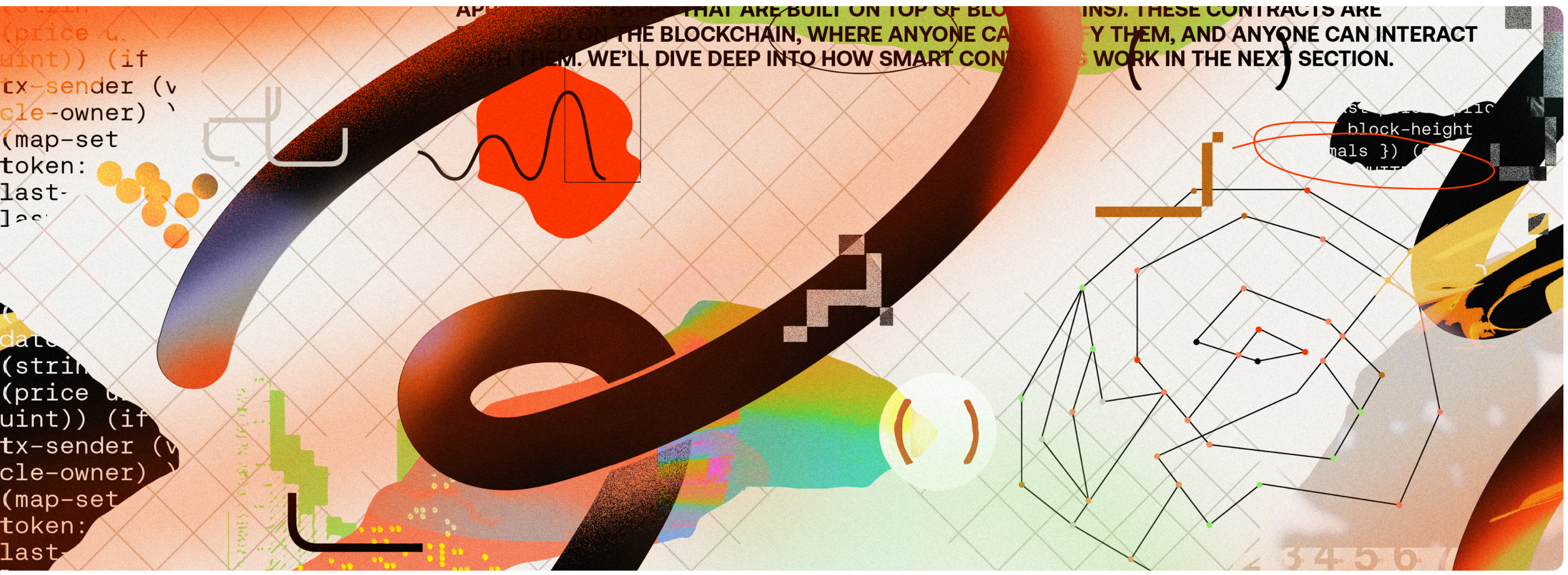
/-/iro : Books

A Guide to Developing Smart Contracts for Web3 Apps



/ Introduction

Web3 comes with the promise of a better internet: an internet designed and built for users. An internet that favors open protocols over walled gardens. An internet without middlemen.

That promise has captured the minds of the investors and the general public alike. Last year, VCs invested $30B into Web3, and hundreds of millions of users owned some form of cryptocurrency. Even in the current bear market, $9.3B has been invested in the first half of 2022: Web3's growth continues.

This concept of an open-source internet—in which users control their own data, access permissionless applications, and engage with each other in a global peer-to-peer network—is enabled by two technological breakthroughs: blockchains and smart contracts.

Blockchains

are decentralized ledgers. These ledgers are public databases that anyone can access, anyone can verify, and anyone can contribute to. There is no central controlling entity, hence decentralization.

Smart Contracts

are the autonomous programs that are the building blocks of decentralized applications (apps that are built on top of blockchains). These contracts are published on the blockchain, where anyone can verify them, and anyone can interact with them. We'll dive deep into how smart contracts work in the next section.

By the end of 2021, these two technologies had caught the eye of nearly 20,000 monthly active developers. The promise of Web3 has attracted some of the brightest minds to build a whole host of new technologies, but building smart contracts takes hard work! There's a lot to learn that developers need to get right. If you want to join the 20,000 developers in Web3, we are here to help you kickstart your own development. In this guide, we'll teach you the basics of developing your own smart contracts.

/ Table of Contents

01 / Smart Contracts 101

What Is a Smart Contract?

Before we dive into the development process of a smart contract, it's worth taking a step back to talk about what exactly a smart contract is. Smart contracts are blockchain programs designed to run autonomously when predefined events or actions occur. What makes them "smart" is that self-execution: these programs do not have administrators or controllers — they are fully autonomous code.

In other words, smart contract's most powerful feature is ensuring trustless transactions. Users don't have to trust a third party, such as a bank, a person, or any other middleman in their transactions. Instead, they trust the code, and the record of those transactions are ultimately stored on a blockchain, the global ledger containing the history of every transaction.

> "What makes them "smart" is that self-execution: these programs do not have administrators or controllers — they are fully autonomous code"

Importantly, not only are the results of those smart contracts on the blockchain, but the smart contracts themselves are too. Since the code of those contracts are on the blockchain, the code can't be tampered with. Indeed, if a developer wanted to publish an updated version of a contract, the original version would still exist on a blockchain, and users could see which version of the contract they were interacting with.

Smart contracts are the building blocks for decentralized apps, and if you want to be a Web3 developer, you will be spending a lot of time developing smart contracts.

The Power of Smart Contract Composability

In software development, composability refers to code's modularity and the ability for developers to take existing code and reuse it, repurpose it, or combine it with other code blocks to create something new. Venture capitalist Chris Dixon describes it aptly when he wrote, “composability is the ability to mix and match software components like Lego bricks.”

Building any application from scratch is a daunting task—writing tens of thousands of lines of code is not easy. The difficulty in building software from scratch is the reason solutions such as WordPress for websites, Shopify for ecommerce, and a long list of other no- and low-code solutions have thrived in Web2. You don’t need to build certain features from scratch and go through a long and expensive R&D process. Instead, you can pay a fee and have an out-of-the-box solution ready to go.

In the world of Web3, we are seeing a renaissance period of composability in the form of smart contracts. Blockchains, by their nature, are open-source public databases. Everyone has access to the same user data. All accounts and all transactions are public, so in terms of building blocks, it’s not just the code that is composable (more on that in a moment), but the data too.

“Blockchains, by their nature, are open-source public databases. Everyone has access to the same user data.”

Developers are able to innovate, build off each other, and have confidence that all the code is referencing the same exact database. No lag due to different databases syncing and no contradictory data sets. This is a critical feature that makes composability so successful in Web3: everything is built on the same foundation. As a result, composability is more effective and can happen more quickly across projects.

Importantly, to encourage that composability, every blockchain network adopts certain standards for how components of the network behave to ensure interoperability between different applications. For example, ERC-20 token standard describes how fungible tokens function on Ethereum, and ARC-0003 specifies the non-fungible token standard on Algorand.

With those standards established, anyone can build an application that involves fungible tokens or non-fungible tokens, and they can trust that any new tokens launched in the ecosystem will be easy to integrate into that application as long as they use the same token standard. It makes the entire ecosystem more composable because now all builders are using the same basic building blocks.

Even better, open-source smart contracts that execute a specific function are perfectly packaged for modularity. Building an app that needs a contract that handles asset swaps? There are contracts for that. Building an app that needs an NFT marketplace inside of it? There are contracts for that. And if you need to adjust them, take the original contract and release a v2 with the additional functionality you need.

This is how SushiSwap was built using Uniswap's code. Instead of building the SushiSwap decentralized exchange from scratch, the SushiSwap team took Uniswap's smart contracts and added a governance token and a liquidity mining program to Uniswap's code. Now, governance tokens and liquidity mining have become core components in DeFi. The whole ecosystem benefited as a result of that example of composability.

Part of what's so exciting about smart contracts is how much of this innovation is happening in the public domain. Code is open and is often subject to review and critique from community members to improve security, functionality, and more, and for developers, they can leverage composability to roll out their own smart contracts faster than they could on their own.

What Are Smart Contract Fees?

When developing smart contracts it's important to keep in mind that there will be fees associated with your smart contract that are paid for by users. Computation isn't free, and someone needs to execute, verify, and store transactions. That's where blockchain miners come in: blockchain miners process smart contract transactions and "mine" new blocks for the network. You need a way to incentivize those miners to provide that computation for the network. If miners are not incentivized, they won't do the work, and the network will stall.

> "Computation isn't free, and someone needs to execute, verify, and store transactions."

This is why blockchains with smart contracts have what's called a "gas asset," and for each blockchain it's different. The Ethereum blockchain has the Ethereum token, the Solana blockchain the Solana token, Stacks the Stacks token, and so on.

Gas assets are necessary for an open-mining network to succeed. The network determines what the gas asset price would be for a smart contract transaction depending on supply and demand for the network's processing capacity, among other factors. For example, during a new app launch or a popular NFT mint, the network can become congested, and some users might be willing to pay higher fees in order to have their transaction verified before others.

Gas fees help to prevent spam transactions from clogging up the network, and they provide miners with a rational reason to participate: they are compensated in the gas token for their computational work. The competitive pricing, in turn, leads to the emergence of a robust fee market, which is a sign of a healthy blockchain network. This is why your users will pay fees when they use your smart contract.

02 / How Do Smart Contracts Fit Into the Web3 Tech Stack?

We've touched on the basics of smart contracts, but how do they fit into the broader Web3 tech stack? A tech stack refers to all of the technology and services used to build an application. A stack generally includes components like development frameworks, programming languages, and technical infrastructure. The six main components of the tech stack for Web3 applications are:

Blockchain	Smart contracts	Development environment
File storage	User identity management	UI/UX components

Blockchain: The Foundation for Web3 Apps

The blockchain is the foundation for Web3 applications, and there are many different options for developers to choose from. Those options have different tradeoffs, ranging from different programming languages, varying levels of security, contrasting approaches to consensus, and more.

In Web2, the equivalent of the blockchain layer would be the operating system, with the most dominant being iOS, Android, macOS, Windows, Linux, and WebOS. The operating system often influences both the design and implementation decisions that developers make when designing Web2 applications, and the same is true with how the blockchain layer influences the applications built on top of it.

> "Different blockchains make different tradeoffs, ranging from different programming languages, varying levels of security, contrasting approaches to consensus, and more."

As a result, the blockchain that serves as the foundation of a Web3 app has important ramifications for the adoption and success of the project. Here are some of the options developers can choose from:

Bitcoin	Bitcoin created the first decentralization currency, one that is durable and highly secure, but at the cost of scalability and programmability.
Stacks	Similar to Bitcoin, Stacks prioritizes security and decentralization—in fact, the Stacks blockchain settles on the Bitcoin blockchain to leverage Bitcoin's security.
Ethereum	Ethereum launched the first smart contract platform and hosts hundreds of apps on a secure and decentralized network, but today that network is congested.
Polygon	Polygon is an L2 blockchain that brings scalability to Ethereum and leverages Ethereum's PoW for security.
Solana	Solana is a blockchain that offers blazingly fast transaction and promises high scalability, but the blockchain is largely funded and controlled by the Solana Foundation.
Polkadot	Polkadot hopes to create a 'blockchain of blockchains,' a framework for a network of interoperable blockchains.
Cosmos	The self-described 'internet of blockchains,' Cosmos hopes to build a network of interoperable blockchains (similar to Polkadot), but makes a number of different design decisions to achieve those goals.

It's important to note that not every blockchain offers smart contract functionality. Perhaps most famously, the Bitcoin blockchain does not offer smart contract functionality because it was designed solely as a decentralized currency. Instead, the Bitcoin layer Stacks brings smart contract functionality to the Bitcoin network.

Tip: having a hard time deciding which ecosystem to build in? Check out our Developer's Guide to Web3 Ecosystems in 2022 here to help you decide what blockchain to build on.

Smart Contracts: The Code for Web3 Apps

Most Web3 developers don't work on the core blockchain itself. Instead, they write smart contracts to build applications on top of the blockchain protocol.

Since Web2 applications are centralized, there is no need for the autonomous, self-executing functionality of smart contracts. Therefore, the closest "equivalent" to the smart contract layer would be the actual code of the applications themselves. There are many different programming languages for writing software code. Developers choose programming languages based on the type of application they intend to build, their technical sophistication, and personal preferences.

"It's worth noting that the choice of programming language is often dependent on the blockchain layer chosen. Most blockchains require smart contracts to be written in a single specific programming language."

The same is true for smart contract programming languages for Web3. However, it's worth noting that the choice of programming language is often dependent on the blockchain layer chosen. Most blockchains require smart contracts to be written in a single specific programming language. We'll talk about different programming languages for smart contracts in a bit (see Smart Contract Programming Languages section).

The Development Environment for Smart Contracts

The development environments in Web3 are not that different from what you may be used to in Web2, and most integrated development environments (IDEs), have built plugins to enable developers to write smart contract code easily.

For example, for building apps on Bitcoin, you would use Clarinet, the development environment for writing smart contracts on the Stacks blockchain for the Bitcoin ecosystem. This tool is designed to ease you into the world of smart contracts without much friction. Clarinet is a local Clarity language runtime packaged as a command-line application for smart contract development, testing, and deployment. Once installed, you can use Clarinet to create a new project, add new contracts to your project, check the syntax of your contracts, measure cost coverage, and make cost optimizations, among many other features.

These development tools offer similar functionality across ecosystems. They often offer syntax assistance, education modules, debugging tools (more on that below), management, deployment, and compilation tools (if the smart contract language requires that the code be compiled into bytecode before the contract can be deployed on the blockchain). Some blockchains even offer multiple IDEs to choose from. Some of the tools you may encounter include:

Blockchain	Development Environment
Stacks	Clarinet, Clarity Tools
Ethereum Polygon	Remix, Truffle, Embark, Hardhat
Solana	Solana Playground, Solana CLI, Anchor
Polkadot	Halva, Substrate, Playground
Cosmos	Agoric, Swingset, Ignite

File Storage Solutions for Web3 Applications

Apps generate data, and that data needs to be stored somewhere. Developers of both Web2 and Web3 applications need file storage, and different storage options can have different prioritizations of scalability, availability, and usability.

> “Apps generate data, and that data needs to be stored somewhere.”

Web3 apps on Bitcoin, Ethereum, and other blockchain networks can use the same decentralized file storage options to securely store their data. With decentralized storage, you can significantly reduce downtime risks and ensure consistent availability because millions of computers serve as nodes to deliver content as needed. Decentralized file storage solutions make your Web3 apps more resistant to censorship because it is difficult to shut down decentralized and distributed servers. Decentralized file storage options include:

IPFS

IPFS is a distributed file system designed to help store and access data across a peer-to-peer network. IPFS empowers developers to store, timestamp, and secure large files without having to put the data itself on-chain.

Arweave

Arweave takes the idea of decentralized file storage further by ensuring the permanence of data. Arweave is building a permaweb, managed by a global community of users and developers who are incentivized to maintain the data storage layer.

Gaia

Gaia is a decentralized storage platform available to developers building apps on the Stacks blockchain. The transactional metadata of the apps is stored on the blockchain itself, while user application data is stored in Gaia to ensure that users enjoy high performance and availability.

Developers can also use centralized storage solutions like Google Cloud or Microsoft Azure (commonly used in Web2 apps) to manage their storage needs. If you choose a centralized storage solution, you can generally expect cheaper costs relative to on-site self-storage and decentralized options. You also get a solution that is optimized for performance. Centralized storage comes with product features like automated server-load balancing and extensive customizations to optimize app performance.

Identity Management in Web3 Applications

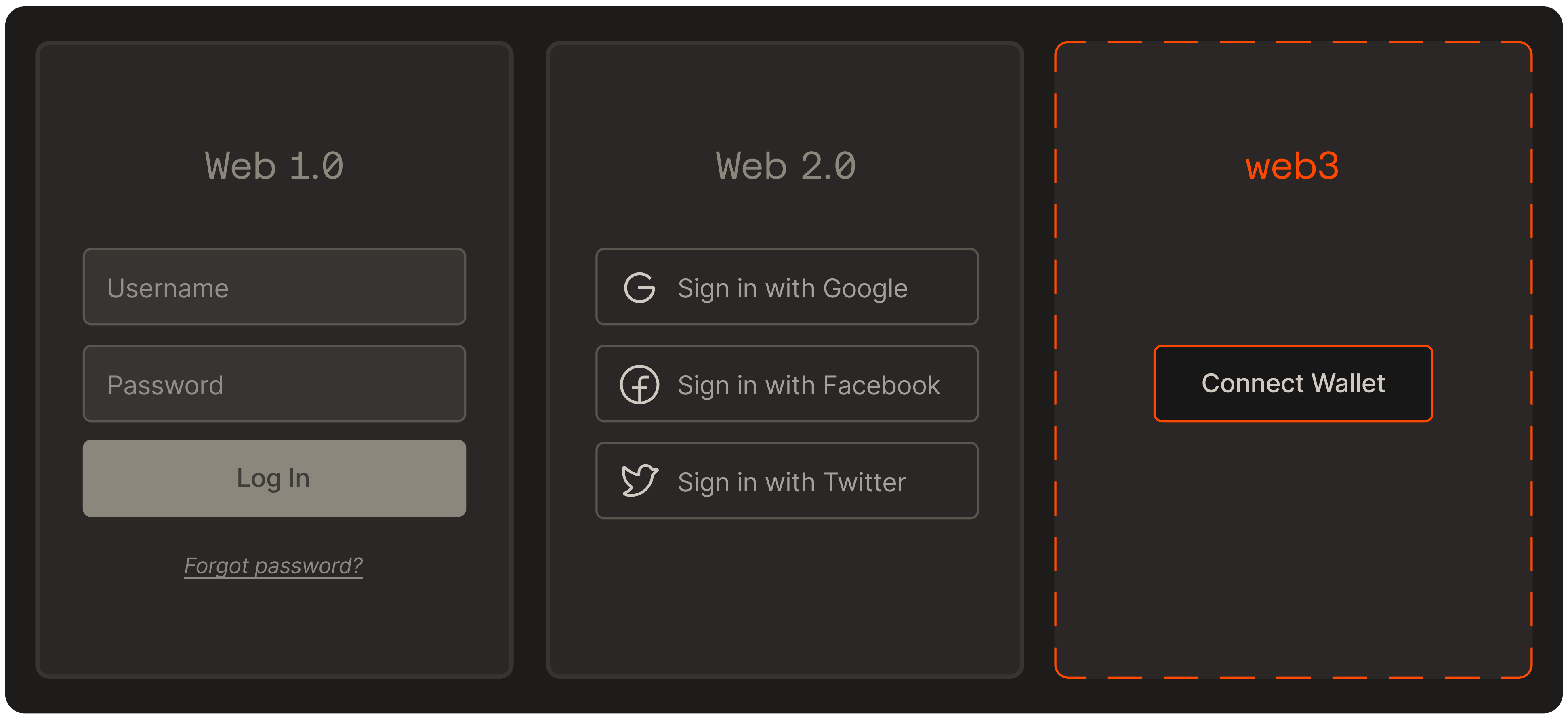
Web3 introduces a new way of managing digital identity for users. Fundamentally, identity management revolves around user authentication and authorization to ensure that the right users have the correct privileges within the app and can access their account. Digital identity is critically important for the security and reliability of any application.

"In Web3 applications, user identity is managed by wallets. A wallet is a unique address on a blockchain that provides a user with an identity and a place to store their digital assets."

In Web1 applications, users needed to provide a unique login and password to identify themselves and access each application they used. Users were encouraged to have different passwords for every website for security. Good luck remembering 50 different passwords!

In Web2 applications, big tech companies, such as Google and Facebook, addressed that pain point and extended the portability of users' identities to other apps. In many Web2 apps, users can leverage their Google or Facebook identities as credentials to sign in to other applications, but that ease comes with risk. If your Google account was hacked, for example, that could mean multiple accounts were compromised.

In Web3 applications, user identity is managed by wallets. A wallet is a unique address on a blockchain that provides a user with an identity and a place to store their digital assets. Wallets serve as a means of establishing users' identities using public-private key cryptography. Public keys are used to encrypt data, and private keys are used to decrypt them. The possession of the correct combination of the public and private keys establish the identity of the user.



Wallets also serve as the primary medium of interaction. They enable users to:

- Buy, hold, and sell digital assets
- Send and receive digital assets from other accounts
- Connect to apps

There are dozens of different wallets in use today, and in general, each blockchain ecosystem will have a number of different wallet providers that users can choose from. Those wallets are available as a combination of browser extensions, mobile applications or desktop applications.

The UI/UX for Web3 Apps

The user interface and experience design define how users interact with applications. In Web3, that user interface is fairly similar to Web2. Developers will often use HTML to structure the app and CSS for styling, regardless of whether the app is built on Bitcoin, Ethereum, or a different blockchain altogether.

Most blockchains have JavaScript libraries with general purpose commands for building applications and managing the user experience. For Bitcoin, you can use the Stacks.js library to access a wealth of resources for managing identity, authentication, storage, transactions, and more on the Stacks blockchain. On the Ethereum network, Web3.js or Ether.js, are general-purpose libraries that you can use to manage the user experience of your Web3 app for access to local or remote Ethereum nodes.

“Most blockchains have JavaScript libraries with general purpose commands for building applications and managing the user experience.”

The tech stack for Web3 applications has familiar origins in the tech stack for Web2 apps, and if you already know how to build Web2 applications, you can easily become a Web3 developer.

03 / Smart Contract Programming Languages

Smart contracts execute autonomously in a public sphere. Writing secure and predictable code is of paramount importance when users have no recourse when things go wrong. Which programming language you use for smart contracts has ramifications for their security, as well as the quality of tooling available to developers and the strength of the developer community. The good news is that developers have a number of different programming languages to choose from, including:

	Solidity	Rust	Clarity
Ecosystem	Ethereum, Polygon, Cosmos, Fantom, Binance Smart Chain, Avalanche	Solana, Polkadot, Cosmos, NEAR, Elrond	Stacks, Algorand
Year created	2014	2010	2020
Decidable	✗ No	✗ No	✓ Yes
Compiled	✓ Yes	✓ Yes	✗ No
Development environment	Remix, Truffle, Hard Hat	VS Code, Eclipse, blockchain-specific tooling (including Anchor, Halva, Agoric Swingset, and more)	VS Code, Clarinet, ClarityTools

Solidity

Solidity, Ethereum's programming language, is the first ever smart contract language. Solidity is an object-oriented programming language with a syntax inspired by JavaScript C++. Those inspirations are apparent in concepts such as loops, overloading functions, and variable declarations, among others. Solidity is specified for the Ethereum Virtual Machine (EVM), and code written in Solidity must be compiled to EVM's bytecode before the smart contract can run. This means that the language is not human-readable once deployed on the blockchain.

Solidity also has a first-mover advantage over other smart contract programming languages. Since it is the oldest smart contract language, developers have lots of learning resources available to them. Solidity offers the most robust developer tooling and has the most deployed smart contracts that developers can reference—in fact, over 44M Solidity smart contracts have been deployed on Ethereum alone (note: the majority of those 44M are clones of a few popular contracts).

Importantly, many other blockchains are EVM-compatible. Learning Solidity provides transferable skills to shorten the learning curve for creating Web3 apps on a number of other blockchains, which may appear to developers who are unsure which ecosystem they want to build in.

Tip: Some smart contract programming languages are compiled, and some are interpreted. For the former, smart contract code must be compiled into bytecode (machine language) before the contract can be deployed on a blockchain. As a benefit, those compiled contracts can often be faster and more efficient to run, but those benefits come at the cost of increased risk: compilers introduce another error vector, and it can be hard to verify a smart contract's code once it is compiled. If you are building with a compiled language, most IDEs offer compilation tools to make that part of the process fast and painless. Solidity and Rust are both compiled languages.

Rust

If you are an experienced developer with general-purpose programming languages, you may prefer to write smart contracts in Rust, which is available on the Solana, Polkadot, and Cosmos blockchains, among others.

Rust is a popular programming language, and it has been voted the “Most Loved Language” six years in a row on the Stack Overflow developer survey. One of Rust’s core strengths is that it is optimized for performance and safety by giving developers the freedom to choose between two modes when writing code. For example, Safe Rust imposes restrictions such as object ownership management on the programmer while Unsafe Rust gives you more autonomy to exploit higher-level abstractions in your code.

Rust’s data structures are compact, and the language offers a number of features that make it easier to write and debug, including type and memory safety. Developers can also take advantage of Rust’s extensive tooling and developer base. If you have experience in Rust, C, or C++, you will likely have a lower learning curve and a faster ramp-up to developing smart contracts on a blockchain that lets you code with Rust.

Clarity

Clarity is a new programming language designed for the realities of smart contracts: the language is transparent, safe, and predictable, critical traits for a decentralized ecosystem that relies on secure code. Unlike Solidity and Rust, Clarity makes two different fundamental decisions: it is not Turing complete, and it is not compiled.

Clarity is decidable. Avoiding turing complexity is a benefit for Clarity. You can still write smart contracts with the same functionality as one written in a Turing complete language, but the code is predictable. In a test environment, you have REPL, print loop, and other features of a decidable language that make it easier to reason with and debug your code. Decidability also helps you to analyze Clarity code for runtime cost and data usage, so you can better predict the fees that will be passed on to your users.

Clarity is an interpreted language. Many popular smart contract programming languages are compiled, meaning the human-readable code is compiled into machine language. This can introduce more bugs and makes it harder to verify whether the smart contract code is the same as the source code. Clarity avoids these issues altogether and prioritizes transparency. Clarity does not use a compiler, and the human-readable source code is made available on the blockchain. This effectively makes the Stacks blockchain a ‘GitHub for smart contracts’ where developers can easily reference the source code for any smart contract on the blockchain.

Clarity is the newest of the 3 smart contracts discussed here, and as a result, its documentation and developer community are not as robust as those you will find in Solidity and Rust. However, the language brings unique fundamental benefits to smart contract development.

04 / Smart Contract Functionality: The Basics

Regardless of what programming language you use, the development of smart contracts shares a lot of universal components. Those components include things like:

Accounts / Principals	Across all ecosystems, there are two “account” types, or entities that can hold token balances: • **Wallets**, that can be controlled by anyone with the private keys • **Contracts**, a smart contract deployed on the network
`tx.origin` `tx-sender`	The principle that sent the original transaction. This is almost always a user.
`msg.sender` `contract-caller`	The account/principal that is calling the current contract. Can be a wallet or a contract. For example, in a simple call chain A → B → C, inside the contract C, msg.sender=B, and tx.origin=A
State variables / Constants	Variables that you want to remain constant throughout a single contract.
Global variables / Keywords	Variables that you want to remain constant throughout a single contract.
`block.number` `block-height`	The current block height of the blockchain.

It’s also important to note that many ecosystem primitives are standardized. Perhaps most notably, fungible tokens and non-fungible tokens have standards in each ecosystem that makes them interoperable. For example, Ethereum has the ERC-20 standard for fungible tokens that defines a set of behaviors that all such tokens should follow (such as being transferable from one account to another, being able to query the total network supply of the token, etc).

> “Many ecosystem primitives are standardized. Perhaps most notably, fungible tokens and non-fungible tokens have standards in each ecosystem that makes them interoperable.”

If a smart contract for a fungible token includes the methods defined in this standard, then any application that is ERC-20 compatible can easily incorporate that new token, making the entire ecosystem more interoperable across applications and projects.

Interestingly, Clarity offers a native trait function that allows smart contracts to define standards that other smart contracts can easily adhere to by invoking the impl-trait function.

If all of that sounds like a lot, it is—learning a new programming language is hard—but here’s the good news. There are a number of great educational resources to learn these programming languages out there, including:

- **Solidity** ↗ docs.soliditylang.org
- **Rust** ↗ doc.rust-lang.org/book
- **Clarity** ↗ clarity-lang.org

As you learn the programming language, each ecosystem also offers helpful documentation to get you up to speed faster too. Some of those resources are linked below:

- **Bitcoin** ↗ developer.bitcoin.org
- **Stacks** ↗ docs.stacks.co
- **Ethereum** ↗ ethereum.org/en/developers/docs
- **Polygon** ↗ wiki.polygon.technology
- **Solana** ↗ docs.solana.com
- **Polkadot** ↗ wiki.polkadot.network/docs/getting-started
- **Cosmos** ↗ docs.cosmos.network

05 / Testing Smart Contracts

The hard thing about software development in Web3 is that once you publish a smart contract to the blockchain, there's no way to change it! Your mistake is there for the world (and every hacker) to see, and that buggy code can severely impact the end user.

According to Elliptic, DeFi users have lost over $12B to theft and fraud, and as much as $5.5B was lost through code exploits. One such code exploit includes the BurgerSwap incident, in which an omission of a single line of code in the smart contract led to a $7.2M hack. In another, a hacker exploited the Proxy Lock Contracts of Poly Network across three different chains — Ethereum, BSC, and Polygon — to steal $611M, in what is the largest single hack in the cryptocurrency industry.

> "The hard thing about software development in Web3 is that once you publish a smart contract to the blockchain, there's no way to change it."

These anecdotes reinforce just how important testing is during smart contract development. The good news is that every Web3 ecosystem comes with some handle tools to test your code.

The Testing Environment for Web3 Apps

Web3 applications don't exist in isolation; they have logic that interacts with the network, including the blockchain's global state, other smart contracts on the blockchain, and other types of data. So, when you develop a smart contract, you not only need to test it in a local environment but also test in one that simulates an actual blockchain environment (called devnet). A devnet refers to an instance of a simulated blockchain that allows a smart contract to interact with live blockchain data (e.g. new blocks being mined, querying a blockchain node, etc), but without the finality and immutability of live blockchain transactions.

> “When you develop a smart contract, you not only need to test it in a local environment but also test in one that simulates an actual blockchain environment.”

When testing your smart contract, you will likely deploy your code on several different environments.

Devnet	Devnet is a local simulated blockchain environment that spins up a blockchain, a blockchain node, a blockchain miner, and a blockchain API—everything a developer needs to test their smart contract in a simulated environment.
Testnet	Testnet is a live blockchain network with real miners and is accessible to everyone. Whereas devnet is the initial testing environment, testnet is where you will stress test your code and monitor network performance and stability.
Mainnet	Mainnet (short for main network) is the live blockchain network. Once you deploy here, your contract is live for users to interact with.

Every blockchain has a testnet environment, and many networks have multiple testnets. These testnets produce new blocks at a similar pace to that on mainnet, and they maintain a network history like mainnet too. Importantly, in order to test your smart contract on testnet, you have to pay gas fees.

The good news for developers is that most testnets offer a “faucet” which provides free gas tokens to developers to facilitate easier testing (note: these tokens are different from the gas tokens used on mainnet and have no real world value). This faucet helps developers test their app or smart contract without spending a lot of money to do so.

Debugging Tools

Many of the IDEs mentioned earlier in this book also offer a number of different debugging tools to help you identify unwanted behavior in your code. Those tools include, but are not limited to:

Trace commands

This command allows you to evaluate any expression, and you'll see a complete trace of all the calls leading up to the problem, complete with arguments, return values, and generated events. With this command, you can trace your way back to exactly where things went sideways.

Inline debuggers

This tool allows you to step line by line through your code and work through the contract. Importantly, this debugger comes with a number of functionalities:

- You can set breakpoints and watchpoints at different lines in your contract. You might set a breakpoint where you think something fishy is happening, and when you run the debugger, the tool will stop the execution when it hits a breakpoint and then you can evaluate the expressions. Similarly, a watchpoint (or data breakpoint) will stop the debugger when a contract variable or data structure will be modified.

- Once you run the debugger, and the execution stops at one of those breakpoints or watchpoints, you can print variables and expressions to see what their values are at that point in the code's execution to investigate the current state of your contract.

- These inline debuggers also allow you to easily navigate through a contract as you step through each expression. For example, you can step over a function call if you don't want to investigate it and continue on to the next expression, or you can step inside and continue investigating inside the called function. This works for calling private functions in the same contract and also for stepping into a contract call across contracts.

As you debug your code, you may receive notifications inside the IDE as you come across certain errors, and those notifications can provide context as to what went wrong—but you shouldn't rely solely on an IDE's notification. They won't catch every bug, but with due diligence and the features described above, you can work your way through your contracts line by line until you find exactly what section of code is misbehaving.

Once you feel like your code is secure and ready to be deployed, there is one additional step you can take.

Hire an Auditor

Another option that many projects turn to is a full code audit—it's beneficial to have an engineer outside the organization review your code with clear eyes, and you also don't want that engineer to then take advantage of any flaws they find in your code. After all, many contracts handle large volumes of money.

In fact, many auditing firms have built a reputation doing just that: reputable code review that protects your assets. Some popular auditing firms include CertiK, OpenZeppelin, Hacken, and Chainsulting.

> "Many auditing firms have built a reputation doing code review for smart contracts while protecting your assets."

These firms work with many of the most well known names in the space and carry out comprehensive reviews of your smart contract code across a number of manual and automated tests. They also provide recommendations on how to address any security risks they find and work with your engineering team to fix any issues their audit uncovers.

06 / Ready to Write Your First Contract?

Smart contracts present a massive opportunity to developers. Today, there are roughly 20,000 monthly active developers in all of Web3, and there is over $1 trillion dollars locked in Web3. That is a hard figure to wrap your head around.

Developing smart contracts has a learning curve—if you've made it to the end of this book, you've learned that much—but smart contracts could transform every single industry as we know them today.

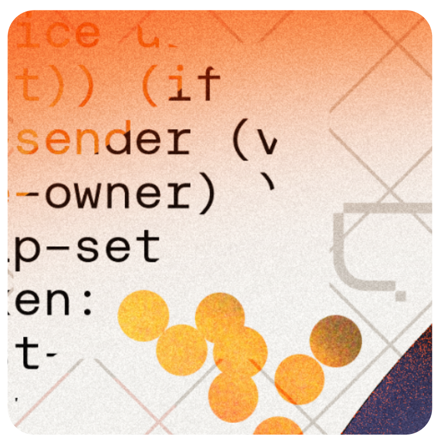
Smart contracts have already completed a speed run of the financial industry in less than a decade. From a simple cryptocurrency to complicated financial instruments, developers have created the first autonomous global market that is liquid 24/7 and accessible to everyone anywhere in the world.

NFTs are transforming the way we think about digital scarcity and digital art, enabling entirely new business models and ways of organizing communities. What industry will be transformed by smart contracts next?

If you're ready to develop your first smart contract, your first step is picking which Web3 ecosystem to build in. Don't worry, we've got you covered there too. We made A Developer's Guide to Web3 Ecosystems in 2022 to help you determine which ecosystem is right for you. In the guide, we take a look at consensus mechanisms, programming languages, blockchain tools, and more to help you identify the right web3 development environment for your needs.

↗ View A Developer's Guide to Web3 Ecosystems in 2022

And if you're ready to learn more about Stacks and would like to speak with someone, Hiro's Developer Advocate Max Efremov also hosts office hours. Book a time to chat with him here.



A Guide to Developing Smart Contracts for Web3 Apps

hiro.so