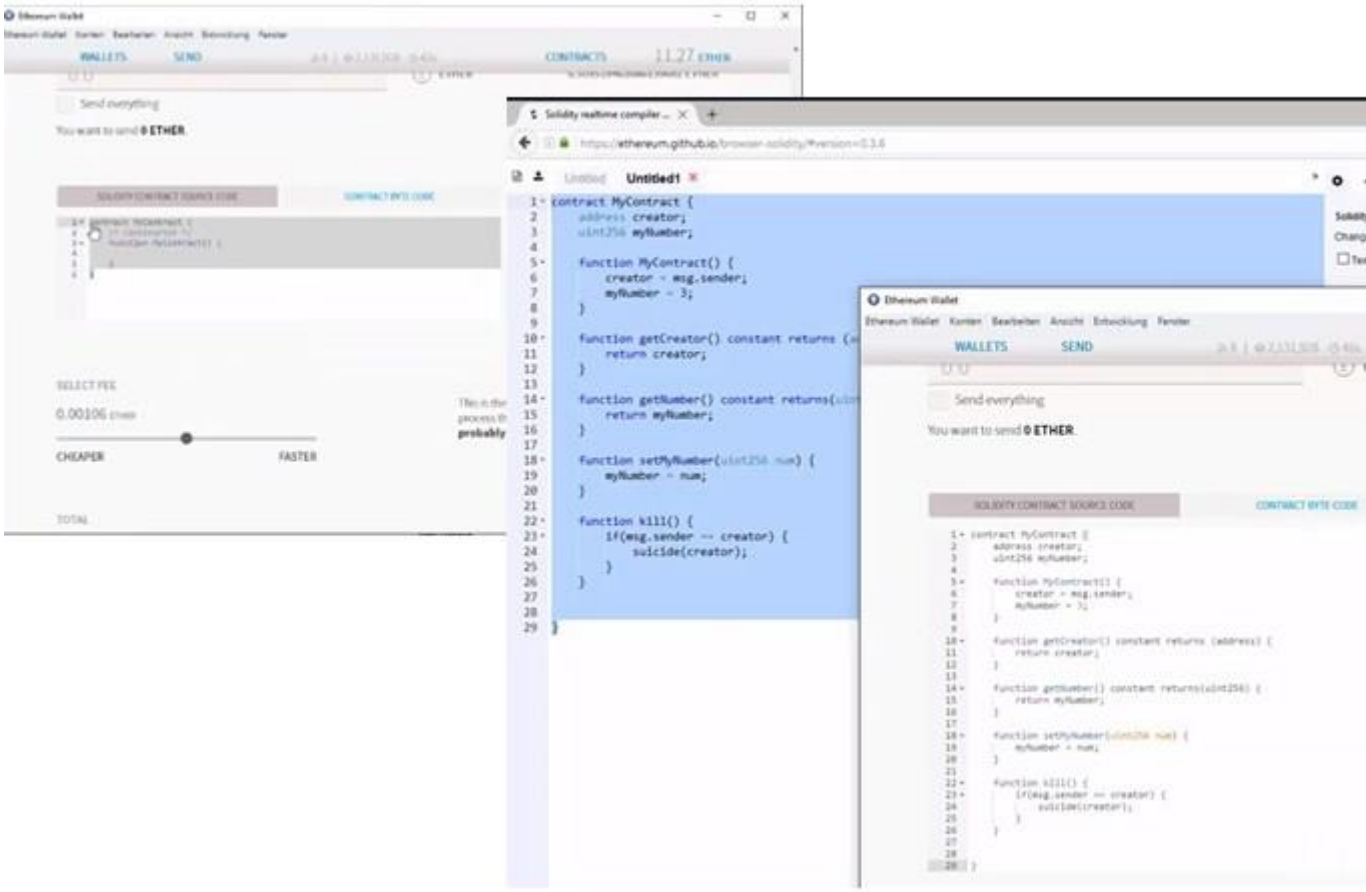
Давайте загрузим контракты в реальный блокчейн. Для этого нужно скопировать контракт, открыть кошелек, открыть раздел Contracts и щелкнуть по кнопке Deploy New Contract.



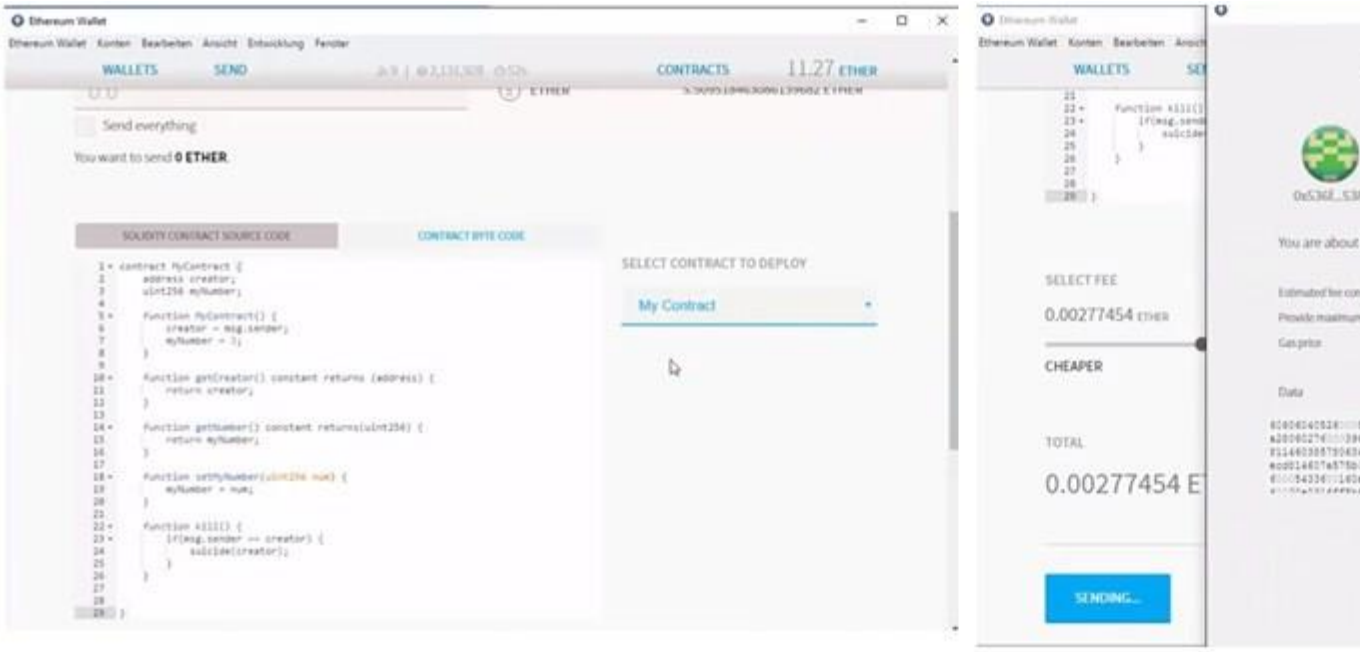




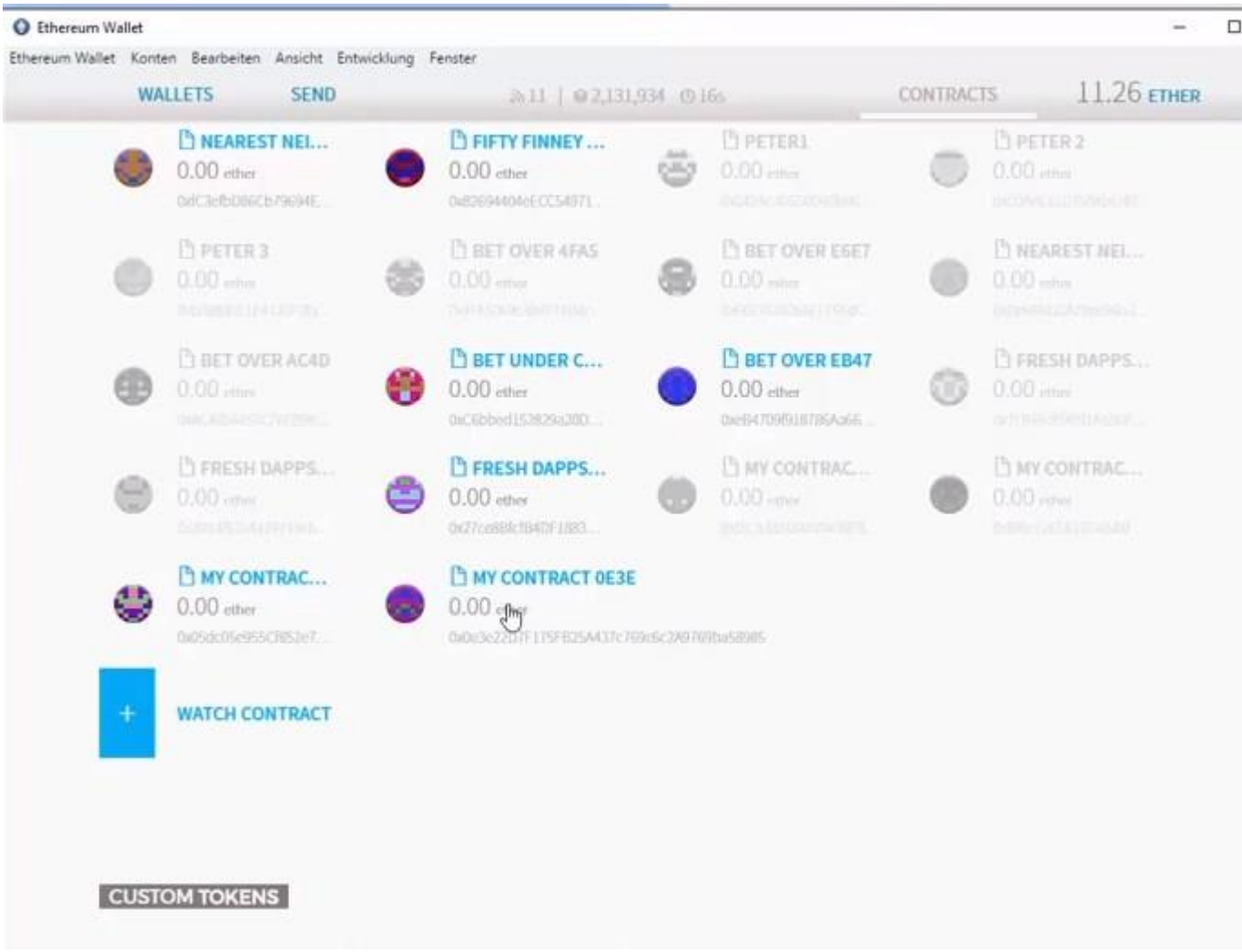
Затем надо выбрать счет, вставить исходный код.



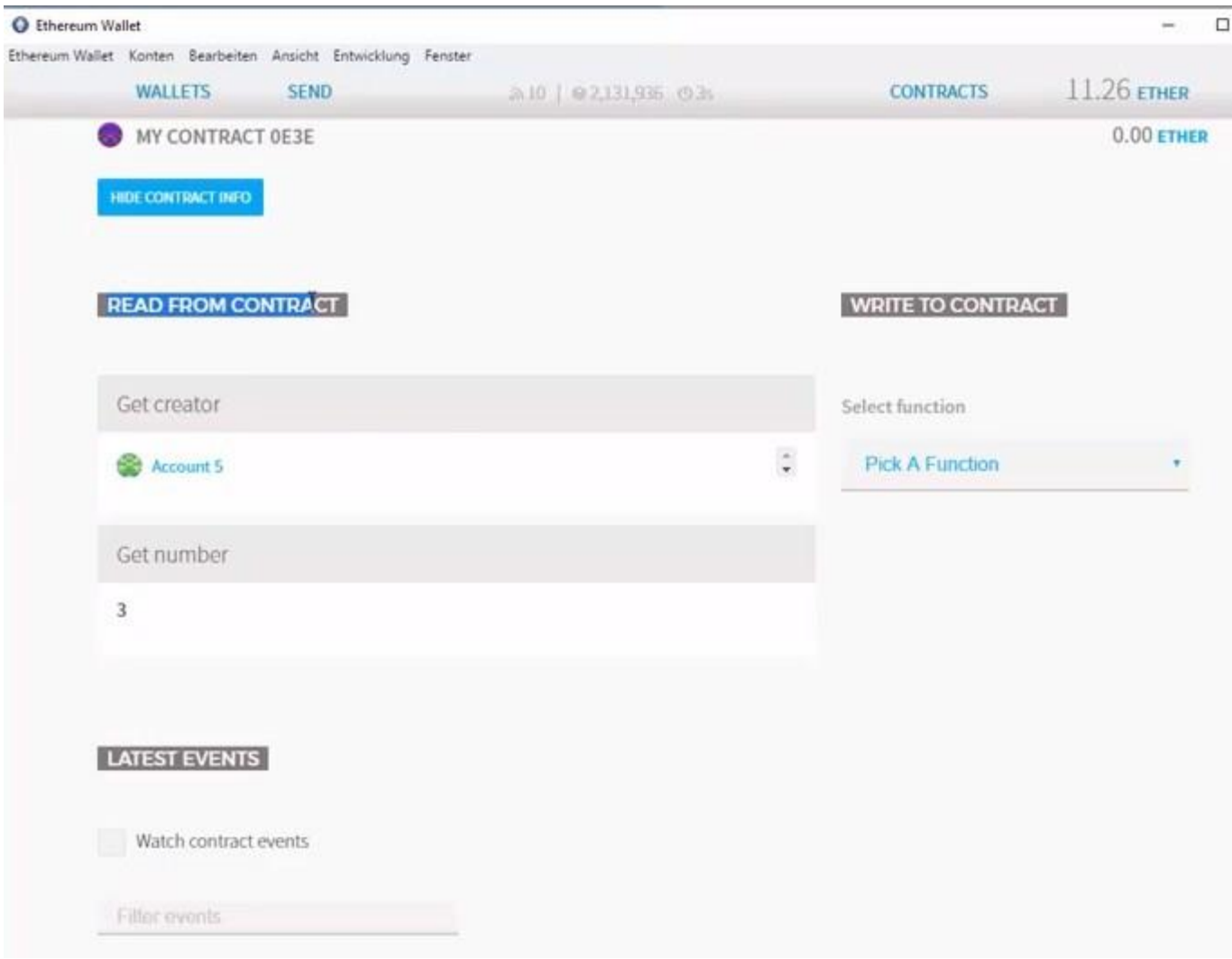
Выбрать имя MyContract и щелкнуть по кнопке Deploy. После ввода пароля необходимо дождаться размещения контракта в блокчейне.



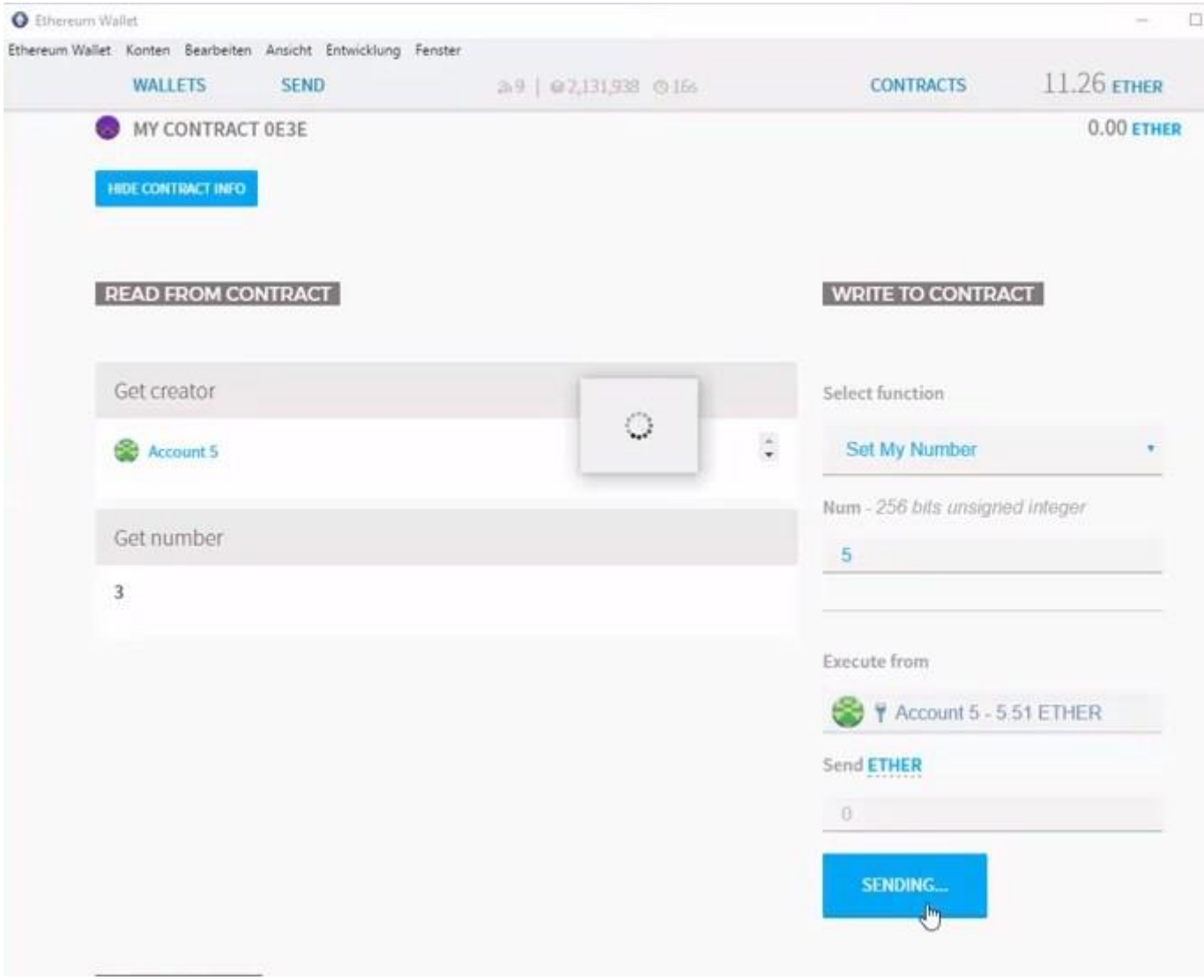
Можно вернуться в раздел Contracts и найти только что созданный контракт.



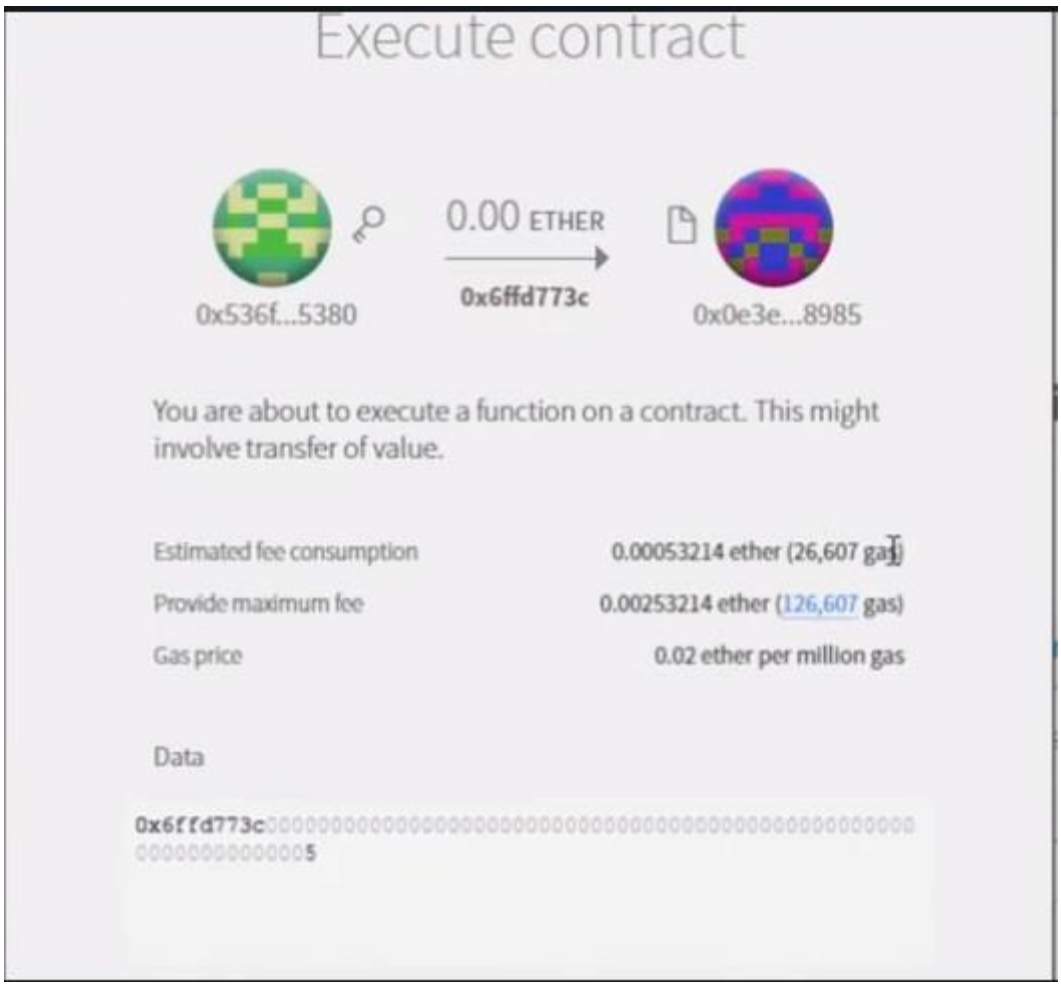
Когда вы добавляете новый контракт, все постоянные функции будут видны в левой части вашего кошелька, в разделе Read From Contract.



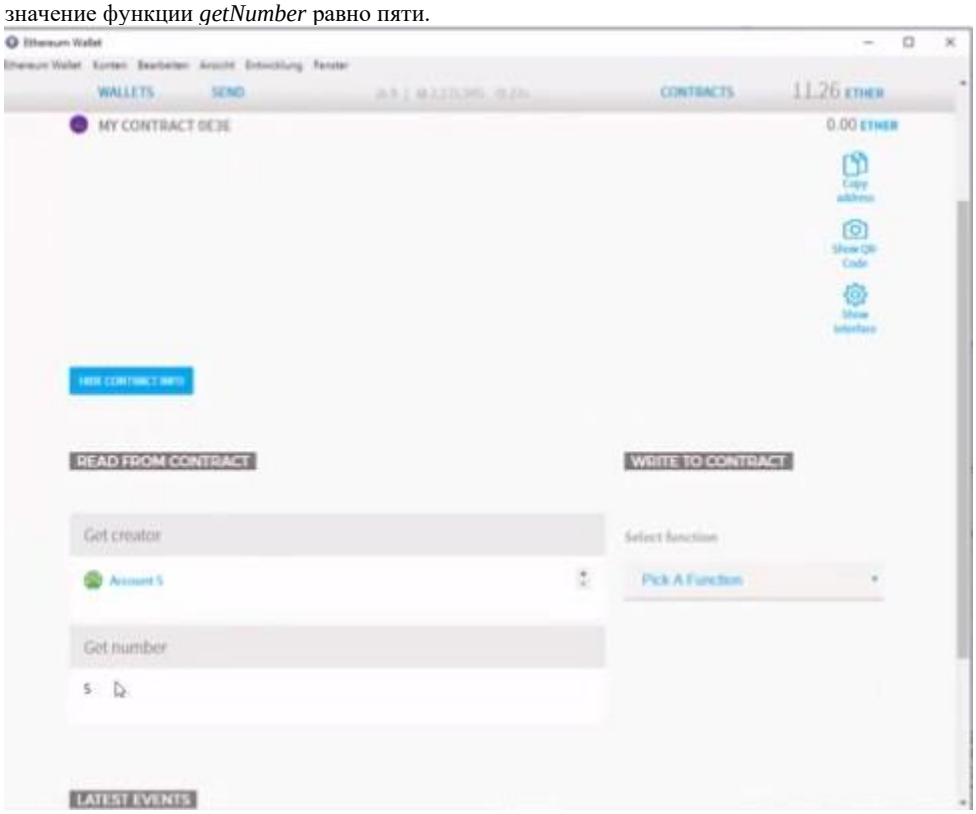
Для записи данных в контракт нужно выбрать функцию и ввести необходимые параметры, а затем отправить ее на выполнение, используя счет с достаточным количеством эфира.



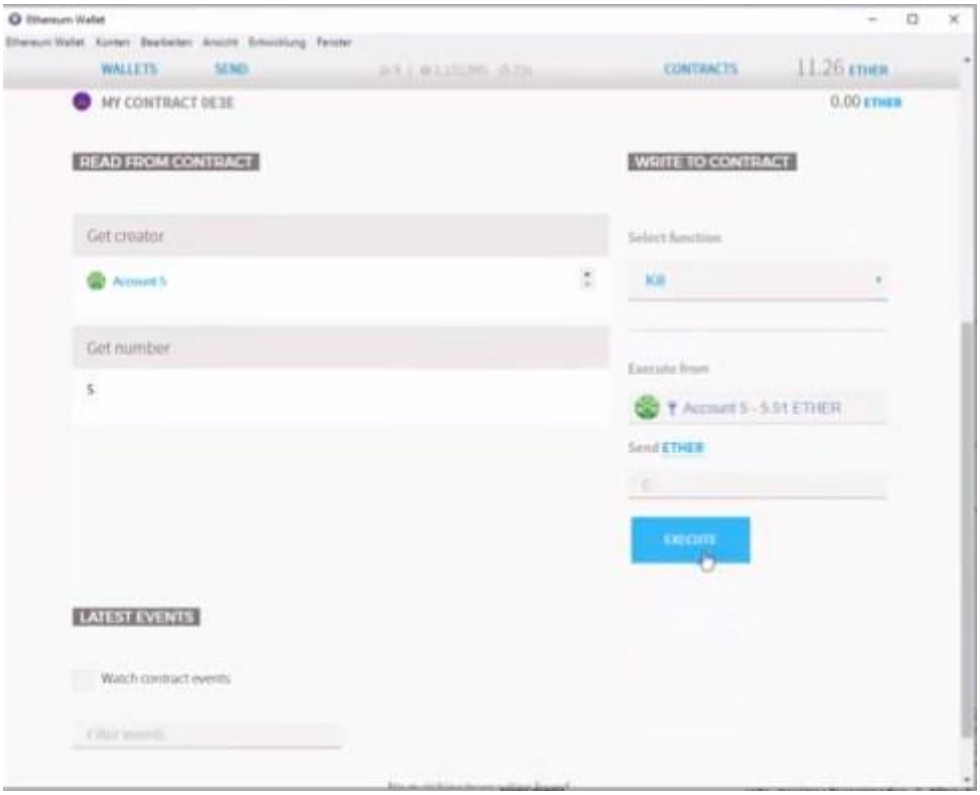
Обратите внимание, что для внесения изменений в контракт эфир не нужно кому-либо отправлять, он используется только для оплаты обсчета транзакции. Разумеется, если сам контракт не предполагает отправку эфира. После ввода пароля можно увидеть, что примерная стоимость обсчета контракта составит двадцать шесть тысяч шестьсот семь единиц газа, а максимальное количество выделяемого для обработки транзакции газа - на сто тысяч единиц газа больше. Неизрасходованные остатки газа будут возвращены на ваш счет.



После обработки транзакции функция *getNumber* вернет значение "пять". Выполнение контракта подтверждено двумя разными узлами. Вернемся в раздел Contracts и откроем контракт MyContract. Видно, что в разделе Read From Contract значение функции *getNumber* равно пяти.

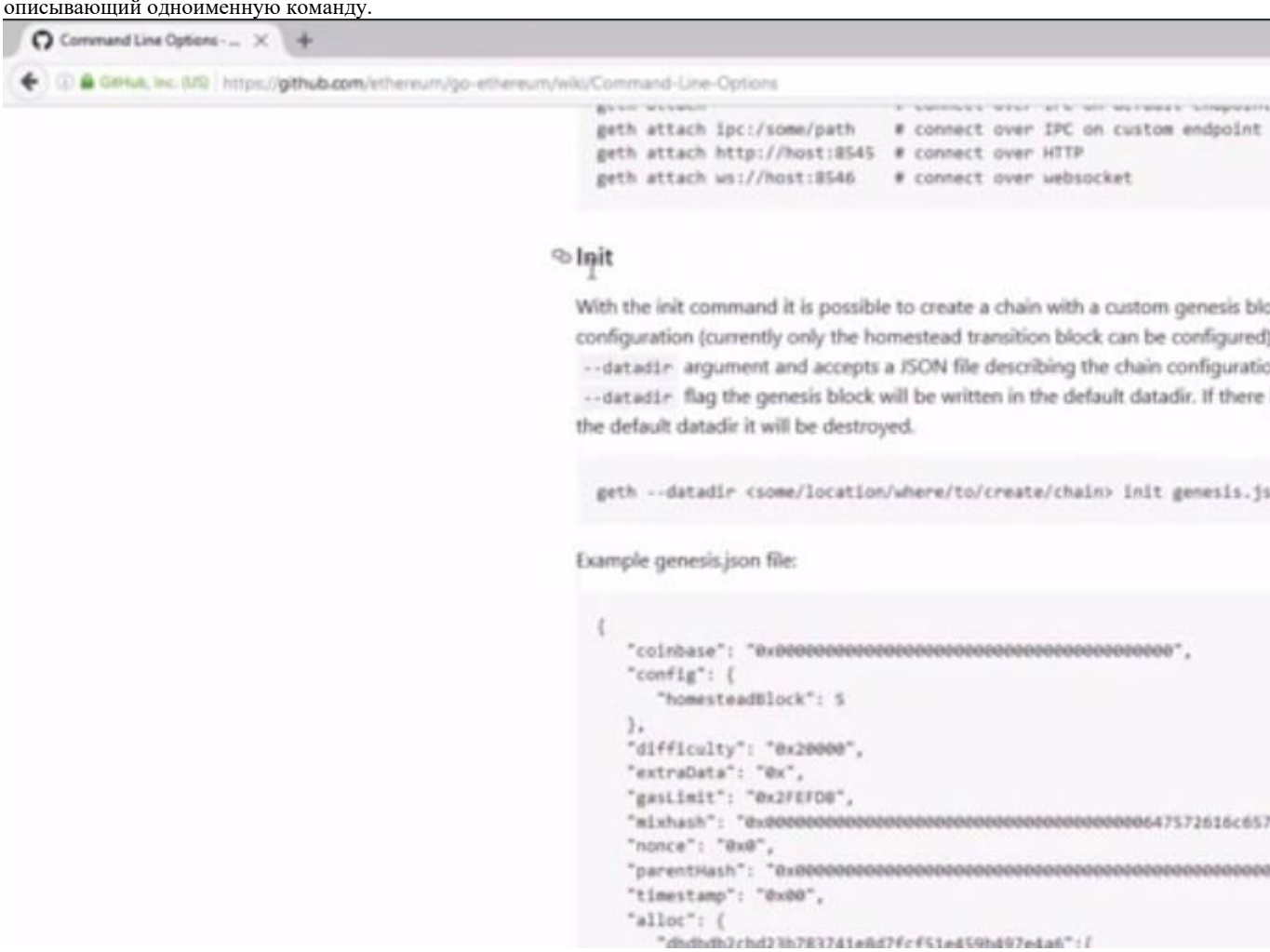


Теперь уничтожим контракт с помощью функции *kill*



Установка частного блокчейна

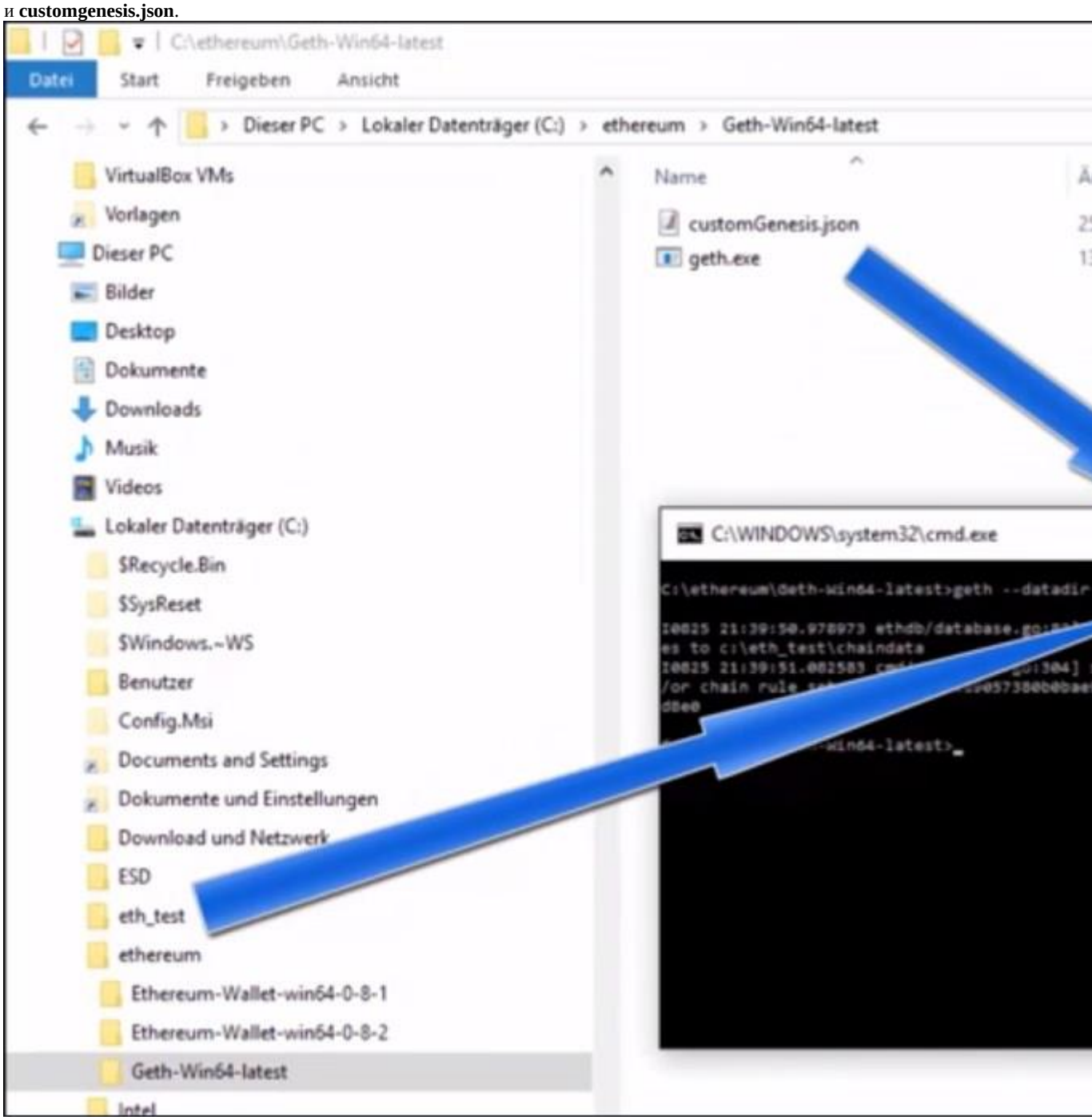
В этой части узнаем, как создать собственную частную сеть. На wiki-страницах Go Ethereum можно найти раздел Init, описывающий одноименную команду.



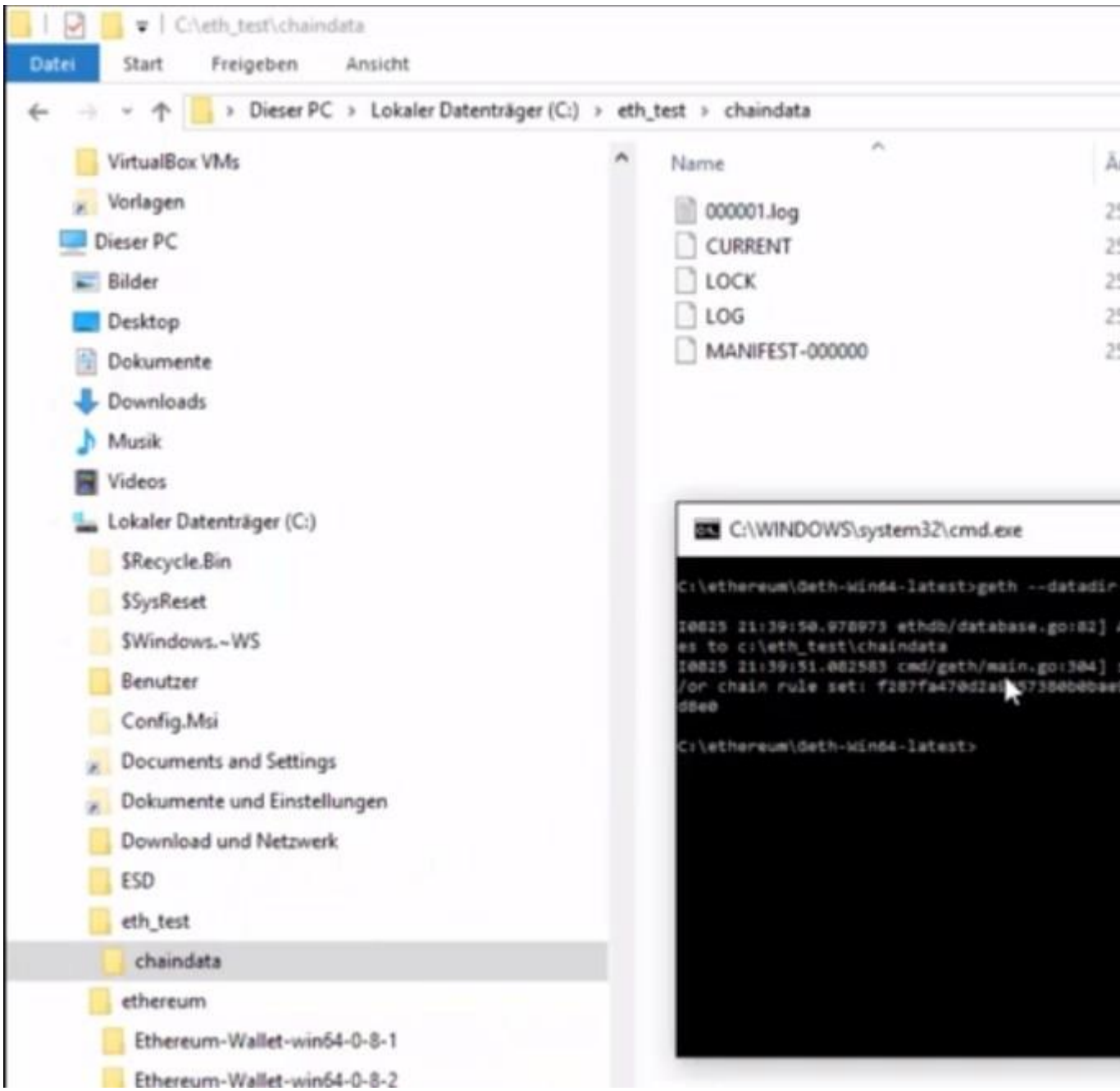
Эта команда позволяет задать в Geth каталог с данными и выбрать JSON-файл для генерации блокчейна, который - необходим для создания первого блока. Он состоит из нескольких конфигурационных параметров. Среда Ethereum предоставляет массу возможностей по созданию собственной частной сети. Единственный блок, к которому нужен особый подход при инициализации - это самый первый блок. В нашем случае это блок под номером ноль. Помимо собственно изначального блока, можно задать несколько внутренних конфигурационных параметров, например, напрямую назначить каким-то счетам определенное количество эфира..

Действуем по документации. Можно задать каталог с данными, хотя это и не обязательно, затем необходимо ввести команду *init*, а затем указать имя файла JSON. Если не задать каталог с данными, то изначальный блок будет записан в каталог по умолчанию. Если вы уже загрузили публичный блокчейн Ethereum, что на сегодняшний день составляет около пятнадцати гигабайт данных, и затем отдали команду на создание частного блокчейна с новым изначальным блоком, Geth произведет запись в каталог по умолчанию и тем самым сделает бесполезным весь объем загруженных данных. Они просто уничтожатся. Поэтому во всех случаях перед созданием частной сети необходимо сообщать Geth новый каталог с данными. Сейчас разберем на примере.

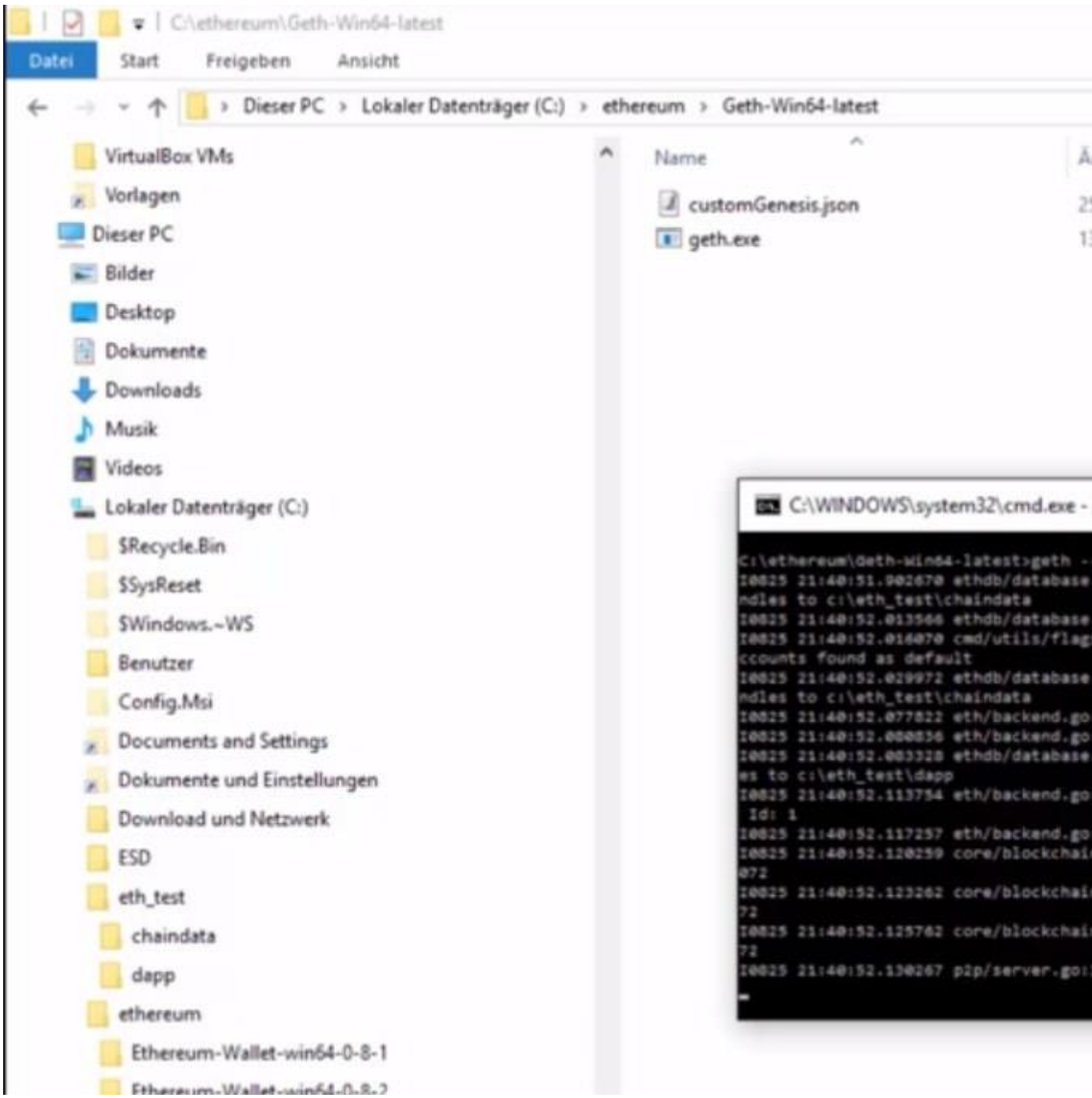
Откроем консоль. Наберем команду **geth-datadir**, а затем укажем путь к предварительно созданному каталогу. Затем **init** и **customgenesis.json**.



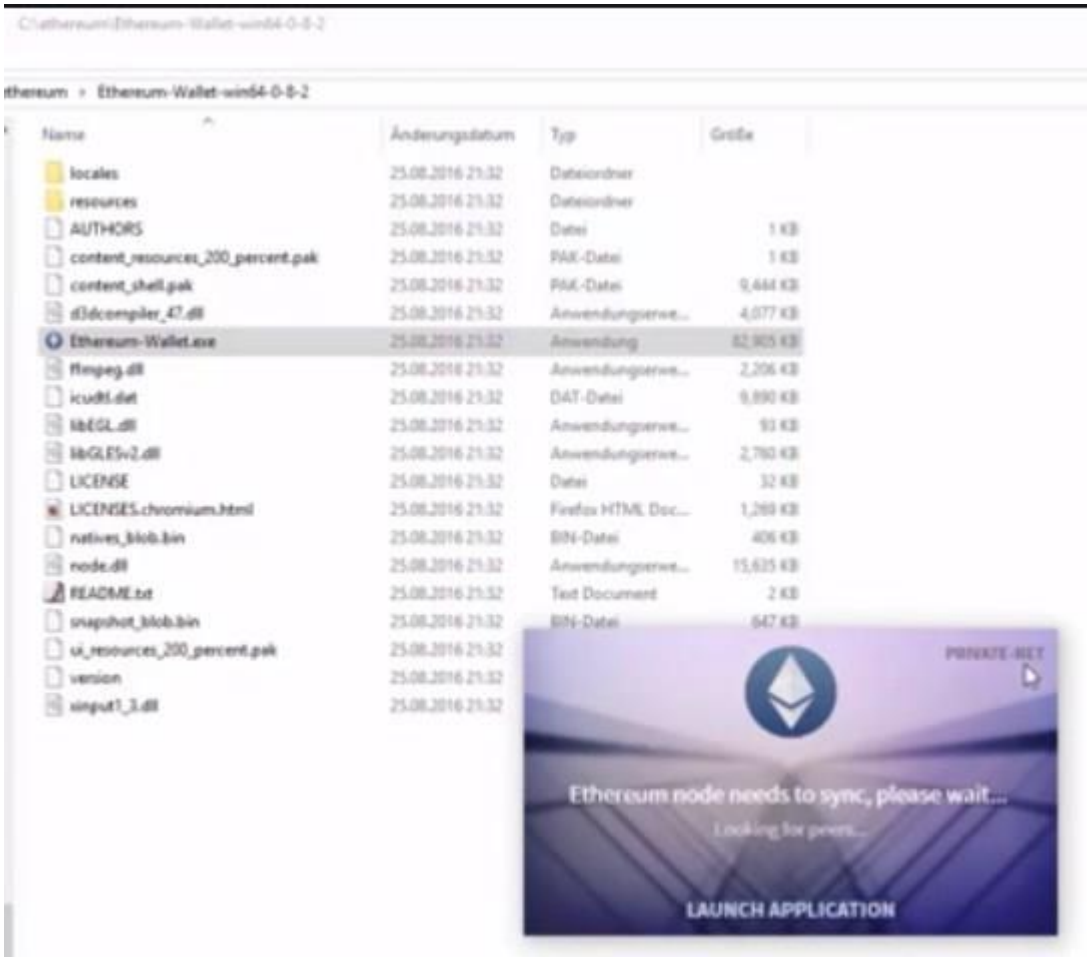
Автоматически выделилось пространство в шестнадцать мегабайт для кэш-памяти и шестнадцать логических номеров для файлов, расположенных в каталоге c:\eth_test\chaindata. Если перейти в этот каталог, можно увидеть все созданные файлы своими глазами.



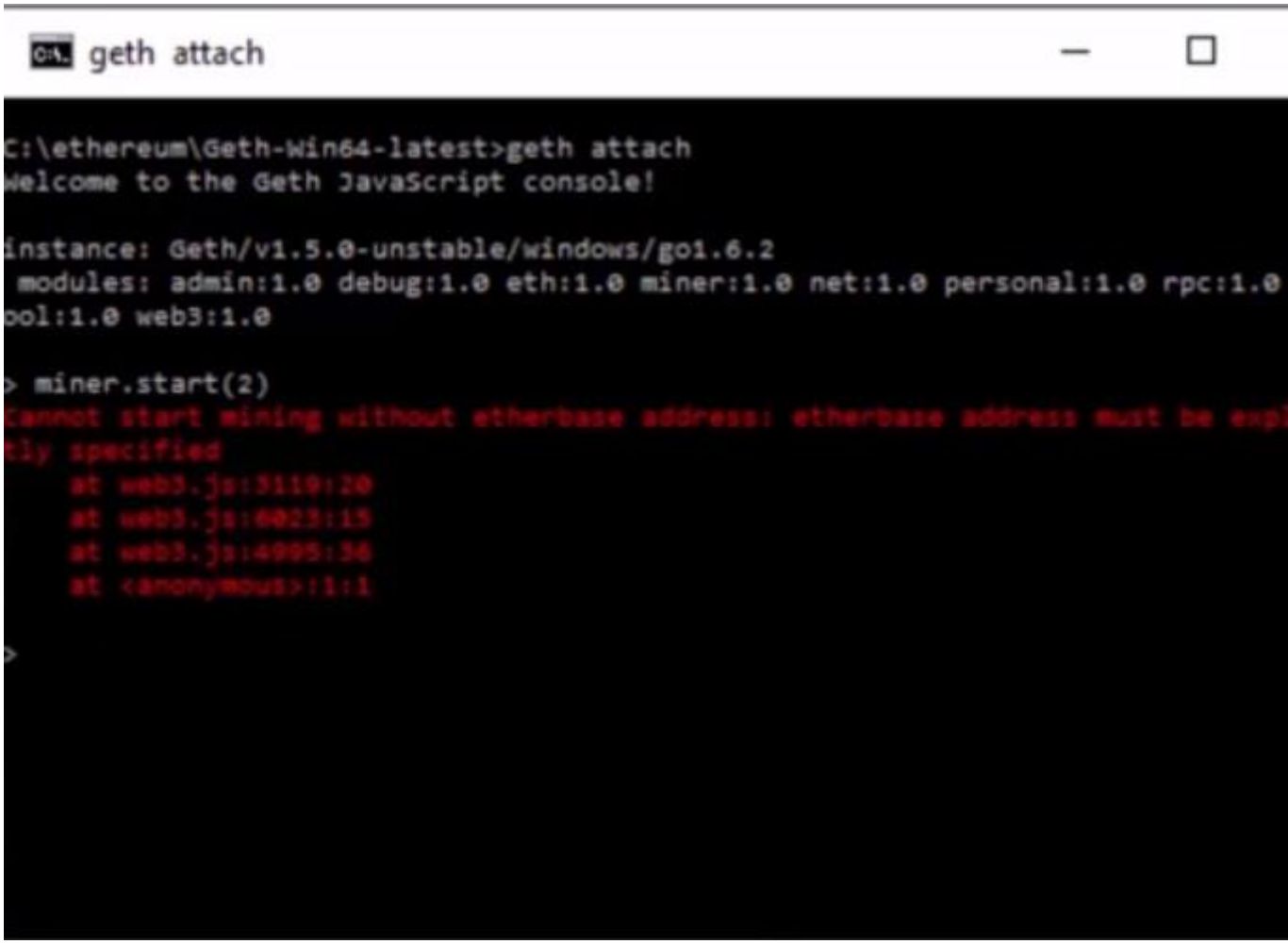
Запустим Geth.



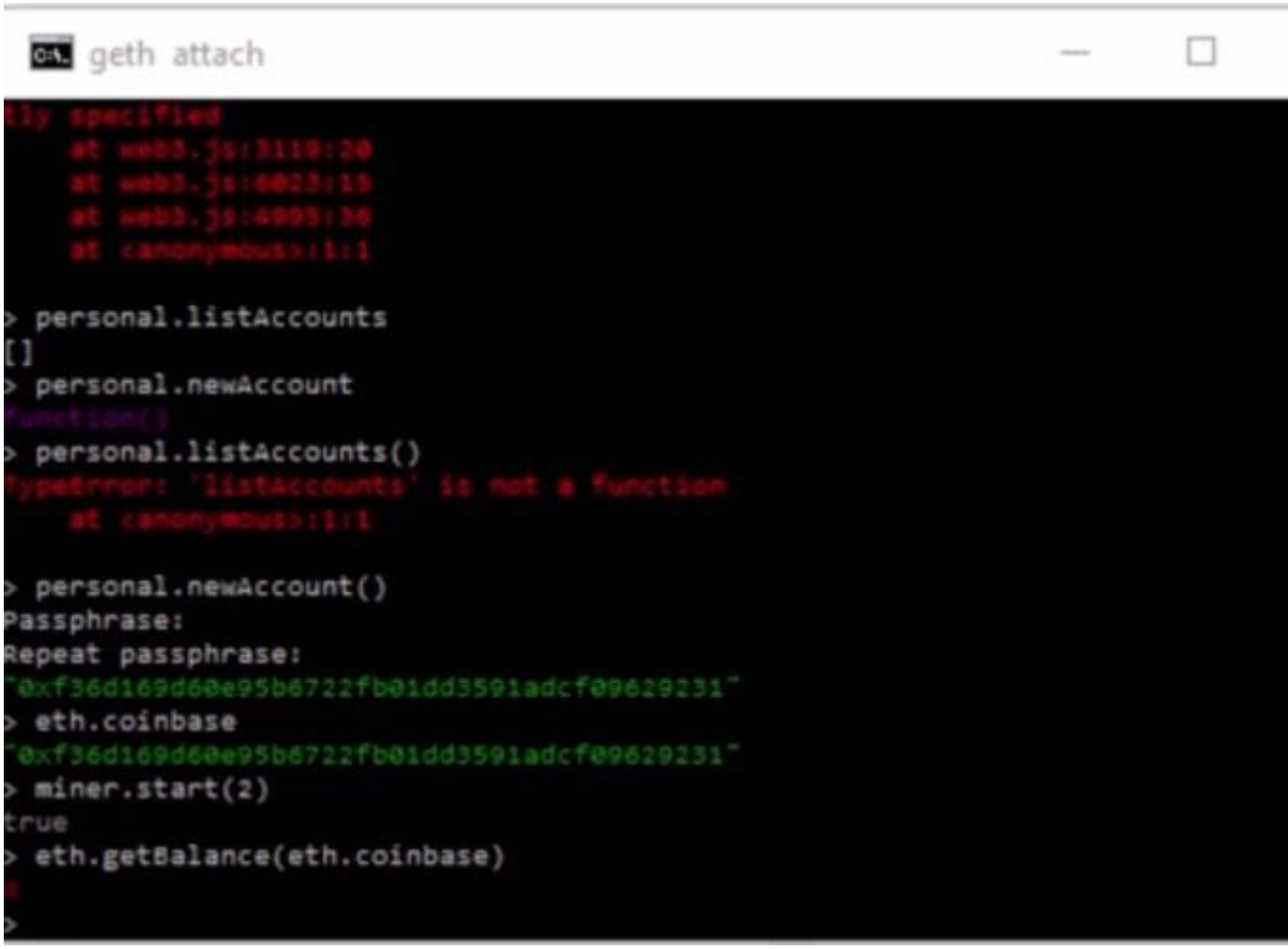
Работа с Geth в условиях частной сети не слишком отличается от работы с публичным блокчейном. Пожалуй, Geth не будет подключаться к другим узлам сети, это запрещено алгоритмами консенсуса Ethereum. Открываем кошелек, который подключен к частной сети.



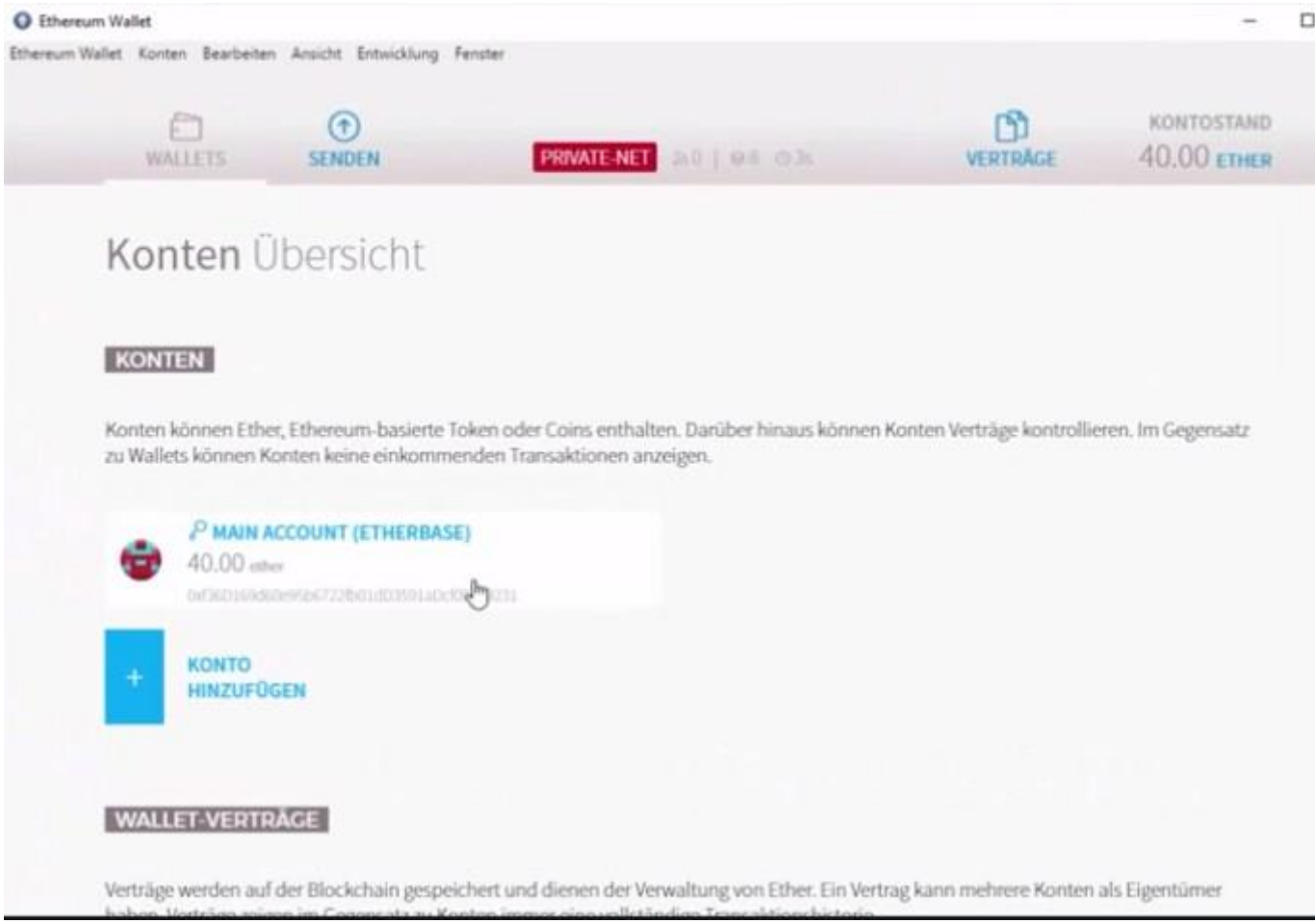
В параллельном окне можно открыть Geth еще раз и запустить майнер. Видим, что счетов нет



Тогда создадим счет и зададим ему пароль test123. Проверим основной криптовалютный счет.



Посмотрим, как майнится эфир.



Майнинг эфира в частной сети идет невероятно быстро, ведь этому нет никаких препятствий. В сети просто нет ни одного другого узла. Это значит, что все добываемые единицы эфира начисляются на наш основной счет. Можно создавать контракты, открывать новые счета, пересылать эфир и тому подобное. Все эти операции делаются в частной сети.

Задание №1 - Понимание основ среды Ethereum перед тем, как начать работу

Публичный блокчейн

В этом задании потребуется установить кошелек Mist и среду Go-Ethereum. Для работы с публичным блокчейном Mist должен его загрузить. Кроме того, потребуется около 20 гигабайт дискового пространства.
Загрузить кошелек Mist можно по этой ссылке: https://github.com/ethereum/mist/releases
Убедитесь, что у вас есть свободное дисковое пространство и достаточная пропускная способность сети. Рекомендуется выполнять загрузку в ночное время.

Частный блокчейн

Если ваша основная цель - разработка распределенных приложений, а размещать их в блокчейне вы планируете позднее (например, с помощью альтернативного кошелька Jaxx), будет полезным научиться работать с частным блокчейном.
Для запуска частной сети загрузите клиент go-ethereum:https://github.com/ethereum/go-ethereum/releases
Загрузите JSON-файл для создания изначального блока: https://github.com/ethereum/go-ethereum/wiki/Command-Line-Options
Убедитесь, что кошелек Mist не запущен в фоновом режиме.
Создайте новый каталог и запустите команду Geth со следующими параметрами:

```
geth --datadir <some/location/where/to/create/chain> init genesis.json
```

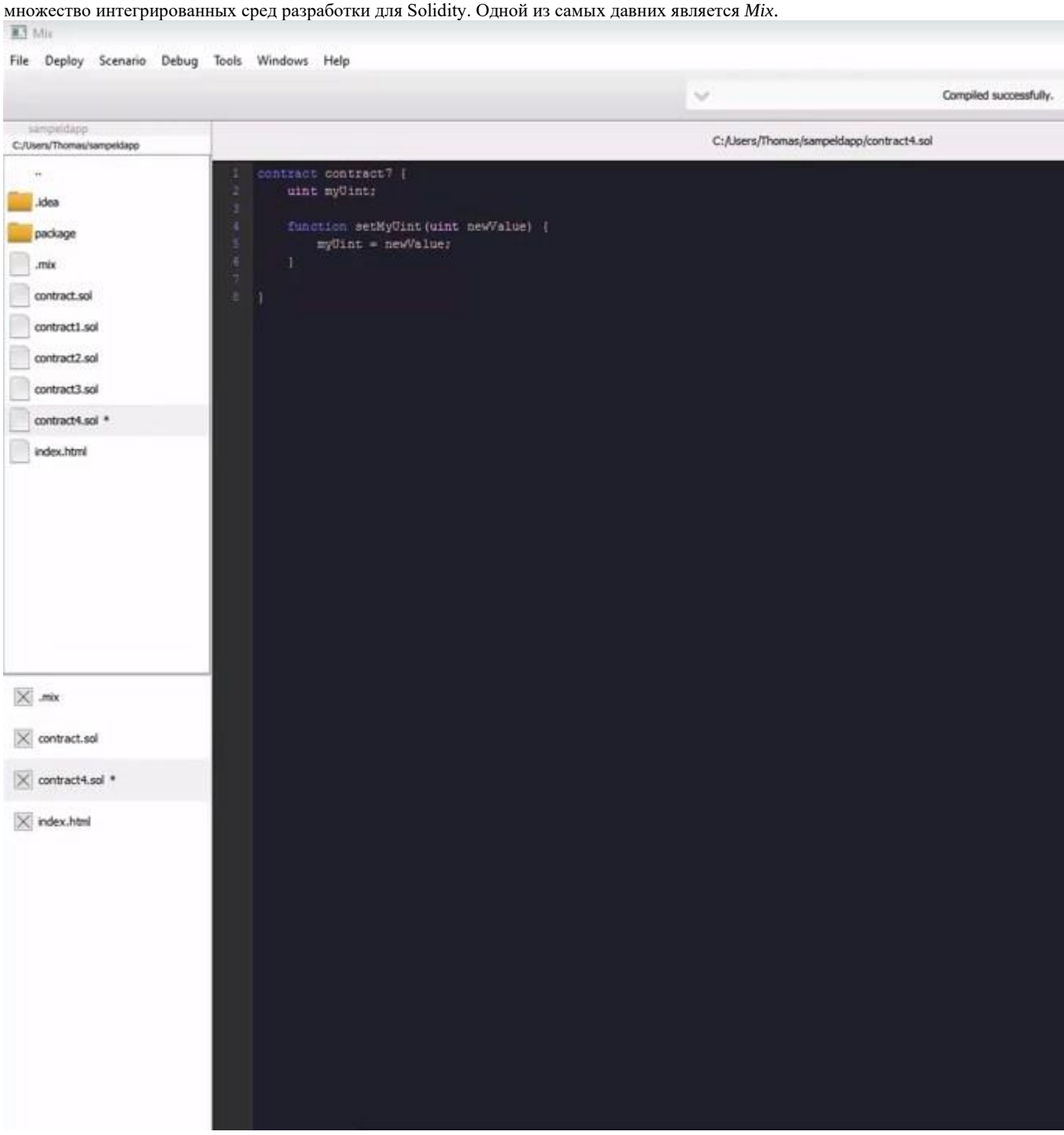
Задание выполнено.

Приложение №2

Отладка контрактов

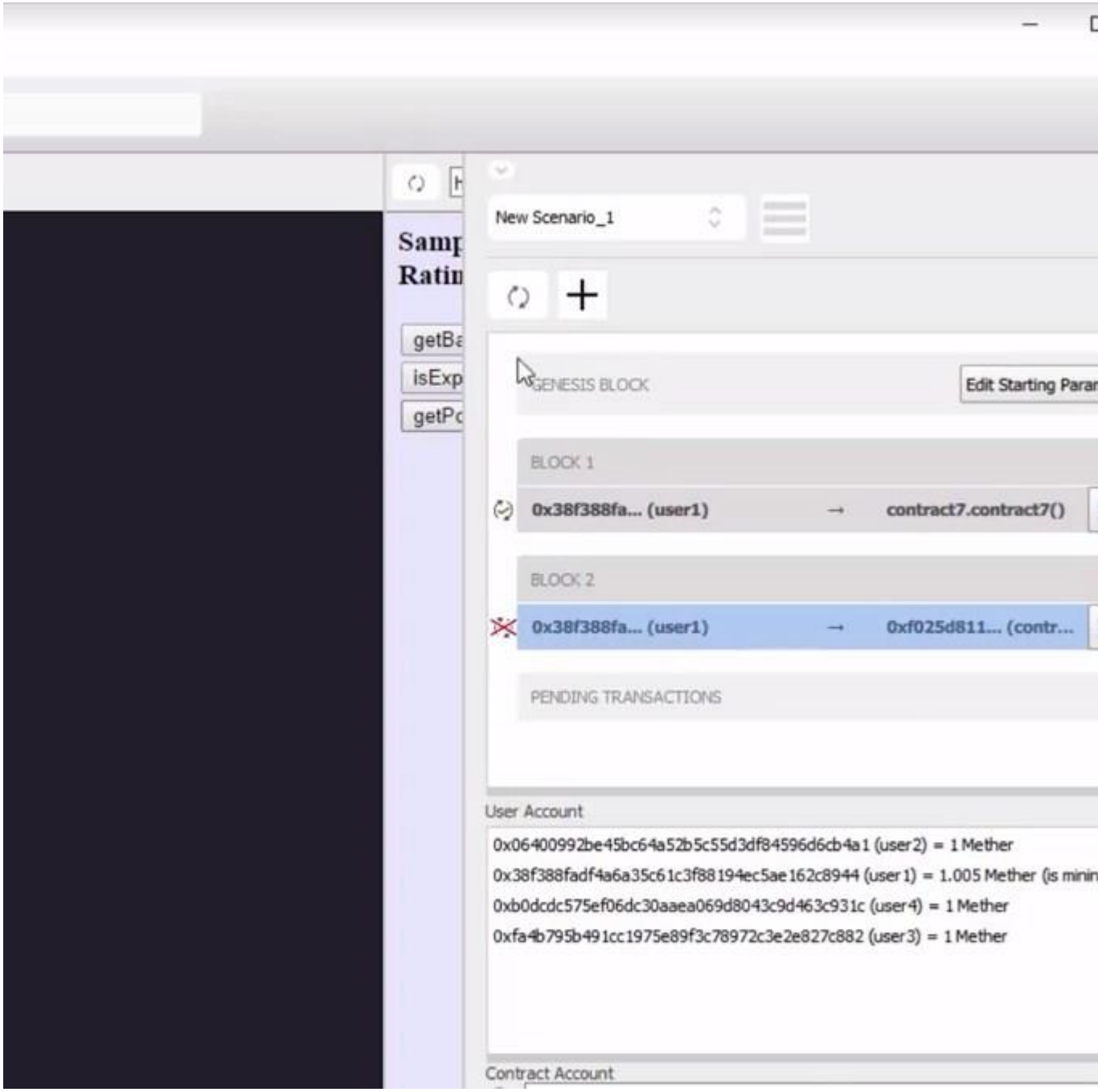
Отладка контрактов

В этой части поговорим о том, как выполнять отладку контрактов в среде Solidity. В настоящее время существует множество интегрированных сред разработки для Solidity. Одной из самых давних является *Mix*.

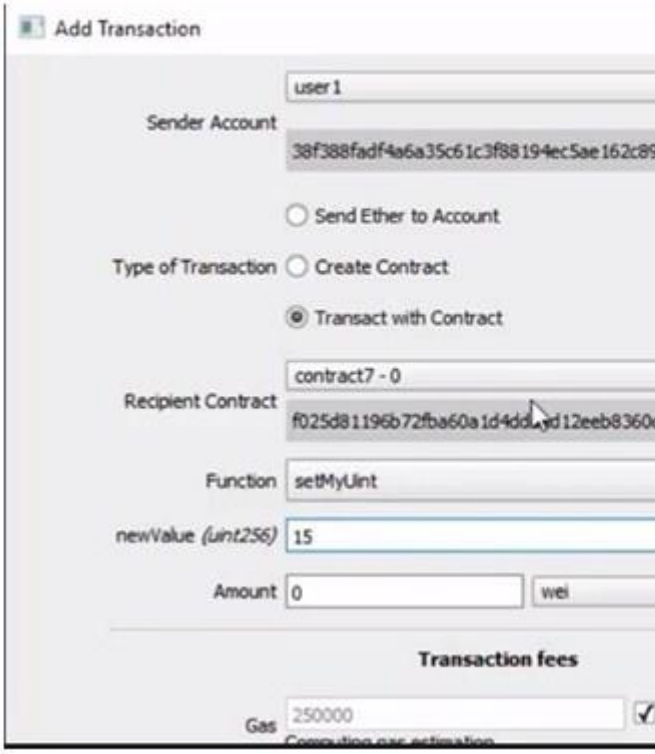


Она не слишком популярна, но ее встроенные *механизмы* отладки весьма функциональны. Давайте разберемся, как они работают, на простом примере.

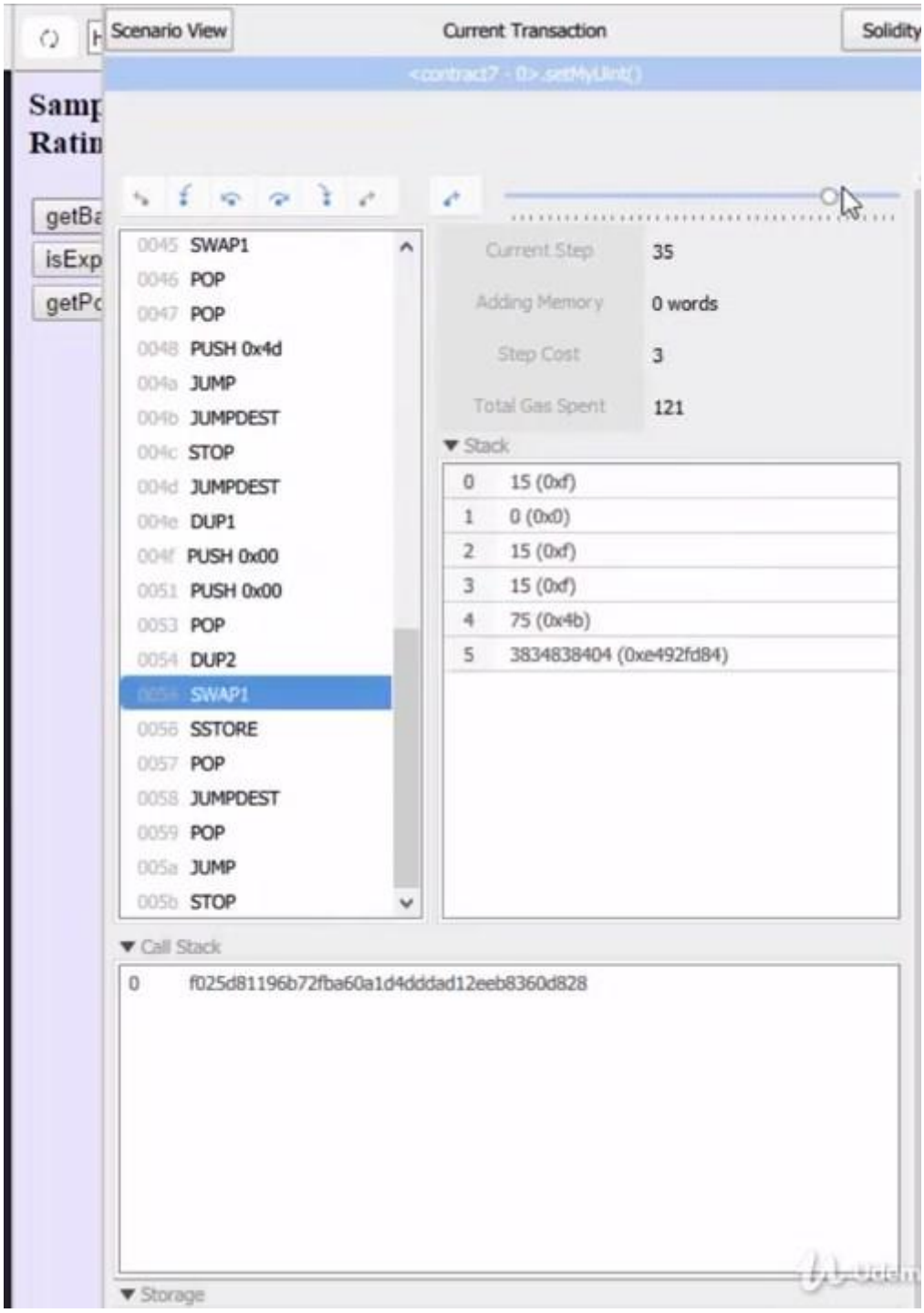
Предположим, у нас есть простой контракт с единственной функцией, которая задает число типа *uint*. В *Mix* эмуляция блокчейна работает *по* так называемым сценариям. Создаем новый *сценарий* и выбираем для компиляции свой контракт. Удалим ненужный блок.



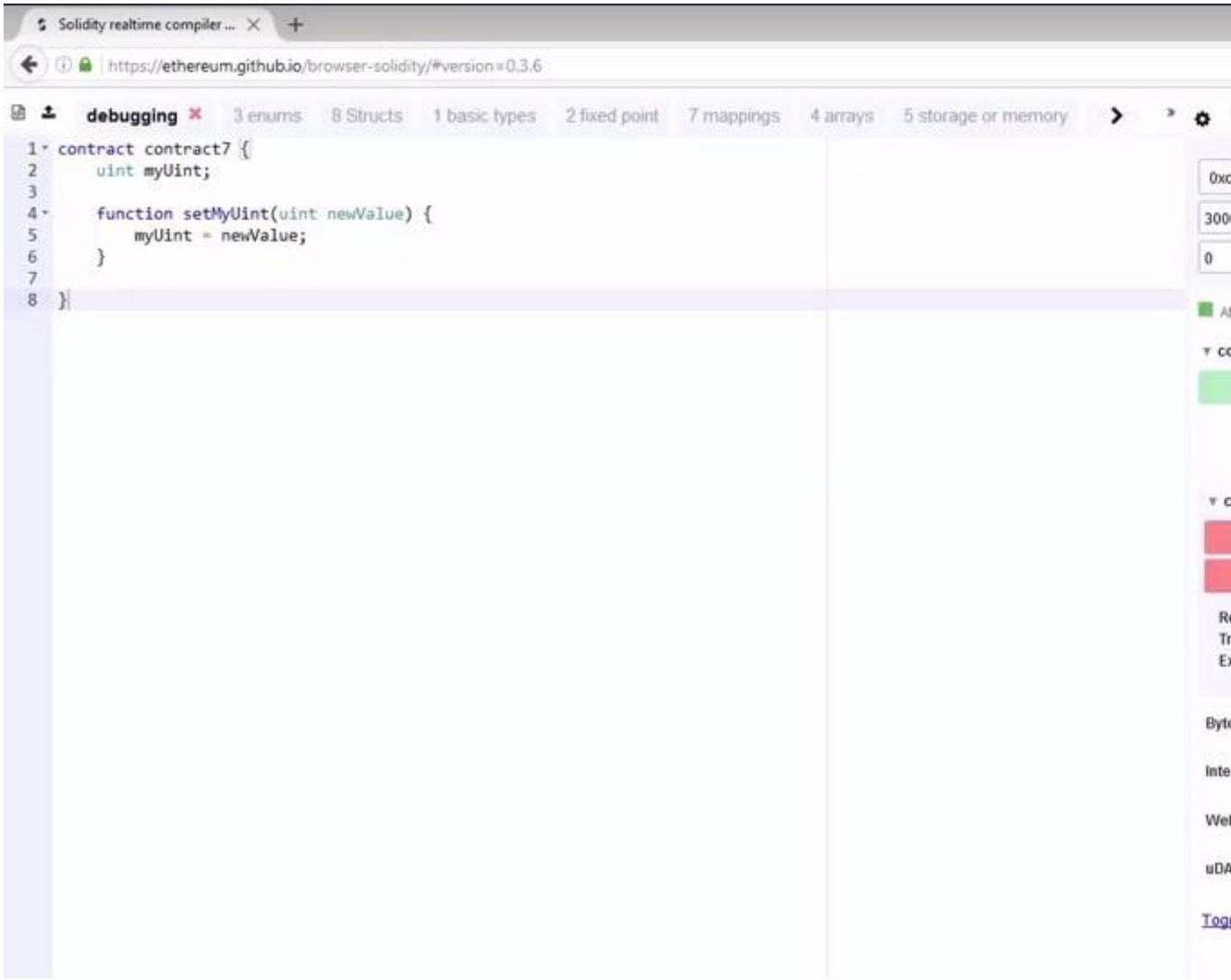
В самом начале процесса отладки существует блок, в котором компилируется контракт и вызывается *конструктор*. Предположим, нам нужно взаимодействовать с этим контрактом, например, вызвать функцию *setMyUint* и установить *значение*, равное пятнадцати.



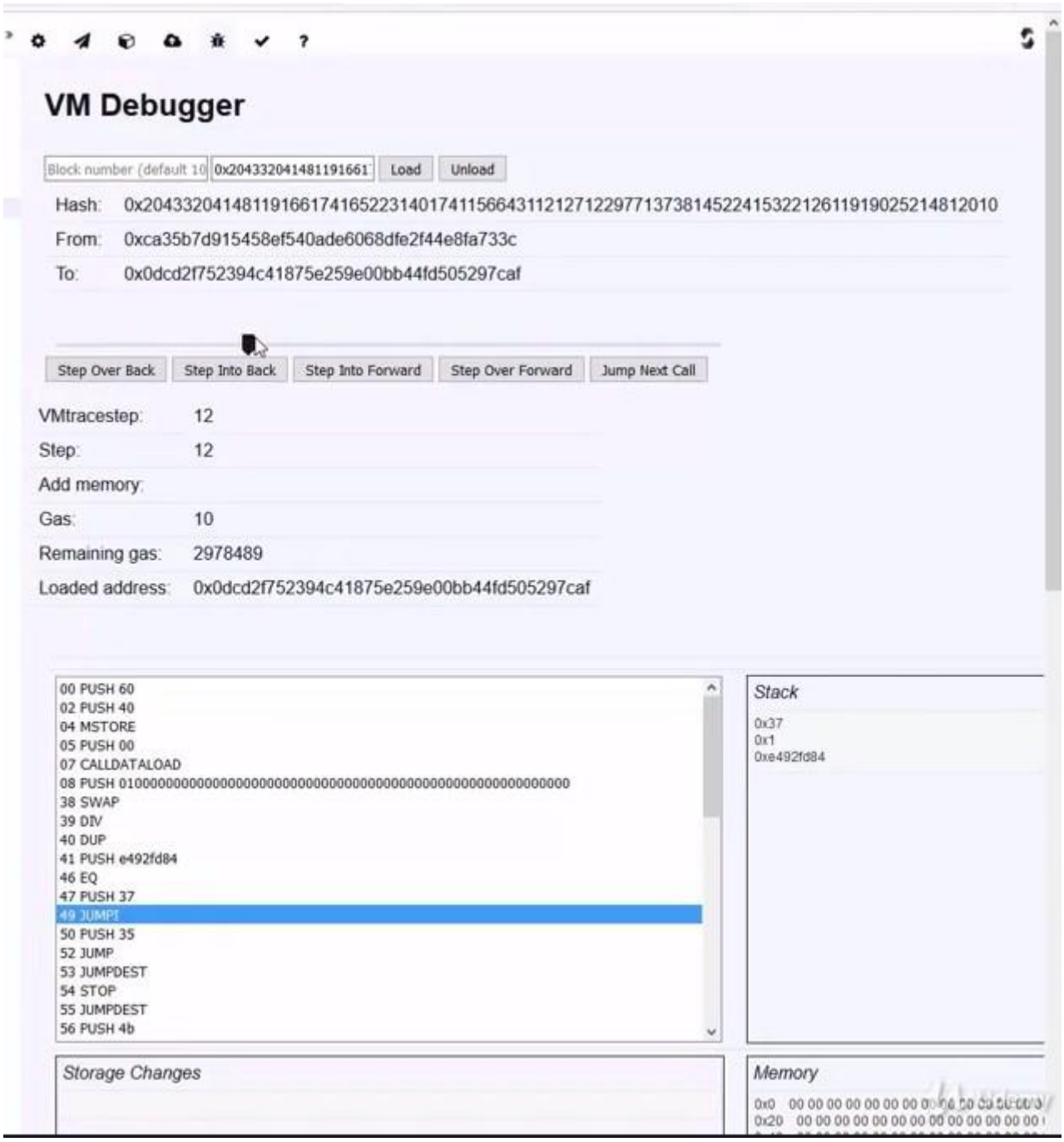
Блок обсчитан, и *транзакция* обработана. В режиме отладки можно пройти через процесс обработки транзакции *по* шагам, и видеть количество газа, израсходованное на каждый из шагов. Можно пойти на уровень глубже и проанализировать весь процесс в кодах для виртуальной машины.



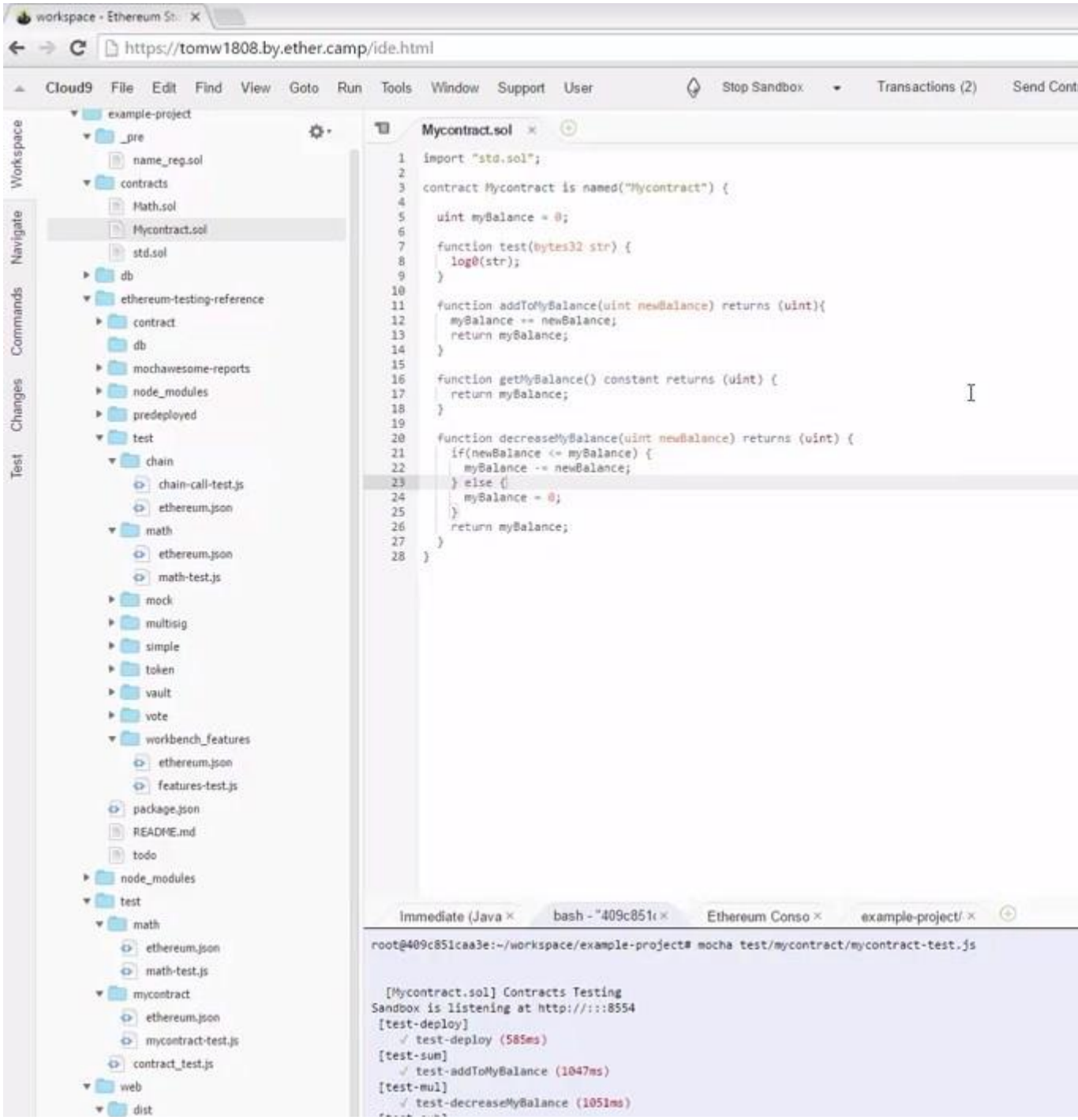
Посмотрим, как тот же пример работает в браузере Solidity. Выполняем *операции по* созданию контракта и выставляем *значение* для *myUint*, равное пятнадцати.



Браузер Solidity тоже может помочь в отладке контрактов, правда, в нем есть только уровень отладки в кодах для виртуальной машины. Понять, в каком месте кода вы находитесь, нелегко, ведь *браузер* этого не показывает.



С другой стороны, *Mix* хорошо работает как для малых, так и для больших контрактов. К сожалению, активная разработка *Mix* остановлена. И, наконец, еще одним полезным инструментом является Solidity Studio, который можно загрузить на сайте http://Ether.camp. Это очень функциональная *среда разработки* на базе Cloud9. В ней отсутствует возможность отладки контрактов, но, с другой стороны, есть множество тестов для отработки функциональности контрактов.



Задание №2 - Обзор всех аспектов среды Solidity

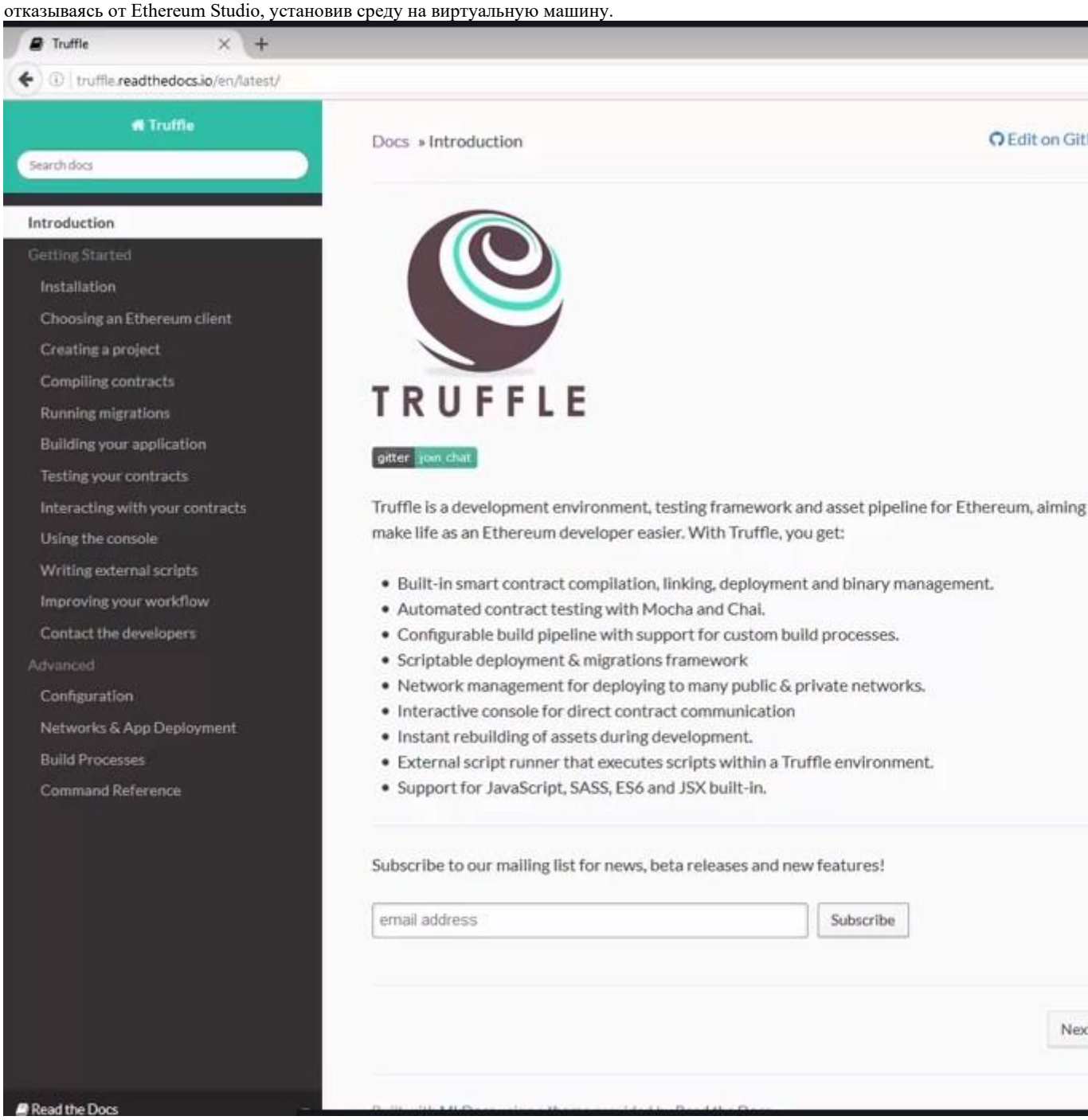
В этом задании вам предстоит ознакомиться со средой разработки Solidity и браузером для работы с ней.
Откройте *браузер* Solidity: https://ethereum.github.io/browser-solidity/
Изучите раздел "Solidity in Depth" из документации *по* среде Solidity: https://solidity.readthedocs.io/en/develop/solidity-in-depth.html
Проработайте примеры из учебного курса, освойте использование переменных различных типов: численных *(int/uint)*, символьных *(byte/string)*, булевых, а также массивов, ассоциаций и структур.
Ответьте на следующие вопросы:

1. Что произойдет, если присвоить переменной типа *uint* значение в -1?
2. Что произойдет, если присвоить переменной типа *uint8* другую переменную типа *uint256* со значением 5?
3. Что происходит, когда значение одной строковой переменной присваивается другой с отличной от первой длиной? Отличается ли количество затрачиваемого на обсчет этой процедуры газа? Почему?
4. Какую функцию несет символ подчеркивания ("_") в модификаторах функций?
5. Сколько веев в одном финни?

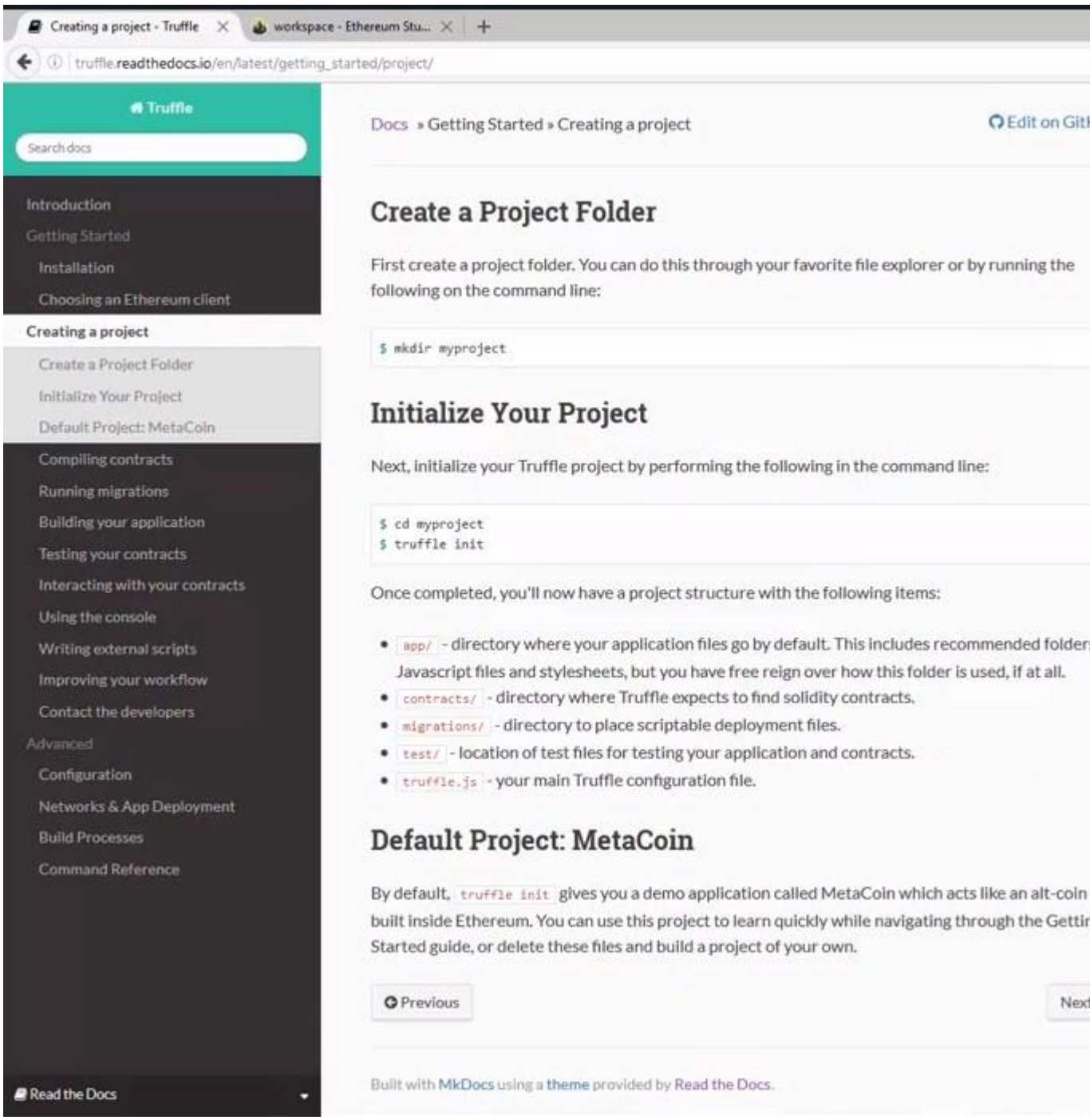
Введение в работу с Truffle и инструкции по установке

Среда разработки Truffle включает в себя тестовую среду и является ценным инструментом для платформы Ethereum. Функционал Truffle напоминает возможности Ethereum Studio, отличакясь только тем, что ее можно запускать локально.

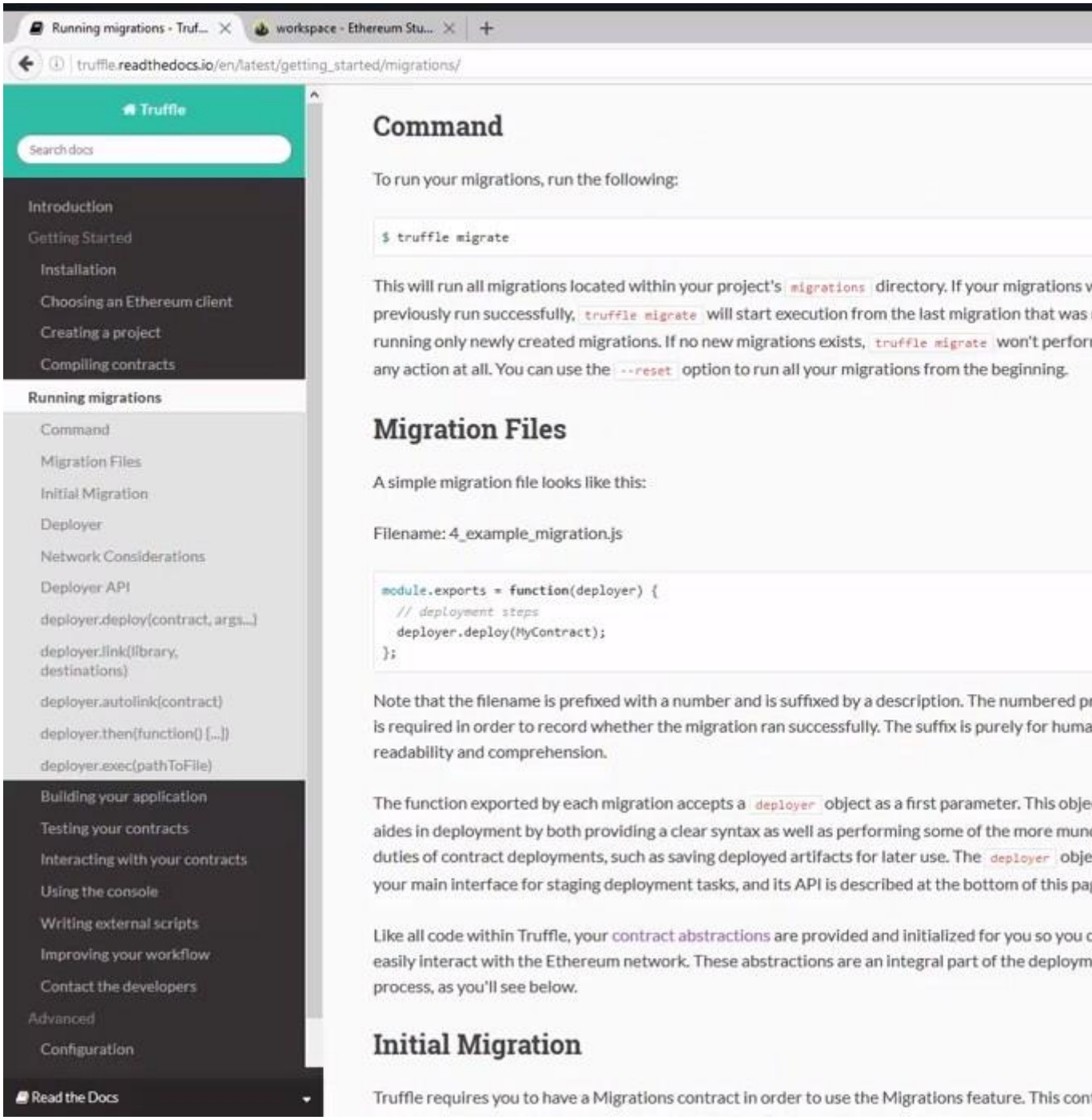
Truffle представляет собой просто программный пакет для узлов сети Ethereum, поэтому работать с Truffle можно, не отказываясь от Ethereum Studio, установив среду на виртуальную машину.



Для полноценной работы с Truffle вам потребуется клиент Ethereum, обеспечивающий подключение к блокчейну - либо реальному, либо тестовому. При локальной разработке можно пользоваться EthereumJS TestRPC. А при работе в Ethereum Studio пригодится режим песочницы. Чтобы подключиться к реальному блокчейну, целесообразно использовать Geth или другой клиент сети Ethereum. Создать проект в среде Truffle легко: необходимо всего лишь перейти в каталог с проектами и набрать команду *truffle init*. *По* умолчанию Truffle содержит проект под названием MetaCoin. Он содержит несколько контрактов, тестов, а также эмуляцию сети.



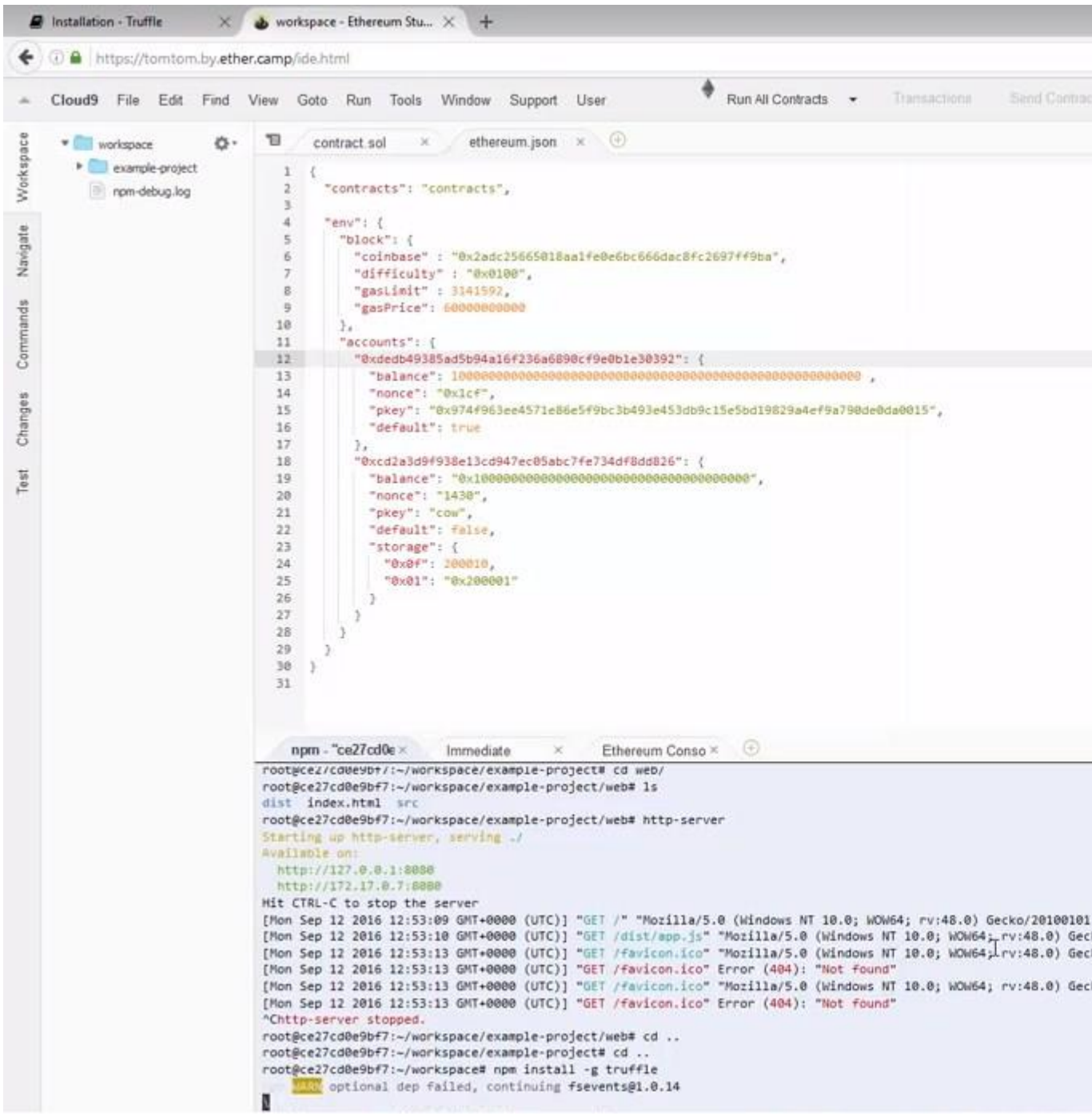
Компиляция контрактов в Truffle проходит легко. Среда сама позаботится о подключении библиотек и импортированных данных. Все скомпилированные контракты размещаются в папке *./build/contracts*.
В среде Truffle можно работать с миграциями. Миграции - это разновидность файлов JavaScript, позволяющие размещать контракты в сети Ethereum; в них указывается порядок, в котором контракты должны обсчитываться сетью. Кроме того, миграции обеспечивают подключение библиотек. Сразу после создания файлов миграции можно ввести команду *truffle migrate*, и начнется *размещение* контрактов в блокчейне.



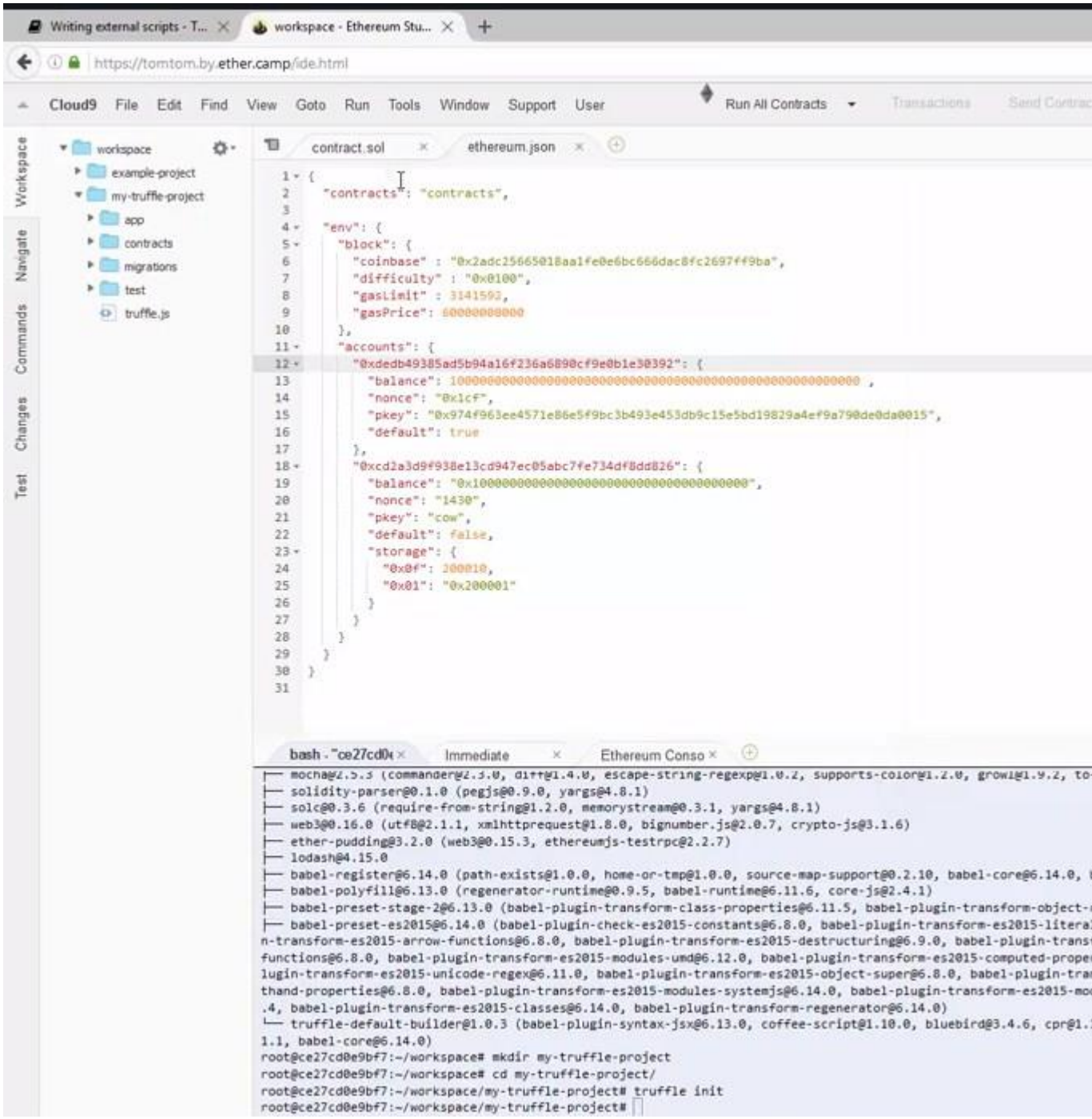
В состав среды Truffle также входит система сборки кода, облегчающая начало работы. Для сборки приложения необходимо набрать команду *truffle build*.

Тестирование контрактов в среде Truffle производится с помощью системы Mocha ("Мокка"). Для запуска теста нужно всего лишь набрать команду *truffle test*.

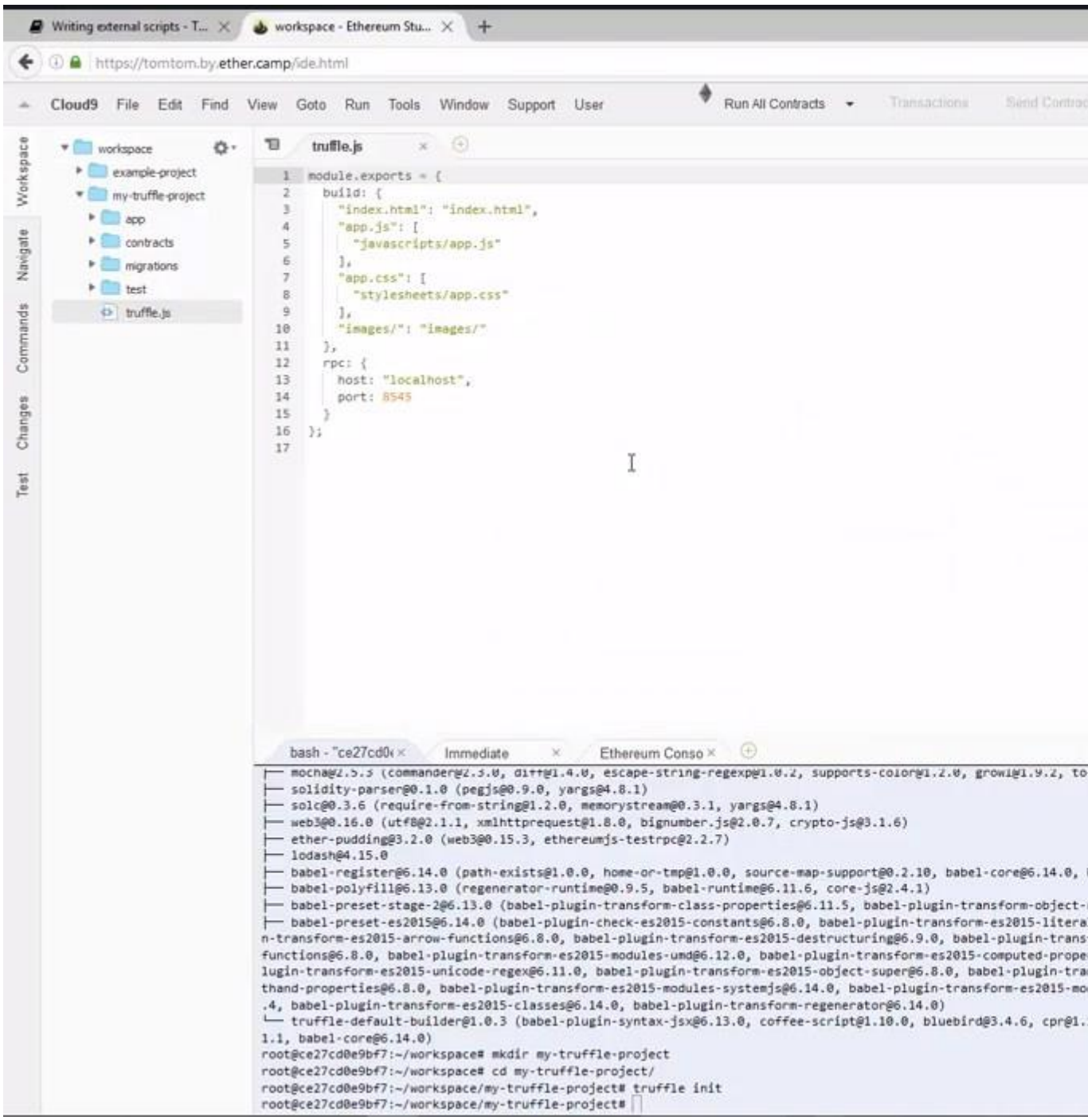
Давайте разберемся, как установить Truffle. Поскольку это пакет для узла сети, установку можно запустить с помощью программы-менеджера подобных пакетов. Для установки Truffle глобальным образом наберите команду *npm install -g truffle*. Выполним эту операцию в Ethereum Studio. Вводим команду и ждем окончания установки.



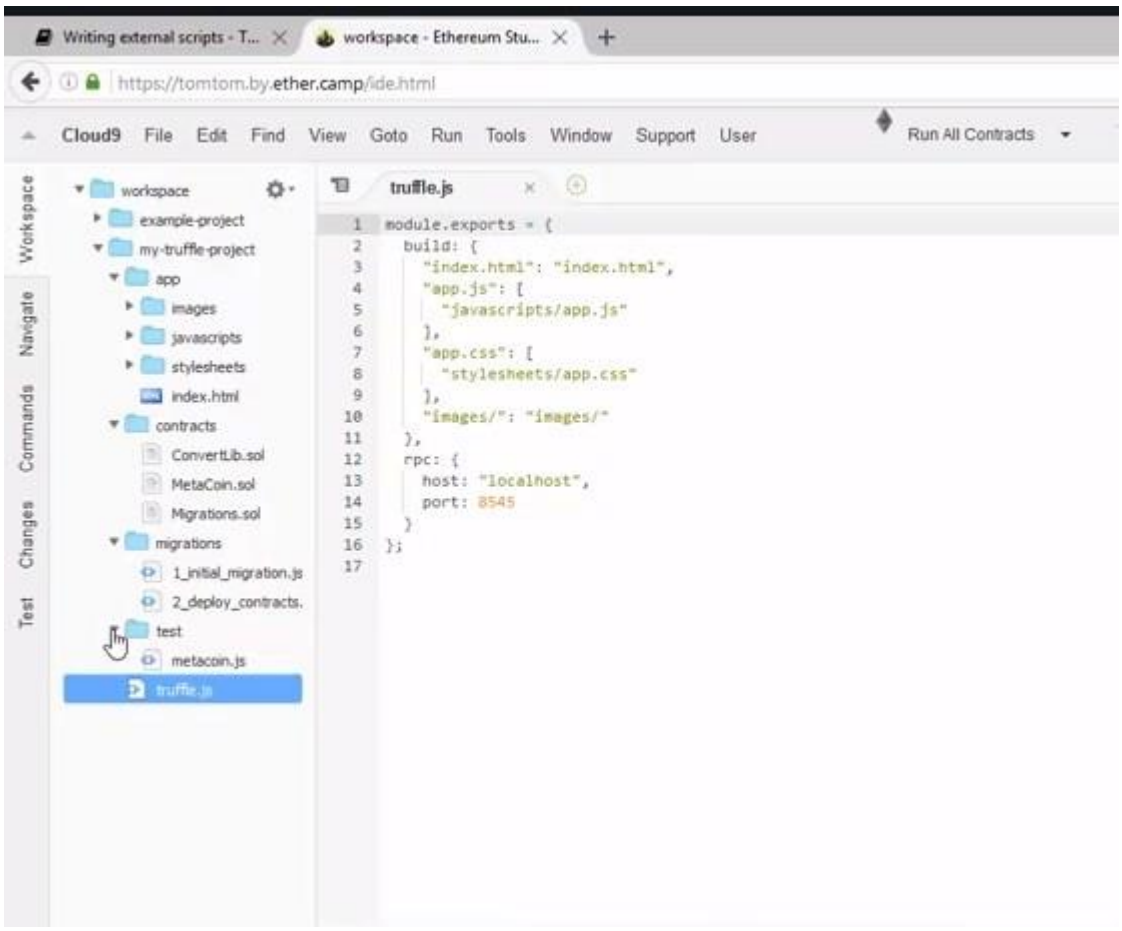
Создадим новый каталог и начнем первый проект в среде Truffle. Он сохраняется в папке **my-truffle-project**.



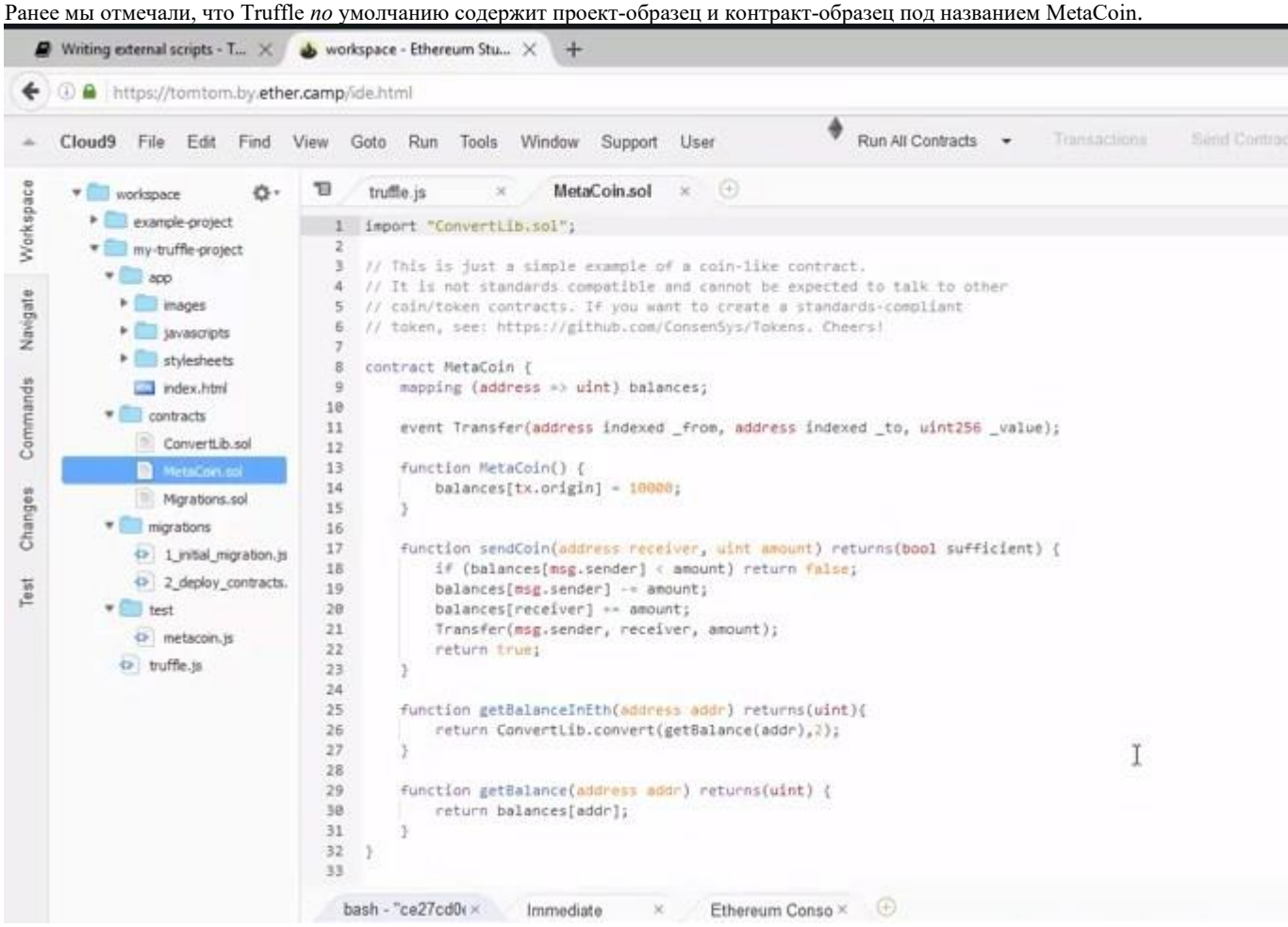
В корневом каталоге проекта Truffle находится *файл* **truffle.js**. Для того, чтобы можно было работать в режиме песочницы, внесем в него некоторые изменения, о них расскажем подробнее позднее.



В каталоге **App** размещено наше *приложение*, а в каталоге **Contracts** - используемые контракты. В каталоге **Migrations** расположены файлы JavaScript для размещения приложения в блокчейне, а в каталоге **Test** - тесты.



Ранее мы отмечали, что Truffle *по* умолчанию содержит проект-образец и контракт-образец под названием MetaCoin.



Видно, как импортируются различные файлы и библиотеки для среды Solidity, а также их взаимодействие при размещении приложения в блокчейне.

Взаимодействие между контрактами и веб-сайтами

В этой части обсудим взаимодействие между сетью Ethereum и веб-сайтами посредством технологии web3. Среда Ethereum представляет собой блокчейн, единый счет, в который записаны все транзакции. Кроме того, есть виртуальные машины Ethereum, на которых *приложение* выполняется в виде кодов операций. *Коды операций* выглядят примерно так: *PUSH* 1 CALLDATALOAD и так далее. Среда Solidity переводит все эти *коды операций* в байткод *Java*. Доступны и другие языки, похожие на Solidity. Все они являются языками высокого уровня. Это напоминает работу на компьютере с компилятором C++ или C *по* сравнению с задачей команд через *ассемблер*.

Web3 работает с кодами *RPC*. Рассмотрим пример с узлом Geth или Ethereum-CPP. К ним может подключиться код *web3.js*. Также можно воспользоваться Ethereum-testRPC. Эти узлы предоставляют *доступ* к блокчейну. В языках *HTML* и JavaScript существует *объект Web3.js*. Здесь можно задать *значение* переменной *web3.setProvider*, указав в ней узел *HTTP*. С помощью протокола *HTTP программа* выполнит подключение к интерфейсу *RPC*. Например, *адрес* счета для зачисления эфира можно узнать с помощью функции *web3.eth.coinbase*, а баланс - с помощью *web3.eth.getBalance*.

В языке JavaScript можно работать с контрактами. Они создаются путем задания массива *ABI* и сообщения в web3, что контракт найден. *Исполнение* процедуры web3.eth.*contract* и сообщение массива *ABI* создаст *объект* **MyContract**. В реальности контракт воплощается после назначения ему адреса. Больше информации на эту тему можно почерпнуть из wiki-пространства Ethereum: https://github.com/ethereum/wiki/wiki/JavaScript-API#web3ethcontract.

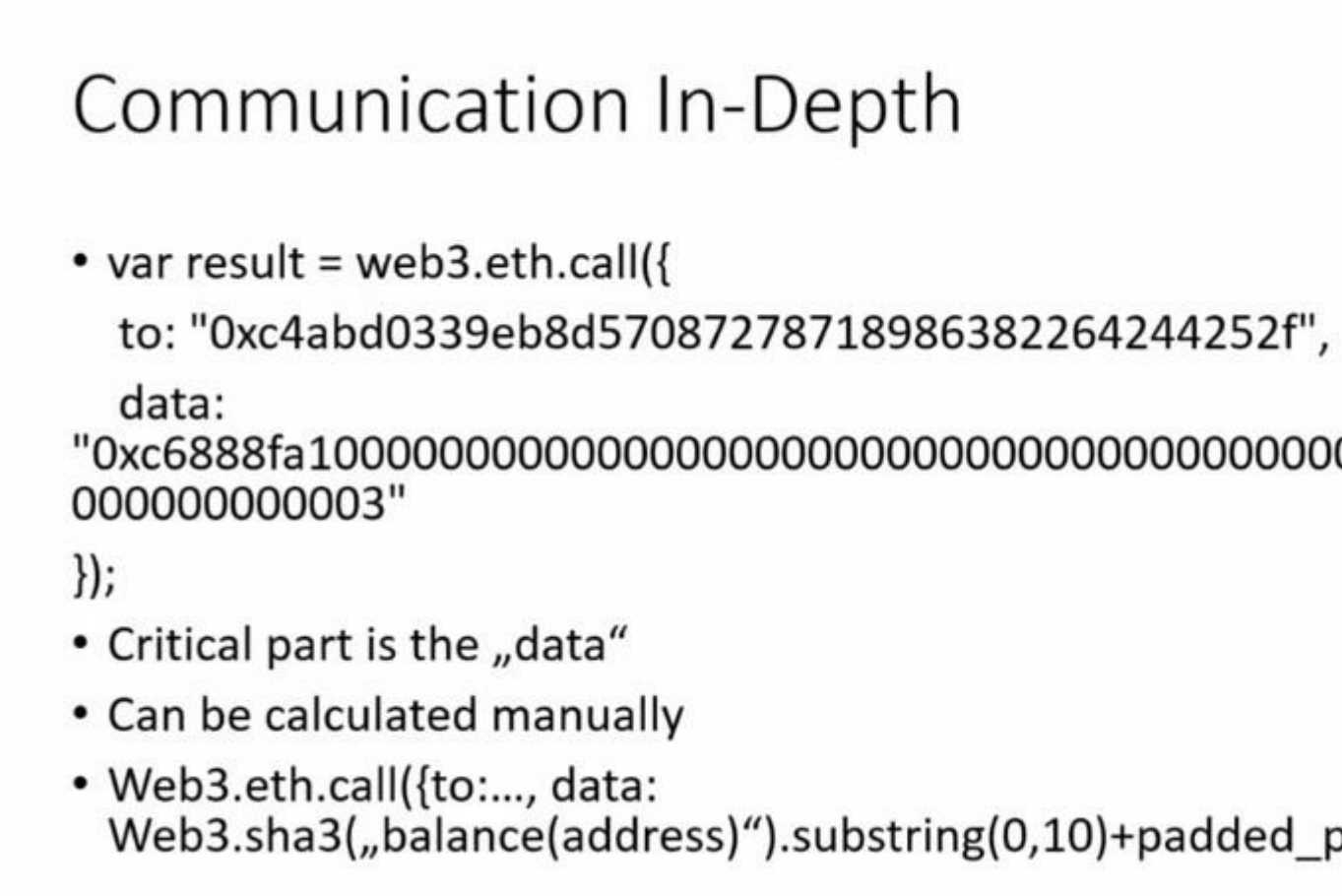
Давайте подробнее разберем двоичный *интерфейс* приложения, или *ABI*. *ABI* всегда является массивом *JSON*.

- Application Binary Interface

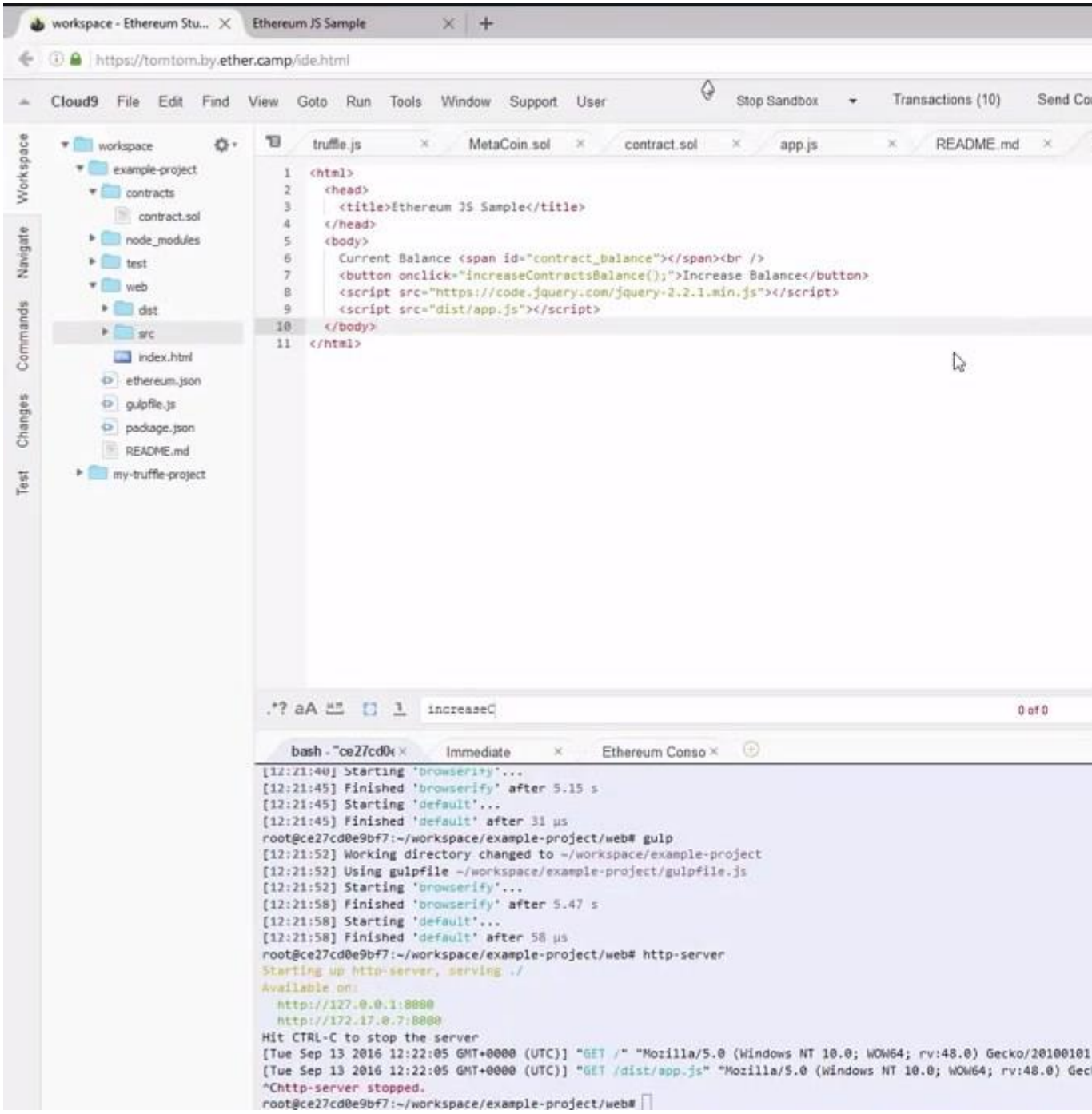
```
[
{ "name": "send", "const": false, "input": [ { "name": "to", "type": "address" }, { "name": "valueInmGAV", "type": "uint
{ "name": "balance", "const": true, "input": [ { "name": "who", "type": "address" } ], "output": [ { "name": "balanceInm
]
```

В нашем примере *массив* содержит два элемента. Имя первого элемента - это *send*, что сообщает web3, что название функции, которую он будет вызывать - это *send*. Эта *функция* не является постоянной и насчитывает два входных параметра. Один из них относится к адресному типу и называется to, а другой - числового типа, *uint256*. Каких-либо выходных данных эта *функция* не предоставляет. Другой функцией является balance, она уже является постоянной, поэтому ее не нужно обсчитывать в блокчейне. У этой функции один *входной параметр* адресного типа, и один выходной - типа *uint256*. Этот двоичный *интерфейс* приложения сообщает **web3.js**, как взаимодействовать с двоичным кодом в блокчейне.

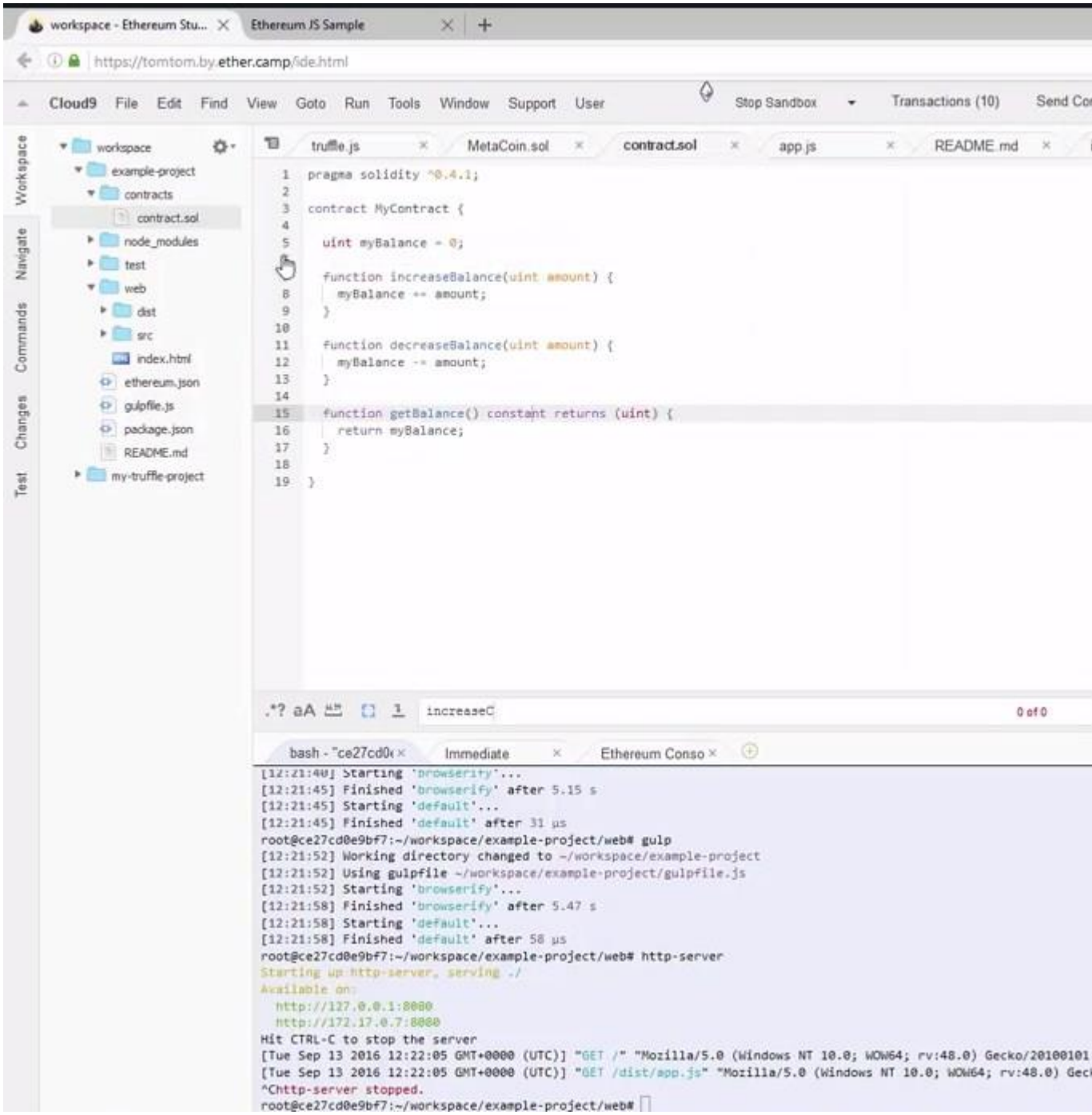
Теперь разберемся, как работает коммуникация.



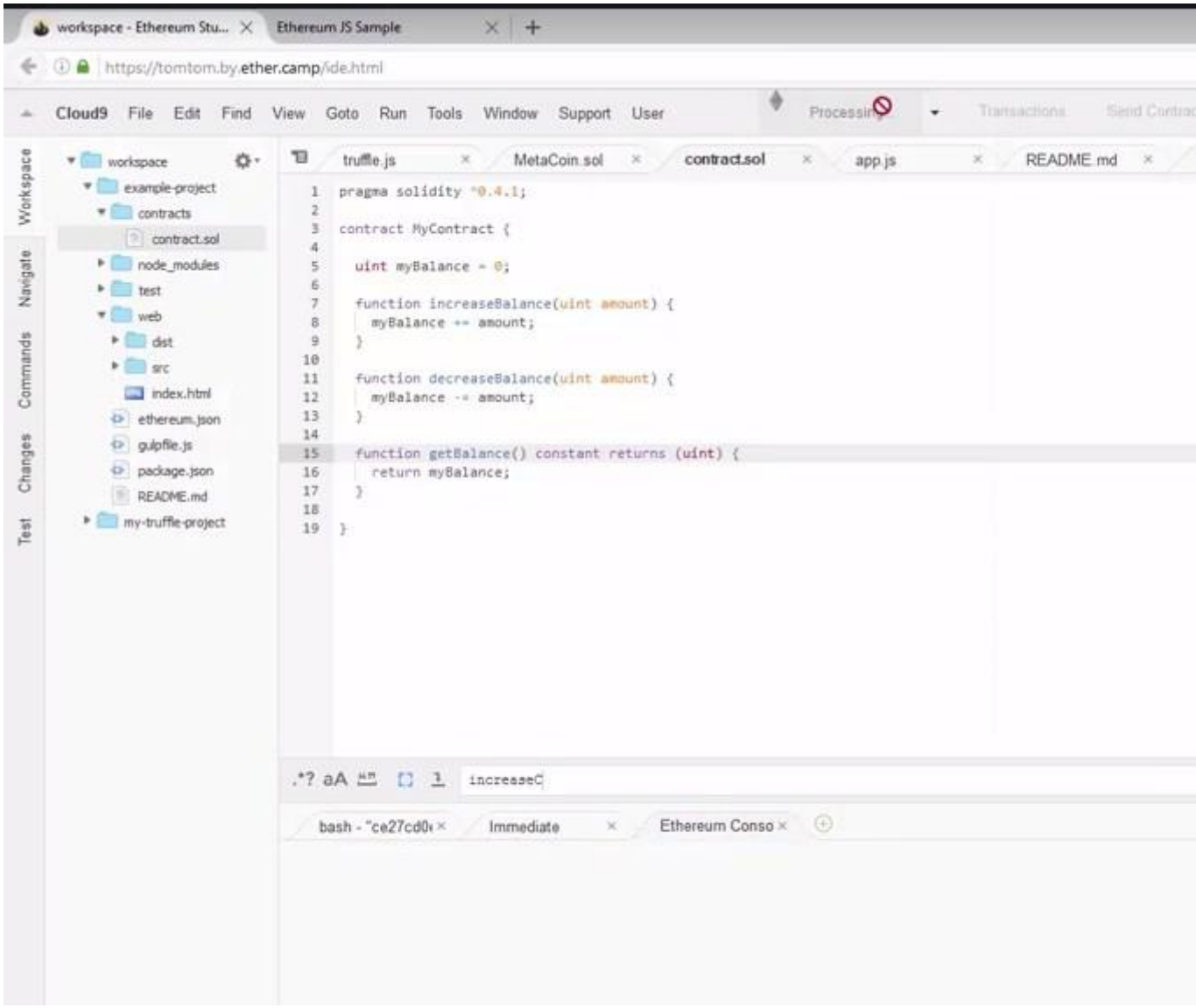
За вызовами процедур стоит *функция web3.eth.call*. Она работает с объектом, который характеризуется адресом *to* и полем с данными *data*. Первые двадцать *байт* поля *data* предназначены для функции вызова *call*. Все остальные - это так называемый дополнительный *параметр*, обозначенный как *padded*. Его можно рассчитать вручную. Например, для объекта *web3.sha3* вы сообщаете название функции, которую нужно вызвать. Придется обрезать имя до первых десяти символов, которые займут двадцать *байт*, и добавить нужные параметры в раздел *padded*. Узнать больше об этом можно в документации Solidity.
С этого момента будем работать исключительно с контрактами. Рассмотрим пример.



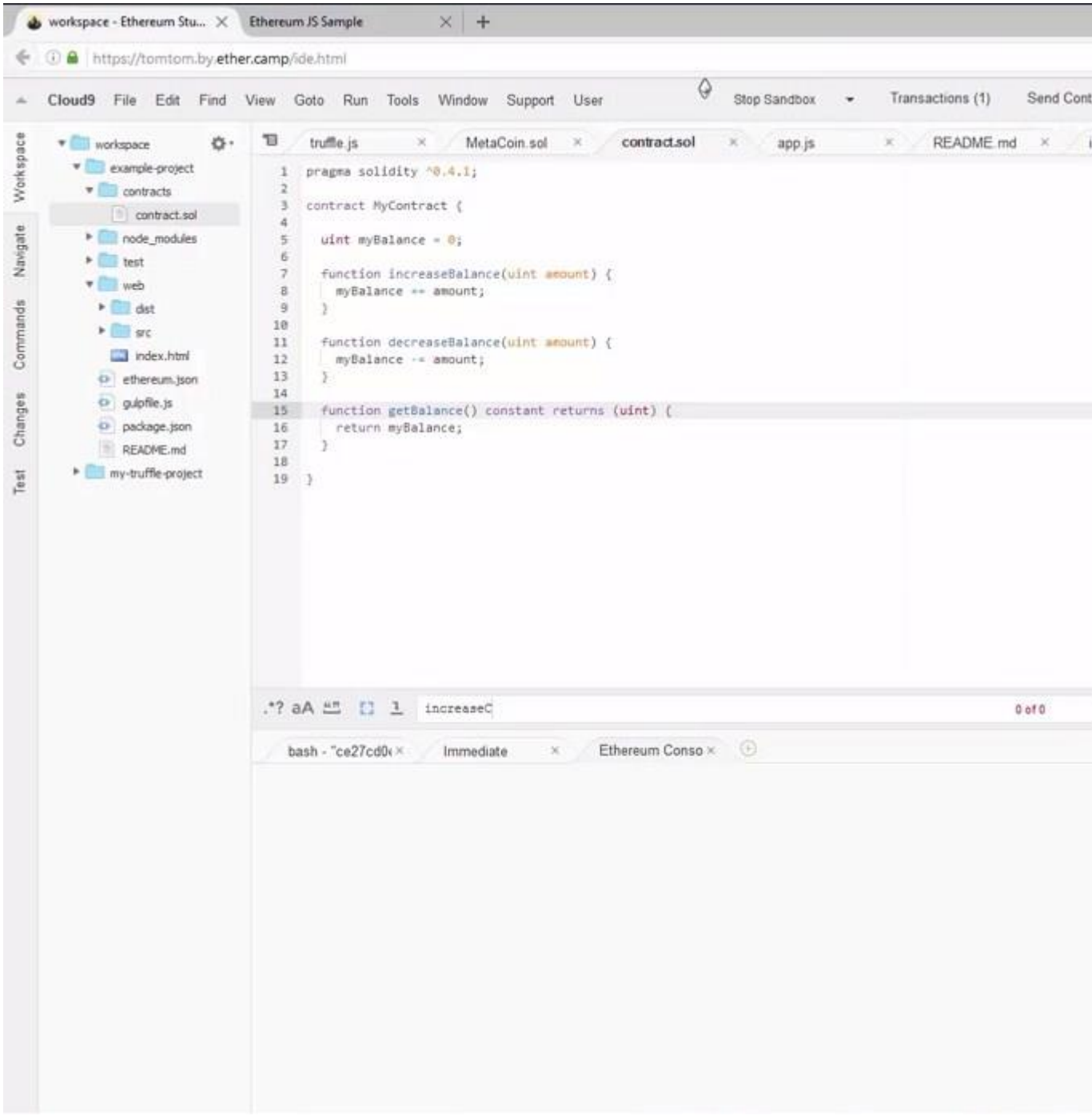
В прошлой части мы создали простой контракт с тремя функциями: *getBalance, increaseBalance* и *decreaseBalance*. В данном примере сфокусируемся на *getBalance* и *increaseBalance*, и вызовем их с веб-сайта. Для демонстрации будем использовать Ethereum Studio.
Давайте посмотрим, из чего состоит контракт.



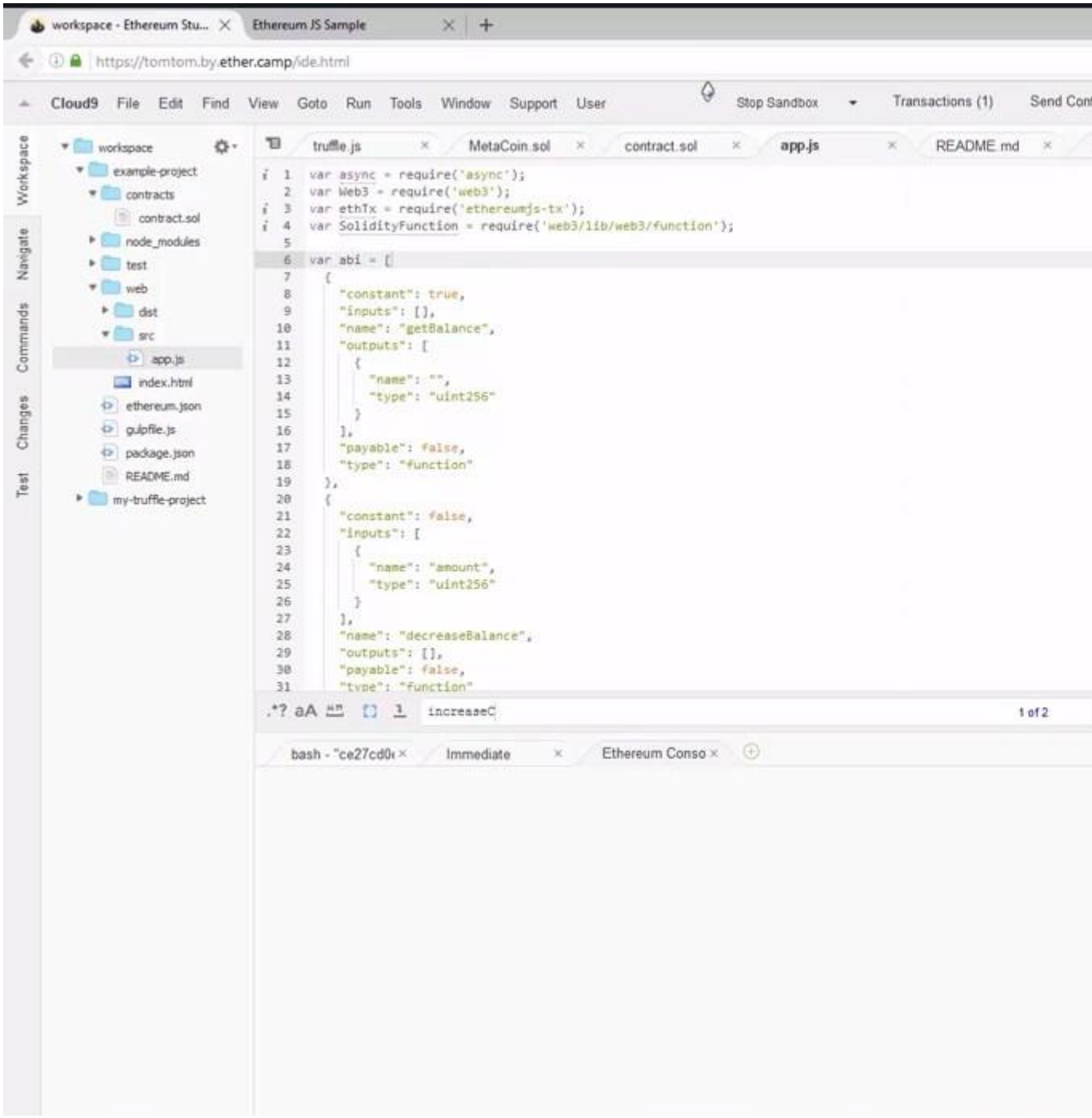
Здесь есть *переменная* balance типа *uint256*, изначально равная нулю. Есть две функции для изменения баланса, *increaseBalance* и *decreaseBalance*, и одна постоянная *функция getBalance* для возврата величины баланса. Перезапустим режим песочницы для демонстрации работы контракта.



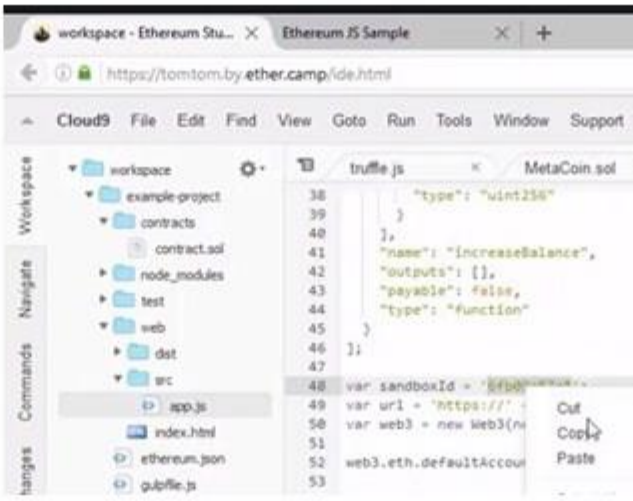
Этот *запуск* режима песочницы получил свой *идентификатор* и разместила контракт *по* следующему адресу.



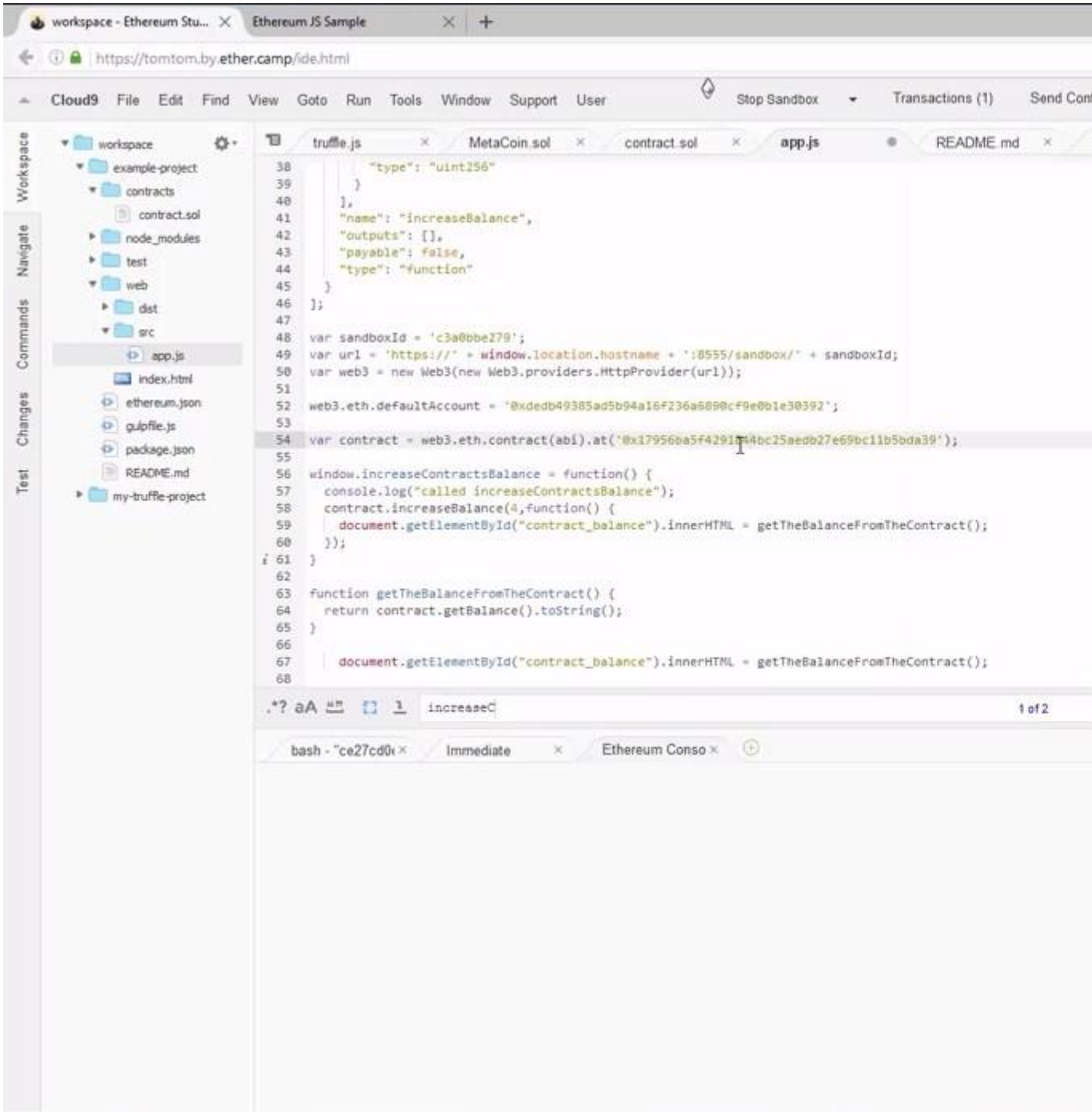
Перейдем к каталогу *web*. В нем хранится другой каталог *src* и далее *файл app.js*, в котором расположен *массив ABI*.



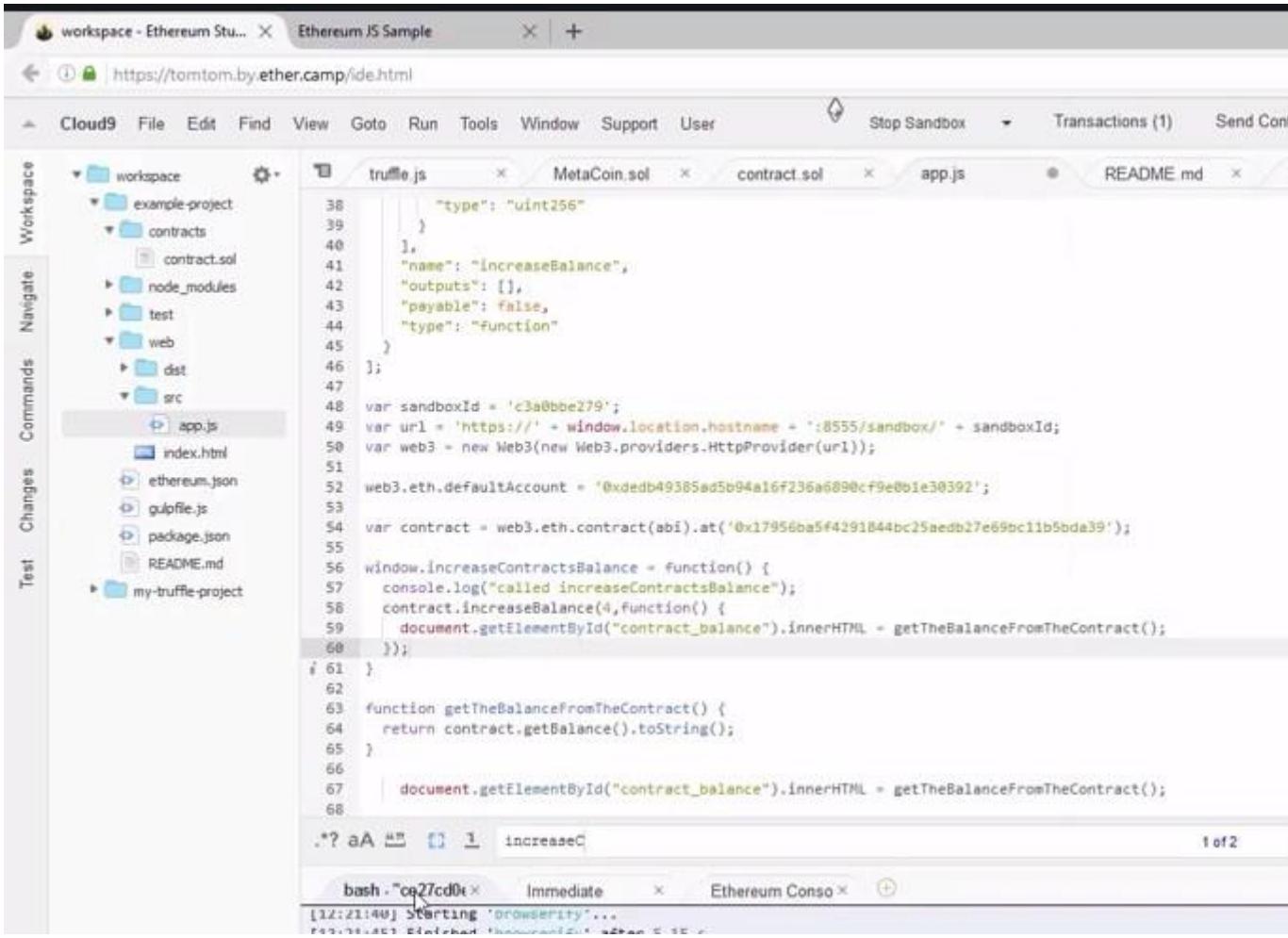
Скопируем этот *массив* из контракта. Необходимо задать *адрес* песочницы.



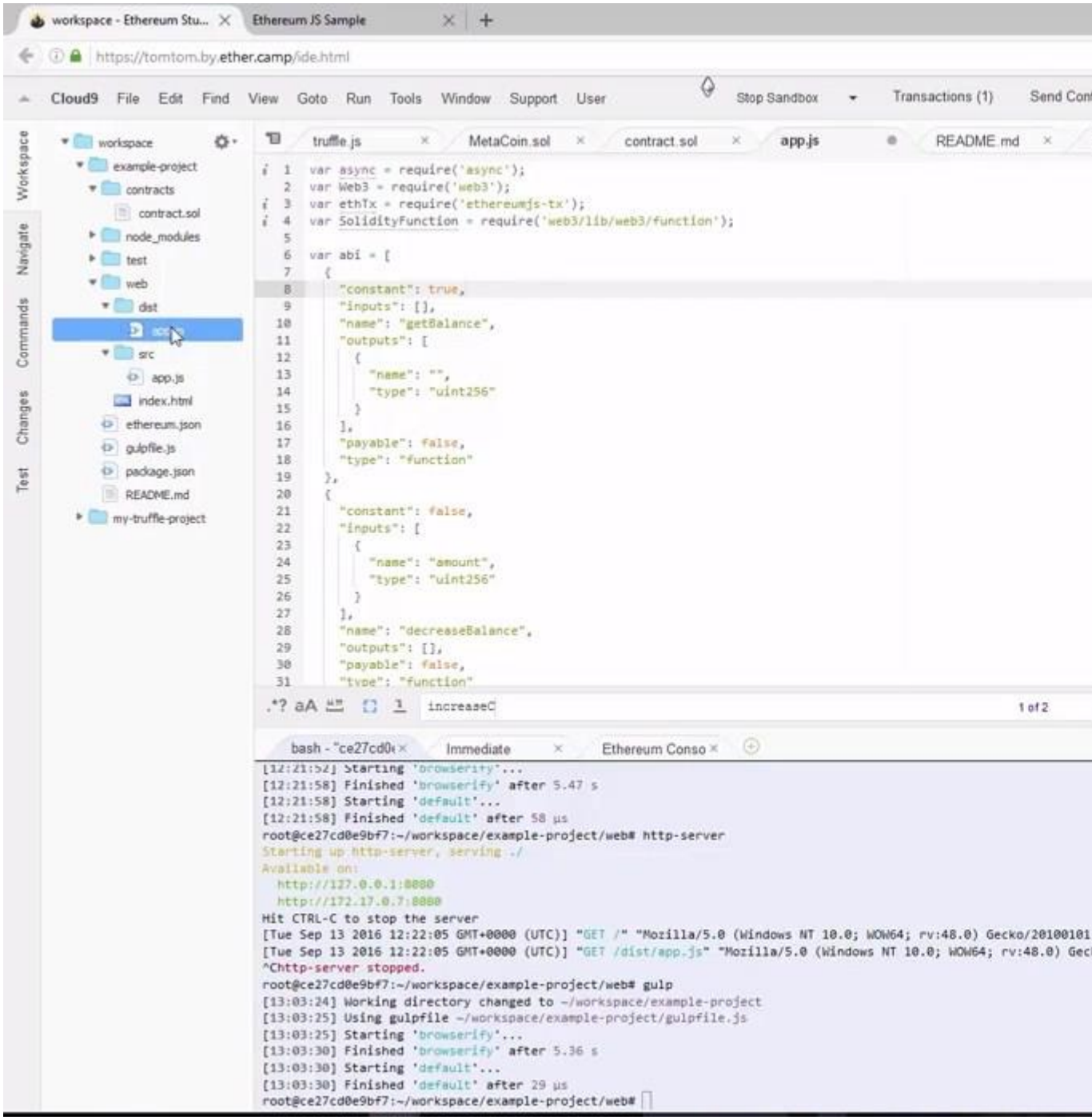
После запуска кода будет создан *объект HttpProvider*. Эта операция выполняется вне виртуальной машины, и созданный *объект* доступен для других пользователей сети Ethereum.
Система выдала мне учетную *запись*. Теперь контракт размещается в сети *по* этому адресу, и с ним можно взаимодействовать разными способами: например, узнать баланс с помощью процедуры *contract.getBalance*, а затем преобразовать полученное большое число в строку с помощью расширения *toString*. Или можно увеличить баланс с помощью *contract.increaseBalance*, конечно, указав объем пополнения.



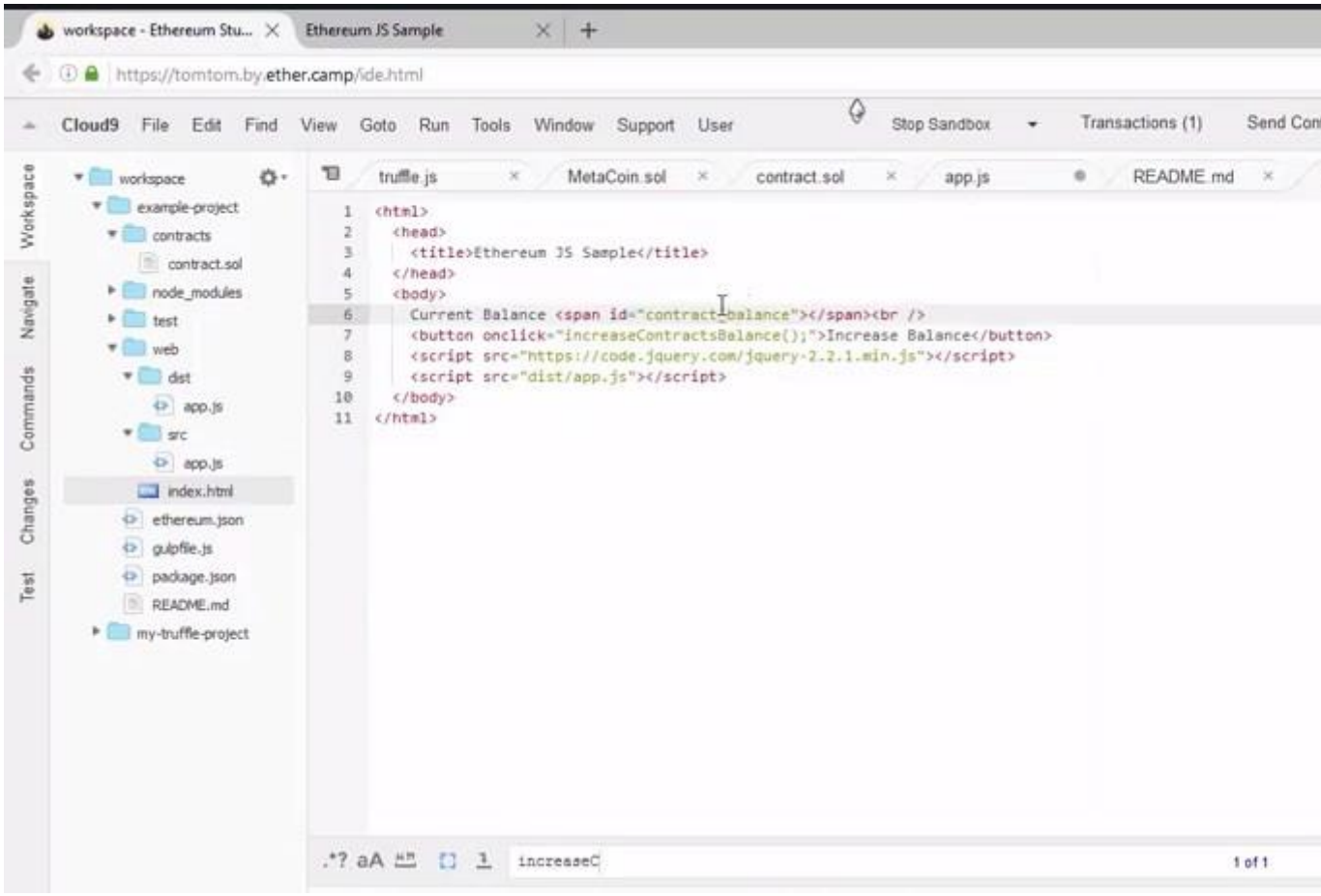
Поскольку эта *функция* не выполняется в реальном времени, придется подождать, пока *транзакция* не будет обсчитана сетью. Можно также вставить *дополнительный код* JavaScript, который будет выполнен вместе с обсчетом транзакции. Отправимся в *консоль*.



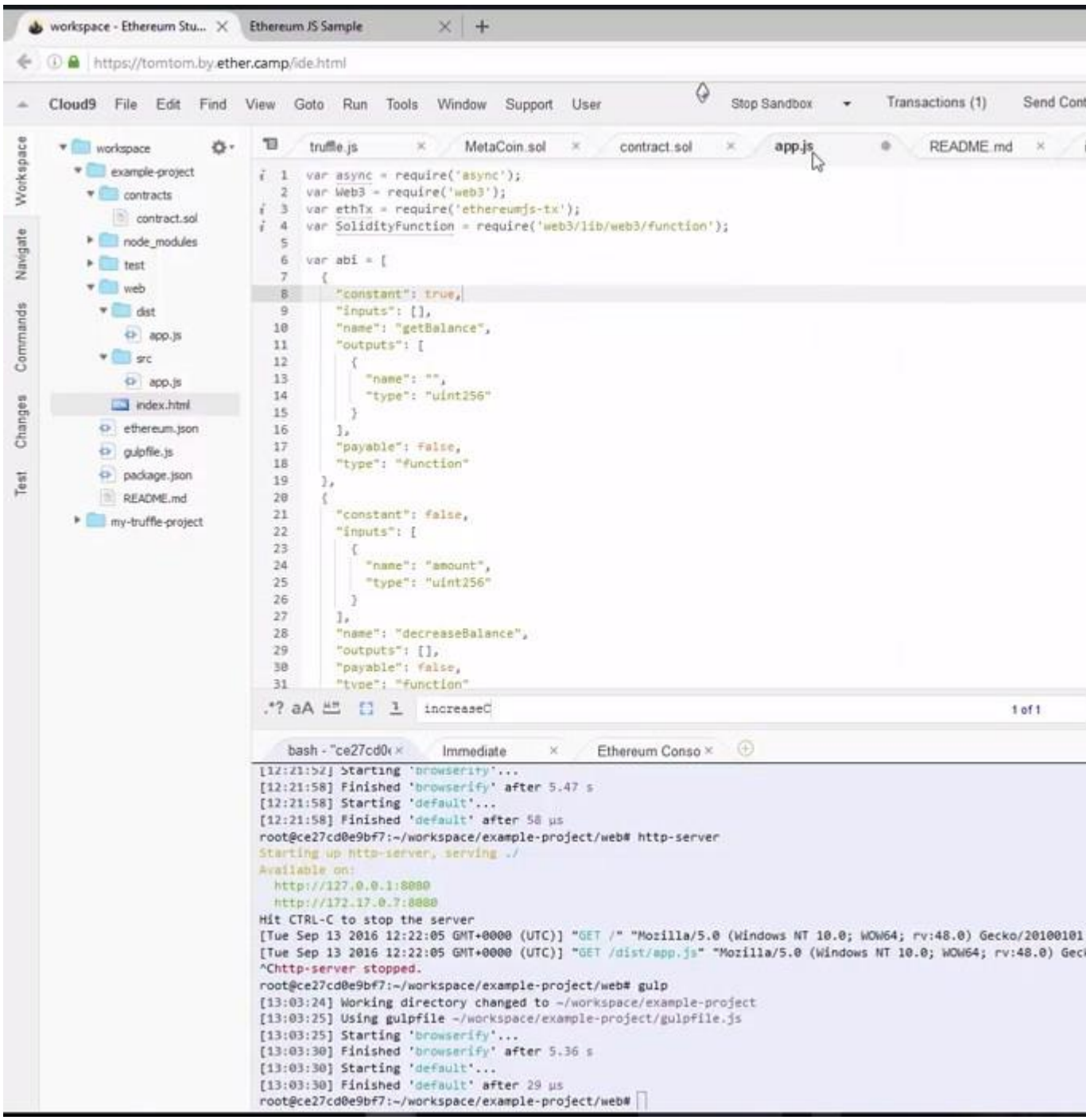
Перейдем в каталог *web* и подадим команду *gulp* для сборки реального приложения. Будет запущена процедура *browserify* и заданы следующие входные параметры. Результат сборки будет размещен в каталоге *dist*. Все необходимые библиотеки web3 и JavaScript тоже будут там.



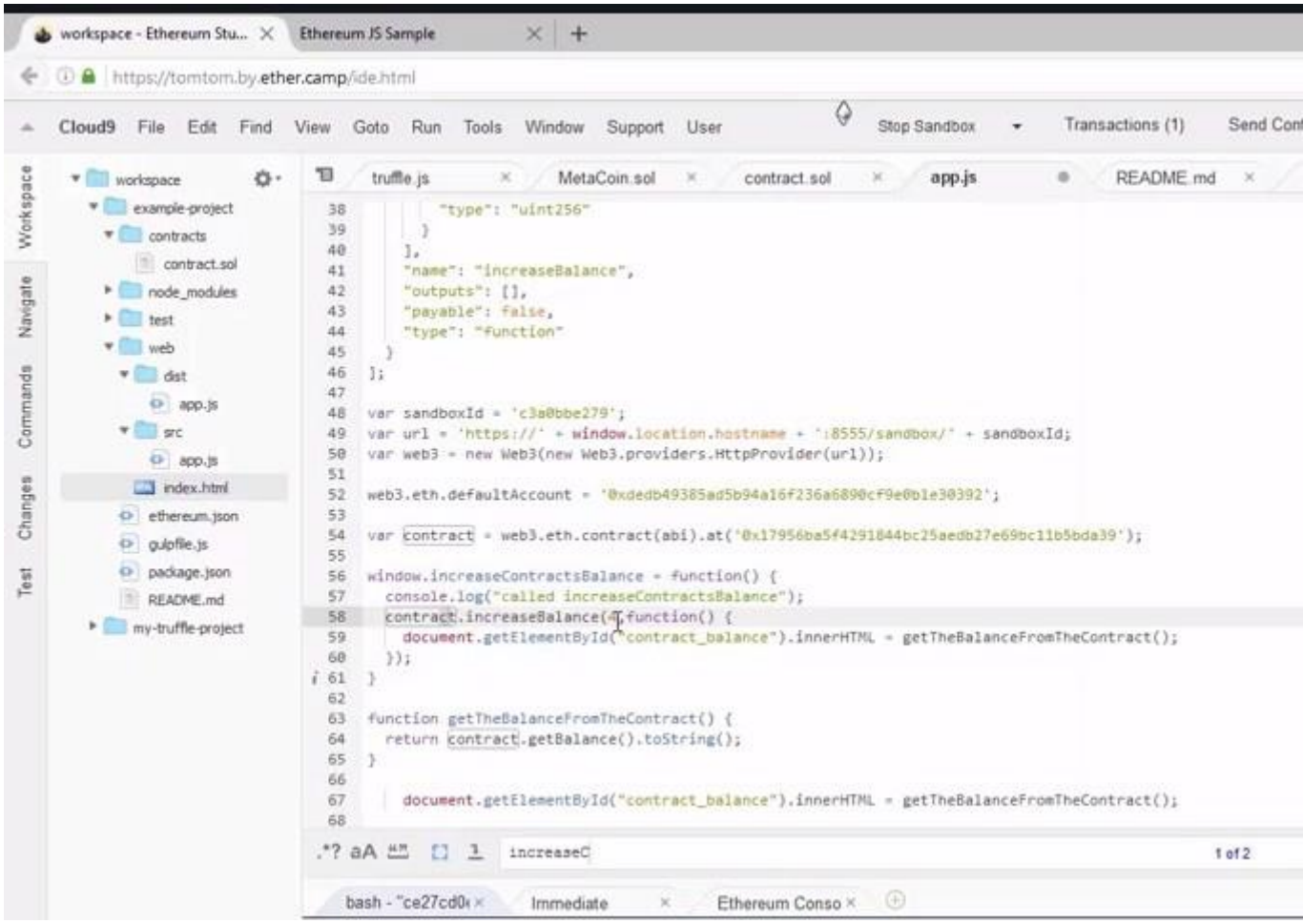
Файл index.html был модифицирован. Текущий баланс будет отображаться с помощью демонстрации значения переменной *contract_balance*.



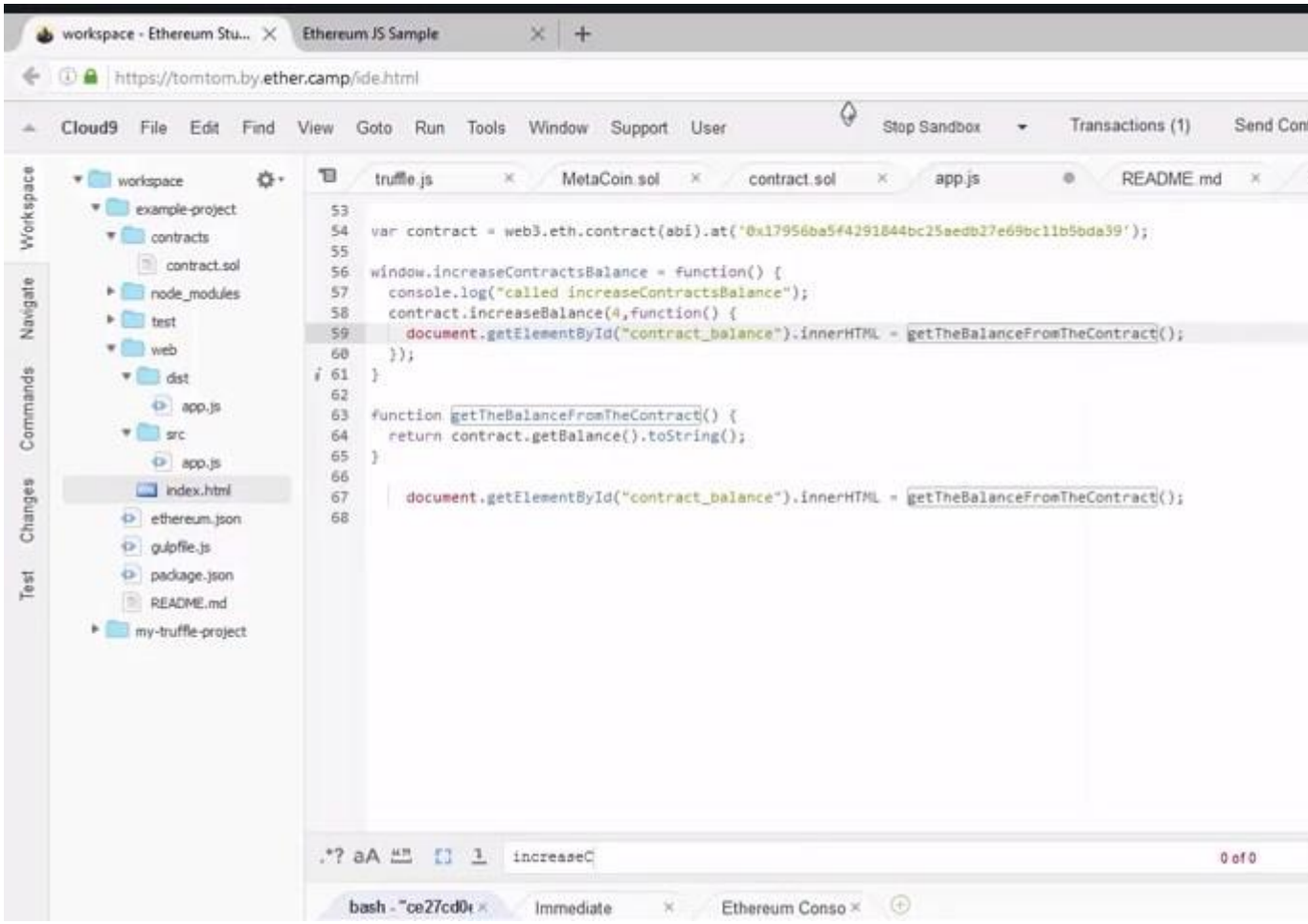
Также предусмотрена кнопка с событием при щелчке *по* ней - вызывается *функция increaseContractBalance*. Функциональность не очень высока, это все собрано в демонстрационных целях.



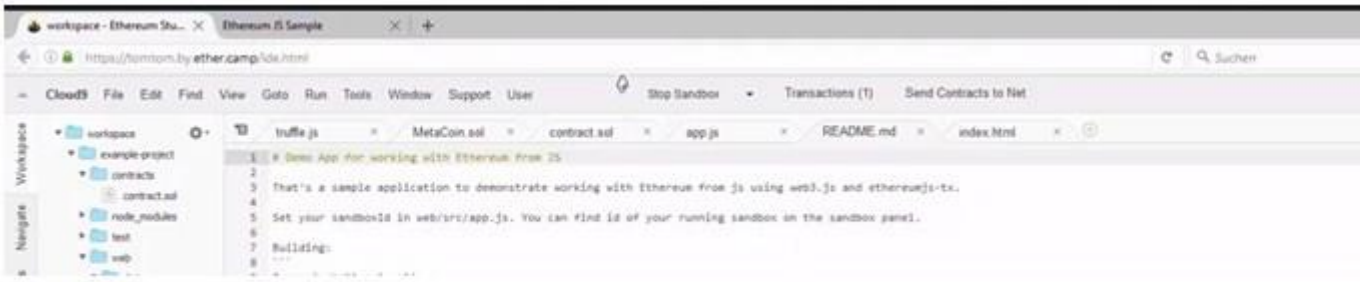
Я подключил к классу *window* функцию *increaseContractBalance*. Она подключается к консоли и вызывает функцию *contract.increaseBalance* с аргументом 4.



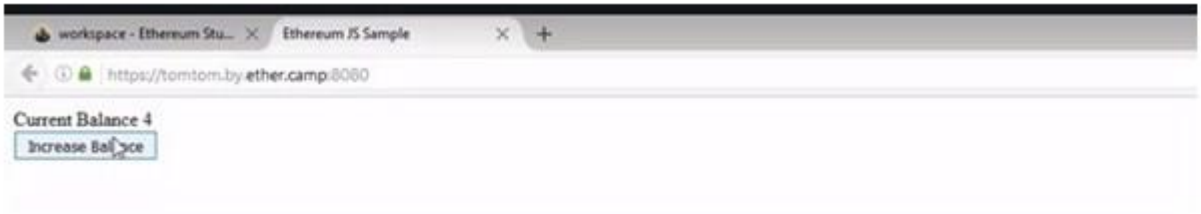
Затем она записывает в переменную *contract_balance* результат работы функции *getTheBalanceFromTheContract*. При первом запуске кода баланс контракта будет также отображен на веб-странице приложения.

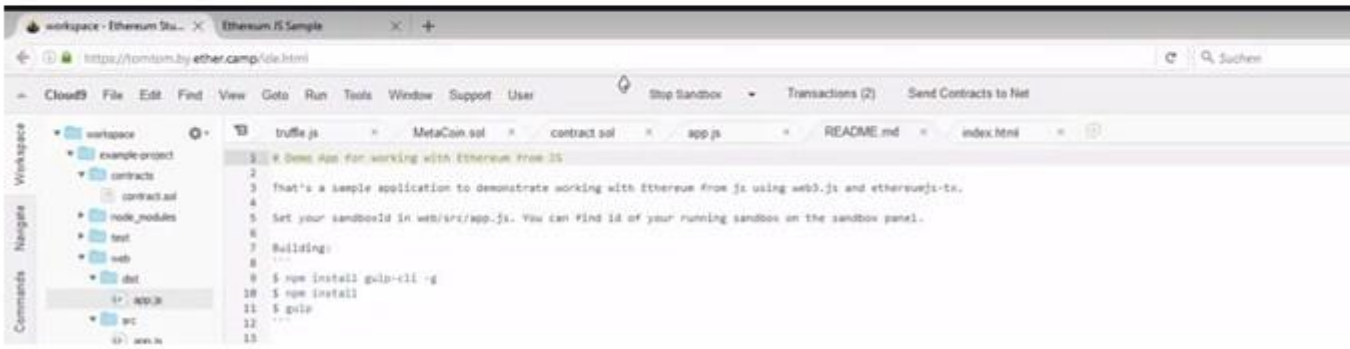


Теперь запустим *HTTP*-сервер. Это позволит получать *доступ* к нашему контракту из внешнего мира. Скопируем *адрес*, вставим его в новой вкладке и сменим *номер порта*. При запуске кода будет выведен баланс, в настоящий момент равный нулю, как и в нашей песочнице.



При щелчке *по* кнопке *Increase Balance* баланс будет увеличен, затем будет вызвана *функция getBalance*, и текст, отображающий его величину, будет изменен. Теперь баланс равен четырем.

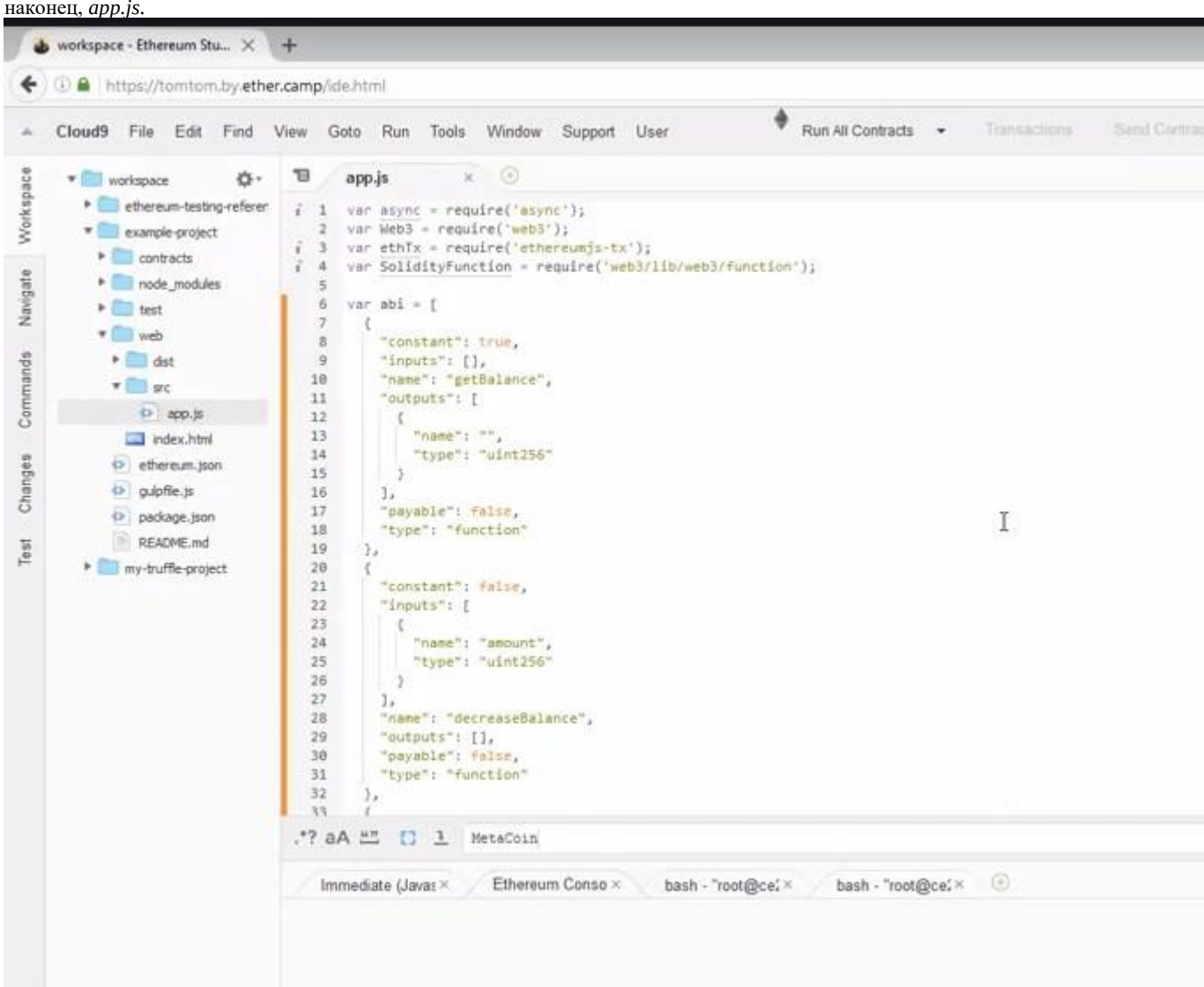




Использование библиотеки Web3 и взаимодействие с HTML-страницами

В этой части рассмотрим способы доступа к контрактам с веб-сайтов

Для начала обратимся к проекту. Откроем каталог с проектом и последовательно перейдем *по* каталогам *web*, затем *src* и, наконец, *app.js*.



В этом файле можно сразу увидеть заданный *массив ABI*.

app.js

```
var async = require('async');
var Web3 = require('web3');
var ethTx = require('ethereumjs-tx');
var SolidityFunction = require('web3/lib/web3/function');

var abi = [
  {
    "constant": true,
    "inputs": [],
    "name": "getBalance",
    "outputs": [
      {
        "name": "",
        "type": "uint256"
      }
    ],
    "payable": false,
    "type": "function"
  },
  {
    "constant": false,
    "inputs": [
      {
        "name": "amount",
        "type": "uint256"
      }
    ],
    "name": "decreaseBalance",
    "outputs": [],
    "payable": false,
    "type": "function"
  },
```

MetaCoin

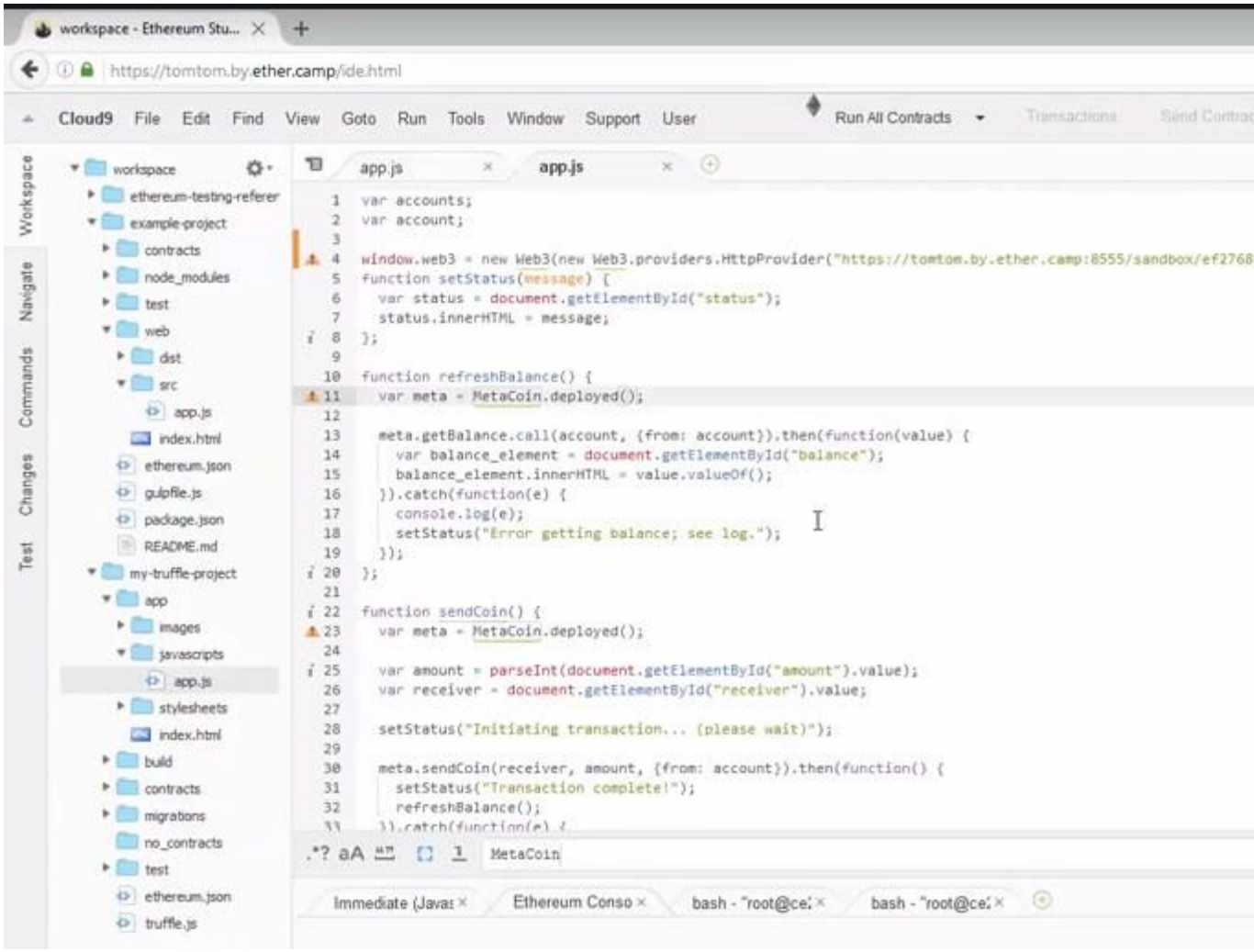
Затем задается узел *HTTP* для доступа к интерфейсу *RPC*. И, наконец, сам контракт.

```
var sandboxId = 'c3a0bbe279';
var url = 'https://' + window.location.hostname + ':8555/sandbox/' + sandboxId;
var web3 = new Web3(new Web3.providers.HttpProvider(url));

web3.eth.defaultAccount = '0xdedb49385ad5b94a16f236a6890cf9e0b1e30392';

var contract = web3.eth.contract(abi).at('0x17956ba5f4291844bc25aedb27e69bc11b5bda39');

window.increaseContractsBalance = function() {
  console.log("called increaseContractsBalance");
  contract.increaseBalance(4,function() {
    document.getElementById("contract_balance").innerHTML = getTheBalanceFromTheContract();
  });
}

function getTheBalanceFromTheContract() {
```

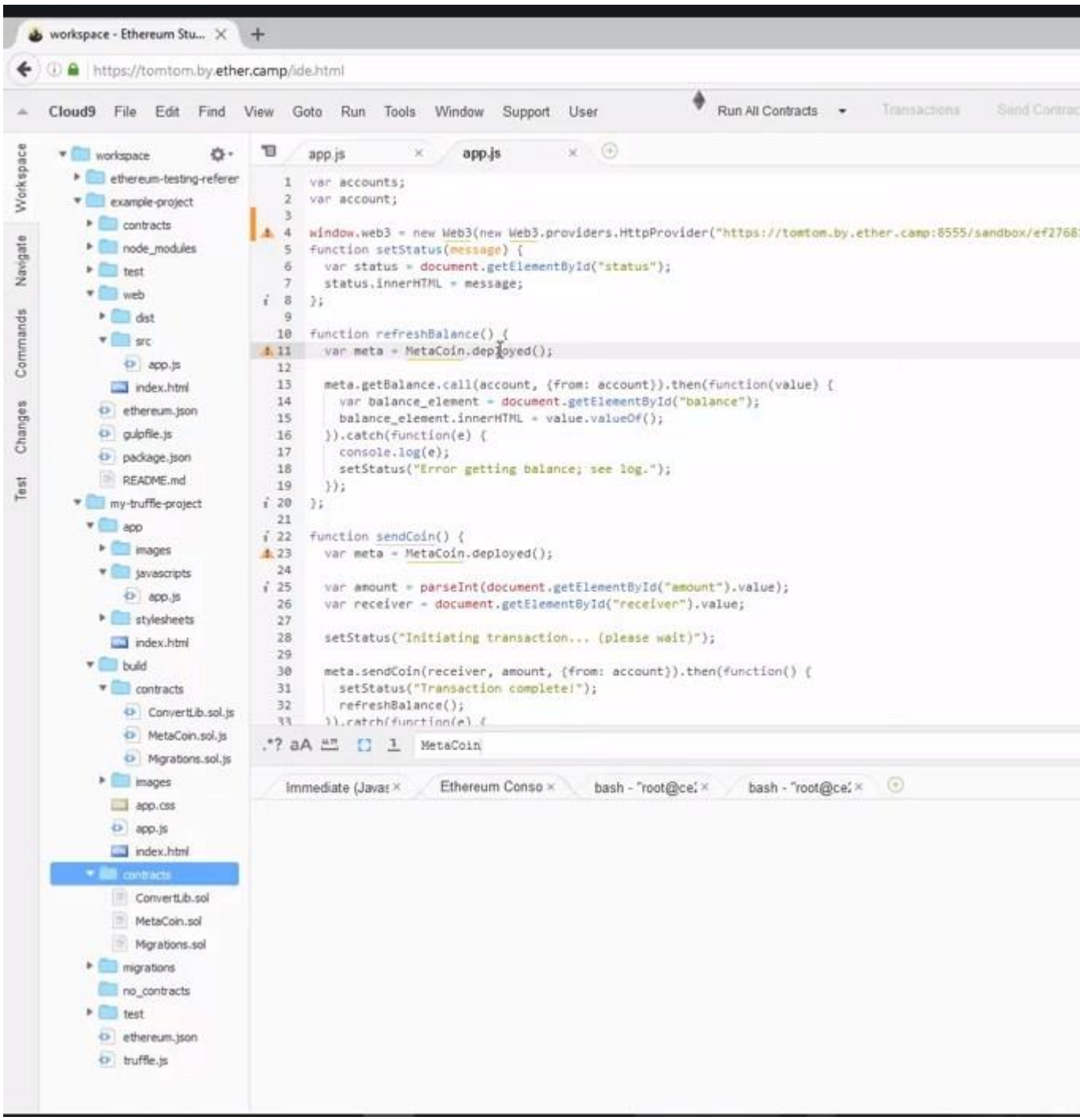
Посмотрим, как строится работа. Первым делом нужно получить *доступ* к объекту Web3. В среде Web3 для этого необходимо вызвать процедуру *web3.eth.contract* и задать ей *массив ABI*, затем указать *адрес* контракта. После выполнения этих операций можно вызывать функции напрямую, например, *contract.increaseBalance* - одну из функций из массива *ABI*.

Если *функция* не является постоянной, то для отражения изменений в блокчейне на ее обсчет потребуется выделить некоторое количество газа. Например, для увеличения баланса контракту необходимо изменить состояние блокчейна, что потребует затрат газа. В режиме песочницы дополнительных действий не требуется, но если запускать контракты из кошелька Mist, то потребуется ввести *пароль* от учетной записи. Вызов постоянных функций, например, *getBalance* в нашем примере, затрат газа не требует.

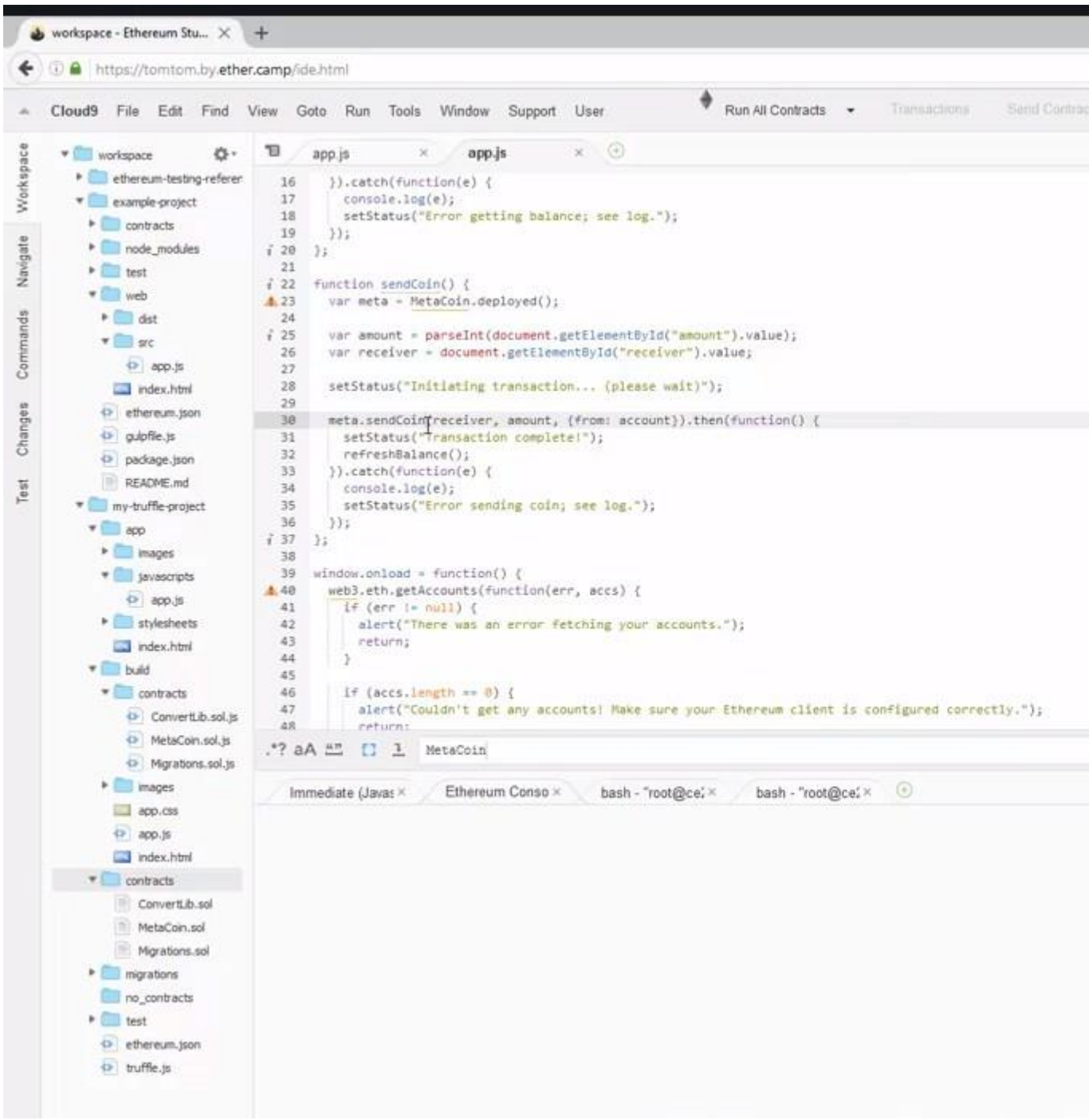
Давайте перейдем к проекту в среде Truffle. Последовательно пройдем *по* каталогам *app* и *javascripts* и откроем *файл app.js*. *По* умолчанию, Truffle подключает *объект* Web3 к узлу *HTTP*, заданному в файле *truffle.js*.



Рассмотрим функцию *refreshBalance*. Видно, что она предусматривает *доступ* к контракту *MetaCoin* посредством команды *MetaCoin.deployed*. Если в каталоге Contracts размещены контракты, то при вызове команды *truffle build* Truffle автоматически создаст соответствующие файлы JavaScript для всех контрактов. При вызове *truffle migrate* Truffle попытается разместить контракты в блокчейне и записать адреса контрактов в соответствующие файлы JavaScript, что избавит от необходимости задавать им адреса, то есть для вызова MetaCoin потребуется только вызвать *MetaCoin.deployed*. Примерно то же самое можно делать и в других проектах с небольшим отличием. *Функция Meta.getbalance.call* вызывает функцию внутри среды Truffle, поскольку для ее работы не требуется менять данные в блокчейне. Поэтому эта *функция* является постоянной.



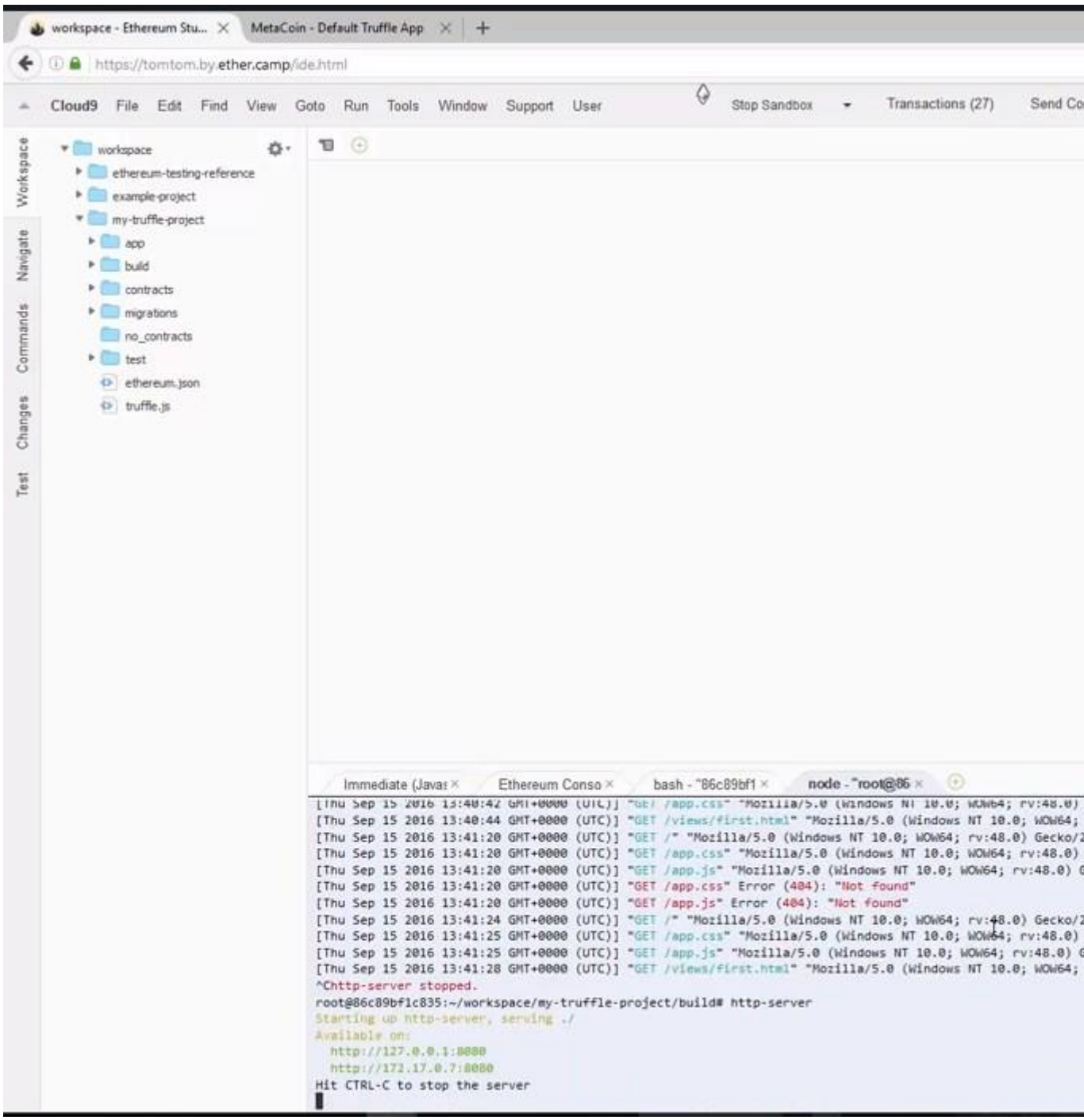
Рассмотрим функцию *sendCoin*. Она тоже ссылается на контракт *MetaCoin*. Однако затем можно напрямую вызывать *meta.sendCoin* без необходимости использования функции *call*. Выполнение этой *операции* требует внесения изменений в блокчейн, поэтому для их обсчета потребуется потратить газ.



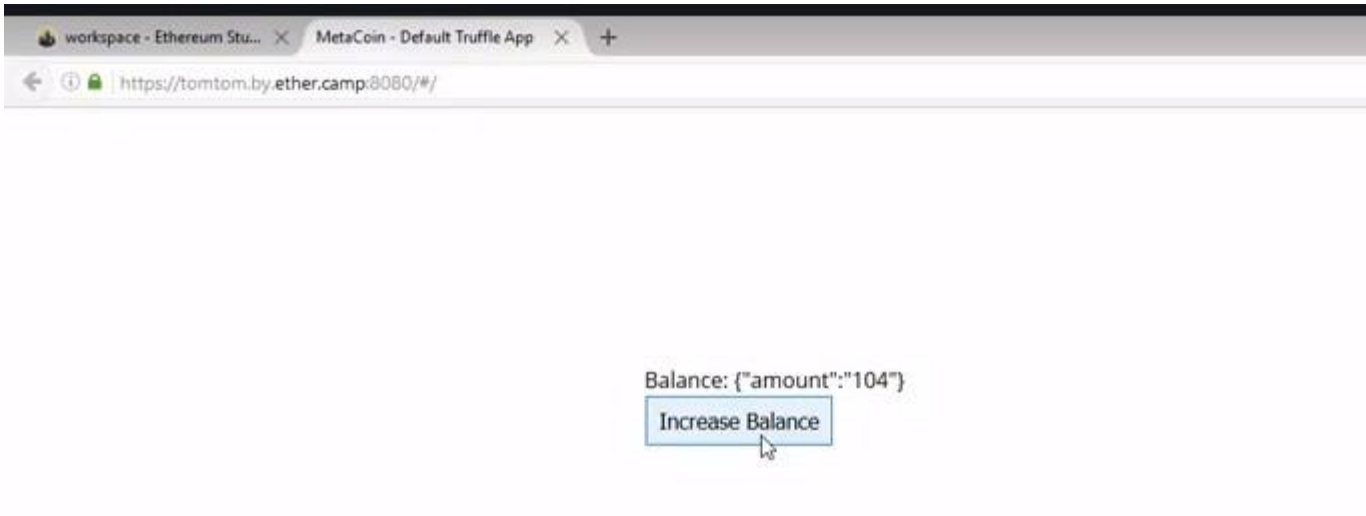
Интеграция библиотек Web3 и Angular.js

В этой части рассмотрим способ интеграции библиотеки Angular в среду Truffle, а также работу с Browserify, стандартным проектом для Ethereum Studio. Подготовленный проект не будет обладать большой функциональностью, рассмотрим его исключительно в учебных целях.

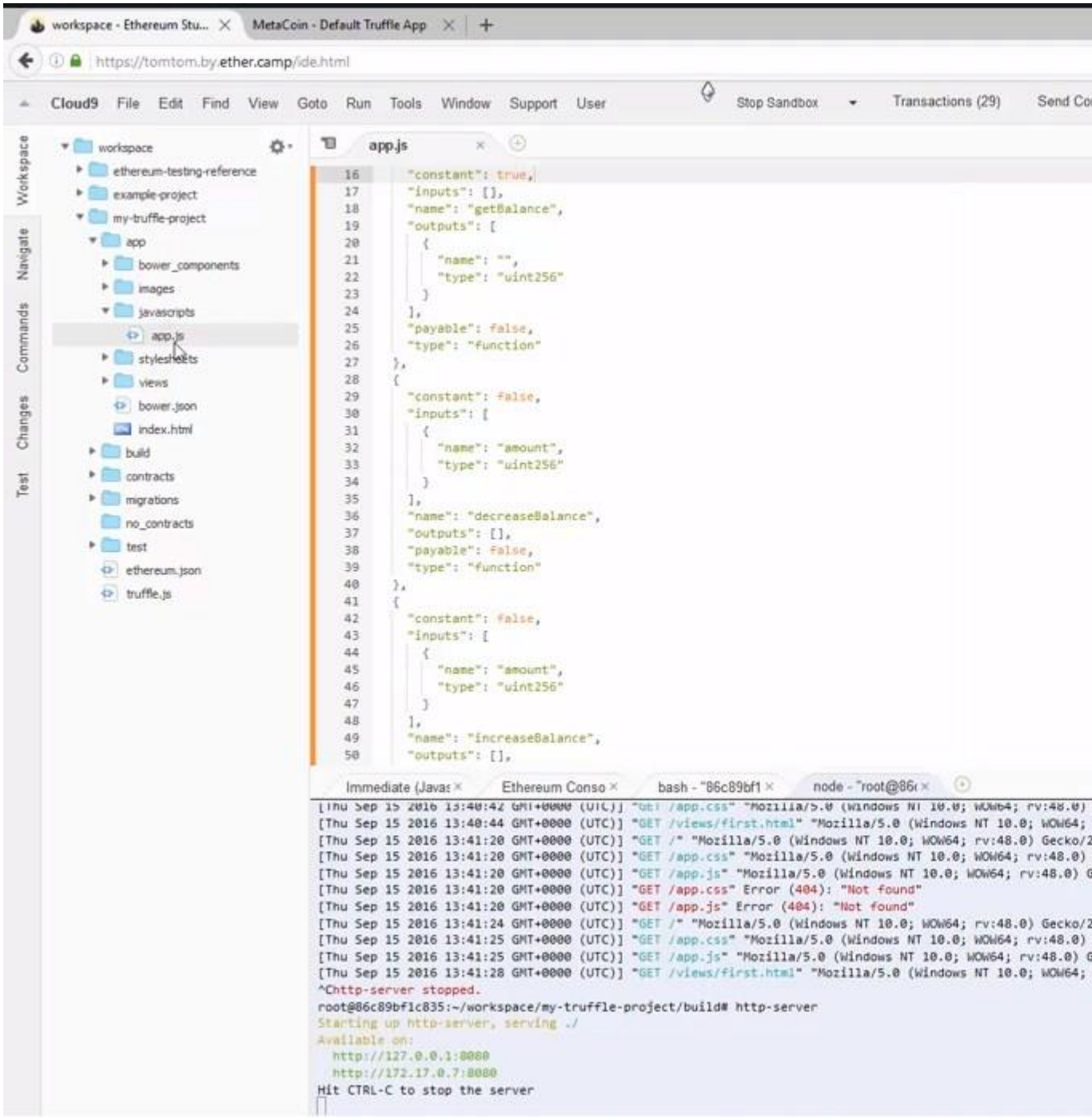
Начнем работу в проекте Truffle с запуска *HTTP*-сервера.



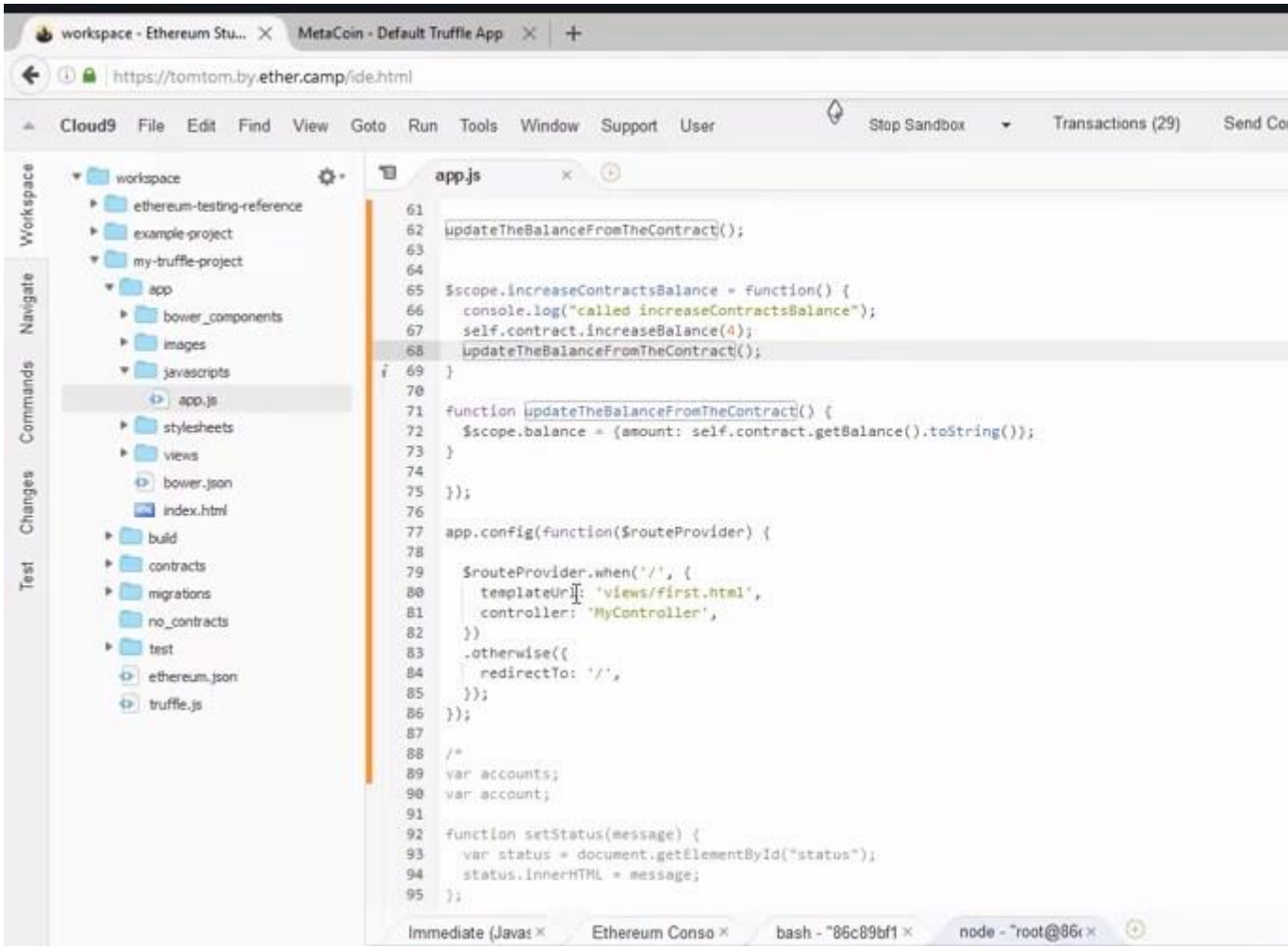
Перейдем на созданный *домен*, используя *порт* 8080. Видно, что у веб-сайта есть *доступ* к блокчейну. Видим величину баланса, и можем увеличить баланс.



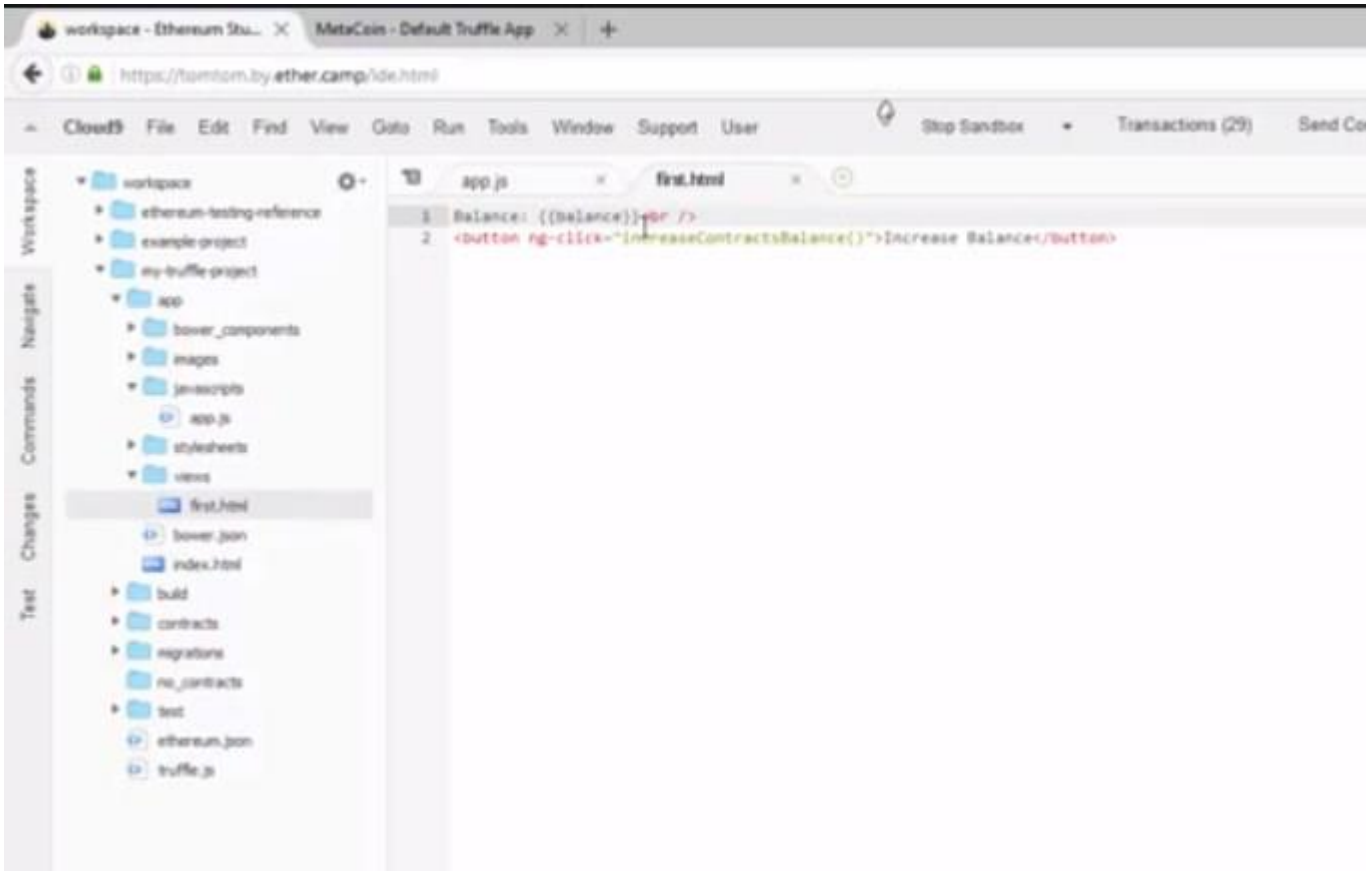
Если вы уже знакомы с библиотекой Angular, то изложение материала будет понятным, а если еще не знакомы с Angular, то настоятельно рекомендуется изучить обучающие материалы *no* Angular1. В проекте Truffle есть каталог *app* с каталогом *javascripts* и файлом *app.js*. Весь код будет размещен в файле *app.js*, конечно, его можно разнести *no* нескольким файлам. Со времени работы в режиме песочницы в данном коде осталась прежняя *конфигурация*, которая пригодится сейчас.



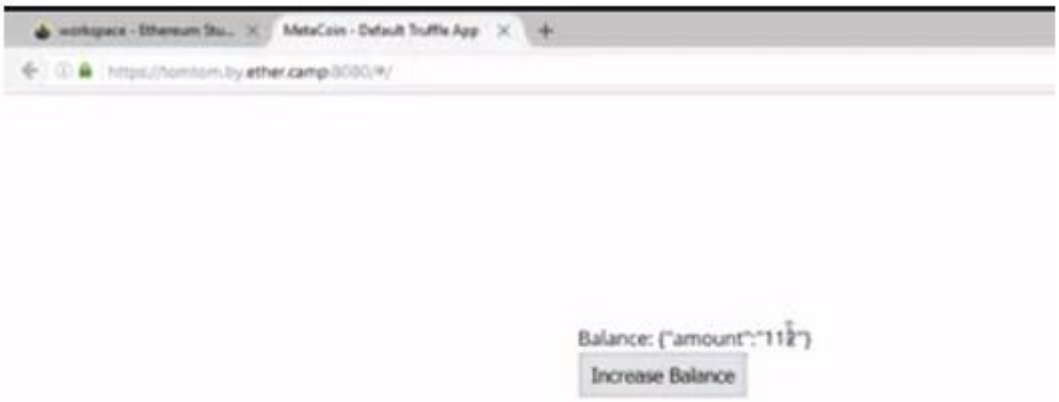
Объявим использование Angular и создадим *контроллер MyController*, внутри которого определим контракт. Есть также несколько функций. Первая называется *increaseContractsBalance*, а вторая - *updateTheBalanceFromTheContract*. В функции *increaseContractsBalance* мы выполняем *запись* в журнал регистрации событий *console.log*, увеличиваем баланс на четыре, и обновляем баланс контракта.

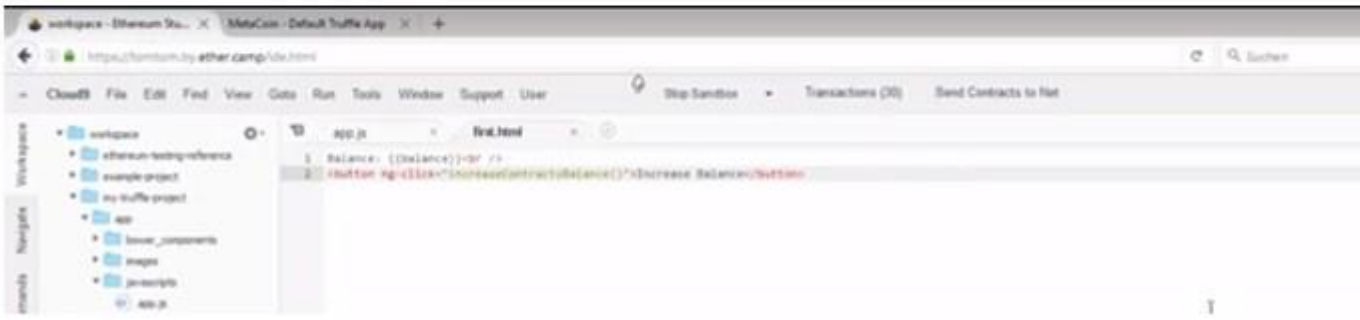


Кроме того, здесь задан *путь* к веб-серверу, а именно одно *отображение* и одна *веб-страница first.html*, а также определяется *контроллер* - в данном случае это *MyController*.



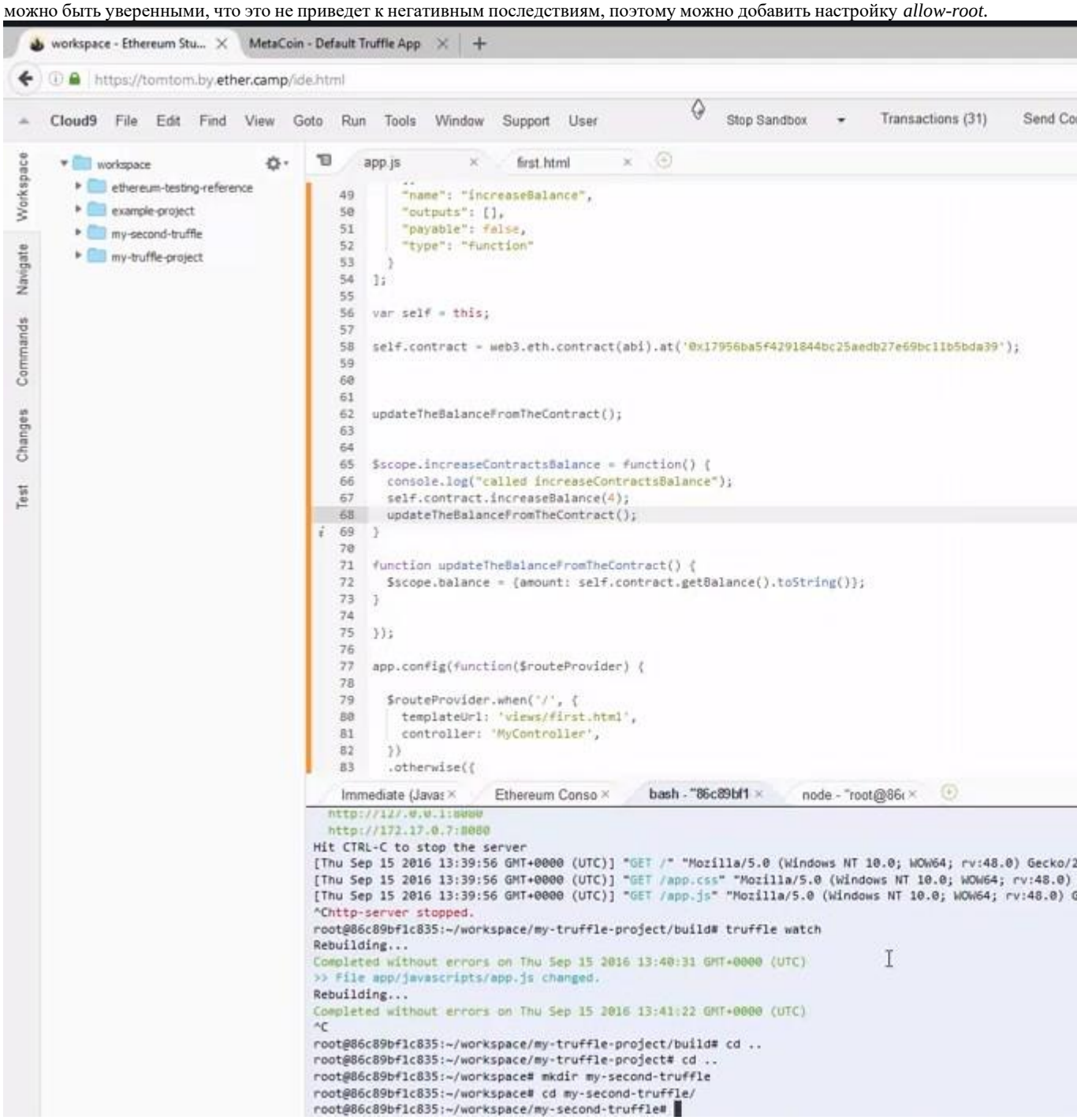
Есть текстовое *поле* для отображения баланса, для работы которого используется *объект balance*, а также кнопка, реагирующая на событие *onClick* и выполняющая вызов из контроллера процедуры *increaseContractsBalance*. Как видно, баланс здесь - это баланс из контракта.



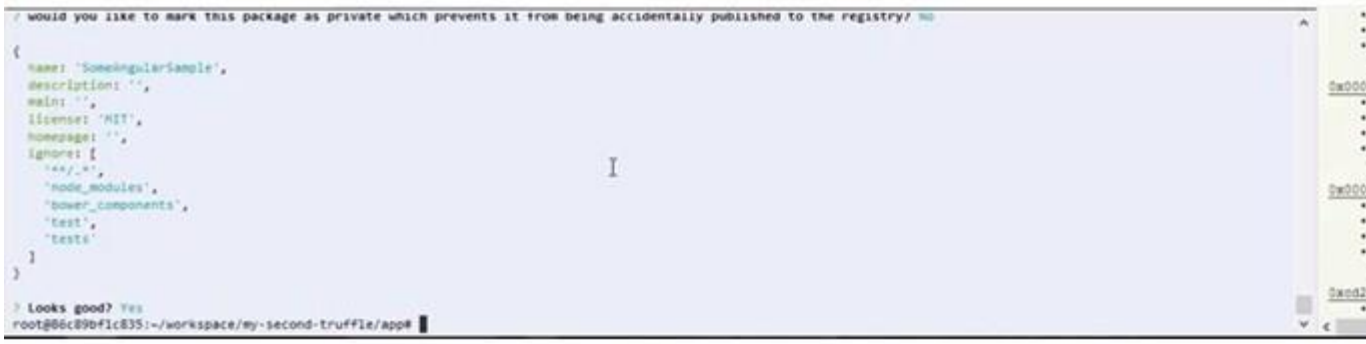


Любопытно отметить, что при обращении к блокчейну, когда *функция increaseContractsBalance* увеличивает баланс на четыре, эти процедуры не работают параллельно. Сначала будет выполнен обсчет функции увеличения баланса в блокчейне, и лишь затем - процедура *updateTheBalanceFromTheContract*.

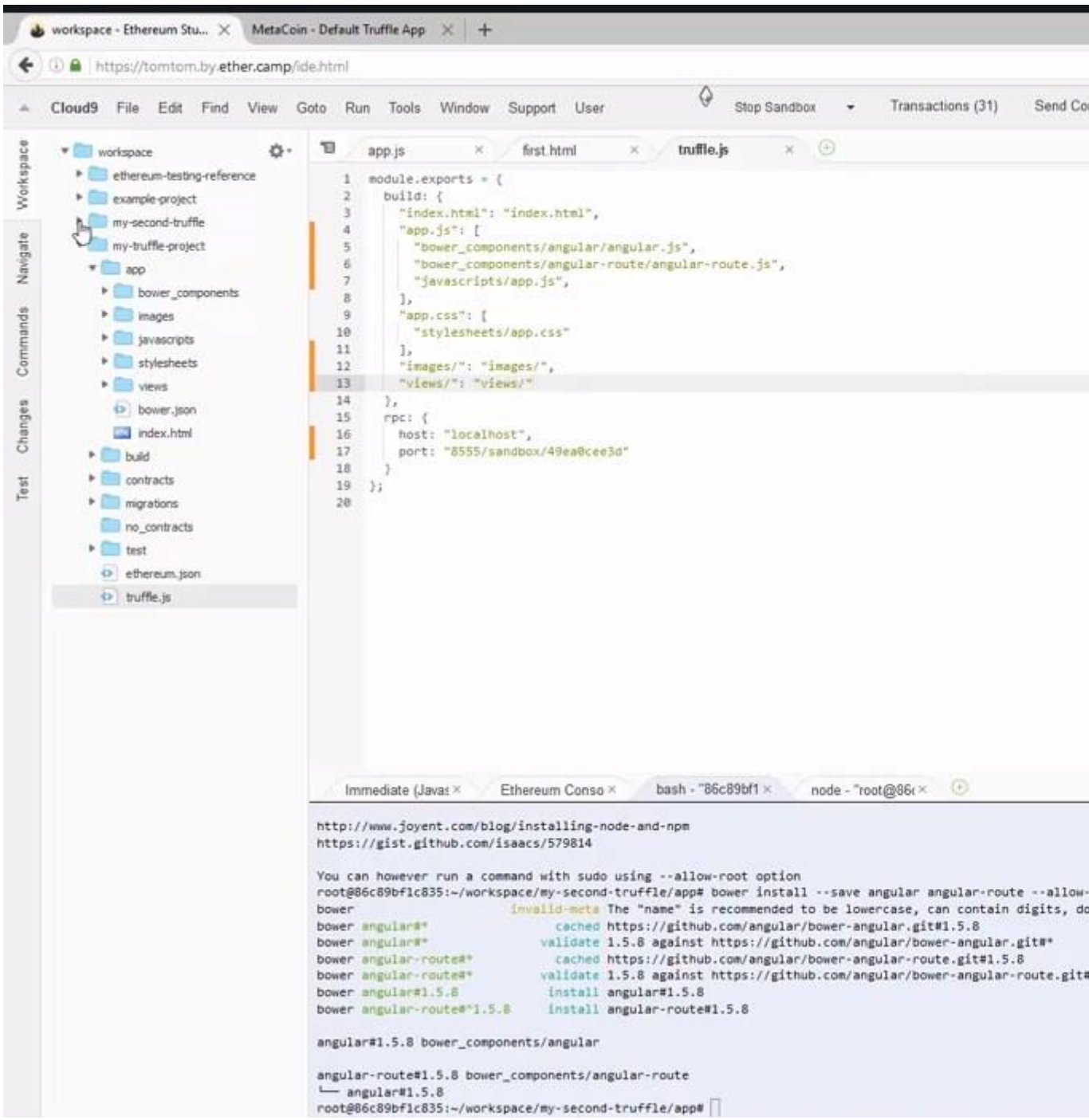
Рассмотрим теперь, как производится *интеграция* библиотеки Angular со средой Truffle. Потребуется создать небольшой участок кода. Для начала создадим новый проект Truffle, а затем установим некоторые пакеты. Один из них - это *bower*. Следует знать, что работа в Ethereum Studio из консоли выполняется от лица пользователя *root*, поэтому при попытке запуска *bower* среда выдаст ошибку о том, что вы пытаетесь запустить *bower* от лица пользователя *root*. В нашем случае можно быть уверенными, что это не приведет к негативным последствиям, поэтому можно добавить настройку *allow-root*.



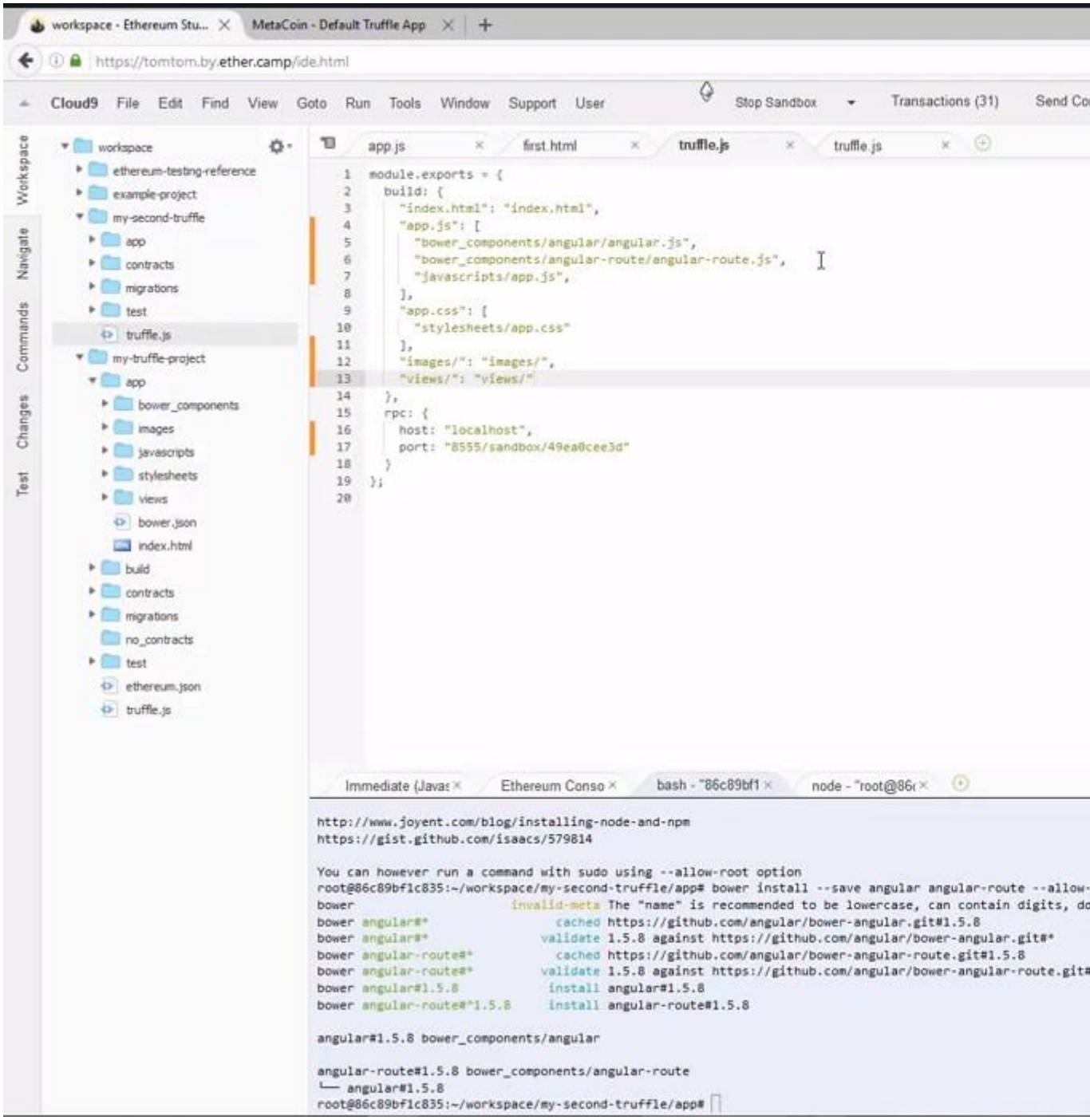
Начнем с инициализации нашего проекта с помощью команды *truffle init*. Добавим *bower* в папку *app*, и инициализируем *файл bower*, затем установим Angular и Angular-route.



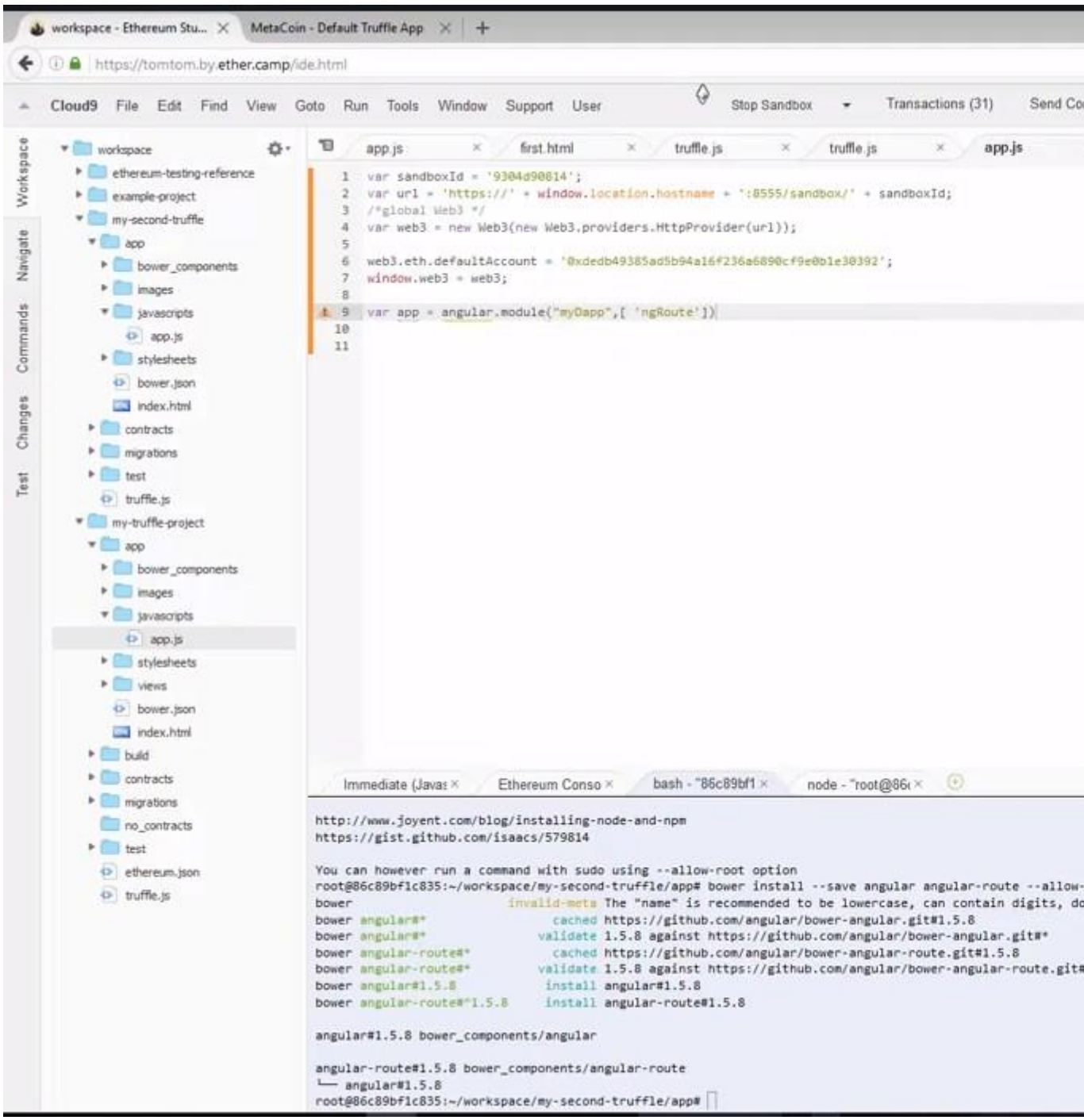
Теперь обратим внимание на каталог. Для включения в работу компонентов *bower* необходимо отредактировать *файл truffle.js*.



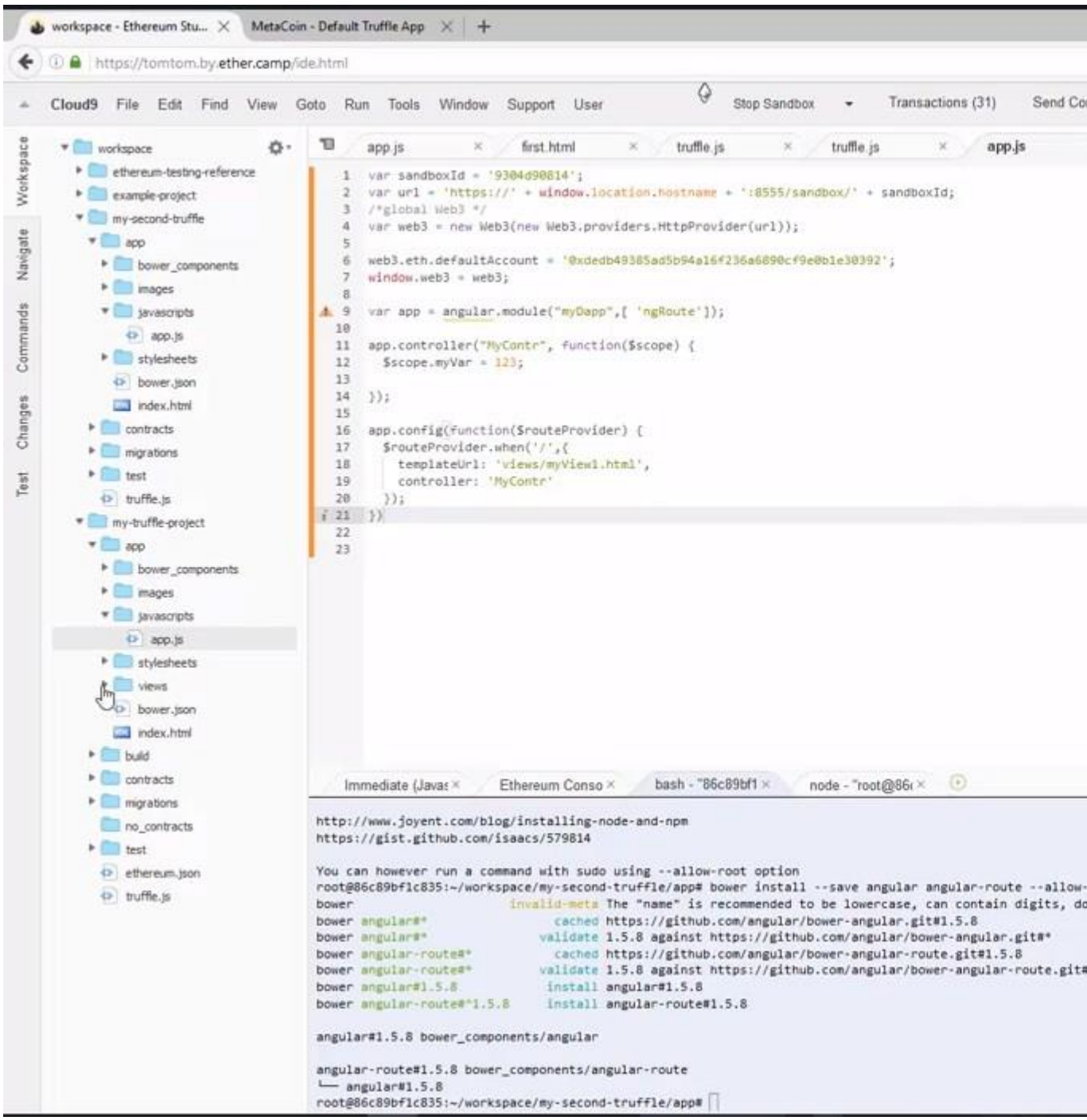
Можно видеть, что в нашем проекте Truffle, где уже подключена библиотека Angular, уже включены компоненты bower, Angular и Angular-route.



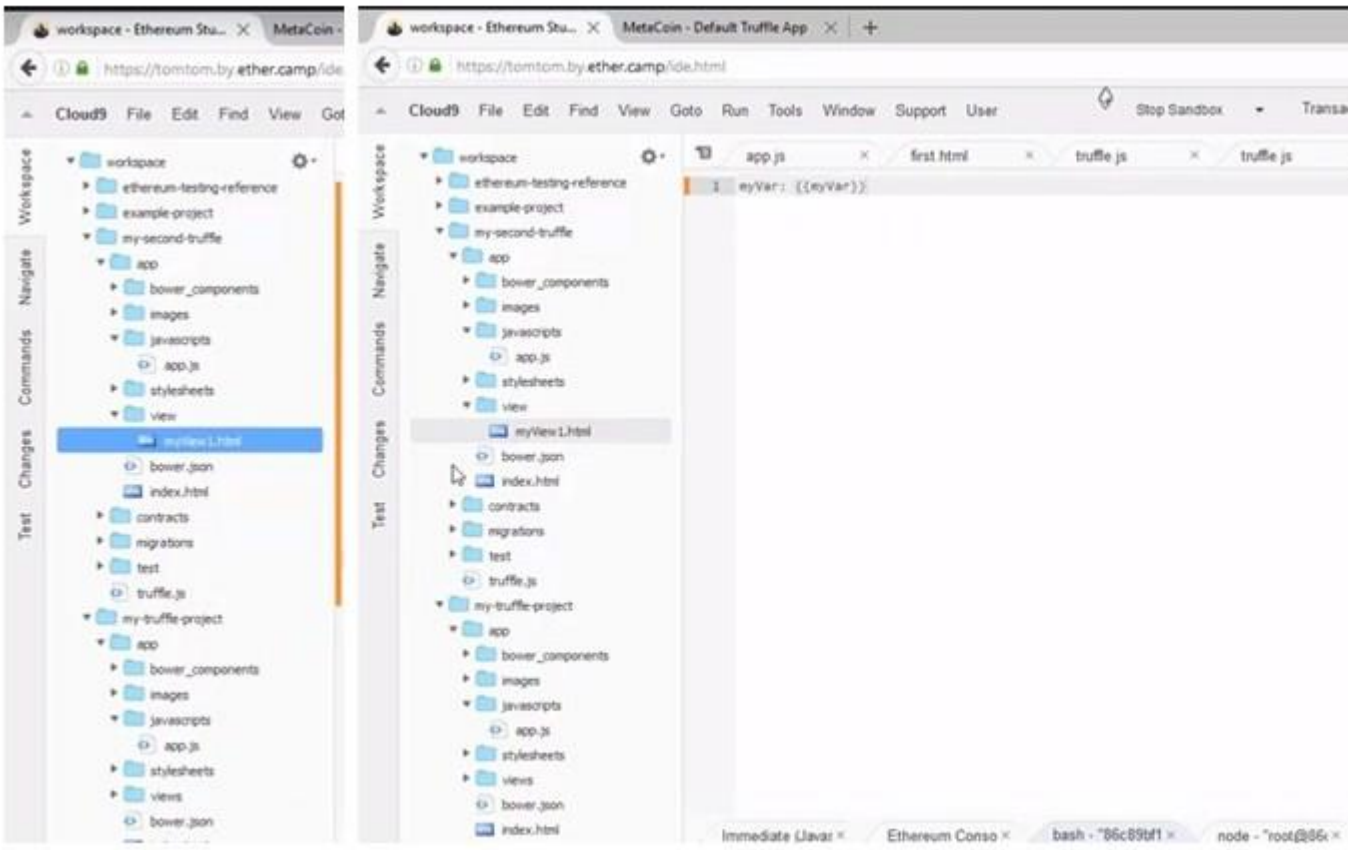
Кроме того, потребуется включить и каталог *views*. Теперь при запуске сборки Truffle будет произведено *объединение* файлов *index.html*, а также файлов *app.js:* сначала *angular.js*, затем *angular-route* и, наконец, *app.js* из каталога *javascripts*. В каталоге *javascripts* находится *приложение*, созданное Truffle *по* умолчанию. Теперь продолжим и скопируем *определение* для режима песочницы Необходимо отредактировать *объект* с адресом узла *HTTP*, поскольку собираемся получать *доступ* к песочнице с проектом с веб-сервера в этом домене, а *по* умолчанию из адресов доступен только *локальный хост*. Затем определим наше *приложение*.



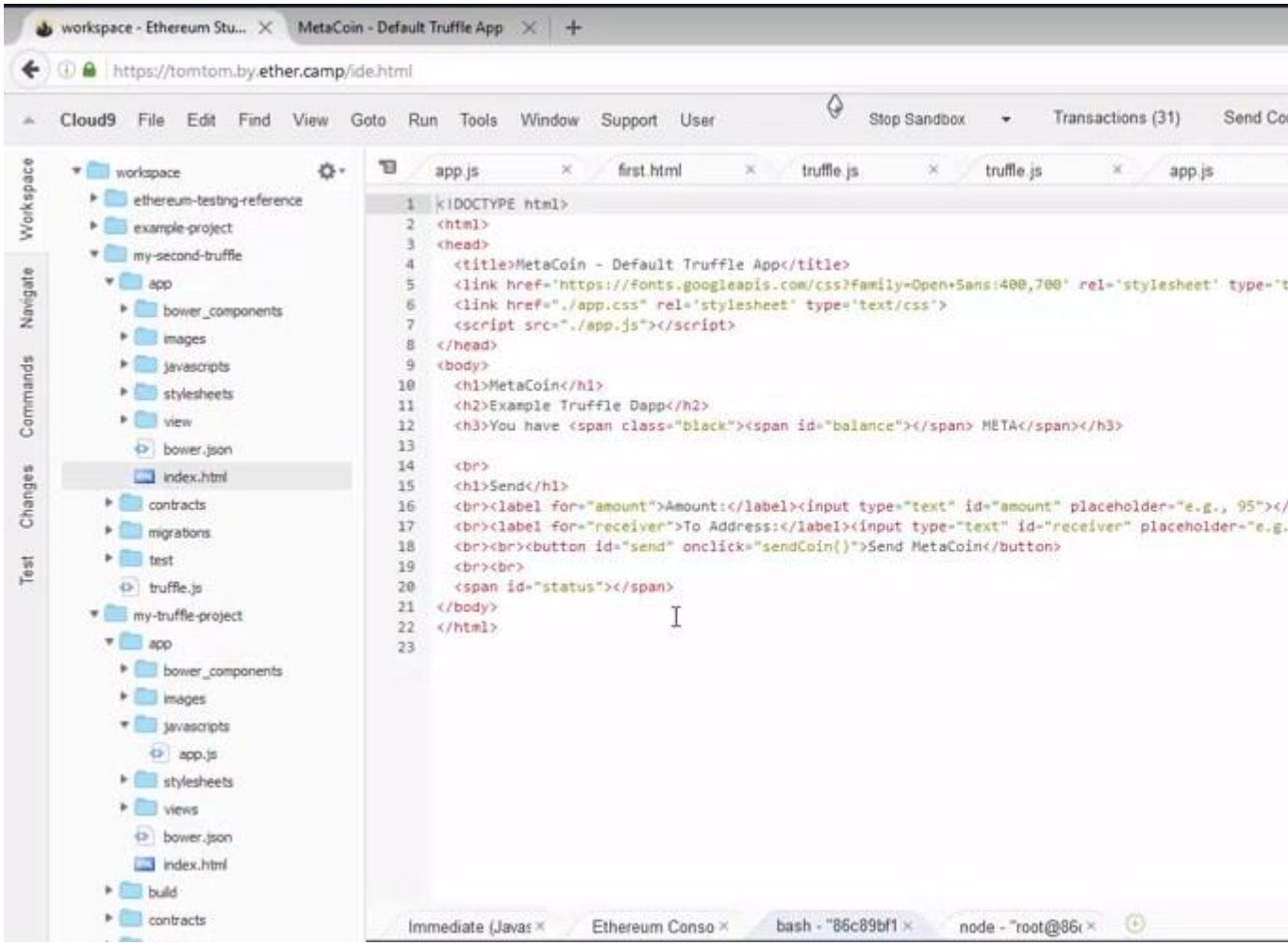
Доступ к Angular есть и задан глобально. Теперь можно определить контроллеры в других модулях. Можно либо разместить их в том же файле, либо разнести *по* разным файлам и указать их адреса в файле *truffle.js*.



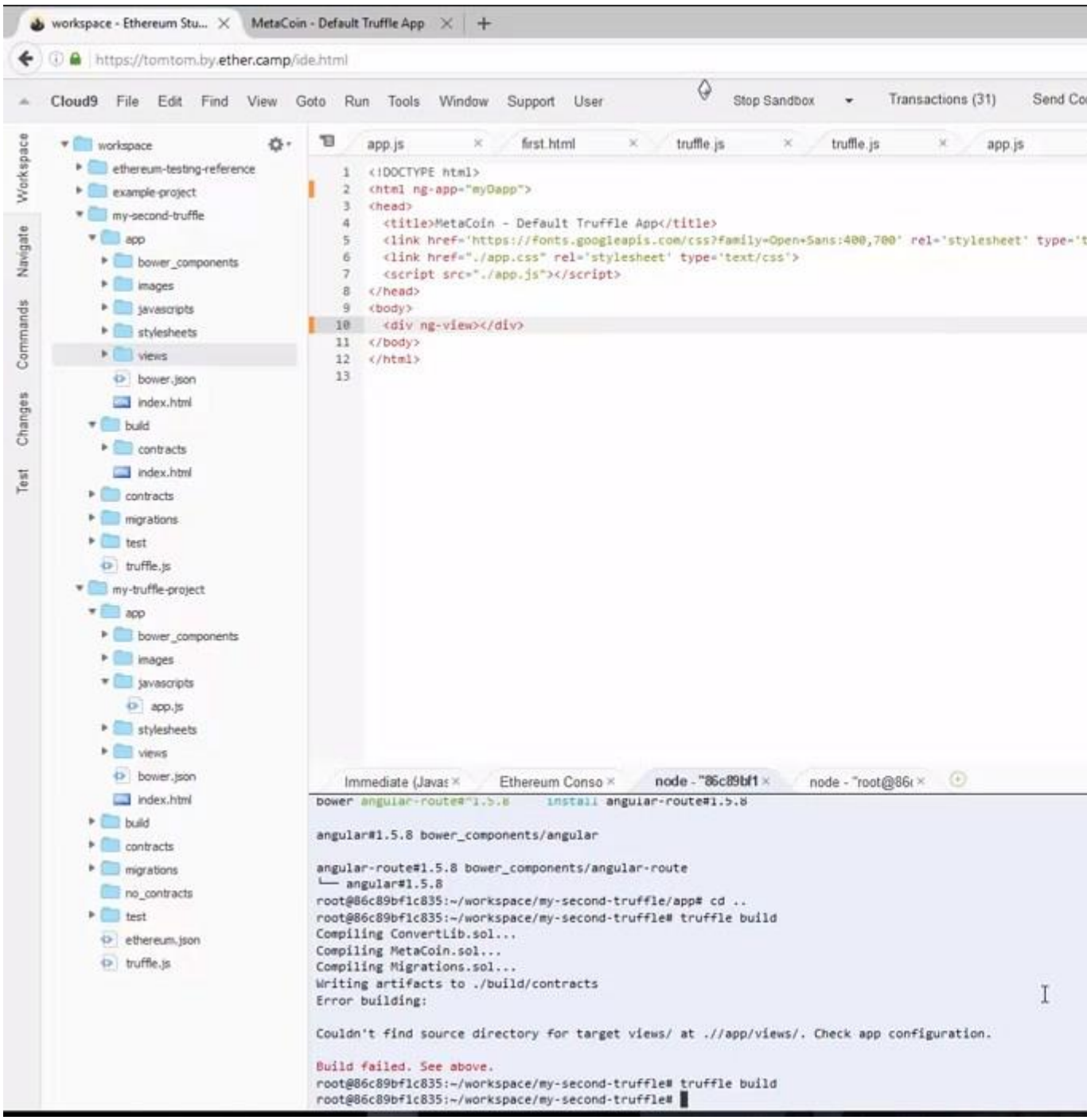
Добавим каталог view и страницу *myView1.html*.



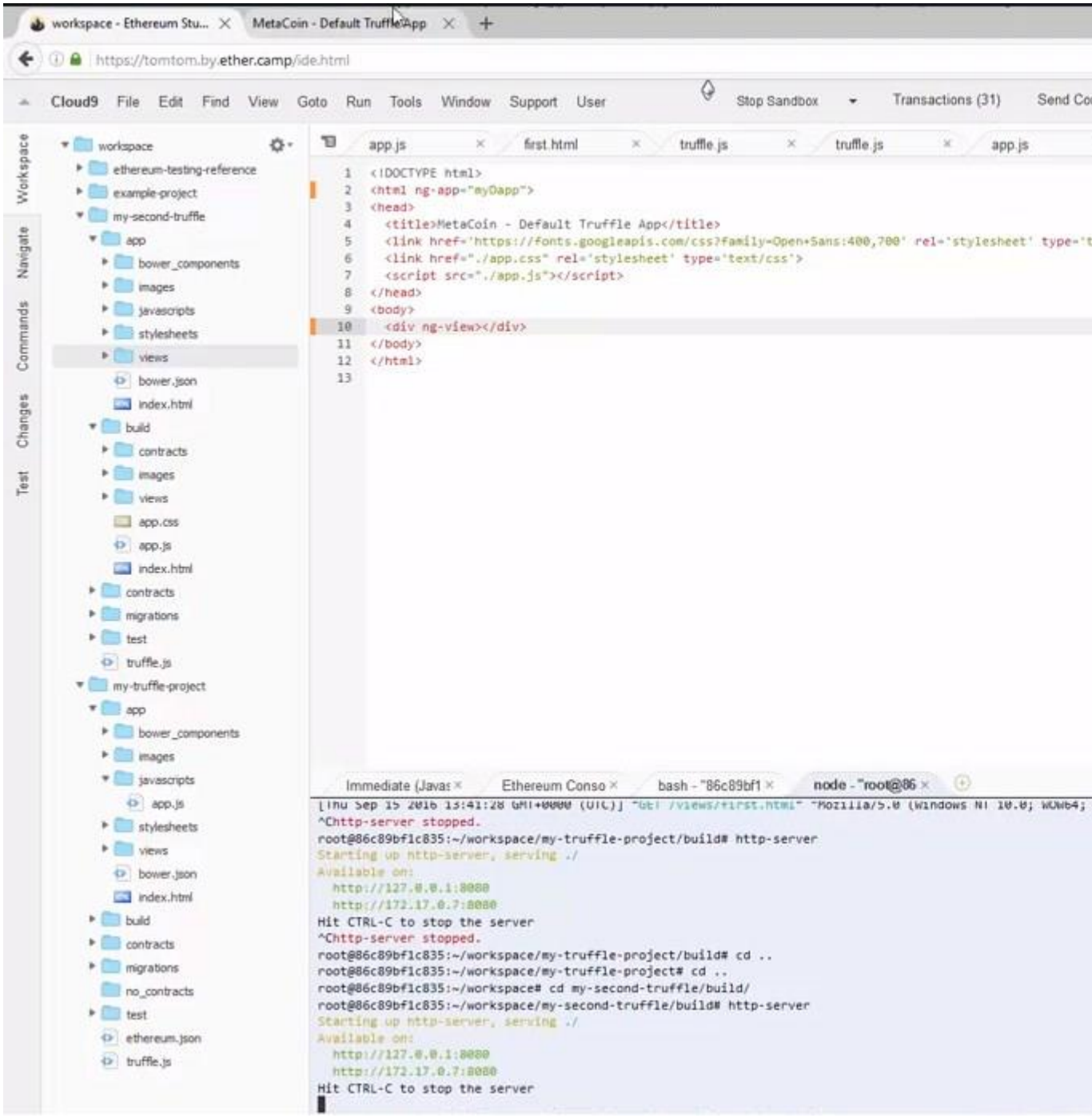
Перейдем к *index.html*.



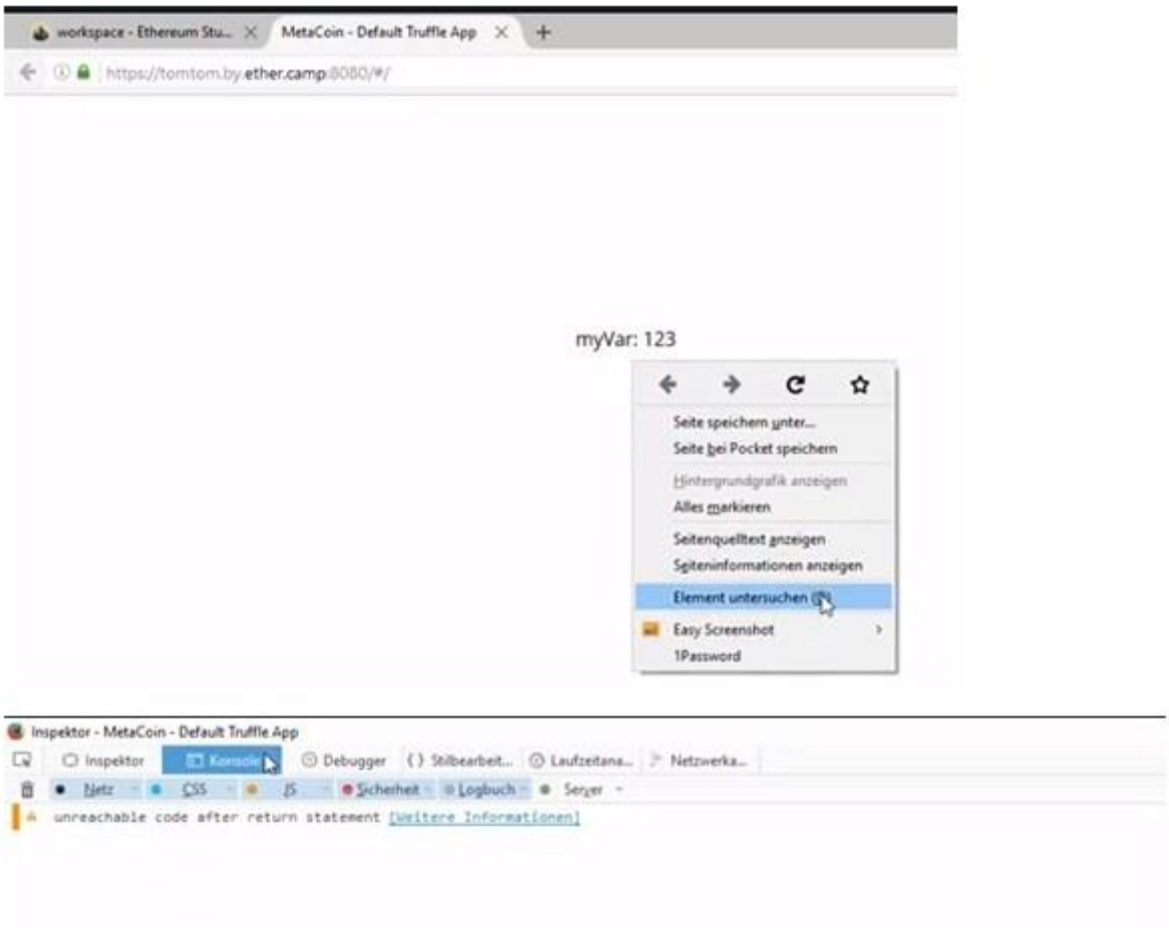
Чтобы задействовать библиотеку Angular, нужно объявить Angular с помощью команды *ng-app*. Кроме того, необходимо задать якорную точку для отображения. При вызове *truffle build* Truffle создает каталог *build* и копирует туда весь код сборки.



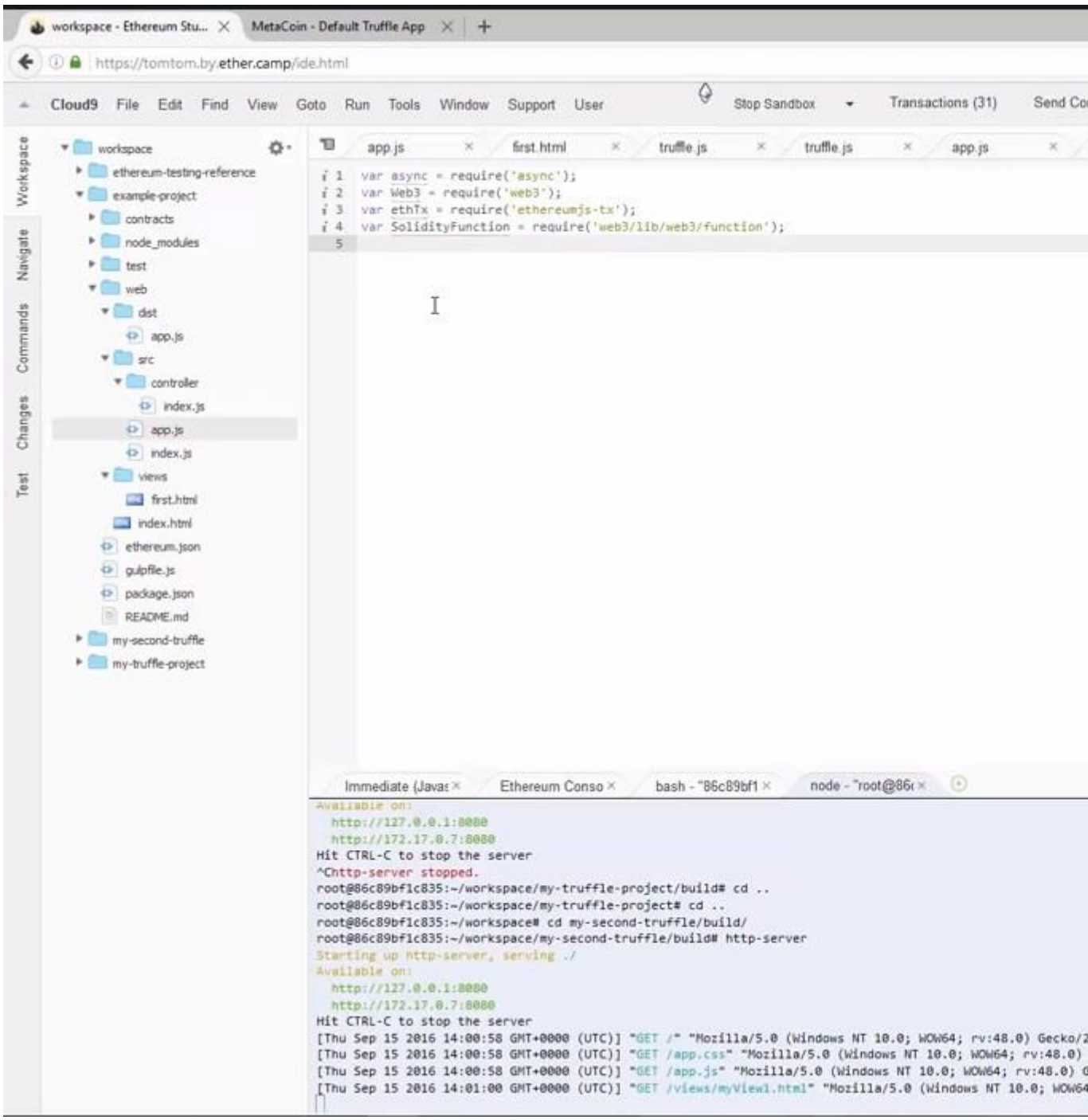
Откроем дополнительное окно консоли. Перейдем в каталог *build* и запустим *HTTP*-сервер.



Теперь можно увидеть, что *сборка* работает корректно, а *консоль* не выдала каких-либо ошибок.

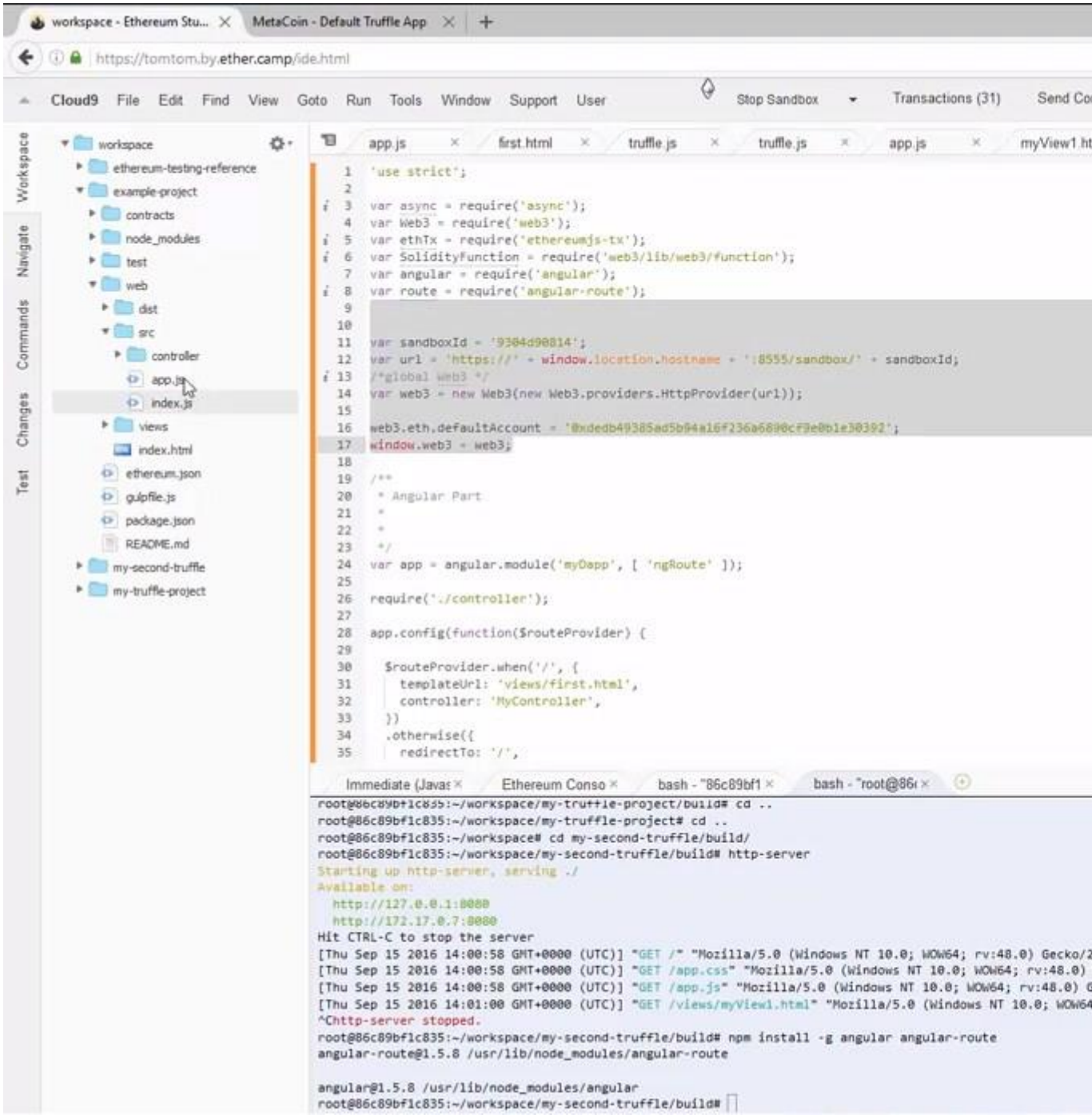


Вот так и производится *интеграция* библиотеки Angular в среду Truffle. Теперь вы можете добавлять свои сервисы, контроллеры или иные объекты *по* необходимости. Посмотрим, как все это выглядит в Ethereum Studio. Проект в Ethereum Studio поддерживает команду *gulp* в качестве диспетчера задач, а также оснастку Browserify. Этот инструмент очень полезен, так как способен использовать пакеты NPM и подключать их к единому файлу JavaScript. Так что если, скажем, *app.js* содержит множество заявлений *require*, оснастка Browserify объединит все эти пакты и обеспечит центральный *доступ* к ним. Поэтому как только *приложение* будет собрано и размещено в каталоге *dist*, значительно возрастут возможности использования контента доступных библиотек.

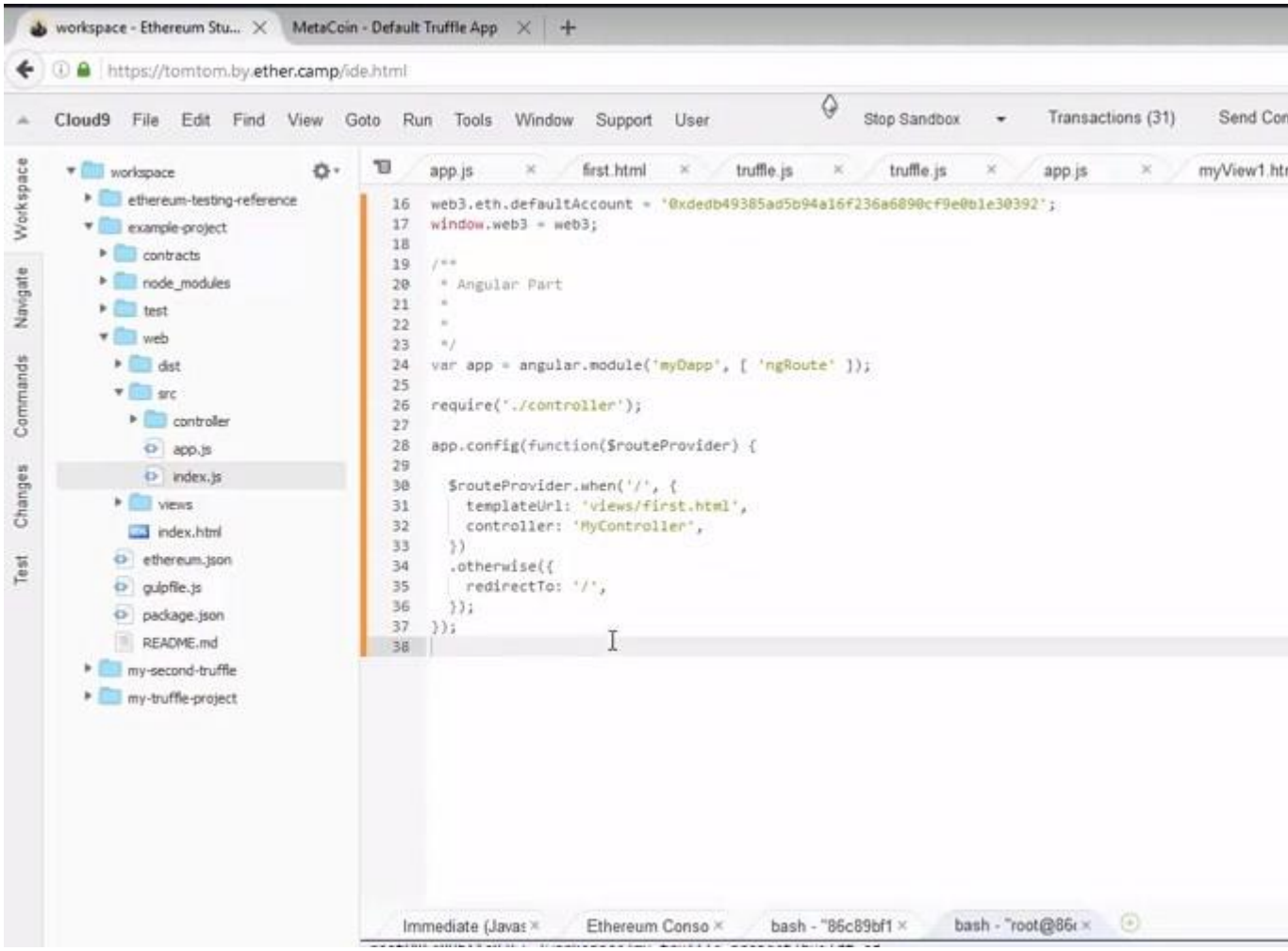


Как же использовать Angular? Библиотека Angular доступна в виде программного пакета для узлов сети Ethereum. Вместо того, чтобы устанавливать Angular посредством команды *bower*, загрузим ее через *менеджер* пакетов и задействуем посредством ключевого слова *require* в оснастке Browserify.

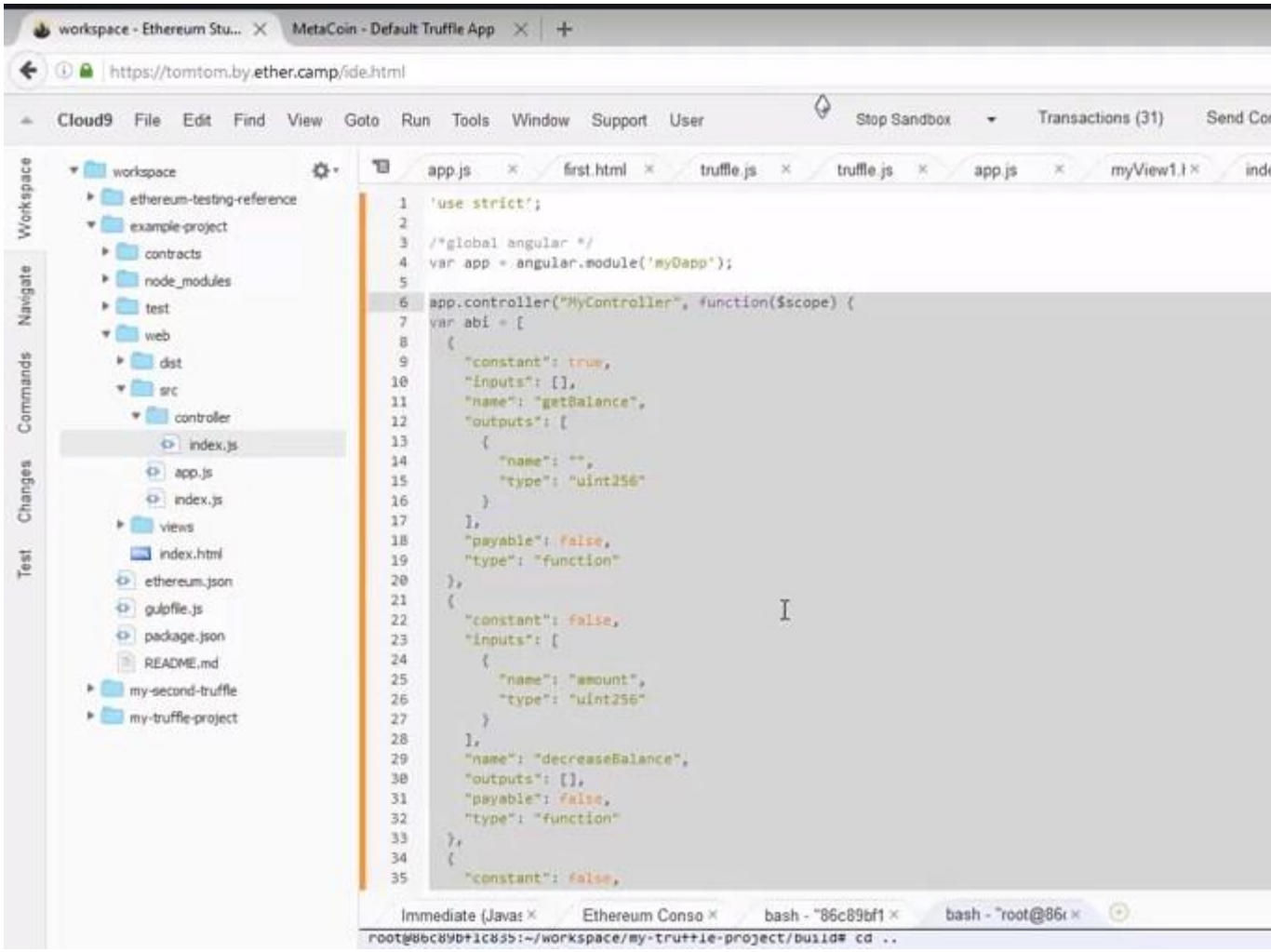
Откроем каталог *src*. Вместо *app.js* используем *index.js*. Итак, вот *список* заявлений *require*, затем - Angular и Angular-route. Теперь при сборке проекта оснастка Browserify автоматически включит библиотеку Angular и сделает возможным использование кода JavaScript. Затем вновь задаем узел *HTTP* и подключаем проект Web3 к окну.



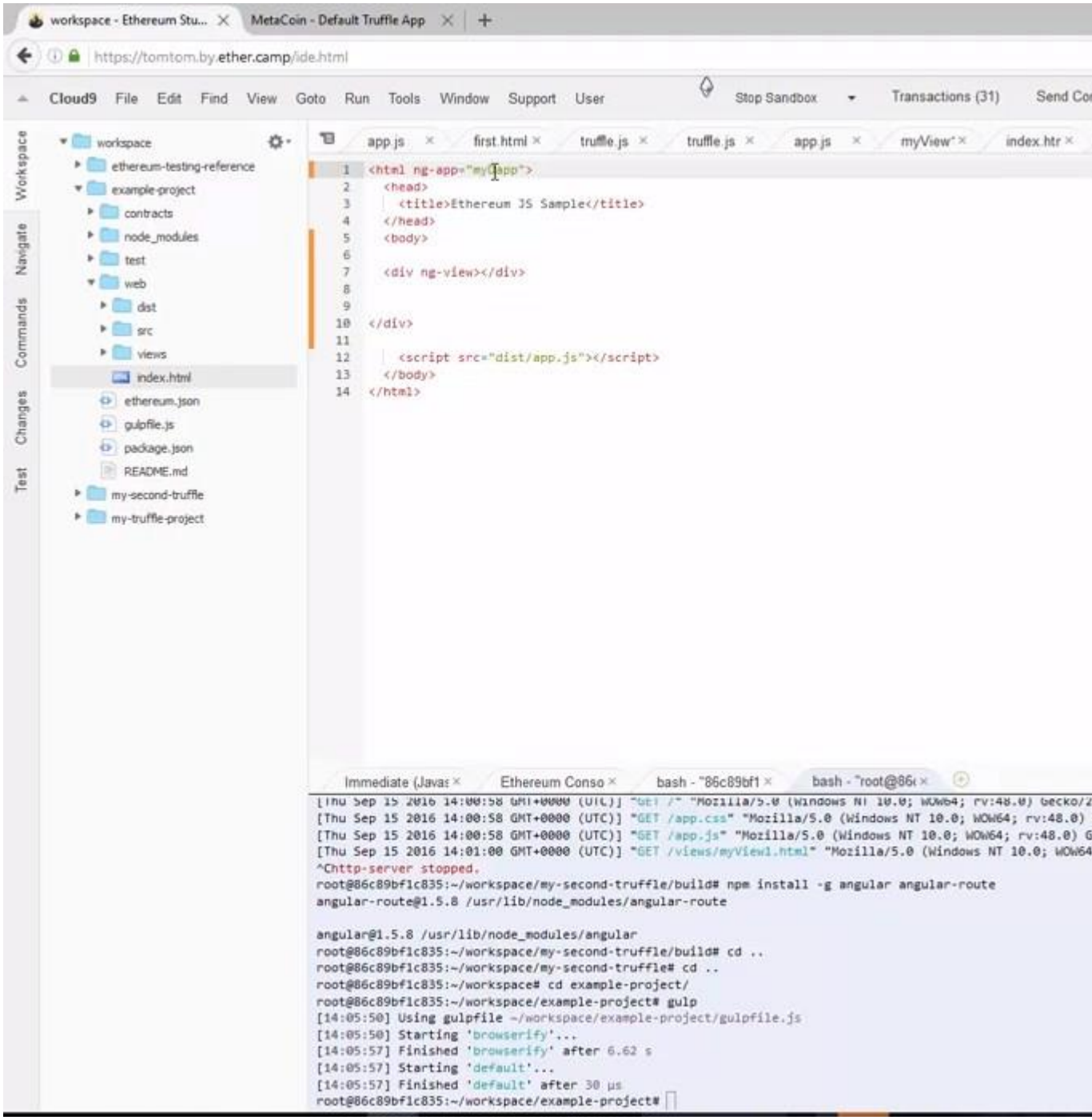
Есть *модуль* и его *конфигурация*.



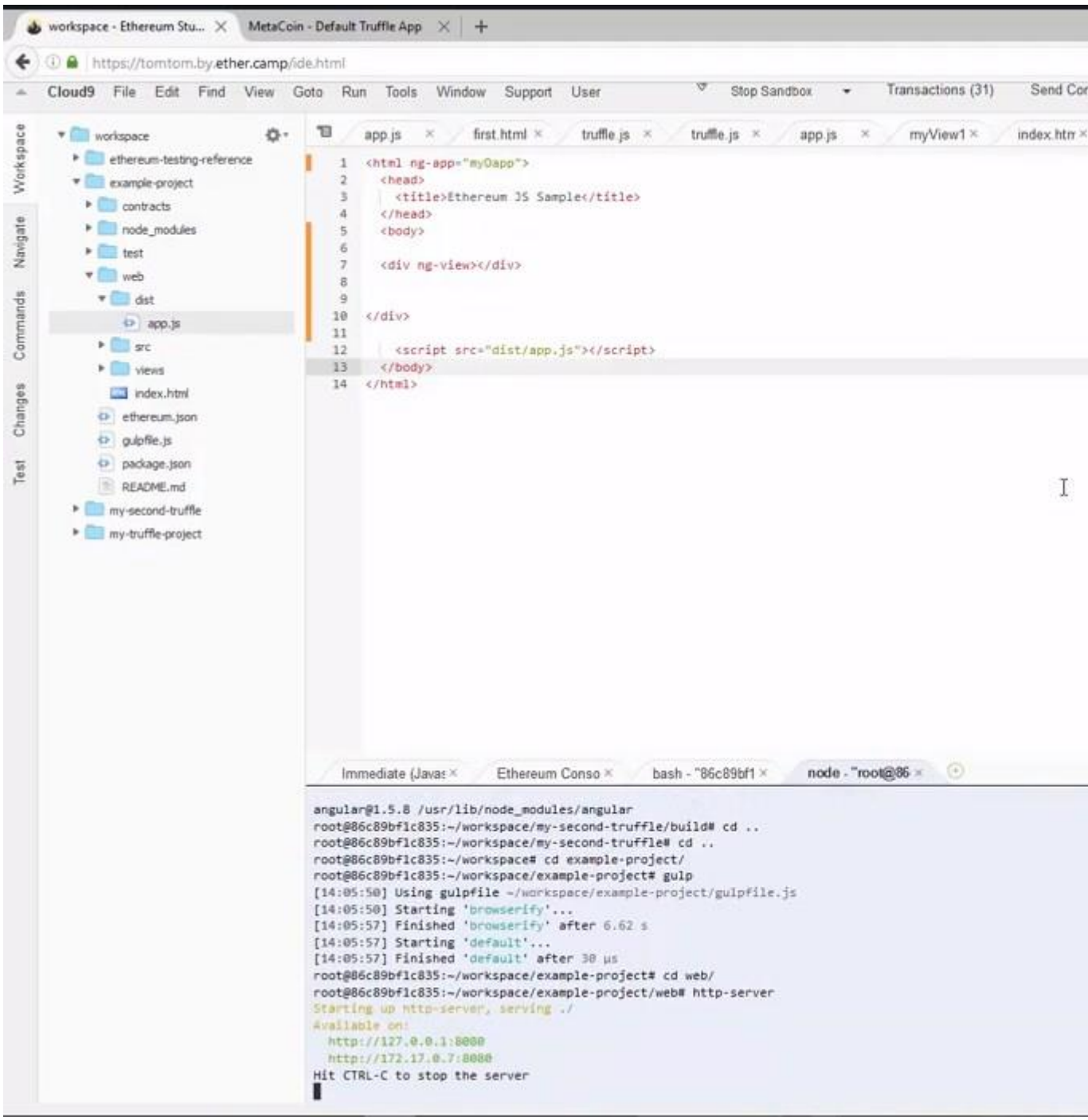
Заявление *require* требует от Browserify найти *файл index.js* в каталоге *controller* и включить его сюда же. А в файле *index.js* из каталога *controller* расположен наш *модуль*, как если бы он входил в число файлов JavaScript библиотеки Angular. *Контроллер* описан здесь же.



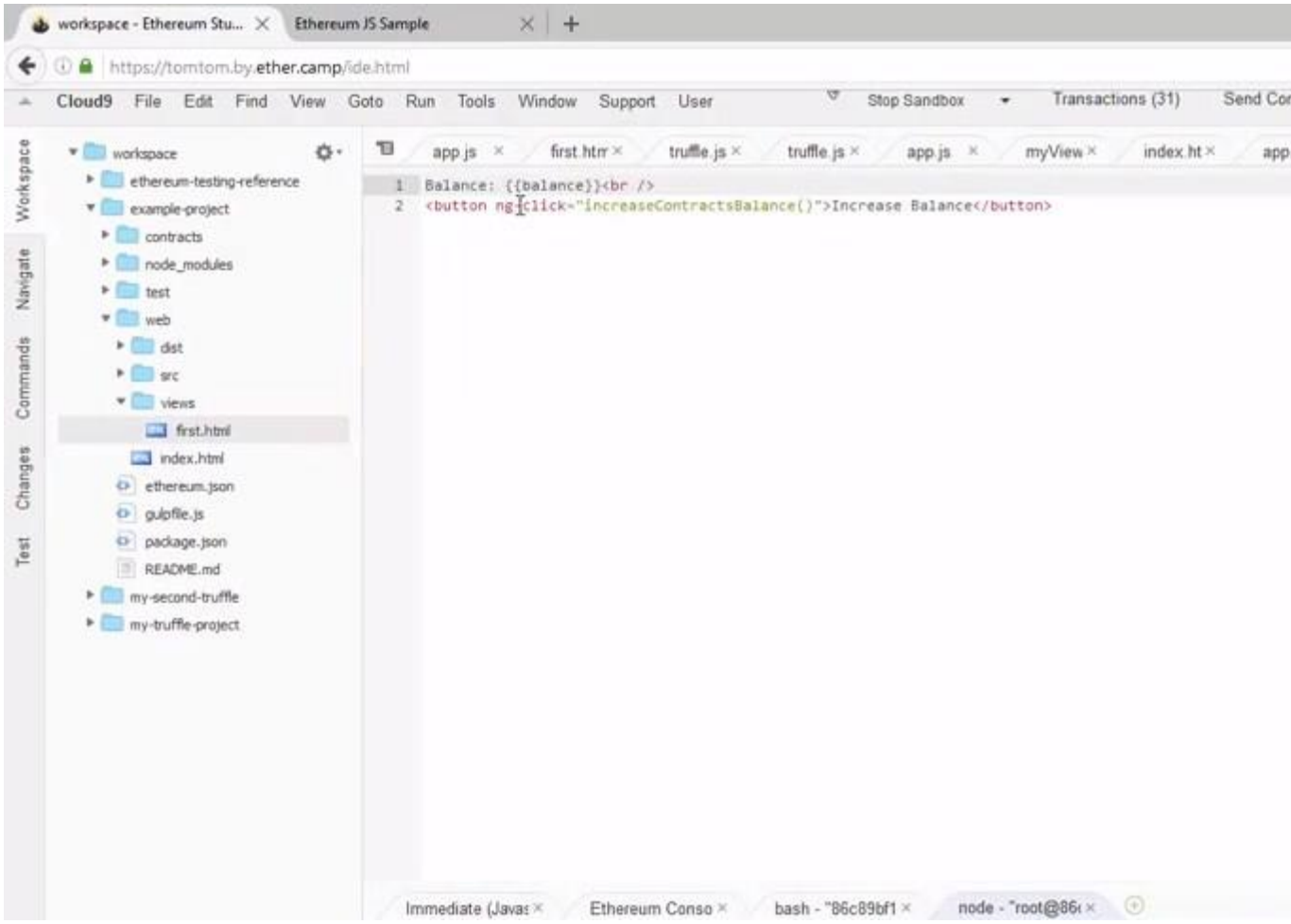
Перейдем в каталог *projects* и подготовимся к сборке проекта. Автоматически запускается Browserify. Если заглянуть в *файл index.html*, можно найти описание приложения *myDapp*, а также *якорь* для отображения, а также *файл app.js*, в котором расположен весь *контент*.



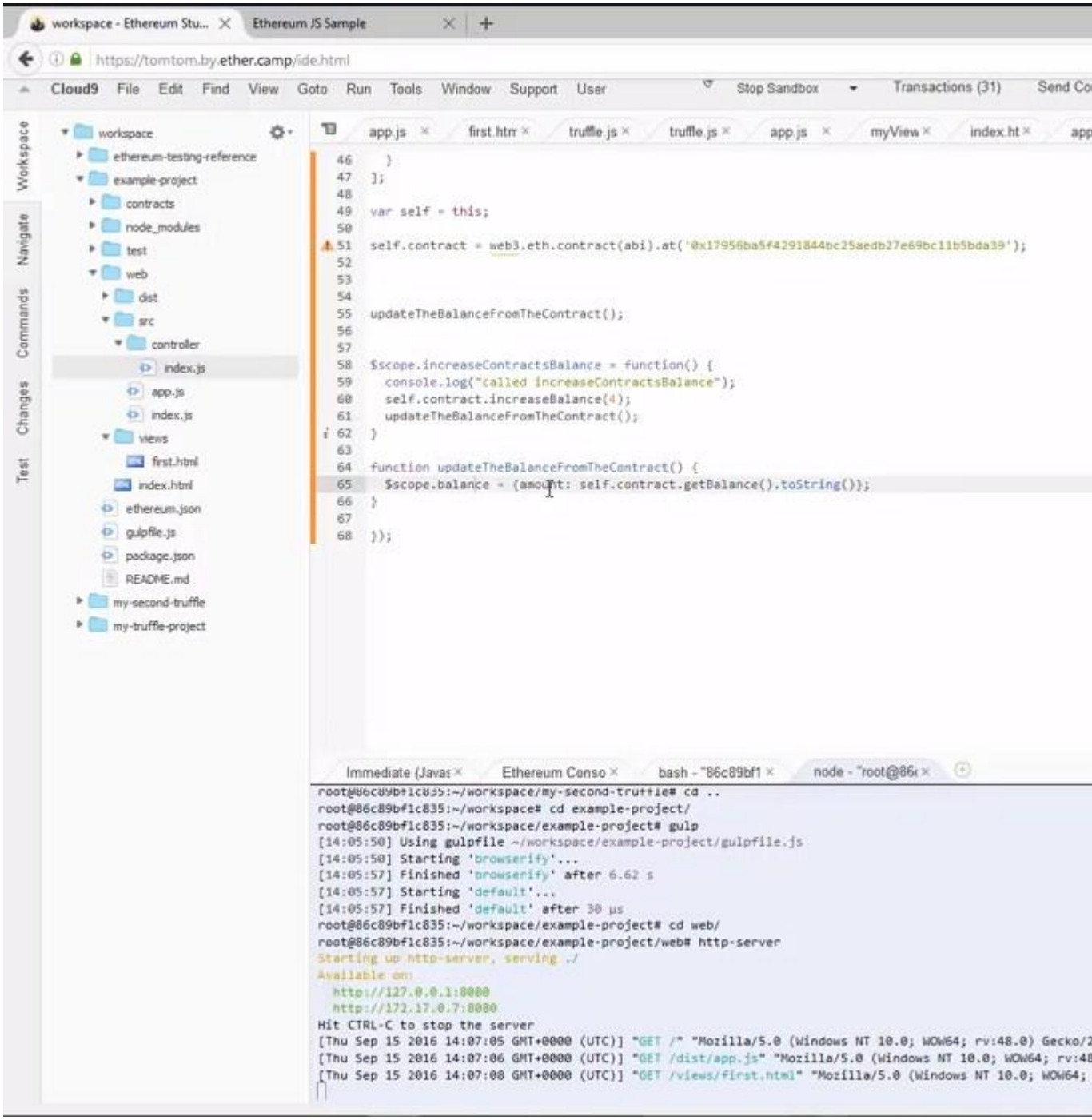
Посмотрим, как это работает после запуска.



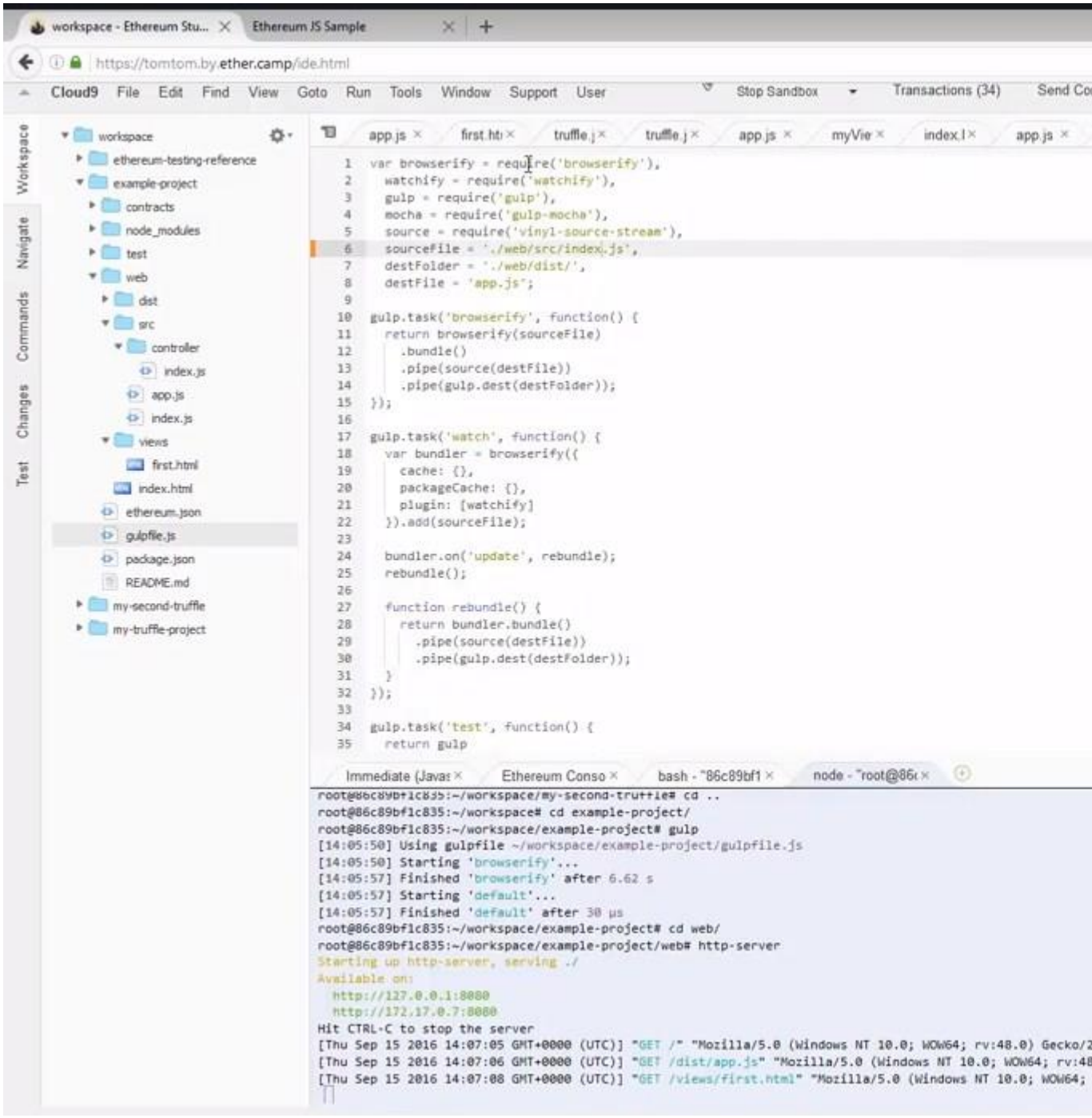
В файле *myView.html* описан *объект* balance, отображающий баланс счета, а также кнопка, при щелчке *по* которой баланс увеличивается.



В контроллере *myController* задана *функция increaseContractsBalance*, увеличивающая баланс на четыре и затем вызывающая функцию *updateTheBalanceFromTheContract*, меняющую отображаемый баланс.



Единственное, что нужно изменить в этом проекте - это отредактировать *файл gulp*. В нем можно увидеть некоторые требования, а также описание исходного файла и каталога и файла назначения. Был изменен исходный *файл* с *app.js* на *index.js*. Теперь *файл app*.js больше не будет использоваться при сборке этого проекта.



Задание №3 - Среда разработки Ethereum, Web3 и Truffle

В этом разделе научились работать со средой Truffle и Ethereum Studio от ether.camp. Оба этих инструмента находятся в активной разработке и часто обновляются. В этом задании можно выполнить одно из следующих упражнений на выбор:

- Зарегистрируйтесь на веб-сайте http://ether.camp и загрузите Ethereum Studio.
  1. Посетите веб-сайтhttps://live.ether.camp и зарегистрируйтесь на нем
  2. Войдите в Ethereum Studio со своей учетной записью
  3. Изучите материалы в readme.md
  4. Загрузите Truffle (выполните команду npm install -g truffle)
  5. Ознакомьтесь с режимом песочницы и запустите стандартные тесты из Ethereum Studio
- Установите Truffle на свой компьютер
  1. Загрузите Node Package Manager, при работе с Windows используйте веб-сайты https://nodejs.org/en/download/ и https://gir-scm.com/download/win
  2. Запустите Git Bash и установите Truffle с помощью команды npm install -g truffle
  3. Установите ethereumjs-testrpc с помощью команды npm install -g ethereumjs-testrpc
  4. Создайте новый каталог и инициализируйте стандартный проект Truffle с помощью команды truffle init
  5. Запустите сеть testrpc и выполните тесты Truffle по умолчанию с помощью команды truffle test