**Capítulo 12: Desenvolvimento Web3 usando Ethereum**

Web3 é uma biblioteca JavaScript que pode ser usada para se comunicar com um nó Ethereum via comunicação RPC. O Web3 funciona expondo métodos que foram habilitados via RPC. Isso permite o desenvolvimento de interfaces de usuário (UIs) que fazem uso da biblioteca Web3 para interagir com contratos implantados na blockchain.

Neste capítulo, exploraremos a API Web3 e introduziremos alguns exemplos detalhados de como contratos inteligentes são escritos, testados e implantados na blockchain Ethereum. Usaremos várias ferramentas como o Remix IDE e o Ganache para desenvolver e testar contratos inteligentes e veremos os métodos usados para implantar contratos inteligentes em redes de teste Ethereum e redes privadas. O capítulo explorará como frontends em HTML e JavaScript podem ser desenvolvidos para interagir com contratos inteligentes implantados na blockchain e apresentará bibliotecas avançadas como Drizzle. Os tópicos que abordaremos são os seguintes:

* Explorando Web3 com Geth
* Implantação de contratos
* Interação com contratos via frontends
* Frameworks de desenvolvimento

Começaremos com o Web3 e, gradualmente, construiremos nosso conhecimento com várias ferramentas e técnicas para desenvolvimento de contratos inteligentes. Você poderá testar seu conhecimento nas páginas de conteúdo bônus deste livro, onde desenvolveremos um projeto utilizando todas as técnicas que este capítulo irá abordar.

**Interagindo com contratos usando Web3 e Geth**

Web3 é uma API poderosa e pode ser explorada conectando-se a uma instância Geth. Para expor as APIs necessárias via Geth, o seguinte comando pode ser usado:

$ geth --datadir ~/etherprivate --networkid 786 --http --http.api \

"web3,net,eth,debug" --http.port 8001 --http.corsdomain <http://localhost:7777>

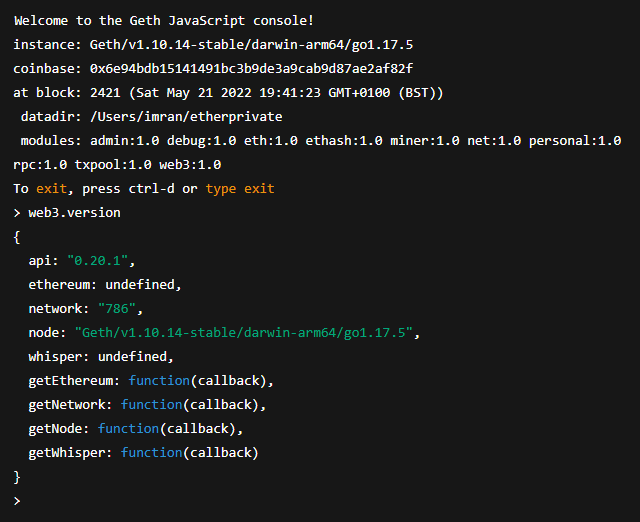
A flag --http no comando anterior permite os métodos web3, eth, net e debug. Existem outras APIs como personal, miner e admin que podem ser expostas adicionando seus nomes a esta lista.

Observe que usei a versão v1.10.14-stable do Geth. Todos os comandos funcionam corretamente para esta versão, mas podem não funcionar em versões anteriores. Tente usar a versão mais recente do Geth ao executar os exemplos deste capítulo, ou pelo menos a v1.10.14-stable.

A instância Geth pode ser conectada usando o seguinte comando:

$ geth attach ~/etherprivate/geth.ipc

Uma vez que o console JavaScript do Geth estiver em execução, o Web3 pode ser consultado:

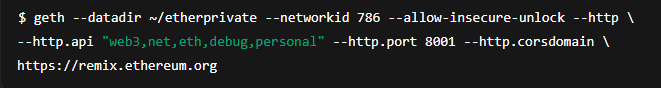


Agora que introduzimos o Web3, vamos considerar como o Remix IDE pode ser usado para implantar um contrato e como o console do Geth pode interagir com o contrato implantado.

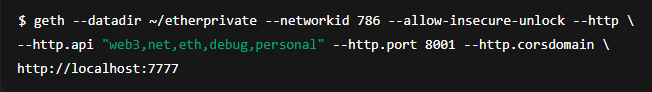
**Implantando contratos**

Um contrato simples pode ser implantado usando o Geth e interagido com ele usando o Web3 via a interface de linha de comando (CLI) que o Geth fornece (console ou attach). A seguir estão os passos para realizar isso.

Execute o cliente Geth usando um dos seguintes comandos. Já usamos todos esses comandos em diferentes contextos antes, e você pode usar qualquer um deles para executar o Geth. O primeiro comando, mostrado a seguir, permite conectividade com o Remix IDE, já que especificamos --http.corsdomain https://remix.ethereum.org:

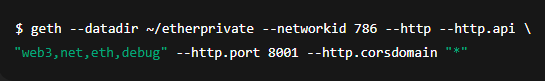


O segundo comando, mostrado a seguir, permite que localhost:7777 acesse o servidor RPC exposto pelo Geth. Esta opção é útil se você tiver um aplicativo em execução nesta interface e quiser dar a ele acesso ao RPC:



Ele também expõe RPC na porta 8001 via a flag http.port 8001, o que é útil no caso de você ter algum outro serviço ou aplicativo já escutando na porta 8545, o que impediria o Geth de usar essa porta já em uso. Isso porque o Geth escuta na porta 8545 para o servidor HTTP-RPC por padrão.

O último comando, mostrado a seguir, permite todas as conexões de entrada, pois --http.corsdomain está definido como \*:

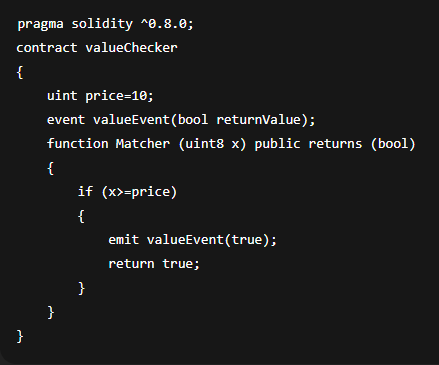


Se o console Geth ainda não estiver em execução, abra outro terminal e execute o seguinte comando:

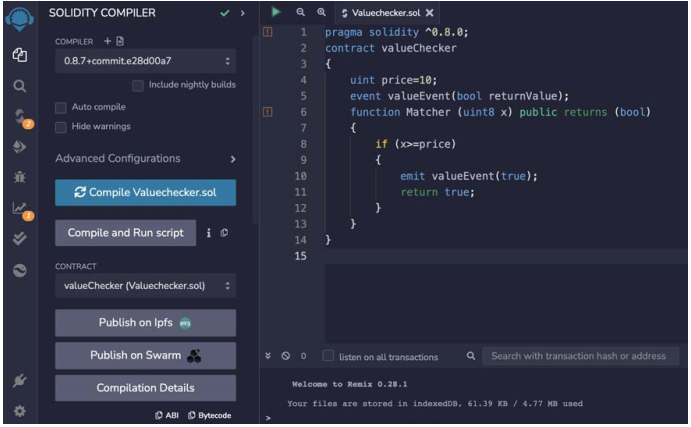
$ geth attach ~/etherprivate/geth.ipc

Para implantar um contrato inteligente, usamos o script de implantação do Web3. Os principais elementos do script, a Interface Binária de Aplicação (ABI) e o bytecode, podem ser gerados a partir do Remix IDE. Discutiremos o Remix IDE em mais detalhes posteriormente neste capítulo; por enquanto, estamos usando este IDE apenas para obter os elementos necessários (ABI e bytecode) para o script de implantação do Web3 usado na implantação do contrato.

Primeiramente, cole o seguinte código-fonte no Remix IDE:



Para aprender como baixar e usar o Remix IDE, consulte o Capítulo 10, *Ethereum na Prática*.



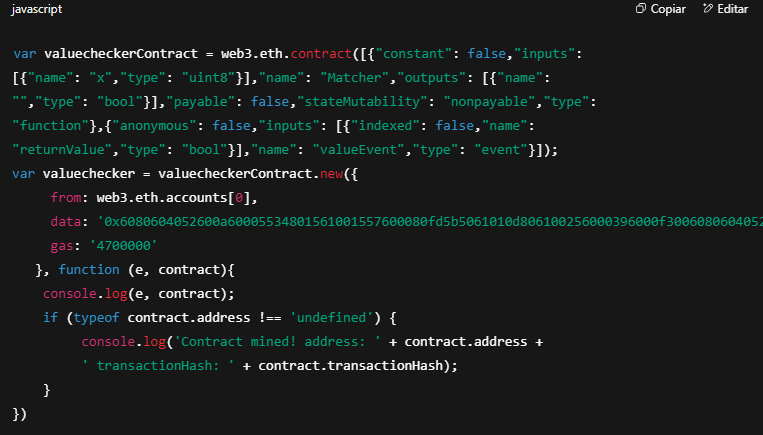
*Figure 12.1: Code shown in Remix*

**Web3 Desenvolvimento Usando Ethereum** (continuação)

Uma vez que o código seja colado no Remix IDE, ele aparecerá conforme mostrado no compilador Solidity:

**Figura 12.1**: Código exibido no Remix

Agora, crie o script de implantação do Web3 como se segue. Primeiro, gere a ABI e o bytecode usando os botões ABI e Bytecode no Remix IDE (mostrados no canto inferior esquerdo da Figura 12.1) e cole-os no seguinte script após os elementos web3.eth.contract( e data: respectivamente. O exemplo mostrado abaixo já tem a ABI e o Bytecode colados:



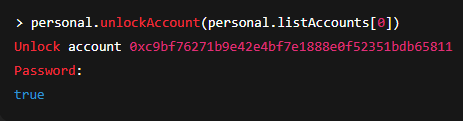
Certifique-se de que a conta esteja desbloqueada. Criamos contas anteriormente no Capítulo 10, *Ethereum na Prática*; podemos usar essa conta ou criar uma nova, se necessário.

Primeiramente, liste as contas usando o seguinte comando, que retorna a conta 0 (a primeira conta), como mostrado aqui:

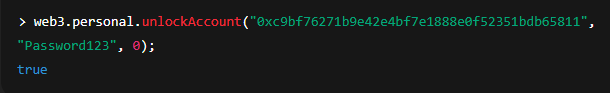
> personal.listAccounts[0]

"0xc9bf76271b9e42e4bf7e1888e0f52351bdb65811"

Agora desbloqueie a conta usando o seguinte comando. Será necessário fornecer a senha (passphrase) que você usou originalmente ao criar esta conta. Digite a senha para desbloquear a conta:



Para mais conveniência, a conta pode ser desbloqueada permanentemente durante a sessão do console/attach do Geth usando o comando mostrado aqui:



Agora podemos abrir o console Geth que já foi iniciado anteriormente e implantar o contrato. No entanto, antes de implantar o contrato, certifique-se de que a mineração está em execução no nó Geth; isso permitirá que contratos sejam minerados e, como resultado, implantados na blockchain. O seguinte comando pode ser usado para iniciar a mineração dentro do console Geth:

> miner.start()

Agora cole o script de implantação Web3 mencionado anteriormente no console Geth, conforme mostrado abaixo:



Depois disso, execute o script como mostrado abaixo, que contém o bytecode do contrato inteligente:



A linha de comando anterior mostra como ela se parece quando o script de implantação Web3 é colado no console Geth para implantação.

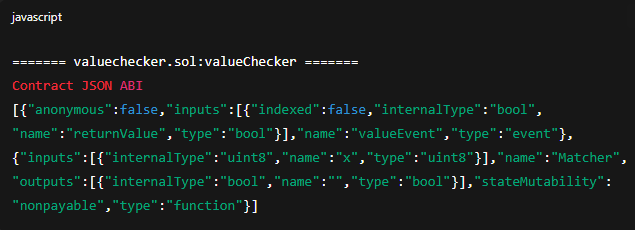
**Usando solc para gerar ABI e código**

A ABI e o código também podem ser obtidos usando o compilador Solidity, conforme mostrado nos trechos de código a seguir.

Para gerar a ABI, podemos usar o comando mostrado a seguir:

$ solc --abi valuechecker.sol

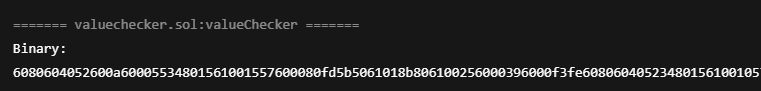
Esse comando produzirá a seguinte saída, com a ABI do contrato em formato JSON:



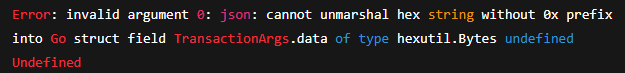
O próximo passo é gerar o código, o que pode ser feito com o seguinte comando:

$ solc --bin valuechecker.sol

Esse comando produzirá o binário (representado como hexadecimal) do código do contrato inteligente:

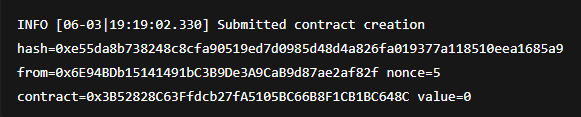


Ao gerar o binário para uso com o script de implantação que vimos anteriormente, basta adicionar 0x como prefixo à string binária gerada antes de usá-la no script, para evitar erros que podem ocorrer devido à não detecção do hexadecimal. Caso contrário, você pode ver uma mensagem semelhante à seguinte:

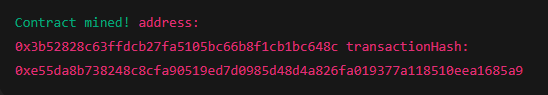


**Consultando contratos com Geth**

Também podemos ver a mensagem relevante nos logs do Geth para verificar se a transação de criação do contrato foi enviada; você verá mensagens semelhantes à mostrada a seguir:

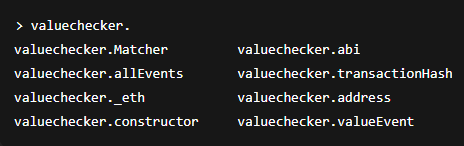


Note também que, no console do Geth, a seguinte mensagem aparece assim que a transação é minerada, indicando que o contrato foi minerado com sucesso:

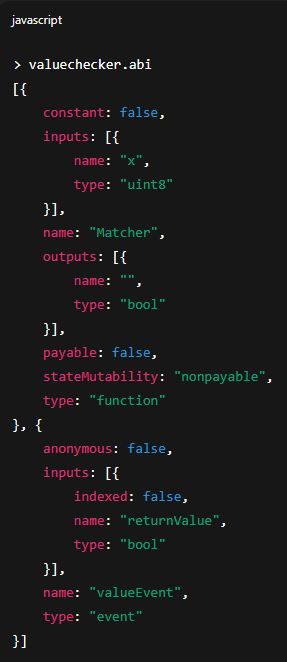


Observe que, na saída anterior, o hash da transação 0xe55da8b738248c8cfa90519ed7d0985d48d4a826fa019377a118510eea1685a9 também é mostrado.

Após o contrato ser implantado com sucesso, você pode consultar vários atributos relacionados a esse contrato, os quais também utilizaremos posteriormente neste exemplo, como o endereço do contrato e a definição da ABI. Lembre-se de que todos esses comandos são emitidos via o console do Geth, que já abrimos e usamos para a implantação do contrato:



Consulta à ABI:

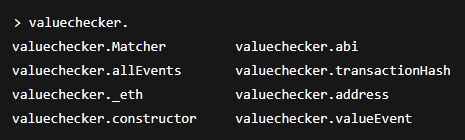


Agora vários métodos estão expostos, e o contrato pode ser consultado ainda mais. Por exemplo:

> eth.getBalance(valuechecker.address)

0

Agora podemos chamar os métodos reais no contrato. A lista dos vários métodos expostos pode ser vista da seguinte forma:



O contrato pode ser consultado ainda mais como se segue.

Primeiro, encontramos o hash da transação, que identifica a transação:

> valuechecker.transactionHash

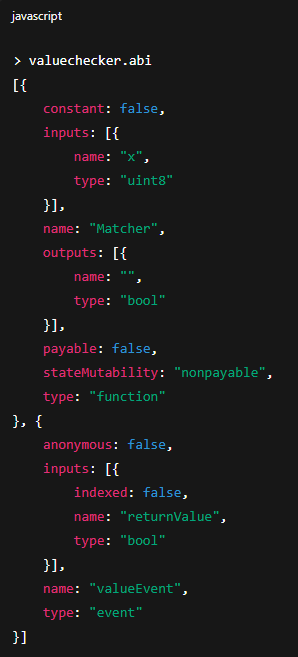
A saída desse comando é o hash da transação de criação do contrato:

0xe55da8b738248c8cfa90519ed7d0985d48d4a826fa019377a118510eea1685a9

Também podemos consultar a ABI, usando o seguinte comando:

> valuechecker.abi

A saída será a seguinte. Observe que mostra todas as entradas e saídas do nosso contrato de exemplo:



No exemplo a seguir, a função Matcher é chamada com argumentos. Argumentos, também chamados de parâmetros, são os valores passados para as funções. Lembre-se de que no código do contrato inteligente há uma condição que verifica se o valor é maior ou igual a 10 e, se for, a função retorna true; caso contrário, retorna false. Para testar isso, digite os seguintes comandos no console do Geth que você abriu.

Passe 12 como argumento, o que retornará true, pois é maior que 10:

> valuechecker.Matcher.call(12)

True

Passe 10 como argumento, o que retornará true, pois é igual a 10:

> valuechecker.Matcher.call(10)

True

Passe 9 como argumento, o que retornará false, pois é menor que 10:

> valuechecker.Matcher.call(9)

False

Nesta seção, aprendemos como usar o Remix IDE para criar e implantar contratos. Também aprendemos como o console do Geth pode ser usado para interagir com um contrato inteligente e exploramos quais métodos estão disponíveis para interagir com contratos inteligentes na blockchain. Também introduzimos a biblioteca Web3.js, que nos permite interagir com um nó Ethereum. Agora veremos como podemos interagir com o Geth usando JSON RPC via HTTP.

**Interagindo com o Geth usando requisições POST**

É possível interagir com o Geth via JSON RPC sobre HTTP. Para esse propósito, a ferramenta curl pode ser usada.

Um exemplo é mostrado aqui para familiarizá-lo com a requisição POST e mostrar como fazer requisições POST usando curl.

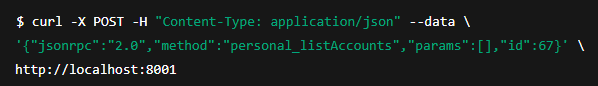
POST é um método de requisição suportado pelo HTTP, o qual é usado para enviar texto para um servidor web para processamento. Ele é amplamente usado em páginas da web para enviar dados de formulários para o servidor e atualizar bancos de dados.

Antes de usar a interface JSON RPC sobre HTTP, o cliente Geth deve ser iniciado com as opções apropriadas, como mostrado aqui:

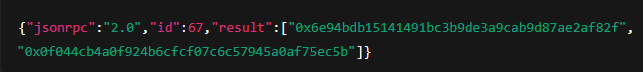
--http.api "web3"

Essa opção habilitará a interface web3 sobre HTTP. O comando curl no Linux pode ser usado com o propósito de comunicação via HTTP, como mostrado no exemplo a seguir.

Por exemplo, para recuperar a lista de contas usando o método personal\_listAccounts, o seguinte comando pode ser usado:



Isso retornará a saída, um objeto JSON com a lista de contas:



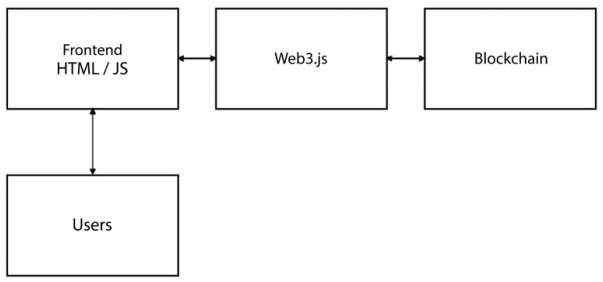
No comando curl anterior, --request é usado para especificar o comando da requisição, POST é a requisição, e --data é usado para especificar os parâmetros e valores. Finalmente, localhost:8545 é usado, onde o endpoint HTTP do Geth está exposto.

Nesta seção, cobrimos como podemos interagir com o contrato inteligente usando JSON RPC sobre HTTP. Embora essa seja uma maneira comum de interagir com os contratos, os exemplos que vimos até agora são baseados em linha de comando. Na próxima seção, veremos como podemos interagir com os contratos criando uma interface web amigável ao usuário.

**Interagindo com contratos via frontends**

Até agora, vimos como podemos interagir com um contrato usando o console do Geth via linha de comando, mas para que um aplicativo seja utilizável por usuários finais — que geralmente estão mais familiarizados com interfaces web — torna-se necessário construir frontends web para que os usuários possam se comunicar com os contratos inteligentes backend usando interfaces de páginas web familiares.

curl está disponível em: <https://curl.haxx.se/>



É possível interagir com os contratos usando a biblioteca web3.js a partir de páginas web baseadas em HTML/JS/CSS.

O conteúdo HTML pode ser servido usando qualquer servidor HTTP, enquanto o web3.js pode se conectar via RPC local ao cliente Ethereum (Geth) em execução e fornecer uma interface para os contratos na blockchain. Esta arquitetura pode ser visualizada no seguinte diagrama:

**Figura 12.2**: Arquitetura de interação entre web3.js, frontend e blockchain

**Instalando a biblioteca JavaScript web3.js**

Se você não tem web3.js instalado, use estas etapas; caso contrário, vá para a próxima seção.

O Web3, que discutimos anteriormente neste capítulo, foi considerado no contexto da API Web3 exposta pelo Geth. Nesta seção, introduziremos a biblioteca JavaScript Web3 (web3.js), que é usada para introduzir diferentes funcionalidades relacionadas ao ecossistema Ethereum em DApps. A biblioteca web3.js é uma coleção de vários módulos, os quais são listados a seguir com as funcionalidades que fornecem:

* **web3-eth**: Blockchain Ethereum e contratos inteligentes
* **web3-shh**: Protocolo Whisper (comunicação P2P e broadcast)
* **web3-bzz**: Protocolo Swarm, que fornece armazenamento descentralizado
* **web3-utils**: Fornece funções auxiliares para desenvolvimento de DApps

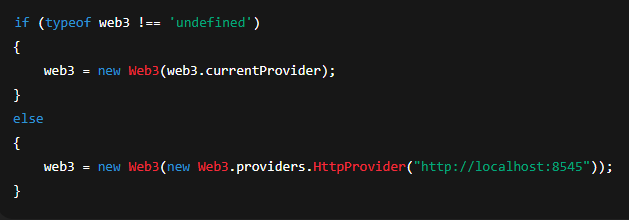
A biblioteca web3.js pode ser instalada via npm simplesmente emitindo o seguinte comando:

$ npm install web3

web3.js também pode ser baixado diretamente de: <https://github.com/ethereum/web3.js>

**Criando um objeto Web3**

O primeiro passo ao criar um aplicativo baseado em web3.js é criar o objeto web3. Ele é criado selecionando o provedor Web3 apropriado disponível, o qual serve como um "ponto de entrada" para a blockchain através do servidor HTTP RPC exposto em um nó Geth em execução local. O código necessário será semelhante ao seguinte:



Esse código verifica se já há um provedor Web3 disponível; se houver, ele definirá o provedor como o provedor atual. Caso contrário, define o provedor como localhost:8545; é nesse endereço que a instância local do Geth está expondo o servidor HTTP-RPC. Em outras palavras, a instância Geth está executando um servidor HTTP-RPC, que escuta na porta 8545. No nosso caso, como iniciamos o Geth com a porta 8001, será 8001 que usaremos em nosso código. Observe que não há necessidade de executar esse código agora, pois criaremos um objeto web3 mais adiante no exercício.

Até agora, exploramos como instalar o web3.js, a biblioteca JavaScript da Ethereum, e como criar um objeto web3 que pode ser usado para interagir com os contratos inteligentes usando o provedor HTTP em execução no localhost como parte da instância Geth.

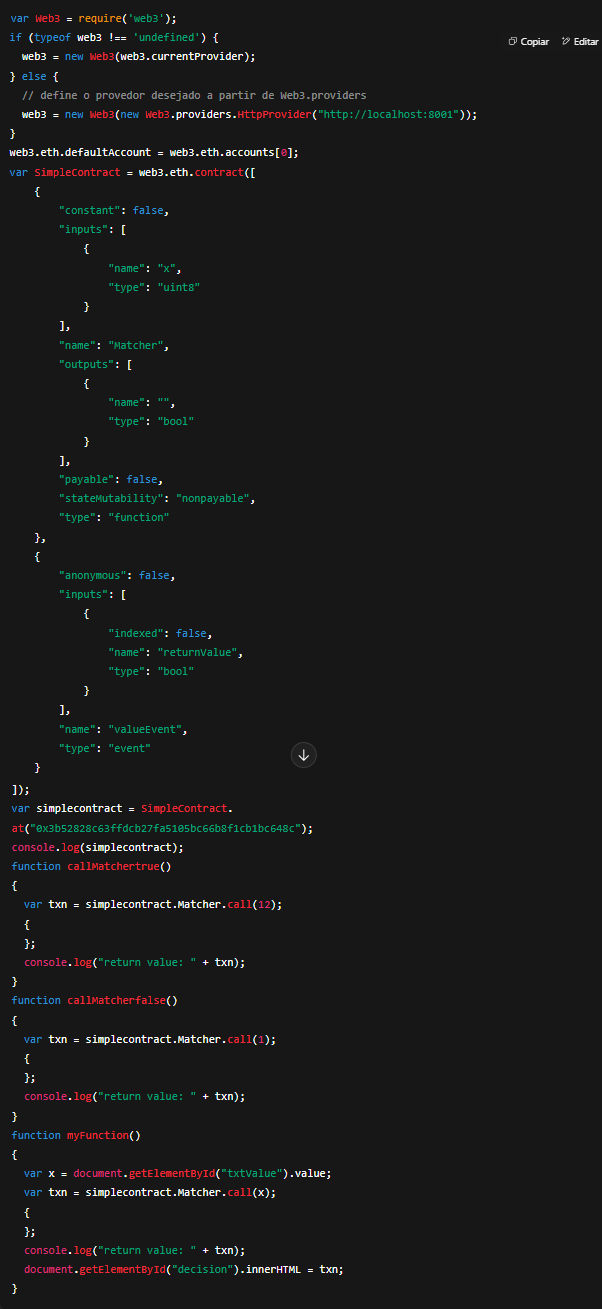
**Nota:** drequinox é específico ao usuário sob o qual esses exemplos foram desenvolvidos; você verá o nome do usuário sob o qual está executando esses comandos.

**Criando um arquivo JavaScript app.js**

Na seção seguinte, será apresentado um exemplo que fará uso de web3.js para permitir interação com os contratos por meio de uma página web servida por um servidor HTTP simples.

Isso pode ser feito seguindo estes passos. Primeiro, crie um diretório chamado /simplecontract/app, o diretório home. Este é o diretório principal sob seu ID de usuário no Linux ou macOS. Pode ser qualquer diretório, mas neste exemplo, usamos o diretório home.

Depois, crie um arquivo chamado app.js e escreva ou copie o seguinte código nele:



Este arquivo contém vários elementos:

Primeiro, criamos o objeto web3 e fornecemos uma instância local do Geth escutando na porta 8545 como o provedor Web3.

Depois disso, web3.eth.accounts[0] é selecionada como a conta com a qual todas as interações com o contrato inteligente serão realizadas.

Em seguida, a **ABI** é fornecida, a qual serve como a interface entre o usuário e o contrato. Ela pode ser consultada usando o Geth, gerada usando o compilador Solidity ou copiada diretamente dos detalhes do contrato no Remix IDE.

Após isso, o simplecontract é criado, o qual se refere ao contrato inteligente com o endereço:

0x3b52828c63ffdcb27fa5105bc66b8f1cb1bc648c

Finalmente, declaramos três funções:

* callMatchertrue()
* callMatcherfalse()
* myFunction()

Uma vez que o objeto web3 foi corretamente criado e uma instância de simplecontract foi criada, chamadas às funções do contrato podem ser feitas facilmente.

callMatchertrue() simplesmente chama a função Matcher do contrato inteligente usando o objeto simplecontract que criamos no arquivo app.js.

De forma similar, callMatcherfalse() chama a função Matcher do contrato inteligente fornecendo o valor 1.

Por fim, a função myFunction() é definida, a qual contém uma lógica simples para ler o valor fornecido pelo usuário na caixa de texto txtValue na página web e usa esse valor para chamar a função Matcher do contrato inteligente.

Após as funções, o valor de retorno também é exibido no console de depuração, disponível nos navegadores através de console.log.

Chamadas podem ser feitas usando simplecontractinstance.Matcher.call e, então, passando o valor para o argumento. Relembre a função Matcher no código Solidity:

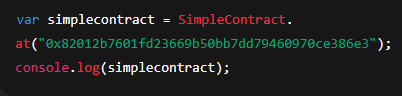
function Matcher (uint8 x) returns (bool)

Ela recebe um argumento x do tipo uint8 e retorna um valor booleano, true ou false. Assim, a chamada é feita ao contrato, como mostrado aqui:

var txn = simplecontractinstance.Matcher.call(12);

No exemplo acima, console.log é usado para imprimir o valor retornado pela chamada da função. Uma vez que o resultado da chamada está disponível na variável txn, ele pode ser usado em qualquer parte do programa, por exemplo, como parâmetro para outra função JavaScript.

Finalmente, verifique a disponibilidade do contrato. Esta linha de código usa console.log para imprimir os atributos do contrato simples, a fim de verificar a criação bem-sucedida do contrato:

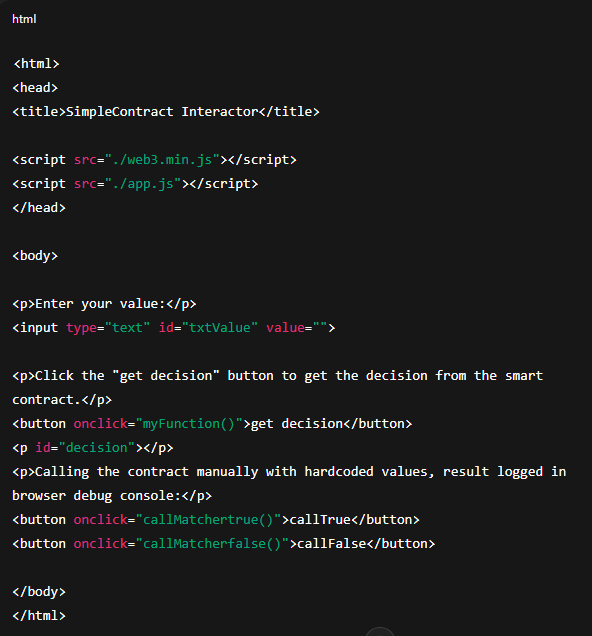


Uma vez que essa chamada seja executada, ela exibirá vários atributos do contrato, indicando que o objeto web3 foi criado com sucesso e que o HttpProvider está disponível. Qualquer outra chamada pode ser usada para verificar a disponibilidade, mas aqui, estamos usando a impressão dos atributos do contrato simples.

Nesta parte do exemplo, criamos uma instância de simplecontract e, então, usamos console.log para exibir alguns atributos do contrato, o que indica a criação bem-sucedida da instância de simplecontract.

**Criando uma página web frontend**

A próxima etapa é criar a página web frontend. Para isso, criamos um arquivo chamado index.html com o código-fonte mostrado a seguir:



Este arquivo servirá como o frontend da nossa aplicação descentralizada. Em outras palavras, ele fornece a interface do usuário para interação com os contratos inteligentes.

Primeiro, referenciamos a biblioteca JavaScript web3.js e o arquivo app.js, que criamos anteriormente nesta seção. Isso permitirá que o arquivo HTML chame as funções necessárias desses arquivos.

Depois disso, HTML padrão é usado para criar um campo de texto de entrada para que os usuários possam digitar os valores.

Em seguida, usamos o evento onclick para chamar a função myFunction() que declaramos em nosso arquivo JavaScript app.js.

Finalmente, dois eventos onclick com os botões callTrue e callFalse são usados para chamar as funções callMatchertrue() e callMatcherfalse(), respectivamente.

Estamos mantendo tudo muito simples de propósito; não há necessidade direta de usar jQuery, React ou Angular aqui — o que seria um tópico à parte. No entanto, esses frameworks frontend facilitam e agilizam muito o desenvolvimento, e são comumente usados para desenvolvimento frontend relacionado a blockchain.

A fim de manter as coisas simples, não utilizaremos nenhum framework frontend JavaScript nesta seção, já que o objetivo principal é focar na tecnologia blockchain e não em frameworks de UI em HTML, CSS ou JavaScript.

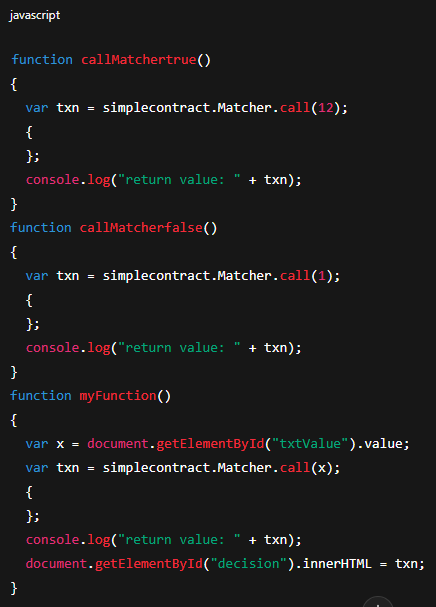
Nesta parte do exemplo, criamos um frontend web e um backend em JavaScript, onde definimos todas as funções exigidas para nossa aplicação.

O arquivo app.js que criamos na seção anterior é o principal arquivo JavaScript que contém o código para criar um objeto web3. Ele também fornece métodos que são usados para interagir com o contrato na blockchain.

Na próxima etapa deste exemplo, exploraremos como as funções do contrato podem ser chamadas.

**Chamando funções do contrato**

Funções de contrato podem ser chamadas como mostrado no seguinte código, que faz parte do arquivo app.js criado anteriormente:



O código acima mostra três funções simples: callMatchertrue(), callMatcherfalse() e myFunction().

**Criando uma página web frontend**

Por fim, o arquivo HTML chamado index.html é criado com o seguinte código. Este arquivo HTML servirá como a interface de usuário (UI) frontend para os usuários, que poderão acessá-lo via um servidor HTTP para interagir com o contrato inteligente:



Recomenda-se que um servidor web esteja em execução para servir o conteúdo HTML (por exemplo, index.html). Alternativamente, o arquivo pode ser acessado diretamente do sistema de arquivos, mas isso pode causar alguns problemas relacionados à forma correta de servir o conteúdo em projetos maiores; como boa prática, sempre use um servidor web.

Um servidor web em Python pode ser iniciado usando o seguinte comando. Este servidor servirá o conteúdo HTML a partir do mesmo diretório onde ele foi iniciado:

$ python -m SimpleHTTPServer 7777

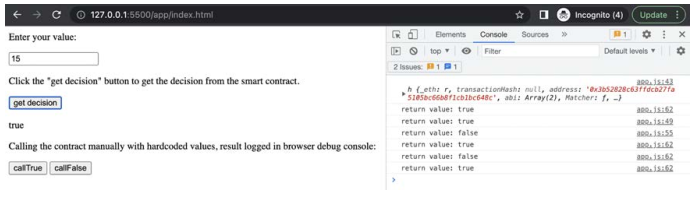
Serving HTTP on 0.0.0.0 port 7777

O servidor web não precisa ser em Python; pode ser um servidor Apache ou qualquer outro contêiner web.

Se você estiver usando o Visual Studio Code, pode usar a extensão *Live Server*, disponível em:  
<https://marketplace.visualstudio.com/items?itemName=ritwickdey.LiveServer>

**Nota:** Não carregue o arquivo HTML diretamente no navegador — isso não funcionará. Você precisa de um servidor web, que pode ser executado como mostrado acima.

Agora qualquer navegador pode ser usado para visualizar a página web servida sobre a porta TCP escolhida. Isso é mostrado na captura de tela a seguir:



**Figura 12.3**: Interação com o contrato

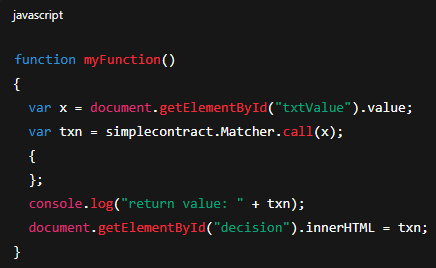
Observe que a saída mostrada aqui aparece na janela do console do navegador. O console do navegador deve estar habilitado para que seja possível visualizar a saída.

Como os valores (12 e 1) estão codificados no código por simplicidade, dois botões mostrados na captura de tela — **callTrue** e **callFalse** — foram criados no index.html. Ambos os botões chamam funções com valores fixos. Isso é apenas para demonstrar que os parâmetros estão sendo passados para o contrato via Web3 e os valores estão sendo retornados adequadamente.

Três funções estão sendo chamadas por trás desses botões. Vamos descrevê-las a seguir:

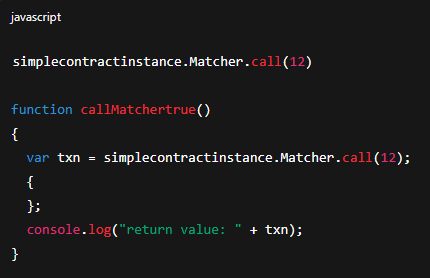
1. O botão **get decision** retorna a decisão do contrato:



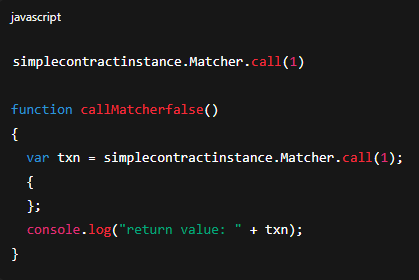


O botão **get decision** invoca a função Matcher do contrato inteligente com o valor inserido na página web. A variável x contém o valor passado para essa função via página web, que é 12. Como o valor é 12 — maior que 10 — o botão **get decision** retornará true.

1. O botão **callTrue** chamará a função Matcher com um valor que é sempre maior que 10, como 12, retornando sempre true. O método callMatchertrue() tem um valor fixo de 12, que é enviado ao contrato usando o seguinte código:



O botão **callFalse** invoca a função callMatcherfalse(). A função callMatcherfalse() funciona passando um valor fixo de 1 para o contrato usando este código:



Observe que não há necessidade real dos métodos callTrue e callFalse aqui; eles são apresentados apenas por razões pedagógicas para que os leitores possam correlacionar as funções com os valores fixos e, em seguida, com a função chamada dentro do contrato inteligente, com value como parâmetro.

Este exemplo demonstra como a biblioteca Web3 pode ser usada para interagir com os contratos na blockchain Ethereum. Primeiro, criamos um frontend web usando o arquivo JavaScript app.js e o arquivo HTML. Também incluímos a biblioteca Web3 em nosso HTML para que pudéssemos criar o objeto web3 e usá-lo para interagir com o contrato inteligente implantado.

**Implantando e interagindo com contratos usando Truffle**

Existem vários frameworks de desenvolvimento disponíveis para Ethereum. Como visto nos exemplos discutidos anteriormente, pode ser bastante demorado implantar os contratos por meios manuais. É aí que o Truffle e frameworks semelhantes como o Embark podem ser usados para tornar o processo mais simples e rápido. Escolhemos o Truffle porque ele possui uma comunidade de desenvolvedores mais ativa e atualmente é o framework mais amplamente usado para desenvolvimento Ethereum. No entanto, observe que não há um “melhor” framework, já que todos eles visam fornecer métodos para facilitar o desenvolvimento, teste e implantação.

Você pode explorar outros frameworks e ferramentas aqui:  
<https://ethereum.org/en/developers/local-environment/>

Discutimos o Truffle brevemente no Capítulo 11, *Ferramentas, Linguagens e Frameworks para Desenvolvedores Ethereum*. Nesta seção, veremos um projeto de exemplo que demonstrará como o Truffle pode ser usado para desenvolver uma aplicação descentralizada. Veremos todas as etapas envolvidas neste processo, como inicialização, teste, migração e implantação. Primeiro, veremos o processo de instalação.

**Instalando e inicializando o Truffle**

Se o Truffle ainda não estiver instalado, ele pode ser instalado executando o seguinte comando:

$ npm install -g truffle

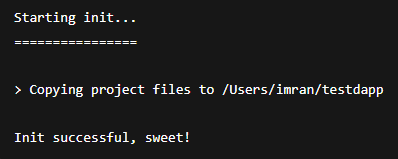
Em seguida, o Truffle pode ser inicializado executando os comandos a seguir. Primeiro, crie um diretório para o projeto, por exemplo:

$ mkdir testdapp

Depois, mude para o diretório recém-criado testdapp e execute o seguinte comando:

$ truffle init

Saída esperada:



Uma vez que o comando seja bem-sucedido, ele criará a estrutura de diretórios, como mostrado abaixo. Isso pode ser visualizado usando o comando tree no Linux:



Esse comando cria três diretórios principais:

* **contracts**: Este diretório contém os arquivos de código-fonte Solidity dos contratos. É aqui que o Truffle buscará os arquivos durante a migração.
* **migrations**: Este diretório contém todos os scripts de implantação.
* **test**: Como o nome sugere, este diretório contém arquivos de teste relevantes para os aplicativos e contratos.

Por fim, a configuração do Truffle é armazenada no arquivo truffle.js (ou truffle-config.js), que é criado na pasta raiz do projeto, onde o comando truffle init foi executado.

Agora que inicializamos o Truffle, vamos ver como ele é usado para compilar, testar e migrar contratos inteligentes.

**Compilando, testando e migrando com o Truffle**

Nesta seção, demonstraremos como usar várias operações disponíveis no Truffle. Aprenderemos como um projeto de exemplo disponível com o Truffle pode ser baixado, e introduziremos como usar os comandos de compilação, teste e migração no Truffle para implantar e testar Truffle boxes, que são essencialmente projetos de exemplo prontos para uso. Usaremos o *MetaCoin Truffle box*. Posteriormente, mais exemplos serão mostrados sobre como usar o Truffle para projetos personalizados.

Usaremos o **Ganache** como blockchain local para fornecer a interface RPC. Certifique-se de que o Ganache está em execução em segundo plano e minerando.

Cobrimos a configuração do Ganache no Capítulo 11, *Ferramentas, Linguagens e Frameworks para Desenvolvedores Ethereum*. Você pode consultar esse capítulo para revisar.

O Ganache normalmente roda na porta 7545 com 10 contas, porém essas escolhas podem ser modificadas nas configurações do Ganache.

Usaremos o workspace do Ganache salvo no Capítulo 11. Alternativamente, também pode-se configurar um novo ambiente.

Depois de configurar com sucesso o Ganache, as seguintes etapas precisam ser realizadas para descompactar o *webpack Truffle box* e executar o projeto *MetaCoin*:

1. Primeiro, crie um novo diretório:

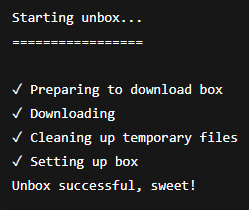
$ mkdir tproject

$ cd tproject

Agora, descompacte o exemplo metacoin do Truffle, emitindo o comando abaixo:

$ truffle unbox metacoin

Saída esperada:



**3. Edite o arquivo truffle.js ou truffle-config.js:**

Se o Ganache estiver rodando em uma porta diferente, altere a porta padrão para a porta na qual o Ganache está escutando.

Edite o arquivo e descomente os valores padrão. O arquivo deve ficar semelhante ao mostrado a seguir:

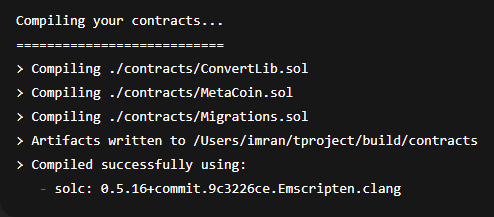


Agora, após descompactar o exemplo *webpack* e fazer as mudanças de configuração necessárias, estamos prontos para compilar todos os contratos.

**4. Agora execute o seguinte comando para compilar todos os contratos:**

$ truffle compile

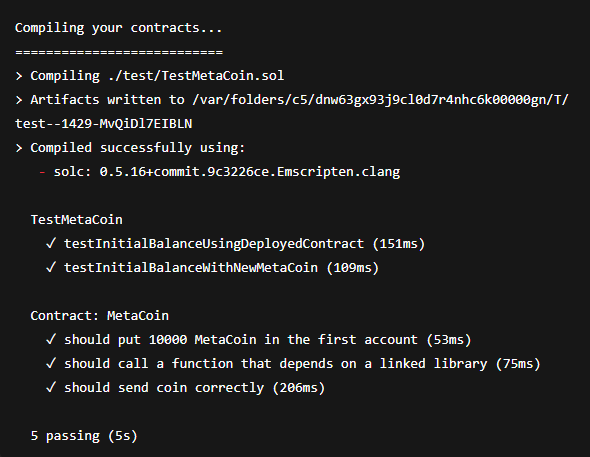
Saída esperada:



**5. Agora podemos testar os contratos usando o comando truffle test, como mostrado abaixo:**

$ truffle test

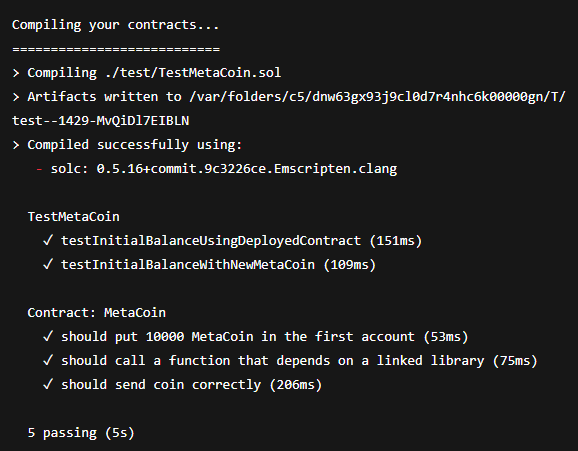
Esse comando produzirá a seguinte saída, indicando o progresso do processo de testes:



**5. Agora podemos testar os contratos usando o comando truffle test, como mostrado abaixo:**

$ truffle test

Esse comando produzirá a seguinte saída, indicando o progresso do processo de testes:

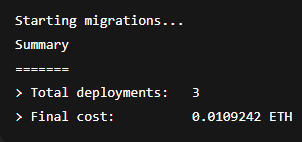


**6. Após a conclusão dos testes, podemos migrar para a blockchain usando o comando abaixo.**  
A migração usará as configurações disponíveis no truffle.js (ou truffle-config.js) que editamos no segundo passo, apontando para a porta onde o Ganache está rodando.

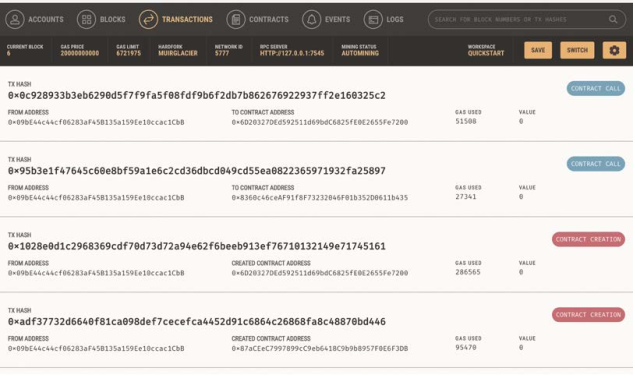
Execute:

$ truffle migrate

Saída esperada (resumo):



Observe que, quando a migração é executada, ela se reflete no Ganache; por exemplo, o saldo das contas diminuirá, e você poderá visualizar as transações que foram executadas. É possível ver o **BALANCE** sendo atualizado no Ganache conforme as transações são processadas e o ETH é consumido.



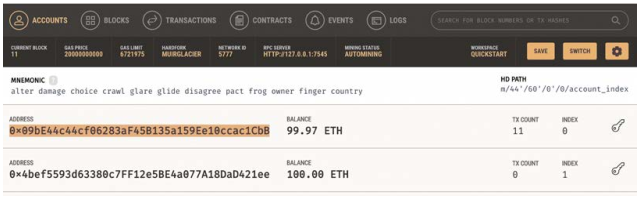
**Figura 12.4**: Tela do Truffle como resultado dos testes

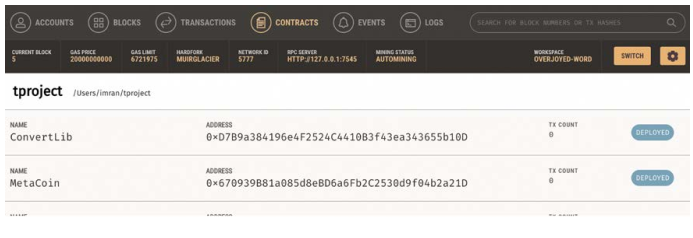
**7. Integração com o Ganache**

No workspace do Ganache, também podemos adicionar o projeto Truffle para habilitar recursos extras. Para isso:

* Clique no ícone de engrenagem no canto superior direito (Configurações)
* Abra a tela de configurações (*Settings*)
* Adicione o projeto Truffle selecionando o arquivo truffle-config.js no diretório tproject

Salve e reinicie para aplicar as mudanças. Quando o Ganache estiver de volta, você poderá ver informações adicionais sobre o contrato, como mostrado na captura de tela a seguir:

**Figura 12.5**: Ganache exibindo contas

**Figura 12.6**: Mais detalhes do contrato visíveis após adicionar o projeto Truffle

**Interagindo com o contrato**

O Truffle também fornece um console (uma CLI) que permite a interação com os contratos. Todos os contratos implantados já estão instanciados e prontos para uso no console. Trata-se de uma interface baseada em REPL (Read, Evaluate, Print, Loop). De forma semelhante ao cliente Geth (via attach ou console), o Truffle também expõe o ambiente de execução JavaScript (JSRE).

**1.** O console pode ser acessado emitindo o seguinte comando:

$ truffle console

Isso abrirá uma CLI:

truffle(development)>

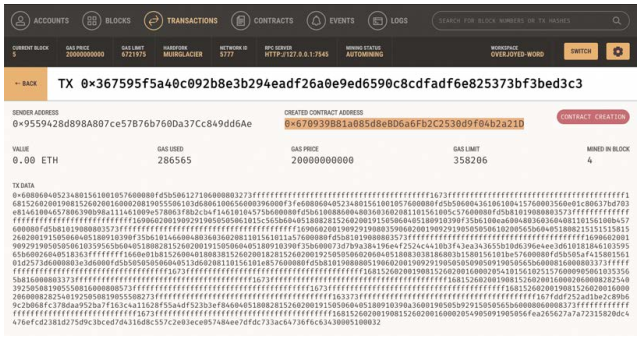
Uma vez dentro do console, diversos métodos podem ser executados para consultar o contrato. Uma lista de métodos pode ser exibida digitando um comando parcial seguido de **tab** para autocompletar.

**3.** Outros métodos também podem ser chamados para interagir com o contrato. Por exemplo, para recuperar o endereço do contrato, o seguinte comando pode ser usado no console do Truffle:

truffle(development)> MetaCoin.address

'0x670939B81a085d8eBD6a6Fb2C2530d9f04b2a21D'

**4.** Este endereço também é exibido na transação de criação do contrato no Ganache:



**Figura 12.7**: Transação de criação do contrato exibida no Ganache

5. Para consultar as contas disponíveis, digite o seguinte comando:

truffle(development)> MetaCoin.web3.eth.getAccounts()

6. Isso retornará a seguinte saída:



7. Para consultar o saldo da conta, insira o seguinte comando:

truffle(development)> MetaCoin.web3.eth.getBalance("0x9559428d898A807ce57B76b760Da37Cc849dd6Ae")

8. Isso mostrará a saída:

99987682160000000000

9. Para encerrar uma sessão no console Truffle, use o comando:

.exit

Isso conclui nossa introdução ao *webpack Truffle box* de exemplo e à aplicação MetaCoin usando o Truffle. Nesta seção, descobrimos como o Truffle pode ser usado para interagir com contratos inteligentes implantados. O MetaCoin, nesta seção, é um exemplo de uma aplicação descentralizada, contudo, usamos esse exemplo apenas para aprender como o Truffle funciona.

**Na próxima seção, veremos como um contrato pode ser desenvolvido do zero, testado e implantado usando Truffle, Ganache e nossa rede privada.**

**Usando Truffle para testar e implantar contratos inteligentes**

Esta seção demonstrará como podemos usar o Truffle para testar e implantar contratos inteligentes. Vamos observar um exemplo de um contrato simples em Solidity, que realiza uma adição. Veremos como migrações e testes podem ser criados para este contrato com os seguintes passos:

1. **Crie um diretório chamado simple:**

$ mkdir simple

1. Mude para o diretório simple:

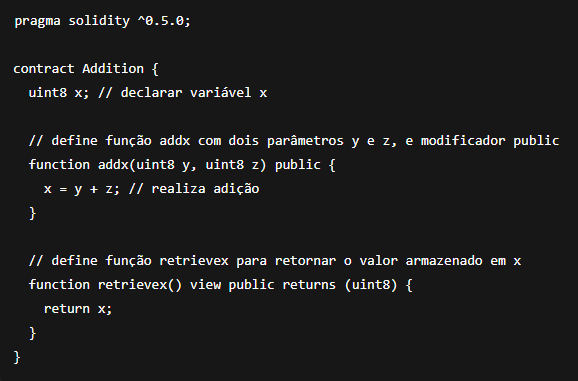
$ cd simple

1. Inicialize o Truffle para criar uma estrutura de diretório para desenvolvimento de contratos:

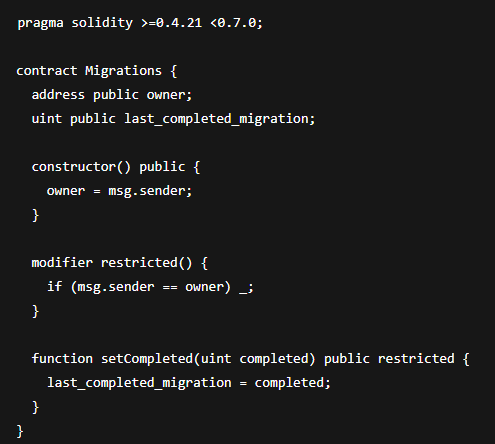
$ truffle init

**4. Após a inicialização, coloque os dois arquivos Addition.sol e Migrations.sol dentro do diretório contracts.**  
O código para ambos os arquivos é listado a seguir:

**Addition.sol:**



Migrations.sol:

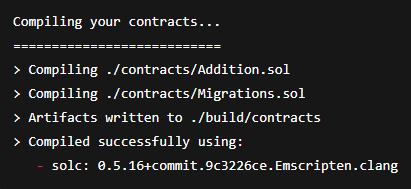


**5. Compile os contratos**

Depois de colocar Addition.sol e Migrations.sol no diretório contracts, compile os contratos com o comando:

$ truffle compile

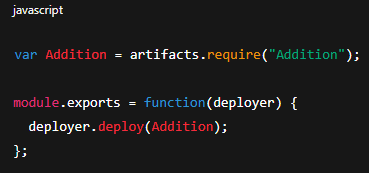
A saída será semelhante a esta:



**6. Crie arquivos de migração**

O Truffle requer arquivos de migração para implantar contratos na blockchain. Crie um arquivo de migração no diretório migrations com o nome 2\_deploy\_contracts.js.

Aqui está o conteúdo de 2\_deploy\_contracts.js:



Este script diz ao Truffle para implantar o contrato Addition.

**7. Crie arquivos de teste**

Crie um arquivo de teste chamado test\_add.js no diretório test com o seguinte conteúdo:



Este teste usa o Mocha, framework de teste JavaScript, com Chai para asserções. Ele faz o seguinte:

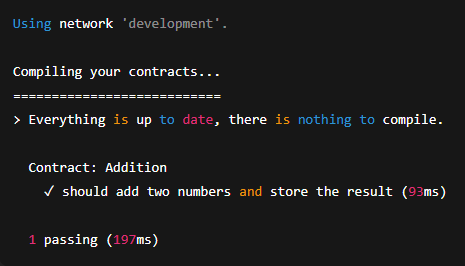
* Obtém a instância do contrato Addition já implantado.
* Chama a função addx com os valores 10 e 5.
* Em seguida, chama retrievex() para recuperar o resultado.
* Verifica se o valor armazenado é 15.

**8. Execute os testes**

Com o Ganache em execução e o ambiente de desenvolvimento configurado, execute o seguinte comando:

$ truffle test

Saída esperada:



Isso indica que o contrato foi testado com sucesso.

**Considerações sobre o uso de Truffle**

Truffle é um dos frameworks mais usados no desenvolvimento Ethereum. Ele fornece:

* Um pipeline de compilação robusto
* Um sistema de migração e implantação
* Suporte integrado para testes com Mocha/Chai
* Console interativo para depuração
* Integração com ferramentas como Ganache e Drizzle

Ele facilita o desenvolvimento, teste e implantação de contratos inteligentes em ambientes locais, de teste (testnets) e produção (mainnet).

**Outras ferramentas e frameworks**

Além do Truffle, outras ferramentas e frameworks úteis no ecossistema Ethereum incluem:

* **Ganache**: um ambiente local para desenvolvimento e testes rápidos
* **Remix**: IDE online para desenvolver, compilar e testar contratos inteligentes
* **Embark**: outro framework completo para desenvolvimento de DApps
* **Hardhat**: alternativa moderna ao Truffle, com foco em flexibilidade e velocidade
* **OpenZeppelin**: biblioteca de contratos reutilizáveis, incluindo padrões ERC
* **Drizzle**: biblioteca JavaScript voltada para gerenciamento de estado de DApps integradas com React

**Conclusão**

Este capítulo começou com a introdução à Web3. Exploramos vários métodos para desenvolver contratos inteligentes. Além disso, vimos como o contrato pode ser testado e verificado usando uma blockchain de teste local antes da implementação em uma blockchain pública ou em uma blockchain de produção privada. Trabalhamos com várias ferramentas, como Ganache, o console do cliente Geth e o IDE Remix, para desenvolver, testar e implantar contratos inteligentes. Além disso, o framework Trufe também foi usado para testar e migrar contratos inteligentes. Também exploramos como o IPFS pode ser usado para hospedar as páginas da web que criamos para nosso DApp, servindo como a camada de armazenamento descentralizada do ecossistema blockchain. No próximo capítulo, apresentaremos como o Ethereum está evoluindo para uma forma mais avançada, anteriormente chamada de Ethereum 2.0, mas agora simplesmente Ethereum – a combinação da camada de execução (Eth1) e da camada de consenso (Eth2).

Neste capítulo, aprendemos como interagir com contratos inteligentes Ethereum usando várias ferramentas:

* Interação via **console do Geth**
* Requisições **JSON-RPC** usando curl
* Criação de **interfaces web frontend** com HTML e web3.js
* Uso do **framework Truffle** para compilar, migrar e testar contratos
* Visualização e monitoramento com o **Ganache**

Cobrimos também como:

* Criar e implantar contratos com Truffle
* Escrever arquivos de migração e testes
* Usar ferramentas modernas para acelerar o desenvolvimento

Estas ferramentas e práticas formam a base para o desenvolvimento de aplicações descentralizadas modernas (DApps) no ecossistema Ethereum. Elas ajudam os desenvolvedores a manter eficiência, segurança e controle no ciclo completo de vida de desenvolvimento de contratos inteligentes.

<imagem\_12.8>

**Figura 12.8**: Visão geral das ferramentas de desenvolvimento Ethereum