

O'REILLY®

Hands-On Smart Contract Development with Solidity & Ethereum

From Fundamentals to Deployment



Kevin Solorio,
Randall Kanna
& David H. Hoover



Hands-On Smart Contract Development with Solidity and Ethereum

Ready to dive into smart contract development for the blockchain? With this practical guide, experienced engineers and beginners alike will quickly learn the entire process for building smart contracts for Ethereum—the open source blockchain-based distributed computing platform. You'll get up to speed with the fundamentals and quickly move into builder mode.

Kevin Solorio, Randall Kanna, and Dave Hoover show you how to create and test your own smart contract, create a frontend for users to interact with, and more. It's the perfect resource for people who want to break into the smart contract field but don't know where to start.

In four parts, this book helps you:

- Explore smart contract fundamentals, including the Ethereum protocol, the Solidity programming language, and the Ethereum Virtual Machine
- Dive into smart contract development using Solidity and gain experience with Truffle framework tools for deploying and testing your contracts
- Use Web3 to connect your smart contracts to an application so users can easily interact with the blockchain
- Examine smart contract security along with free online resources for smart contract security auditing

"This book is the perfect place for the aspiring smart-contract developer to start. The hands-on approach will empower anyone trying to wrap their head around the many ideas and technologies needed to build a distributed application."

—Destry Saul
Senior Engineer, Unchained Capital, Inc.

Kevin Solorio has been developing applications on web and mobile platforms for nearly 10 years. He also taught and wrote curriculum for Dev Bootcamp.

Randall Kanna started coding at the age of 12 and has worked as a Senior Software Engineer at companies such as Eventbrite and Pandora.

David H. Hoover is the founder and CEO of Red Squirrel Technologies. A self-taught software engineer, he's also cofounder of Dev Bootcamp.

CRYPTOCURRENCY

US \$59.99

CAN \$79.99

ISBN: 978-1-492-04526-7



Twitter: @oreillymedia
facebook.com/oreilly

Hands-On Smart Contract Development with Solidity and Ethereum

From Fundamentals to Deployment

Kevin Solorio, Randall Kanna, and David H. Hoover

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Hands-On Smart Contract Development with Solidity and Ethereum

by Kevin Solorio, Randall Kanna, and David H. Hoover

Copyright © 2020 Kevin Solorio, Randall Kanna, and David H. Hoover. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editor: Michelle Smith

Production Editor: Kristen Brown

Copypeditor: nSight, Inc.

Proofreader: Charles Roumeliotis

Indexer: Ellen Troutman-Zaig

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

December 2019: First Edition

Revision History for the First Edition

2019-11-25: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781492045267> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Hands-On Smart Contract Development with Solidity and Ethereum*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-04526-7

[LSI]

Table of Contents

Preface.....	ix
--------------	----

Part I. Intro to the Ethereum Blockchain

1. Blockchain Concepts.....	3
A Brief History	4
The Character of a Blockchain	5
Decentralized Networks	5
Consensus Protocols	6
Transaction Processing	9
Transaction Finality	11
Hard Forks	13
Ethereum Fundamentals	14
Ether and Gas	14
Accounts	17
Contracts	17
Blocks and Transactions	18
What Time Is It?	19
Signing Transactions	21
Summary	22
2. Decentralized Applications.....	23
Tokens	23
ERC-20	23
Non-Fungible Token (ERC-721)	25
Supply Chain	26
Permanent Records	28

Evaluating Blockchain for Your Application	29
Verifiable	29
Transparent	29
Resilient	29
Summary	30
3. Before We Get Started.....	31
Ethereum Clients	31
Installing Parity	32
Installing MetaMask	33
Installing Node.js	42
Ubuntu Installation (Including Windows WSL Ubuntu)	42
Mac Installation	43
Installing the Truffle Suite	44
Truffle	44
Ganache	45
Summary	46
<hr/>	
Part II. Developing Smart Contracts	
4. Our First Smart Contract.....	49
Setup	49
Our First Test	50
Saying Hello	54
Making Our Contract Dynamic	56
Making the Greeter Ownable	62
Summary	69
5. Deploying and Interacting with Contracts.....	71
Contract Compilation and Deployment	71
Deployment Process	72
Setting Up the UI	73
Deploying to Ganache	74
Deploying to Goerli with Parity	80
Deploying to Rinkeby with Infura	84
Summary	86
6. The Fundraiser Application.....	87
Application Overview	87
Creating the Project	91
Initializing Fundraisers	93

Setting the Beneficiary and Custodian	101
Editing the Beneficiary	104
Making Donations	109
Structs	111
Mappings	112
Donation Tests	112
myDonations	116
Fundraiser Totals	119
Events	124
Withdrawing Funds	126
Fallback Functions	131
Summary	134
7. FundraiserFactory.....	135
Migrating Our FundraiserFactory	135
Creating Fundraisers	137
Viewing Available Fundraisers	142
Testing Pagination When Empty	142
Testing the Limit	144
Testing the Offset	147
Setting Up the UI	151
Summary	152
<hr/>	
Part III. Interacting with Our Smart Contracts Through Web3	
8. What Is Web3?.....	155
The Frontend, Web3, and the Blockchain	155
Web3 Methods	156
getAccounts()	156
getBlockNumber()	156
getBalance()	156
sendTransaction()	157
Providers	158
Promises with Web3	158
MetaMask for Web3 Injection	160
Send (State Updates/Write)	160
Call (Reads)	160
Summary	161
9. Connecting the UI to Our Contracts.....	163
Why React?	163

Dev Setup	164
Truffle	165
React Truffle Box Setup	166
Importing our Greeter Contracts	172
Adding Our Greeter Contract Functionality	173
Setting Our Greeting Through React	175
Summary	176
10. Our Larger DApp.....	179
Starting with React Truffle Box	179
Fundraiser Setup	182
React Routing	185
React and Material UI	189
Creating Our New Fundraiser Page View	191
Displaying the Current Fundraisers List	195
Summary	201
11. Finishing Our Fundraiser UI.....	203
Adding Detailed Information About Each Fundraiser	203
Implementing a Material UI Dialog	205
Accepting a Donation with React	208
Testing a Donation	212
Displaying the Current ETH Exchange Rate	214
Generating a Donation Receipt for Our User	218
Handling a Withdrawal from Our Contract	224
Adding Edit Beneficiary to the Contract	230
Summary	231

Part IV. Securing Your Smart Contracts

12. Smart Contract Security.....	235
Why Do We Need to Worry About Security?	235
Types of Smart Contract Vulnerabilities	236
Unprotected Function	236
Transaction Ordering Dependence	236
Integer Overflow and Underflow	237
Reentrancy	237
Block Gas Limit	238
Timestamp Dependence	238
And Many More	239
Preparing Your Contract for an External Audit	239

External Auditing	240
Auditing Companies	240
Solidified	240
Free Auditing Resources	240
Growing Your Auditing Skills	241
Summary	241
Index.....	243

Preface

We wrote this book to make smart contract development more accessible and easy to understand for beginners looking to explore developing on the blockchain.

It can be daunting when you have terms such as networks, nodes, and forks being thrown around. This book will help you learn how to create and test your own smart contract, create a frontend for users to interact with, and more. We wrote this book because we want to provide a resource for people who want to break into the field but don't know where to start. We've had so many people ask us how they can get started and tell us how overwhelmed they feel about diving into the blockchain. Even engineers with years of experience don't know where to start when trying to dive into smart contract development.

This book will guide you through the entire process of building a smart contract like you would in the real world, and deploying it to help users interact with it through an application.

This book is for anyone who wants to dive into blockchain development and get hands-on experience writing smart contracts. You should have an editor as well as some basic engineering knowledge like how to start a terminal.

There are various ways you can get involved in the community and learn more about smart contract development beyond this book. Here are a few to check out.

Mastering Ethereum

O'Reilly has another book, *Mastering Ethereum* by Andreas Antonopoulos and Gavin Wood, which is a deep dive on Ethereum. While this book focuses on getting hands-on experience with smart contract development, *Mastering Ethereum* will provide you with a deeper dive into topics like cryptography, wallets, transactions, and more.

ConsenSys Academy

ConsenSys, the global blockchain company, offers a course on Ethereum development. The course is available twice a year and the company hires from within the program. The 11-week course offers active support from instructors and a connection with the rest of the cohort.

The nice thing about the ConsenSys course is that it offers a certification upon completion of the quizzes and the final project. The top students in the cohort are also considered for employment inside ConsenSys.

B9 Academy

If you're just starting out with Ethereum and want to learn more, B9 Academy offers a [free Ethereum course](#) available to anyone. Introduction to Ethereum for Developers gives you an overview of the basics of Ethereum. The course will walk through an introduction to Ethereum development such as how private keys work, what a merkle tree is, what the differences are between the types of networks, and more. While no certification is offered for the free course, this is a great way to dip your toes into Ethereum development and determine if a paid course might be for you.

Another option is the paid course from B9, which is one of the best online courses on Ethereum currently available. The course has extensive content and offers a [certificate](#) upon completion. You'll build three projects and have access to the B9 Slack, which is a great resource for connecting with other people in the community and getting help with Ethereum questions or projects. The course also offers instructors for tutoring and mentorship throughout the duration of the course. If you do well on the final exam, you'll receive the B9 certification.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.



This element signifies a tip or suggestion.



This element signifies a general note.



This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at <https://github.com/RedSquirrelTech/hoscdev>.

If you have a technical question or a problem using the code examples, please send an email to bookquestions@oreilly.com.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Hands-On Smart Contract Development with Solidity and Ethereum* by Kevin Solorio, Randall Kanna, and David H. Hoover (O'Reilly). Copyright 2020 Kevin Solorio, Randall Kanna, and David H. Hoover, 978-1-492-04526-7."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

O'Reilly Online Learning



For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit <http://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <https://oreil.ly/handsOn-smart-contract-dev-solidity-ethereum>.

Email bookquestions@oreilly.com to comment or ask technical questions about this book.

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

Acknowledgments

We would like to thank our editor, Melissa Potter from O'Reilly, for all the feedback and guidance as we were writing this book. We would also like to thank our tech reviewers, Brian Wu, John Mardlin, and Destry Saul, for their detailed feedback and recommendations on how to improve our examples. Thanks also go to our earlier reviewers Alina Chu, Phillip Birtcher, and Corey Haines for their insights as we were just getting started in our writing journey.

Kevin would like to thank his partner Alyssa for all the support and encouragement during crunch time. He would also like to thank his mother Peggy and brother Luis for always encouraging him to take chances and pursue his dreams.

Randall would like to thank her partner Jack for supporting her in everything she does. Always. And her family of authors who inspired her to write this book and always have grit. And her Aunt Kat, who always gives encouragement and support at any hour of the day and any day of the week.

Dave would like to thank Kevin and Randall for working valiantly to finish this book! Also thanks to Brian Forde for nudging him down the blockchain rabbit hole, and Conrad Barski for his mentoring, leadership, and camaraderie. Finally, he would like to thank his partner Heather Corallo for her encouragement and insights.

PART I

Intro to the Ethereum Blockchain

This book is narrowly focused on helping you learn to develop smart contracts on Ethereum. This focus requires that we cover a subset of the Ethereum protocol, the Solidity programming language, and the Ethereum Virtual Machine. For a deeper and more comprehensive understanding of Ethereum, we recommend *Mastering Ethereum* by Andreas M. Antonopoulos and Gavin Wood. That said, in order to develop smart contracts effectively, it's critical that you understand certain fundamental smart contract concepts. We will cover those concepts in barely sufficient detail so that we don't waste any time getting you into builder mode. We want you deploying your first smart contracts as soon as possible. We know this approach is the most reliable way for people to learn new technologies.

CHAPTER 1

Blockchain Concepts

Fundamentally, a blockchain is a data structure. It is a linked list, or chain, of unique “blocks.” Each block points to the previous one, and is itself a list of transactions. On top of this relatively simple list-of-lists data structure is laid the key innovation that blockchains have given us: a protocol for how blocks are added to the chain without any central authority. [Figure 1-1](#) illustrates this structure.

The cryptocurrencies that have come along with blockchains are a means to an end, providing incentives for people to run software that secures the network. For the first time in history, we have the ability to share digital information without trusting any person, government, organization, or corporation to facilitate the interaction. Ethereum provides a cryptographically secure platform for storing, updating, and removing data from a blockchain using what is referred to as “smart contracts.” We are still in the early days of learning how to use smart contracts to improve things in the real world, and it’s hard to predict how the technology will be used in the future. Similar to the “World Wide Web” in the 1990s, there has been a global influx of inspiring and creative problem-solvers who are working every day to deploy decentralized applications (DApps) that they hope will “make a dent in the universe.”

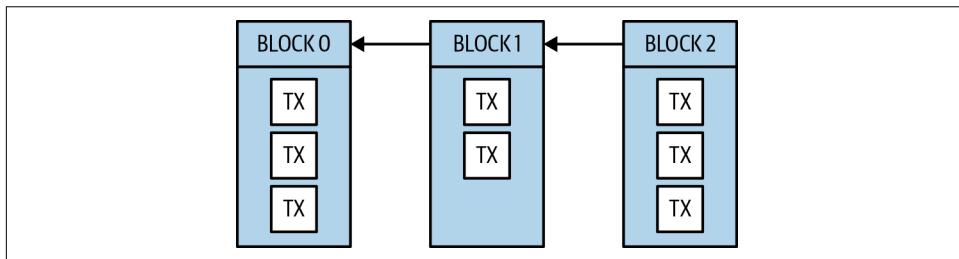


Figure 1-1. The structure of a blockchain

We will be spending most of this book focused on the programming language Solidity and smart contract development. Solidity is a popular programming language for developing smart contracts, and was designed to run on the Ethereum Virtual Machine (EVM). Solidity wasn't the first programming language to run on the EVM, and it certainly isn't the last. Many other languages, such as Vyper, will be written to run on the EVM, attempting to improve on Solidity's design or provide powerful domain-specific languages. Before we dig into the revolutionary world of smart contracts, we need to lay a conceptual foundation for you to build on. Due to the historically unique nature of running code on a blockchain, it's critical for developers to have a sufficient understanding of how the pieces fit together underneath the powerful abstractions that Ethereum provides.

A Brief History

The concept of a “blockchain” was born out of the [Bitcoin white paper](#) published in 2008 by the pseudonymous Satoshi Nakamoto. Although the term “blockchain” doesn't actually appear in the paper, the concept was concisely articulated. Transactions of value exchange enter a peer-to-peer network, and are periodically grouped into “blocks,” or lists. When a “block” of transactions is persisted, it is “chained” to the previous block. This append-only data structure and the protocol that constructs it create an immutable record of transactions.

Bitcoin's launch in early 2009 marked the beginning of public blockchain networks. Since then, countless cryptocurrencies have attempted to build on Bitcoin's success as a new form of currency. Many early adopters of Bitcoin realized that the properties of a blockchain had applications beyond financial transactions. Communities spawned to try to extend, fork, and build on top of Bitcoin in order to stretch it in directions that were not previously conceived. Ultimately, though, the Bitcoin protocol is purposely constrained and ill-fitted for extension. Blockchain prodigy Vitalik Buterin made the ambitious decision to stop trying to extend Bitcoin and instead create a more general-purpose protocol from scratch. In 2013, Vitalik wrote the [Ethereum white paper](#).

When Ethereum launched in 2015, it quickly became one of the most valuable cryptocurrencies on the planet, second only to Bitcoin. The markets valued Ethereum because it provided a platform for deploying and running smart contracts on a public blockchain. The term “smart contract” was coined by [Nick Szabo in 1994](#). The idea then was that many legal contracts, notaries, and other analog agreements could be enforced nearly automatically using digital protocols and cryptographic signatures. Despite this historical context, the implementation of smart contracts on Ethereum actually feels more like general-purpose programming than anything specific to legal contracts. The virtual machine as defined by the Ethereum protocol is Turing-complete. This means that as long as you can fit your computations within the

limitations of a single block, smart contract developers have few other constraints to contend with beyond their own imagination.

How is a blockchain different from technologies you might have worked with before? In the next section, let's discuss what makes the blockchain unique.

The Character of a Blockchain

Many software developers will have worked with a technology stack that includes 1) a native mobile user interface and/or web user interface 2) with a server-side programming language that ultimately interacts with 3) a database. In the most basic versions of these systems, the interactions with the database are essentially instantaneous and permanent.

Like a more typical database, blockchains can store arbitrary data, but they share few other similarities. Becoming a competent smart contract developer means understanding the character of a blockchain. One cannot simply treat Solidity like a server-side programming language; you will quickly find yourself lost and frustrated. Unlike typical databases, interactions with even the most basic smart contract systems are not instantaneous and are not guaranteed to be permanent.

Unlike a typical database, which is a single program running on one computer, a blockchain is typically made up of many nodes in a worldwide network. When we refer to “nodes” in the context of a blockchain, we are referring to software that someone has installed on a computer and connected to the blockchain network. Just like there are many different software implementations of the HTTP protocol (Apache, NGINX) called “web servers,” there are many software implementations of the Ethereum protocol (geth, Parity) called Ethereum nodes.

Next, let's discuss how we can use a network to connect with a blockchain.

Decentralized Networks

In order to work with a typical database, we need a database connection and sufficient rights to update the database. In the simplest cases there's a single database to contend with, so we need a single IP address to make the connection. The entire system relies on that database being available. If the database were to crash or lose connectivity, or if our rights were revoked, the application stops working. This is what is referred to as a centralized system.

Blockchains were designed to run over decentralized networks. People and companies run nodes in the network, and, in many respects, all nodes are peers. This is possible because every node contains the full history of every transaction that ever occurred on the blockchain. With every node as a self-sufficient, independent entity, there is no center of the network. All nodes validate transactions and propagate them

to their peers. On top of this, some nodes also participate in the block creation process, and are financially incentivized to do so via receiving a “block reward” in the blockchain’s native cryptocurrency. Unlike our database example, any node in a decentralized network can join or drop at will. No special permission or rights are needed to read or write to the blockchain as long as the protocol is followed.

Without a single, centralized database to connect to, how do nodes find a connection to the network? Many smart contract developers punt on this issue and use a service that hides this complexity behind a web API. That is a tradeoff between coupling your system to a third-party service and the complexity of running your own Ethereum node. If you choose to run your own node, it will need to pull itself up by its bootstraps and connect to the public Ethereum network. Every major Ethereum node software has a hardcoded list of “bootnodes.” These are well-known, relatively reliable IP addresses that can be used to create a sufficiently large pool of network peer connections. If for some reason those nodes aren’t available or are no longer trustworthy, the node software can be given a custom list of bootnodes to use.

In 2019, the global, public Ethereum network had more nodes than any other blockchain network. With over 12,000 nodes, it’s unlikely that any two nodes will have the exact same set of peers. As a transaction enters the network through a single peer, it spreads quickly around the world to every node. Similarly, when a new block is added to the blockchain, news of this addition spreads quickly. Unfortunately, quickly is not the same as instantaneously, meaning that when two blocks are created at the same time by two different nodes, the respective peers of those diverging nodes will be operating on two different versions of the blockchain. This is called a temporary fork. In general, temporary forks are resolved by granting precedence to whichever fork adds the next block, summarized as “longest chain wins.” These rules are determined by a blockchain’s consensus protocol.

Consensus Protocols

Without a central source of truth, the nodes in a blockchain network need a way to come to consensus on the state of the system. Consensus protocols are how this is accomplished, and are an active area of research. A consensus protocol is simply a system of agreement across a blockchain. There are three protocols that we need to cover, and we’ll start with Proof-of-Work, which Ethereum used when they launched their blockchains.

Proof-of-Work

When you hear about “mining ether” or “cryptomining,” this terminology is due to the nature of the Proof-of-Work (PoW) protocol. Like a miner in search of gold, the PoW protocol for creating a block requires considerable effort, and the more effort you expend, the more likely you are to “mine a block.” Also similar to miners in

search of gold, PoW “mining” is competitive. Each Ethereum block contains a block reward in Ethereum’s currency, known as “ether.” This reward is granted to the miner who successfully adds a valid block to the blockchain before any other miners. The miner succeeds by successively “hashing” the block data in an attempt to find a cryptographic hash that has specific characteristics. The protocol determines the rarity of these “hashes.” This rarity is called the “difficulty,” the number that determines the relative effort that miners have to expend to “mine a block.”

Proof-of-Work uses a difficulty variable to maintain a stable block time as miners come and go, adding to or subtracting from the overall mining power (or hashrate) of the network. For example, if the Ethereum network had a cumulative hashrate of 100 tera hashes per second, and a large mining pool went offline, lowering the cumulative hashrate to 90 TH/s, blocks would suddenly take, on average, 10% longer to mine. Ethereum’s protocol adjusts the difficulty to ensure that the correct block time is targeted. In this case, the protocol would adjust by making finding blocks about 10% easier. This process means that in the event of significant changes in the network’s hashrate, the block times will be affected. Additionally, due to the brute-force nature of the mining process, randomness will be an ever-present source of differences in block times, even if the network hashrate stays constant.

Proof-of-Stake

Proof-of-Stake (PoS) has always been on Ethereum’s roadmap. If designed correctly, PoS has significant benefits over PoW. First, there is no need to burn large amounts of electricity to find a valid block. As the respective names suggest, there is no “work” to do, only “stake” to lose. Second, the downside of being a bad actor in a PoS system can be much more severe than in PoW. Block creators stake ether in order to participate, and if they are proven to be operating maliciously, that stake can be taken away (“slashed”). In PoW, malicious operators can be ignored, but we can’t take away their hardware. That asset lives to corrupt another day. Finally, there are less economies of scale in PoS. When a PoW miner earns a block reward, they can use that money to achieve non-linear advantages over other miners, such as faster network connectivity. In PoS, while people with more stakes will earn more ether, that advantage is linear. The rich still get richer, but not exponentially so. That said, in PoW in order to obtain currency, you can buy hardware and connect to the network, whereas in PoS you must buy from existing currency holders, which gives newcomers a disadvantage.

For the smart contract developer, the behavior of PoS is similar to PoW. We still need to determine when we consider transaction finality, since block creators (called “proposers” in PoS) can still end up on different forks due to network partitions. Where things will get significantly different is when Ethereum transitions to a sharded blockchain.

Under the project name “Serenity,” Ethereum is introducing Proof-of-Stake, sharding, and several other improvements as a way of resolving Ethereum’s fundamental scaling issues.¹ Serenity will also replace the existing EVM with eWASM (Ethereum flavored WebAssembly). None of these initiatives should have a significant effect on the Solidity language. The one significant new complication will be inter-contract communication. Currently, each node has every Ethereum smart contract ever deployed, so it is fairly straightforward for nodes to interact. But once we have a sharded blockchain, contracts will be spread across 1000+ separate blockchains. The protocol for how these contracts will interact is still being developed, but it’s safe to say that smart contract developers will no longer be able to assume that any arbitrary contract will be synchronously available to them. Furthermore, every Ethereum address will need to have a shard ID associated with it in order to know where its state is stored.

Proof-of-Authority

In some situations, it may make sense to use a blockchain while restricting block creation to only certain entities. These situations cut against the typical use case of having it be open to the public. This sort of restricted block creator status is provided by the Proof-of-Authority (PoA) protocol. PoA is commonly used for private blockchains that are hidden within internal networks. The block creators simply take turns appending the next block at the correct frequency. Due to the round-robin nature of PoA, temporary forks are far less likely.

There are many more consensus protocols, but they are beyond the scope of this book. This is an active area of research, so expect to see new consensus protocols emerging for years to come.

Regardless of which protocol is used, there is a reward for any node that adds a new block to the blockchain. Block creators receive a block reward as well as the sum of all transaction fees in the block. In Ethereum, transaction fees are referred to as “gas.” We will dig more deeply into the topic of gas later in this chapter.

For the purposes of this book, we don’t need to understand the mechanics of consensus protocols. It’s only important to know the role these protocols play within a blockchain, and understand the differences between the most popular protocols. Smart contract developers can essentially treat consensus protocols like a black box. We need to learn each box’s external behavior, but the internals have no effect on our work.

We will now work to understand how transactions are eventually added to a blockchain.

¹ Serenity is still under active development at the time of this writing.

Transaction Processing

In 2019, a new block of transactions was added to the Ethereum blockchain approximately every 13 seconds, on average. That said, a sample of block times across a 65-hour span (between blocks 8,662,243 and 8,679,755) contained a block that took over 2 minutes to be written to the chain (block 8,674,540) and others that took just 1 second. This wide variation in block times is due to the random nature of Proof-of-Work. As Ethereum transitions to Proof-of-Stake, we expect to see block times and block time variability decrease.

While block times are regulated by the protocol, the actual time that a transaction takes, from the moment it is first broadcast on the network to its execution in a block, can vary from a few seconds to a few hours. This variability is due to the limitations of the current Ethereum protocol combined with its popularity. For example, if Ethereum can process around 25 transactions per second, but there are over 30 transactions entering the network every second, then there will be transactions that will remain unexecuted until network demand slows down to under 25 transactions per second. These pending transactions are held in-memory by each Ethereum node in what is called a “mempool.” The most reliable way to quickly get a transaction out of the mempool and into a block is to pay more fees to the block creator. In Ethereum, these fees are called “gas,” and as a smart contract developer, you’ll have to frequently consider “gas prices.” See more on the topic of gas and gas prices in [Figure 1-2](#).

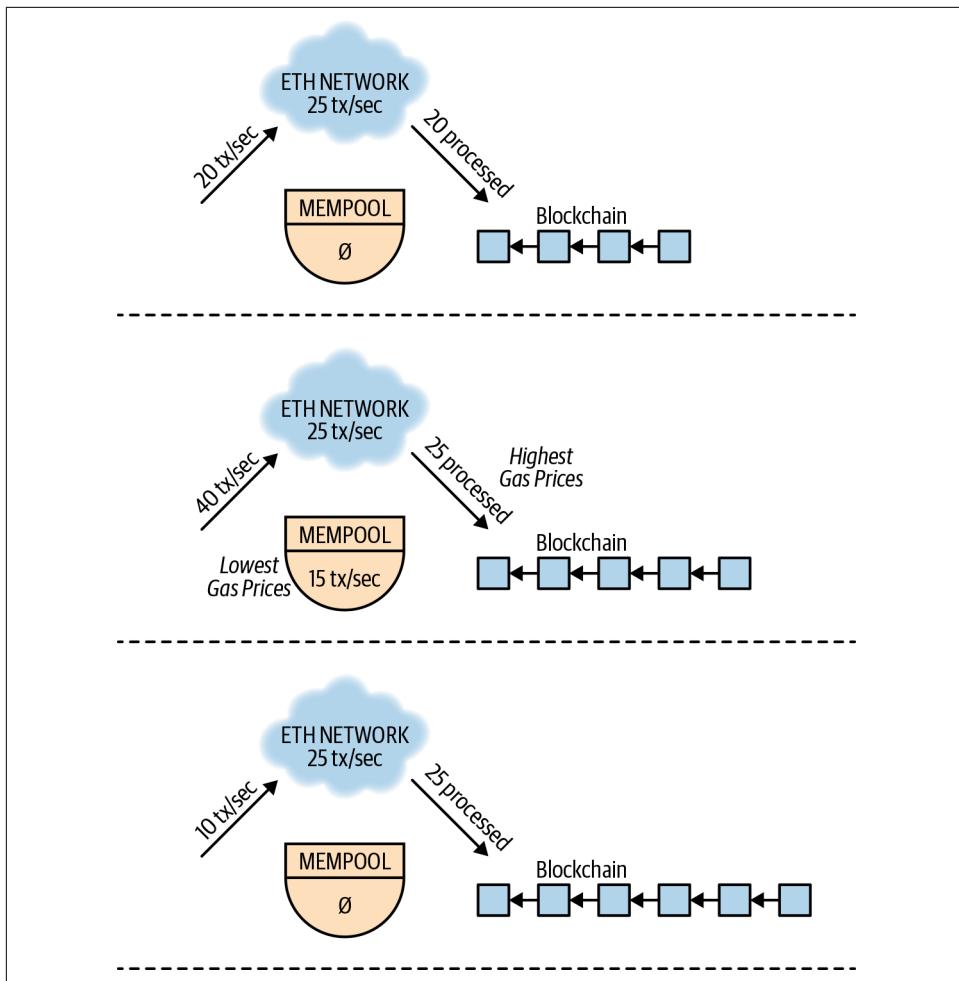


Figure 1-2. In step 1, just 20 transactions/second are broadcasted on the network. Assuming the network can currently handle 25 transactions/second, all transactions are mined into the next block. In step 2, 40 transactions/second are broadcasted and the network falls behind, creating a mempool of pending transactions. In step 3, few transactions are sent—just 10 per second—and the mempool drains as the network catches up. The fewer transactions in step 3 would have likely been caused by rising gas prices.

If the same account has two transactions in the mempool, the block creators know which transaction to choose first based on its “nonce,” an account-specific counter that is incremented with each transaction. This means that while the mempool is generally a heap of transactions that block creators can choose from, it’s actually a heap of queues in the cases where accounts have sent multiple transactions before seeing

them succeed. [Figure 1-3](#) shows how each account's transactions are sequenced by its nonce.

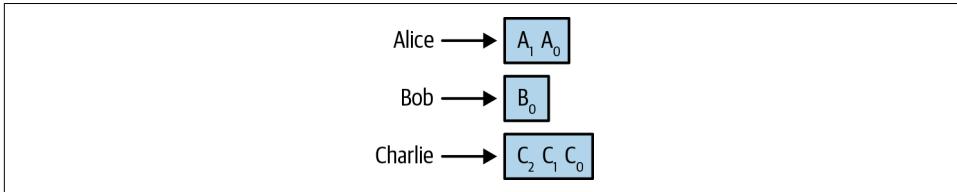


Figure 1-3. Alice's current nonce is 1, Bob's is 0, and Charlie's is 2

Regardless of whether a transaction is delayed by the nature of the protocol or due to network congestion, as a developer, you need to consider the unpredictable and asynchronous nature of your users' experiences with your software system. In a world where financial, commerce, and social media software have been tuned to accept transactions in less than a second, systems backed by smart contracts have significant user experience challenges. The next section digs a level deeper into these challenges.

Transaction Finality

When you successfully execute a transaction in a typical database, developers assume that its effects will never be rolled back. In the case of blockchains, though, it's possible that a user could see a successful transaction included in a block, only to see that block promptly orphaned and replaced with a different block. This block may not have included the previously successful transaction! This lack of finality with blockchain transactions is due to the decentralized nature of the network. It's possible for two different nodes to create blocks nearly simultaneously, and for their respective peers to be on different "forks" of the blockchain, as mentioned earlier. Whichever chain creates the next block first will beat the other fork, and previously successful transactions can disappear in the process. From the user's standpoint, it appears that their once successful transaction gets reverted. Figures 1-4 through 1-6 illustrate the issue.

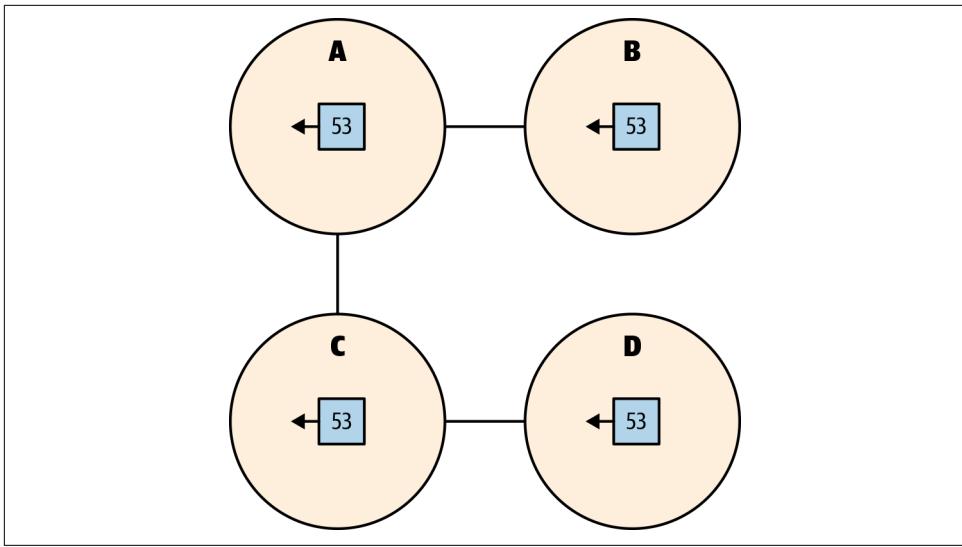


Figure 1-4. Illustrating a temporary fork, step 1: all four nodes have the same block 53 and consensus with each other.

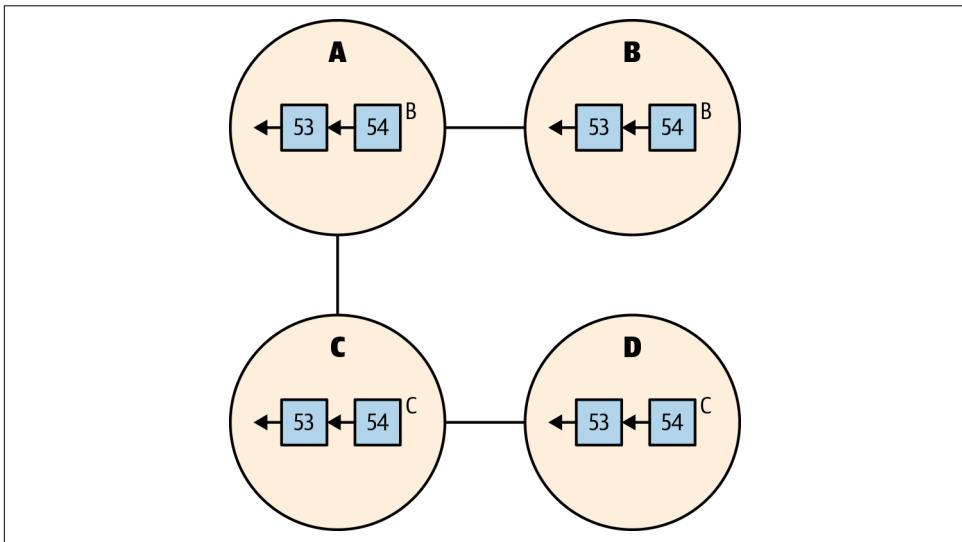


Figure 1-5. Illustrating a temporary fork, step 2: nodes B and C have forked from each other due to mining (or receiving) two different block 54s simultaneously. Node D has received Node C's block 54, while Node A has received Node B's block 54. There are now two divergent blockchains.

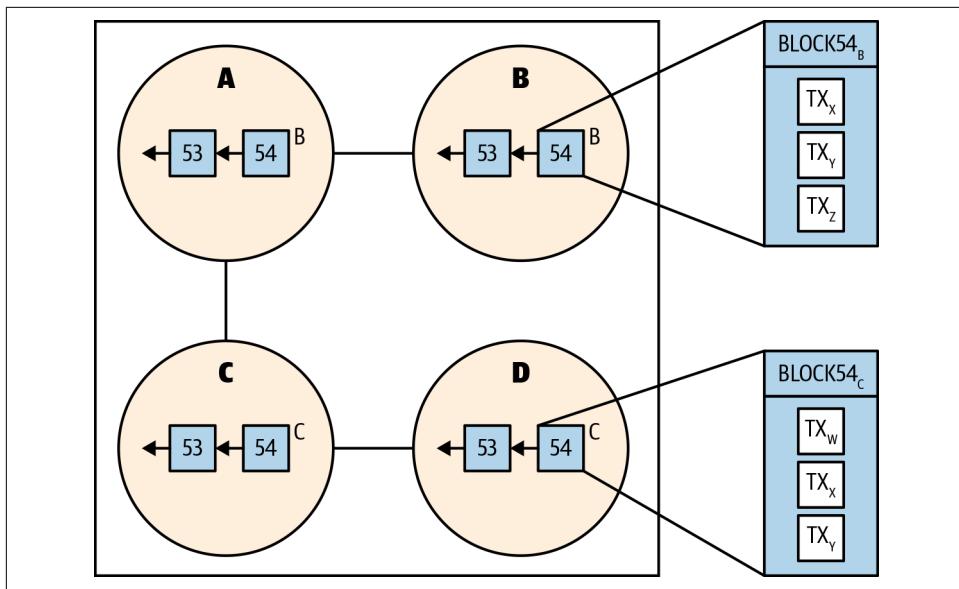


Figure 1-6. Illustrating a temporary fork, step 3: note that the competing blocks of a temporary fork can contain different transactions. Once this fork is resolved, either TX_w or TX_z will no longer exist on the blockchain.

This problem can be solved by waiting for additional blocks to be added to the chain before considering the transaction settled. Each application will need to have its own threshold for considering a transaction final. Popular cryptocurrency exchanges will wait for up to 50 blocks for a transaction to be treated as “confirmed” on the Ethereum network. When the stakes are lower, it’s less necessary to wait that long. Each application should consider how long it wants to wait to consider transactions final. In some cases, this “confirmation” will be built into the user experience, and in others, it may just be a general warning that not all transactions are final.

Now that we’ve covered temporary forks, let’s talk about their more permanent cousins.

Hard Forks

Each node in a blockchain network must run software that is compatible with the protocol in order to participate in the propagation of blocks and transactions as well as the creation of new blocks. The community of blockchain node developers, protocol developers, block creators, exchanges, and application developers all weigh in on how the protocol should best evolve to suit the needs of the community. Ultimately, this evolution happens through developers changing the blockchain node software, and the people who run the node software (what we call “node runners”) choosing whether to use the new version or not. Sometimes, the changes made to the protocol

are so significant that the new version is not compatible with previous versions of the software. This situation is called a “hard fork” because a new blockchain will spawn from the old, creating a fork in the road for node runners, who will have to decide for themselves which fork they will support.

Hard forks can be contentious or non-contentious. Non-contentious hard forks happen when the entire community upgrades, effectively abandoning the old fork for the new one. This is typical when there is a serious defect discovered in the protocol. Contentious hard forks happen when a critical mass of node runners decide to continue using the existing fork and to evolve the old protocol separately from the new version. Ethereum (ETH) had a contentious hard fork in 2016 that resulted in the previous blockchain surviving and calling itself Ethereum Classic (ETC). There could be additional contentious hard forks as the community implements the various components of the Serenity release.

We've been focused mainly on general aspects of blockchains; now, let's focus in on Ethereum specifically.

Ethereum Fundamentals

We have been describing relatively general characteristics of blockchains, with some Ethereum-specific examples. Now we dig into the fundamentals of Ethereum, and discuss how these pieces fit together to enable smart contract development. It is important to understand transaction costs, how accounts and contracts are identified, and how transactions are executed within blocks. These Ethereum fundamentals form the foundation for understanding how to design and develop decentralized applications.

Ether and Gas

The Ethereum protocol has its own currency, called ether. The fundamental use of this currency is to pay block creators to include transactions in blocks. Much like the US dollar, ether is divisible, though to much smaller fractions than a cent. The smallest unit of ether is called a wei, which is a quintillio nth of an ether (for perspective, a quintillion is a billion billions, or 10^{18}). Due to a wei's small size in proportion to ether, you'll often see ether denominated by Gwei, particularly when it comes to gas prices. A Gwei is a billion wei, and a billion Gwei is one ether.

While there is a single Ethereum protocol, there is more than one network running that protocol. It's possible to create private networks running Ethereum, similar to how a private “internet” is called an “intranet.” These private networks still deal in ether just like the public Ethereum network, but their ether isn't worth anything on the open market. The ether on the public Ethereum network is known as ETH and it has real-world value. The public Ethereum network is referred to as “mainnet” by

developers. There are also public test networks or “testnets” that the community uses as staging environments. These test networks typically have “faucets,” or mechanisms to give developers free ether in order to test their smart contracts. Every network running the Ethereum protocol has ether, but mainnet’s ETH is the one that has actual value. To deploy and execute smart contracts on mainnet, we need an account that holds ETH. To acquire ETH you will need to purchase it via an exchange, receive it from a friend, earn it from a business, or mine it yourself.

Smart contract code, such as Solidity, is compiled to bytecode, which provides a series of opcodes to the EVM. An opcode is an instruction such as `PUSH1` or `MLOAD` that is interpreted by the EVM. Each of these opcodes has an associated “gas cost.”² Let’s take a look at a simple smart contract, and inspect its bytecode and opcodes. Thankfully, as smart contract developers, we don’t need to understand the contents of the bytecode or opcodes since we can understand the Solidity easily enough. That said, as you progress as a smart contract developer, digging into these lower-level concepts will give you a deeper understanding of what’s possible.

```
pragma solidity ^0.4.25;

contract Incrementer {
    uint256 public count;

    function addOne() public {
        count++;
    }
}
```

Compiling this contract using `solc` results in the following bytecode:

And the opcodes as reported by `solc` look like this:

² A mapping of opcodes to gas costs can be found in Appendix G of the Ethereum yellow paper at <https://ethereum.github.io/yellowpaper/paper.pdf>.

```
PUSH1 0x0 DUP1 REVERT JUMPDEST PUSH1 0x70 PUSH1 0x99 JUMP JUMPDEST PUSH1 0x40
MLOAD DUP1 DUP3 DUP2 MSTORE PUSH1 0x20 ADD SWAP2 POP POP PUSH1 0x40 MLOAD DUP1
SWAP2 SUB SWAP1 RETURN JUMPDEST PUSH1 0x0 DUP1 DUP2 SLOAD DUP1 SWAP3 SWAP2 SWAP1
PUSH1 0x1 ADD SWAP2 SWAP1 POP SSTORE POP JUMP JUMPDEST PUSH1 0x0 SLOAD DUP2 JUMP
STOP LOG1 PUSH6 0x627A7A723058 KECCAK256 DUP15 SWAP11 CREATE2 SELFDESTRUCT 0xaf
0xd3 DUP8 0xe6 PUSH28 0x7C38D8239AAA70FDE96A805CEBF86F30517DD68E8664BE0029000000
```

The concept of “gas” exists to decouple the price of ether from Ethereum’s transaction fees. Without this decoupling, Ethereum transaction fees would be pegged to the price of ether, which would have negative impacts on the ecosystem due to the volatility of ETH’s value. To decouple the exchange rate between gas and ether, every Ethereum transaction sets its own **gasprice** to determine how many wei a single unit of gas costs. As block creators decide which transactions to include in a block, they are incentivized to include the transactions that will give them the most generous **gasprice** for their computations. At any given moment there is an **implicitly understood market rate for gas**. Paying below market rate will mean waiting longer than most other transactions for your transaction to be executed, while paying above market rate will allow you to leave the mempool sooner than most.

Ethereum transaction creators need to consider the costs versus benefits of setting their gas price above or below the current market rate. Some transactions are not time-sensitive and can wait many hours to be executed. In this case, it makes sense to pay relatively little ether for a transaction’s execution. Block creators finally include low-gas-price transactions when the mempool is shallow and there isn’t a critical mass of transactions with a generous **gasprice**. Inversely, some transactions are incredibly time-sensitive, and are so valuable that it makes sense to pay significantly above the market rate for fast block inclusion. There are even people who run software to observe the mempool in real time and try to “front-run” certain financial trading transactions by seeing a pending trade and jumping ahead of it by setting a higher **gasprice**.

Each Ethereum transaction has to include **gas** and **gasprice** attributes, which when multiplied will set the maximum transaction fee for the transaction, denominated in wei. This **gas** attribute sets a limit to how many computations the transaction can perform. If that limit is reached, the smart contract execution reverts, but the transaction is still written to the blockchain, and the fees are consumed by the block creator. If the smart contract call completes with leftover gas, the gas is returned to the transaction creator. The sum of the gas used across all transactions in a block cannot exceed the block’s specified **gaslimit**. This also means that no single transaction’s gas usage can exceed the block’s **gaslimit**.

The next section will dig into the details of the players within a transaction and how they are identified.

Accounts

The most basic Ethereum transactions are just externally owned addresses (EOAs) sending ether to each other. Additionally, Ethereum transactions can be sent from an EOA to a smart contract. Both EOAs and smart contracts are identified by an Ethereum address like the following:

`0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5`

An address is represented by a hexadecimal number. Given the magnitude of the numbers involved in generating an address, it is practically impossible to generate two identical addresses. There is no way to distinguish an address for a smart contract from an EOA without inspecting the blockchain. While accounts and contracts have undistinguishable addresses, they have some important differences.

Every transaction in the Ethereum blockchain is initiated by an EOA. Smart contracts can't spontaneously perform an action. They can call other smart contracts, but every transaction originates from an EOA. When contracts are called, they can emit events, store data, receive ether, send ether to EOAs, or send data or ether to other contracts. Besides initiating transactions, EOAs, on the other hand, can only receive ether. They cannot react to any transactions that involve them the way that a smart contract does.

Contracts

Contracts in Solidity are organized in an object-oriented style similar to the Java programming language. In object-oriented parlance, a contract is really a class, or a collection of state variables and functions. In order to reuse common functionality, object-oriented languages allow a class to inherit from another class (or contracts, in this case). Since Solidity uses `function` as a keyword, we will refer to as “functions” what most object-oriented languages refer to as “methods.” Solidity functions can be separated into two distinct types: write-only and read-only.

Read-only Solidity functions are denoted with the `pure` and `view` keywords. These functions can take input data, read contract data, operate on that data, and return data. Read-only functions cannot change the state of the contract or emit events. Because no updates are needed on-chain, read-only functions are instantaneous—they are very similar to web API calls, particularly GET requests. It's important to note that being able to skip on-chain updates means that read-only functions can be called without paying any gas costs, and there will be no transaction created.

Write-only functions are the default in Solidity so they require no additional keywords. Despite being “write-only,” they can actually return data, but due to the asynchronous nature of Ethereum, the return data is practically useless, hence “write-only.” These functions are the workhorses of Ethereum, and their data must be sent via transaction and included in a block in order for the function to be executed. An unsuccessful write-only method will revert, either because it has run out of gas or

reached an invalid EVM state, or because of an explicit statement in the contract, such as failing a `require` statement. A successful write-only method doesn't actually have to change anything, but it typically changes *something* and often emits one or more events in the process.

The purpose of events in Ethereum is generally twofold: to provide a custom historical log of what has occurred in the contract, and to allow observers to subscribe to real-time updates. Due to the nature of the blockchain, we already have a historical ledger of everything that has ever happened, but events are a convenient way to provide more domain-specific logging and updates so that users don't have to create their own interpretations to transactions and state transitions.

To better understand the mechanics of Ethereum's smart contracts, let's now focus on the vehicles of their modification.

Blocks and Transactions

Only Ethereum block creators determine the attributes of a block, such as which transactions are included. Similarly, only Ethereum users determine the attributes of a transaction, such as which contract to send data to. As smart contract developers, we frequently need to be aware of the state of the block as well as the state of the transaction that is currently executing.

Solidity exposes the following transaction (`tx`) attributes:

`gasprice`

The gas price (in wei) set by the EOA that created the transaction.

`origin`

The address of the EOA that created the transaction. Surprisingly, this address is rarely useful and often insecure.

A transaction can have an arbitrary number of contracts involved in its execution, provided that its execution fits within the constraints of the block's `gaslimit`. Solidity exposes a number of other transaction-related attributes, but groups them into a message (`msg`) abstraction. Messages refer to the communication between contracts and anything that can call them, such as other contracts. For example, a contract function call will always have a `msg.sender`. That `msg.sender` could be equal to the `tx.origin` or the creator of the transaction, or it could be the address of an intermediary contract.

Solidity's message (`msg`) attributes are as follows:

`data`

The raw bytes of data that were sent to the currently executed external or public function. This is also referred to as `calldata`.

sender

The address of the caller of the currently executed external or public function.

sig

The first four bytes of the calldata determine which function is being called. This is also referred to as the function identifier.

value

The amount of wei that is being sent to this function.

Solidity exposes the following **block** attributes:

number

Each block increments this number. The genesis block was block 0.

timestamp

The time in seconds since epoch that the block was created. You may also see code that uses its alias, `now`.

blockhash

In addition to its sequential block number, each block is uniquely identified by its hash. A hash is a hexadecimal number such as `0x88e96d4537bea4d9c05d12549907b32561d3bf31f45aae734cdc119f13406cb6`. The hash of the current block is not available, but by providing a block number, you can get the hash of any block within the past 256 blocks.

difficulty

The mining difficulty level of the current block.

gaslimit

The maximum amount of gas that can be consumed by this block. This is set by the block creator.

coinbase

The address of the block creator.

Next, we will address the quirky aspect of how time works in a smart contract.

What Time Is It?

Most programming languages allow developers to check the time, which typically uses the time reported by the computer the program is running on. For instance, this JavaScript program running on Node.js reports the current time as it is executing:

```
for (let i = 0; i < 10; i++) {
    console.log(new Date());
}
```

The output of this program would be something like this:

```
2019-10-05T05:08:45.058Z  
2019-10-05T05:08:45.059Z  
2019-10-05T05:08:45.060Z  
2019-10-05T05:08:45.060Z  
2019-10-05T05:08:45.060Z  
2019-10-05T05:08:45.060Z  
2019-10-05T05:08:45.060Z  
2019-10-05T05:08:45.060Z  
2019-10-05T05:08:45.060Z  
2019-10-05T05:08:45.060Z
```

Not surprisingly, you can see the time changing as the program executes. Here is the equivalent code in Solidity:

```
pragma solidity ^0.5.0;  
  
contract TimeReporter {  
    event TimeLog(uint256 time);  
  
    function reportTime() public {  
        for (uint8 i = 0; i < 10; i++) {  
            emit TimeLog(block.timestamp);  
        }  
    }  
}
```

The events emitted by this function would all have the same exact `time` attribute. That `time` attribute, set by `block.timestamp`, is the time that the block was added to the blockchain. For every transaction in that block, the `block.timestamp` attribute will be identical. While the clock on a computer ticks at least once a millisecond, the “clock” on a blockchain only ticks as often as blocks are added to the chain. Due to a blockchain’s low-resolution clock, we can never expect an exact second to occur. When you write code that checks the time, comparisons should always involve greater or less than, rather than exactly equal.

When designing smart contracts, it is also important to keep in mind that that block creators can manipulate the time a block is created as well as the ordering of the transactions to their advantage. For instance, if a smart contract has a built-in deadline, and it would significantly benefit most block creators to have that deadline missed, then they can choose to delay any transactions that would meet the deadline. Block creators manipulating the ordering of transactions to their benefit can come in the form of “front-running” token trades. For example, someone sends a transaction to buy 5 ABC tokens for 1 ETH, due to some new information about ABC tokens. A block creator with significant hash power could delay that 5 ABC / 1 ETH transaction and add their own transaction to a block for a lesser amount and grab those ABC tokens for themselves. This is only possible if the block creator actually succeeds at creating a block, which is highly competitive. So these concerns aren’t a significant

vulnerability of smart contracts, but they are important to keep in mind as you consider your designs and incentives.

Finally, let's consider some of the "crypto" in cryptocurrency.

Signing Transactions

We need to appreciate the reasons why when we "sign" a transaction, we can be sure that signature was created by a specific private key. This comes down to good old public-key cryptography. A private key is the secret counterpart to a public key. The address of an EOA is a truncation of the hash of its public key.³

Mercifully, in the course of developing smart contracts and decentralized applications (DApps), we don't often work directly with private keys. We strongly recommend to leave private key management and cryptographic signatures to wallet software such as MetaMask. Knowing an EOAs private key is synonymous with owning that account because the private key is what is used to sign transactions. Without this cryptographic signature, there is no way to authenticate whether a transaction was actually sent by its specified EOA.

When we send an Ethereum transaction using any of the Web3 libraries, the cryptographic signature happens in the background. The following transaction attributes are concatenated, encoded, and then signed with the configured private key:

nonce

Sequence number of this transaction for this EOA.

gasPrice

Amount of wei that this transaction is paying per unit of gas.

gas

Amount of gas that this transaction is willing to spend.

to

Recipient address of this transaction. It could be an EOA or a contract.

value

Amount of wei (if any) that this transaction is sending to the recipient.

data

In the case of a contract call, this contains the function name and all parameters. In the case of a contract deployment, this contains the contract bytecode. If no contracts are involved, this is generally blank.

³ For a deep dive into cryptography, read *Applied Cryptography* by Bruce Schneier (John Wiley & Sons).

`chainId`

Each public Ethereum network has a `chainId`. Mainnet is 1, the Kovan testnet is 42, etc.

Once those attributes are signed, the signature itself is included in the transaction so that Ethereum nodes can validate that the sender is legitimate. In order to validate this, nodes use the sender's address to validate the signature. If someone were to try to send a transaction with a bad signature, the nodes would reject it.

Summary

We have touched lightly on many aspects of blockchains and Ethereum in this chapter. We hope that we have whetted your appetite and that you are eager to dig deeper into decentralized application development via our next chapter. The rest of the chapters in [Part I](#) of the book are increasingly pragmatic preparations for [Part II](#), where we will begin developing smart contracts and DApps in earnest.

Decentralized Applications

In this chapter we will take a look at a few sample use cases for building applications on the Ethereum blockchain. For each case we will take a look at what characteristics of the blockchain lend themselves well to supporting these applications and, where applicable, we will take a look at a few production examples. We will then conclude this chapter with a few questions to ask yourself when starting your next project to make sure the blockchain is the right technology to use.

Let's jump into our first and likely most common use, tokens.

Tokens

Tokens are an abstraction that represents ownership. Ownership implies certain privileges such as the right to use or sell an item like a vehicle or a house. In these cases the token, or proof of ownership, is represented by a title or deed. It can also provide the right to access something such as software or a movie. Here the tokens may be represented by a license key or a physical ticket. With ownership being something that can change often, tracking these changes on a cryptographically secured platform makes a lot of sense.

To assist developers in the creation of tokens, the Ethereum community has developed several different types of token standards through the [Ethereum Improvement Proposal \(EIP\) process](#). Let's take a look at two of these standards, [ERC-20](#) and [ERC-721](#).

ERC-20

The ERC-20 standard is used when creating a fungible, or mutually interchangeable token. These tokens would be ideal replacements for things like rewards points from retailers, miles from airlines, or a currency. All tokens created from an ERC-20

contract are considered to have the same value and are effectively indistinguishable from each other. Because all the tokens are considered identical, the primary responsibility of an ERC-20 contract is tracking balances.

This standard has several optional fields such as name and symbol, but requires the implementation of the following:

`totalSupply()`

Returns the number of tokens in circulation.

`balanceOf(address owner)`

Returns the balance for the provided address.

`transfer(address to, uint value)`

Returns `true` or `false` based on the success or failure of the transfer.

`transferFrom(address from, address to, uint value)`

Returns `true` or `false` based on the success or failure of the transfer. In this case the transfer is initiated by another address that is not the owner, and must have been previously approved to spend these funds on the owner's behalf.

`approve(address spender, uint value)`

Returns `true` or `false` if successful. The method is used to set an amount that another address can spend on the owner's behalf.

`allowance(address owner, address spender)`

Returns the amount still available for the spender to withdraw.

If you ever find yourself working on an application that requires an ERC-20 implementation, you can create an ERC-20 token by defining these functions; however, we recommend using the [OpenZeppelin contracts](#) as a basis to get started. These contracts have all been thoroughly audited and well documented, making them a great starting place for crafting your own token.

ERC-20 tokens have been used for many different purposes, but one that has likely caught your attention is the Initial Coin Offering (ICO). In an ICO, an organization will sell tokens as a means to raise funds. In some regards this is similar to what [Kickstarter](#) does for organizations or new projects but in a decentralized way.

If the token is expected to gain in value based on the performance of the issuing organization, the token may be considered a security. If that is the case, there are likely going to be some regulatory requirements that need to be considered when developing these smart contracts. For tokens that may fall into this category, there is a [draft proposal](#) in the works that may help keep the token in compliance.

For real-world examples, you can check out a list of deployed tokens on [Etherscan](#).

Now let's talk about tokens that are not mutually interchangeable or, in other words, non-fungible.

Non-Fungible Token (ERC-721)

ERC-721 provides the standard for non-fungible tokens. These tokens are—or at least have the ability to become—different from each other. Because these tokens are distinct, the contract cannot simply track a balance of tokens but must track each individual token that it issues.

Like the ERC-20, ERC-721 contracts may optionally implement fields for name and symbol, and they can also optionally include a token URI (Uniform Resource Identifier), but they *must* implement the following:

`balanceOf(address owner)`

Returns the number of non-fungible tokens (NFT) for the passed-in owner.

`ownerOf(uint256 tokenId)`

Returns the address of the owner for that particular token ID. Remember, every token is unique and the contract has to maintain a data structure that tracks ownership.

`setTransferFrom(address from, address to, uint256 tokenId, bytes data)`

Transfers ownership of the token. In a case where the `to` address is a contract, it will subsequently invoke the `onERC721Received` function to ensure it can accept the token.

`setTransferFrom(address from, address to, uint256 tokenId)`

Same as the previous function, but sets data to an empty string. This also shows that the Solidity programming language supports method overloading, or two methods with the same name but different parameters.

`transferFrom(address from, address to, uint256 tokenId)`

Transfers ownership of the token. Unlike the `safeTransferFrom` methods, it does not check to make sure the receiver is capable of receiving an ERC-721 token. If the receiver is not able to accept ERC-721 tokens, and this method is used, it may be permanently lost. If there's any doubt, use the `safeTransferFrom` method instead.

`approve(address approved, uint256 tokenId)`

Allows another address to transfer the token on behalf of the owner. This is much like the `approve` method for ERC-20, but instead of an amount, we specify which token the approved address can transfer.

```
setApprovalForAll(address operator, bool approve)
```

When `approve` is set to `true`, it approves all tokens from the contract owned by the current owner (message sender) for the address of the operator. When `approve` is set to `false`, it removes access from all tokens owned by the current owner (message sender) for the operator address.

```
getApproved(uint256 tokenId)
```

Returns the approved address for the provided token ID.

Just like ERC-20, OpenZeppelin has created contracts that can be used as the basis for your own implementation of [ERC-721](#).

For an example of how ERC-721s have been used, let's take a look at [CryptoKitties](#). CryptoKitties is a game that allows users to collect virtual cats, where each cat has a unique set of `cattributes`, which are represented by a 256-bit genome. Users then breed their cats to generate new ones, which inherit a mix of the traits from the parent genomes.

This may sound silly, but as of this writing, people have spent over \$27 million USD buying these kitties, with a single kitty going for over \$172,625 USD. If you'd like to see current figures, check out the [Kitty Sales website](#).

ERC-721 tokens can represent just about any unique item that can be transferred. This makes it an ideal token for managing digital collectables or license keys, or even for a registry of physical real-world goods.

Speaking of real-world goods, let's jump into our next major use case, the supply chain.

Supply Chain

In general, a supply chain is made up of the organizations, people, and processes that help products move from supplier to customer. To illustrate a supply chain, let's talk about coffee.

Coffee is typically grown in a region referred to as the Bean Belt. This is an area near the equator that is bounded by the Tropic of Capricorn and Tropic of Cancer. Here you have farmers that plant and harvest coffee, eventually selling their crops to exporters. Exporters will transport the coffee overseas, where they will sell their shipments to the local distributors. The local distributors will then transport the coffee beans to the roasters. The roasters will then roast, blend, and package up coffee that can then be sold to coffee shops for customers to enjoy. This chain is illustrated in [Figure 2-1](#).



Figure 2-1. Coffee supply chain

Each one of these roles, from the farmer to the coffee shop, can be its own company or business entity. This means that each organization is likely going to have an independent system for tracking purchases from supplier to sales. Some will link the sales to a specific lot of incoming product, while others may not make this connection. In cases where the link is not explicitly made, tracing a cup of coffee back to the farm and specific harvest could take a tremendous amount of time.

Shifting this to the blockchain would allow us to trace the origin of the cup of coffee back to the harvest far more quickly. Each harvest would be recorded on-chain, along with each change of ownership, all the way through to the coffee shop. Couple this with IoT devices that ride along with the shipment, posting conditions such as temperature and humidity to the blockchain, and you can have a highly detailed journey from farm to cup. Imagine scanning a QR code on a bag of coffee and seeing where it was farmed, along with the path and environmental conditions from the moment it left the farm until it arrived in the coffee shop.

Now, this may be extreme for coffee, but this is not too far removed from the experiment that Walmart is running. In [Walmart's experiment](#), they are tracking mangos in Mexico and pork in China, with the goal of improving traceability. In order to set a baseline, a team was asked to identify the farm from which a packet of sliced mangos originated. It took the team nearly a week to do the legwork required to trace the mango slices to the farm. After implementing the blockchain solution, it took 2.2 seconds. In a case of food contamination, identifying the source could go a long way toward saving lives.



Walmart is using the [Hyperledger Fabric for its blockchain experiment](#). Hyperledger Fabric is a permissioned blockchain, which means that certain roles and activities, such as transaction validation (mining), account creation, etc., may be restricted or controlled by rules that may or may not live on the chain.

So far we have mostly discussed tracking food, but supply chain applications could be used for just about anything you see on the store shelf. For instance, imagine using this as a consumer to see where the cotton of the shirt you purchased was sourced, or the journey of a piece of furniture took to reach your living room. Having this would give consumers much more insight into their products, allowing them to make informed decisions.

Now that we know how supply chains could be great applications on the blockchain, what about record keeping?

Permanent Records

When you have a long-term asset like a house or a car, it would be great to track all the maintenance that has happened to it in some type of log. Even better than just tracking it would be to have it in a place where these records cannot be destroyed, and this is where the blockchain can help.

This type of record-keeping application is something that is being developed by [Aeron Labs](#). Aeron is focused on the aviation industry, with the goal of making records such as aircraft maintenance and pilot logbooks available on the blockchain.

In the United States, pilot experience requirements and aircraft maintenance requirements are regulated by the Federal Aviation Administration (FAA). Looking at the requirements for pilots, there are different levels of pilot certificates that, once earned, allow certain privileges such as flying in the clouds or flying airplanes with multiple engines. In addition to earning these certificates, pilots may be required to maintain a level of currency by performing specific maneuvers within a given time period. As an example, a pilot may not take passengers with them unless they have [performed 3 takeoffs and landings in the last 90 days](#), and the current way to prove this would be to look at the pilot's logbook.

Proving something like this could be nearly impossible if the logbook in question were lost or destroyed. Putting this type of information on the blockchain would mean the information is permanent, accessible to appropriate parties such as the FAA or National Transportation Safety Board (NTSB), and protected from forgery.

Just like pilots, aircraft are required to have logbooks. Aircraft logbooks track maintenance and compliance with Airworthiness Directives, which are similar to recall notices from an auto manufacturers. With aircraft logbooks, mechanics add records every time they perform any type of upkeep item or repair to the aircraft. If this were moved to the blockchain, as Aeron is attempting, these records would be more quickly available and could not be lost or destroyed.

Aeron is currently building two applications. One is focused on the pilots and would replace traditional logbooks, and the other is on aircraft care to replace the

maintenance logbooks. This would be a significant change for the aviation industry, and its success is dependent on many organizations adopting the technology. If Aeron can help shepherd in blockchain services to this industry, they can demonstrate an example for many other industries to adopt. This type of logging or additive record system could be great for any long-lived asset, and if the technology were adapted to help with health care, you could even have medical records that follow you your entire life.

With these types of applications in mind, let's now turn our attention to deciding if the blockchain is a good fit for you.

Evaluating Blockchain for Your Application

How do you know if the blockchain is right for your application? What things should you consider when making this decision? The following sections describe some strengths of the blockchain.

Verifiable

If you need to make sure that the changes are being made by authorized individuals, the Ethereum blockchain has you covered. This is specifically useful for trustless environments, or environments without a centralized authority. By using cryptographically signed and verified transactions, every state change is protected from forgery or manipulation. The only way someone other than the owner of an EOA could make a transaction on their behalf is if they obtained a copy of their private key.

Transparent

With the Ethereum blockchain being public, anyone can see the transactions that are taking place. This means everything is fully available for auditing. There are tools such as [Etherscan](#) that allow you to inspect all the blocks and their transactions via a browser, or you can run your own Ethereum node and dive into transactions on your local machine. This means users no longer have to rely on the information provided to them from any entity; when the application is on the blockchain, they can go look for themselves. If your application is there to provide clarity for consumers or other industry collaborators, this functionality is already baked in.

Resilient

Each Ethereum node includes a full copy of the blockchain data. This means you will never find yourself in a situation where data was accidentally destroyed and not recoverable. If the data is on the chain, you can get it back by resyncing a node.

If any of these match the requirements of your application, then Ethereum is worth a closer look.

Summary

In this chapter we discussed how the blockchain is being used to build out tokens, provide transparency in supply chains, and create a way to audit pilot and airplane logbooks. These applications are just the tip of the iceberg of what is an exciting tech stack.

In the next chapter we will begin getting our machine ready for smart contract development!

Before We Get Started

In this chapter we are going to set up all the tools needed for smart contract development. We will start by installing an Ethereum client, which is software that can be used to interact with the Ethereum blockchain. We will then install Node.js, which provides the JavaScript environment for Truffle. Lastly, we will install Truffle and Ganache from the Truffle Suite. Truffle provides a fantastic set of utilities used for testing and deploying our contracts, while Ganache gives us a local blockchain environment to run our application locally.

Once these tools are installed, we'll be ready to write our first smart contract. Without further ado, let's get started.

Ethereum Clients

Before we dive into installing an Ethereum client, we should take a moment to discuss how an Ethereum client is different than HTTP clients, which you are likely far more familiar with.

In a traditional web application, the server is centralized and located with a URL or IP address. This software can be written in any programming language that is capable of sending an HTTP request. The client software would make interacting with the server easier since it would contain all the logic and abstractions for building the requests and parsing responses. The client could also be released as a library and made available for other applications to speed up adoption of the service.

Twilio is a great example for this type of client. If you visit [their GitHub page](#), you can see they support libraries in languages like Java, C#, Ruby, Node, etc. This means installing a Twilio client is normally done by using the languages dependency manager and running a few commands on the command line or IDE.

When you install an Ethereum client, you are installing software that will allow you to run an Ethereum node on your machine. This software comes with a command-line interface (CLI), which allows you to create accounts or launch an interactive console that preloads Web3 (more on that in later chapters). Additionally, this software will run a server to expose the Ethereum JSON RPC API.

If you are unfamiliar with JSON RPC, it is a lightweight remote procedure call (RPC) protocol. This means we use JSON to send a request to a server, which then executes some predefined operation. It is through this JSON RPC that we will be interacting with the blockchain. A full specification of what is available can be found on the [Ethereum wiki](#).

There are a number of Ethereum clients you can use, including cpp-ethereum, go-ethereum (otherwise known as Geth), and a few others, but the one we will be using is called Parity. Parity is a client written in Rust and provides one of the faster syncing options of the available clients. It also happens to be the client that is maintained by Parity Technologies, a company founded by the author of the Ethereum yellow paper, Dr. Gavin Wood. Though we have opted for Parity for the purposes of this book, any of the clients that have implemented the Ethereum spec will work. For a full up-to-date list, check out [the docs](#).

Installing Parity

Parity maintains prebuilt binaries for Windows, Mac, and Linux operating systems. If you are running Mac or Ubuntu (or the Windows 10 WSL version of Ubuntu), then you can use the automated installation script by running the following:

```
$ bash <(curl https://get.parity.io -L)
```

Once the script finishes the installation, you can begin to sync the blocks from the network. In our case, instead of syncing from the mainnet, or what you would likely consider “production,” we will sync one of the test networks. This will be much faster while providing the tooling needed for our development. We have opted to use the Goerli, which is a network designed to work with several different Ethereum clients such as Geth and Parity. Let’s start syncing with the following command:

```
$ parity --chain=goerli
```

Once you run this command, you will begin to see terminal output that looks similar to [Example 3-1](#).

Example 3-1. Parity syncing output

```
2019-07-17 14:28:17 Starting Parity-Ethereum/v2.5.5-stable-3ebc76975-20190708/...
2019-07-17 14:28:17 Keys path ~/Library/Application Support/io.parity.ethereum/...
2019-07-17 14:28:17 DB path ~/Library/Application Support/io.parity.ethereum/...
2019-07-17 14:28:17 State DB configuration: fast
2019-07-17 14:28:17 Operating mode: active
2019-07-17 14:28:17 Configured for Görli Testnet ...
2019-07-17 14:28:19 Listening for new connections...
2019-07-17 14:28:24 Syncing #954677 0x6105...6a9a ...
...
...
```

Reviewing the output, the first line gives us feedback about what the client is launching and what version. The next few lines output configuration details, which include where the keys and DBs are stored. After that, we see a server is started on 127.0.0.1:8546. Lastly, we see the sync begin. After the initial sync, launching Parity will be quick, as it will only need to grab the latest blocks that have been added to the chain. At this point you can go ahead and kill the process with Ctrl-C.

We will go into Parity more when it comes time to deploy a smart contract in [Chapter 6](#). For now, we're ready to move on to MetaMask.

Installing MetaMask

If your application required users to download and run a full Ethereum client such as Parity or Geth, you just lost a lot of potential users. Asking a user to install and run a full Ethereum node is a bit much for all but the devoted crypto enthusiast. Outside of early adopters, you will need to provide a much easier way for those less tech-savvy to begin using the application, and that is exactly where MetaMask comes in.

MetaMask is distributed as a browser extension available on Chrome, Firefox, Opera, and Brave. The MetaMask software provides users with the ability to create accounts, and loads a preconfigured instance of Web3 into the browser that is used to interact with the blockchain via JSON RPC. It also provides the user with a 12-word mnemonic that they can store to later recover their accounts should they change computers, browsers, or lose their password. If that's not enough, MetaMask also provides a few options for users to buy ether to fund their accounts, making it the easiest way for new users to get going with Ethereum.

Let's first get MetaMask installed so we can interact with our application later. Go to <https://metamask.io> and install the Chrome extension. Let's walk through the setup, as illustrated in [Figure 3-1](#).

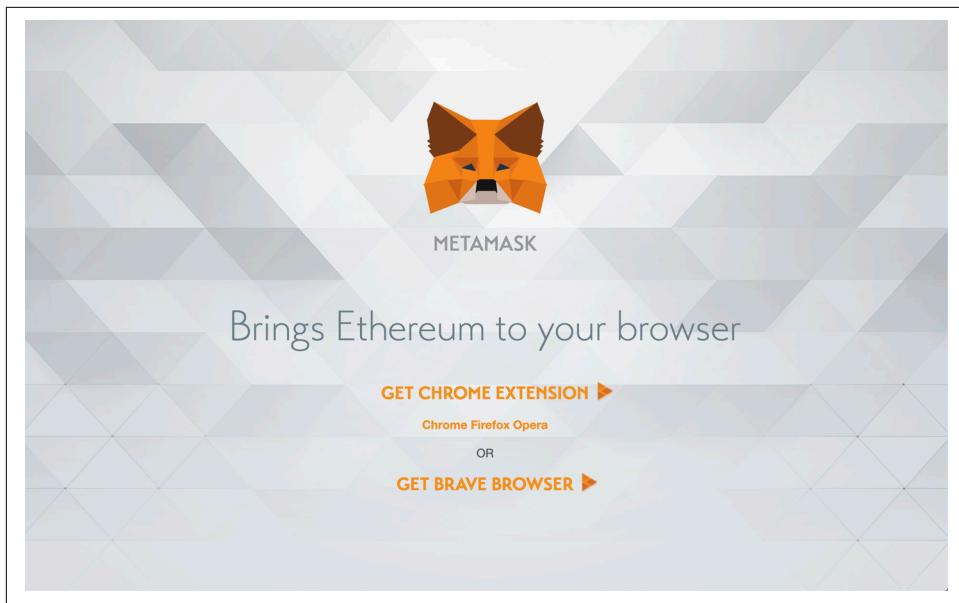


Figure 3-1. Installing MetaMask

Once you click Download, you'll be directed to the Chrome store, as depicted in Figure 3-2.

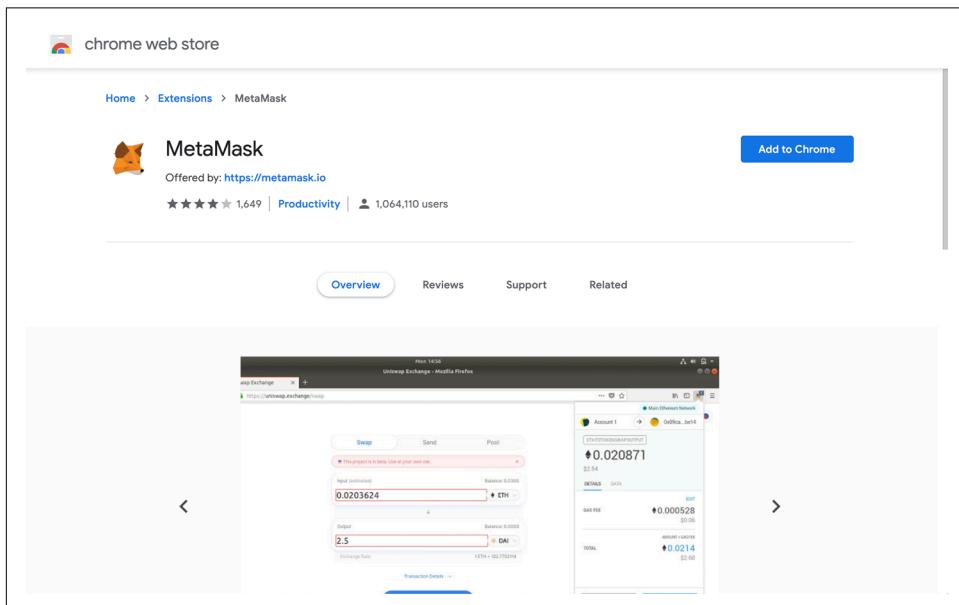


Figure 3-2. Google Chrome store

Install the extension and accept the “Add MetaMask?” prompt in Chrome, as shown in [Figure 3-3](#).

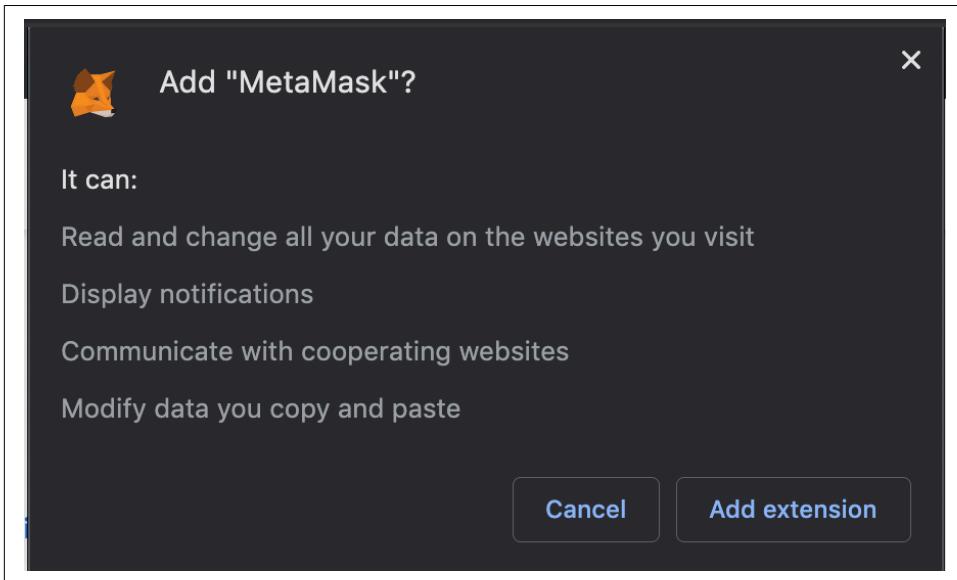


Figure 3-3. Accept MetaMask access controls

If it's installed correctly, you should be redirected to a page asking you to set up MetaMask, as shown in [Figure 3-4](#).

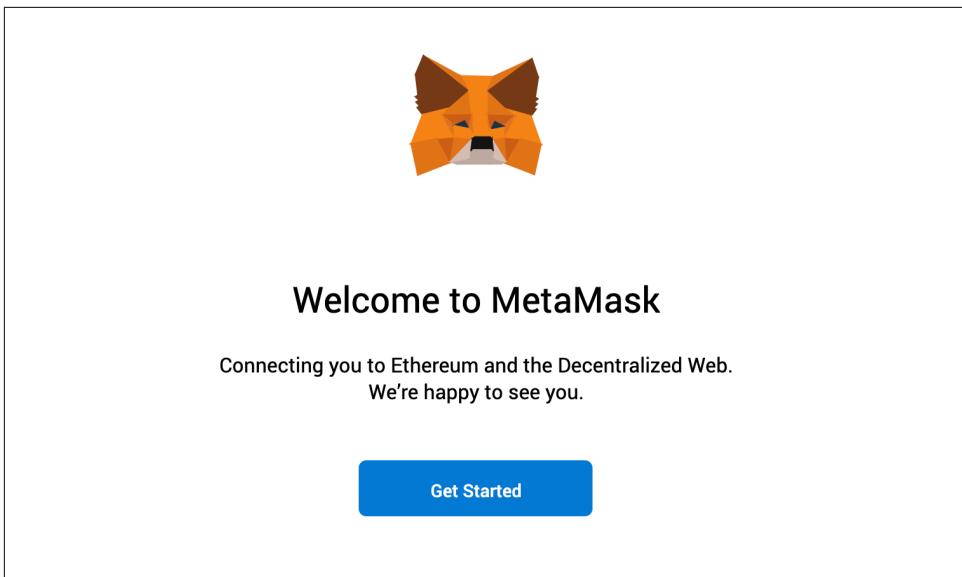


Figure 3-4. Getting started after install

As part of the setup, we're going to create a new wallet on MetaMask. We'll click the button on the right in [Figure 3-5](#) labeled Create a Wallet.

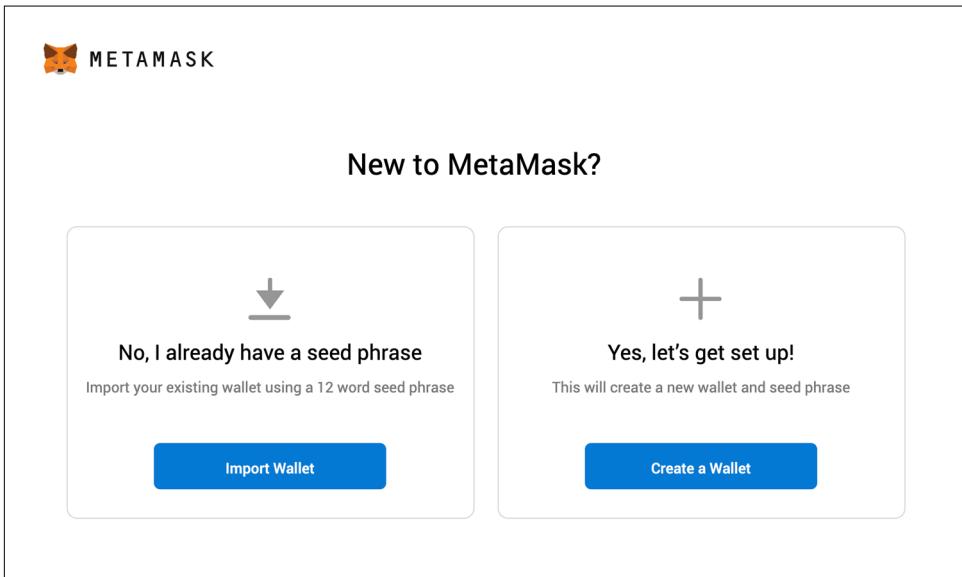


Figure 3-5. Creating a new wallet

In **Figure 3-6** you are prompted to create a password. Be sure to make this a difficult password; better yet, use a password manager to generate one for you. If someone is able to guess your password on your computer, they will gain access to your private keys and with that they would be able to send transactions from your EOAs.

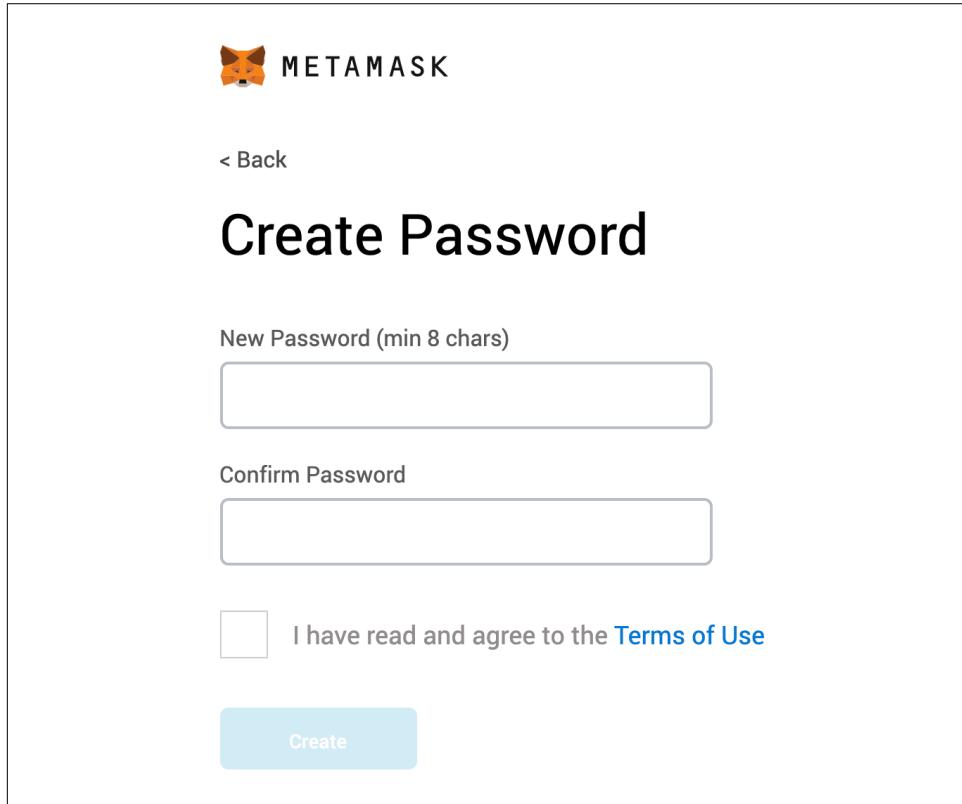


Figure 3-6. Create a secure password

Moving on to **Figure 3-7**, you'll see a clickable area, which reveals a secret backup phrase. Write this down on a piece of paper and secure it in a safe location. If you are using a password manager, save this there as well.

**METAMASK**

Secret Backup Phrase

Your secret backup phrase makes it easy to back up and restore your account.

WARNING: Never disclose your backup phrase. Anyone with this phrase can take your Ether forever.


CLICK HERE TO REVEAL SECRET WORDS

Tips:

- Store this phrase in a password manager like 1Password.
- Write this phrase on a piece of paper and store in a secure location. If you want even more security, write it down on multiple pieces of paper and store each in 2 - 3 different locations.
- Memorize this phrase.

Download this Secret Backup Phrase and keep it stored safely on an external encrypted hard drive or storage medium.

Next

Figure 3-7. Save the secret backup phrase

After you save your backup phrase, click Next and you'll then be taken to the page shown in [Figure 3-8](#). Here you will need to select these one by one. This is a nice little reminder from MetaMask to save these!

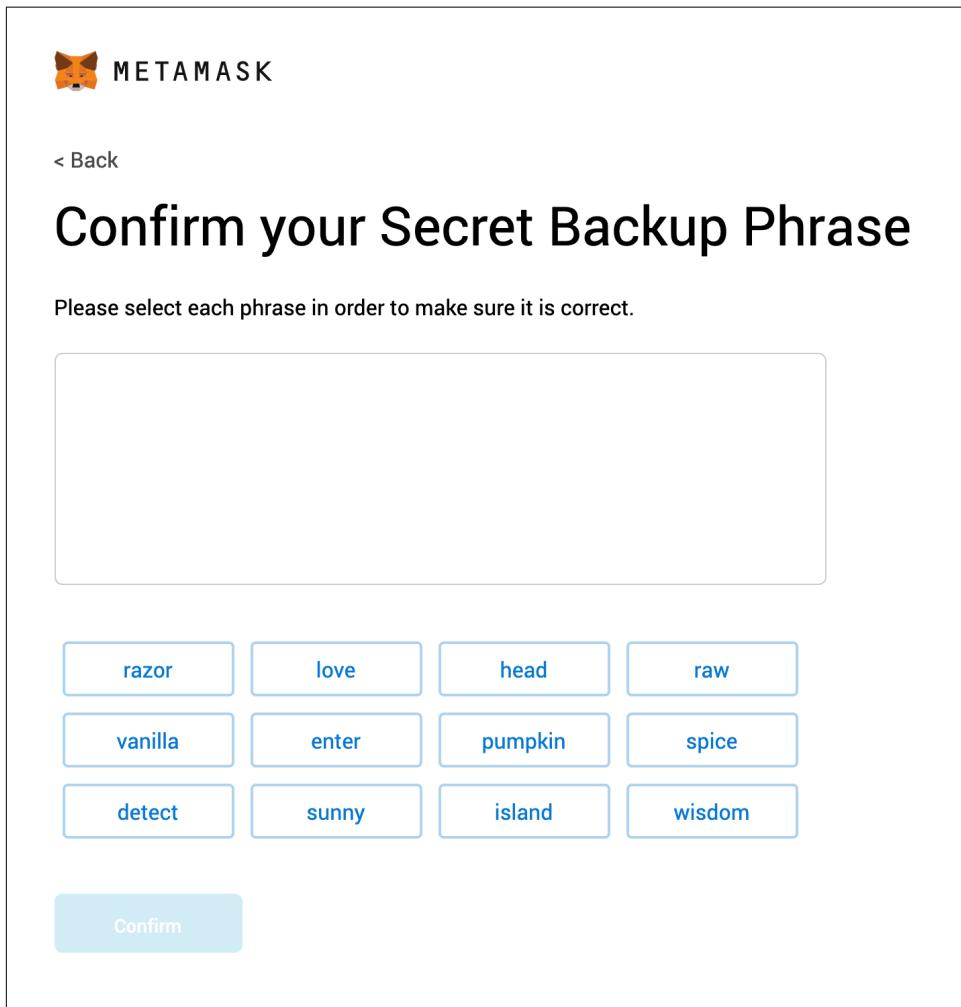


Figure 3-8. Select the phrase in order

Once you've finished the setup, you should be redirected to the page in [Figure 3-9](#).

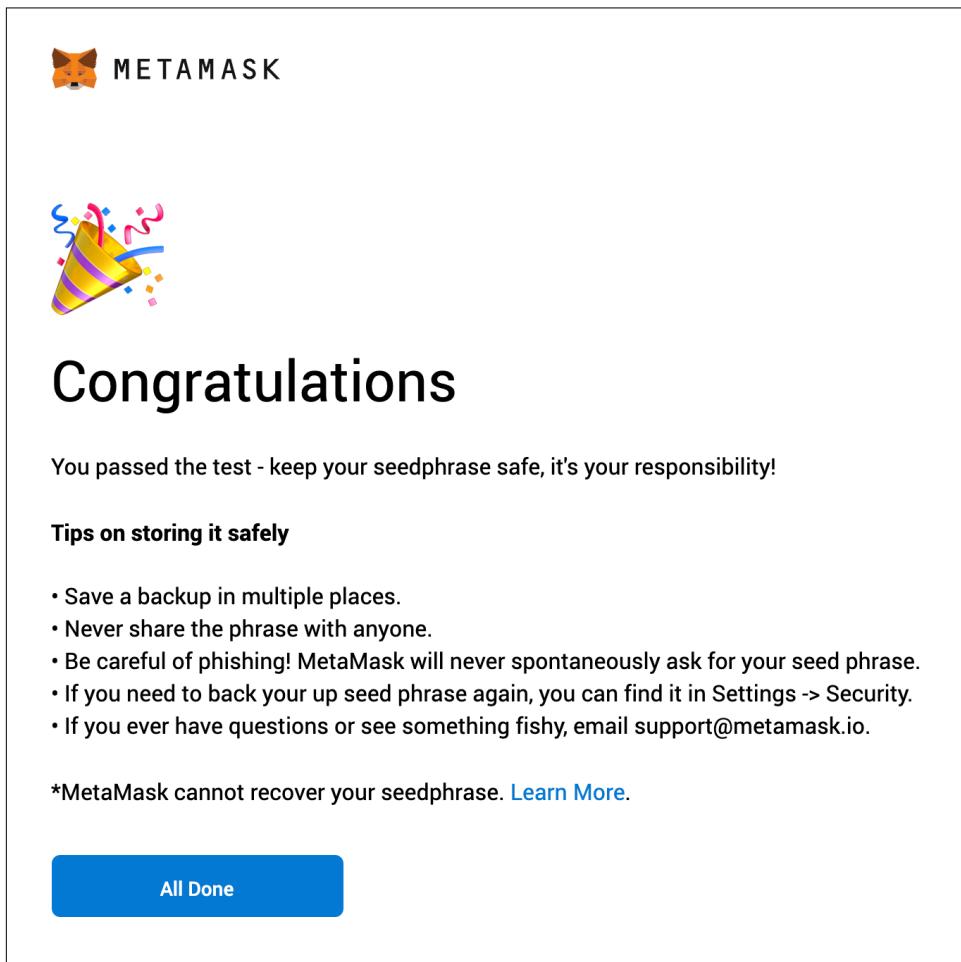


Figure 3-9. Completed setup

Now, if you select the MetaMask extension in your browser, you should be able to see your new wallet and address, as shown in [Figure 3-10](#).

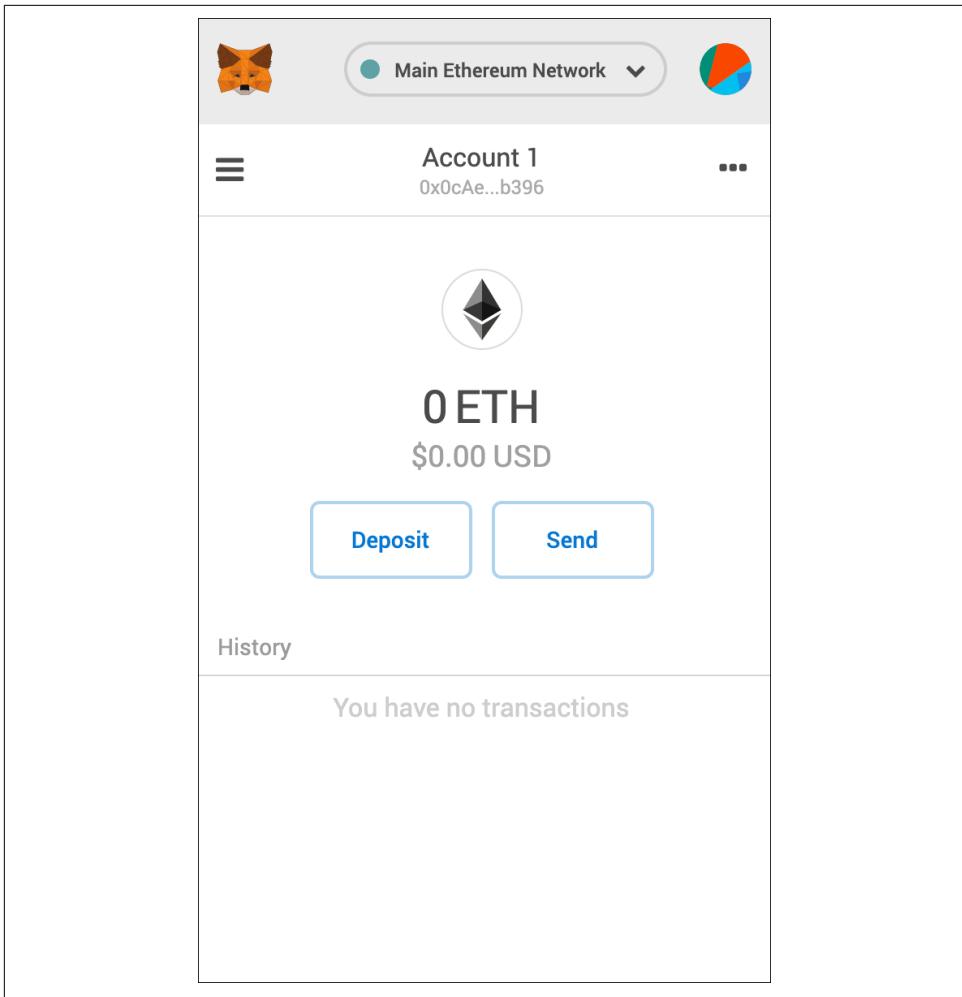


Figure 3-10. Viewing MetaMask account

With MetaMask installed, you can now begin to interact with existing decentralized applications (DApps) from your browser. If you would like to explore a few games or other applications, make sure to stop by [State of the DApps](#).

When you are done exploring some options, it's time to install Node.js for the JavaScript tools we'll need for smart contract development.

Installing Node.js

For better or worse, the little language of the web has become one of the most ubiquitous languages in software, and Ethereum has not escaped its reach. In fact, the primary development tools we will use to develop our smart contracts or interact with the Ethereum network have been built using JavaScript, which means we need Node.js to provide the JavaScript runtime environment.

Ubuntu Installation (Including Windows WSL Ubuntu)

Ubuntu users, including those using Windows WSL, can use the following commands to install Node.js:

```
$ curl -sL https://deb.nodesource.com/setup_10.x | sudo -E bash -
$ sudo apt-get install -y nodejs
```

The `curl` command will begin running a script that will prepare everything for the package manager to install Node.js, and will have output that begins with the following example:

```
## Installing the NodeSource Node.js 10.x repo...

## Populating apt-get cache...

+ apt-get update
Hit:1 http://security.ubuntu.com/ubuntu bionic-security InRelease
Hit:2 http://archive.ubuntu.com/ubuntu bionic InRelease
Hit:3 http://archive.ubuntu.com/ubuntu bionic-updates InRelease
Hit:4 http://archive.ubuntu.com/ubuntu bionic-backports InRelease
Reading package lists... Done
...
...
```

The `apt-get` command will then install the package and its dependencies to your system. Once this is complete, run the following:

```
$ node -v
```

This will verify you are all set with Node.js 10.16.2.



The installation commands assume that the following packages were previously installed on the system:

- curl
- gcc
- g++
- make

Mac Installation

The easiest way to install Node.js on Mac is to use Homebrew, the package manager for OSX. If this is not already installed on your Mac, you can install it with the following script:

```
$ /usr/bin/ruby -e "$(curl -fsSL  
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Once the script is installed, you can run the following command to install node:

```
$ brew install node@10
```

When running this command, you will see output similar to the following snippet:

```
==> Downloading https://homebrew.bintray.com/bottles/node@10-10.16.2.mojave...
==> Downloading from https://akamai.bintray.com/73/73c748436730dab39a79c579c6...
#####
# ##### Pouring node@10-10.16.2.mojave.bottle.tar.gz
#####
# ##### Caveats
node@10 is keg-only, which means it was not symlinked into /usr/local,
because this is an alternate version of another formula.

If you need to have node@10 first in your PATH run:
echo 'export PATH="/usr/local/opt/node@10/bin:$PATH"' >> ~/.zshrc

For compilers to find node@10 you may need to set:
export LDFLAGS="-L/usr/local/opt/node@10/lib"
export CPPFLAGS="-I/usr/local/opt/node@10/include"

==> Summary
^ /usr/local/Cellar/node@10/10.16.2: 4,144 files, 49.4MB
```

Once this step is completed, you will likely need to update your PATH manually by following the instructions in your terminal output to add node@10 before other instances of Node.js. As of this writing, there are some dependencies in the React Truffle Box (we'll need it later) that are not compatible with Node.js 12; therefore, we will stick with the most recent long-term support (LTS) version for this book.

To verify that the correct version has been installed, run the following command:

```
$ node -v
```

Now that the JavaScript environment is ready, we can install the Truffle Suite, which is used to create our smart contracts.

With Node.js all set, we are ready to consider tools that specifically relate to smart contract development: Truffle and Ganache.

Installing the Truffle Suite

We are in the home stretch of setting up our machines for smart contract development. The next two things we will install are part of the [Truffle Suite](#): Truffle and Ganache.

Truffle

Though Truffle is often referred to as a framework, it is really more of a blockchain utility belt. It provides tools that make compiling, testing, deploying, and packaging your application as easy as possible. These utilities rely on the generated directory structure, which is where the framework aspect comes into play, and the Truffle CLI.

Truffle is distributed as a npm package. To install it, run the following command:

```
$ npm install -g truffle
```

After running this command, you should see the following output:

```
/usr/local/bin/truffle -> /usr/local/lib/node_modules/truffle/build/cli.bundled.js

> truffle@5.0.31 postinstall /usr/local/lib/node_modules/truffle
> node ./scripts/postinstall.js

- Fetching solc version list from solc-bin. Attempt #1
- Downloading compiler. Attempt #1
+ truffle@5.0.31
added 27 packages from 439 contributors in 22.559s
```

Since we used the `-g` flag when running the `install` command, the output lets us know where to find the symlink that is creating the globally available executable.

To confirm everything is working, run the following command:

```
$ truffle -v
```

This should show that you are using version 5.0.31 and should list the available commands.

With Truffle installed, the last thing we need to do is add Ganache.

Ganache

Ganache is your very own blockchain. In many regards, it is very much like the Ethereum client we downloaded earlier. It provides tools for creating accounts and runs a JSON RPC API server for you to connect and read/write to the blockchain. The main difference is that it doesn't actually connect to the Ethereum network; in fact, older versions would destroy all data when you shut down so you had a clean slate when starting up again.

The full version of Ganache comes with a GUI that provides controls to update configuration settings and to explore blocks, transactions, and logs. You can also install a CLI-only version of Ganache through npm, but for now we will go ahead and install the full version.

Go to the [Ganache page](#) and there will be a button to download the version for your OS, as seen in [Figure 3-11](#).

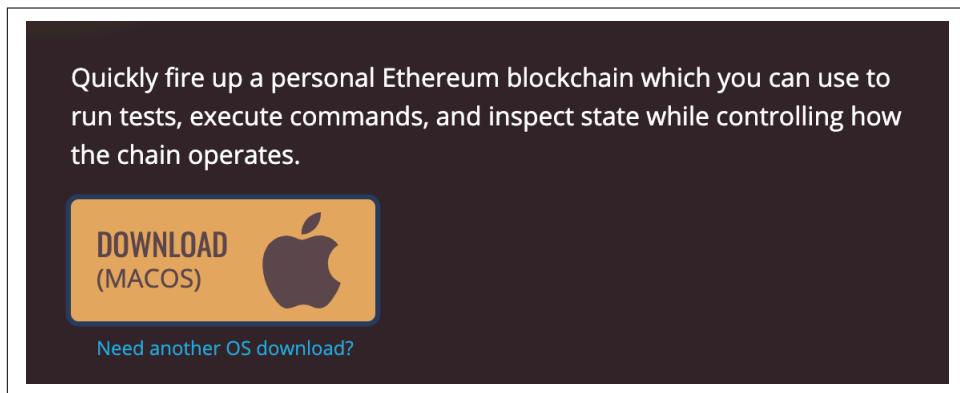


Figure 3-11. Ganache launch screen

After installation is complete, launch the application and you should see a launch screen like the one in [Figure 3-12](#).

With that piece in place, we have everything needed to begin developing smart contracts.



Figure 3-12. Installing Ganache

Summary

In this chapter, we set up a local environment for blockchain development. This included setting up a full Ethereum client with Parity, which included syncing the Goerli test network. We also installed the Chrome plug-in MetaMask for interaction with DApps from our browser. We then installed the JavaScript environment needed for our smart contract tools by using Node.js. Lastly, we installed both Truffle and Ganache from the Truffle Suite, giving us everything we need for smart contract development.

In the next chapter, we will develop a small application to explore the Truffle toolset and the Solidity programming language.

PART II

Developing Smart Contracts

In [Part II](#) we will begin exploring smart contract development, using the Solidity programming language. We will also gain experience with the Truffle framework, which provides tools for deploying and testing our contracts.

In order to limit the number of new topics, links to web-based user interfaces have been provided. Building these interfaces is the focus of [Part III](#).

Our First Smart Contract

Now that we have everything installed, it is time to build our first contract. In following the tradition of introductory programming books, our first program will greet us with “Hello, World!”

In developing this program, we are going to learn how to use the tools provided by Truffle for creating and testing our application. We will also begin to explore the Solidity language, including a look at functions and state variables.

Our goal for this chapter is to find a rhythm in how we go about building our application and discovering if we are on the right track. To help us there, we will adopt test-driven development (TDD) for that instant feedback loop.

Let start by setting up our project.

Setup

As we get set up, we will need a directory to hold our new application. Let’s first create a directory called *greeter* and change into our new directory. Open your terminal and use the following commands:

```
$ mkdir greeter  
$ cd greeter
```

We are now going to initialize a new Truffle project as follows:

```
$ truffle init
```

This command will generate the following output:

```
✓ Preparing to download  
✓ Downloading  
✓ Cleaning up temporary files  
✓ Setting up box
```

```
Unbox successful. Sweet!
```

Commands:

```
Compile:      truffle compile
Migrate:      truffle migrate
Test contracts: truffle test
```

Our *greeter* directory should now include the following files:

```
greeter
├── contracts
│   └── Migrations.sol
├── migrations
│   └── 1_initial_migration.js
└── test
    └── truffle-config.js
```

Notice that the commands specified in the output line up well with the directory structure generated when initializing our application. `truffle compile` will compile all the contracts in the *contracts* directory, `truffle migrate` will deploy our compiled contracts by running the scripts in our *migrations* directory, and lastly, `truffle test` will run the tests in our *test* directory.

The last item created for you is the *truffle-config.js* file. This is where we will place our application-specific configurations.

Now that we have our initial structure, we are ready to begin development.

Our First Test

As we implement features for our contracts, we will use [TDD](#) to take advantage of the short feedback loop it provides. If you are not familiar with TDD, it is a way of writing software where we first start with a failing test and then write the code required to make the test pass. Once everything is working, we can then refactor the code to make it more maintainable.

The test support Truffle provides is one of the areas where the toolbelt truly shines. It offers testing support in both JavaScript and Solidity; for our examples, we'll use JavaScript since it is far more widely adopted, which makes it easier to find additional resources should you get stuck. If you would like to explore writing tests in Solidity, you can reference the [Truffle testing documentation](#).

Our first test in [Example 4-1](#) is going to make sure our empty contract can deploy properly. This may seem unnecessary, but the errors we will experience in getting this to pass provide a great way to see some of the errors we are likely to encounter in our career.

In the `test` directory, create a file called `greeter_test.js`:

```
$ touch test/greeter_test.js
```

Then add the test code, as in [Example 4-1](#).

Example 4-1. Testing our contract can deploy

```
const GreeterContract = artifacts.require("Greeter"); ①

contract("Greeter", () => {
  it("has been deployed successfully", async () => { ②
    const greeter = await GreeterContract.deployed(); ③
    assert(greeter, "contract was not deployed"); ④
  });
});
```

- ➊ Truffle provides a way to load and interact with contracts that have been compiled through the `artifacts.require` function. Here, you will pass in the name of the contract, not the name of the file since a file may contain multiple contract declarations.
- ➋ Truffle tests use [Mocha](#), but with a twist. The `contract` function will act similar to the built-in `describe` but with the added benefit of using Truffle's clean room feature. This feature means that fresh contracts will be deployed before the tests nested within are executed. This helps prevent state from being shared between different test groups.
- ➌ Every interaction with the blockchain is going to be asynchronous, so instead of using Promises and the `Promise.prototype.then` method, we will take advantage of the [`async/await` syntax](#) now available in JavaScript.
- ➍ If the greeter is truthy (exists), our test will pass.

Running our tests, we will receive an error that looks like this:

```
$ truffle test
Compiling your contracts...
=====
> Compiling ./contracts/Migrations.sol

Error: Could not find artifacts for Greeter from any sources
  at Resolver.require (/usr/local/lib/node_modules/truffle/build/webpack:...
  at TestResolver.require (/usr/local/lib/node_modules/truffle/build/...
  at Object.require (/usr/local/lib/node_modules/truffle/build/webpack:...
...
...omitted..
```

```
Truffle v5.0.31 (core: 5.0.31)
Node v12.8.0
```

This gives us actionable feedback. The error tells us that after our contracts compiled, Truffle could not find one called Greeter. Since we have not created this contract yet, this error is perfectly reasonable. However, if you have already created a contract and are still getting this error, it is likely caused by a typo in the contract declaration found in the Solidity file or in the `artifacts.require` statement.

Let's create the Greeter file and add the code from [Example 4-2](#) to see what feedback running our test suite will provide.

In the terminal, create the Greeter file:

```
$ touch contracts/Greeter.sol
```

Example 4-2. Empty Greeter contract

```
pragma solidity >= 0.4.0 < 0.7.0; ①
contract Greeter { ②
}
```

- ① The `pragma` line is a compiler instruction. Here we tell the Solidity compiler that our code is compatible with Solidity version 0.4.0 up to, but not including, version 0.7.0.
- ② The contracts in Solidity are very similar to classes in object-oriented programming languages. The data and functions or methods defined within the contract's opening and closing curly braces will be isolated to that contract.

With these changes made, let's run our tests again:

```
$ truffle test
Compiling your contracts...
=====
> Compiling ./contracts/Greeter.sol
> Compiling ./contracts/Migrations.sol
```

```
Contract: Greeter
1) has been deployed successfully
  > No events were emitted
```

```
0 passing (34ms)
1 failing
```

```
1) Contract: Greeter
  has been deployed successfully:
Error: Greeter has not been deployed to detected network...
at Object.checkNetworkArtifactMatch (/usr/local/lib/node_modules/...
at Function.deployed (/usr/local/lib/node_modules/truffle/build/...
at processTicksAndRejections (internal/process/task_queues.js:85:5)
at Context.<anonymous> (test/greeter_test.js:5:21)
```

The error here indicates that our contract does not yet exist on the network; in other words, it has not yet been deployed. Every time we run the `truffle test` command, Truffle first compiles our contracts, and then deploys them to a test network. In order to deploy our contract, we need to turn to another tool provided by the Truffle toolbelt: migrations.

Migrations are scripts written in JavaScript that are used to automate the deployment of our contracts. The default `Migrations` contract found in `contracts/Migrations.sol` is the contract that is deployed by `migrations/1_initial_migration.js` and is currently the only contract that has made its way to the test network. In order to add our Greeter contract to the network, we will need to create a migration using the code found in [Example 4-3](#).

First, we will need to create the file to hold our migrations code:

```
$ touch migrations/2_deploy_greeter.js
```

Then we can add the code in [Example 4-3](#).

Example 4-3. Deploying the Greeter contract

```
const GreeterContract = artifacts.require("Greeter");

module.exports = function(deployer) {
  deployer.deploy(GreeterContract);
}
```

Our initial migration does not have much going on. We use the provided `deployer` object to deploy the Greeter contract. For now, this is all we'll need to do in order to have our contract available on the local test network, but don't worry: we will go deeper into migrations in the next chapter. With that in place, run the tests one more time:

```
$ truffle test
Compiling your contracts...
=====
> Compiling ./contracts/Greeter.sol
> Compiling ./contracts/Migrations.sol
```

Contract: Greeter

✓ has been deployed successfully

1 passing (27ms)

Success! This test lets us know we have everything set up correctly and we are ready to begin implementing features.

Saying Hello

This is normally the point where we would use a call to some variant of `printf` or `println` to output our greeting, but in Solidity we do not have access to standard out, or the file system, the network, or any other input/output (I/O). What we do have are functions.

Once deployed, our smart contract will be stored on the Ethereum network at a specific address. It will be dormant until a request comes in asking it to perform some work and the work our contract can do is defined by our functions. What we want is a function that can say “Hello, World!” and just like before, we will start with a test.

In `test/greeter_test.js`, let’s add the test from [Example 4-4](#).

Example 4-4. Testing for Hello, World!

```
describe("greet()", () => {
  it("returns 'Hello, World!', async () => {
    const greeter = await GreeterContract.deployed();
    const expected = "Hello, World!";
    const actual = await greeter.greet();

    assert.equal(actual, expected, "greeted with 'Hello, World!'");
  });
});
```

In this case, we set an expected value, and then retrieve the value from our contract and see if they are equal. We have to mark the test function as `async` since we are going to make a call to our local test blockchain to interact with this contract.

Running the test results in the following:

```
$ truffle test
Compiling your contracts...
=====
> Compiling ./contracts/Greeter.sol
> Compiling ./contracts/Migrations.sol
```

Contract: Greeter

```

✓ has been deployed successfully
greet()
  1) returns 'Hello, World!'
> No events were emitted

1 passing (42ms)
1 failing

1) Contract: Greeter
  greet()
    returns 'Hello, World!':
TypeError: greeter.greet is not a function
  at Context.<anonymous> (test/greeter_test.js:13:36)
  at processTicksAndRejections (internal/process/task_queues.js:85:5)

```

When we focus in on the error, we see that `greeter.greet` is not a function. It's time to add a function to our contract. Update the Greeter contract with the function in [Example 4-5](#).

Example 4-5. Adding the `greet` function to Greeter

```

pragma solidity >= 0.4.0 < 0.7.0;

contract Greeter {

  function greet() external pure returns(string memory) {
    return "Hello, World!";
  }
}

```

Here we created a function with the identifier or name `greet`, which does not take any parameters. Following the identifier, we indicated that our function is an `external` function. This means that it is part of our contract's interface and can be called from other contracts, or from transactions, but cannot be called from within the contract or at least not without an explicit reference to the object it is being called on. Our other options here are `public`, `internal`, and `private`.

`public` functions are also part of the interface, meaning they can be called from other contracts or transactions, but additionally they can be called internally. This means you can use an implicit receiver of the message when invoking the method inside of a method.

`internal` and `private` functions must use the implicit receiver or, in other words, cannot be called on an object or on `this`. The main difference between these two modifiers is that `private` functions are only visible within the contract in which they are defined, and not in derived contracts.

Functions that will not alter the state of the contract's variables can be marked as either `pure` or `view`. `pure` functions do not read from the blockchain. Instead, they operate on the data passed in or, as in our case, data that did not need any input at all. `view` functions are allowed to read data from the blockchain, but again they are restricted in that they cannot write to the blockchain.

After our declaration that this function is `pure`, we identify what we expect our function to return. Solidity does allow multiple return values, but in our case we will only have one value being returned: the `string` type. We also indicate that this is a value that is not referencing anything located in our contract's persisted storage by using the keyword `memory`.

The body of our function returns the string we are looking for, "Hello, World!" This should satisfy the requirements of our test. But don't take our word for it—run the tests again to verify:

```
$ truffle test
Compiling your contracts...
=====
> Compiling ./contracts/Greeter.sol
> Compiling ./contracts/Migrations.sol
```

```
Contract: Greeter
✓ has been deployed successfully
greet()
✓ returns 'Hello, World!' (51ms)
```

```
2 passing (82ms)
```

With this test passing, let's move on to making our contract a little more flexible by giving our users the ability to change the greeting.

Making Our Contract Dynamic

Now that our contract has returned a hardcoded value, let's continue and make the greeting dynamic. In order to do this, we need to add another function that allows us to set the message that will be returned by our `greet()` function.

Earlier we mentioned the "clean room" feature when using the `contract` function in our tests. This feature will deploy new instances of our contracts for use within the callback function for that block of code. In the test we are about to write, we want to make sure our state changes stay isolated from the rest of the tests so that we do not find ourselves in a situation where the order of our tests will impact the success or

failure of our test suite. In order to do this, we will create another `contract` block in our `test/greeter-test.js` file, as illustrated in [Example 4-6](#).

Example 4-6. Testing the greeting can be made dynamic

```
const GreeterContract = artifacts.require("Greeter");

contract("Greeter", () => {
  it("has been deployed successfully", async () => {
    const greeter = await GreeterContract.deployed();
    assert(greeter, "contract failed to deploy");
  });

  describe("greet()", () => {
    it("returns 'Hello, World!'", async () => {
      const greeter = await GreeterContract.deployed();
      const expected = "Hello, World!";
      const actual = await greeter.greet();

      assert.equal(actual, expected, "greeted with 'Hello, World!'");
    });
  });
});

contract("Greeter: update greeting", () => {
  describe("setGreeting(string)", () => {
    it("sets greeting to passed in string", async () => {
      const greeter = await GreeterContract.deployed()
      const expected = "Hi there!";

      await greeter.setGreeting(expected);
      const actual = await greeter.greet();

      assert.equal(actual, expected, "greeting was not updated");
    });
  });
});
```

You'll recognize the setup is much like our previous test. We set a variable to hold our expected return value, which is the string we will also pass to the `setGreeting` function. We then update the greeting and ask the `greet` for its return value. These are both asynchronous calls requiring us to use the `await` keyword. Lastly, we check the value from `greet` against our expected value.

When running the tests, we get the following output:

```

$ truffle test
Compiling your contracts...
=====
> Compiling ./contracts/Greeter.sol
> Compiling ./contracts/Migrations.sol


Contract: Greeter
  ✓ has been deployed successfully
    greet()
      ✓ returns 'Hello, World!' (48ms)

Contract: Greeter: update greeting
  setGreeting(string)
    1) sets greeting to passed in string
  > No events were emitted


2 passing (111ms)
1 failing

1) Contract: Greeter: update greeting
   setGreeting(string)
     sets greeting to passed in string:
     TypeError: greeter.setGreeting is not a function
       at Context.<anonymous> (test/greeter_test.js:26:21)
       at processTicksAndRejections (internal/process/task_queues.js:85:5)

```

Our tests indicate that the `setGreeting` function does not yet exist; let's add this function to our contract. Back in our `contracts/Greeter.sol` file, after our `greet` function, add the function signature from [Example 4-7](#).

Example 4-7. Adding `setGreeting()` to Greeter

```
function setGreeting(string calldata greeting) external {
}
```

Our `setGreeting` function is intended to update the state of our contract with a new greeting, which means we need to accept a parameter for this new value. This new value is expected to be a string, and will be referred by the identifier `greeting`. Just like our `greet` function, this function is intended to be called from external scripts or other contracts and will not be referred to internally.

Because this function is being called from the outside world, the data being passed in as a parameter is not part of the contract's persisted storage, but is included as part of the `calldata` and must be labeled with the data location `calldata`. The `calldata` location is only needed when the function is declared as external and when the data type

of the parameter is a reference type such as a mapping, struct, string, or array. Using value types like `int` or `address` do not require this label.

With the function declared, if we run our tests now, we will get the following output:

```
$ truffle test
Compiling your contracts...
=====
> Compiling ./contracts/Greeter.sol
> Compiling ./contracts/Migrations.sol

Contract: Greeter
  ✓ has been deployed successfully
    greet()
      ✓ returns 'Hello, World!' (45ms)

Contract: Greeter: update greeting
  setGreeting(string)
    1) sets greeting to passed in string
    > No events were emitted

2 passing (215ms)
1 failing

1) Contract: Greeter: update greeting
   setGreeting(string)
     sets greeting to passed in string:

     greeting was not updated
     + expected - actual

     -Hello, World!
     +Hi there!

     at Context.<anonymous> (test/greeter_test.js:29:14)
     at processTicksAndRejections (internal/process/task_queues.js:85:5)
```

Reviewing the test failure, we learn that the value returned by the `greet` function was not what was expected. Our function does not yet do anything and therefore the greeting never changed.

In order to update a variable in one function, and have that variable be available in another function, we'll need to store data in the contract's persisted storage by using a state variable, as demonstrated in [Example 4-8](#).

In our `contracts/Greeter.sol` file, add the code in [Example 4-8](#) to the Greeter contract.

Example 4-8. Adding state variable to Greeter contract

```
pragma solidity >= 0.4.0 < 0.7.0;

contract Greeter {
    string private _greeting;

    function greet() external pure returns(string memory) {
        return "Hello, World!";
    }

    function setGreeting(string calldata greeting) external {
    }
}
```

State variables will be available to all functions defined inside of a contract, similar to instance variables or member variables of other object-oriented languages. They are also where we will store data that will exist for the entire lifetime of our contract. Like functions, state variables can be declared with different levels of visibility modifiers, including `public`, `internal`, and `private`. In our previous example, we used the `private` modifier, which means this variable is accessible in our Greeter contract.



All data on the blockchain is publicly visible from the outside world. State variable modifiers only restrict how the data can be interacted with from within the contract or other contracts.

When writing a function that updates a state variable, as our `setGreeting` is about to do, we cannot name the parameter the same as our state variable. To avoid these types of conflicts, it is common practice to prefix state variable or parameter names with an underscore (`_`) character.

Let's update the `setGreeting` function to set the state variable, as shown in [Example 4-9](#).

Example 4-9. Updating state variable

```
function setGreeting(string calldata greeting) external {
    _greeting = greeting;
}
```

Even with this change, our test would still fail. What we now want to do is to update the `greet` function to read from this state variable, but there are some things we need to consider before making this change. The first is that our function is currently

marked as `pure`. We will need to update the function to be a `view` function since we are now going to access data stored on the blockchain. Once we switch to reading from the state variable in our `greet` function, we will no longer have our default greeting and the initial test will fail. To address these issues, we will give `greeting` a default of “Hello, World!” We will also change the function from `pure` to `view` and update the return value to use the value stored in `greeting`. [Example 4-10](#) shows our contract with all these changes made.

Example 4-10. Reading from our state variable

```
pragma solidity >= 0.4.0 < 0.7.0;

contract Greeter {
    string private _greeting = "Hello, World!";

    function greet() external view returns(string memory) {
        return _greeting;
    }

    function setGreeting(string calldata greeting) external {
        _greeting = greeting;
    }
}
```

After running our tests, we'll now see all three tests are passing!

```
$ truffle test
Compiling your contracts...
=====
> Compiling ./contracts/Greeter.sol
> Compiling ./contracts/Migrations.sol

Contract: Greeter
✓ has been deployed successfully
greet()
✓ returns 'Hello, World!' (57ms)

Contract: Greeter: update greeting
setGreeting(string)
✓ sets greeting to passed in string (116ms)

3 passing (224ms)
```

Making the Greeter Ownable

As it stands now, anyone can change the message of our Greeter contract. This may be fine in some cases, but it could also lead to someone changing the message to something less than welcoming. To prevent this, we will now add the idea of ownership to the contract, and then restrict the ability to change the greeting to the owner.

In order to do this, we want to set the owner of the Greeter contract to the address that deployed the contract. This means we'll need to store the address during initialization, and for that, we'll need to write a constructor function. We will also need to access some information from the `msg` object. The `msg` object is globally available and includes the calldata, sender of the message, the signature of the function being called, and the value (how much wei was sent).

Our first test is going to assert that an owner exists by invoking an `owner` getter function. Since this is not depending on changing state on anything, we will put this test in the initial Greeter block of tests, as shown in [Example 4-11](#).

Example 4-11. Testing an owner exists

```
const GreeterContract = artifacts.require("Greeter");

contract("Greeter", () => {
  it("has been deployed successfully", async () => {
    const greeter = await GreeterContract.deployed();
    assert(greeter, "contract failed to deploy");
  });

  describe("greet()", () => {
    it("returns 'Hello, World!'", async () => {
      const greeter = await GreeterContract.deployed();
      const expected = "Hello, World!";
      const actual = await greeter.greet();

      assert.equal(actual, expected, "greeted with 'Hello, World!'");
    })
  });

  describe("owner()", () => {
    it("returns the address of the owner", async () => {
      const greeter = await GreeterContract.deployed();
      const owner = await greeter.owner();

      assert(owner, "the current owner");
    });
  });
})
```

Running this test will generate the following failure:

```
$ truffle test

Compiling your contracts...
=====
> Compiling ./contracts/Greeter.sol
> Compiling ./contracts/Migrations.sol


Contract: Greeter
  ✓ has been deployed successfully
    greet()
      ✓ returns 'Hello, World!' (54ms)
    owner()
      1) returns the address of the owner
    > No events were emitted

Contract: Greeter: update greeting
  setGreeting(string)
    ✓ sets greeting to passed in string (274ms)

3 passing (409ms)
1 failing

1) Contract: Greeter
  owner()
    returns the address of the owner:
    TypeError: greeter.owner is not a function
      at Context.<anonymous> (test/greeter_test.js:22:35)
      at processTicksAndRejections (internal/process/task_queues.js:85:5)
```

This failure looks familiar. We do not have a function defined on our Greeter contract called `owner`. Since this is a getter function, we'll need to add a state variable that will hold the address of the owner, and then our function should return that address. The Solidity language provides two `address` types: one is `address` and the other is `address payable`. The difference between them is that `address payable` gives access to the `transfer` and `send` methods, and variables of this type can also receive ether. We are not sending ether to this address and we can use the `address` type for our purposes.

Let's go ahead and update the Greeter contract with these changes, as illustrated in [Example 4-12](#).

Example 4-12. Adding the ownership state variable and getter function

```
pragma solidity >= 0.4.0 < 0.7.0;

contract Greeter {
  string private _greeting = "Hello, World!";
```

```

address private _owner;

function greet() external view returns(string memory) {
    return _greeting;
}

function setGreeting(string calldata greeting) external {
    _greeting = greeting;
}

function owner() public view returns(address) {
    return _owner;
}
}

```

Running our tests show that we are once again passing:

```

Compiling your contracts...
=====
> Compiling ./contracts/Greeter.sol
> Compiling ./contracts/Migrations.sol

Contract: Greeter
✓ has been deployed successfully
greet()
    ✓ returns 'Hello, World!' (57ms)
owner()
    ✓ returns the address of the owner (50ms)

Contract: Greeter: update greeting
setGreeting(string)
    ✓ sets greeting to passed in string (125ms)

4 passing (292ms)

```

What we really want to test here is that the owner address is the same as the deploying address. To do that we'll now add a test to check that the ownership address is equal to the default account (the one used to deploy the contract). To do this, we'll need access to the accounts in our test environment, and luckily Truffle has made this accessible to use via the `accounts` variable. We'll need to pass this in to the `contract` level block of test code, as illustrated in [Example 4-13](#).

Example 4-13. Test owner is the same as deployer

```

const GreeterContract = artifacts.require("Greeter");

contract("Greeter", (accounts) => {
    it("has been deployed successfully", async () => {

```

```

const greeter = await GreeterContract.deployed();
assert(greeter, "contract failed to deploy");
});

describe("greet()", () => {
  it("returns 'Hello, World!'", async () => {
    const greeter = await GreeterContract.deployed();
    const expected = "Hello, World!";
    const actual = await greeter.greet();

    assert.equal(actual, expected, "greeted with 'Hello, World!'");
  })
});

describe("owner()", () => {
  it("returns the address of the owner", async () => {
    const greeter = await GreeterContract.deployed();
    const owner = await greeter.owner();

    assert(owner, "the current owner");
  });

  it("matches the address that originally deployed the contract", async () => {
    const greeter = await GreeterContract.deployed();
    const owner = await greeter.owner();
    const expected = accounts[0];

    assert.equal(owner, expected, "matches address used to deploy contract");
  });
});
});
}

```

Notice that our `contract` block is now passing in an `accounts` parameter to the function containing our test cases. Our new test is then asserting that the first account is the one that deployed the Greeter contract. When we run the tests, we will get a failure that informs us the `owner` and `expected` do not match. Now it's time to write that constructor function.

Up until now we have been using the default constructor, `constructor() public {}`. Now we need to record the `msg.sender` as the owner when the contract is initialized. Our Greeter contract should now look like [Example 4-14](#).

Example 4-14. Adding constructor to the Greeter contract

```

pragma solidity >= 0.4.0 < 0.7.0;

contract Greeter {
  string private _greeting = "Hello, World!";
  address private _owner;

```

```

constructor() public {
    _owner = msg.sender;
}

function greet() external view returns(string memory) {
    return _greeting;
}

function setGreeting(string calldata greeting) external {
    _greeting = greeting;
}

function owner() public view returns(address) {
    return _owner;
}
}

```

With this change, our tests are once again passing.

Now that we know who created the contract, we can create a restriction that only the owner can update the greeting. This type of access control is normally done with a function modifier.

Function modifiers allow us to extend a function with code that can run before and/or after a function. These typically take the form of a guard clause and will prevent the function from being invoked if the clause is not met, which is exactly what we want for our Greeter contract.

Example 4-15 has updated the `setGreeting` tests, setting the expectation that only the owner can change the greeting.

Example 4-15. Test restricting setGreeting to owner

```

contract("Greeter: update greeting", (accounts) => {
    describe("setGreeting(string)", () => {
        describe("when message is sent by the owner", () => {
            it("sets greeting to passed in string", async () => {
                const greeter = await GreeterContract.deployed()
                const expected = "The owner changed the message";

                await greeter.setGreeting(expected);
                const actual = await greeter.greet();

                assert.equal(actual, expected, "greeting updated");
            });
        });
    });

    describe("when message is sent by another account", () => {
        it("does not set the greeting", async () => {
            const greeter = await GreeterContract.deployed()
            const expected = await greeter.greet();
        });
    });
})

```

```

try {
    await greeter.setGreeting("Not the owner", { from: accounts[1] });
} catch(err) {
    const errorMessage = "Ownable: caller is not the owner"
    assert.equal(err.reason, errorMessage, "greeting should not update");
    return;
}
assert(false, "greeting should not update");
});
});
}

```

In our updated tests, we again pass in the `accounts` parameter to the `contract` callback function to make them available to our test cases. We also added a second section for the case when a non-owner account sends the message. We are now expecting an error to be raised when a non-owner address attempts to make this change. If you look at the `setGreeting` call, there is now a second parameter being passed in; it's an object with a `from` property, and we are explicitly sending this message from a different account. This is also how we could set a `value` (in units of wei) to be sent to the contract as well.

Running our tests should result in a failure like this:

```

1 failing

1) Contract: Greeter: update greeting
    setGreeting(string)
        when message is sent by another account
            does not set the greeting:
                AssertionException: greeting should not update

```

Let's create and apply our modifier to fix this right up. Back in our Greeter contract, after the constructor, add the following code:

```

modifier onlyOwner() {
    require(
        msg.sender == _owner,
        "Ownable: caller is not the owner"
    );
    _;
}

```

The modifier syntax looks very much like function syntax but without the visibility declaration. Here, our modifier is using the `require` function, where the first argument is an expression that will evaluate to a boolean. When this expression results in a false, the transaction is completely reverted, meaning all state changes are reversed and the program stops execution. The `revert` function also takes an optional string parameter that can be used to give more information to the caller as to why the operation failed.

The last part of our modifier function is the `_`; line. This line is where the function that is being modified will be called. If you put anything after this line, it will be run after the function body completes.

Now let's update our `setGreeter` function to use the modifier by adding the `onlyOwner` declaration after `external` in the function definition:

```
function setGreeting(string calldata greeting) external onlyOwner {
    _greeting = greeting;
}
```

Running our tests, we see they are all passing!

This is great, but we are going to make one final change before we end this chapter. In [Chapter 2](#), we discussed OpenZeppelin and how they have created contracts that can be used as the basis for creating tokens. Well, they also have contracts that implement the idea of ownership, and we are duplicating some of that behavior. Instead of duplicating it, we will update our Greeter contract to leverage their implementation.

Back in our terminal, in the root directory of our application, enter the following command:

```
$ npm install openzeppelin-solidity
```

Once that has completed, update the top of the Greeter contract to look like [Example 4-16](#).

Example 4-16. Inheriting from Ownable

```
pragma solidity >= 0.4.0 < 0.7.0;

import "openzeppelin-solidity/contracts/ownership/Ownable.sol";

contract Greeter is Ownable {
    ...rest of file...
```

Here, we added an import statement that will pull in all the global symbols from the imported file, such as `Ownable`, and make them available in the current scope. The next thing to notice is that our Greeter contract now inherits from `Ownable` through the `is` syntax. Solidity supports multiple inheritance much like Python or even C++. The inheriting classes are listed with a comma separating them.

With this in place, we can remove our implementations of the `onlyOwner` modifier, the `owner` getter function, and the constructor function since `Ownable` provides these definitions. This may seem like overkill, and normally I would agree with that thought. However, using code that has been through a thorough audit and well tested is prudent when working with smart contracts since security of smart contracts is critically important.

This wraps up our Greeter contract. But before we move on, let's reflect on what we have learned in this chapter.

Summary

So far in our journey, we have learned how to create a new smart contract project using `truffle init`, which provided the directory structure for our application. This structure includes directories to house our contracts, tests, and migrations.

We wrote a deployment test that helped guide us through the initial migration needed to get our contract onto the test network. This allowed our subsequent tests to interact with the deployed contract.

Lastly, we started exploring the Solidity language and learned about the different function visibility modifiers (`external`, `public`, `internal`, `private`). Those same modifiers are also available for state variables (with the exception of `external`) used in persisting data on the blockchain. We also implemented the concept of `Ownable` and refactored to using the `OpenZeppelin` contract via inheritance.

In the next chapter, we are going to dive into how to deploy our contract locally and also onto one of the publicly available test networks.

Deploying and Interacting with Contracts

Now that we have completed development of our Greeter contract, it is time to look into how to deploy it.

In this chapter, we are going to learn how to deploy our contract three different ways. First, we will focus on deploying to Ganache, a local blockchain that will allow you to experiment with your application quickly.

Then, we will deploy our contract to the Goerli test network using the Parity Ethereum client. This process is how you would deploy your application directly to the Ethereum network by using a node you are managing yourself.

Lastly, we will deploy our contract to the Rinkeby test network using Infura, a third-party provider of managed Ethereum nodes. Using services like Infura can provide some additional analysis that may be helpful in understanding application performance and utilization.

By the end of this chapter, you will be able to start sharing your application with the decentralized community.

Contract Compilation and Deployment

Up to this point, the only way we have interacted with our contract is through our tests. When we have been running our test suite, Truffle has been compiling and deploying our contracts to a test network and executing our test scripts. When the tests have completed their run, Truffle destroys the test network and all the associated data.

If we want to use our contracts with something other than our tests, we will need to get our contract onto a network that runs outside of the test process. To do this, we use the `truffle migrate` command. Unlike `truffle test`, we will need to specify

which network we are going to deploy our contracts using the `--network` flag. Truffle will then look up the configuration details for the corresponding network in our `truffle-config.js` file and deploy our contracts based on those settings.

Before we jump to configuring and deploying our contracts, let's first take a moment to discuss what happens when we deploy a smart contract.

Deployment Process

The first step in deployment starts with compiling our contracts. For both the `truffle test` and `truffle migrate` commands, this step happens for you automatically. However, we can compile our contracts at anytime using `truffle compile`.

Let's go ahead and compile our contracts. Back in our `greeter` directory, open a terminal and use the following command:

```
$ truffle compile
```

We should see output that indicates the contracts that have been compiled and the location, relative to our project directory, where the files have been saved.

The output looks like this:

```
Compiling your contracts...
=====
> Compiling ./contracts/Greeter.sol
> Compiling ./contracts/Migrations.sol
> Compiling openzeppelin-solidity/contracts/ownership/Ownable.sol
> Artifacts written to ./build/contracts
> Compiled successfully using:
  - solc: 0.5.8+commit.23d335f2.Emscripten.clang
```

Looking at the `build/contracts` directory, we have three new files that have been generated for us:

```
build
└── contracts
    ├── Greeter.json
    ├── Migrations.json
    └── Ownable.json
```

If we take a look at the the `build/contracts/Greeter.json` file, we will see a JSON formatted data structure. The first field we come across is the `contractName` field. Not surprisingly, this matches the name of our contract. Though there is a lot of data here, we are going to focus on two fields, the `abi` and `bytecode` fields.

The application binary interface (ABI) describes the functions and events (we will cover events later) of our contract. The ABI will be the basis for the client-side abstraction used to interact with our contract, which will be covered in detail in [Part III](#).

The `bytecode` field contains the result of compiling your contract. This is the code the Ethereum network will execute when our contract has been invoked from a client.

With our compiled contracts in hand, let's talk about what happens during deployment.

When we deploy a contract, we submit a transaction to the Ethereum network. Unlike a normal transaction that contains both a sender (`from`) and receiver (`to`) address, our deployment transaction will need to set the receiving address to the `0x0` address. Our deployment transaction will also need to include the bytecode mentioned earlier, which will be sent as the transaction data. With our contract being sent as a transaction, it has to be mined before we will be able to interact with it. When the contract is mined, it will execute the code in the constructor, setting the initial state for the contract.

Before we deploy, let's first set up our UI.



Warning: Deployment Can Become Expensive

Deploying costs gas. Depending on the current value of ether and the size of your contract, it could be a very expensive operation. Make sure you test your contracts locally and on the public test networks before deploying to the mainnet.

Setting Up the UI

Later in this part of the book we dive into building the UIs for our applications, and we'll borrow some of the code from those chapters. The book's code is located on [GitHub](#), and we will copy into our Greeter project the code found in the *chapter-9* directory code. Let's start by cloning the directory. The following commands assume you are at the Greeter project's root directory:

```
# go to the directory containing our greeter project
$ cd ..
# clone the repo
$ git clone git@github.com:RedSquirrelTech/hoscdev.git
```

Now let's copy the client code from [Chapter 9](#) into our Greeter project:

```
$ cp -r hoscdev/chapter-9/client greeter/client
```

This will copy all the UI-related code into our project. The UI code depends on the build files that were generated when we compiled our contract. In order for them to be accessible, we are going to update where Truffle puts the artifacts of compilation. This is a setting that can be adjusted in the `truffle-config.js` file by adding the following setting:

```
module.exports = {
  contracts_build_directory: "./client/src/contracts",
  // ...rest of settings...
}
```

Back in our *greeter* directory, let's recompile the contracts to get them in their new home:

```
$ truffle compile
```

After we have double-checked everything worked as expected, our */client/src/contracts/* directory should contain the following files:

```
client/src/contracts
├── Greeter.json
├── Migrations.json
└── Ownable.json
```

The client was written using React and uses a few other JS dependencies we haven't downloaded just yet. Let's change to the *client* directory and get everything installed:

```
$ cd greeter/client
$ npm install
```

Once all the dependencies are installed, you can try running the web server to allow your browser to connect to the UI with the following command:

```
# in the greeter/client directory
$ npm run start
```

This command will launch a new tab in your browser and attempt to connect to your Ethereum accounts via MetaMask. At this point, this will crash since the contracts have not been deployed to the network. It may seem odd, but this failure is a win. This means that our UI was built successfully and it tried to communicate with our contract, but it wasn't there. Let's fix that problem and do our first deploy by using the local test blockchain, Ganache.

Deploying to Ganache

Now that we have our UI in place, let's get our contracts set up on Ganache.

In the *truffle-config.js* file, located in the root directory of our Greeter project, we will see a configuration object that has a network configuration as depicted in Example 5-1.

Example 5-1. Truffle network configuration

```
networks: {
  // Useful for testing. The `development` name is special - truffle
  // uses it by default if it's defined here and no other network
  // is specified at the command line.
```

```

// You should run a client (like ganache-cli, Geth, or Parity) in a separate
// terminal tab if you use this network and you must also set the `host`,
// `port` and `network_id` options below to some value.
//
// development: {
//   host: "127.0.0.1",      // Localhost (default: none)
//   port: 8545,              // Standard Ethereum port (default: none)
//   network_id: "*",         // Any network (default: none)
// },
//
// Another network with more advanced options...
// advanced: {
//   port: 8777,                // Custom port
//   network_id: 1342,          // Custom network
//   gas: 8500000,              // Gas sent with each transaction (default: ~6700000)
//   gasPrice: 200000000000,    // 20 gwei (in wei) (default: 100 gwei)
//   from: <address>,          // Account to send txs from (default: accounts[0])
//   websockets: true,          // Enable EventEmitter interface for
//                             // web3 (default: false)
// },
//
// Useful for deploying to a public network.
// NB: It's important to wrap the provider as a function.
// ropsten: {
//   provider: () =>
//     new HDWalletProvider(mnemonic, `https://ropsten.infura.io/${infuraKey}`),
//   network_id: 3,              // Ropsten's id
//   gas: 5500000,               // Ropsten has a lower block limit than mainnet
//   confirmations: 2,            // # of confs to wait between deployments. (default: 0)
//   timeoutBlocks: 200,          // # of blocks before a deployment times out
//                             // (minimum/default: 50)
//   skipDryRun: true,            // Skip dry run before migrations?
//                             // (default: false for public nets )
// },
//
// Useful for private networks
// private: {
//   provider: () => new HDWalletProvider(mnemonic, `https://network.io`),
//   network_id: 2111,             // This network is yours, in the cloud.
//   production: true,             // Treats this network as if it was a public net.
//                             // (default: false)
// }
},

```



If you haven't already downloaded Ganache, you can grab the latest release from [Truffle Suite's GitHub](#) repository.

Looking at our configuration file, Truffle created a commented-out section for the development network. Since these commented values are what we want, we can go ahead and uncomment them by removing the `//` leading each line of the development section. With that out of the way, we can open up the Ganache application, and you should see a screen similar to the one in [Figure 5-1](#).



Figure 5-1. Ganache network configuration

Click on the New Workspace button, and you will be taken to a configuration screen asking for the name of the workspace and the Truffle project. Let's name the workspace `greeter-home`. Next, click on the Add Project button to navigate to the `truffle-config.js` file in the project. When you're all done, your screen should look similar to [Figure 5-2](#).

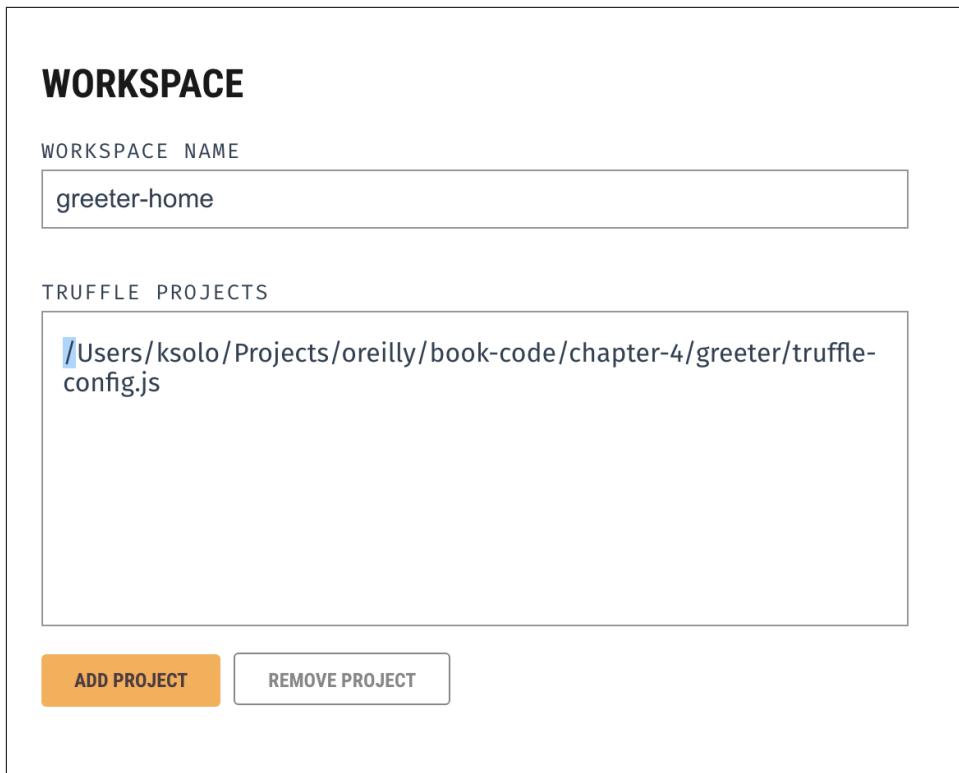


Figure 5-2. Configuring the Ganache workspace

Clicking on Save Workspace will set up the space and then take you to the main screen with all the accounts and server information, as depicted in Figure 5-3.

MNEMONIC	HD PATH			
audit frog student pink bonus tree short domain recycle worth reason produce	m/44'/60'/0'/0/account_index			
ADDRESS 0x0eF75E7c48bd3F0eC90731C6a69B62a00B5C87De	BALANCE 100.00 ETH	TX COUNT 0	INDEX 0	
ADDRESS 0x77565d9E0F142Dfa980929b232db912269e957b4	BALANCE 100.00 ETH	TX COUNT 0	INDEX 1	

Figure 5-3. Ganache network configuration

Now that Ganache is running, we will need to import the accounts into MetaMask using the depicted mnemonic. If you are logged in to MetaMask with your secured

account, log out and then click on “Import using account seed phrase” to unlock the Ganache-generated accounts.

Before we run the migration command, let’s first take a moment to discuss what is about to happen. When we created our project using `truffle init`, a Migrations contract was generated for us in the `contracts/` directory. This contract stores what migrations in the `migrations/` directory have been executed, allowing subsequent calls to `truffle migrate` to skip running those migrations. With this being our first deployment, we will run both deployment scripts in the `migrations/` directory. The first script deploys the Migrations contract, and the second script deploys our Greeter contract.

Without further ado, let’s deploy our contracts to the development network:

```
$ truffle migrate --network development
```

Once executed, we will see output similar to this example:

```
Compiling your contracts...
=====
> Everything is up to date, there is nothing to compile.
```

```
Starting migrations...
=====
> Network name:    'development'
> Network id:      5777
> Block gas limit: 0x6691b7

1_initial_migration.js
=====

Deploying 'Migrations'
-----
> transaction hash: 0x297a3e1c371adc4c25376e0df5bdff024af0ebd01c5db702...
> Blocks: 0          Seconds: 8
> contract address: 0x310C9d3709158d256DD0cD04958207aD772Df83e
> block number:     26
> block timestamp:  1565838314
> account:          0x0eF75E7c48bd3F0eC90731C6a69B62a00B5C87De
> balance:          99.99477214
> gas used:         261393
> gas price:        20 gwei
> value sent:       0 ETH
> total cost:       0.00522786 ETH

> Saving migration to chain.
> Saving artifacts
-----
```

```

> Total cost:          0.00522786 ETH

2_deploy_greeter.js
=====
Deploying 'Greeter'
-----
> transaction hash: 0x6bc7882ef6b7b0aa5cc532ba31ea1d665f211ba0024fe1f...
> Blocks: 0           Seconds: 12
> contract address: 0x81a0970E3c27C8CbEb85a01Fa3FB190197a1a3EC
> block number:      28
> block timestamp:   1565838344
> account:           0x0eF75E7c48bd3F0eC90731C6a69B62a00B5C87De
> balance:            99.98097166
> gas used:          648001
> gas price:          20 gwei
> value sent:         0 ETH
> total cost:         0.01296002 ETH

> Saving migration to chain.
> Saving artifacts
-----
> Total cost:          0.01296002 ETH

Summary
=====
> Total deployments:  2
> Final cost:          0.01818788 ETH

```

Looking through this output, we can confirm that we first deployed the default Migrations contract and see how much gas was used. Then we see the deployment of our Greeter contract and again how much gas was used. With our contracts deployed to the local blockchain, they are now ready to use.

Restart the web server that we were running in the client directory by using Control-C to stop the server and the following command to restart it:

```
$ npm run start
```

Now when we visit `localhost:3000`, we will see our Greeter UI with the current greeting and an input box ready for us to change the greeting, as depicted in Figure 5-4.



Figure 5-4. Seeing our Greeter UI

As long as you are using the first account, this will allow you to update it, but if you switch to another account, the transaction should be rejected. Try it out: switch accounts in MetaMask and then refresh the page and attempt to change the message.

Ganache is great for testing your application out locally, but when it comes time for others to experiment with your application, we'll need to deploy to one of the test networks. Our next deployment will walk us through deploying on the Goerli test network, and we'll do the deployment using the Parity client.

Deploying to Goerli with Parity

Before we begin, let's talk about the mnemonic we have been using and why mnemonics are so important. When using Ganache, the mnemonic was generated for us, and we then imported it to MetaMask. This was all we needed to do to start signing transactions, but in [Chapter 1](#) we discussed needing to sign transactions with a private key, so what's going on?

Back in 2012, Pieter Wuille created the hierarchical deterministic (HD) wallet in [BIP-32](#). This created a standard for generating a tree or hierarchy of private/public key pairs using a seed. Then in 2013, Satoshi Labs created a way of using a mnemonic phrase to generate the seed for the HD wallet in [BIP-39](#). Using these phrases makes it much easier to import or export accounts than managing individual private keys. These mnemonic phrases must be kept secret; the ones used in this book are known public phrases and are not secure. They should not be used to deploy on mainnet or test networks.

Log out of the Ganache account and log back in to the MetaMask account you created earlier in [Chapter 3](#). Since we are going to deploy on the Goerli testnet, we'll need some Goerli ether. Just in case you don't recall from [Chapter 3](#), Goerli is the testnet that is compatible with both Geth and Parity. To get our test ether, head over to the [Goerli Faucet](#) and you will see something similar to [Figure 5-5](#).

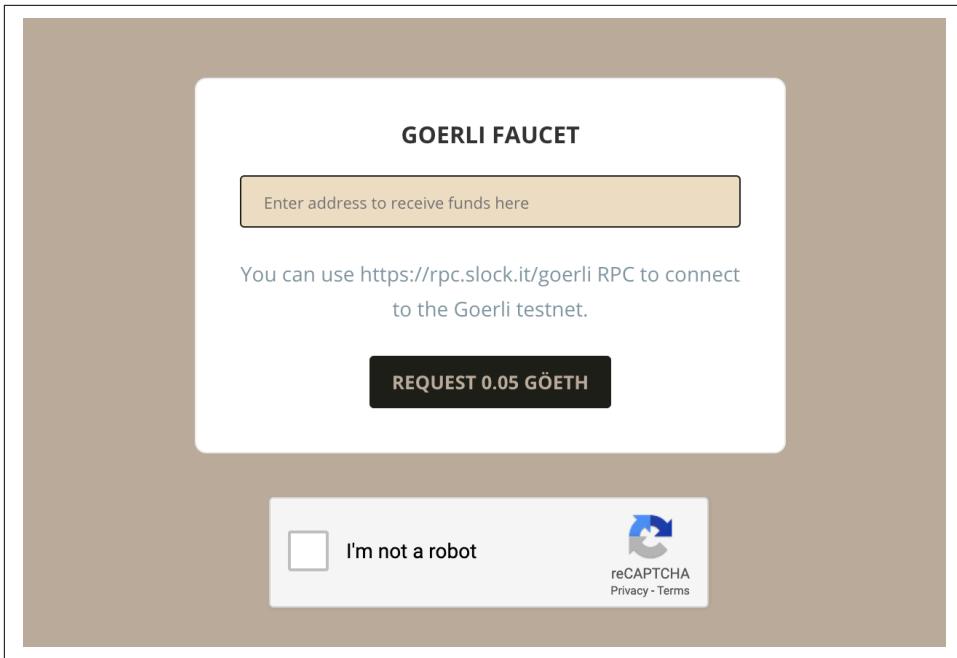


Figure 5-5. Goerli Faucet request

Click on the MetaMask plug-in, and then copy your account address to the clipboard and paste it into the address field. Make sure to fill out the Captcha to prove you are not a robot and click the REQUEST button. Once the transaction is complete, the deposit will be reflected in MetaMask (make sure you have selected the Goerli network). In order to have enough gas for the deployment, repeat this process three to four times.

Next we'll need to update the network configuration to connect to Goerli. This process will require the Truffle HD wallet library to create the provider that can be used during the deployment process. A provider is a component of Web3 that manages how to connect to the Ethereum network. In order to sign the transactions we are going to create in the deployment process, the provider will need access to either your mnemonic or your private key. We will provide this access by using an environment variable.

Let's go ahead and install the `HDWalletProvider` by using the following command:

```
$ npm install truffle-hdwallet-provider --save-dev
```

Now we'll set the mnemonic environment variable:

```
$ export MNEMONIC="YOUR MNEMONIC PHRASE GOES HERE"
```

With that installed, we'll now head back to our `truffle-config.js` file and add the following to the top of the file:

```
const HDWalletProvider = require("truffle-hdwallet-provider");
```

Then in the network section of our `config` object, we will add the Goerli configuration, as shown here:

```
goerli: {
  provider: () => {
    const mnemonic = process.env["MNEMONIC"]
    return new HDWalletProvider(mnemonic, "http://127.0.0.1:8545");
  },
  network_id: "*",
}
```

You can see this network configuration looks different than the development configuration. Instead of relying on the defaults, we set up a `provider` property, which is a function that will return the provider that should be used. The `provider` takes a mnemonic and the URL of the JSON RPC node we will be connecting. Parity, by default, will open the JSON RPC on port 8545. Notice that we are accessing the mnemonic through an environment variable; this is to prevent accidental commits of the mnemonic to the repo (if you are using version control).

With all that in place, open a new terminal or terminal tab and launch Parity on the Goerli network.

```
$ parity --chain=goerli
```

If it has been a while since your last sync, it may take a few moments to catch back up, but once it has resumed tracking the latest blocks, we can deploy to Goerli using the following command:

```
# from the greeter root directory
$ truffle migrate --network goerli
```

Just like when we deployed to Ganache, you'll see the output letting you know the transaction was sent and mined:

```
Starting migrations...
=====
> Network name:    'goerli'
> Network id:      5
> Block gas limit: 0x7a1200

1_initial_migration.js
=====

Deploying 'Migrations'
-----
> transaction hash: 0x6e57a078b7d9fdf6a1bc7c777951271220ae3b9f03a62c92...
```

```
> Blocks: 0          Seconds: 0
> contract address: 0x74c7382e11311530757beDb953D7F302cfAef417
> block number:    1149113
> block timestamp: 1566255037
> account:         0xcBe303aF8DF77602227A1a77dEF2983f915f2a12
> balance:         0.19477342
> gas used:        261329
> gas price:       20 gwei
> value sent:      0 ETH
> total cost:      0.00522658 ETH

> Saving migration to chain.
> Saving artifacts
-----
> Total cost:      0.00522658 ETH
```

2_deploy_greeter.js

```
Deploying 'Greeter'
-----
> transaction hash: 0x94cc8f0ca97b2f4de8f1d33c47796d2f95b211e6b0470c2...
> Blocks: 1          Seconds: 28
> contract address: 0x321374D4CD736Ce5484BFb3aF59C290639b73eA4
> block number:    1149116
> block timestamp: 1566255082
> account:         0xcBe303aF8DF77602227A1a77dEF2983f915f2a12
> balance:         0.18097294
> gas used:        648001
> gas price:       20 gwei
> value sent:      0 ETH
> total cost:      0.01296002 ETH

> Saving migration to chain.
> Saving artifacts
-----
> Total cost:      0.01296002 ETH
```

Summary

```
> Total deployments: 2
> Final cost:        0.0181866 ETH
```

Now with that in place, you can launch the web server:

```
$ cd client
$ npm run start
```

This will once again open a new browser tab connecting you to the application. Make sure to select the Goerli network in MetaMask, and you are ready to interact with your contract that has been deployed to the testnet!

We are going to walk through another way of deploying to a testnet—this time we will use the Rinkeby network, managing the deploy through Infura.

Deploying to Rinkeby with Infura

With both Ganache and Goerli deployments, we had to have some local software running, but now we are going to do a deployment where you do not have to download a personal blockchain or an Ethereum client. Instead, the node we will communicate with will be managed by Infura.

Infura is a service provider for managed Ethereum nodes. This saves you the time of syncing a local client, which (depending on your hardware and bandwidth) could take anywhere from hours to days to get fully synced. Infura allows deployments to the mainnet and to the Kovan, Rinkeby, Goerli, and Ropsten testnets. For this example, we will use Rinkeby to walk through another way of getting test ether from a different faucet implementation, but you can use any of the testnets.

To get started, head over to <https://infura.io/register> to create an account. Once you have verified your email, you will be taken to a page where you can pick your plan. For our purposes, the free plan will suffice.

When you get to your dashboard, you will have an empty project section and a button to create a new project like in [Figure 5-6](#).

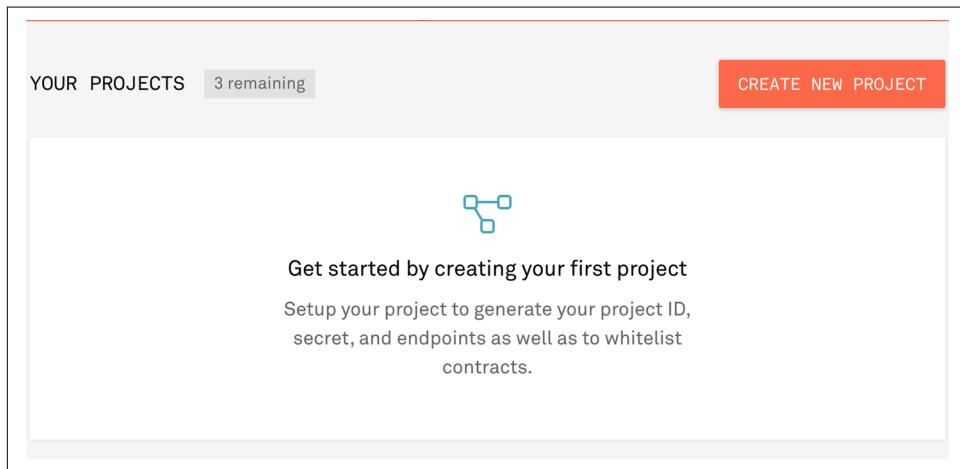


Figure 5-6. Creating a new project on Infura

Click on the CREATE NEW PROJECT button and give your project a name, like “greeter.” Once this project appears in your project list, click on the VIEW PROJECT button as shown in [Figure 5-7](#).

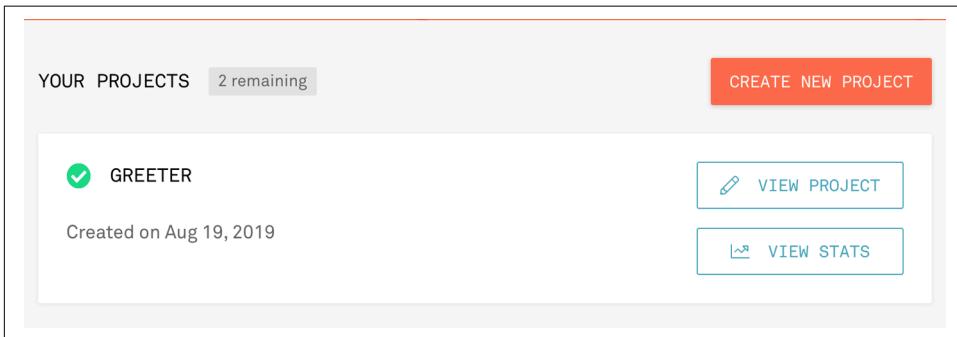


Figure 5-7. Infura project list

On the VIEW PROJECT page there is a section labeled Keys that includes an endpoint selection. It is defaulted to mainnet, but we want to use the Rinkeby network. Select that option and then copy the URL, which will look like the following:

```
rinkeby.infura.io/v3/<PROJECT_ID>
```

Just like we did with our mnemonic, let's go ahead and add our project ID to an environment variable:

```
$ export INFURA_PROJECT_ID=<PROJECT_ID>
```

Now we can update the network configuration in our *truffle-config.js* file to include a Rinkeby section:

```
rinkeby: {  
  provider: () => {  
    const mnemonic = process.env["MNEMONIC"]  
    const project_id = process.env["INFURA_PROJECT_ID"]  
    return new HDWalletProvider(  
      mnemonic,  
      `https://rinkeby.infura.io/v3/${project_id}`  
    );  
  },  
  network_id: "*"  
}
```

With the configuration in place, we now need to fund our account on the Rinkeby network by visiting the [Rinkeby Faucet](#). Here it asks you to create a social media post on either Twitter or Facebook and include the URL with the post. Once completed, you will have the funds needed for the deployment.

Now let's run our migrations:

```
# from greeter root directory  
$ truffle migrate --network rinkeby
```

When the scripts finish running, they will have deployed your contracts to the Rinkeby testnet. Launch the web server and then select the Rinkeby network in MetaMask:

```
$ cd client  
$ npm run start
```

When the browser launches, we'll have a clean contract welcoming us with "Hello, World!" You can now update this on the Rinkeby network.

Summary

In this chapter, we deployed contracts to a local blockchain for development using Ganache, and then deployed to two different public test networks using two different methods, one using a local Ethereum client (Parity) and the other using a hosted solution (Infura).

We will now turn our attention to building a larger application, learning the language features of Solidity and utilities of Truffle as we go along.

CHAPTER 6

The Fundraiser Application

Now that we have successfully built and deployed our “Hello, World!” application, let’s continue our journey into building smart contracts by creating an application to allow beneficiaries to receive donations in ether. This application was chosen as a way for us to explore working with ether in our contracts and how we can use data structures to track balances over time, similar to what you would find in token contracts.

By building this application, we will take a closer look at Solidity, including learning more about constructors, getter and setter methods, structs and events, and a few other helpful tips along the way.

Before we start writing code, let’s take a moment to understand the application that we are about to build.

Application Overview

Our application is going to help a beneficiary, such as a nonprofit, or any other entity that may be looking for funding. The beneficiary will have an address set up on a service like [Coinbase](#) so that they can easily sell the ether and convert it to USD. With the Ethereum address being managed by Coinbase, the beneficiary will not have access to the private key and therefore cannot interact with a DApp from that address. This is where a custodian address can help.

The custodian address is another Ethereum address that either the beneficiary controls directly or someone operating on behalf of the beneficiary will own. This address can interact with a DApp, and it is this address that will be given the ability to issue the transaction that will transfer funds collected by the Fundraiser contract to the beneficiary address.

Our home page depicted in [Figure 6-1](#) will have the call to action to create a new fundraiser in the top section and then list the previously created fundraisers.

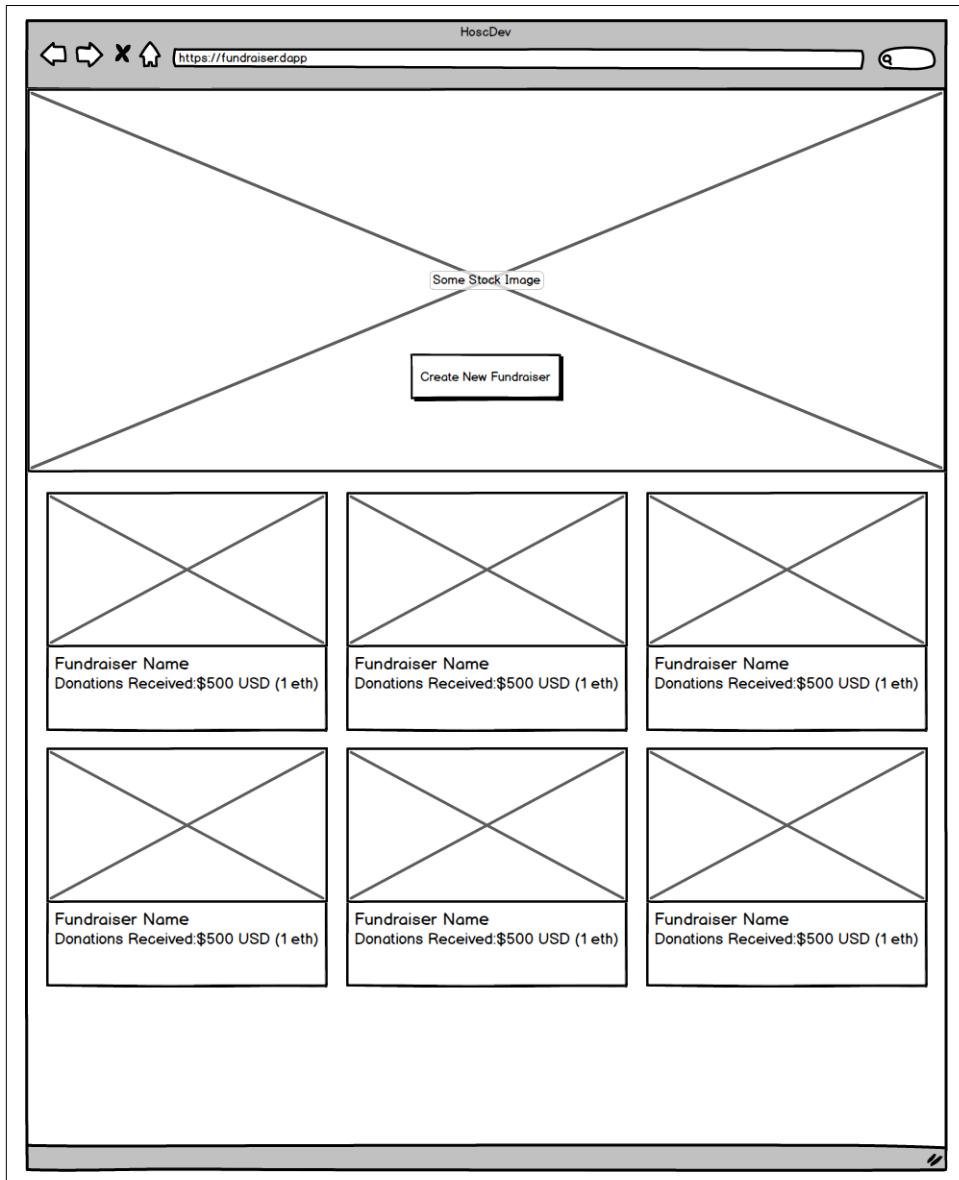


Figure 6-1. Fundraiser application home page

By clicking on the Create New Fundraiser button, you navigate to the new page as shown in [Figure 6-2](#).

The screenshot shows a web browser window with the title "HoscDev" at the top. The address bar displays the URL <https://fundraiserdapp/new>. The main content area is titled "Create a new Fundraiser". Below this, a section titled "Beneficiary Information" contains the following fields:

- Name: An input field.
- Website: An input field containing "https://".
- Ethereum Address: An input field.
- Confirm Ethereum Address: An input field.
- Image URL: An input field.
- Description: An input field.

At the bottom of the form is a "Create Fundraiser" button and a "cancel" link.

Figure 6-2. Creating a new fundraiser

This form is where we collect the details we can share about the beneficiary and the address to which we will eventually transfer the funds. This would be the exchange address we mentioned previously. Once submitted, this will send a transaction to the blockchain to create the fundraiser, and set the sending address as the owner of the fundraiser. When the transaction completes, we will be taken to the show page illustrated in [Figure 6-3](#). This is also the page users would navigate to from the home page when clicking on an existing fundraiser.

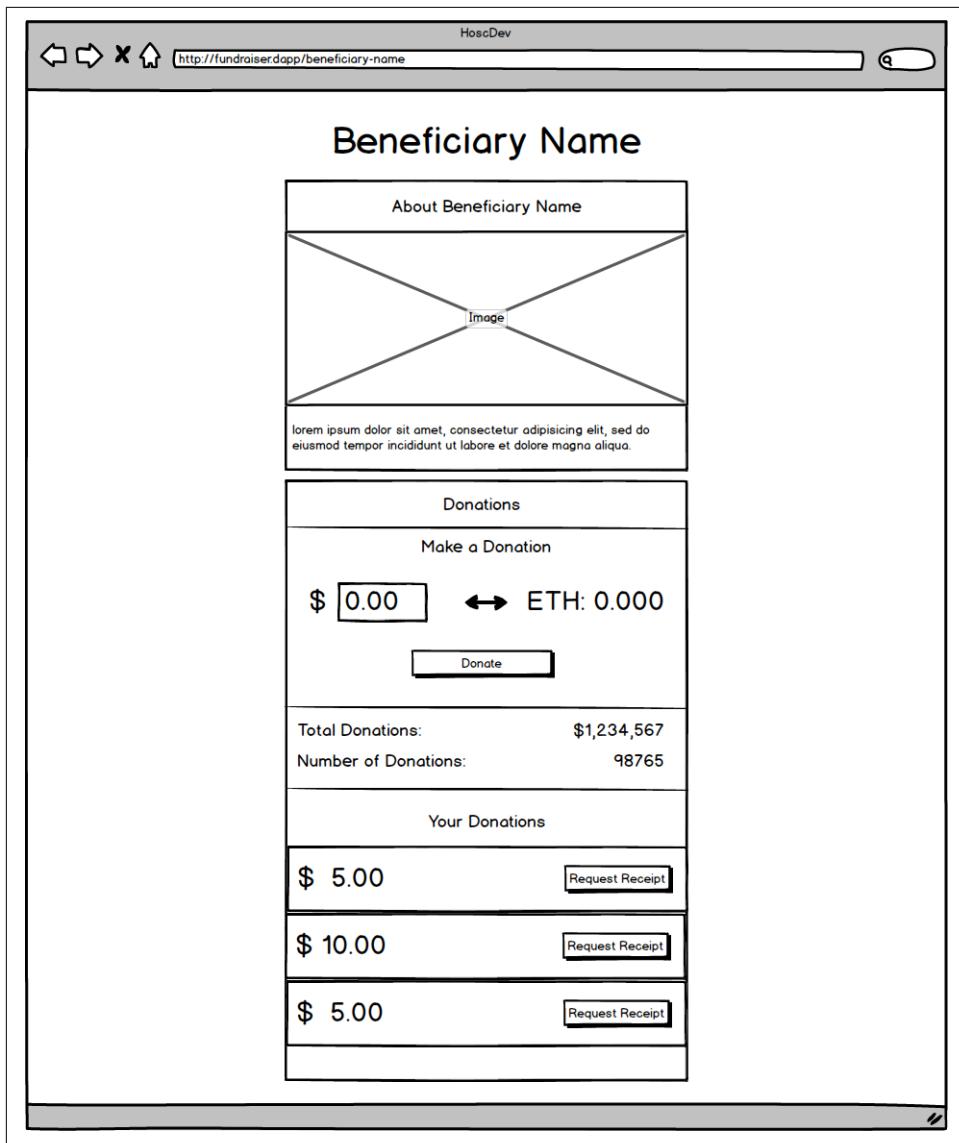


Figure 6-3. Fundraiser details page

The details page is where you can make a donation. It shows all the info about the beneficiary, and also shows summary data about how many donations have been made and the total dollar value of those donations. Since the value of ether is highly volatile, we will use a third-party API to grab the exchange rates when displaying this information.

There is also a section for a user to see their previous donations. By clicking on the Request Receipt button, users will see a receipt view of their donation that they can print and keep for their records. The receipt view is illustrated in [Figure 6-4](#).

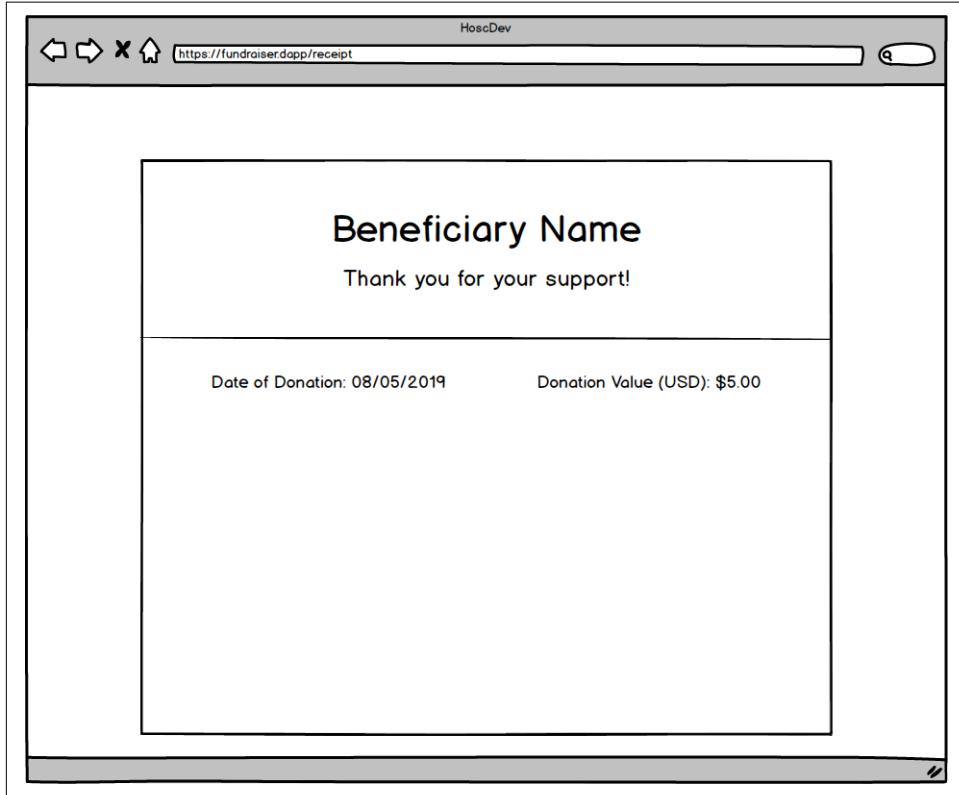


Figure 6-4. Donation receipt

Now that we know the features and have a general idea of the interface we will need to support, we can start breaking this down into something we can code. So let's get started.

Creating the Project

We are going to create another project for our fundraiser application. Back in our terminal, navigate to a directory in which you would like to contain this application and then make the fundraiser directory:

```
$ cd Projects # example directory containing our app  
$ mkdir fundraiser
```

Now we will create the Truffle directory structure. This time, instead of using `truffle init`, we are going to use one of the official Truffle Boxes. Truffle Boxes are ways to start new projects that have already been preconfigured for integration with other services or libraries. Since our UI will be written using React, we'll start with the React box. If React is not your preferred UI library, you can find other preconfigured boxes for things like Vue or even Drizzle, Truffle's own UI toolset on the [Truffle Boxes page](#).

```
$ truffle unbox react
```

Excluding the `node_modules` directory, our directory structure should now look like the following:

```
fundraiser
├── LICENSE
├── client
│   ├── README.md
│   ├── package-lock.json
│   ├── package.json
│   ├── public
│   │   ├── favicon.ico
│   │   ├── index.html
│   │   └── manifest.json
│   ├── src
│   │   ├── App.css
│   │   ├── App.js
│   │   ├── App.test.js
│   │   ├── index.css
│   │   ├── index.js
│   │   ├── logo.svg
│   │   ├── serviceWorker.js
│   │   └── utils
│   │       └── getWeb3.js
│   └── yarn.lock
└── contracts
    ├── Migrations.sol
    └── SimpleStorage.sol
└── migrations
    ├── 1_initial_migration.js
    └── 2_deploy_contracts.js
└── test
    ├── TestSimpleStorage.sol
    └── simplestorage.js
└── truffle-config.js
```

This time, instead of having only the `contracts`, `migrations`, and `test` directories, we have a `client` directory that includes the setup for our React application.

This box includes a sample contract, `SimpleStorage.sol`, along with the migration and tests for the contract. Let's go ahead and remove them, and then we will create our files for the fundraiser project:

```
$ rm contracts/SimpleStorage.sol \
migrations/2_deploy_contracts.js \
test/*
$ touch contracts/Fundraiser.sol test/fundraiser_test.js
```

Notice that we skipped the migration. We did this because we are going to write another contract, the `FundraiserFactory` contract, which will initialize individual fundraisers. We will build the factory contract in the next chapter, and will discuss the migrations in more detail at that time.

With our test file and contract files in place, let's get started initializing our fundraiser!

Initializing Fundraisers

Fundraisers are at the core of our application. We will start by modeling the data attributes of the fundraiser and then move on to the behaviors. Based on the form in [Figure 6-2](#), when we create a new fundraiser, we'll need the following:

- The name of the beneficiary
- The website where users can go to learn more
- The beneficiary's address (this is the address that can receive the funds)
- An image URL used for the card on the index page
- A brief description about the beneficiary
- The custodian or *owner* address

These are all state variables that should be set when initializing the contract. In order to test that everything is set up properly, we will write some test code that will create a new fundraiser instance, and then test each of the properties listed previously to make sure we get the values we intended to set during initialization.

In our test file, let's start with the following setup code illustrated in [Example 6-1](#).

Example 6-1. Fundraiser test setup

```
const FundraiserContract = artifacts.require("Fundraiser");

contract("Fundraiser", accounts => {
  let fundraiser;
  const name = "Beneficiary Name"

  describe("initialization", () => {
    beforeEach (async () => {
      fundraiser = await FundraiserContract.new(name);
```

```
    });
  });
});
```

We start our test file by requiring the `FundraiserContract` from the `artifacts` object. Then, using the `contract` function, we enter a clean room (fresh deploy of the test network) and set up some variables to hold the fundraiser and the name of the beneficiary. All set there, we enter a `describe` block with a `beforeEach` to set the fundraiser before each test example is run. This is the first time we have used `beforeEach`, which is a great tool for extracting the repetitive setup that would be required in each of the tests,¹ such as the creation of the subject under test.

With the setup done, we can focus on what it is that we want to assert. Since we do not have access to the internals of our contract, we will need to check the data was set up properly using getter functions. Let's add a test just after the closing of the `beforeEach` block, as demonstrated here:

```
it("gets the beneficiary name", async () => {
  const actual = await fundraiser.name();
  assert.equal(actual, name, "names should match");
});
```

Here our test is asserting that the name of our fundraiser should match the name we passed in to the constructor. Let's run our test and see what happens:

```
$ truffle test

Using network 'test'.

Compiling your contracts...
=====
Error: Could not find /Users/ksolo/Projects/fundraiser/contracts/Fundraiser.sol
from any sources
```

We've seen this error before. This means that we do not yet have a `Fundraiser` contract. We created the file, but we have not defined anything in it. Let's see if we can get past this error by creating a shell contract:

```
pragma solidity >0.4.23 <0.7.0;

contract Fundraiser {
```

Running our tests again, we get the following:

¹ You can find more information on `beforeEach` and other Mocha hooks at <https://mochajs.org/#hooks>.

```
$ truffle test

Using network 'test'.

Compiling your contracts...
=====
> Compiling ./contracts/Fundraiser.sol

Contract: Fundraiser
  initialization
    1) gets the beneficiary name
      > No events were emitted

0 passing (115ms)
1 failing

1) Contract: Fundraiser
  initialization
    gets the beneficiary name:
      TypeError: fundraiser.name is not a function
        at Context.it (test/fundraiser_test.js:29:39)
        at process._tickCallback (internal/process/next_tick.js:68:7)
```

Now our test is letting us know we do not have a `name` function on our contract. The `name` function is intended to return a stored value containing the name of our contract. It seems that a string would make sense here. Let's add a string state variable to our contract and then have the getter function return it. In our Greeter, we made the state variable private and created a public getter function to read that data. For our Fundraiser contract, we will take a different approach and instead make the state variable public. By making the state variable public, we will automatically have a public getter function with the same name as our storage variable created for us without having to write any additional code:

```
pragma solidity >0.4.23 <0.7.0;

contract Fundraiser {

  string public name;
```

With that update complete, let's see what happens when we run our test:

```
$ truffle test
Using network 'test'.

Compiling your contracts...
=====
> Compiling ./contracts/Fundraiser.sol
```

```

Contract: Fundraiser
  initialization
    1) gets the beneficiary name
      > No events were emitted

0 passing (143ms)
1 failing

1) Contract: Fundraiser
   initialization
     gets the beneficiary name:

   names should match
   + expected - actual

   +Beneficiary Name

   at Context.it (test/fundraiser_test.js:30:14)
   at process._tickCallback (internal/process/next_tick.js:68:7)

```

Finally, we have a failure that is pushing us towards building out our constructor. Since this value was intended to be set upon initialization, we need to write a constructor that can take in the arguments and start setting up the state of the contract.

Let's add the following constructor to the Fundraiser contract:

```

constructor(string memory _name) public {
  name = _name;
}

```

In order to prevent an ambiguous reference between our storage variable `name` and the parameter being passed in, we have prefixed the parameter with an underscore. This is a fairly common practice when you have functions setting public storage variables.

With the constructor in place, let's rerun our tests:

```

$ truffle test

Using network 'test'.

Compiling your contracts...
=====
> Compiling ./contracts/Fundraiser.sol

Contract: Fundraiser
  initialization
    ✓ gets the beneficiary name (38ms)

1 passing (143ms)

```

Great! We have our first test passing. Now we need to write a similar test for the rest of the state variables we want to set up. The website, image URL, and description state variables will all be represented by strings. Let's go ahead and add the tests for those. Then we'll discuss the other variables since they are going to be a little different.

Let's update the test file to look like [Example 6-2](#).

Example 6-2. Fundraiser string fields test

```
const FundraiserContract = artifacts.require("Fundraiser");

contract("Fundraiser", accounts => {
  let fundraiser;
  const name = "Beneficiary Name";
  const url = "beneficiaryname.org";
  const imageURL = "https://placekitten.com/600/350";
  const description = "Beneficiary description";

  describe("initialization", () => {
    beforeEach(async () => {
      fundraiser = await FundraiserContract.new(
        name,
        url,
        imageURL,
        description
      )
    });
  });

  it("gets the beneficiary name", async () => {
    const actual = await fundraiser.name();
    assert.equal(actual, name, "names should match");
  });

  it("gets the beneficiary url", async () => {
    const actual = await fundraiser.url();
    assert.equal(actual, url, "url should match");
  });

  it("gets the beneficiary image url", async () => {
    const actual = await fundraiser.imageURL();
    assert.equal(actual, imageURL, "imageURL should match");
  });

  it("gets the beneficiary description", async () => {
    const actual = await fundraiser.bio();
    assert.equal(actual, description, "description should match");
  });
});
```

By running the tests at this point, we will get the following failure:

```

$ truffle test
Using network 'test'.

Compiling your contracts...
=====
> Compiling ./contracts/Fundraiser.sol

Contract: Fundraiser
  initialization
    1) "before each" hook for "gets the beneficiary name"

  0 passing (30ms)
  1 failing

  1) Contract: Fundraiser
    initialization
      "before each" hook for "gets the beneficiary name":
      Error: Invalid number of parameters for "undefined". Got 4 expected 1!

```

This is letting us know our constructor is only set up to handle one argument. Let's add in the rest of the string parameters to our constructor function to match [Example 6-3](#).

Example 6-3. Adding string parameters to constructor

```

constructor(
  string memory _name,
  string memory _url,
  string memory _imageURL,
  string memory _description
)
public
{
  name = _name;
}

```

Running our tests now will give us the following output:

```

$ truffle test

Using network 'test'.

Compiling your contracts...
=====
> Compiling ./contracts/Fundraiser.sol
  > compilation warnings encountered:

/Users/ksolo/Projects/fundraiser/contracts/Fundraiser.sol:10:9:
Warning: Unused function parameter. Remove or comment out the variable
name to silence this warning.
      string memory _url,
      ^-----^

```

```

,/Users/ksolo/Projects/fundraiser/contracts/Fundraiser.sol:11:9:
Warning: Unused function parameter. Remove or comment out the variable
name to silence this warning.
    string memory _imageURL,
    ^-----^
,/Users/ksolo/Projects/fundraiser/contracts/Fundraiser.sol:12:9:
Warning: Unused function parameter. Remove or comment out the variable
name to silence this warning.
    string memory _description
    ^-----^

Contract: Fundraiser
  initialization
    ✓ gets the beneficiary name (40ms)
      1) gets the beneficiary url
        > No events were emitted
        2) gets the beneficiary image url
        > No events were emitted
        3) gets the beneficiary description
        > No events were emitted

1 passing (465ms)
3 failing

1) Contract: Fundraiser
  initialization
    gets the beneficiary url:
TypeError: fundraiser.url is not a function
  at Context.it (test/fundraiser_test.js:26:39)
  at process._tickCallback (internal/process/next_tick.js:68:7)

2) Contract: Fundraiser
  initialization
    gets the beneficiary image url:
TypeError: fundraiser.imageURL is not a function
  at Context.it (test/fundraiser_test.js:31:39)
  at process._tickCallback (internal/process/next_tick.js:68:7)

3) Contract: Fundraiser
  initialization
    gets the beneficiary description:
TypeError: fundraiser.description is not a function
  at Context.it (test/fundraiser_test.js:36:39)
  at process._tickCallback (internal/process/next_tick.js:68:7)

```

This output shows compiler warnings about unused variables and then the failing tests. Each of the tests are failing for a similar reason: the getter function for the corresponding value doesn't exist.

To fix the warnings, we will need to add the state variables to the contract and assign them in the constructor just like we did with `name`. These changes are illustrated in [Example 6-4](#).

Example 6-4. Fundraiser state variables and constructor

```
pragma solidity >0.4.23 <0.7.0;

contract Fundraiser {

    string public name;
    string public url;
    string public imageURL;
    string public description;

    constructor(
        string memory _name,
        string memory _url,
        string memory _imageURL,
        string memory _description
    )
    public
    {
        name = _name;
        url = _url;
        imageURL = _imageURL;
        description = _description;
    }
}
```

With those pieces in place, we should have a passing test suite with output like the following:

```
$ truffle test

Using network 'test'.

Compiling your contracts...
=====
> Compiling ./contracts/Fundraiser.sol

Contract: Fundraiser
initialization
  ✓ gets the beneficiary name
  ✓ gets the beneficiary url
  ✓ gets the beneficiary image url
  ✓ gets the beneficiary description

4 passing (572ms)
```

Before we move on, let's talk a little bit more on data locality and how it relates to function parameters. In our constructor function, we labeled all the string parameters as being from the `memory` location, but what does that mean?

Parameters containing reference data types such as arrays or strings need to note if they are pointing directly to persisted data, notated with `storage`, or a local copy of that data, notated as `memory`. In most cases, you will want `memory` as the data location. In fact, `memory` was the default location and was not required in the parameter lists before 0.5.0.

The final data location `calldata` was discussed last chapter with the Greeter contract and is the required location when the function visibility is set to `external`.

With our string fields out of the way, we can move on to discussing how to handle the beneficiary and custodian.

Setting the Beneficiary and Custodian

Both the beneficiary and custodian will be represented by addresses. The custodian address is the address associated with the manager of the fundraiser. This address will not be receiving any of the funds but will be the address allowed to make changes and request the funds transferred to the beneficiary. Remember, we are doing this extra step with the assumption that the beneficiary address is associated with something like Coinbase or another currency exchange, and not an address that can be used to send transactions.

Solidity offers two different types for address: the `address` type and the `address payable` type. The distinction is that payable addresses can receive ether, and will have the `transfer` and `send` methods available. With this in mind, only the beneficiary address needs to be payable.

Let's go ahead and add our tests for the beneficiary and custodian fields. Back in our test file, let's update the setup to set the beneficiary address and the custodian address, update the call to create a new fundraiser with these parameters, and then add the tests for the new fields through getter functions. The updated test file is shown in [Example 6-5](#).

Example 6-5. Fundraiser tests with beneficiary and custodian

```
const FundraiserContract = artifacts.require("Fundraiser");

contract("Fundraiser", accounts => {
  let fundraiser;
  const name = "Beneficiary Name";
  const url = "beneficiaryname.org";
  const imageURL = "https://placekitten.com/600/350";
  const description = "Beneficiary description";
  const beneficiary = accounts[1];
  const custodian = accounts[0];

  describe("initialization", () => {
```

```

beforeEach(async () => {
  fundraiser = await FundraiserContract.new(
    name,
    url,
    imageURL,
    description,
    beneficiary,
    custodian
  )
});

it("gets the beneficiary name", async () => {
  const actual = await fundraiser.name();
  assert.equal(actual, name, "names should match");
});

it("gets the beneficiary url", async () => {
  const actual = await fundraiser.url();
  assert.equal(actual, url, "url should match");
});

it("gets the beneficiary image url", async () => {
  const actual = await fundraiser.imageURL();
  assert.equal(actual, imageURL, "imageURL should match");
});

it("gets the beneficiary description", async () => {
  const actual = await fundraiser.description();
  assert.equal(actual, description, "description should match");
});

it("gets the beneficiary", async () => {
  const actual = await fundraiser.beneficiary();
  assert.equal(actual, beneficiary, "beneficiary addresses should match");
});

it("gets the custodian", async () => {
  const actual = await fundraiser.custodian();
  assert.equal(actual, custodian, "bios should match");
});
});

```

Running our tests, we will see a similar error from before. Our constructor on the Solidity side was not updated to take the new parameters. Let's update the constructor to match [Example 6-6](#).

Example 6-6. Updating constructor with beneficiary and custodian addresses

```
constructor(
    string memory _name,
    string memory _url,
    string memory _imageURL,
    string memory _description,
    address payable _beneficiary,
    address _custodian
)
public
{
    name = _name;
    url = _url;
    imageURL = _imageURL;
    description = _description;
}
```

After we run our tests, our compiler should once again give us unused parameter warnings since we are not yet doing anything with the beneficiary or custodian addresses. We should also see two more failing tests indicating that there are some missing getter functions.

In order to fix the warnings, we'll need to add the state variables to the contract, and then update the constructor to match [Example 6-7](#).

Example 6-7. Setting beneficiary and custodian

```
pragma solidity >0.4.23 <0.7.0;

contract Fundraiser {

    string public name;
    string public url;
    string public imageURL;
    string public description;

    address payable public beneficiary;
    address public custodian;

    constructor(
        string memory _name,
        string memory _url,
        string memory _imageURL,
        string memory _description,
        address payable _beneficiary,
        address _custodian
    )
}
```

```

    public
{
    name = _name;
    url = _url;
    imageURL = _imageURL;
    description = _description;
    beneficiary = _beneficiary;
    custodian = _custodian;
}

```

Now with this in place, when we run our tests, we should see all of them pass:

```

$ truffle test

Using network 'test'.

Compiling your contracts...
=====
> Compiling ./contracts/Fundraiser.sol

Contract: Fundraiser
  initialization
    ✓ gets the beneficiary name (49ms)
    ✓ gets the beneficiary url
    ✓ gets the beneficiary image url
    ✓ gets the beneficiary description
    ✓ gets the beneficiary
    ✓ gets the custodian

6 passing (873ms)

```

Great! We are all set with how to handle creating a new fundraiser. Next, we'll look into creating a way for our custodian to update the beneficiary address.

Editing the Beneficiary

There may be a time when the custodian will need to update the beneficiary address. This could happen if the custodian accidentally entered the wrong address or if the beneficiary decides to switch the exchange it is using. Much like the Greeter contract from last chapter, we will want to restrict who can perform this operation, but unlike [Chapter 4](#), we have a custodian instead of an owner. To avoid reimplementing known behavior from an audited contract, let's make a few changes to the Fundraiser contract to support `Ownable`.

We'll start by removing the custodian state variable and then updating the constructor to use the internal function `_transferOwnership` defined in the `Ownable` contract. The resulting changes are illustrated in [Example 6-8](#).

Example 6-8. Updating the constructor

```
address payable public beneficiary;

constructor(
    string memory _name,
    string memory _url,
    string memory _imageURL,
    string memory _description,
    address payable _beneficiary,
    address _custodian
)
public
{
    name = _name;
    url = _url;
    imageURL = _imageURL;
    description = _description;
    beneficiary = _beneficiary;
    _transferOwnership(_custodian);
}
```

Back in our test file, we need to update all the references of `custodian` to `owner`. [Example 6-9](#) includes the updated test file.

Example 6-9. Update custodian to owner

```
const FundraiserContract = artifacts.require("Fundraiser");

contract("Fundraiser", accounts => {
    let fundraiser;
    ...omitted...
    const owner = accounts[0];

    describe("initialization", () => {
        beforeEach(async () => {
            fundraiser = await FundraiserContract.new(
                name,
                url,
                imageURL,
                description,
                beneficiary,
                owner
            );
        ...
        ...omitted...

        it("gets the owner", async () => {
            const actual = await fundraiser.owner();
```

```
        assert.equal(actual, owner, "bios should match");
    });
});
});;
```

With this change in place, let's run the tests:

```
$ truffle test

Compiling your contracts...
...omitted...
Contract: Fundraiser
  initialization
  ...omitted...
5 passing (864ms)
1 failing

1) Contract: Fundraiser
   initialization
   gets the owner:
TypeError: fundraiser.owner is not a function
  at Context.it (test/fundraiser_test.js:51:39)
  at process._tickCallback (internal/process/next_tick.js:68:7)
```

Luckily the `Ownable` contract implements this function, and we will pick it up once we begin inheriting from the contract.

Install OpenZeppelin with the following command:

```
$ npm install openzeppelin-solidity --save
```

Then in our `Fundraiser` contract, we'll import and inherit from the `Ownable` contract:

```
pragma solidity >0.4.23 <0.7.0;

import "openzeppelin-solidity/contracts/ownership/Ownable.sol";

contract Fundraiser is Ownable {
  ...omitted...
}
```

With these changes in place, we are back to passing tests:

```
$ truffle test
...omitted...
Contract: Fundraiser
  initialization
  ...omitted...
  ✓ gets the owner

6 passing (892ms)
```

Now let's plan out our new function. We want a `setBeneficiary` function that will fail if it is called by anyone other than the owner. If the sender is the owner, it will

then update the `beneficiary` address with the one passed in as a parameter. Any time we are dealing with access controls, we should plan to test the function in the various roles to verify we have appropriately restricted the behavior.

Let's create a new `describe` block for the `setBeneficiary` function, like the one in [Example 6-10](#).

Example 6-10. Describe block for setBeneficiary

```
const FundraiserContract = artifacts.require("Fundraiser");

contract("Fundraiser", accounts => {
  ...omitted...

  describe("initialization", () => {
    beforeEach(async () => {
      fundraiser = await FundraiserContract.new(
        name,
        url,
        imageURL,
        description,
        beneficiary,
        owner
      )
    });
    ...omitted...
  });

  describe("setBeneficiary", () => {
  });
});;
```

We will need to set up a fundraiser just like we do in the initialization block of tests. Instead of duplicating this code, we can move the `beforeEach` into the parent `contract` scope to look like [Example 6-11](#).

Example 6-11. Describe block for setBeneficiary

```
const FundraiserContract = artifacts.require("Fundraiser");

contract("Fundraiser", accounts => {
  ...omitted...

  beforeEach(async () => {
    fundraiser = await FundraiserContract.new(
      name,
      url,
      imageURL,
      description,
```

```

        beneficiary,
        owner
    )
});

describe("initialization", () => {
    ...omitted...
});

describe("setBeneficiary", () => {
});
});

```

With our setup completed, we can add the tests in [Example 6-12](#).

Example 6-12. setBeneficiary test cases

```

describe("setBeneficiary", () => {
    const newBeneficiary = accounts[2];

    it("updated beneficiary when called by owner account", async () => {
        await fundraiser.setBeneficiary(newBeneficiary, {from: owner});
        const actualBeneficiary = await fundraiser.beneficiary();
        assert.equal(actualBeneficiary, newBeneficiary, "beneficiaries should match");
    });

    it("throws an error when called from a non-owner account", async () => {
        try {
            await fundraiser.setBeneficiary(newBeneficiary, {from: accounts[3]});
            assert.fail("withdraw was not restricted to owners")
        } catch(err) {
            const expectedError = "Ownable: caller is not the owner"
            const actualError = err.reason;
            assert.equal(actualError, expectedError, "should not be permitted")
        }
    })
});

$ truffle test

Compiling your contracts...
...omitted...

Contract: Fundraiser
  initialization
    ...omitted...
  setBeneficiary
    1) updated beneficiary when called by owner account
    ...omitted...
    2) throws an error when called from a non-owner account
    ...omitted...

```

```
6 passing (1s)
2 failing

1) Contract: Fundraiser
    setBeneficiary
        updated beneficiary when called by owner account:
        TypeError: fundraiser.setBeneficiary is not a function
        at Context.it (test/fundraiser_test.js:59:24)

2) Contract: Fundraiser
    setBeneficiary
        throws an error when called from a non-owner account:
        AssertionERROR: should not be permitted: expected undefined to
        equal 'Ownable: caller is not the owner'
        at Context.it (test/fundraiser_test.js:71:16)
```

Our tests are failing because we have not yet defined the `setBeneficiary` function. Let's add that function to our contract:

```
function setBeneficiary(address payable _beneficiary) public onlyOwner {
    beneficiary = _beneficiary;
}
```

Let's run our tests again:

```
$ truffle test

Compiling your contracts...
...omitted...
Contract: Fundraiser
  initialization
  ...omitted...
  setBeneficiary
    ✓ updated beneficiary when called by owner account (72ms)
    ✓ throws an error when called from a non-owner account (60ms)

8 passing (2s)
```

Success! We can do this for all the fields, but it is critical that we update the beneficiary since this account will receive all the funds. We will discuss this more when we work on the withdraw functionality later in the chapter. For now, let's move on to making donations.

Making Donations

Donations are the focal point of this application. This is the main area of interaction for the majority of users, and will be the area we need to get right so that users can come back and donate to different organizations. The UI for donations is depicted in [Figure 6-5](#).

Figure 6-5. Making donations

To make a donation, a donor would input the dollar amount they would like to donate in USD, which will then update the amount shown in ether (ETH). In the UI chapters, we will walk through connecting to a third-party API to get current exchange rates for this conversion. After the user clicks the button to donate, we want to observe the following changes:

- A donation record is created and associated with the donor's address.
- The donations count is incremented.
- The total donations amount is increased by the amount of the donation.
- An event is emitted to add the donation to the logs (more on this later).

Let's take a moment to discuss the donation record a bit more. A donation will belong to an address. The record itself will contain how much ether was donated and the

date of the donation. With this information, we can look up historic exchange rates for the time of the donation to determine the value and provide a receipt for off-chain uses.

Both of the fields (date and value) can be represented by unsigned integers. The date will be represented by the number of seconds since Unix epoch, and the value will be represented in units of wei, the smallest denomination of ether. Integers in Solidity come in signed or unsigned types, and can be sized from 8 to 256 bits. If you do not specify the size of the `int` or `uint`, it will default to 256. However, it is recommended that you explicitly size your integers when possible. Since these fields are the same type, we could create an array to hold them, as shown in the following snippet:

```
uint256[2] memory donation;
donation[0] = 500;
donation[2] = block.timestamp;
```

Here we create a new in-memory array of unsigned integers that will contain two items, and assign each item a value. When using the `donation` variable, it would be up to the developer to remember which index contained which item. Instead of using an array, we can use a `struct` and give the data a name.

Structs

Structs in Solidity allow us to define a new type. The type will have properties that can be made up of different types. To create a `Donation` struct, we can use the following snippet:

```
struct Donation {
    uint256 value;
    uint256 date;
}
```

There are two ways we can initialize a struct. We can initialize a struct using positional arguments. The order in which the properties appear in the struct is the order of the parameters. Here the value would be the first parameter followed by the date, as illustrated in the following snippet:

```
Donation memory donation = Donation(
    100,
    block.timestamp
);
```

Alternatively, we can use the keyword or dictionary for creating our struct, as shown here:

```
Donation memory donation = Donation({
    value: 100,
    date: block.timestamp
});
```

Notice in the case of the keyword, you need to add curly braces {} to encapsulate the mapping of key to the value. With this option, the order is no longer a hard dependency since the inputs can be labeled. Not having to remember the ordering is a big win—therefore, we will use keyword syntax for the rest of the samples in the book.

Since this `Donation` struct is an internal structure, we cannot test for it directly. In order to see the number of donations a specific donor has made, we'll need to track the donations by donor. To make this association, we can use another Solidity type: mappings.

Mappings

Mappings allow you to associate a key to a value. In Solidity, the key types are restricted to integers, booleans, addresses, bytes, and strings. The value types can be anything, including another mapping. Mappings in Solidity are sadly not enumerable, meaning that you cannot use a `for` loop to iterate through the key/value pairs as you would an array. For our purposes, this restriction will not be an issue. However, if you need to iterate for a mapping, you would need to create the behavior yourself or use a third-party library to extend the default mapping behavior.

To declare a mapping, we'll need to identify the type of key and the type of value, like the following:

```
mapping(address => Donation[]) private _donations;
```

This mapping associates an address to a dynamic array of donations. If we wanted to access an individual donor's donations we would access them using [] notation:

```
address donor = msg.sender;
_donations[donor];
```

With these new data structures at our disposal, we have all the pieces needed to write the first part of our donation tests.

Donation Tests

In our test file, let's create another `describe` block to contain the tests for making a donation and initialize a few variables we'll use to make donations. This setup is illustrated in [Example 6-13](#).

Example 6-13. Setup for making donations

```
const FundraiserContract = artifacts.require("Fundraiser");

contract("Fundraiser", accounts => {
  ...omitted...

  describe("initialization", () => {
```

```

    ...omitted...
});

describe("making donations", () => {
  const value = web3.utils.toWei('0.0289');
  const donor = accounts[2];

  it("increases myDonationsCount");
  it("includes donation in myDonations");
});
});

```

In our test group, we created variables to hold the value of the donation and a donor address. The value of our donation is represented in units of wei, which is the smallest denomination of ether. Instead of typing a lot of zeros, we use the web3 `toWei` helper function to make the conversion for us.

Our donor is represented with an address from our `accounts` variable that was passed in to our `contract` function. When we send the donation, we will make sure to send it from this account instead of the default account.

There are two pending tests that we now need to implement. We'll start with the one labeled "`increases myDonationsCount`". Update the test with the following:

```

it("increases myDonationsCount", async () => {
  const currentDonationsCount = await fundraiser.myDonationsCount(
    {from: donor}
  );

  await fundraiser.donate({from: donor, value});

  const newDonationsCount = await fundraiser.myDonationsCount(
    {from: donor}
  );

  assert.equal(
    1,
    newDonationsCount - currentDonationsCount,
    "myDonationsCount should increment by 1"
)
})

```

In our new test, we are going to get the current donations count for the specific donor, make a donation from that donor, and then recheck their donation count. We then assert that the difference between the counts is 1.

Running our tests, we will get the following failure:

```

$ truffle test
Using network 'test'.

Compiling your contracts...
=====

```

```

> Compiling ./contracts/Fundraiser.sol

Contract: Fundraiser
  initialization
    ...omitted..
  setBeneficiary
    ...omitted...
  making donations
    1) increases myDonationsCount
  > No events were emitted
    - includes donation in myDonations

8 passing (981ms)
1 pending
1 failing

1) Contract: Fundraiser
  making donations
    increases myDonationsCount:
      TypeError: fundraiser.myDonationsCount is not a function
        at Context.it (test/fundraiser_test.js:72:54)
        at process._tickCallback (internal/process/next_tick.js:68:7)

```

Our tests are failing because we do not have a `myDonationsCount` function. This function should return the count of donations for the donor making the request. In order to get this test to pass, we'll need to add the donor to donation mapping, and then get the size of the array of donations associated with the sender. Let's update our Fundraiser contract, as illustrated in [Example 6-14](#).

Example 6-14. Adding myDonationsCount

```
pragma solidity >0.4.23 <0.7.0;
```

```

contract Fundraiser {
  struct Donation {
    uint256 value;
    uint256 conversionFactor;
    uint256 date;
  }
  mapping(address => Donation[]) private _donations;

  string public name;
  string public url;
  string public imageURL;
  string public description;

  address payable public beneficiary;

  constructor(
    ...omitted...

```

```

    )
  public
{
  ...omitted...
}

function setBeneficiary(address payable _beneficiary) public onlyOwner {
  beneficiary = _beneficiary;
}

function myDonationsCount() public view returns(uint256) {
  return _donations[msg.sender].length;
}
}

```

Running our tests will still result in a failure since we haven't created the `donate` function yet. Add the following function to the Fundraiser contract:

```

function donate() public payable {
  Donation memory donation = Donation({
    value: msg.value,
    date: block.timestamp
  });
  _donations[msg.sender].push(donation);
}

```

There are a few new things here to discuss. This is the first function that we have implemented that accepts ether, and therefore we need to add the `payable` modifier to our function declaration. When we send ether to a contract or EOA, we will increase the balance of those addresses automatically, and we will not need to add any code for that behavior.

In the body of our function, we create a new `Donation` struct in memory, and then push that onto the `Donation` array associated with our donor through the `push` function. The `push` function is only available on arrays that are declared as storage arrays. We implicitly made that declaration early by adding the mapping of addresses to donations.

With our `donate` function in place, our tests should now pass:

```

$ truffle test
Using network 'test'.


Compiling your contracts...
=====
> Compiling ./contracts/Fundraiser.sol


Contract: Fundraiser
  initialization
  ...omitted...

```

```

setBeneficiary
...omited...
making donations
✓ increases myDonationsCount (106ms)
- includes donation in myDonations

9 passing (1s)
1 pending

```

Success! We'll now turn our attention to the `myDonations` function.

myDonations

The `myDonations` function should return back the donations made for the donor to the fundraiser. Unfortunately, due to limitations in the current ABI, we cannot return an array of structs from external or public functions. Instead, we will need to leverage multiple return values and arrays to pull this off.

So far, we have been returning either nothing at all or just one value from a function. Now we are going to return two arrays from a function, one array for each of the properties of our struct. When we declare our return values in Solidity, we can either declare only the types or we can declare the types with an identifier. Both of the following examples are valid:

```

function multipleReturns() pure public returns(uint256, uint256) {
    return (1, 2);
}

function multipleReturnsWithNames() pure public returns(uint256 a, uint256 b) {
    a = 1;
    b = 2;
    return (a, b);
}

```

When this function returns, it will return a tuple. Tuples are a type that can be composed of other types. But unlike other Solidity types, we cannot create tuples directly in our code; they are reserved for returns like this. When our test code receives multiple return values, they can be accessed by index like an array, or if the return values are named, they can be accessed by name.

In order for `myDonations` to return the data, we need to build up an array of values and dates. When we get the data back, our test will need to check values to make sure they are correct. Let's update the test with the label "includes donation in `myDonations`" with the following:

```

it("includes donation in myDonations", async () => {
    await fundraiser.donate({from: donor, value});
    const {values, dates} = await fundraiser.myDonations(
        {from: donor}
    );
}

```

```

    assert.equal(
      value,
      values[0],
      "values should match"
    );
    assert(dates[0], "date should be present");
  });

```

In our new test, we make a donation from our donor address and then call `myDonations` to get a list of donations. We will actually get back two arrays from the function and, using destructuring in JavaScript, assign them to their respective variables. Then we check the first item of the values array to make sure it is what we expect. For the date, we are just going to check that it exists.

Running our tests, we will get the failure indicated here:

```

$ truffle test
Using network 'test'.

Compiling your contracts...
=====
> Compiling ./contracts/Fundraiser.sol

Contract: Fundraiser
  initialization
    ...omited...
  setBeneficiary
    ...omitted...
  making donations
    ✓ increases myDonationsCount (107ms)
      1) includes donation in myDonations
    > No events were emitted

9 passing (1s)
1 failing

1) Contract: Fundraiser
   making donations
     includes donation in myDonations:
       TypeError: fundraiser.myDonations is not a function
         at Context.it (test/fundraiser_test.js:84:67)
         at process._tickCallback (internal/process/next_tick.js:68:7)

```

The failure tells us we need to define the function on the Fundraiser contract. Let's add the following function to our contract:

```

function myDonations() public view returns(
  uint256[] memory values,
  uint256[] memory dates
)
{

```

```

uint256 count = myDonationsCount();
values = new uint256[](count);
dates = new uint256[](count);

for (uint256 i = 0; i < count; i++) {
    Donation storage donation = _donations[msg.sender][i];
    values[i] = donation.value;
    dates[i] = donation.date;
}

return (values, dates);
}

```

Let's review this code. One of the first things to note is that we are returning the two arrays we discussed earlier, both of which are of the `uint256` type. Entering the function body, the first thing we do is get the count of donations for the message sender using the `myDonationsCount` function we created earlier. This lets us know how big our arrays need to be.

We then assign arrays of the appropriate type to each of the named return variables. Since the size will not be known until runtime, we have to use the `new` keyword to create these arrays in memory. Once they are created, we then iterate over the donations using a `for` loop.

A `for` loop in Solidity operates like a `for` loop in many other languages. We first initialize a new variable like `i`. We then follow it with an expression that will evaluate to true or false to know if we should continue iterating. Lastly, we update the variables to prevent an infinite loop.

Inside of the `for` loop, we create a donation variable. This time we use the `storage` keyword instead of `memory`. Here, we are now referencing a struct that has already been saved to state variables of the contract and not creating a new one in memory. Once we have our donation record, we access the `value` and `date` properties off of it and insert them into the appropriate array.

Once the arrays have been built up, we can return them. With the function implemented, our tests should now pass:

```

$ truffle test
Using network 'test'.

Compiling your contracts...
=====
> Compiling ./contracts/Fundraiser.sol

Contract: Fundraiser
  initialization
  ...omitted...
  setBeneficiary
  ...omitted...

```

```
making donations
✓ increases myDonationsCount (105ms)
✓ includes donation in myDonations (144ms)
```

Alright! With that piece done, we now need to update the fundraiser totals.

Fundraiser Totals

Each fundraiser tracks the total number of donations received along with the total donation amount. These values are what are illustrated in our UI excerpt in [Figure 6-6](#).

Total Donations:	\$1,234,567
Number of Donations:	98765

Figure 6-6. Fundraiser totals

Since our donations are stored under each donor address, and maps are not iterable, we will update these values every time a donation is made instead of trying to calculate them on the fly. To support this behavior, we'll need to add two more state variables to our contract and then the associated getter functions.

We'll start with the total donations test:

```
it("increases the totalDonations amount", async () => {
  const currentTotalDonations = await fundraiser.totalDonations();
  await fundraiser.donate({from: donor, value});
  const newTotalDonations = await fundraiser.totalDonations();

  const diff = newTotalDonations - currentTotalDonations;

  assert.equal(
    diff,
    value,
    "difference should match the donation value"
  );
});
```

When we make a donation, we should see this value go up by the amount donated. One value that would also increase by this amount is the balance of the contract, so why not use that value? The balance will periodically be withdrawn by the beneficiary, taking it back to 0. Tracking the total amount donated separately will show us how much the fundraiser has raised instead of how much is currently available for the beneficiary to withdraw.

This test and our next test will look pretty similar to the `myDonationsCount` test from before. We check the `totalDonations` before and after the donation and make sure it increases by the amount of the donation. Running this test, we should see the following failure:

```
$ truffle test
Using network 'test'.

Compiling your contracts...
=====
> Compiling ./contracts/Fundraiser.sol

Contract: Fundraiser
  initialization
    ...omitted...
  setBeneficiary
    ...omitted...
  making donations
    ✓ increases myDonationsCount (103ms)
    ✓ includes donation in myDonations (94ms)
      1) increases the totalDonations amount
    > No events were emitted
      - increments the donorsCount

10 passing (1s)
1 failing

1) Contract: Fundraiser
   making donations
     increases the totalDonations amount:
     TypeError: fundraiser.totalDonations is not a function
       at Context.it (test/fundraiser_test.js:92:54)
       at process._tickCallback (internal/process/next_tick.js:68:7)
```

In order for this test to pass, we need to add the state variable and update our donation method to update the value when a donation is made. Since anyone can make a donation and there is no limit to the amount of the donation, we should consider using `SafeMath` to [prevent integer overflow issues](#).



`SafeMath` is included in the OpenZeppelin contracts we have already installed, and its use is encouraged anytime you are dealing with arithmetic operations that could cause an integer variable to exceed its bounds.

Let's update our contract with the code from [Example 6-15](#).

Example 6-15. Updating total donations

```
pragma solidity >0.4.23 <0.7.0;

import "openzeppelin-solidity/contracts/ownership/Ownable.sol";
import "openzeppelin-solidity/contracts/math/SafeMath.sol";

contract Fundraiser is Ownable {
    using SafeMath for uint256;

    ...omitted..

    uint256 public totalDonations;

    constructor(
        ...omitted...
    )
    public
    {
        ...omitted...
    }

    function setBeneficiary(address payable _beneficiary) public onlyOwner {
        beneficiary = _beneficiary;
    }

    function myDonationsCount() public view returns(uint256) {
        return _donations[msg.sender].length;
    }

    function donate() public payable {
        Donation memory donation = Donation({
            value: msg.value,
            date: block.timestamp
        });
        _donations[msg.sender].push(donation);
        totalDonations = totalDonations.add(msg.value);
    }

    function myDonations() public view returns(
        uint256[] memory values,
        uint256[] memory dates
    )
    {
        uint256 count = myDonationsCount();
        values = new uint256[](count);
        dates = new uint256[](count);

        for (uint256 i = 0; i < count; i++) {
            Donation storage donation = _donations[msg.sender][i];
            values[i] = donation.value;
            dates[i] = donation.date;
        }
    }
}
```

```

    }

    return (values, dates);
}
}

```

With these changes, our tests should pass:

```

$ truffle test

Compiling your contracts...
...omitted...

Contract: Fundraiser
  initialization
  ...omitted...
  setBeneficiary
  ...omitted...
  making donations
    ✓ increases myDonationsCount (107ms)
    ✓ includes donation in myDonations (87ms)
    ✓ increases the totalDonations amount (92ms)

11 passing (2s)

```

Great! Now we need to do something similar for the donations count. Let's add the new test to our "making donations" group:

```

it("increases donationsCount", async () => {
  const currentDonationsCount = await fundraiser.donationsCount();
  await fundraiser.donate({from: donor, value});
  const newDonationsCount = await fundraiser.donationsCount();

  assert.equal(
    1,
    newDonationsCount - currentDonationsCount,
    "donationsCount should increment by 1");
});

```

Running our tests should fail since the `donationsCount` function has not yet been defined. Just like with our `totalDonations` functionality, in order for this test to pass, we must add the public state variable and increment it in the `donate` function. Since we are incrementing by a constant of 1 and the chances of this value ever exceeding the bounds of a `uint256` are extremely low, we can use `++` to increment this value. Let's add these to our contract, as illustrated in [Example 6-16](#).

Example 6-16. Increment the donors count

```

pragma solidity >0.4.23 <0.7.0;

import "openzeppelin-solidity/contracts/ownership/Ownable.sol";

```

```

import "openzeppelin-solidity/contracts/math/SafeMath.sol";

contract Fundraiser is Ownable {
    using SafeMath for uint256;

    ...omitted...

    uint256 public totalDonations;
    uint256 public donationsCount;

    constructor(
        ...omitted...
    )
    public
    {
        ...omitted...
    }

    function setBeneficiary(address payable _beneficiary) public onlyOwner {
        beneficiary = _beneficiary;
    }

    function myDonationsCount() public view returns(uint256) {
        return _donations[msg.sender].length;
    }

    function donate() public payable {
        Donation memory donation = Donation({
            value: msg.value,
            date: block.timestamp
        });
        _donations[msg.sender].push(donation);
        totalDonations = totalDonations.add(msg.value);
        donationsCount++;
    }

    function myDonations() public view returns(
        uint256[] memory values,
        uint256[] memory dates
    )
    {
        uint256 count = myDonationsCount();
        values = new uint256[](count);
        dates = new uint256[](count);

        for (uint256 i = 0; i < count; i++) {
            Donation storage donation = _donations[msg.sender][i];
            values[i] = donation.value;
            dates[i] = donation.date;
        }
    }

    return (values, dates);
}

```

```
    }
}
```

Running our tests, we should now have everything passing:

```
$ truffle test

Contract: Fundraiser
  initialization
    ...omitted...
  setBeneficiary
    ...omitted...
  making donations
    ✓ increases myDonationsCount (111ms)
    ✓ includes donation in myDonations (84ms)
    ✓ increases the totalDonations amount (101ms)
    ✓ increases donationsCount (144ms)

12 passing (3s)
```

We are almost done with our donation feature! The last thing we are going to look into is emitting an event when a donation is made.

Events

Events are essentially a way to write to the Ethereum logs. These logs send out notifications when new entries are made and have been set up in a way that you can subscribe or watch for new entries. Since the logs are considered part of the blockchain, writing to them is not allowed in `view` or `pure` functions.

You can test for events by looking at the transaction receipt. Let's create the following test in our "making donation" block that will assert a specific event has been emitted:

```
it("emits the DonationReceived event", async () => {
  const tx = await fundraiser.donate({from: donor, value});
  const expectedEvent = "DonationReceived";
  const actualEvent = tx.logs[0].event;

  assert.equal(actualEvent, expectedEvent, "events should match");
});
```

Running our tests, we should see the following failure:

```
$ truffle test
Using network 'test'.

> Compiling ./contracts/Fundraiser.sol
Contract: Fundraiser
  initialization
    ...omitted...
```

```
1) Contract: Fundraiser
   making donations
   emits the DonationReceived event:
TypeError: Cannot read property 'event' of undefined
  at Context.it (test/fundraiser_test.js:119:38)
  at process._tickCallback (internal/process/next_tick.js:68:7)
```

Our test is failing because our log at index 0 is undefined. This means no events were emitted. Let's define our event and then update our donate function to emit the event whenever we receive a donation, as illustrated in [Example 6-17](#).

Example 6-17. Creating events

```
...omitted...
contract Fundraiser is Ownable {
  ...omitted...
  mapping(address => Donation[]) private _donations;

  event DonationReceived(address indexed donor, uint256 value);

  ...omitted...
  constructor(
    ...omitted...
  )
  public
  {
    ...omitted...
  }

  ...omitted...

  function donate() public payable {
    Donation memory donation = Donation({
      value: msg.value,
      date: block.timestamp
    });
    _donations[msg.sender].push(donation);
    totalDonations = totalDonations.add(msg.value);
    donationsCount++;

    emit DonationReceived(msg.sender, msg.value);
  }
}
```

Here we define the `DonationReceived` event. The event takes the address of the donor and the amount. Notice the address is marked as `indexed`. This means that the EVM will make it easier for subscribers to filter the events that may be relevant to them. Up to three parameters can be indexed when defining an event.

With our event defined, we now update the `donate` function to emit the event. Running our tests, we should now see everything pass.



Events can also be used as a more cost-effective (lower-gas) means of storing data on the chain. The information in events is accessible to subscribers, but it is not available from within a smart contract.

```
$ truffle test
Using network 'test'.

> Compiling ./contracts/Fundraiser.sol
Contract: Fundraiser
  initialization
  ...omitted...
  making donations
    ✓ increases myDonationsCount (118ms)
    ✓ includes donation in myDonations (103ms)
    ✓ increases the totalDonations amount (107ms)
    ✓ increases donationsCount (116ms)
    ✓ emits the DonationReceived event (61ms)

13 passing (2s)
```

That takes care of everything we wanted to discuss about making a donation. We'll now turn our attention to transferring funds to our beneficiary address.

Withdrawing Funds

The next thing to work on is transferring the balance of the contract to the beneficiary. This transfer could happen at the time of each donation, but it is considered a best practice to separate the deposit and withdraw actions.

Remember, the beneficiary address is something that is in control of something like Coinbase, and without the private key for that address, we cannot send transactions to withdraw the funds directly. We will turn to our owner address as the address that can trigger the withdraw.

For our withdraw function, we are going to want to do the following:

- Require the sender to match the owner address
- Transfer the contract balance to the beneficiary
- Emit a withdraw event

Let's add the test group and tests illustrated in [Example 6-18](#) to our test file.

Example 6-18. Adding withdraw test group

```
describe("withdrawing funds", () => {
  beforeEach(async () => {
    await fundraiser.donate(
      {from: accounts[2], value: web3.utils.toWei('0.1')}
    );
  });

  describe("access controls", () => {
    it("throws an error when called from a non-owner account", async () => {
      try {
        await fundraiser.withdraw({from: accounts[3]});
        assert.fail("withdraw was not restricted to owners")
      } catch(err) {
        const expectedError = "Ownable: caller is not the owner"
        const actualError = err.reason;
        assert.equal(actualError, expectedError, "should not be permitted")
      }
    });

    it("permits the owner to call the function", async () => {
      try {
        await fundraiser.withdraw({from: owner});
        assert(true, "no errors were thrown");
      } catch(err) {
        assert.fail("should not have thrown an error");
      }
    });
  });
});
```

As stated earlier in the chapter, when we are testing functions that have permissions, we should test for when the permission condition is met and when it fails.

Running our tests, we should see two failures, as shown here:

```
$ truffle test
Using network 'test'.

Compiling your contracts...
=====
> Compiling ./contracts/Fundraiser.sol
Contract: Fundraiser
  initialization
    ...omitted...
  access controls
    1) throws an error when called from a non-owner account
    2) permits the owner to call the function
```

13 passing (2s)

```
2 failing
```

```
1) Contract: Fundraiser
    withdrawing funds
    access controls
        throws an error when called from a non-owner account:
    AssertionError: should not be permitted: expected undefined to
    equal 'Ownable: caller is not the owner'
        at Context.it (test/fundraiser_test.js:130:18)

2) Contract: Fundraiser
    withdrawing funds
    access controls
        permits the owner to call the function:
    AssertionError: should not have thrown an error
        at Context.it (test/fundraiser_test.js:139:18)
```

To fix both the tests, all we need to do is define the `withdraw` function and use the `onlyOwner` modifier to restrict access. These first two tests do not check for any other side effects, and therefore the function body can remain empty for now. Add the following function definition to your Fundraiser contract:

```
function withdraw() public onlyOwner {
}
```

Running our tests, we are back to green!

```
$ truffle test
Using network 'test'.

Compiling your contracts...
=====
> Compiling ./contracts/Fundraiser.sol
Contract: Fundraiser
  initialization
  ...omitted...
  withdrawing funds
  access controls
    ✓ throws an error when called from a non-owner account (60ms)
    ✓ permits the owner to call the function (42ms)
```

```
15 passing (2s)
```

Next, we need to transfer the balance of the contract to the beneficiary. The observable side effects are that the contract balance goes to 0 and the beneficiary balance increases by what was previously in the contract balance. Let's use the following test to make our assertions:

```
it("transfers balance to beneficiary", async () => {
  const currentContractBalance = await web3.eth.getBalance(fundraiser.address);
  const currentBeneficiaryBalance = await web3.eth.getBalance(beneficiary);
```

```

    await fundraiser.withdraw({from: owner});

    const newContractBalance = await web3.eth.getBalance(fundraiser.address);
    const newBeneficiaryBalance = await web3.eth.getBalance(beneficiary);
    const beneficiaryDifference = newBeneficiaryBalance - currentBeneficiaryBalance;

    assert.equal(
      newContractBalance,
      0,
      "contract should have a 0 balance"
    );
    assert.equal(
      beneficiaryDifference,
      currentContractBalance,
      "beneficiary should receive all the funds"
    );
  });
}

```

Much like our counter tests from making donations, we gather the required state (balances) before the withdrawal, make the withdrawal, and then validate the changes. In order to access the balances, we use the `getBalance` method from Web3 and we need to access the `address` property off of our `Fundraiser` instance. With all this set up, running our tests provides the following failure:

```

$ truffle test
Using network 'test'.

> Compiling ./contracts/Fundraiser.sol
Contract: Fundraiser
  initialization
    ...omitted...

  withdrawing funds
    1) transfers balance to beneficiary

13 passing (3s)
1 failing

1) Contract: Fundraiser
   withdrawing funds
     transfers balance to beneficiary:
AssertionError: contract should have a 0 balance: expected
'1000000000000000' to equal 0
  at Context.it (test/fundraiser_test.js:154:14)
  at process._tickCallback (internal/process/next_tick.js:68:7)

```

Our problem here is that our balance hasn't changed. The `withdraw` function is currently empty, so this test seems to be failing for the correct reason. Let's update the `withdraw` function with the following code:

```
function withdraw() public onlyOwner {
    uint256 balance = address(this).balance;
    beneficiary.transfer(balance);
}
```

In order to access the balance of the contract, we have to first cast the contract to an address type, as we did in the first line of the function. Once we have the balance, we can transfer it to the beneficiary. With these changes in place, let's run our tests:

```
$ truffle test

> Compiling ./contracts/Fundraiser.sol
Contract: Fundraiser
  initialization
  ...omitted...
  withdrawing funds
    ✓ transfers balance to beneficiary (66ms)

16 passing (3s)
```

With that test passing, our last piece is to emit the `Withdraw` event. Just like before, we will set up a test to check the transaction logs and then assert that our events match. Add the following test to our “withdrawing funds” group:

```
it("emits Withdraw event", async () => {
  const tx = await fundraiser.withdraw({from: owner});
  const expectedEvent = "Withdraw";
  const actualEvent = tx.logs[0].event;

  assert.equal(
    actualEvent,
    expectedEvent,
    "events should match"
  );
});
```

Running our tests should result in the following failure:

```
$ truffle test

> Compiling ./contracts/Fundraiser.sol
Contract: Fundraiser
  initialization
  ...omitted...
  withdrawing funds
    ✓ transfers balance to beneficiary (66ms)
    1) emits Withdraw event

16 passing (3s)
1 failing

1) Contract: Fundraiser
   withdrawing funds
```

```
    emits Withdraw event:  
TypeError: Cannot read property '0' of undefined  
  at Context.it (test/fundraiser_test.js:169:34)  
  at process._tickCallback (internal/process/next_tick.js:68:7)
```

Since we didn't log an event in the `withdraw` function, the log at index 0 is undefined. To get this test to pass, let's define the `Withdraw` event and make sure to emit it from our `withdraw` function. These changes are illustrated in [Example 6-19](#).

Example 6-19. Log Withdraw event

```
contract Fundraiser is Ownable {  
  
  event Withdraw(uint256 amount);  
  
  ...omitted...  
  
  function withdraw() public onlyOwner {  
    uint256 balance = address(this).balance;  
    beneficiary.transfer(balance);  
    emit Withdraw(balance);  
  }  
}
```

Running our tests one last time, we should see the following:

```
$ truffle test  
  
> Compiling ./contracts/Fundraiser.sol  
Contract: Fundraiser  
  initialization  
  ...omitted...  
  withdrawing funds  
    ✓ transfers balance to beneficiary (67ms)  
    ✓ emits Withdraw event (50ms)  
  access controls  
    ✓ throws an error when called from a non-owner account (59ms)  
    ✓ permits the owner to call the function (42ms)  
  
17 passing (3s)
```

With our `withdraw` function completed, there is one last thing to discuss, and that is the fallback function.

Fallback Functions

Fallback functions are functions that are unnamed and will supply the default behavior in the event that the contract receives ether through a plain transaction or if the contract is called with a method signature that does not match any of the defined functions. When defining this function, it cannot accept any parameters and must be

marked external. Though it is not required, it is a good idea to go ahead and make it payable as well.

For our purposes, we will use this to simulate the idea of an anonymous donation. Since the blockchain will have a record of the transaction, it will not truly be anonymous, but maybe the address used to send the donation was set up specifically to make tracing it more difficult.

Our function should increment both the `totalDonations` and `donationsCount` state variables, and that's about it. This is pretty similar behavior to what we tested when making a donation. [Example 6-20](#) shows the new test group for our fallback behavior.

Example 6-20. Testing fallback behavior

```
describe("fallback function", () => {
  const value = web3.utils.toWei('0.0289');

  it("increases the totalDonations amount", async () => {
    const currentTotalDonations = await fundraiser.totalDonations();
    await web3.eth.sendTransaction(
      {to: fundraiser.address, from: accounts[9], value}
    );
    const newTotalDonations = await fundraiser.totalDonations();

    const diff = newTotalDonations - currentTotalDonations;

    assert.equal(
      diff,
      value,
      "difference should match the donation value"
    )
  });

  it("increases donationsCount", async () => {
    const currentDonationsCount = await fundraiser.donationsCount();
    await web3.eth.sendTransaction(
      {to: fundraiser.address, from: accounts[9], value}
    );
    const newDonationsCount = await fundraiser.donationsCount();

    assert.equal(
      1,
      newDonationsCount - currentDonationsCount,
      "donationsCount should increment by 1");
  });
});
```

The most significant change from our earlier tests is the use of `sendTransaction`. This will allow us to invoke the fallback function. With this in place, we can run the tests:

```
$ truffle test

...omitted...
fallback function
  1) increases the totalDonations amount

17 passing (3s)
2 failing

  1) Contract: Fundraiser
    fallback function
      increases the totalDonations amount:
    Error: Returned error: VM Exception while processing transaction: revert

  2) Contract: Fundraiser
    fallback function
      increases donationsCount:
    Error: Returned error: VM Exception while processing transaction: revert
```

Without a payable fallback function, our tests revert. Let's go ahead and add our fallback function in [Example 6-21](#).

Example 6-21. Fallback function

```
function () external payable {
  totalDonations = totalDonations.add(msg.value);
  donationsCount++;
}
```

Let's run the tests one last time:

```
$ truffle test

Contract: Fundraiser
  initialization
    ✓ gets the beneficiary name (54ms)
    ✓ gets the beneficiary url (43ms)
    ✓ gets the beneficiary image url
    ✓ gets the beneficiary description
    ✓ gets the beneficiary
    ✓ gets the owner
  setBeneficiary
    ✓ updated beneficiary when called by owner account (78ms)
    ✓ throws an error when called from a non-owner account (68ms)
  making donations
    ✓ increases myDonationsCount (111ms)
    ✓ includes donation in myDonations (133ms)
    ✓ increases the totalDonations amount (137ms)
    ✓ increases donationsCount (107ms)
    ✓ emits the DonationReceived event (90ms)
  withdrawing funds
```

```
✓ transfers balance to beneficiary (91ms)
✓ emits Withdraw event (46ms)
access controls
    ✓ throws an error when called from a non-owner account (49ms)
    ✓ permits the owner to call the function (45ms)
fallback function
    ✓ increases the totalDonations amount (81ms)
    ✓ increases donationsCount (80ms)
```

19 passing (4s)

Nice job! We have completed all the features for our Fundraiser contract. But before heading off for a break, let's recap this chapter.

Summary

This chapter introduced a lot of concepts and it is worth taking a moment to summarize them.

We first built up our constructor function using the needs of the UI to influence the properties we needed to support. In order to make sure our constructor function was working properly, we wrote a lot of getter functions for all the state variables we set up. When setting up our beneficiary address, we got to distinguish between the `address` and `address payable` type, with the latter being able to receive ether.

Once our constructor was completed, we focused a lot on the donation functionality. This led us to creating our first custom type with the `Donation` struct and using a `mapping` type to associate the donor with their donations. We also got to explore the differences between dynamic storage arrays and fixed-size memory arrays. The last piece of our donation functionality included emitting the `DonationReceived` event. This even included an indexed field for the donor.

We then moved on to the `withdraw` function. This let us transfer the balance of the contract to our beneficiary, and we finished the Fundraiser contract by implementing a fallback function.

In our next chapter, we are going to focus on another contract, the FundraiserFactory contract. This will provide most of the functionality for the home page. Whenever you are ready, jump over to the next chapter.

FundraiserFactory

In the last chapter, we created the Fundraiser contract and implemented the functionality to support creating donations and withdrawing funds. The Fundraiser contract has a lot of functionality, but currently it has no means of being created. In this chapter, we are going to add the contract that will create the individual instances of Fundraiser using a pattern that is likely familiar: the Factory pattern.

At the end of the chapter, we will also walk through grabbing the UI from the GitHub repo and deploying to Ganache. Once it has been deployed, we can launch the app in our browser, create new fundraisers, make donations, and withdraw the funds.

Migrating Our FundraiserFactory

Our Fundraiser contract from the last chapter was designed with the idea that it would be initialized from another contract, the FundraiserFactory, and therefore did not require its own migration. The FundraiserFactory, however, will need to be deployed so that our users can interact with the factory and create their own fundraisers.

Create a new file in the test directory using the following command:

```
$ touch test/fundraiser_factory_test.js
```

Let's add a deployment test case to our new file:

```
const FundraiserFactoryContract = artifacts.require("FundraiserFactory");

contract("FundraiserFactory: deployment", () => {
  it("has been deployed", async () => {
    const fundraiserFactory = FundraiserFactoryContract.deployed();
    assert(fundraiserFactory, "fundraiser factory was not deployed");
  });
});
```

Running our tests should throw an error with the following message:

```
Error: Could not find artifacts for FundraiserFactory from any sources
```

In order to get past this error, we'll need to define an empty contract with the FundraiserFactory name. Let's create the *contracts/FundraiserFactory.sol* file and then define our contract, as illustrated next:

```
$ touch contracts/FundraiserFactory.sol
pragma solidity >0.4.23 <0.7.0;

contract FundraiserFactory {
```

```
}
```

Running our tests now will produce the following output:

```
$ truffle test
Using network 'test'.

Compiling your contracts...
> Compiling ./contracts/Fundraiser.sol
> Compiling ./contracts/FundraiserFactory.sol
Contract: FundraiserFactory: deployment
  1) has been deployed
    > No events were emitted

...omitted...

19 passing (3s)
1 failing

1) Contract: FundraiserFactory: deployment
   has been deployed:
     Error: FundraiserFactory has not been deployed to detected network
     ...omitted...
```

This failure means it is time to write a migration. Create the migration file with the following command:

```
touch migrations/2_deploy_fundraiser_factory.js
```

Then add the following deployment code to the file:

```
const FundraiserFactoryContract = artifacts.require("FundraiserFactory");

module.exports = function(deployer) {
  deployer.deploy(FundraiserFactoryContract);
}
```

With the migration in place, let's run our tests:

```
$ truffle test
Using network 'test'.

Compiling your contracts...
> Compiling ./contracts/Fundraiser.sol
> Compiling ./contracts/FundraiserFactory.sol

Contract: FundraiserFactory: deployment
✓ has been deployed

Contract: Fundraiser
...omitted...

16 passing (3s)
```

With our deployment test out of the way, we can begin working on the primary purpose of our contract: creating Fundraisers.

Creating Fundraisers

The FundraiserFactory's primary job is to create new instances of Fundraisers. In order to do this, we will need to add a function that can initialize them on behalf of our users. The `createFundraiser` function we are about to create will take the data entered in the new page shown in [Figure 6-2](#) to create a new Fundraiser and then store a reference to it to allow pagination of all the Fundraisers available.

The observable behaviors that we can test against are that the `fundraisersCount` will increment and a `LogNewFundraiserCreated` event will be emitted.

Back in `test/fundraiser_factory_test.js`, let's add the test illustrated in [Example 7-1](#).

Example 7-1. Incrementing the fundraisersCount

```
contract("FundraiserFactory: createFundraiser", (accounts) => {
  let fundraiserFactory;
  // fundraiser args
  const name = "Beneficiary Name";
  const url = "beneficiaryname.org";
  const imageURL = "https://placekitten.com/600/350"
  const bio = "Beneficiary Description"
  const beneficiary = accounts[1];

  it("increments the fundraisersCount", async () => {
    fundraiserFactory = await FundraiserFactoryContract.deployed();
    const currentFundraisersCount = await fundraiserFactory.fundraisersCount();
    await fundraiserFactory.createFundraiser(
      name,
      url,
```

```

    imageURL,
    description,
    beneficiary
);
const newFundraisersCount = await fundraiserFactory.fundraisersCount();

assert.equal(
  newFundraisersCount - currentFundraisersCount,
  1,
  "should increment by 1"
)
});

```

In our new test, we created another contract block. Remember, Truffle will deploy a new instance of our contract when we use the contract function; this will prevent state from being maintained between test groups and prevent ordering issues on test runs. In our contract block, we set variables to hold onto the fundraiser data that would be entered into the form.

In the test, we retrieve the deployed instance, and like our counter tests in the last chapter, we save the current count (`currentFundraisersCount`), create a new fundraiser, and then check the count again. We expect the count to have increased by 1.

Running our test will generate the following output:

```

$ truffle test
Using network 'test'.
Compiling your contracts...
=====
> Compiling ./contracts/Fundraiser.sol
> Compiling ./contracts/FundraiserFactory.sol

Contract: FundraiserFactory: deployment
✓ has been deployed

Contract: FundraiserFactory: createFundraiser
  1) increments the fundraisersCount
    > No events were emitted

Contract: Fundraiser
...omitted...

20 passing (3s)
1 failing
1) Contract: FundraiserFactory: createFundraiser
   increments the fundraisersCount:
     TypeError: fundraiserFactory.fundraisersCount is not a function
       at Context.it (test/fundraiser_factory_test.js:22:61)
       at web3.eth.getBlockNumber.then.result (/usr/local/lib/node_modules/...
       at process._tickCallback (internal/process/next_tick.js:68:7)

```

Our test is letting us know we do not have the `fundraisersCount` function set up. In order to get past this error, we will need to set up a new state variable on our contract to store the fundraisers that have been created. Then we'll need to create a function to retrieve the length of the collection. The code in [Example 7-2](#) will get us past this first part.

Example 7-2. FundraiserContract counting the number of items in the collection

```
pragma solidity >0.4.23 <0.7.0;

import "./Fundraiser.sol";

contract FundraiserFactory {
    Fundraiser[] private _fundraisers;

    function fundraisersCount() public view returns(uint256) {
        return _fundraisers.length;
    }
}
```

Running our tests again, we should see a similar error as before, but this time the `createFundraiser` function will be the source of our trouble. Let's create this function, as illustrated next:

```
function createFundraiser(
    string memory name,
    string memory url,
    string memory imageURL,
    string memory description,
    address payable beneficiary
)
public
{
    Fundraiser fundraiser = new Fundraiser(
        name,
        url,
        imageURL,
        description,
        beneficiary,
        msg.sender
    );
    _fundraisers.push(fundraiser);
}
```

Looking at this code, we are taking in all the parameters from the form and then initializing a new fundraiser. This is the first time we have created a new instance of a contract from a contract! When initializing contracts, we will receive back the address of the new instance, and then we place that in our `Fundraisers` array.

Running our tests, we should see everything pass:

```
$ truffle test

Using network 'test'.
Compiling your contracts...
> Compiling ./contracts/Fundraiser.sol
> Compiling ./contracts/FundraiserFactory.sol
Contract: FundraiserFactory: deployment
  ✓ has been deployed
Contract: FundraiserFactory: createFundraiser
  ✓ increments the fundraisersCount (213ms)

Contract: Fundraiser
...omitted...

21 passing (3s)
```

With that test passing, we can move on to testing that the `FundraiserCreated` event has been emitted. We will start by adding the following test:

```
it("emits the FundraiserCreated event", async () => {
  fundraiserFactory = await FundraiserFactoryContract.deployed();
  const tx = await fundraiserFactory.createFundraiser(
    name,
    url,
    imageURL,
    description,
    beneficiary
  );
  const expectedEvent = "FundraiserCreated";
  const actualEvent = tx.logs[0].event;

  assert.equal(
    actualEvent,
    expectedEvent,
    "events should match"
  );
});
```

Running our tests will generate the following failure:

```
$ truffle test
Using network 'test'.

Compiling your contracts...
...omitted...
Contract: FundraiserFactory: createFundraiser
  ✓ increments the fundraisersCount (216ms)
  1) emits the FundraiserCreated event
    ...omitted...

Contract: Fundraiser
...omitted...
```

```
21 passing (3s)
1 failing

1) Contract: FundraiserFactory: createFundraiser
   emits the FundraiserCreated event:
TypeError: Cannot read property 'event' of undefined
  at Context.it (test/fundraiser_factory_test.js:49:36)
  at process._tickCallback (internal/process/next_tick.js:68:7)
```

In order to pass this test, we'll need to define a new event and then emit it from the `createFundraiser` function. The event will be called `FundraiserCreated` and will include the address of the new fundraiser and the address of the owner. Let's update our `FundraiserFactory` contract to look like [Example 7-3](#).

Example 7-3. Add FundraiserCreated event

```
pragma solidity >0.4.23 <0.7.0;

import "./Fundraiser.sol";

contract FundraiserFactory {
    Fundraiser[] private _fundraisers;

    event FundraiserCreated(Fundraiser indexed fundraiser, address indexed owner);

    function createFundraiser(
        string memory name,
        string memory url,
        string memory imageURL,
        string memory bio,
        address payable beneficiary
    )
        public
    {
        Fundraiser fundraiser = new Fundraiser(
            name,
            url,
            imageURL,
            description,
            beneficiary,
            msg.sender
        );
        _fundraisers.push(fundraiser);
        emit FundraiserCreated(fundraiser);
    }

    function fundraisersCount() public view returns(uint256) {
        return _fundraisers.length;
    }
}
```

Running our test, we should now see it pass! With creation of Fundraisers completed, we will turn our attention to getting data out of the Fundraisers collection. This next bit of functionality will support the bottom of the home page shown in [Figure 6-1](#).

Viewing Available Fundraisers

In order for our users to find the fundraisers that have been created, we need a way of displaying the options. One approach would be to get all of the fundraisers created in a single call to our contract, which is similar to what we did for the `myDonations` function in our Fundraiser contract. That approach works great when we expect the collection sizes to remain relatively small, but if our app gains popularity, we are going to want to break the loading of this data into chunks. Even though viewing data from the blockchain is free, meaning it doesn't cost gas, it is still using resources on the EVM. If a user comes to our application and sees the fundraiser they want right away, we end up wasting a lot of those resources. The smaller chunks will also allow our page to render faster than waiting for all the items to be returned from a single request.

We are going to borrow from our SQL friends and create a function that will take `limit` and `offset` parameters. We will also add a `maxLimit` constant to our FundraiserFactory contract to cap the number of records that can be requested at a time. In other words, if a request comes in asking for 50 records but our cap is 20, they will receive only 20 records. In the case where a user request includes an offset that is greater than the `fundraiserCount`, we will exit with an out-of-bounds error.

Because of the number of scenarios we will need to cover in our tests, we will create a helper method to initialize a new instance of our FundraiserFactory contract instead of using the deployed instance, and then create a passed-in number of fundraisers.

Testing Pagination When Empty

When we first deploy our FundraiserFactory, the collection of Fundraisers will be empty. That seems like the perfect starting case for our tests.

The code in [Example 7-4](#) includes our setup and the first test.

Example 7-4. Setting up retrieval of our fundraisers

```
contract("FundraiserFactory: fundraisers", (accounts) => {
  async function createFundraiserFactory(fundraiserCount, accounts) {
    const factory = await FundraiserFactoryContract.new();
    await addFundraisers(factory, fundraiserCount, accounts);
    return factory;
  }

  async function addFundraisers(factory, count, accounts) {
```

```

const name = "Beneficiary";
const lowerCaseName = name.toLowerCase();
const beneficiary = accounts[1];

for (let i=0; i < count; i++) {
  await factory.createFundraiser(
    // create a series of fundraisers. The index will be used
    // to make them each unique
    `${name} ${i}`,
    `${lowerCaseName}${i}.com`,
    `${lowerCaseName}${i}.png`,
    `Description for ${name} ${i}`,
    beneficiary
  );
}

describe("when fundraisers collection is empty", () => {
  it("returns an empty collection", async () => {
    const factory = await createFundraiserFactory(0, accounts);
    const fundraisers = await factory.fundraisers(10, 0);
    assert.equal(
      fundraisers.length,
      0,
      "collection should be empty"
    );
  });
});
});

```

Reviewing this setup, our `createFundraiserFactory` function creates a new instance of the `FundraiserFactory` and then passes that into the `addFundraisers` function. The `addFundraisers` function then creates the passed-in count of fundraisers.

Our first test is the empty collection scenario. When we run this test, we should get a test failure indicating the `fundraisers` function doesn't exist, like in the output here:

```

$ truffle test
Using network 'test'.
...omitted...
Contract: FundraiserFactory: fundraisers
  when fundraisers collection is empty
    1) returns an empty collection
      > No events were emitted
...omitted...

1 failing

1) Contract: FundraiserFactory: fundraisers
   when fundraisers collection is empty
     returns an empty collection:
     TypeError: factory.fundraisers is not a function

```

```
at Context.it (test/fundraiser_factory_test.js:86:41)
at process._tickCallback (internal/process/next_tick.js:68:7)
```

Let's add the function to our contract, returning the empty collection. The code is illustrated in [Example 7-5](#).

Example 7-5. Defining the fundraisers function

```
function fundraisers(uint256 limit, uint256 offset)
  public
  view
  returns(Fundraiser[] memory coll)
{
  return coll;
}
```

Our tests should all be passing again, but we still have a ways to go before we can say our function is behaving correctly. Let's jump into our next group of tests, which are focused on how many results will come back.

Testing the Limit

For the next few tests, we are going to keep our offset at 0. We will create 30 fundraisers and will vary our limit, which is the number of items we want to receive back. If we ask for 10, we'll get 10 results back; asking for 20 will get us 20; asking for 30 will get us 20...wait, what? Remember, we mentioned that we'll put a cap on the number of items that can be requested—we'll set our limit to 20. These tests are illustrated in [Example 7-6](#).

Example 7-6. Testing the limit parameter

```
describe("varying limits", async () => {
  let factory;
  beforeEach(async () => {
    factory = await createFundraiserFactory(30, accounts);
  })

  it("returns 10 results when limit requested is 10", async ()=>{
    const fundraisers = await factory.fundraisers(10, 0);
    assert.equal(
      fundraisers.length,
      10,
      "results size should be 10"
    );
  });

  // xit marks the test as pending
  xit("returns 20 results when limit requested is 20", async ()=>{
    const fundraisers = await factory.fundraisers(20, 0);
  });
}
```

```

        assert.equal(
            fundraisers.length,
            20,
            "results size should be 20"
        );
    });

xit("returns 20 results when limit requested is 30", async ()=>{
    const fundraisers = await factory.fundraisers(30, 0);
    assert.equal(
        fundraisers.length,
        20,
        "results size should be 20"
    );
});
})
)

```

In this code sample, our first test uses the normal `it` that we have been using, but the following tests are marked with `xit` to indicate they are pending. We can then remove the leading `x` when it's time to focus on that particular use case.

The size of the collection that is returned is now going to be the smaller of the `fundraisersCount` or the limit being requested. If we just return a collection of the limit requested, our 0 size test will break. [Example 7-7](#) shows our updated `fundraisers` function.

Example 7-7. Determining the size of the collection to return

```

function fundraisers(uint256 limit, uint256 offset)
    public
    view
    returns(Fundraiser[] memory coll)
{
    uint256 size = fundraisersCount() < limit ? fundraisersCount() : limit;
    coll = new Fundraiser[](size);

    return coll;
}

```

This will now pass our tests, but we're not getting the right data. We are getting back a collection of the right size, but it will be an array of the `0x0` address. We'll write more tests to assert against the contents in a moment, but for now, we will continue the focus on returning a collection of the appropriate size.

Moving on to the next test, we can remove the leading `x` to let it run. When we do, we will see that it too is passing and that we have one more pending test to work through. The next test will fail because we haven't declared a max size.

Our failure is shown in this output:

```
$ truffle test
Using network 'test'.
...omitted..
Contract: FundraiserFactory: fundraisers
  when fundraisers collection is empty
    ✓ returns an empty collection (96ms)
  varying limits
    ✓ returns 10 results when limit requested is 10
    ✓ returns 20 results when limit requested is 20
    1) returns 20 results when limit requested is 30
      ...omitted...

Contract: Fundraiser
...omitted...

25 passing (10s)
1 failing

1) Contract: FundraiserFactory: fundraisers
   varying limits
     returns 20 results when limit requested is 30:

       results size should be 20
       + expected - actual
       -30
       +20
```

To get this test to pass, we need to create our max limit. We will set this as a constant state variable on our contract and then update our `fundraisers` function to use the minimum value of our `fundraisersCount`, `limit`, or `maxLimit`. Our updated function is shown in [Example 7-8](#).

Example 7-8. Limiting the size

```
contract FundraiserFactory {
  // most items that can be returned from fundraisers function
  uint256 constant maxLimit = 20;

  ...omitted...

  function fundraisers(uint256 limit, uint256 offset)
    public
    view
    returns(Fundraiser[] memory coll)
  {
    // size should be the smaller of the count or the limit
    uint256 size = fundraisersCount() < limit ? fundraisersCount() : limit;
    // size should not exceed the maxLimit
    size = size < maxLimit ? size : maxLimit;
```

```

        coll = new Fundraiser[](size);

        return coll;
    }
}

```

With this change in place, our collection size seems to be working properly. We'll now look into varying the offset.

Testing the Offset

The offset determines what index we use as a starting point. It will be in these examples that we also assert that we are getting the fundraisers that we expected. For these examples, we will keep the limit to just one and make sure that the beneficiary name includes the appropriate index. [Example 7-9](#) illustrates our next test group.

Example 7-9. Testing offset

```

describe("varying offset", () => {
  let factory;
  beforeEach(async () => {
    factory = await createFundraiserFactory(10, accounts);
  });

  it("contains the fundraiser with the appropriate offset", async ()=>{
    const fundraisers = await factory.fundraisers(1, 0);
    const fundraiser = await FundraiserContract.at(fundraisers[0]);
    const name = await fundraiser.name();
    assert.ok(await name.includes(0), `${name} did not include the offset`);
  });

  xit("contains the fundraiser with the appropriate offset", async ()=>{
    const fundraisers = await factory.fundraisers(1, 7);
    const fundraiser = await FundraiserContract.at(fundraisers[0]);
    const name = await fundraiser.name();
    assert.ok(await name.includes(7), `${name} did not include the offset`);
  });
});

```

In addition to the tests in [Example 7-9](#), you will need to add the following line to the top of the test file to load the Fundraiser:

```
const FundraiserContract = artifacts.require("Fundraiser");
```

In our new tests, we create 10 fundraisers through the FundraiserFactory and test against setting the offset to 0 and then setting the offset to 7. These numbers were arbitrarily selected; to be more complete, we could create a test helper setting the offset to all the values from 0 to 10 and assert that the offset is being adjusted appropriately.

Running the test will get a failure like this:

```
$ truffle test
...omitted...
Contract: FundraiserFactory: fundraisers
...omitted..
varying offset
  1) contains the fundraiser with the appropriate offset

    ...omitted...
26 passing (11s)
1 pending
1 failing

1) Contract: FundraiserFactory: fundraisers
   varying offset
     contains the fundraiser with the appropriate offset:
     Error: Cannot create instance of Fundraiser; no code at address 0x000000...
```

Now is when that `0x0` address mentioned previously is preventing us from passing our test. Now it's time to fill the collection we are returning with actual fundraiser addresses. We can update the body of our `fundraisers` function to look like [Example 7-10](#).

Example 7-10. Adding fundraisers to the collection

```
function fundraisers(uint256 limit, uint256 offset)
  public
  view
  returns(Fundraiser[] memory coll)
{
  uint256 size = fundraisersCount() < limit ? fundraisersCount() : limit;
  size = size < maxLimit ? size : maxLimit;
  coll = new Fundraiser[](size);

  for(uint256 i = 0; i < size; i++) {
    coll[i] = _fundraisers[i];
  }

  return coll;
}
```

That is now letting our first test pass. Let's move on to the second test. Remove the leading `x` and then run the tests again. Now our test fails, with a message similar to the following:

```
1) Contract: FundraiserFactory: fundraisers
   varying offset
     contains the fundraiser with the appropriate offset:
     Assertion Error: Beneficiary 0 did not include the offset: expected
       false to be truthy
```

We need to adjust our starting point when filling the collection with Fundraisers. Adjust the `for` loop in our `fundraisers` function to include the offset, and we'll get back to green tests.

```
for(uint256 i = 0; i < size; i++) {
    coll[i] = _fundraisers[offset + i];
}
```

There are just two more cases that we should handle. In the case where the offset is larger than our `fundraisersCount`, our application will need to exit with an out-of-bounds message. We should also watch out for cases in which the sum of the offset and limit would exceed the size of our `fundraisersCount`.

Let's create another test group with our boundary cases and add the tests in [Example 7-11](#).

Example 7-11. Boundary tests

```
describe("boundary conditions", () => {
  let factory;
  beforeEach(async () => {
    factory = await createFundraiserFactory(10, accounts);
  });

  it("raises out of bounds error", async () => {
    try {
      await factory.fundraisers(1, 11);
      assert.fail("error was not raised")
    } catch(err) {
      const expected = "offset out of bounds";
      assert.ok(err.message.includes(expected), `${err.message}`);
    }
  });

 xit("adjusts return size to prevent out of bounds error", async () => {
  try {
    const fundraisers = await factory.fundraisers(10, 5);
    assert.equal(
      fundraisers.length,
      5,
      "collection adjusted"
    );
  } catch(err) {
    assert.fail("limit and offset exceeded bounds");
  }
});
});
```

In our new example group, we first test the situation where our offset is outside of our `fundraiserCount`. Our expectation here is that the error thrown will include the out-of-bounds message.

Running our tests, we will have a failure that indicates an invalid opcode, like this error:

```
1) Contract: FundraiserFactory: fundraisers
   boundary conditions
   raises out of bounds error:
AssertionError: Returned error: VM Exception while processing transaction:
invalid opcode: expected false to be truthy
  at Context.it (test/fundraiser_factory_test.js:161:16)
```

This is happening when we are trying to access an index that doesn't exist in our `for` loop. We can add a `require` statement to give this error more meaning. Adding the following line of code to the top of our `fundraisers` function will be enough for our test to pass:

```
require(offset <= fundraisersCount(), "offset out of bounds");
```

Remove the leading `x` from our pending test, and running the test, we get a failure like this one:

```
1) Contract: FundraiserFactory: fundraisers
   boundary conditions
   adjusts return size to prevent out of bounds error:
AssertionError: limit and offset exceeded bouds
  at Context.it (test/fundraiser_factory_test.js:174:16)
```

In this case, the offset meets the criteria of our `require` statement, but the iteration in the `for` loop causes us to access places in memory beyond our `fundraisersCount`. We will update our `fundraisers` function to start the local `size` variable to represent the difference between the `fundraisersCount` and the offset. The rest of the logic will keep us from going too far. [Example 7-12](#) has the final version of our `fundraisers` function.

Example 7-12. Keeping in bounds

```
function fundraisers(uint256 limit, uint256 offset)
public
view
returns(Fundraiser[] memory coll)
{
    require(offset <= fundraisersCount(), "offset out of bounds");

    uint256 size = fundraisersCount() - offset;
    size = size < limit ? size : limit;
    size = size < maxLimit ? size : maxLimit;
    coll = new Fundraiser[](size);
```

```

    for(uint256 i = 0; i < size; i++) {
        coll[i] = _fundraisers[offset + i];
    }

    return coll;
}

```

Whew! That wraps up the Solidity code we'll need to support our application. The next section will walk through setting up the UI so that you can click around the application.

Setting Up the UI

Like our Greeter contract, we are going to download the UI and deploy locally to Ganache so that we can interact with our application via browser instead of only through the tests.

If you haven't already downloaded the book code, you can access it from [GitHub](#). Assuming the cloned repository is a sibling of our fundraiser directory, you can use the following command to copy the files:

```
$ cp -r ../hoscddev/chapter-10+11/ ./client
```

In a new terminal tab, enter the client directory and install the packages using the following commands. We'll reference this terminal as the client terminal.

```
$ cd client
$ npm install
```

Once all the packages have been installed, we can update our *truffle-config.js* file with host and network ID keys for the develop network. The final configuration will look like [Example 7-13](#).

Example 7-13. Truffle configuration

```

const path = require("path");

module.exports = {
    // See <http://truffleframework.com/docs/advanced/configuration>
    // to customize your Truffle configuration!
    contracts_build_directory: path.join(__dirname, "client/src/contracts"),
    networks: {
        develop: {
            host: "127.0.0.1",
            port: 8545,
            network_id: "*",
        }
    }
};

```

Fire up the Ganache app and create a new workspace. When prompted for a project, include the *truffle-config.js* file and save the workspace. With that running, we are ready to migrate our contracts.

In the terminal located at the root directory of our fundraiser application, use the following command:

```
$ truffle migrate --network develop
```

In our browser, import the new accounts created by the workspace to MetaMask by using the mnemonic in Ganache.

When the migration completes, we're ready to start the client server. Back in the client terminal, run the following command:

```
$ npm run start
```

The command will open a browser tab for you, taking you to your app. Congratulations! You can now create fundraisers and make donations on your local blockchain.

Summary

In this chapter, we created a contract mostly to handle the creation and the management of a potentially large collection of fundraisers. In doing this, we learned that contracts can initialize other contracts and we also dove into pagination.

We spent a lot of time testing our pagination cases to make sure we got it right. This led us to covering empty cases and various limits (some of which exceeded our `maxLimit`) and also moving the offset around, including to places outside the range of the collection. With these features wrapped up, it's time to start learning more about Web3 and the frontend technologies that we can use to build the UIs of our applications.

PART III

Interacting with Our Smart Contracts Through Web3

In this part, we will learn how to use Web3 and connect our smart contracts to an application so our users can interact with the blockchain easily. With Web3, we have a much simpler solution to a complicated problem. By using Web3, we can easily connect our contracts to our frontend and read and write data.

What Is Web3?

Web3 is a collection of JS libraries that lets you interact with an Ethereum node remotely or locally. Simply, it provides us with an API to use so we can easily work with the blockchain. Web3 works as a wrapper for JSON RPC to connect to a remote or local Ethereum node with either a HTTP or IPC connection. Web3 is basically a connection between the Ethereum blockchain and your smart contract.

Behind the scenes, Web3 uses JSON RPC. RPC is used in many different types of programming languages. You can learn more about JSON RPC [here](#). This chapter focuses on Web3 because Web3 makes connecting to an Ethereum node less complicated and much easier to understand than RPC.

The Frontend, Web3, and the Blockchain

In a traditional web 2.0 application, your user will interact with the frontend of an application such as React or Ember to make a request to the backend that will have databases, APIs, and models, which will then return a response from the backend. The frontend will serve up the data from the backend to the frontend, and the frontend will display it to your user.

The flow of a web 2.0 application is illustrated in [Figure 8-1](#).

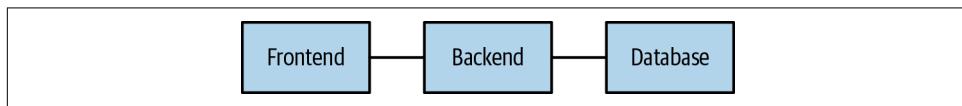


Figure 8-1. Connection between a traditional frontend and backend

If you have experience in traditional web development, Web3 is similar to an API request that is getting or adding data to the backend. But with Web3, you'll be reading and writing data to the blockchain instead of a traditional backend.

Your user will interact with the frontend of your application such as React, which will then use Web3 to make a request that connects to the blockchain. Web3 will interact with the blockchain through RPC and provide the response to the frontend, which you will then render to the user.

A Web3 application flow will look like [Figure 8-2](#).

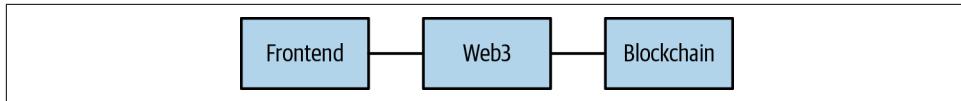


Figure 8-2. The connection between Web3, frontend, and backend

We'll take a look at a few of the methods to interact with the blockchain that we can use and that are made available from Web3 in the next section.

Web3 Methods

Web3 offers many methods that we can use to interact with the blockchain. We'll use these methods to read or write data to the blockchain from our React application. We can send ether from your address to another address, call methods, get the balance of a blockchain address, get gas costs, and more. We'll use these methods later when we connect our application to an Ethereum node.

Let's start by taking a look at a few examples of Web3 methods.

getAccounts()

The `getAccounts()` method can be used to return a list of your Web3 wallets.

It's important to note that `getAccounts` will only return your unlocked wallets. This might trip you up later, so always remember to Google error messages and follow this book carefully.

```
web3.eth.getAccounts()
```

getBlockNumber()

The `getBlockNumber` method will return the current block number.

```
web3.eth.getBlockNumber()  
> 8911
```

getBalance()

`getBalance` can be used to get the balance of the account you pass in as an argument. We'll use this later on when we connect our contracts to React, the frontend framework we'll be using.

Here's an example of using the `getBalance` method to find the balance of this address:

```
web3.eth.getBalance  
  web3.eth.getBalance("0xa589BDa5379fDE626dD3eBF462fA22F5dff43f30")  
    .then(console.log);
```

Here is the result:

```
"2000000000000"
```

sendTransaction()

If we wanted to send funds from one address to another, we would use `sendTransaction` to do it.

```
web3.eth.sendTransaction()
```

Here's an example of sending funds from one account to another:

```
web3.eth.sendTransaction({  
  from: "0xa589BDa5379fDE626dD3eBF462fA22F5dff43f30",  
  to: "0xa289BDa5379fDE626dM3eBF462fA22F5dff43f31",  
  value: web3.toWei(1, "ether"),  
})
```

The `from` param is the account you want to transfer the funds from. The `to` param is the account that you want to send the funds to. The `value` param is the amount of ether that you will send from the sender address to the receiver address.



Never try to write your address from another source. Always copy and paste when sending funds. It won't matter if you lose some test ether, but it will matter if you accidentally transfer real funds. Also always be sure to double-check that the address is correct before you send funds.

If we want to use Web3, we have to set a provider so Web3 knows which node to connect to:

```
web3.setProvider
```

We'll use this method later on when we set up a React application to work with our contracts.

For now, while we're developing locally, we'll set our provider to localhost. If we want to connect to our localhost, we'll set our provider to `localhost:8545` as shown in the following example:

```
new Web3(new Web3.providers.HttpProvider("http://localhost:8545"))
```

We'll use some of these Web3 methods later on when we build larger applications. Next, we'll learn exactly what a provider is.

Providers

A Web3 provider will tell your project which node you'll talk to. Simply, it's how your project can talk to the blockchain. This can be compared to the URL you might use to make an API call to in a traditional web application. The provider will send our RPC instructions to the correct Ethereum node.

There are various different ways to link to a running node. A provider links to a running node such as Parity or Geth. Once you're connected through a provider, the node can interact with the blockchain.

Later in the DApp we'll build, we'll connect to our provider as in the following. We'll call Web3 and set the provider for our application so our users can interact with the blockchain.

```
web3.setProvider(new web3.providers.HttpProvider('http://localhost:8545'))
```

If we wanted to check the current provider, we can use `web3.currentProvider`, which will return the current provider if we are using one and `null` if no provider exists.

Next, let's learn about how we can make a request to the blockchain and see the result returned through a promise.

Promises with Web3

We'll use promises quite a bit with Web3. Promises are a concept where something may or may not return a value in the future. We can attach a callback to a promise, which will either return an error if it does not resolve.

If we need to make multiple calls, promises are the best solution because they allow us to return the result or an error. If we wanted to use the `getBalance()` function to see a user's balance, we would use the following:

```
return new Promise (function (resolve, reject) {
  web3.eth.getBalance("0xa589...", function (error, balance) {
    if (error) {
      reject(error)
    } else {
      resolve(balance)
    }
  })
})
```

Without a promise, the call would not return the proper value. Here's a more complicated example of a `web3.js` file that uses promises, which you would use to set up

Web3 with your application. We'll use this later to interact with Web3 and our frontend.

```
import Web3 from 'web3'

let getWeb3 = new Promise(function(resolve, reject) {
  window.addEventListener('load', function() {
    var results
    var web3 = window.web3

    if (typeof web3 !== 'undefined') {
      web3 = new Web3(web3.currentProvider)

      results = {
        web3: web3
      }

      resolve(results)
    } else {
      var provider = new Web3.providers.HttpProvider('http://127.0.0.1:9545')

      web3 = new Web3(provider)

      results = {
        web3: web3
      }

      resolve(results)
    }
  })
})

export default getWeb3
```

In the preceding code snippet, we're first waiting for our page to load with our promise. After we have Web3, we'll use `resolve(results)` to view our promise results.

If we don't have a Web3 injection, we'll use an `else` statement and set it to a localhost and use Web3 locally. `http://127.0.0.1:9545` would be your own port, which is set up with your application.

Next, let's learn about MetaMask and how we'll interact with Web3 applications through it. We previously set it up in the book, but now let's learn about how we'll use it in the application in this book.

MetaMask for Web3 Injection

MetaMask is a browser extension that allows us to easily interact with websites that use Web3. MetaMask manages your Ethereum accounts and private keys. It's like a wallet but in your browser instead.

MetaMask will store your ether and let you sign transactions and pay gas fees. It also provides a very convenient way for users to interact with your Web3 application without running their own node. Refer back to the installation chapter for more information about MetaMask.

But how do we interact with the blockchain? We'll use the next method, `send`, to interact with the blockchain and open a MetaMask popover. Let's discuss the `send` method and how we can write to the blockchain with it.

Send (State Updates/Write)

The other type of method we'll use with Web3 is the `send` method. Your DApp will need to use the `send` method whenever you want to add to the blockchain. You'll see later when we set up our frontend for our DApp that the `send` method requires your user to use MetaMask, sign the transaction, and pay gas fees.

We'll use the `send` method for times that we need to alter the blockchain.

Next, let's learn about a less intrusive method: `call`. The `call` method does not cost gas, and it's used for methods where we might want to get the result of something in a contract. The `send` method forwards gas, so it is more invasive than the `call` method.

Call (Reads)

`Call` is used for `view` and `pure` functions.

While the `send` method will prompt your user to use MetaMask, the `call` method won't prompt your user to sign anything with MetaMask, and it won't cost any gas. Note that the `call` method is also safer to use because it doesn't forward gas. It also won't create a transaction on the blockchain, so it's safer to use because it's not publishing anything.

The return value is returned immediately and is a read-only function. Due to this, the `call` method is also faster. We're going to use the `call` method a lot in this book because it's fast and it won't cost any gas or transaction fees.

Summary

In this chapter, we've learned that Web3 is a collection of JS libraries that allow us to connect to an Ethereum node. Web3 is a wrapper for JSON RPC, and provides us an API we can use in our applications to interact with the blockchain in a simple way. We've learned about how Web3 is different from a traditional web 2.0 application.

We've also learned about how our frontend will connect to Web3, which will transfer data with the blockchain. And we've learned a few Web3 methods, such as `call` and `send`, which we'll begin to use in the next chapter.

In the next chapter, we'll work on connecting a smart contract to a UI. We will use Web3 in our React application to connect and interact with the blockchain.

Connecting the UI to Our Contracts

Let's get started with creating our DApp's UI. We'll use a frontend framework called React to implement the features we need to allow our users to interact with the blockchain. React will help us to easily create a frontend application with different routes for a user to navigate through, input forms and buttons for them to submit data, and more. In this chapter, we'll learn how to use React to work with the blockchain and submit functions that will read and write to the blockchain.

Why React?

First of all, what is React? React is a JavaScript library built by Facebook that is used to build frontend applications or the *client* side of an application. If your application uses React, anything that a user sees—the view layer—is being served from the React application. React uses components and data management, such as reducers and actions, along with providing JSX (a syntax extension) and a Virtual DOM, not to mention easily reusable library code that will help you build out the frontend to your application fast.

If you do become a blockchain engineer, most likely you'll need to be familiar with Web3 and React to develop Web3 applications. To learn more about React, take a look at its [official site](#).

Why do we want to use React? React is the most widely used frontend framework with Truffle. Because it is so widely used, it'll make finding documentation and answers to your questions so much easier. Most of the questions you'll find on Stack Overflow about the frontend and Ethereum will use React. In addition, most of the tutorials you'll find online will be written using React.

We're going to use [React Box](#). React Box is lightweight scaffolding that will allow us to quickly interact with our smart contracts through a React frontend. You can view the

documentation on the React Box GitHub repository and play around with it yourself. While React is a frontend framework, React Box is a quick way to get a React application running and interacting with Web3. Without React Box, we'd need to create a bunch of folders and setup that would hold our contracts, migrations, and Web3 setup.

Let's get started with setting up React Truffle Box so we can read and write to the blockchain.

Dev Setup

Now, let's set up our Greeter contract with React Truffle Box. We'll need to follow the steps in the [React Truffle Box](#) repository.

Create a new directory:

```
mkdir greeter-dapp
```

Change into the new directory:

```
cd greeter-dapp
```

Install Truffle globally:

```
npm install -g truffle
```

And run this command inside it to unbox the React Truffle Box:

```
truffle unbox react
```

Your Greeter project should now include the following files:

```
greeter
├── client
│   └── src
├── contracts
│   └──
├── migrations
└── test
    └── truffle-config.js
```

Our React application can be found inside `/client/src` in our `greeter` directory. Here you'll find all the frontend files we'll be using in this chapter and later on with our fundraiser application.

Let's start with getting our contracts imported to React and running with Truffle so we can call the methods from our Greeter contract and interact with them in the frontend of our application.

Truffle

Later in this chapter, we'll import our own Greeter contracts. But let's first get everything running with the standard React Truffle Box default contracts. Later on when you build your own Truffle applications, you can follow this same process to build your own unique blockchain application.

First, we'll run the following command to start our Truffle server:

```
truffle develop
```

`truffle develop` not only starts up your server for you, but creates accounts with ether that you can use with your DApp for testing. It also includes private keys and a mnemonic that you can use.

When it is running, you'll see something similar to this in your terminal:

```
Truffle Develop started at http://127.0.0.1:9545/
```

Accounts:

```
(0) 0xb94454c83ff541c82391b4331e75d2e2b16bae9e
(1) 0xf1f8d85aef05e7587cbc3ddae9753f4865a3a349
(2) 0xe6a27378e478eb6d3fe2d97c916e8b4813889d1e
(3) 0x3e41e8e0cdace3079cd6084380f00cd855cd28ea
(4) 0x135d3e70c2a2633141a433fa349d18d3dcf36c40
(5) 0x3c84bd6957fc3d6093be672df7e2b40caa427f5e
(6) 0x661c74f3765d87ab881fe09eec4ef019fc8098bd
(7) 0x44ee71f3e047ac88b203775621a4e8bfd4112b24
(8) 0x3a9553825d228b6c2f5fdfcc6d6cd2a96a527c0a
(9) 0x2ddb954595ef1a9f5f27ed22e04f3009c8789a03
```

Private Keys:

```
(0) 98831bde4e476ffc7dd7021860bd93c4a0ac95b08368057256db49cef900990d
(1) 632b963c3d3b1b03197e6120fb5200bbfe155f36f60c9e0a461a046aca3e53e6
(2) df389a349ae4f7c29fc4da3ee70e5bcd0bf7e4eeeeb3cafcecb38cb43266bb1f
(3) 88eb15bb7e98a24b76eafcbecd4de419c17f2c5b00cb129ab84b276857d9f303
(4) 11b862620fd7150ba20a255a88bd4046c9d674d36d297811e06008fde588983a
```

```
Mnemonic: flush cupcake slice area urban slice motor click rocket response
lobster sustain(("mnemonics"))
```

⚠ Important ⚠ : This mnemonic was created for you by Truffle. It is not secure. Ensure you do not use it on production blockchains, or else you risk losing funds.

```
truffle(develop)>
```

Copy `http://127.0.0.1:9545/` and the first private key you see from the output after running `truffle develop`. We'll be using them later.

Once you see the interactive Truffle console is up, we can compile our Greeter contract with the `compile` command:

```
truffle(develop)> compile
```

Here's the output:

```
Compiling your contracts...
=====
> Compiling ./contracts/Migrations.sol
> Compiling ./contracts/SimpleStorage.sol
```

Running `compile` will create JSON files of all your contracts under `build/contracts` in your project.

Next, we'll deploy the default React Truffle Box contracts to our test Ethereum network by running the following:

```
migrate
```

Here's the output:

```
Starting migrations...
=====
> Network name:    'develop'
> Network id:      5777
> Block gas limit: 0x6691b7

1_initial_migration.js
=====
Deploying 'Migrations'

2_deploy_contracts.js
=====

> Saving migration to chain.
> Saving artifacts
-----
> Total cost:        0.0021003 ETH

Summary
=====
> Total deployments:  2
> Final cost:         0.00732816 ETH
```

Now that we have the Truffle side of things ready to go, let's get our React application up and running.

React Truffle Box Setup

Before we get the development of our frontend application started, we need to switch our MetaMask network to match what we saw earlier when we ran `truffle develop`.

We need this to be the same so we're connecting to the right localhost and serving up the right information.

When we ran `truffle develop`, we saw this:

```
Truffle Develop started at http://127.0.0.1:9545/
```

Copy `http://127.0.0.1:9545/` to your clipboard. Next, we need to open MetaMask and import that network. Open the MetaMask extension and select Main Ethereum Network. We'll need to select Custom RPC and enter `http://127.0.0.1:9545` where it says New RPC URL."

You should see this in the MetaMask extension after selecting Custom RPC, as illustrated in [Figure 9-1](#).

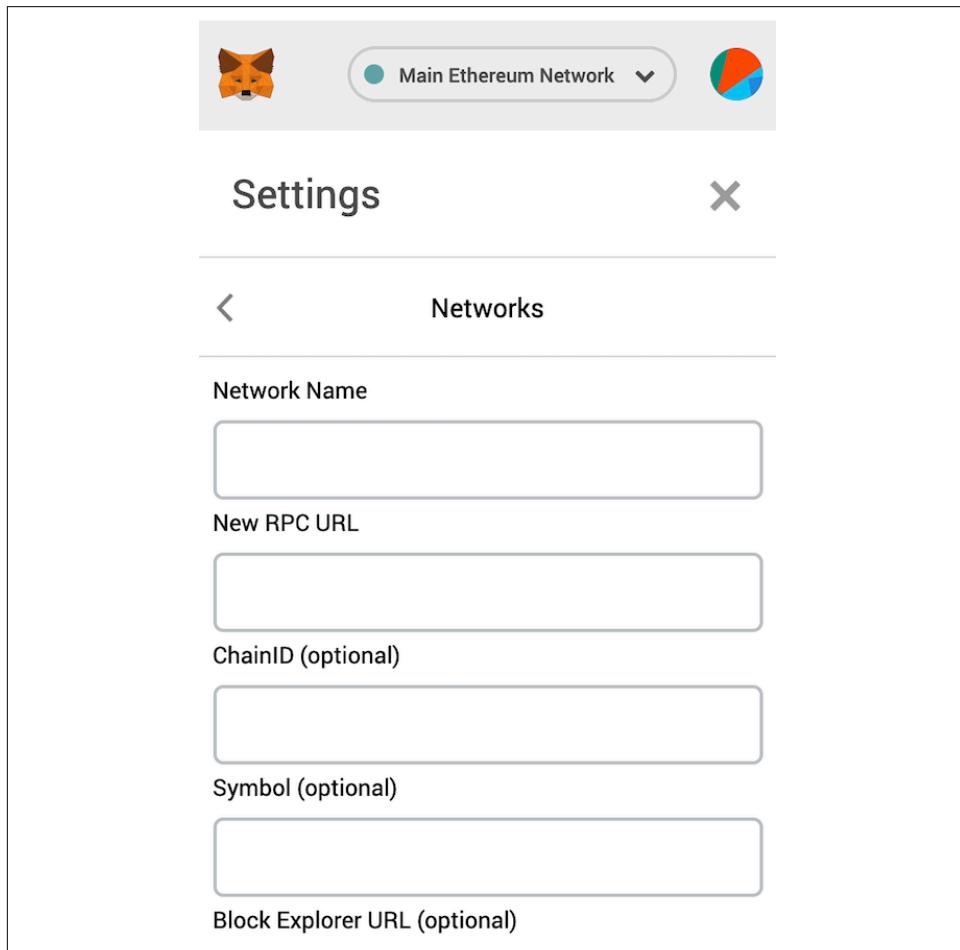


Figure 9-1. Select a custom RPC to connect to Truffle

Next, we'll need to import one of the fake test accounts that we saved earlier. In MetaMask, select the top right circular icon where you'll see something similar to the dropdown in Figure 9-2.

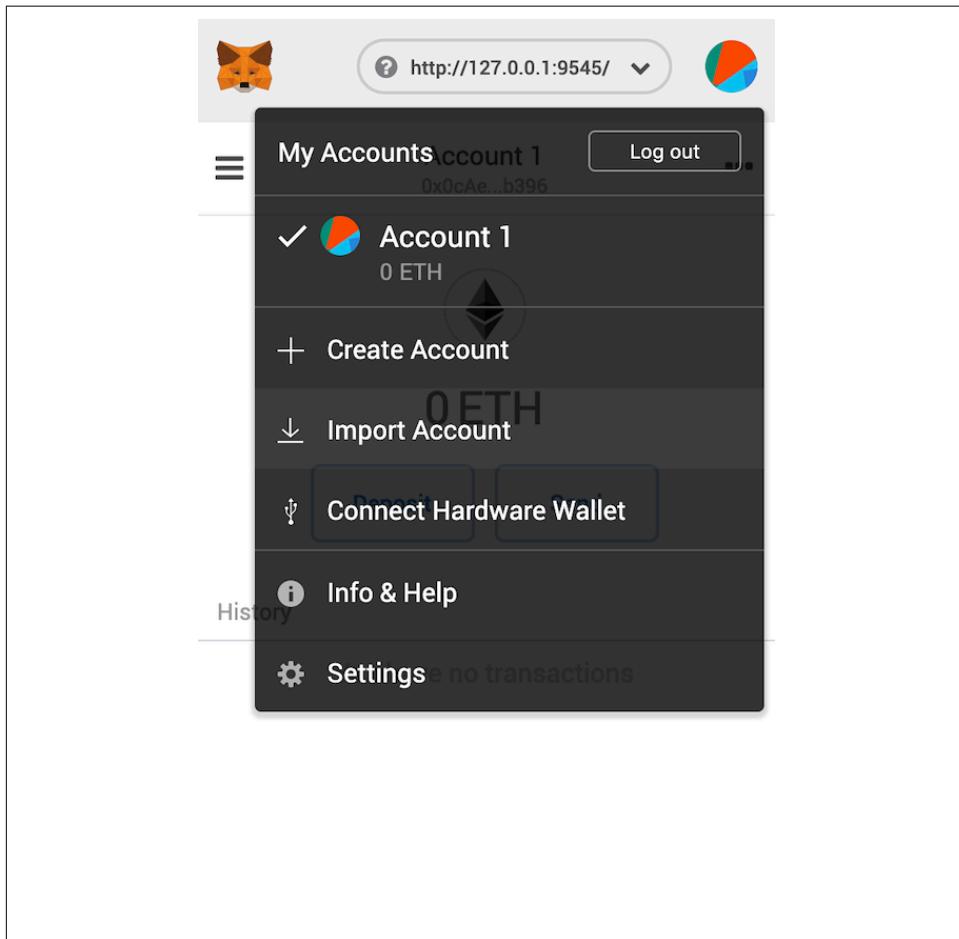


Figure 9-2. Import your test account from the Truffle Develop console

Paste the private key in the MetaMask import and select Import (see [Figure 9-3](#)).

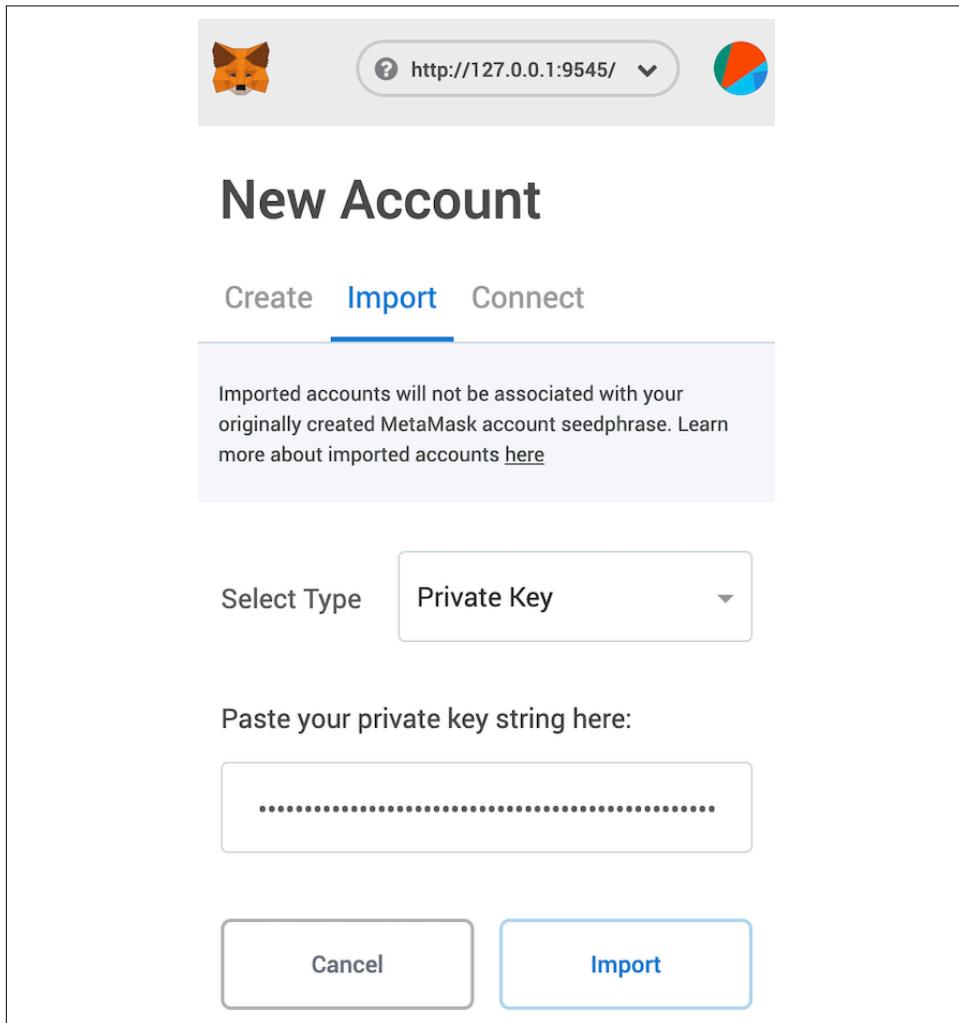


Figure 9-3. Enter your private key to import your test account

If the account was successfully imported, you'll see a bunch of ether show up in your new account, as illustrated in [Figure 9-4](#).

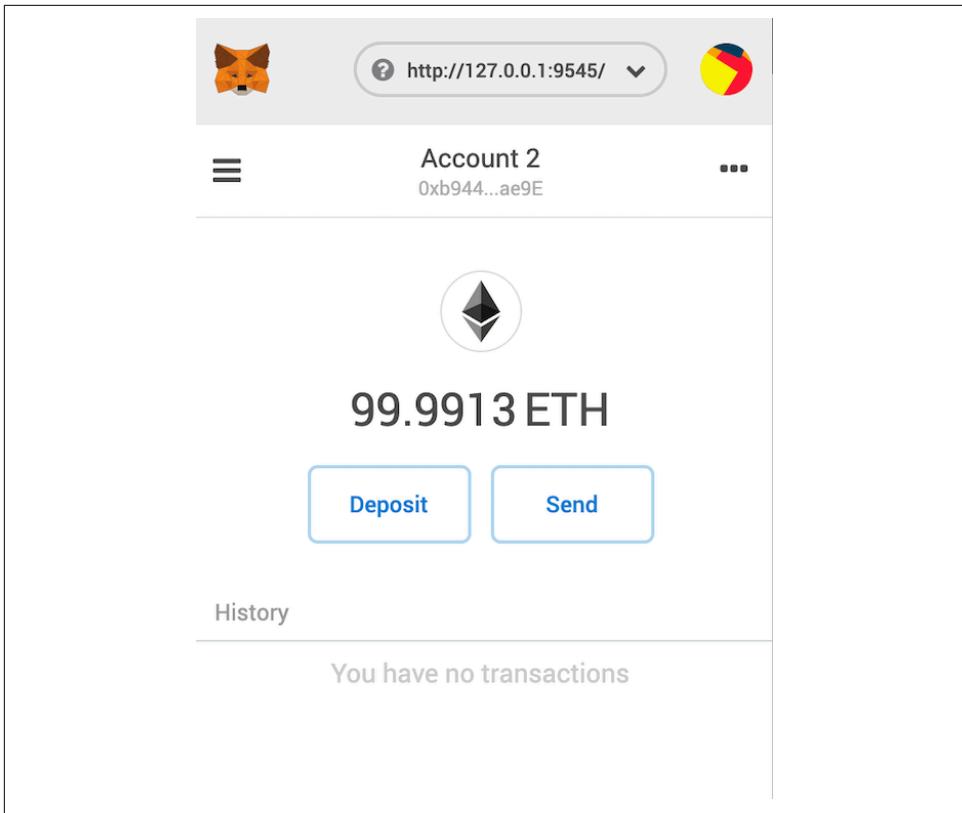


Figure 9-4. What the successful addition of your test account should look like

Now that we have Truffle and MetaMask up and running, we'll get the React application running.

Change to the `src` directory inside of our Greeter repository and install the dependencies:

```
cd client // this is the cd command that lets you move into directories
npm i
```

After everything is installed, we can start the frontend of your application so we can actually see the app in action:

```
npm run start
```

Now you'll be able to see your DApp in your browser if you navigate to `http://localhost:3000/`. It should look similar to [Figure 9-5](#).

You should see MetaMask pop up asking you to accept a connection request. See [Figure 9-6](#).

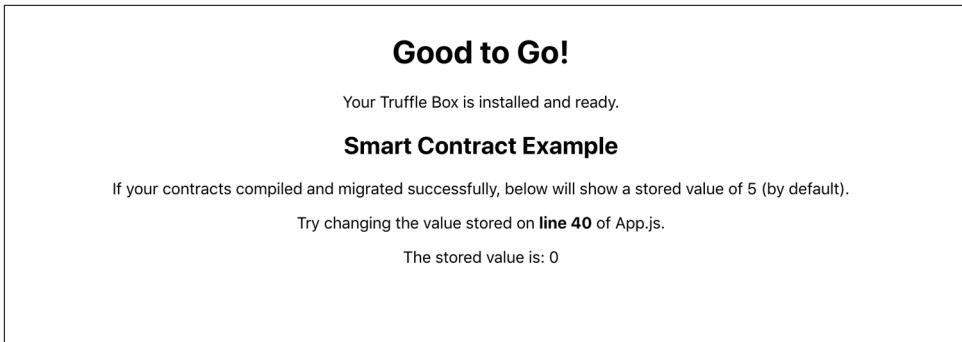


Figure 9-5. Congratulations! If your app is running properly, it should look like this.

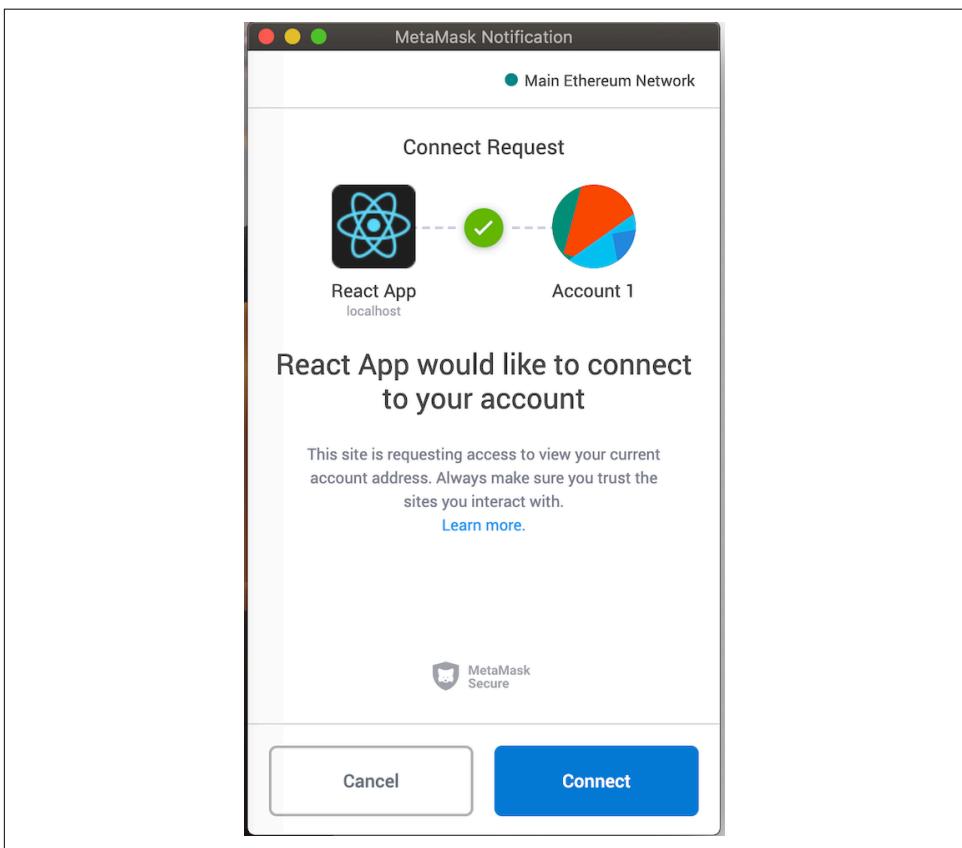


Figure 9-6. Accept the React Box application setup

Now that we have our basic application up and running, let's import our contracts so we can use the Greeter contract.

Importing our Greeter Contracts

Currently, we're using the default contracts with React Box. We'll need to import our Greeter contracts written previously so we can interact with them. Go back to our Greeter contract repository, and copy the *Greeter.sol* file. In the *contracts* file in our Greeter repository we just created, create a new file:

```
touch contracts/Greeter.sol
```

Open up the Greeter repository in your text editor and paste the Greeter contract into that file.

Create a new file for the migration like we did previously:

```
touch migrations/3_deploy_greeter.js
```

We'll now run our `compile` and `migrate` commands again so we can be sure we're interacting with the new Greeter contract we've imported. If we don't do this, our Truffle Develop terminal will still be interacting with the standard React Truffle Box contracts.

After our contracts have successfully compiled and the migrations are finished, let's open up our `client` directory in our application and get started on creating a UI for our own contracts.

Open up the `App.js` file. We're going to modify some of the code to interact with our contracts. Start by removing all the standard Truffle boilerplate in the `return` function. Your code should now look like this:

```
return (
  <div className="App">
    </div>
);
```

Let's add our own Greeter contract header now:

```
return (
  <div className="App">
    <h1>Greeter</h1>
    </div>
);
```

Next, instead of importing the boilerplate contract `SimpleStorage`, let's import our Greeter contract so we can start calling methods from the Greeter contracts that we wrote earlier:

```
import GreeterContract from "./contracts/Greeter.json";
```

Still in the `App.js` file, and in the `componentDidMount` function, we'll replace the following line with our Greeter contract network ID:

```
const deployedNetwork = SimpleStorageContract.networks[networkId];
```

This will allow us to actually connect to our Greeter contract:

```
const deployedNetwork = GreeterContract.networks[networkId];
```

In the same `componentDidMount` function, replace the `SimpleStorage` contract with our own `GreeterContract` reference:

```
const instance = new web3.eth.Contract(  
  GreeterContract.abi,  
  deployedNetwork && deployedNetwork.address,  
);
```

Now let's add our Greeter contract so our users can interact with it through our React frontend.

Adding Our Greeter Contract Functionality

We're going to need to make a few more modifications to the `App.js` boilerplate code before we can work with our Greeter contract.

In our `App.js` file, we'll need to use Web3 so we can connect to an Ethereum network. In our `componentDidMount` method, we'll access Web3 and store it in our React state. We'll stick with most of the boilerplate code, with a few additions.

To set up our Greeter contract to work with our frontend, we'll first need to get rid of some of the boilerplate code from Truffle Box. We'll first need to change the state of our `App.js` component to match our contracts code:

```
state = { storageValue: 0, web3: null, accounts: null, contract: null };
```

Where it says `storageValue: 0`, we'll replace that with a new state to store our return from our greeting function. Now, instead of `storageValue: 0`, you can replace that with `greeting: ''` as shown here. This will use the React state to display our greeting to the user:

```
state = { greeting: '', web3: null, accounts: null, contract: null };
```

Next, we'll need to replace the current `runExample` function code with our own. Remove all the old code from Truffle Box, except for the following code:

```
runExample = async () => {  
  const { accounts, contract } = this.state;  
  
};
```

Now, let's add our own code, which we'll call our Greeter contract, and set our React state:

```
const response = await contract.methods.greet().call()  
  
this.setState({ greeting: response });
```

We'll want to call our Greeter contract method right after the first line in `runExample`. That will store the response from our `greet()` method in a `const`.

After we've gotten our response, we're setting our React state with the response, as you can see with `greeting: response`.

Next, we need to return our new React state from our render function! Replace the current render function with the following code:

```
render() {
  if (!this.state.web3) {
    return <div>Loading Web3, accounts, and contract...</div>;
  }
  return (
    <div className="App">
      <h1>Greeter</h1>

      {this.state.greeting}
    </div>
  );
}
```

We're accessing our React state and now rendering it in our page. You should now see "Hello World" on `localhost:3000` ([Figure 9-7](#)).

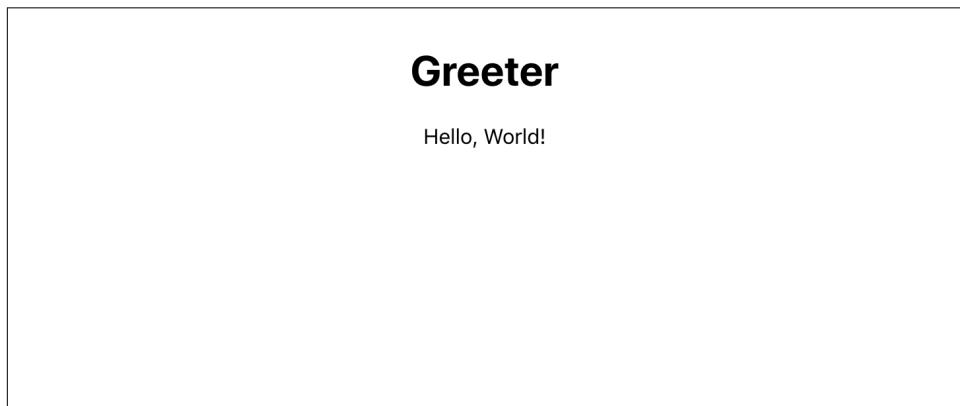


Figure 9-7. The home page of your application should be displaying "Hello World!"

Congratulations! You've just displayed your first contract response on a React application! But what if we want to display our greeting? In the next section, we'll update our greeting in the UI through a user interaction.

Setting Our Greeting Through React

Our next step will be setting a new greeting using React. Right now, our contract is only returning the default greeting. We'll use the `setGreeting` function in our `Greeter.sol` file and set our own message.

Let's start with creating a React text input so we can write our own message to set. Here is a simple form that we'll add right after `{this.state.greeting}` in our render function:

```
<form>
  <label>
    New Greeting:
    <input type="text" value={this.state.greeting}
           onChange={this.handleGreetingChange} />
  </label>
</form>

<button onClick={this.formSubmitHandler}> Submit </button>
```

After your app rerenders, you should now see your new input on the screen, as illustrated in [Figure 9-8](#).

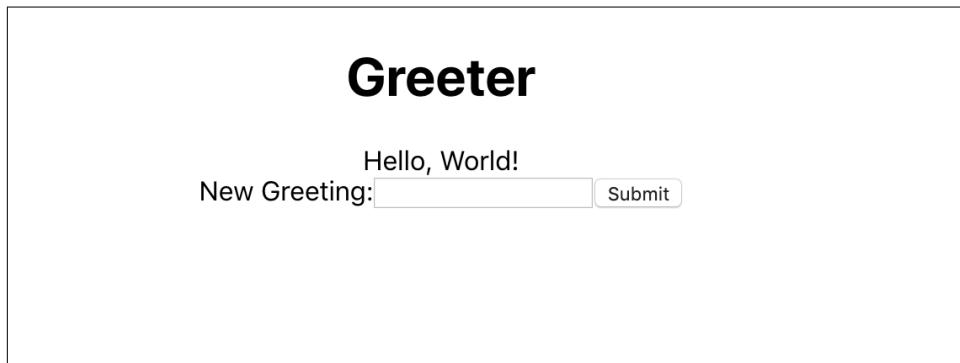


Figure 9-8. Your application can now interact with users through an input

If you type something in right now, nothing happens! Let's fix that.

First, create the empty skeleton functions for our `onSubmit` and `onChange` actions:

```
formSubmitHandler = () => {
  // do awesome Web3 stuff here!
}

handleGreetingChange = (e) => {
  // update our react state here
}
```

Ultimately, we're going to use Web3 in our `formSubmitHandler` function to connect to our contract and update our greeting. But let's first start with getting the data from our user into our `handleGreetingChange` function.

We'll need to update our `onChange` function in our input form, as shown here. This will allow us to access the actual input from the user in our function:

```
<input type="text" value={this.state.greeting}  
      onChange={e => this.handleGreetingChange(e)} />
```

In the `handleGreetingChange` function, we need to now grab the data from our input when the function is invoked. Add the following code to your `handleGreetingChange` function:

```
handleGreetingChange = (e) => {  
  const inputVal = e.target.value  
  this.setState({ greeting: inputVal })  
}
```

This function will now grab the input from the user and set the state. We are just updating the state, however, and not yet sending it to the blockchain. If you refresh the page right now, you'll see the data isn't stored. Let's work on that next.

In our `formSubmitHandler` function, we need to first grab the state for our current greeting input so we know what information we want to send to the blockchain.

```
formSubmitHandler = async () => {  
  const { accounts, contract, greeting } = this.state  
}
```

We'll just need to add one last thing now to get our data to the blockchain. We'll use Web3 to interact with our `setGreeting` method from our contract:

```
formSubmitHandler = async () => {  
  const { accounts, contract, greeting } = this.state;  
  const updatedGreeting = await contract.methods.setGreeting(greeting)  
}
```

Now if you type something into the input form, and click submit, which calls our submit action, our data is actually saved on the blockchain. A transaction permanently exists on the blockchain with our new greeter input. If we refresh our page again, we can see our greeter frontend has changed.

Summary

In this chapter, we learned about React and how it's the most popular framework to use with Ethereum and smart contract development. Most of the tutorials online and answers on Stack Overflow for frontend development with Truffle use React.

Interacting with our smart contract was made easy by using React Truffle Box for our frontend application. And with React, we created a few simple actions and inputs that let us set our greeter message in the blockchain.

We also learned more about Web3 and MetaMask and how we can actually use these tools to connect to the blockchain and our smart contracts.

In the next chapter, we'll use all the information we learned to create a more complicated frontend for our Fundraiser contract.

Our Larger DApp

Now that we've finished the contracts for our large-scale application, let's create the client side so we can have users interact with our contracts in the browser. After we create the frontend for our application, our users will be able to create their own fundraisers and see them show up on our home page. Our users will also be able to view more information about an individual fundraiser and donate to it. We'll be using the Fundraiser contracts we wrote in the previous chapters.

Let's get started on the development of our application.

Starting with React Truffle Box

We're going to use React Truffle Box once more to generate frontend code for our Web3 application to get it running quickly and interacting with Web3. Let's start by making a new directory for our fundraiser application and creating a fresh Truffle React Box application inside it.

The first thing we will need to do is create the new empty repository for our Fundraiser application. Next, we'll change into that directory and unbox the React Truffle Box:

```
mkdir fundraiser
cd fundraiser
truffle unbox react
```

Just like in [Chapter 8](#), this will quickly create scaffolding of a React application that is set up to interact with our fundraiser smart contracts and Web3. After the React Truffle Box has finished unboxing, let's import all of our contracts and migrations.

We'll first need to delete the `2_deploy_contracts.js` file that has the migration for the SimpleStorage contract because we will no longer need it.

Next, you'll need to create new files for our contracts and the migration. First, create all the empty files for our contracts and our migration:

```
touch contracts/Factory.sol  
touch contracts/Fundraiser.sol  
touch migrations/2_factory_contract_migrations.js
```

After you've created the empty files, import all the Solidity code we wrote for our Fundraiser contracts and migration into the proper files.



Make sure you create all of these in the outermost contracts and migrations folders and that you are not in the *client* directory.

We'll also need to install OpenZeppelin because we are using the Ownable contract:

```
npm install @openzeppelin/contracts
```

And in the *Fundraiser.sol* file, change the import statement to use the `node_module` we just installed from OpenZeppelin:

```
import '../client/node_modules/@openzeppelin/contracts/ownership/Ownable.sol';
```

After we've added the contracts, let's get our Truffle development environment up and running again. In the fundraiser repository, let's run the commands to compile and migrate our Fundraiser contracts so we can interact with them from our frontend application:

```
truffle develop
```

After that command has finished running, we'll run the compile and migrate in that same terminal window:

```
compile  
migrate
```



If you ever run into migration issues and get stuck on the solution, try running `migrate --reset`.

If your contracts have migrated successfully, they should look like this output:

```
Starting migrations...
=====
> Network name:    'develop'
> Network id:      5777
> Block gas limit: 0x6691b7
```

1_initial_migration.js

```
Replacing 'Migrations'  
-----  
> transaction hash: 0xf04ee2a0c62330e7a051148d4660de6441abd817caf6...  
> Blocks: 0 Seconds: 0  
> contract address: 0x6Af651D4c6E9f32a627381BFC771a82C882B1E8C  
> block number: 1  
> block timestamp: 1566526994  
> account: 0xb94454C83ff541c82391b4331e75d2E2b16bae9E  
> balance: 99.99477342  
> gas used: 261329  
> gas price: 20 gwei  
> value sent: 0 ETH  
> total cost: 0.00522658 ETH  
  
> Saving migration to chain.  
> Saving artifacts  
-----  
> Total cost: 0.00522658 ETH
```

2_factory_contract_migrations.js

```
Replacing 'Factory'  
-----  
> transaction hash: 0xebc5a26bbe12f52b809d9144a839befd0eebd877f7845...  
> Blocks: 0 Seconds: 0  
> contract address: 0xB7780C9AD3ef38bb4C8B48fab37Ef176603E7787  
> block number: 3  
> block timestamp: 1566526995  
> account: 0xb94454C83ff541c82391b4331e75d2E2b16bae9E  
> balance: 99.95063086  
> gas used: 2165105  
> gas price: 20 gwei  
> value sent: 0 ETH  
> total cost: 0.0433021 ETH  
  
> Saving migration to chain.  
> Saving artifacts  
-----  
> Total cost: 0.0433021 ETH
```

Summary

```
> Total deployments: 2  
> Final cost: 0.04852868 ETH
```

Great! Now that we have our Fundraiser contracts compiled, migrated, and deployed successfully, let's get started on running the frontend of our application so users can interact with it and create a fundraiser.

Fundraiser Setup

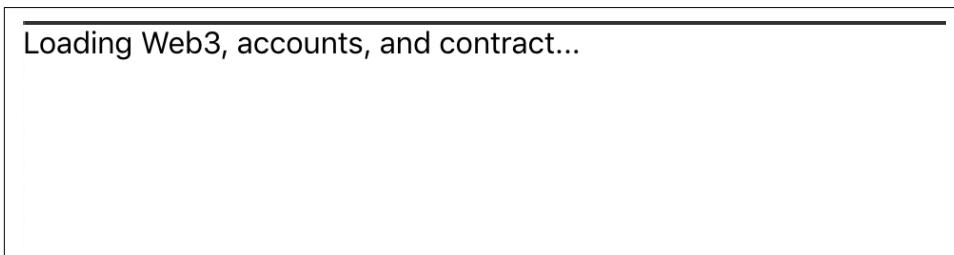
Let's work on getting our React Truffle Box up and running so we can interact with our application. We'll first need to `cd` into the `src` file of our React application and install all of the dependencies that the React Truffle Box provided for us. After that, we'll need to start up the frontend of our application just like we did in the previous chapter:

```
cd client  
npm i
```

After the modules have finished installing, start up the server:

```
npm start
```

If everything has installed properly and started successfully, we can now navigate to `localhost:3000` to see our Web3 application. See [Figure 10-1](#).



Loading Web3, accounts, and contract...

Figure 10-1. The default oops screen because we haven't switched our network quite yet

Oops! We forgot to switch our network to match our new application. We'll need to be on the same network so our app can run properly. Let's first go into MetaMask and switch the network at the top to be Localhost 8545 (see [Figure 10-2](#)). Localhost 8545 should be in the default list. If not, you can refer back to the previous chapter and add it.

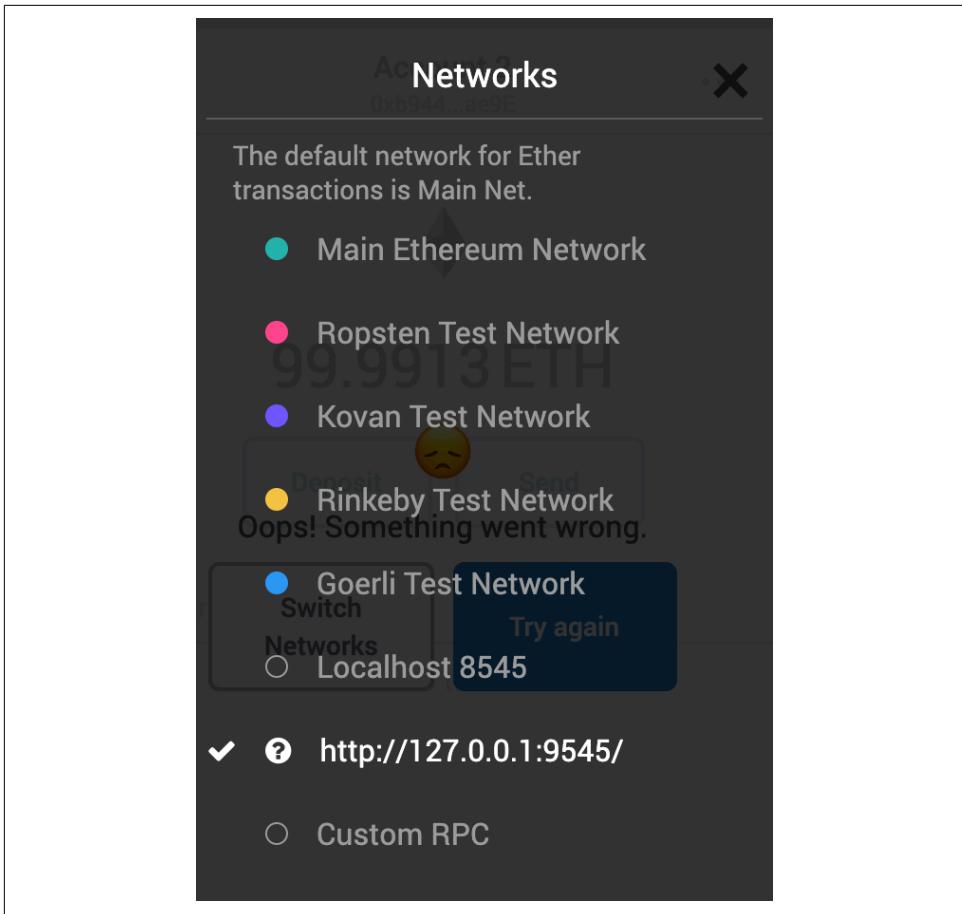


Figure 10-2. Switching our network will fix our error

Now, refresh the screen and you should see the simple storage UI we saw before. Verify that your page looks like [Figure 10-3](#).

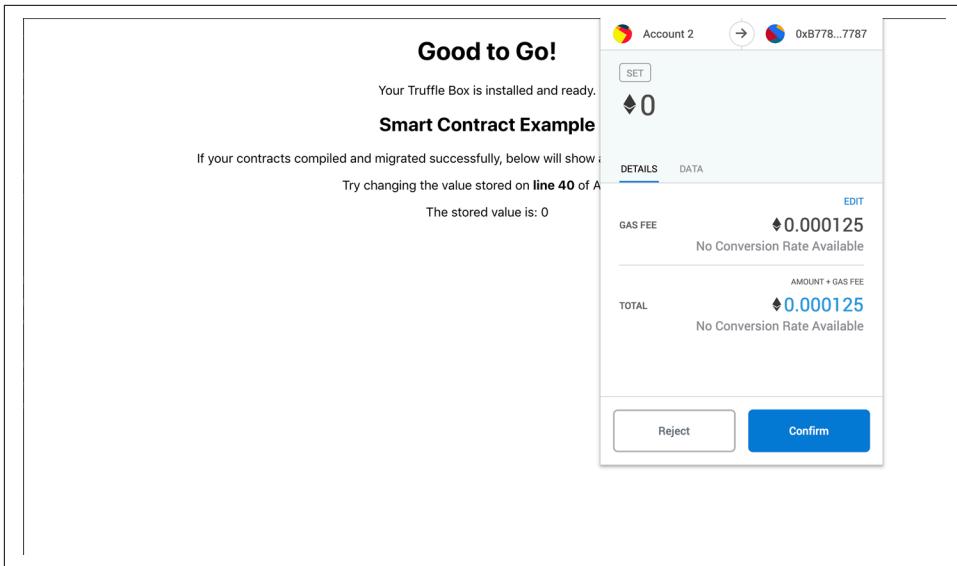


Figure 10-3. This is the default frontend for React Truffle Box

With our app running, let's start in our *App.js* file and rip out some of the boilerplate code to prepare our frontend to interact with the fundraiser.

First, navigate to the *client/src* directory and open the *App.js* file. We'll need to remove the old React code and replace it with our own React code:

```
import React, { useState, useEffect } from "react";
import FactoryContract from "./contracts/Factory.json";
import getWeb3 from "./utils/getWeb3";
import "./App.css";

const App = () => {
  const [state, setState] =
    useState({web3: null, accounts: null, contract: null});
  const [storageValue, setStorageValue] = useState(0);

  useEffect(() => {
    const init = async() => {
      try {
        const web3 = await getWeb3();
        const accounts = await web3.eth.getAccounts();
        const networkId = await web3.eth.net.getId();
        const deployedNetwork = FactoryContract.networks[networkId];
        const instance = new web3.eth.Contract(
          FactoryContract.abi,
          deployedNetwork && deployedNetwork.address,
        );
        setState({web3, accounts, contract: instance});
      }
    }
  }, []);

  return (
    <div>
      <h1>Smart Contract Fundraiser</h1>
      <p>Total Raised: ${storageValue}</p>
      <button>Contribute</button>
    </div>
  );
}

export default App;
```

```

    } catch(error) {
      alert(
        `Failed to load web3, accounts, or contract.
        Check console for details.`,
      )
      console.error(error);
    }
  }
  init();
}, []);
}

const runExample = async () => {
  const { accounts, contract } = state;
}

};

return(
  <div>
    <h1>Fundraiser</h1>
  </div>
);
}

export default App;

```

This will let us set up the app to import our contracts. Let's start with importing our Fundraiser contract, as shown in the preceding code.

Now that we've imported our contracts, let's work on getting a frontend for our application and displaying our list of fundraisers!

React Routing

In order to navigate between different pages of our application with a home page that connects to a new fundraiser page and a donation receipt page, we'll need to set up routes in our React application. We'll use `react-router-dom` to let a user see different pages, depending on what they've selected from the navbar.

We'll start by installing the npm package for `react-router-dom`:

```
npm install --save react-router-dom
```

After you've installed the npm package and restarted your frontend server, import the necessary files from `react-router-dom` in your `App.js` file, as shown here:

```
import { BrowserRouter as Router, Route, NavLink } from "react-router-dom"
```

We'll also need to import our two new components for the home page and the new fundraiser to use in our routes:

```
import NewFundraiser from './NewFundraiser'
import Home from './Home'
```

Next, let's replace the render function with the following code. This will enable us to use the navbar provided from Material UI to navigate to different pages throughout our app:

```
<Router>
  <div>
    <nav>
      <ul>
        <li>
          <NavLink to="/">Home</NavLink>
        </li>
        <li>
          <NavLink to="/new/">New</NavLink>
        </li>
      </ul>
    </nav>
    <Route path="/" exact component={Home} />
    <Route path="/new/" component={NewFundraiser} />
  </div>
</Router>
```

If you try to run the code right now, you'll get an error stating that our components do not exist. We'll need to create a few new components first to fix this. In your *src* directory, you'll need to create two new components.

Create two new files: *Home.js* and *NewFundraiser.js*. We'll use the home page component as our main landing page for our application and the New Fundraiser page to create a new fundraiser in our application:

```
touch Home.js
touch NewFundraiser.js
```

Later on, we'll also create a receipts route. But for now, let's work on creating the Home view. We'll leave it as a simple render function for now until we need to build out the home page view. Inside your *Home.js* file, use the following placeholder code and save that file:

```
import React, { useState, useEffect } from "react";

const Home = () => {
  useEffect(() => {
    , []);
  }

  return (
    <div><h2>Home</h2></div>
  )
}

export default Home;
```

Similarly, in the `NewFundraiser.js` file, add this code:

```
import React, { useState, useEffect } from "react";

const NewFundraiser = () => {
  useEffect(() => {
    }, []);

  return (
    <div><h2>Create a New Fundraiser</h2></div>
  )
}

export default NewFundraiser;
```

That should fix our error. But one last thing we'll need to do is modify the code in our `index.js` file so we can render routes properly. We'll use `BrowserRouter` from `react-router-dom` to wrap our application in a Router:

```
import React from 'react';
import ReactDOM from 'react-dom';

import { BrowserRouter } from 'react-router-dom'
import App from './App';

ReactDOM.render((
  <BrowserRouter>
    <App />
  </BrowserRouter>
), document.getElementById('root'))
```

We should be good to go! The preceding code will show you a different page each time you click a nav link. Now if you reload `localhost:3000`, you should see the page illustrated in [Figure 10-4](#).

And if you click New, it will navigate you to the second view (see [Figure 10-5](#)).



Figure 10-4. Our unstyled navbar view

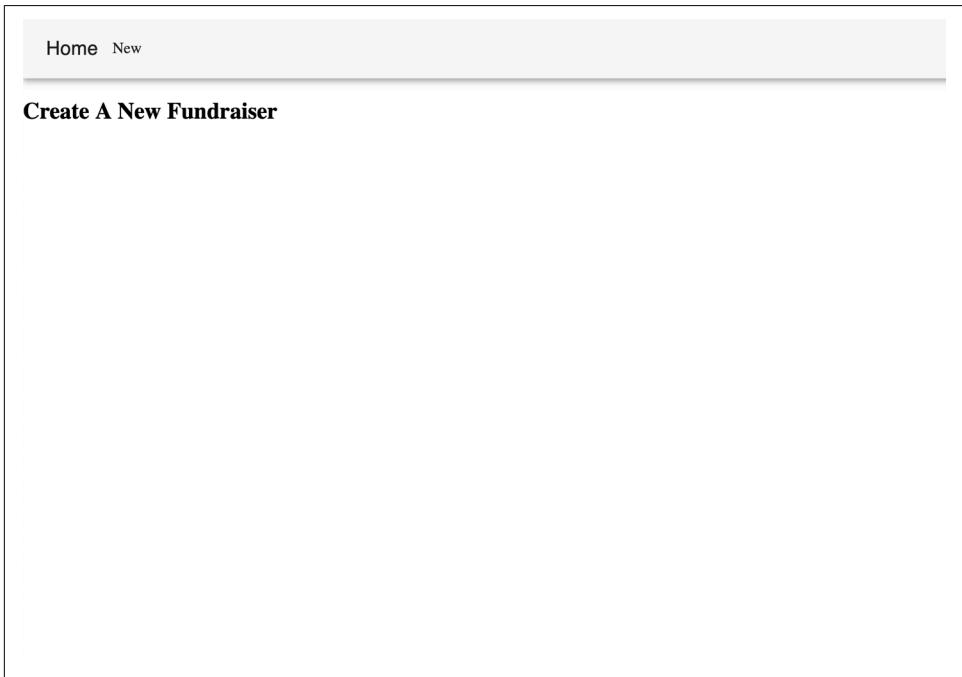


Figure 10-5. Our fundraiser view (empty for now, but we'll fix that soon)

Try it out and make sure it's working. Let's work on styling this next.

React and Material UI

Our main objective in this book is to learn blockchain development, not frontend development, so we're going to use a React UI framework to ease the time we spend on frontend development for our Web3 application. We're going to use Material UI, which is a UI framework that provides React components such as popovers, buttons, tooltips, navigation, and more. We'll also be using Material UI's default styling with some minor modifications. You can find out more about Material UI [here](#).

Let's begin by installing Material UI into our application.

Run the `install` command in your `src` directory to install `react-bootstrap` in our fundraiser application as follows:

```
npm install @material-ui/core --save
```

Note that you'll have to stop the terminal window where your frontend application is running and restart it after installing Material. You won't see any changes on the frontend quite yet. Let's add some! First, let's start with creating better navigation for our application so the user can click through the different sections of our fundraiser application to create a new fundraiser, donate to fundraisers, or get a donation receipt.

From the Material UI site, let's look at the app bar page [here](#).

Let's start with the simple app bar. First, we'll need to add all the imports to the top of our `App.js` file. This will allow us to use the Material UI components we need:

```
import { makeStyles } from '@material-ui/core/styles';
import AppBar from '@material-ui/core/AppBar';
import Toolbar from '@material-ui/core/Toolbar';
import Typography from '@material-ui/core/Typography';
```

Now that we have the imports, we need to add in the styles so our fundraiser app looks good. You'll need to add in these styles after your `useEffect` function:

```
const useStyles = makeStyles({
  root: {
    flexGrow: 1,
  },
});

const classes = useStyles();
```

After you've imported the correct files and added the style preceding code, let's replace the actual render code so we can see our new navbar:

```
<Router>
  <div>
    <AppBar position="static" color="default">
      <Toolbar>
        <Typography variant="h6" color="inherit">
          <NavLink className="nav-link" to="/">Home</NavLink>
        </Typography>
        <NavLink className="nav-link" to="/new/">New Fundraiser</NavLink>
      </Toolbar>
    </AppBar>

    <Route path="/" exact component={Index} />
    <Route path="/new/" component={NewFundraiser} />
  </div>
</Router>
```

Let's quickly add a little styling to the `NavLink` component so it looks more professional. In your `App.css` file, add these to the file:

```
body {
  margin: 0 !important;
}

.nav-link {
  color: inherit;
  text-decoration: none;
  margin-right: 15px;
}

.nav-link:hover,
.nav-link:active,
.nav-link:visited {
  color: black;
  text-decoration: none;
}
```

If everything worked, you should now see the new app bar and an empty application. See [Figure 10-6](#) for reference.

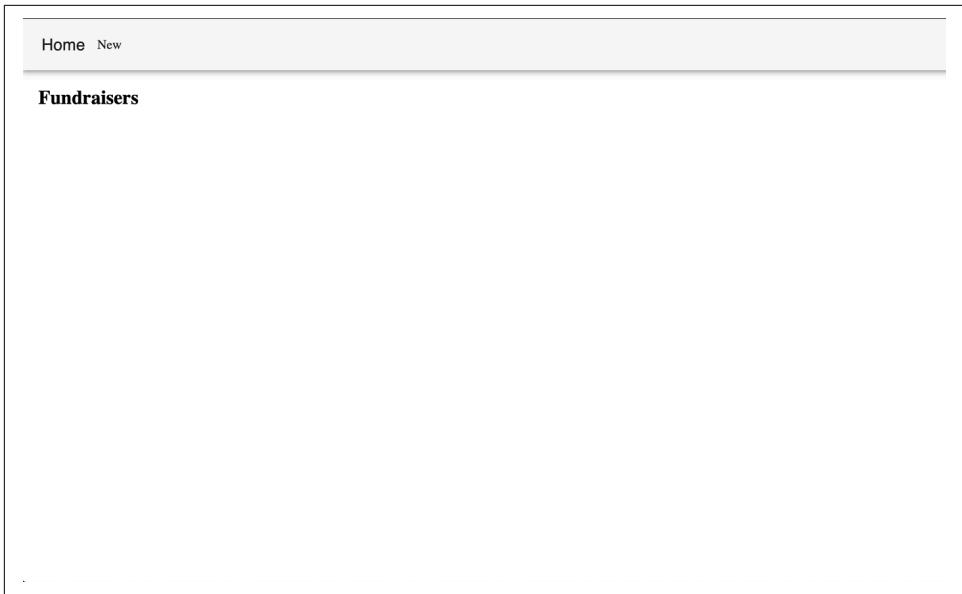


Figure 10-6. Our navbar is now simple but set up

Now that we have navigation and routes set up, let's work on our first page view to allow our users to create a new fundraiser.

Creating Our New Fundraiser Page View

In order to create a new fundraiser, we want to display a simple form to our user that will take the user inputs for the fundraiser name, website, Ethereum address, etc. Following our wireframes, we'll want to use forms and create a submit button so our user can create a new fundraiser of their own. Since we created the skeleton code and file for our New Fundraiser view earlier, let's open that file up again. It should exist in `src/NewFundraiser.js`. The first thing we'll want to do is create the set of inputs to display to our users.

We'll start by using Material UI's `text field` component. Add an `import` statement to your `NewFundraiser.js` file right after your latest `import` statement:

```
import { makeStyles } from '@material-ui/core/styles';
import TextField from '@material-ui/core/TextField';

const useStyles = makeStyles(theme => ({
  container: {
    display: 'flex',
    flexWrap: 'wrap',
  },
  textField: {
    marginLeft: theme.spacing(1),
```

```

    marginRight: theme.spacing(1),
},
dense: {
  marginTop: theme.spacing(2),
},
menu: {
  width: 200,
},
});

```

We'll also need to add a new state item for each piece of data we'll pull from our contract about our fundraiser. Create each empty useState as shown here:

```

const [ name, setFundraiserName ] = useState(null)
const [ website, setFundraiserWebsite ] = useState(null)
const [ description, setFundraiserDescription ] = useState(null)
const [ image, setImage ] = useState(null)
const [ address, setAddress ] = useState(null)
const [ custodian, setCustodian ] = useState(null)
const [ contract, setContract ] = useState(null)
const [ accounts, setAccounts ] = useState(null)

```

Now in your render function, add this right after your header. We'll also add an onChange function to allow the user to type in input, see it actually change in the page, and save to our React state:

```

<label>Name</label>
<TextField
  id="outlined-bare"
  className={classes.textField}
  placeholder="Fundraiser Name"
  margin="normal"
  onChange={(e) => setFundraiserName(e.target.value)}
  variant="outlined"
  inputProps={{ 'aria-label': 'bare' }}
/>

<label>Website</label>
<TextField
  id="outlined-bare"
  className={classes.textField}
  placeholder="Fundraiser Website"
  margin="normal"
  onChange={(e) => setFundraiserWebsite(e.target.value)}
  variant="outlined"
  inputProps={{ 'aria-label': 'bare' }}
/>

<label>Description</label>
<TextField
  id="outlined-bare"
  className={classes.textField}
  placeholder="Fundraiser Description"

```

```

margin="normal"
onChange={(e) => setFundraiserDescription(e.target.value)}
variant="outlined"
inputProps={{ 'aria-label': 'bare' }}
/>>

<label>Image</label>
<TextField
  id="outlined-bare"
  className={classes.textField}
  placeholder="Fundraiser Image"
  margin="normal"
  onChange={(e) => setImage(e.target.value)}
  variant="outlined"
  inputProps={{ 'aria-label': 'bare' }}
/>

<label>Address</label>
<TextField
  id="outlined-bare"
  className={classes.textField}
  placeholder="Fundraiser Ethereum Address"
  margin="normal"
  onChange={(e) => setAddress(e.target.value)}
  variant="outlined"
  inputProps={{ 'aria-label': 'bare' }}
/>

<label>Custodian</label>
<TextField
  id="outlined-bare"
  className={classes.textField}
  placeholder="Fundraiser Custodian"
  margin="normal"
  onChange={(e) => setCustodian(e.target.value)}
  variant="outlined"
  inputProps={{ 'aria-label': 'bare' }}
/>

```

Let's also add these styles to our *App.css* file so we can make our inputs a little nicer:

```

.MuiTextField-root {
  display: block !important;
}

.MuiInputBase-root {
  width: 300px !important;
  margin-left: 3px;
}

```

The last thing we'll need to do is create a button to submit the new data to our Fundraiser contract and create the contract.

First, import the button into our *NewFundraiser.js* file. Add this Material UI code so we can use the default styles:

```
import { makeStyles } from '@material-ui/core/styles';
import Button from '@material-ui/core/Button';

const useStyles = makeStyles(theme => ({
  button: {
    margin: theme.spacing(1),
  },
  input: {
    display: 'none',
  },
}));
```

And then right after our `useEffect` function, add the following `useStyles` code:

```
const classes = useStyles();
```

After we've imported the button and set up the default styles, add the button to render the following:

```
<Button
  onClick={handleSubmit}
  variant="contained"
  className={classes.button}>
  Submit
</Button>
```

Now, let's move on to getting Web3 so we can access our Factory contract. In our *NewFundraiser.js* file, update the `useEffect` function to use the Web3 code. The following code will create a new instance of our contract and set the state of Web3, our contract, and our current account:

```
useEffect(() => {
  const init = async() => {
    try {
      const web3 = await getWeb3();
      const networkId = await web3.eth.net.getId();
      const deployedNetwork = FactoryContract.networks[networkId];
      const accounts = await web3.eth.getAccounts();
      const instance = new web3.eth.Contract(
        FactoryContract.abi,
        deployedNetwork && deployedNetwork.address,
      );

      setWeb3(web3)
      setContract(instance)
      setAccounts(accounts)

    } catch(error) {
      alert(`Failed to load web3, accounts, or contract. Check console for details.`)
    }
  }
});
```

```
    );
    console.error(error);
}
init();
}, []);
```

Next, we need to import our contract and point to our contract that has been deployed locally in our *NewFundraiser.js* file. Add this code right after all your import statements:

```
import getWeb3 from "./utils/getWeb3";
import FactoryContract from "./contracts/Factory.json";
```

Finally, let's call our contracts `createFundraiser` function and pass in the parameters from our state, including name, website, description, etc. After that has been called and created, we'll just show a simple alert to the user:

```
const handleSubmit = async () => {
  await contract.methods.createFundraiser(
    name,
    website,
    imageURL,
    description,
    beneficiary,
    custodian
  ).send({ from: accounts[0] })

  alert('Successfully created fundraiser')
}
```

Let's create a few new fundraisers. Next, we'll need a way to display our new fundraisers to the user so they can donate to them and view more information about each individual fundraiser.

Displaying the Current Fundraisers List

Now that we can create a new fundraiser, let's add a frontend so we can display the fundraiser we've created on the page and all of the other fundraisers created as well. After we've created the UI for displaying a fundraiser, we'll also use a modal so the user can click an individual fundraiser and donate through there. But let's start with displaying the list of fundraisers.

We're going to use Material UI's `cards` component to display each individual fundraiser. The first thing we'll need to do is to call our Factory contract using Web3 and return the data about a fundraiser.

To do this, we'll need to again import Web3 and our `FactoryContract` JSON at the top of our import file. We'll also need to add the `web3`, `contracts`, and `accounts` to

our state. Last, we'll need to update our `useEffect` function to use the same code as in the `NewFundraiser.js` file.

We'll make one modification to this `useEffect` function. In the try/catch statement, we'll add a function called `getFunds()` in the try statement. We'll get our fundraisers in that function and add it in just a moment.

```
useEffect(() => {
  init()
}, []);

const init = async () => {
  try {
    const networkId = await web3.eth.net.getId();
    const deployedNetwork = FactoryContract.networks[networkId];
    const accounts = await web3.eth.getAccounts();
    const instance = new web3.eth.Contract(
      FactoryContract.abi,
      deployedNetwork && deployedNetwork.address,
    );
    setContract(instance)
    setAccounts(accounts)

    // We'll add our fundraisers() function call here
  }
  catch(error) {
    alert(
      `Failed to load web3, accounts, or contract. Check console for details.`,
    );
    console.error(error);
  }
}
```

After we set the state, we'll call our contract and return the addresses of each contract in the Factory contract we created. We'll store the funds in our state. Right after the declaration of our `Home` component, add in one more `setState`:

```
const [ funds, setFunds ] = useState(null)
```

Next, add the following code to our `useEffect` code where we left the placeholder previously. The following code will call our contract method to get a list of the fundraisers:

```
const funds = await instance.methods.fundraisers(10, 0).call()
setFunds(funds)
```

Our `fundraiser()` method will return the following output. We'll use this later on to set the instance of a contract and pull data about each fundraiser.

```
(2) ["0x500116801EAE512672648c80C7a5DB6AdFb2b283",
"0xB705bb647cf75580133540855d015f3Ba5D38C71"]
```

Now that we've received the data from our contract, we'll work on the frontend to display each Fundraiser card to the user. The first thing we'll need to do is create a new component that we'll use to display our Card component:

```
touch FundraiserCard.js
```

In the *FundraiserCard.js* file, we'll add the card code from Material UI. This will display a simple card to the user with some basic information about the fundraiser:

```
import React, { useEffect } from 'react';
import { makeStyles } from '@material-ui/core/styles';
import Card from '@material-ui/core/Card';
import CardActionArea from '@material-ui/core/CardActionArea';
import CardActions from '@material-ui/core/CardActions';
importCardContent from '@material-ui/core/CardContent';
import CardMedia from '@material-ui/core/CardMedia';
import Button from '@material-ui/core/Button';
import Typography from '@material-ui/core/Typography';
import FundraiserContract from "./contracts/Fundraiser.json";
import Web3 from 'web3'

const useStyles = makeStyles({
  card: {
    maxWidth: 450,
    height: 400
  },
  media: {
    height: 140,
  },
});

const FundraiserCard = (props) => {
  const classes = useStyles();
  const web3 = new Web3(new Web3.providers.HttpProvider('http://localhost:8545'))

  const [ contract, setContract ] = useState(null)
  const [ accounts, setAccounts ] = useState(null)
  const [ fundName, setFundname ] = useState(null)
  const [ description, setDescription ] = useState(null)
  const [ totalDonations, setTotalDonations ] = useState(null)
  const [ donationCount, setDonationCount ] = useState(null)
  const [ imageURL, setImageURL ] = useState(null)
  const [ url, setURL ] = useState(null)

  useEffect(() => {
    // we'll add in the Web3 call here
  }, []);

  return (
    <div className="fundraiser-card-content">
      <Card className={classes.card}>
        <CardActionArea>
```

```

        <CardMedia
          className={classes.media}
          image={props.fundraiser.image}
          title="Fundraiser Image"
        />
      <CardContent>
        <Typography gutterBottom variant="h5" component="h2">
          {fundName}
        </Typography>
        <Typography variant="body2" color="textSecondary" component="p">
          <p>{description}</p>
        </Typography>
      </CardContent>
    </CardActionArea>
    <CardActions>
      <Button size="small" color="primary">
        View More
      </Button>
    </CardActions>
  </Card>
</div>
)
}

export default FundraiserCard;

```

If you navigate to `localhost:3000`, you'll see that it's blank for now. Let's create a render function to go through each fundraiser and display them in a card.

In our render function that follows our main image and header, let's add a call to a function that will render each fundraiser:

```
{displayFundraisers()}
```

Next, let's create the function. We'll need to iterate through each fundraiser from our props and display that in a card. But first, we'll need to import the Fundraiser card we just created.

At the top of the `App.js` file, add the following `import` statement so we can use the Fundraiser card:

```
import FundraiserCard from './FundraiserCard'
```

Next, we'll iterate through the list of fundraisers and display the Card component for each one:

```

const displayFundraisers = () => {
  return funds.map((fundraiser) => {
    return (
      <FundraiserCard fundraiser={fundraiser} />
    )
  })
}

```

We'll also want to add some styling to the *Home.js* component so our cards will render correctly. In the *Home.js* file, add this container div:

```
<div className="main-container">  
  {displayFundraisers()}\n</div>
```

In the *App.css* file, add the following styling so our cards are rendered properly with CSS:

```
.main-container {  
  margin: 20px;  
}  
  
.fundraiser-card-container {  
  display: inline-flex;  
  width: 250px;  
  height: 250px;  
  margin: 20px;  
}
```

Next, we'll need to call our Fundraiser contract, get information about each fundraiser, and display it to the user. First, let's only call our `init` function if the fundraiser exists. Pass in the `fundraiser` as an argument to the `init` function so we can use it in the function to fetch the data we need:

```
useEffect(() => {  
  if (fundraiser) {  
    init(fundraiser)  
  }  
, [fundraiser]);  
  
const init = async (fundraiser) => {  
  try {  
    const fund = fundraiser  
    const networkId = await web3.eth.net.getId();  
    const deployedNetwork = FundraiserContract.networks[networkId];  
    const accounts = await web3.eth.getAccounts();  
    const instance = new web3.eth.Contract(  
      FundraiserContract.abi,  
      fund  
    );  
    setContract(instance)  
    setAccounts(accounts)  
  
    // Placeholder for getting information about each contract  
  }  
  catch(error) {  
    alert(`Failed to load web3, accounts, or contract. Check console for details.`);  
    console.error(error);  
  }  
};
```

```
    }  
}
```

The last thing we'll need to do is to call our Fundraiser contract and get information about the fundraiser. After we call our Fundraiser contract, we'll set it in our React state. After the `setAccounts` set state in the `useEffects` function, add the code shown here:

```
const name = await instance.methods.name().call()  
const description = await instance.methods.description().call()  
const totalDonations = await instance.methods.totalDonations().call()  
const imageURL = await instance.methods.imageURL().call()  
const url = await instance.methods.url().call()  
  
setFundname(name)  
setDescription(descriptions)  
setImageURL(imageURL)  
setTotalDonations(totalDonations)  
setURL(url)
```

Let's save that and visit our `localhost:3000/` page. If everything is working correctly, you should see the fundraisers displayed on the page, just like in [Figure 10-7](#).

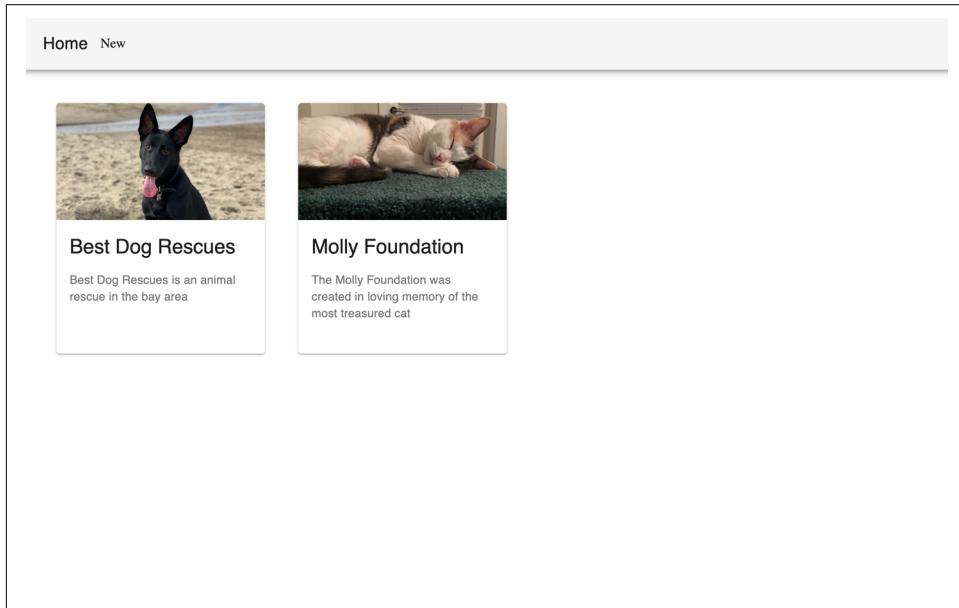


Figure 10-7. Our main fundraiser home page

Now that we have our list of fundraisers displayed on the page, we'll work on displaying more information to our users in the next chapter and adding our donation functionality.

Summary

In this chapter, we revisited how to set up a Truffle application with a frontend. We also learned how to set up the React Router so we could render different views to our users through a navbar component.

We worked on creating a new form and displaying it to our users so they can now interact with our Web3 application and create their own fundraiser. We learned about submitting that data to the blockchain and returning it as well.

We learned about Material UI, using it to quickly set up a React application, and using the components and styles from Material UI to quickly get our application up and running.

We also displayed the fundraisers from our application and rendered them on the page. In the next chapter, we'll work on displaying detailed information about the fundraisers and submitting a donation.

Finishing Our Fundraiser UI

In this chapter, we'll continue where we left off in [Chapter 10](#) working on our new Web3 application. We'll add more detailed information about each fundraiser where our user can click into a fundraiser to see more information, add donation receipts for each individual donation, and more.

We'll also learn how to accept ether through our application so our user can submit funds to a fundraiser and interact with our contract.

Our first steps will be to display more information about each fundraiser. That will allow our donors to more easily decide what fundraiser they want to donate to and help our fundraiser creators make a more compelling case.

Adding Detailed Information About Each Fundraiser

The next item on our agenda is to display more information about each fundraiser to our user. We'll use Material UI's modal to let the user click on an individual fundraiser and see the information display quickly in a pop-up. When the modal is open, they'll be able to click a donation amount and read more about the fundraiser. They'll also be able to quickly close the modal. But first, we need to create a button on each card that opens the right fundraiser.

Copy the following code and add it right after your latest `import` statement. This will first import the button we need from Material UI and also use the default styles provided by Material UI:

```
import Button from '@material-ui/core/Button';

const useStyles = makeStyles(theme => ({
  button: {
    margin: theme.spacing(1),
  },
})
```

```
    input: {
      display: 'none',
    },
  }));
}
```

And right before your `useEffect` function, add the following to use the default styles from Material UI:

```
const classes = useStyles();
```

The last thing we'll need to do is add our button with an action to open our modal. We'll leave the `handleOpen` function empty for now. In the `FundraiserCard.js` file, add this button to the `CardActions` section in the `Card`:

```
<CardActions>
  <Button
    onClick={handleOpen}
    variant="contained"
    className={classes.button}>
    View More
  </Button>
</CardActions>
```

And in the `FundraiserCard.js` file, add an `onClick` method to open the modal to an individual Fundraiser card:

```
<Card className={classes.card} onClick={handleOpen}>
```

Now, create the empty `handleOpen` function for now so we can fix our warning. We'll use this later to open up our dialog:

```
const handleOpen = () => {
  // we'll open the dialog here later on
}
```

Your Fundraiser component should now look like this:

```
const FundraiserCard = (props) => {
  const classes = useStyles();

  useEffect(() => {
    }, []);

  const handleOpen = () => {
  }

  return (
    <div>
      <Card className={classes.card} onClick={handleOpen}>
        <CardActionArea>
          <CardMedia
            className={classes.media}
            image={imageURL}
```

```

        title="Fundraiser Image"
    />
<CardContent>
    <Typography gutterBottom variant="h5" component="h2">
        {fundName}
    </Typography>
    <Typography variant="body2" color="textSecondary" component="p">
        <p>{description}</p>
        <p>Total Donations: {totalDonations}</p>
    </Typography>
</CardContent>
</CardActionArea>
<CardActions>
    <Button
        onClick={handleOpen}
        variant="contained"
        className={classes.button}>
        View More
    </Button>
</CardActions>
</Card>
</div>
)
}

```

Let's move on to our next step!

Implementing a Material UI Dialog

Now that we've finished our button setup, let's implement the [dialog](#) from Material UI next so we can use a popover in our application. The dialog is where we'll actually display all the additional information about a fundraiser to our user. When the user clicks the button we just created, the modal will open with the fundraiser.

Inside the *FundraiserCard.js* component, add the dialog code from Material UI.

First, add the `import` statements to the top of the file after your previous imports:

```

import Dialog from '@material-ui/core/Dialog';
import DialogActions from '@material-ui/core/DialogActions';
importDialogContent from '@material-ui/core/DialogContent';
importDialogContentText from '@material-ui/core/DialogContentText';
importDialogTitle from '@material-ui/core/DialogTitle';

```

Next, add in the dialog state and the functions we'll use to open the dialog:

```

// Add this code right below the useEffect method.
const [open, setOpen] = React.useState(false);

const handleOpen = () => {
  setOpen(true);
  // this will set our state to true and open the modal
};

const handleClose = () => {
  setOpen(false);
  // this will close the modal on click away and on button close
};

```

Finally, let's update the render to open the dialog. Add in the new dialog code right before our `<Card>` code in our `Fundraiser.js` file:

```

<div>
  <Dialog open={open} onClose={handleClose} aria-labelledby="form-dialog-title">
    <DialogTitle id="form-dialog-title">Donate to {fundName}</DialogTitle>
    <DialogContent>
      <DialogContentText>
        <img src={imageURL} width='200px' height='200px' />
        <p>{description}</p>
      </DialogContentText>
    </DialogContent>
    <DialogActions>
      <Button onClick={handleClose} color="primary">
        Cancel
      </Button>
    </DialogActions>
  </Dialog>

  <Card className={classes.card} onClick={handleOpen}>

```

Great! If you navigate to the home page of our application now, you'll see the fundraiser we created earlier in [Figure 11-1](#).



Best Dog Rescues

Best Dog Rescues is an animal rescue in the bay area

[VIEW MORE](#)

Figure 11-1. Our Fundraiser card

If you click View More, the dialog will open and we'll see the empty dialog we added in [Figure 11-2](#).

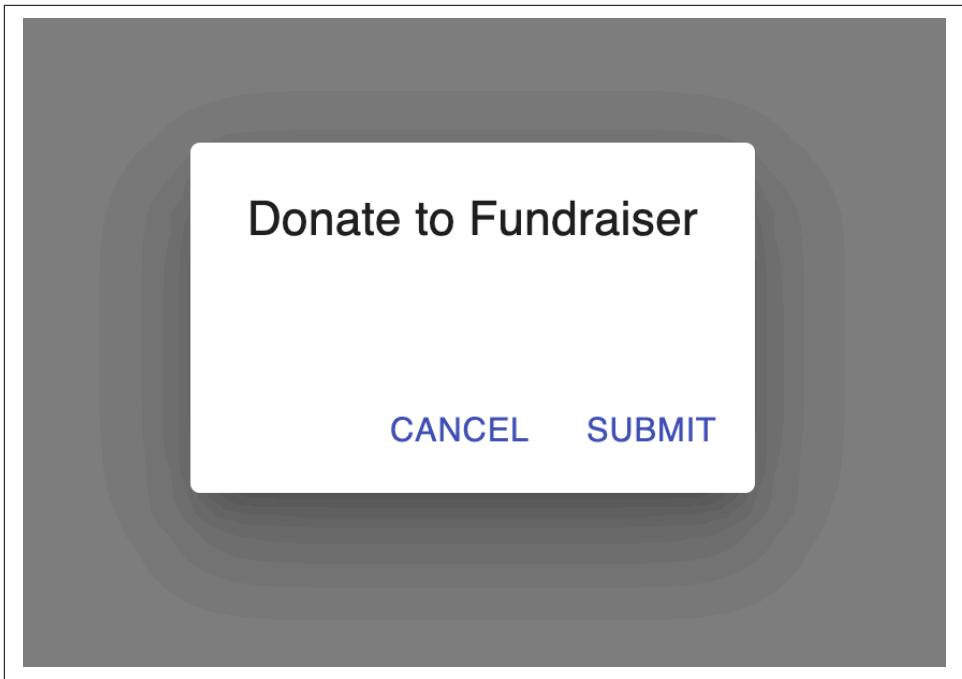


Figure 11-2. Example of our dialog thus far

In the next section, let's add in the functionality for donating in our new dialog from Material UI.

Accepting a Donation with React

Now that we've gotten most of our application UI completed, let's finish up our next feature: the ability to donate to a fundraiser in our Web3 application and see it update on the page.

We'll be using Material UI's forms again to create a simple input so the user can change how much they want to donate to a fund.

Import the form from Material UI into our *FundraiserCard.js* file:

```
import { makeStyles } from '@material-ui/core/styles';
import FilledInput from '@material-ui/core/FilledInput';
import FormControl from '@material-ui/core/FormControl';
import FormHelperText from '@material-ui/core/FormHelperText';
import Input from '@material-ui/core/Input';
import InputLabel from '@material-ui/core/InputLabel';
import OutlinedInput from '@material-ui/core/OutlinedInput';

const useStyles = makeStyles(theme => ({
```

```
container: {
  display: 'flex',
  flexWrap: 'wrap',
},
formControl: {
  margin: theme.spacing(1),
  display: 'table-cell'
},
card: {
  maxWidth: 450,
  height: 400
},
media: {
  height: 140,
},
paper: {
  position: 'absolute',
  width: 400,
  backgroundColor: theme.palette.background.paper,
  border: 'none',
  boxShadow: 'none',
  padding: 4,
},
}),
});
```

We'll also need to add `useState` to our React import in the first line of our `Fundraiser.js` file:

```
import React, { useEffect, useState } from 'react';
```

And we'll need to add an input for our donation:

```
const [donationAmount, setDonationAmount] = useState(null)
```

Now in your dialog, add the form shown here:

```
<FormControl className={classes.formControl}>
  <Input value={donationAmount} />
</FormControl>
```

After we've created this input, we'll also need to change the state of the donation amount:

```
$
<Input
  id="component-simple"
  value={donationAmount}
  onChange={(e) => setDonationAmount(e.target.value)}
  placeholder="0.00"
/>
```

By using `setDonationAmount`, we're immediately setting the new donation amount and storing it in the state in just one line of code. Let's navigate back to `localhost:3000` and see what it looks like in [Figure 11-3](#)!

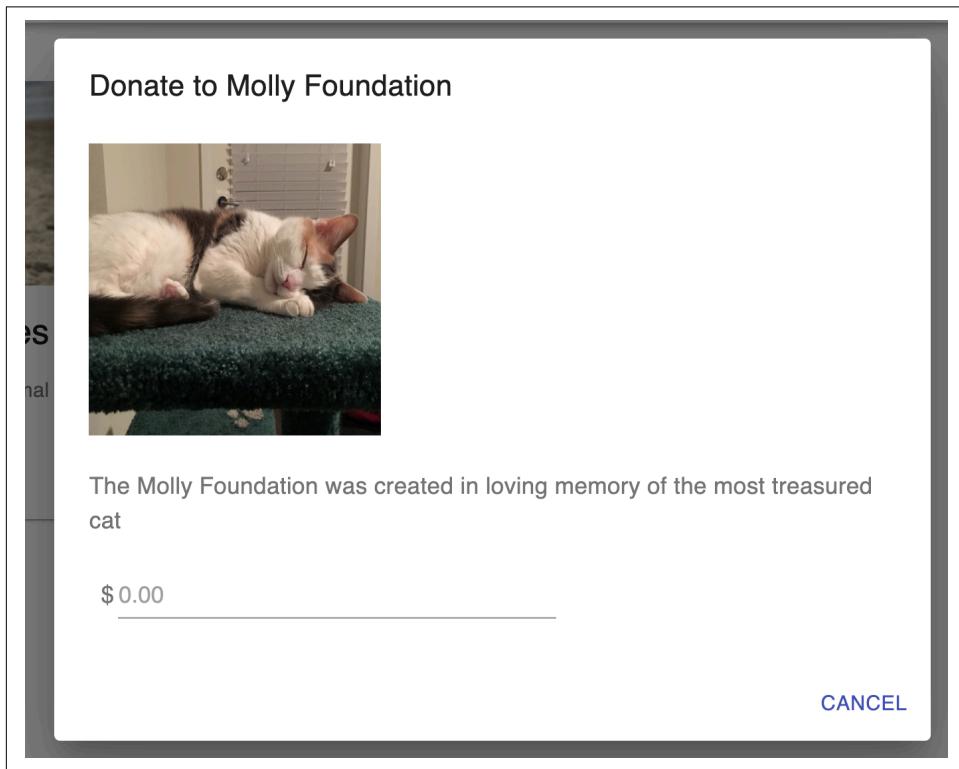


Figure 11-3. Our donation modal form

But right now when a user wants to donate and types something in the input, nothing happens. We need to add in a submit form button for our donation amount so the user can submit the amount they want to donate to a fundraiser. Let's start with importing the button into our modal. Next, import the button from Material UI and the `useStyles` code:

```
import { makeStyles } from '@material-ui/core/styles';
import Button from '@material-ui/core/Button';

const useStyles = makeStyles(theme => ({
  button: {
    margin: theme.spacing(1),
  },
  input: {
    display: 'none',
  },
});
```

```
}));

const DonateModal = () => {
  const classes = useStyles();

  useEffect(() => {
```

Now that we've imported the button and set up the default styles, add the button to our modal right after our donation input:

```
<Button onClick={submitFunds} variant="contained" color="primary">
  Donate
</Button>
```

Now that we have our button rendering, let's write the function to call Web3 and submit our donation to the blockchain:

```
const submitFunds = async () => {
  const fundraisercontract = contract
  const donation = web3.utils.toWei(donationAmount)

  await contract.methods.donate().send({
    from: accounts[0],
    value: donation,
    gas: 650000
  })
  setOpen(false);
}
```

The last thing we'll have to add is displaying the total funds donated to a fundraiser so we can see the donations change in the frontend. In the Card component in *FundraiserCard.js*, add the following code to the Card content:

```
<Typography variant="body2" color="textSecondary" component="p">
  <p>{description}</p>
</Typography>
```

Now that we've finished the code for the donations, let's take a look at the modal. Navigate to the fundraiser we created earlier and open up the donation modal in [Figure 11-4](#).

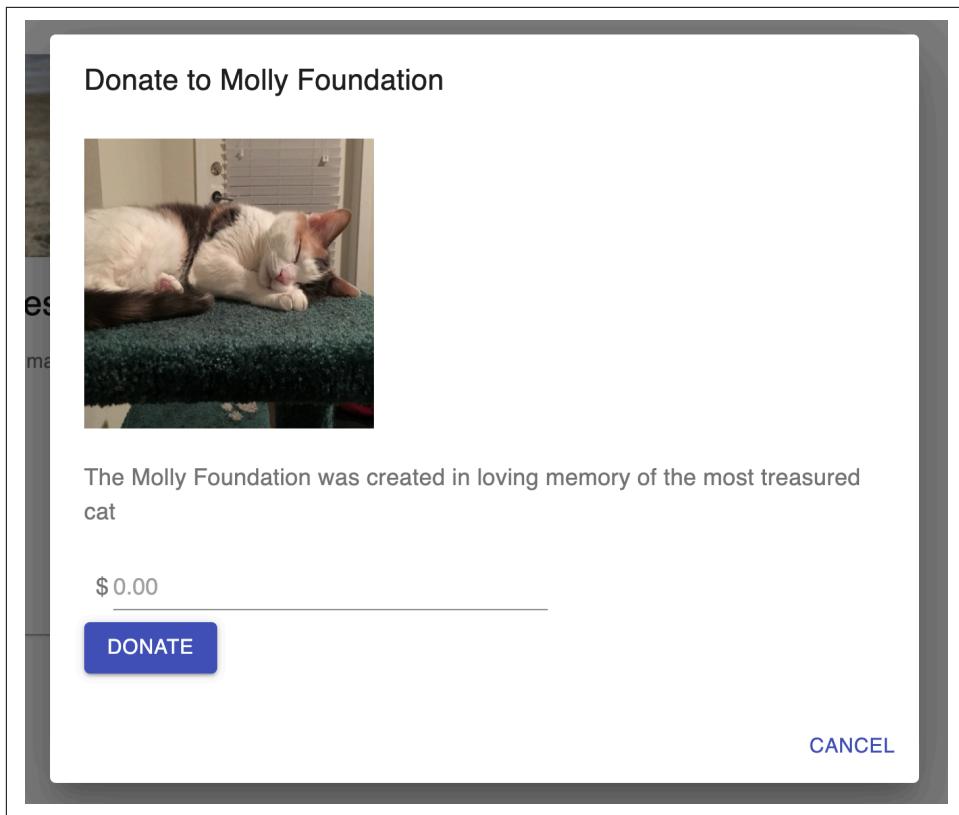


Figure 11-4. Our completed donation modal

Great. Next, let's work on testing our code and see if we can create a donation.

Testing a Donation

Now that you've finished the donation code, let's test out a donation. First, make sure you're on the right MetaMask network. We always want to double-check that we're not sending ether to our local networks. After you've verified you're on the right network, select a donation amount and click submit. MetaMask should open asking you to confirm or reject the transaction.

If everything looks correct, go ahead and confirm the transaction. Once you've confirmed the transaction, you should be able to check your balance and see that the eth has been removed from your test account.

Now if you navigate to that fundraiser again, you'll see the donation amount has increased in that fundraiser, as shown in [Figure 11-5](#).



The Molly Foundation

The Molly Foundation was created
in loving memory of the most
treasured cat

Total Donations:
20000000000000000000

[VIEW MORE](#)

Figure 11-5. The donation increase for our Molly Foundation

The fundraiser should now display a donation amount. But it's not formatted in dollars. Let's next call an outside API to find the current exchange rate for ether so we can display how many dollars our fundraiser has raised.

Displaying the Current ETH Exchange Rate

What if we want to display the current exchange rate of ether so the user can see how much ether they're actually purchasing? Let's work on displaying the current exchange rate of ether in our application. We'll use an add-on to make this easier:

```
npm install --save cryptocompare
```

Next, import the module so we can get the current ETH price. Add the require statement to our *FundraiserCard.js* file:

```
const cc = require('cryptocompare')
```

The add-on will work without an API key, but you can also generate your own at [CryptoCompare](#) and use that:

```
cc.setApiKey('<your-api-key>')
```

In the *Fundraisercard.js* file, add in the `init` function right after we get the data from our Fundraiser contract, set it in a variable, and add in a call to the npm add-on to get the current exchange rate. For this tutorial, we'll see USD, but the add-on supports many other currencies as well.

```
const imageURL = await instance.methods.imageURL().call()
const url = await instance.methods.url().call()

// calculate the exchange rate here
const exchangeRate = await cc.price('ETH', ['USD'])
// pass in the coin you want to check and the currency
const eth = web3.utils.fromWei(totalDonations, 'ether')
const dollarDonationAmount = exchangeRate.USD * eth
```

Next, change the `set` state in React to use the new `dollarDonationAmount` variable we've created:

```
setTotalDonations(dollarDonationAmount)
```

In the Card content, we'll also need to add in our USD symbol:

```
<p>Total Donations: ${totalDonations}</p>
```

Now if you refresh the page, you'll see in [Figure 11-6](#) that we get the dollar amount raised for our current fundraiser.



The Molly Foundation

The Molly Foundation was created in loving memory of the most treasured cat

Total Donations: \$337.14

[VIEW MORE](#)

Figure 11-6. Our donation amount now displayed in a dollar amount

Let's also use this add-on in our donation modal next so our users don't have to calculate how much ETH they're sending and instead can just input a dollar amount.

First, let's update our `useEffect` function and store the current exchange rate in our React state so we can easily reuse it without calling the API in the add-on again:

```
const exchangeRate = await cc.price('ETH', ['USD'])
setExchangeRate(exchangeRate.USD)
const eth = web3.utils.fromWei(totalDonations, 'ether')
const dollarDonationAmount = exchangeRate.USD * eth
```

Remember to create a new state as well in React:

```
const [ exchangeRate, setExchangeRate ] = useState(null)
```

Finally, in our `submitFunds` function, let's use the new add-on to calculate how much ETH a person wants to buy in a dollar amount:

```
const submitFunds = async () => {
  const fundraisercontract = contract
  const ethRate = exchangeRate
  const ethTotal = donationAmount / ethRate
  const donation = web3.utils.toWei(ethTotal.toString())

  await contract.methods.donate().send({
    from: accounts[0],
    value: donation,
    gas: 650000
  })
  setOpen(false);
}
```

Now if you submit a donation in dollar amounts, it will calculate the correct dollar amount in ether in the contract. We can also use this add-on to calculate how much ETH a user is actually buying with the dollar amount they submit. First, create a new variable right after our exchange rate `useState` in React:

```
const [ exchangeRate, setExchangeRate ] = useState(null)
const ethAmount = donationAmount / exchangeRate || 0
```

And right after our donation input, use the variable to display how much the user is purchasing in eth:

```
<div className="donation-input-container">
  <Input
    id="component-simple"
    value={donationAmount}
    onChange={(e) => setDonationAmount(e.target.value)}
    placeholder="0.00"
  />
</FormControl>

<p>Eth Amount: {ethAmount}</p>
```

```
</div>
</DialogContentText>
```

And in the *App.css* file, add in the following CSS:

```
.donation-input-container {
  display: flex;
}
```

Refresh the page and open up a fundraiser. Type in an amount and verify that you see the conversion. Your page should look like [Figure 11-7](#).

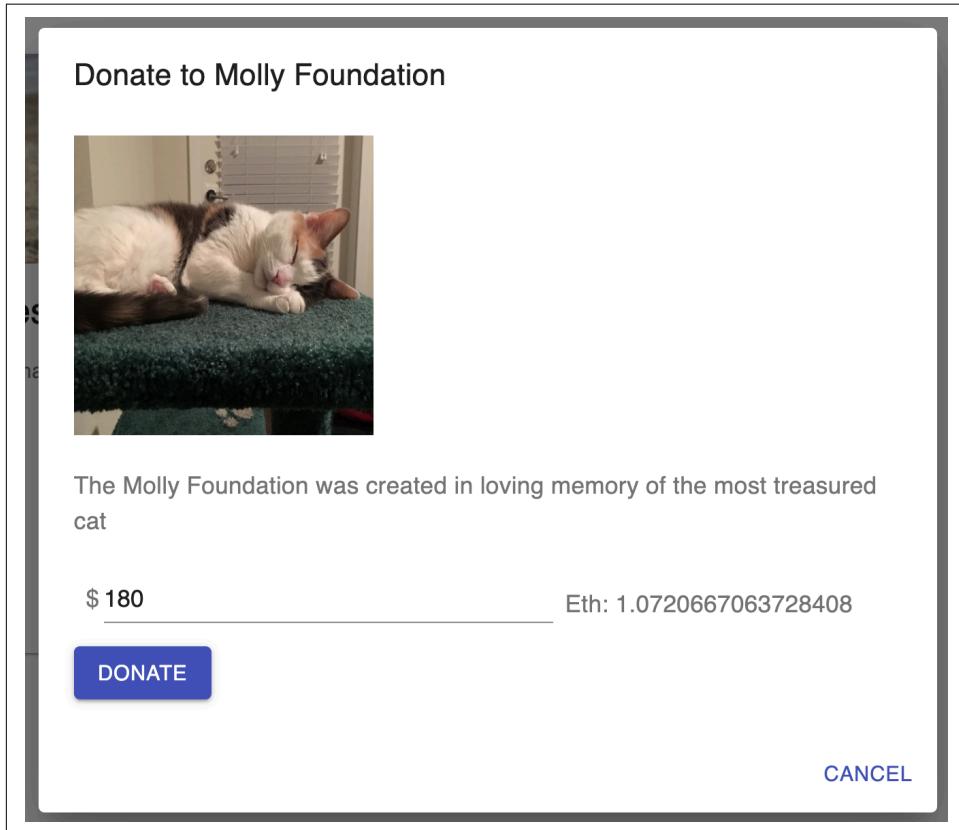


Figure 11-7. What your conversion rate UI should now look like

Next, let's generate a donation receipt for our user.

Generating a Donation Receipt for Our User

If our user wants to prove they've donated to a campaign, we need a way to provide the user with a donation receipt from the fundraiser they've donated to. We'll first need to access our contract again.

In *FundraiserCard.js* in the `useEffect` function, add in the following code after our `setURL` state set:

```
setURL(url)

const userDonations = await instance.methods.myDonations().call({ from: accounts[0] })
console.log(userDonations)
setUserDonations(userDonations)
```

If you navigate back to `localhost:3000`, you'll see that we have an error:

```
./src/FundraiserCard.js
Line 123: 'setUserDonations' is not defined  no-undef
```

Add in `userDonations` to the React state, next to all of our other `useState` statements:

```
const [ userDonations, setUserDonations ] = useState(null)
```

If you check out your console now, you should see the following output:

```
Result {0: Array(1), 1: Array(1), 2: Array(1), values: Array(1),
dates: Array(1), conversionFactors: Array(1)}
0: ["1198178768272226200"]
1: ["1567278668"]
2: ["18460"]
conversionFactors: Array(1)
0: "18460"
dates: Array(1)
0: "1567278668"
length: 1
values: Array(1)
0: "1198178768272226200"
length: 1
```

We'll use the preceding information to then create a receipt for our user. In the *FundraiserCard.js* file, add this to our render function right after our eth amount code:

```
Eth Amount: {ethAmount}
// add in the below code
<div>
  <h3>My donations</h3>
  {renderDonationsList()}
</div>
```

Create an empty function where we'll render all of the users' donations:

```

const renderDonationsList = () => {
  var donations = userDonations
  if (donations === null) {return null}
  // we'll return null so we don't get an error when the
  // donations aren't in the state yet

  return (
    <div>
      </div>
    )
}

```

Now that we have our function, let's iterate through our donations and display individual receipts to a user. Let's get the `donationList`, and create a `for` loop that iterates through each donation and creates a `donationList` array we can use for our render function:

```

const renderDonationsList = () => {
  var donations = userDonations
  if (donations === null) {return null}

  const totalDonations = donations.values.length
  let donationList = []
  var i
  for (i = 0; i < totalDonations; i++) {
    const ethAmount = web3.utils.fromWei(donations.values[i])
    const userDonation = exchangeRate * ethAmount
    const donationDate = donations.dates[i]
    donationList.push({ donationAmount: userDonation.toFixed(2),
      date: donationDate})
  }
}

```

Next, let's add the return so we can display the donation amounts and the request receipt buttons for each donation. Add in the following code right after our `for` loop:

```

return donationList.map((donation) => {
  return (
    <div className="donation-list">
      <p>${donation.donationAmount}</p>
      <Button variant="contained" color="primary">
        Request Receipt
      </Button>
    </div>
  )
})

```

Remember to add some styling so the amount and receipt button display properly. In the `App.css` file, add in the following CSS:

```
.donation-list {
  margin-top: 10px;
  margin-bottom: 10px;
```

```
display: flex;
justify-content: space-between;
}

.donation-receipt-link {
color: inherit;
text-decoration: none;
}
```

If you navigate to `localhost:3000` and select a fundraiser, you should see the new donation list in the dialog, just like in [Figure 11-8](#).

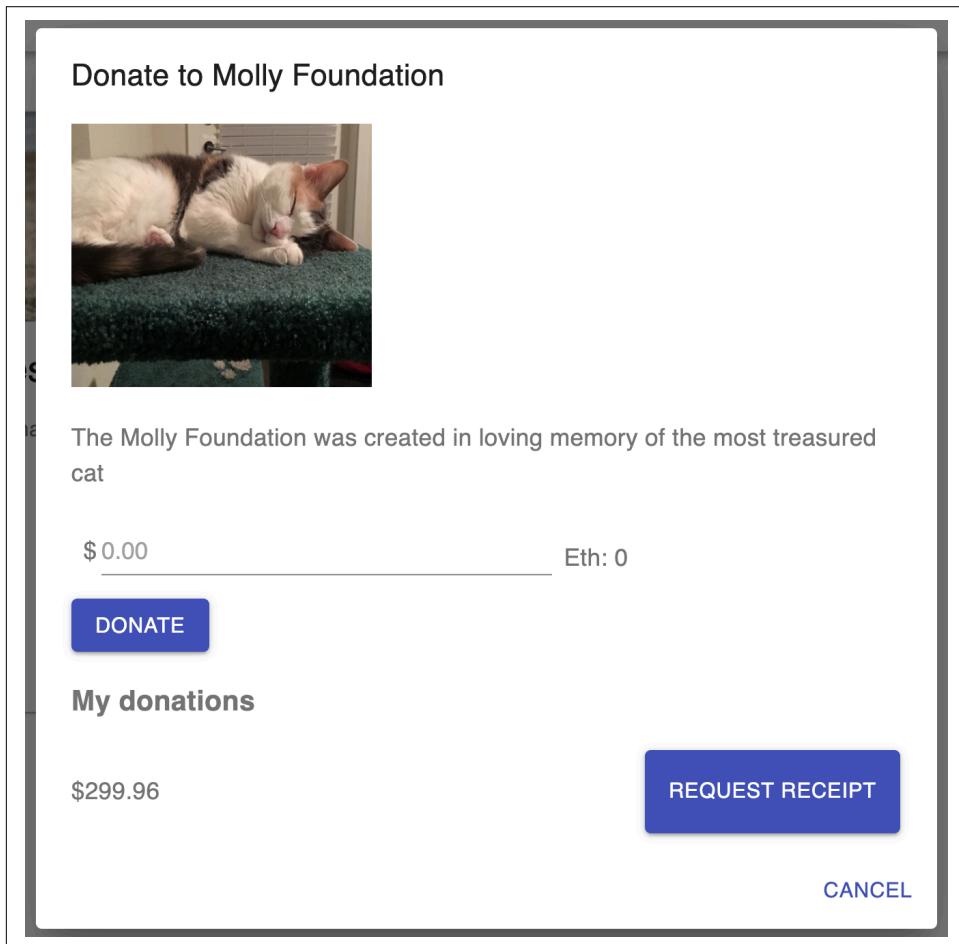


Figure 11-8. Our donation receipt list

Excellent! Now let's finish the functionality to show a receipt. Let's work on the `showReceipt` function.

In our *FundraiserCard.js* file, let's add a new `import` so we can use the link:

```
import { Link } from 'react-router-dom'
```

In the same file, adjust the button in our `renderDonationsList` function so it has a `Link` component for receipts:

```
<Button variant="contained" color="primary">
  <Link
    className="donation-receipt-link"
    to={{ pathname: '/receipts',
      state: { donation: donation.donationAmount, date: donation.date }
    }}>
    Request Receipt
  </Link>
</Button>
```

We'll get an error, so now let's create our `Receipts` component. First, let's start with creating a new component for receipts that we'll display in the navigation:

```
touch Receipts.js
```

Inside the `Receipts.js` file, add in the following code as we have done in the previous sections. This will connect to Web3 for us and allow us to access the donation receipts for an account:

```
import React, { useState, useEffect } from "react";

const Receipts = (props) => {

  useEffect(() => {
    const { donation, date, fund } = props.location.state
    debugger
  }, []);

  return (
    <div>
      <h1>Receipts</h1>
    </div>
  )
}

export default Receipts;
```

Next, we'll want to open up our `App.js` file and add in our new route under the existing routes:

```
<Route path="/receipts" component={Receipts} />
```

And in the same file, make sure you import the `Receipts` component we created so we have access to it in the render:

```
import Receipts from './Receipts'
```

In *Receipts.js*, we'll need to do a few things to access our data and display it properly to the user. First, we'll need to set the state for the three props passed in, which are `donation`, `fundName`, and `date`. We'll also need to format the date and set the state for all of the props so we can use them in our render:

```
const [ donation, setDonation ] = useState(null)
const [ fundName, setFundName ] = useState(null)
const [ date, setDate ] = useState(null)

useEffect(() => {
  const { donation, date, fund } = props.location.state

  const formattedDate = new Date(parseInt(date))

  setDonation(donation)
  setDate(formattedDate.toString())
  setFundName(fund)
}, []);
```

In our *Receipts.js* file, add in the code shown here so we can render our receipt:

```
return (
  <div className="receipt-container">
    <div className="receipt-header">
      <h3>Thank you for your donation to {fundName}</h3>
    </div>

    <div className="receipt-info">
      <div>Date of Donation: {date}</div>
      <div>Donation Value: ${donation}</div>
    </div>
  </div>
)
```

Let's also make it look a little nicer by adding in the following CSS to our *App.css* file:

```
.receipt-header {
  border-bottom: 1px solid gray;
}

.receipt-container {
  text-align: center;
}

.receipt-info {
  display: flex;
  padding: 50px;
  justify-content: space-between;
}
```

Let's navigate back to the fundraisers home page and open up one of our fundraisers in [Figure 11-9](#).

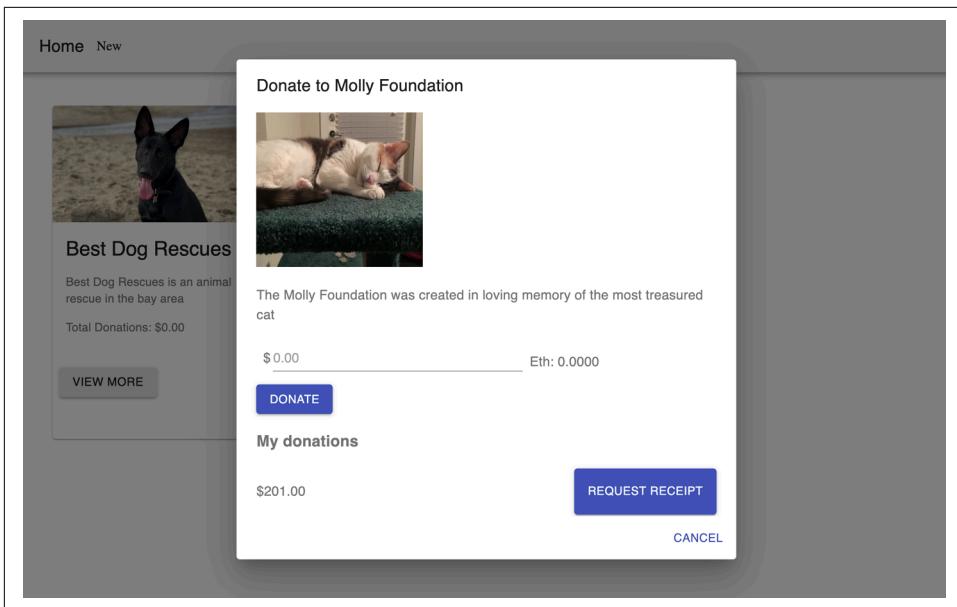


Figure 11-9. Our donation modal with our request receipt button

Click Request Receipt. It should redirect you to our receipts page. See Figure 11-10.



Figure 11-10. Our actual donation receipt

Now your user should be able to make a donation and see a donation receipt. Finally, let's add the ability to withdraw funds from our contract so a fundraiser owner can benefit from the donations.

Handling a Withdrawal from Our Contract

Our first step to be able to withdraw from a contract will be to add a button that only the owner of our contract can see. We'll need to verify that they are the contract owner, and display the withdrawal button we'll add to our frontend if they are. If they are not a contract owner, they'll only see the option to donate to a fundraiser.

In order to check if the current account owns the contract, we'll need to add the following code to our `useEffect` function. We'll access `accounts[0]` to see what current account the user is on, and we'll await an `owner()` call to see who owns the contract. Finally, we'll use an `if` statement to verify that the user is the contract owner:

```
const isUser = accounts[0]
const isOwner = await instance.methods.owner().call()
if (isOwner === accounts[[0]]) {
  setIsOwner(true)
}
```

Make sure to add the `useState` for `isOwner` so that in a moment we can use the conditional in our render function:

```
const [ userDonations, setUserDonations ] = useState(null)
// add in the below code
const [ isOwner, setIsOwner ] = useState(false)
```

Next, let's add a button in the fundraiser modal. Open the `FundraiserCard.js` file and in `DialogActions`, add a button with an `onClick` action for withdrawing the funds. We'll use our state conditional to wrap the button as well so only a contract owner can see the button:

```
<DialogActions>
  <Button onClick={handleClose} color="primary">
    Cancel
  </Button>

  // Add in the code below
  {isOwner &&
    <Button
      variant="contained"
      color="primary"
      onClick={withdrawalFunds}>
    >
      Withdrawal
    </Button>
  }
}
```

```
</DialogActions>
```

It should look like [Figure 11-11](#).

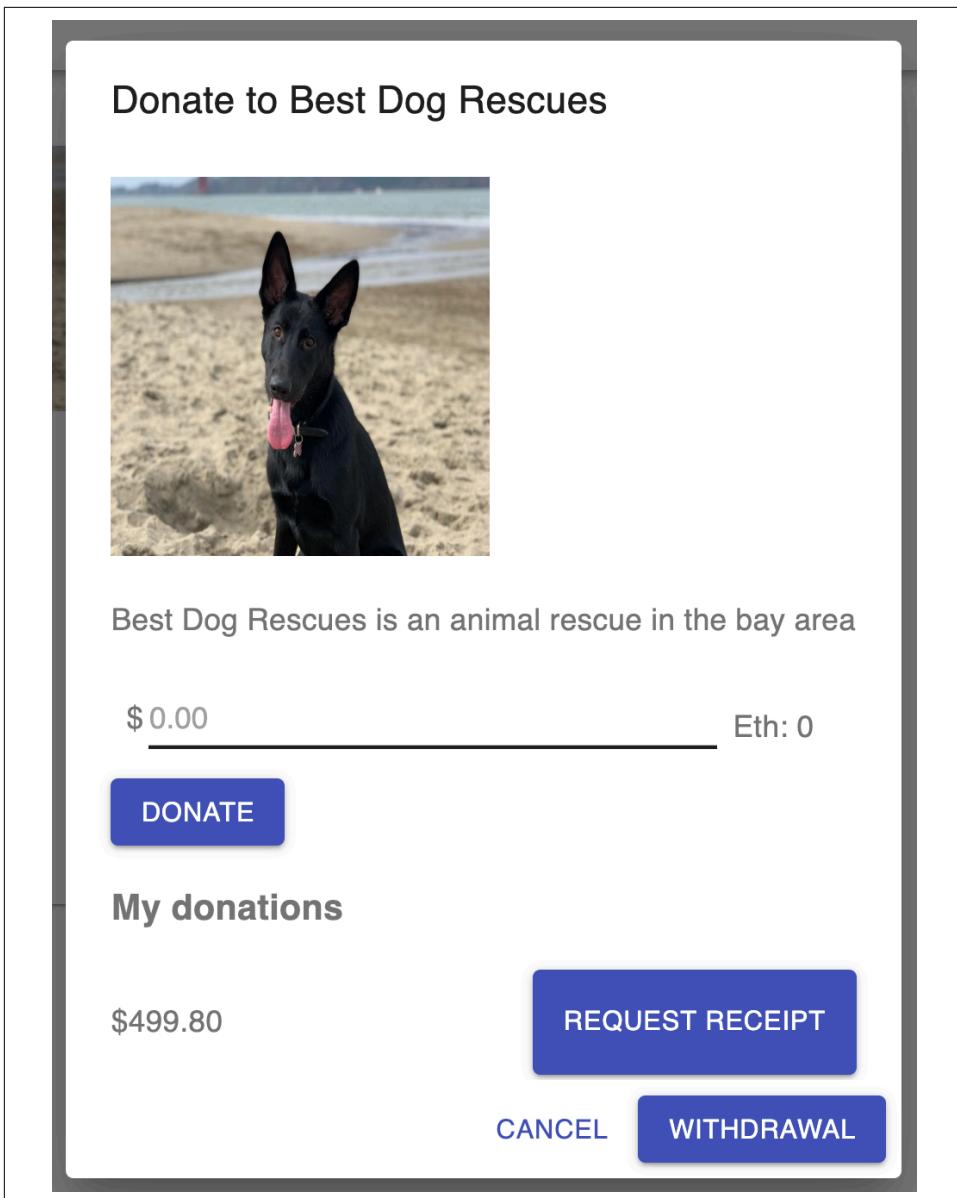


Figure 11-11. Our fundraiser with a withdrawal button

The page will now display an error because we don't have the withdrawal funds function, so let's go ahead and create it:

```
const withdrawalFunds = async () => {
  await contract.methods.withdraw().send({
    from: accounts[0],
  })

  alert('Funds Withdrawn!')
}
```

For now, let's refresh the page if the account changes. Add this in your *Fundraiser-Card.js* file after the `useEffect` function:

```
window.ethereum.on('accountsChanged', function (accounts) {
  window.location.reload()
})
```

Let's navigate back to `localhost:3000` and test it out. Start by selecting a fundraiser and donating some funds to it (see [Figure 11-12](#)).

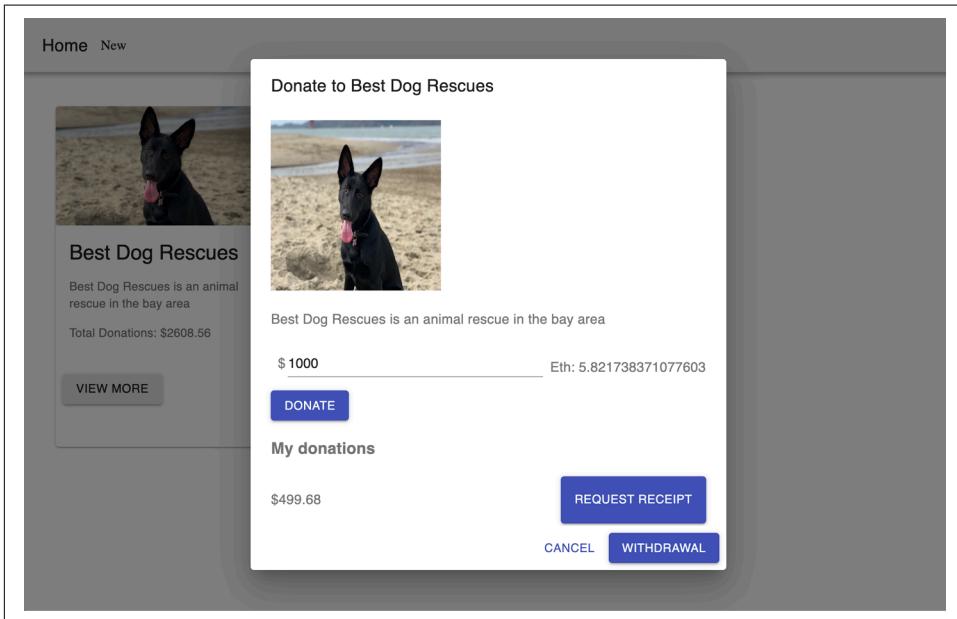


Figure 11-12. Our fundraiser with a new donation

Now if you open up MetaMask, you'll see that your balance has decreased. Refer to [Figure 11-13](#).

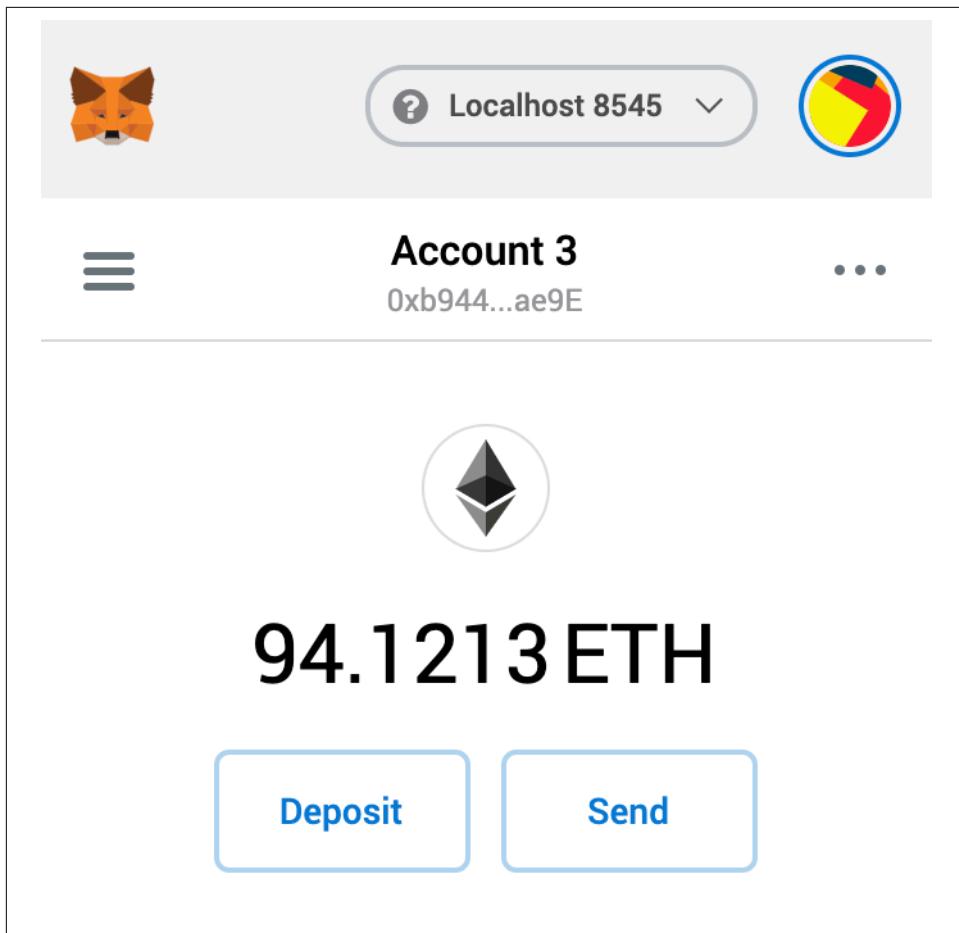


Figure 11-13. MetaMask balance has decreased

You should also see the donation amount increase on the fundraiser (see Figure 11-14).

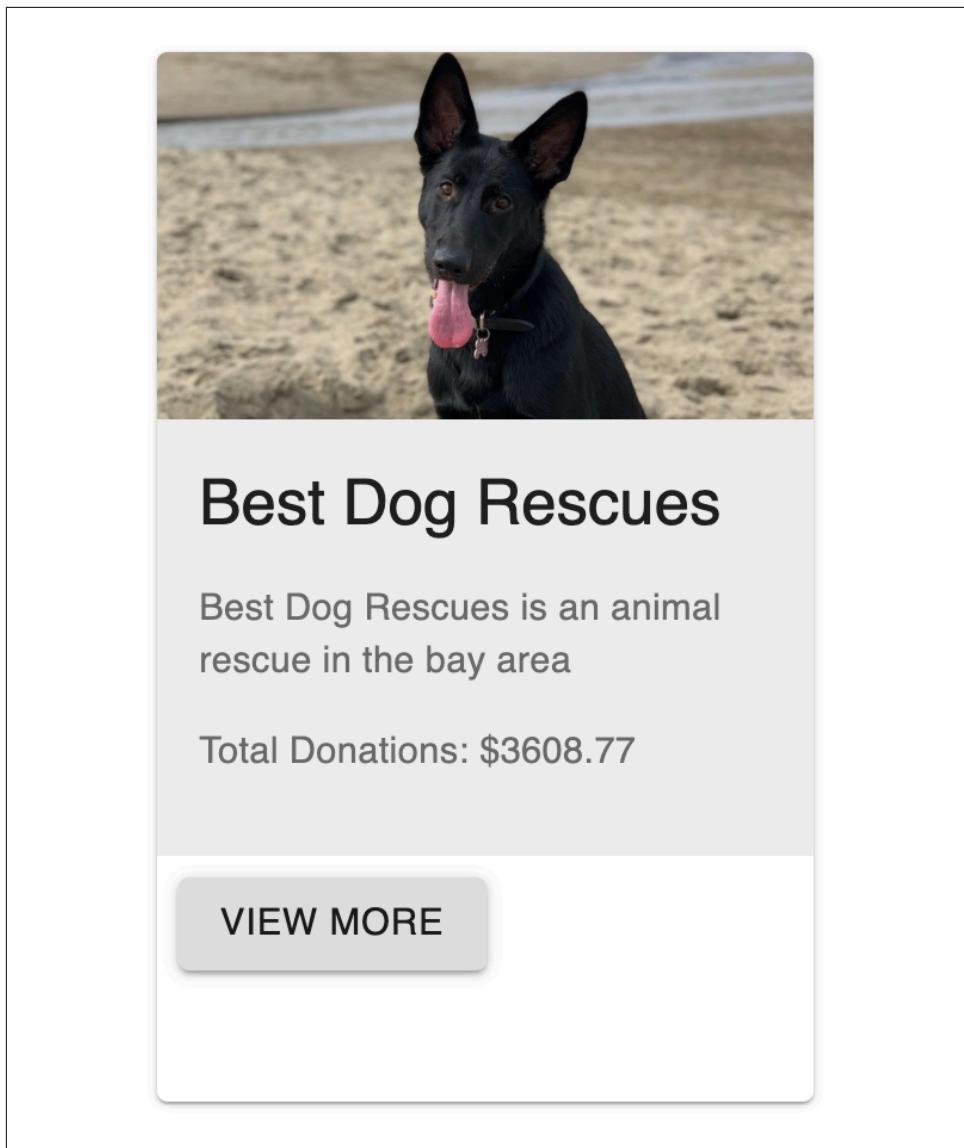


Figure 11-14. Donation amount has increased for our fundraiser

Now, go back to the fundraiser and, using the same address on MetaMask that created the contract, click Withdraw Funds. After, you should see an alert displayed on the page, as shown in [Figure 11-15](#).

And finally, if you open up MetaMask again, you should see your balance has increased, as illustrated in [Figure 11-16](#). You can also see your history of past transactions in MetaMask as well.

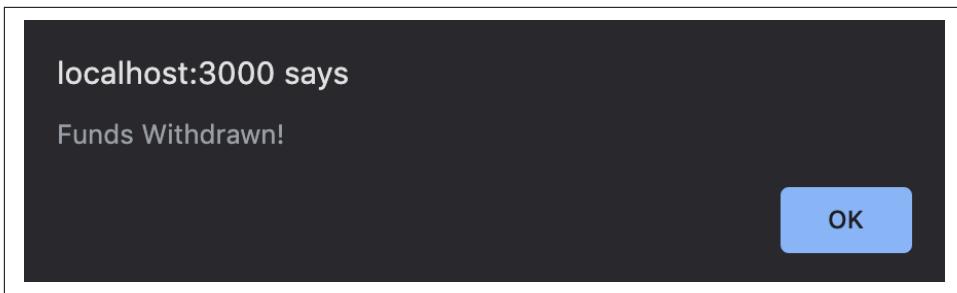


Figure 11-15. Our withdraw funds alert

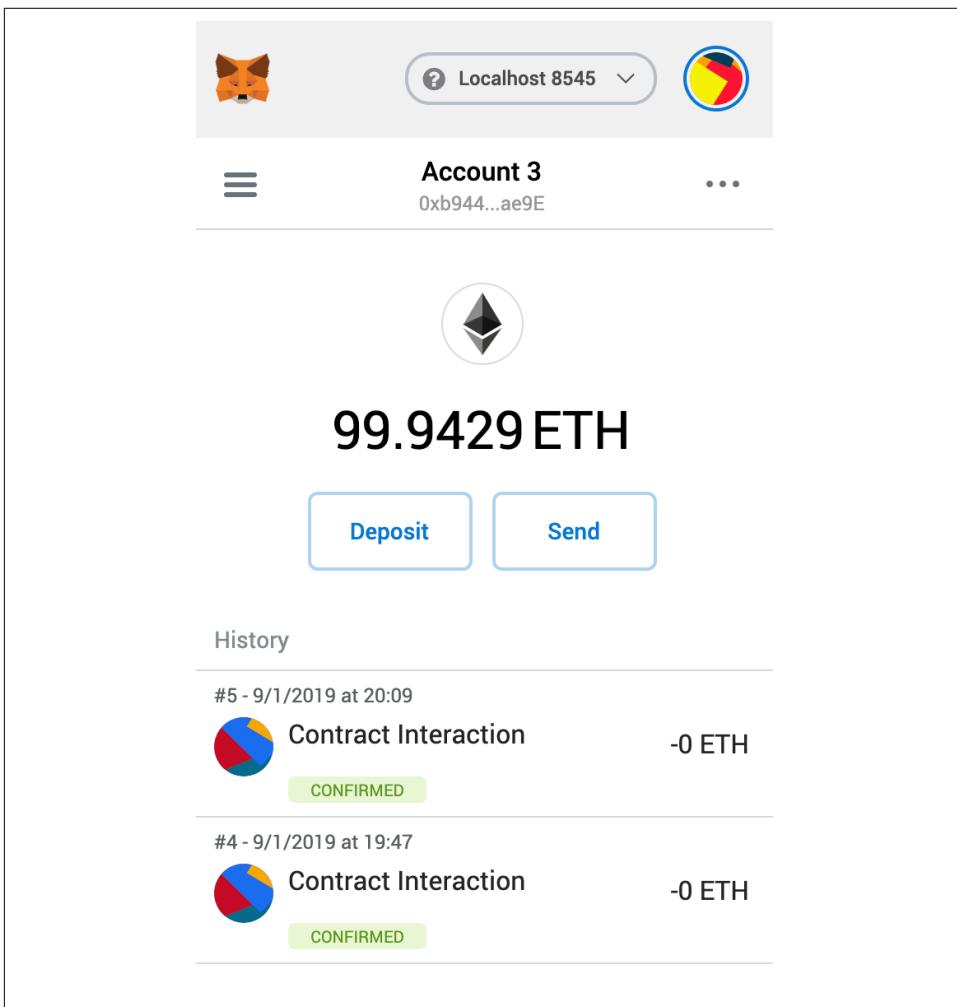


Figure 11-16. MetaMask funds have increased

With the withdrawal functionality complete, your application should look like Figure 11-17.

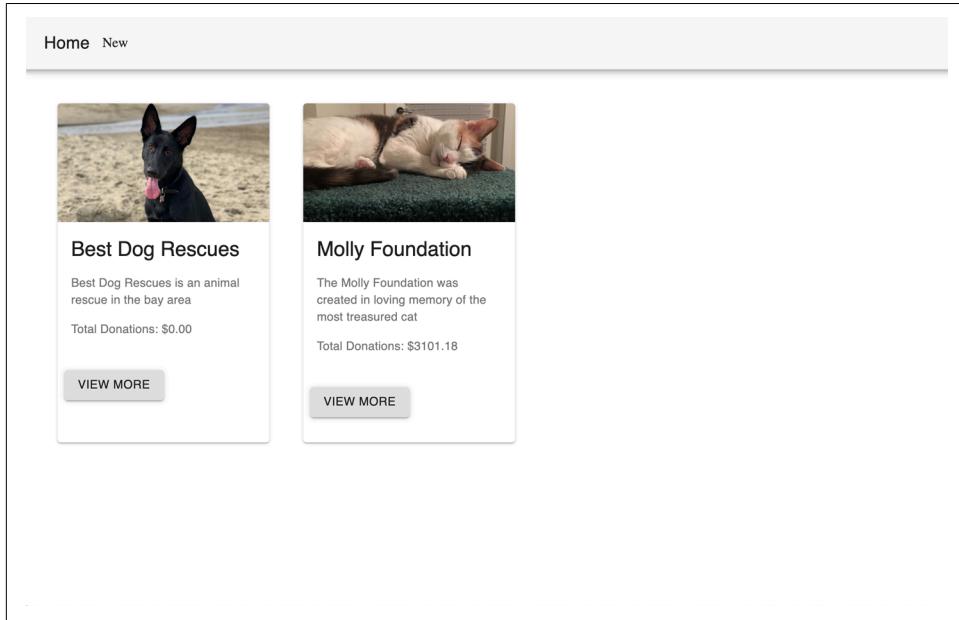


Figure 11-17. Our final application

We do want to add one more use case for our fundraisers: the ability to edit a fundraiser to change the beneficiary.

Adding Edit Beneficiary to the Contract

What if we want to change the beneficiary and make an update to the blockchain? We'll need to call the `setBeneficiary` function and update the address of the user who will be the beneficiary of the contract. Let's start with updating our `FundraiserCard.js` file. Just like in our previous inputs, we'll use `useState` in React in our input:

```
// Add this to our list of React hooks.  
const [ beneficiary, setNewBeneficiary ] = useState(false)
```

Next, right after our donations list is rendered in the UI, let's add a second div with an input and a button to submit the new beneficiary. We'll also need to wrap it in the `isOwner` check as we did before to make sure only an owner can call the function:

```
<div>  
  <h3>My donations</h3>  
  {renderDonationsList()}  
</div>  
  
{isOwner &&
```

```

<div>
  <FormControl className={classes.formControl}>
    Beneficiary:
    <Input
      value={beneficiary}
      onChange={(e) => setNewBeneficiary(e.target.value)}
      placeholder="Set Beneficiary"
    />
  </FormControl>

  <Button variant="contained" style={{ marginTop: 20 }} color="primary"
    onClick={setBeneficiary}>
    Set Beneficiary
  </Button>
</div>
}

```

And finally, let's create the function to call our contract and change the beneficiary:

```

const setBeneficiary = async () => {
  await contract.methods.setBeneficiary(beneficiary).send({
    from: accounts[0],
  })

  alert(`Fundraiser Beneficiary Changed`)
}

```

Let's test it out. Navigate back to the home page and try to change the beneficiary address. Import another test account from `truffle develop` and switch to that account on MetaMask. Next, copy and paste the address to the input we've created. You should see the alert and be able to see the withdraw methods now on that address.

Congratulations! You've finished your first large DApp!

Summary

Great job! Now you have a Web3 application that allows users to create and donate to fundraisers. From here, you can think of an idea for your own Web3 application and use the framework here to get it started.

In this chapter, we learned how to interact with our smart contract by first displaying a dialog to show our application users more information about our fundraiser. We calculated the exchange rate and used our npm add-on in multiple places to make our app easier to use for those who aren't familiar with Ethereum.

We also worked on interacting with our smart contract by creating a set of inputs that our users could submit data to and donate to through Web3. We created a JavaScript function that connected with the blockchain to submit a user donation.

And finally, we created a receipt for our users that will prove they donated to a particular fundraiser. Opening up the card will display each donation a user has made to a particular fundraiser.

In our final chapter, we'll discuss security and common ways that developers can find security issues within their contracts. Once a contract is deployed, we can't edit it like we can a normal web 2.0 application. Security issues are a much bigger concern with a DApp.

PART IV

Securing Your Smart Contracts

In the final part of our book, we will briefly discuss smart contract security and the various free resources that are available online for smart contract security auditing. We will also provide a few suggestions about where you can dig deeper into Ethereum development such as online courses and other books.

Smart Contract Security

Being able to write a smart contract is only one aspect of blockchain development. In Ethereum development, writing secure code that has been thoroughly tested and audited is essential. If your contract is hacked, you can't easily take it down and fix it. Smart contract security is a serious issue and it can be very costly if a mistake is made.

In this chapter, we're going to talk about smart contract security and various techniques we can use to secure our contracts.

Why Do We Need to Worry About Security?

One reason why having a secure smart contract is so important is that you cannot edit your contract once it's on the blockchain. Once it's out there, it's not possible to patch a fix like we can in other types of software development. Ether can become stuck in the blockchain and impossible to release.

As an example, Parity's Multi-Sig contract was hacked on November 6, 2017. A hacker found a small bug in the contract and **disabled it**. The entirety of the \$300 million is lost forever.

In 2016 there was a large hack of the Decentralized Autonomous Organization (DAO), which was created as a decentralized investment fund. A group of unknown hackers found security issues that could be exploited in the contract. The contract should have first removed the ability to call the contract again, but instead it allowed users to recursively call the contract, and the hacker was able to retrieve **3.6 million ether**.

Let's discuss a few ways your smart contract can be hacked.

Types of Smart Contract Vulnerabilities

There are many different ways that your smart contract could be vulnerable to an attack. We'll discuss a few common attacks in this section.

Unprotected Function

As you write your smart contracts, you should always ask yourself what roles should be allowed to invoke the function. If only certain addresses or addresses with specific roles can perform a function, then make sure to use appropriate `require` statements or modifiers to prevent unauthorized access.

In addition to what addresses can perform certain functions, you should consider what context can invoke the function and then use the appropriate visibility modifier. For a refresher, refer to [Chapter 4](#) where we discussed visibility modifiers during the creation of our Greeter contract.

As an example of how to protect your functions, let's examine the `Ownable` contract from OpenZeppelin. The following snippet is for the `transferOwnership` function:

```
function transferOwnership(address newOwner) public onlyOwner {
    _transferOwnership(newOwner);
}
```

Notice the function visibility of `public` and the `onlyOwner` modifier. This means it can be called from anywhere, but this is restricted to the owner of the contract. The function then delegates to `_transferOwnership`, which is shown in our next snippet:

```
function _transferOwnership(address newOwner) internal {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    emit OwnershipTransferred(_owner, newOwner);
    _owner = newOwner;
}
```

The function visibility has been marked `internal`, which means it can be called from within the contract or one of its derivatives. This is a great practice for locking down who can call a function.

Now that we can protect our functions, let's talk about transaction ordering.

Transaction Ordering Dependence

Transaction ordering is a race condition attack where the attacker raises the gas price of a transaction in an attempt to have their transaction executed first. The more you pay in gas, the higher priority your transaction receives from miners. This means someone observing the mempool of pending transactions could jump the line if they see a transaction come in that would be detrimental to their interests.

This attack is very difficult to defend against and commonly targets applications in which timing is important, such as auctions or marketplaces. To mitigate this attack, try batching transactions or separating the details into a separate request.

Integer Overflow and Underflow

Integer overflow and underflow is an issue where a result is produced that is larger than the maximum or smaller than the minimum value of an integer. For instance, if we have a variable with the `uint8` data type, its maximum value is 255. If you were to add 1 to 255, an overflow would occur and the result would be 0. Now that we have 0, if we were to subtract 1 from it, we would underflow and return 255. Instead of allowing overflows and underflows, we recommend using `SafeMath` when performing arithmetic operations.

For an example of how to use `SafeMath`, refer back to [Chapter 6, Example 6-15](#), where we were adding the donate functionality to our Fundraiser contract.

The next vulnerability we will look into is reentrancy.

Reentrancy

A reentrancy attack occurs when an important state variable is changed after a contract calls a function of another contract or invokes the fallback function of another contract. In these cases, the malicious contract is able to call back into the original contract, invoking the same function again before the state has been updated.

To avoid this attack update the state of your contract before calling a function on any contract you do not own or before sending or transferring to any address that may be added by your users. It is also best to avoid using lower-level functions on addresses like `call` when dealing with unknown or untrusted addresses.

Here's an example of a reentrancy. The attacker finds the vulnerability in the contract where two functions rely on the same state. And the hacker is able to call the function again before the balance is set to 0. This was the attack used by hackers in the DAO attack in 2016. The attacker recursively called certain functions to hack the contract, thus exploiting a reentrancy vulnerability in the contract.

Let's take a look at cross-function race conditions, which is another type of reentrancy.

Cross-function race conditions

One interesting attack is when multiple functions share the same state. An example of this is a contract where your balance is reset to 0 only after the transfer occurs, so the hacker could keep calling this even after they've received the withdrawal to hack

more ether from the contract. Basically, they can call another function halfway through the first function's execution to hack your contract.

Here's an example:

```
function withdrawFunds() external {
    uint256 withdrawAmount = balances[msg.sender];
    require(msg.sender.call.value(withdrawAmount)());
    balances[msg.sender] = 0;
}
```

The balance is not set to 0 until after the other function is called. The attacker can call this recursively and successfully hack the smart contract. One way to protect against this attack is to update the balance before executing `call.value` on the sending address:

```
function withdrawFunds() external {
    uint256 withdrawAmount = balances[msg.sender];
    balances[msg.sender] = 0;
    require(msg.sender.call.value(withdrawAmount)());
}
```

It may seem simple, but this reordering of lines could save quite a few smart contracts from being compromised.

Up next, we'll talk about running out of gas.

Block Gas Limit

The Ethereum network, specifically the EVM, can only process a certain computational load at a time. This restriction is conceptualized in the form of the block gas limit, as discussed in [Chapter 1](#). If you surpass the limit, the transaction fails. This can happen when a function processes a significant number of state changes like when iterating over a large array.

To avoid exceeding the gas limit, break large—or potentially large—operations into batches. The pagination function we implemented in [Chapter 7, Example 7-12](#), could be a great example for how to break things up into smaller chunks.

Timestamp Dependence

Because the blockchain is decentralized, it relies on the computational power of distributed nodes (miners) instead of centralized servers to process transactions. The timestamp of the miner can therefore be manipulated by their owners in an attempt to circumvent any logic based on timestamps within a contract.

In order to avoid this vulnerability, make sure your contract does not use timestamps for randomization seeds. It is also a good idea to make sure that your conditional logic can tolerate 15 second variance and still provide the desired behavior. If you are

wondering why 15 seconds are required, it is because both Parity and Geth would reject a block that is more than 15 seconds into the future.

And Many More

These are just a few of the common attacks against smart contracts. It's important to stay updated on the latest types of hacks and keep your contracts secure. ConsenSys provides a [list](#) of common smart contract attacks that you should review when writing your smart contracts.

If you intend to release your smart contract to the public, it's important to have a smart contract audit.

Preparing Your Contract for an External Audit

Before you outsource to an external company for an audit of your smart contract, you should first try to secure it as much as possible and then try to hack it yourself. Let's discuss a few ways we can try to find a vulnerability in a contract.

1. Your contract should have 100% test code coverage. Smart contracts must always have tests. Test-driven development (TDD) is a great option for keeping your contracts safe.
2. Run your tests. Try to break each test and see what info you can glean from that.
3. Check for reordering, replay, or short address attacks.
4. Compile your contract and check for warnings.
5. Verify that your contract has no race conditions, reentrancy issues, or cross-function race conditions.
6. Check for transaction ordering or timestamp dependence.
7. Check for integer overflow and underflow.
8. Check for an unexpected revert.
9. Check for denial of service (DoS) with block gas limit.
10. Check for method execution permissions.
11. Verify that OpenZeppelin and your Solidity versions are up to date.
12. If you use a library in your smart contract, verify that you're using it correctly.
13. Check that the contract uses `SafeMath`.
14. Verify that you don't have any compiler warnings.

CryptoFin provides a great checklist for securing your smart contracts [here](#).

These are only a few examples of ways you can secure your smart contracts. The best thing to do to create a secure smart contract is to find an external auditor. It's not always easy to see potential issues on code you've been working on.

External Auditing

One of the best ways to mitigate security vulnerabilities in your smart contract is to hire an external auditor or a smart contract auditing company. An external auditing company will look at your contract with fresh eyes and run it through different auditing programs to find any security issues. The smart contract auditor will provide you with a list of issues found in your contract and ways you can fix them. Securing smart contracts has become such a large issue that entire companies now exist to audit and provide ways to secure smart contracts.

Auditing Companies

Some of the larger smart contract auditing-focused companies provide public audits of past smart contracts. One example of that is [Authio](#) where you can see examples of vulnerabilities they found and recommendations of how to fix them. If you're interested in smart contract security, many auditing companies make these audits public, and you can read through them to get better at spotting out potential security issues in your own smart contracts.

Solidified

Another way to check if your smart contract is secure is to post it on Solidified. Solidified is a platform where you can post your smart contract and a group of Solidity experts, bounty hunters, and developers will work together to scout out vulnerabilities. If you're interested in becoming an auditor, you can become one at Solidified.io as well. Solidified is great because it first provides a team of Solidity experts to find vulnerabilities in your contract. After they've finished their report, they'll provide suggestions on how to fix the issues in your smart contract. Once you've fixed the issues in the contract, your contract is posted on the Solidified Bounty platform where it is public to the entire Solidified community.

There are also a number of free resources available to help you keep your smart contract secure.

Free Auditing Resources

There are a variety of smart contract auditing resources available online for free. [Echidna](#) is a popular tool from Trail of Bits that provides unit tests and allows you to write tests for your contract. Trail of Bits also provides a long list of [resources](#) for smart contract auditing.

Check out the list of security tools from [ConSensys](#).

You can use these free online resources to start auditing your own contracts internally before you get an external audit. Let's learn about other ways to grow your skills in the next section.

Growing Your Auditing Skills

If you want to become an expert auditor, there are many ways to work on your auditing skills.

Solidified offers a bug bounty program where you are able to interview, and if accepted, you can become a professional auditor at Solidified. You'll be given access to the bounty platform, and be given the opportunity to audit contracts and reap the rewards if you can find a security vulnerability in the contract.

Solidified also offers an [auditing course](#). They've partnered with another company, B9 Academy, to offer a course on auditing. At the time of this writing, Solidified also offers a paid internship to the individuals who have the highest scores upon completion of the course. The course will walk through common security vulnerabilities in smart contracts, how to write an audit, how to start a career as an auditor, how to fix security issues, and more.

Summary

In this chapter, we discussed the different ways in which our smart contracts could be susceptible to vulnerabilities. We learned about a few different types of attacks, such as reentrancy and integer underflow. Securing your smart contract is important because a hacker can exploit your contract and freeze funds or steal them.

Auditing a contract externally is vital to the security of your smart contract. In this chapter, we discussed how you can find an external auditor or use Solidified, a site where you can get your contract audited and posted to a large community of bounty hunters. We also learned how we can audit a smart contract and the various free resources available.

We created this book to help you take your first steps into smart contract development—from building smart contracts and deploying them to connecting your contract to a frontend. This book can be used as a framework to expand on the contracts and code we provide to allow you to write your own DApps.

Best of luck on your smart contract development adventure!

Index

Symbols

`++` (increment) operator, 122
`[]` (square brackets), array operator, 112
`_` (underscore), prefixing state variable names, 60, 96
`_;` line for modifier functions, 68

A

abi field, 72
accounts, 17
 account variable, 64
 checking if current account owns a contract, 224
 creating and funding, 33
 getBalance method, 156
 importing into MetaMask, 77
 viewing MetaMask account, 41
address payable type, 63, 101
address type, 63, 101
addresses
 restrictions on performing functions, 236
 to and from parameters of sendTransaction, 157
Aeron Labs, 28
aircraft logbooks, 28
application binary interface (ABI), 72
apt-get command, 42
arrays
 parameters containing, 101
 push function, 115
 returning from myDonations function, 116-119
artifacts.require function, 51
async/await syntax function, 51

auditing

 free auditing for smart contracts, 240
 growing your auditing skills, 241
 preparing your contract for external audit, 239-240
 auditing companies, 240
 external auditing, 240
 posting your contract on Solidified, 240

Authio, auditing company, 240

await keyword, 57

B

B9 Academy, courses on Ethereum, x
backend and frontend, connection between, 155
backup phrase for MetaMask, 38
beforeEach hook, 94
 moving to parent contract scope, 107
beneficiary address, 101
 editing, 104-109
 transferring Fundraiser balance to, 126-131
beneficiary for a fundraiser, changing, 230-231
Bitcoin white paper, 4
block.timestamp attribute, 20
blockchains, 3-22
 characteristics of, 5-14
 consensus protocols, 6-8
 decentralized networks, 5
 hard forks, 13
 transaction finality, 11
 transaction processing, 9-11
developing on, ix
evaluating for your application, 29-29
 transparency, 29

- verifiable transactions, 29
 frontend, Web3, and, 155
 Ganache, 45
 structure of, 4
 submitting donations in Fundraiser application, 211
 blocks
 and transactions, 18
 block attributes exposed by Solidity, 19
 gaslimit, 238
 getBlockNumber method, 156
 new, adding to blockchains, 6
 reward for adding new block to a block-chain, 8
 bootnodes, 6
 boundary cases, testing for fundraisers, 149
 BrowserRouter, 187
 browsers, MetaMask extension, 33
 build/contracts directory, 72
 Buterin, Vitalik, 4
 bytecode field, 73
- C**
- call method, 160
 - calldata, 58, 101
 - chainId, 22
 - Chrome, installing MetaMask extension, 33
 - clean room feature (Truffle), 51, 56
 - client directory, 92
 - client terminal, 151
 - code examples from this book, xi
 - Coinbase, 87
 - command-line interface, Ethereum clients, 32
 - compile command, 180
 - compiling contracts, 72, 165
 - using solc, 15
 - componentDidMount function, 172
 - confirmation of transactions, 13
 - consensus protocols, 6-8
 - Proof-of-Authority (PoA), 8
 - Proof-of-Stake (PoS), 7
 - Proof-of-Work (PoW), 6
 - ConsenSys
 - course on Ethereum development, x
 - list of common smart contract attacks, 239
 - list of security tools, 241
 - constructors
 - adding string parameters to, 98
 - assigning Fundraiser state variables to, 99
- default, constructor() public {}, 65
 Fundraiser, adding beneficiary and custodian addresses, 102
 writing to make Greeter contract ownable, 62, 65
 contentious hard forks, 14
 contract function, 94, 138
 contractName field, 72
 contracts, smart (see smart contracts)
 conversion rate UI for Fundraiser, 214-217
 createFundraiser function, 195
 cross-function race conditions, 237
 cryptocompare add-on, 214
 cryptocurrencies, 3
- Ethereum, 4
 - CryptoFin, checklist for securing smart contracts, 239
 - cryptography, signing transactions, 21
 - CryptoKitties, 26
 - curl command, using to install Node.js, 42
 - currentProvider method, 158
 - custodian address, 87, 101
- D**
- data locality and function parameters, 100
 - databases, 5
 - blockchains and, 5
 - decentralized applications (DApps), 3, 23-30
 - evaluating blockchain for your application, 29-29
 - larger Fundraiser application, 179-201
 - permanent records, 28
 - supply chain, 26-28
 - tokens, 23-26
 - Decentralized Autonomous Organization (DAO), security breach, 235, 237
 - decentralized networks, blockchains running over, 5
 - deploying contracts, 71-86
 - contract compilation and deployment, 71-73
 - deployment process, 72-73
 - default React Truffle Box contracts to Ethereum test network, 166
 - deploying to Ganache, 74-80
 - deploying to Goerli using Parity, 80-84
 - deploying to Rinkeby with Infura, 84-86
 - deployment becoming expensive, 73
 - FundraiserFactory contract, 136

describe block, 94
for Fundraiser donation tests, 112
development environment, setting up, 31-46
Ethereum clients, 31-33
Greeter contract with React Box, 164
installing MetaMask, 33-41
installing Node.js, 42-43
installing Truffle Suite, 44-45
React Box application in Fundraiser application, 180
development network, 76
difficulty, 7
directories
 directory for Truffle project, initializing, 49
 directory structure for Greeter project, 164
 Truffle directory structure for Fundraiser app, 92

E

Echidna tool, 240
else statement, using Web3 locally, 159
EOAs (see externally owned addresses)
ERC-20 tokens standard, 23
ERC-721 tokens standard, 25
ether, 7, 14-16
 and gas, 16
 translating USD donations to, 110
 units of wei, 113
Ethereum
 contentious hard fork in 2016, 14
 courses offered by B9 Academy, x
 development course at ConsenSys, x
 displaying current exchange rate in Fundraiser UI, 214-217
 fundamentals, 14-22
 accounts, 17
 blocks and transactions, 18
 checking the time, 19
 contracts, 17
 ether and gas, 14-16
 signing transactions, 21
 Mastering Ethereum (Antonopoulos and Wood), ix
 network nodes, 6
 software implementations of protocol, 5
Ethereum clients, 31-33
 client directory for Greeter contract, 74
 installing Parity, 32

Ethereum Improvement Proposal (EIP), token standards ERC-20 and ERC-721, 23
Ethereum white paper, 4
Etherscan
 examples of deployed tokens, 24
 inspecting all blocks and transactions via, 29
events, 18
 for Fundraiser application donations, 124-126
 FundraiserCreated event, 141
 Withdraw event in Fundraiser, 130
eWASM (Ethereum flavored WebAssembly), 8
exchange address, 89
external auditing of smart contracts, 240
external functions, 55
 requiring calldata, 58
externally owned addresses (EOAs), 17
 public key, 21

F

factory pattern, 135
fallback functions, 131-134
faucets on test networks, 15
for loops, 118
 in fundraiser function, offset in, 149
forks
 hard, 13
 temporary, 6
formSubmitHandler function, 176
 getting state for current greeting input, 176
from property, 67
frontend, Web3, and the blockchain, 155
function keyword, 17
functions
 cross-function race conditions, 237
 in Solidity contracts, 17
 multiple returns from, 116
 reentrant, 237
 requiring calldata, 58
 unprotected, 236
 updating state variables, 60
 visibility modifiers for, 55
Fundraiser application, 87-134
 creating the project, 91-93
 creating the UI, 179-201
 React and Material UI, 189-200
 React Box, 179-189
 editing the beneficiary address, 104-109
 fallback functions, 131-134

- finishing the UI, 203-232
accepting donations with React, 208-212
adding detail about each fundraiser, 203-208
displaying current ETH exchange rate, 214-217
generating receipts for donations, 218-224
handling a withdrawal from our contract, 224-231
testing donations, 212
initializing fundraisers, 93-104
setting beneficiary and custodian, 101
making donations, 109-126
defining Donation struct, 111
donation tests, 112-116
events, 124-126
mapping address to dynamic array of Donations, 112
myDonations function, 116-119
tracking total donations, 119-124
overview, 87-91
withdrawing funds, 126-131
- Fundraiser contract, 94
FundraiserCreated event, 141
FundraiserFactory contract, 93, 135-152
creating fundraisers, 137-142
migrating, 135-137
setting up the UI, 151
viewing available fundraisers, 142-151
testing pagination when empty, 142-144
testing the limit, 144-147
testing the offset, 147-151
- fundraisers function
returning an empty collection, 144
testing limit parameter for returns, 144-147
- G**
- Ganache
deploying contracts to, 74
configuring Ganache workspace, 77
download site, 75
installing, 45
- gas, 9, 16
block gas limit, 238
gas and gasprice attributes for transactions, 16
gaslimit, 16, 18
gasprice, 16
- getAccounts method, 156
getBalance method, 156
using a promise, 158
getBlockNumber method, 156
Geth, 5, 32
rejection of blocks more than 15 seconds in the future, 239
Goerli network, 32
deploying contracts to with Parity, 80-84
- Greeter contract
deploying, 53
making dynamic, 56-62
updating with greet function, 55
- Greeter project contracts
adding Greeter contract functionality to React Box application, 173
importing into Truffle, 172-173
- H**
- handleGreetingChange function, 176
hard forks, 13
hashrate, 7
HDWalletProvider, installing, 81
"Hello, World!", testing, 54-56
hierarchical deterministic (HD) wallets, 80
Homebrew, using to install Node.js on Mac, 43
Hyperledger Fabric, 27
- I**
- incrementing currentFundraisersCount, 138
incrementing donors count, 122
Infura, deploying to Rinkeby with, 84-86
Initial Coin Offering (ICO), 24
Integer overflow and underflow, 237
inter-contract communication, 8
internal functions, 55, 236
is syntax, inheritance through, 68
- J**
- JavaScript
installing Node.js for runtime environment, 42
program running on Node.js to report current time, 19
JSON RPC, 32, 155
JSX, 163

K

key/value mappings, 112
Kitty Sales website, 26

L

limit parameter for fundraisers function, testing, 144-147
localhost, setting provider to, 157
longer chain wins, 6

M

Mac computers, installing Node.js, 43
mappings, 112
Mastering Ethereum (Antonopoulos and Wood), ix
Material UI
 form accepting donations in Fundraiser UI, 208-212
 implementing dialog box for Fundraiser UI, 205-208
 navbar from, 186
 using to display details about fundraisers, 203-205
 using with React in Fundraiser application, 189-201
 creating NewFundraiser page view, 191-195
 displaying current fundraisers list, 195-200
maxLimit parameter, fundraisers function, 146
memory keyword, 56
memory, local data notated with, 101
mempool, 9
message (msg) attributes, 18
MetaMask
 installing, 33-41
 network switch, 182
 React Box application setup, 167
 using for Web3 injection, 160
migrate command, 180
migrations
 default React Truffle Box contracts to Ethereum, 166
 deploying to Goerli network with Parity, 82
 deployment to Ganache, 78
 FundraiserFactory contract, 135-137
 testing Migrations contract, 53
 to Rinkeby testnet, 85

Mocha, use in Truffle tests, 51

 information on hooks, 94
modifier, onlyOwner, 67
msg object

 accessing information from, 62
 msg.sender, 65

N

Nakamoto, Satoshi, 4
name function, 95
networks
 configuration for Rinkeby, 85
 Goerli configuration, 82
 Truffle network configuration, 74
new keyword, 118
node -v command, 42
Node.js
 installing, 42-43
 on Mac, 43
 on Ubuntu, 42
nodes
 in context of blockchains, 5
 in decentralized blockchain networks, 5
non-contentious hard forks, 14
non-fungible tokens (ERC-721), 25
nonces (transaction), 10
npm, using to install Truffle, 44

O

objects, internal and private functions not called on, 55
offset, testing for selecting correct fundraisers, 147-151
onChange action, 175
onChange function, 192
 updating in React input form, 176
onlyOwner modifier, 67
onSubmit action, 175
opcodes, 15
 reported by solc, 15
OpenZeppelin contracts
 for ERC-20 tokens, 24
 for ERC-721 tokens, 26
 implementing ownership, 68
 Ownable contract, 180
 SafeMath, 237
ownable contracts, 62-69, 180
 transferOwnership function, 236

P

Parity, 32
deploying contract to Goerli with, 80-84
 launching Parity on Goerli, 82
installing, 32
Multi-Sig contract, security breach, 235
rejection of blocks more than 15 seconds in
 the future, 239
syncing with Goerli network, 32
payable addresses (see address payable type)
payable modifier, 115
permissioned blockchains, 27
pilots' logbook, 28
pragma (compiler instruction), 52
private functions, 55
private keys, 21
promises, 158-159
Proof-of-Authority (PoA) protocol, 8
Proof-of-Stake (PoS) protocol, 7
Proof-of-Work (PoW) protocol, 6
proposers, 7
providers
 about, 158
 creating, 81
 setting, 157
public functions, 55
public-key cryptography, 21
pure functions, 56
 call, 160
pure keyword, 17
push function, 115

R

React, 74
 accepting donations in Fundraiser application, 208-212
 reasons for using, 163
 UI for Fundraiser application, 92
React and Material UI, using in Fundraiser frontend, 189-200
 creating New Fundraiser page view, 191-195
 displaying current fundraisers list, 195-200
React Box, 163
 creating text input for, 175
 in Fundraiser application, 179-189
 compiling, migrating, and deploying contracts, 180
 Fundraiser setup, 182
 importing contracts and migrations, 179

React routing, 185
setting greeting through React, 175
setup, 166-171
 unboxing, 164
react-router-dom package, installing, 185
read-only Solidity functions, 17
receipts for donations, generating, 218-224
records, permanent, 28
reentrancy, 237
 cross-function race conditions, 237
remote procedure calls (RPCs), 155
 Ethereum JSON RPC API, 32
render function, 174
require statement, 18
resilience in blockchains, 29
resolve function, 159
return function, 172
return values
 declaring in Solidity, 116
 identifying for functions, 56
Rinkeby network, deploying to with Infura, 84-86
routing in React application, 185
runExample function, 173

S

SafeMath, 120, 237
security for smart contracts, 235-241
 free auditing resources, 240
 growing your auditing skills, 241
 preparing your contract for external audit, 239-240
 auditing companies, 240
 external auditing, 240
 posting the contract on Solidified, 240
reasons for concern, 235
types of vulnerabilities, 236-239
 block gas limit, 238
 integer overflow and underflow, 237
 reentrancy, 237
 timestamp dependence, 238
 transaction ordering dependence, 236
 unprotected functions, 236
send methods, 63, 160
sendTransaction method, 157
Serenity project, 8
setAccounts function, 200
setGreeting function, 175, 176
setProvider method, 157

sharding, Proof-of-Stake, 8
smart contracts, ix, 3, 4
 changing beneficiary for, 230-231
 creating new instances of a contract from a
 contract, 139
 deploying (see deploying contracts)
 Ethereum address for, 17
 in sharded blockchains, 8
 in Solidity, 17
 security, 235-241
 free auditing resources, 240
 growing your auditing skills, 241
 preparing your contract for external
 audit, 239-240
 reasons to be concerned, 235
 types of vulnerabilities, 236-239
system backed by, user experience challenges, 11
writing your first, 49-69
 first test, 50-54
 making Greeter dynamic, 56-62
 making Greeter ownable, 62-69
 setup, 49
 testing for "Hello, World!", 54-56
solc, compiling contracts, 15
Solidified
 becoming a professional auditor on, 241
 posting your contract on, 240
Solidity, 4
 block attributes, 19
 contracts in, 17
 declaring function return values, 116
 msg attributes, 18
 structs, 111
 transaction (tx) attributes, 18
state variables, 59
 adding to Fundraiser contract and assigning
 to constructor, 99
 beneficiary and custodian, adding to Fund-
 raiser contract, 103
 changing for Greeter contract in React Box
 application, 173
 in Fundraiser contract, 95
 reentrancy attacks through, 237
 setting for current fundraisers list, 196
 setting state for React Box application, 173
 visibility modifiers, 60
storage, persistent data notated with, 101
structs in Solidity, 111
styles
 setting in current fundraisers list display,
 199
 using in donation receipts, 222
 using in submit form button for donations,
 210
submitFunds function, 216
supply chains, 26-28
Szabo, Nick, 4

T

temporary forks, 6
 illustration of, 11
test-driven development (TDD), 49
testing
 Fundraiser application donations, 112-116
 Fundraiser application tests, 93-100
 Fundraiser tests with beneficiary and custo-
 dian, 101
 Fundraiser, setting beneficiary test cases,
 108
 "Hello, World!" contract, 56
 test for donations in Fundraiser application,
 212
 total test coverage for contracts, 239
 writing your first test, 50-54
text field component (Material UI), 191
this, internal and private functions not called
 on, 55
time
 checking in Ethereum, 19
 time attribute, 20
timestamps, 19
 block.timestamp attribute, 20
 manipulation of, 238
tokens, 23-26
 fungible or mutually interchangeable, 23
 non-fungible, 25
toWei helper function, 113
Trail of Bits, resources for smart contract audit-
 ing, 240
transaction (tx) attributes, 18
transactions
 blocks and, 18
 fees for, 8, 16
 finality, 11
 gas and gasprice attributes, 16
 initiation by EOA in Ethereum blockchain,
 17

- ordering dependence, 236
processing in blockchains, 9-11
sendTransaction method, 157
signing, 21
 transaction attributes, 21
transfer methods, 63
transferOwnership function, 236
transparency in blockchains, 29
Truffle, 165-176
 adding Greeter contract functionality, 173
 importing Greeter contracts, 172-173
 React Box setup, 166-171
Truffle Boxes, 92
 React, 163
truffle compile command, 72
truffle develop command, 165, 167
Truffle HD wallet library, 81
truffle migrate --network command, 71
Truffle Suite
 initializing a new project, 49
 installing, 44-45
 Ganache, 45
 test support, 50
truffle test command, 53
truffle unbox react command, 164
truffle-config.js file, 50
tuples, 116
- U**
- Ubuntu
 installing Node.js, 42
 installing Parity, 32
uint256 type, 118
 using SafeMath for, 122
unprotected functions, 236
useEffect function, 194
user experience, challenges in systems backed by smart contracts, 11
user interfaces (UIs)
 connecting the UI to contracts, 163-177
 development environment setup, 164
 React, advantages of, 163
 Truffle, 165-176
 finishing the Fundraiser UI, 203-232
 accepting donations with React, 208-212
 adding detail about each fundraiser, 203-208
 displaying current ETH exchange rate, 214-217
- generating receipts for donations, 218-224
handling a withdrawal from our contract, 224-232
testing donations, 212
Fundraiser donations UI, 109
Greeter contract UI on Ganache, 79
React and Material UI in Fundraiser application, 189-200
setting up, 73-74
setting up client UI for FundraiserFactory on Ganache, 151
using React Box in Fundraiser application, 179-189
using React for Fundraiser application, 92
useState function, 192
useStyles function, 194
- V**
- value (in units of wei), setting to send to contracts, 67
view functions, 56
 call, 160
view keyword, 17
Virtual DOM, 163
visibility modifiers
 for functions, 55
 for state variables, 60
 using to protect functions, 236
- W**
- wallets, 21
 creating on MetaMask, 36
 getting a list of, 156
 hierarchical deterministic (HD) wallet, 80
Walmart's experiment in supply chains, 27
web page for this book, xii
web servers, 5
Web3, 33, 155-161
 accessing and storing in React state, 173
 calling and submitting donations to blockchain, 211
 connecting to and accessing donation receipts, 221
 frontend, Web3, and blockchains, 155
 getting to access Factory contract, 194
 methods for interaction with blockchains, 156-160
 call, 160

getAccounts, 156
getBalance, 156
getBlockNumber, 156
MetaMask, 160
promises, 158-159
providers, 158
send, 160

sendTransaction, 157
toWei helper function, 113
Windows WSL Ubuntu, installing Node.js on,
42
withdrawals from our contract, 224-231
write-only Solidity functions, 17

About the Authors

Kevin Solorio has been developing applications on web and mobile platforms for nearly 10 years. In addition to writing software, he spent nearly four years teaching and writing curriculum for Dev Bootcamp, and is eager to bring his education experience to helping developers learn the fundamentals of blockchain application development.

Randall Kanna started coding at the age of 12 and selling websites online for online currency on neopets.com. Everything came full circle again when she started working in blockchain. Randall has worked as a Senior Software Engineer at companies such as Eventbrite and Pandora.

A self-taught software engineer since 2000, **David H. Hoover** has been intrigued for decades by how people learn to program. Dave cofounded Dev Bootcamp in 2012 and was coauthor of *Apprenticeship Patterns* in 2009. In 2015, Dave fell down the blockchain rabbit hole and soon after landed at ConsenSys where he focused on Solidity and reusability. Dave is the founder and CEO of Red Squirrel Technologies, a consultancy focused on smart contract audits, engineering, and coaching.

Colophon

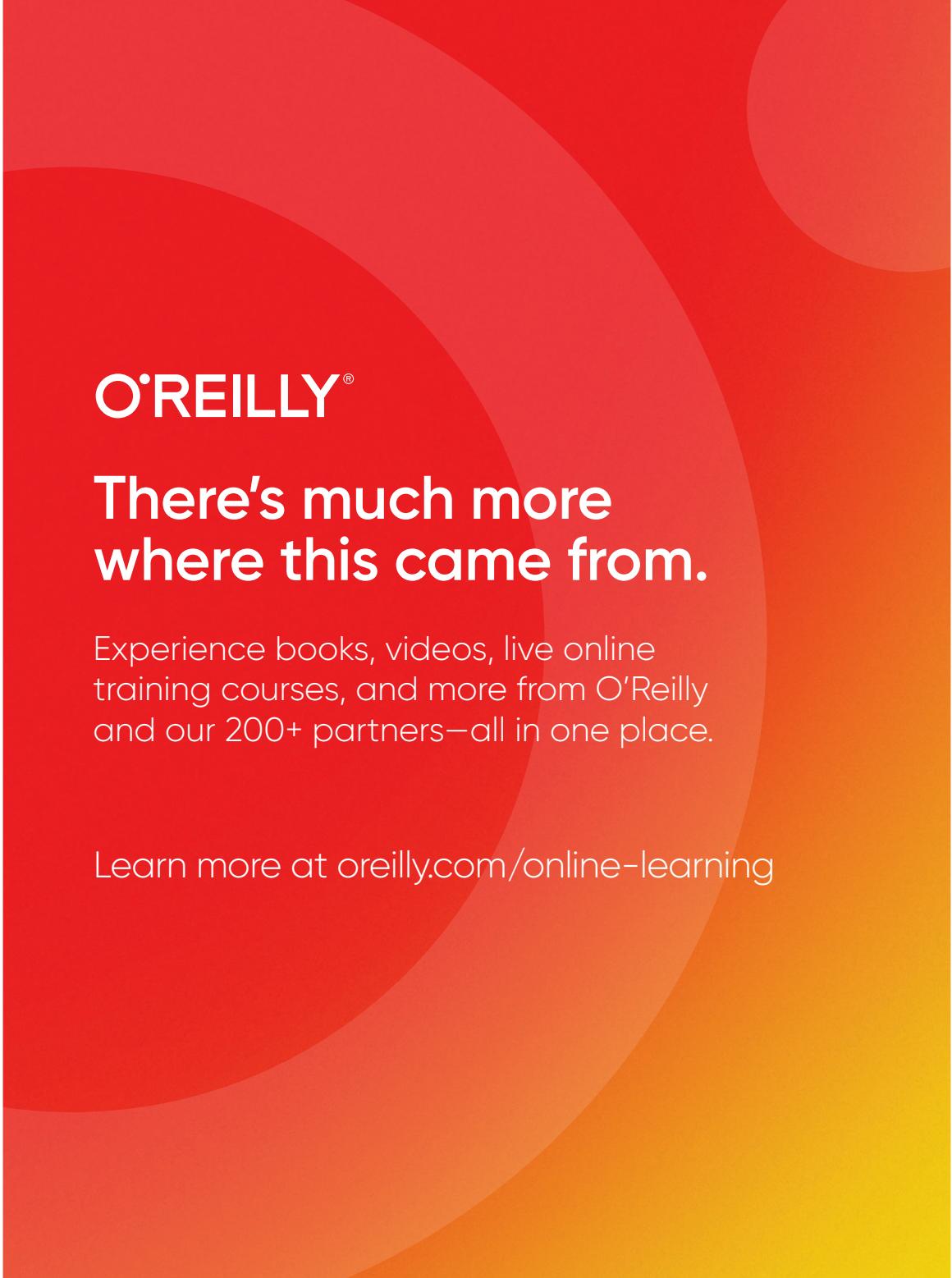
The animal on the cover of *Hands-On Smart Contract Development with Solidity and Ethereum* is a pine marten (*Martes martes*). Similar in appearance to their American cousins (*Martes americana*), the European pine marten is covered in rich brown fur and has a creamy orange patch on its throat. They are about the size of a large domestic cat, so it is no coincidence that in Ireland they are called “Cat crainn.” And just like their feline counterparts, they will twist their bodies when falling in order to land on all fours. During the summer season, they can be heard making similarly shrill mating calls.

The European pine marten is a member of the mustelid family, which includes weasels, badgers, and otters. Of that group, it is the only one with semiretractable claws, which allow it to climb trees. This, together with thick, furry paws and a long bushy tail, help it to climb and traverse the woody forests of northern Europe. It is among these trees where pine martens build dens within natural cavities, although they have been known to take over squirrel dreys and bird nests, as well.

Given its territorial nature, the European pine marten uses feces (called *scat*) to mark its range, and to pin particular spots such as tree stumps and trails. Their scat can be identified by a heart-shaped form and a sweet, musty odor that is reputedly compared to violet-flavored candy. Since 1981, they have been protected in the UK by the Wildlife and Countryside Act.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from *Meyers Kleines Lexicon*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.



O'REILLY®

There's much more where this came from.

Experience books, videos, live online training courses, and more from O'Reilly and our 200+ partners—all in one place.

Learn more at oreilly.com/online-learning